XML Data Management

P. Atzeni (heavily from Peter Wood)
Chapter 1

Introduction
What is XML?

- The eXtensible Markup Language (XML) defines a generic syntax used to mark up data with simple, human-readable tags
- Has been standardized by the World Wide Web Consortium (W3C) as a format for computer documents
- Is flexible enough to be customized for domains as diverse as:
  - Web sites
  - Electronic data interchange
  - News feeds (RSS, e.g., BBC World News)
  - Vector graphics
  - Mathematical expressions
  - Microsoft Word documents
  - Music libraries (e.g., iTunes)
  - ...
Data in XML documents is represented as strings of text.

This data is surrounded by text markup, in the form of *tags*, that describes the data.

A particular unit of data and markup is called an *element*.

XML specifies the exact syntax of how elements are delimited by tags, what a tag looks like, what names are acceptable, and so on.
Which is Easier to Understand?

TCP/IP
Stevens
Foundations of Databases
Abiteboul
Hull
Vianu
The C Programming Language
Kernighan
Ritchie
Prentice Hall

...
XML vs. HTML

- Markup in an XML document looks similar to that in an HTML document.
- However, there are some crucial differences:
  - XML is a meta-markup language: it doesn’t have a fixed set of tags and elements.
  - To enhance interoperability, people may agree to use only certain tags (as defined in a DTD or an XML Schema — see later).
  - Although XML is flexible in regard to elements that are allowed, it is strict in many other respects (e.g., closing tags are required).
  - Markup in XML only describes a document’s structure; it doesn’t say anything about how to display it.
Very Brief Review of HTML

- A document structure and **hypertext** specification language
- Specified by the **World Wide Web Consortium** (W3C)
- Designed to specify the **logical structure** of information
- Intended for presentation as **Web pages**
- Text is marked up with **tags** defining the document’s logical units, e.g.
  - title
  - headings
  - paragraphs
  - lists
  - ...

- The displayed properties of the logical units are determined by the browser (and stylesheet, if present)
HTML Example

- The following is a (very simple) complete HTML document:

```html
<html>
  <head>
    <title>A Title</title>
  </head>
  <body>
    <h1>A Heading</h1>
  </body>
</html>
```

- When loaded in a browser
  - “A Title” will be displayed in the title bar of the browser
  - “A Heading” will be displayed big and bold as the page contents
These days, most web pages use *XHTML* rather than HTML

- XHTML uses the syntax of XML
- XHTML corresponds to a particular XML *vocabulary* or *document type*
- A document type is specified using a *Document Type Definition* (*DTD*) — see later
- HTML is essentially a less strict form of XHTML
Limitations of (X)HTML

So why not use XHTML rather than XML?

- (X)HTML defines a *fixed set* of elements (XHTML is *one* XML vocabulary)
- elements have *document* structuring semantics
- for presentation to human readers
- organisations want to be able to define their own elements
- applications need to exchange structured *data* too
- applications cannot consume (X)HTML easily
- use XML for *data* exchange and (X)HTML for document representation
XML versus Relational Data

- Why not use data from relational databases for exchange?
- XML is more flexible:
  - XML data is *semi-structured* rather than structured
  - Conformance of the data to a schema is not mandatory
  - XML schemas, if used, allow for more varied structures
- Relational data can always be (and often is) wrapped as XML
Motivating Example

- Say we want to store information about a personal CD library
- The CDs are all of classical music
- Some CDs contain simply solo (e.g., piano) works
- Some CDs have orchestral works (with orchestra, conductor)
- Some CDs contain performances of works by different composers
- We want to avoid repeating information in the descriptions
- We have only 4 CDs (see the next few slides)!
Example (1)

<CD-library>
  <CD number="724356690424">
    ...
  </CD>

  <CD number="419160-2">
    ...
  </CD>

  <CD number="449719-2">
    ...
  </CD>

  <CD number="430702-2">
    ...
  </CD>
</CD-library>
Example (2)

<CD number="724356690424">
  <performance>
    <composer>Frederic Chopin</composer>
    <composition>Waltzes</composition>
    <soloist>Dinu Lipatti</soloist>
    <date>1950</date>
  </performance>
</CD>
Example (3)

<CD number="419160-2">
  <composer>Johannes Brahms</composer>
  <soloist>Emil Gilels</soloist>
  <performance>
    <composition>Piano Concerto No. 2</composition>
    <orchestra>Berlin Philharmonic</orchestra>
    <conductor>Eugen Jochum</conductor>
    <date>1972</date>
  </performance>
  <performance>
    <composition>Fantasias Op. 116</composition>
    <date>1976</date>
  </performance>
</CD>
Example (4)

<CD number="449719-2">
  <soloist>Martha Argerich</soloist>
  <orchestra>London Symphony Orchestra</orchestra>
  <conductor>Claudio Abbado</conductor>
  <date>1968</date>
  <performance>
    <composer>Frederic Chopin</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
  <performance>
    <composer>Franz Liszt</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
</CD>
Example (5)

<CD number="430702-2">
  <composer>Antonin Dvorak</composer>
  <performance>
    <composition>Symphony No. 9</composition>
    <orchestra>Vienna Philharmonic</orchestra>
    <conductor>Kirill Kondrashin</conductor>
    <date>1980</date>
  </performance>
  <performance>
    <composition>American Suite</composition>
    <orchestra>Royal Philharmonic</orchestra>
    <conductor>Antal Dorati</conductor>
    <date>1984</date>
  </performance>
</CD>
Future of XML

- XML offers the possibility of truly cross-platform, long-term data formats:
  - Much of the data from the original moon landings is now effectively lost
  - Even reading an older Word file might already be problematic
- XML is a very simple, well-documented data format
- Any tool that can read text files can read an XML document
- XML may be the most portable and flexible document format since the ASCII text file
Overview

- In these lectures we are going to look at
  - some basic notions of XML
  - how to define XML vocabularies (DTDs, XML schemas)
  - how to query XML documents (XPath, XQuery)
  - how to process these queries (very little, indeed)
Literature

- H. Katz (editor). *XQuery from the Experts*. Addison Wesley, 2004
- These slides . . .
Chapter 2

XML Fundamentals
Elements, Tags, and Data

- A very simple fragment of an XML document:

```xml
<person>
    Alan Turing
</person>
```

- Composed of a single *element* whose name is `person`
- Element is delimited by the *start tag* `<person>` and the *end tag* `</person>`
- Everything between the start tag and end tag (exclusive) is the element’s *content*
Content of the above element is the text string *Alan Turing*

Whitespace is part of the content (although many applications choose to ignore it)

<person> and </person> are *markup*,

The string *Alan Turing* and surrounding whitespace are *character data*
Special syntax for *empty elements*, elements without content

- Each can be represented by a *single* tag that begins with `<` but ends with `/>`
- e.g., `<person/>` instead of `<person></person>`

XML is case sensitive, i.e. `<Person>` is not the same as `<PERSON>` (or `<person>`)
XML Documents and Trees

XML documents can be represented as trees

```xml
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>
    computer scientist
  </profession>
  <profession>
    mathematician
  </profession>
</person>
```
XML Documents and Trees

XML documents can be represented as trees

```xml
<person>
    <name>
        <first_name>Alan</first_name>
        <last_name>Turing</last_name>
    </name>
    <profession>
        computer scientist
    </profession>
    <profession>
        mathematician
    </profession>
</person>
```
XML Documents and Trees (2)

- The `person` element contains three `child` elements: one `name` and two `profession` elements.
- The `person` element is called the `parent` element of these three elements.
- An element can have an arbitrary number of child elements and the elements may be nested arbitrarily deeply.
- Children of the same parent are called `siblings`.
- Overlapping tags are prohibited, so the following is not possible:

```
<strong>
  <em>
    example from HTML
  </em>
</strong>
```
Every XML document has one element without a parent
This element is called the document’s root element (sometimes called document element)
The root element contains all other elements of a document
Attributes

- XML elements can have *attributes*
- An attribute is name-value pair attached to an element’s start tag
- Names are separated from values by an equals sign
- Values are enclosed in single or double quotation marks
- An element cannot have two attributes with the same name
- Example:

  `<person born='1912/06/23' died='1954/06/07'>
    Alan Turing
  </person>`

- The order in which attributes appear is not significant
Attributes (2)

- We could model the contents of the original document as attributes:

  `<person>
    <name first='Alan' last='Turing'/>
    <profession value='computer scientist'/>
    <profession value='mathematician'/>
  </person>`

- This raises the question of when to use child elements and when to use attributes

- There is no simple answer
Attributes vs. Child Elements

- Some people argue that attributes should be used for metadata (about the element) and elements for the information itself
  - It’s not always easy to distinguish between the two
- Attributes are limited in structure (their value is simply a string)
- There can also be no more than one attribute with a given name
- Consequently, an element-based structure is more flexible and extensible
Entities and Entity References

- Character data inside an element may not contain, e.g., a raw unescaped opening angle bracket `<
- If this character is needed in the text, it has to be escaped by using the `< entity reference`
- `<` is the name of the entity; `&` and `;` delimit the reference
- XML predefines five entities:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lt</code></td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td><code>amp</code></td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td><code>gt</code></td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td><code>quot</code></td>
<td><code>&quot;</code></td>
</tr>
<tr>
<td><code>apos</code></td>
<td><code>'</code></td>
</tr>
</tbody>
</table>

- We will cover entities in more detail when discussing DTDs later
CDATA Sections

- When an XML document includes samples of XML or HTML source code, all `<`, `>`, and `&` characters must be encoded using entity references.
- This replacement can become quite tedious.
- To facilitate the process, literal code can be enclosed in a **CDATA section**.
- Everything between `<![CDATA[` and `]]>` is treated as raw character data.
- The only thing that cannot appear in a CDATA section is the end delimiter `]]>`.
Comments

- XML documents can also be commented
- Similar to HTML comments, they begin with <!-- and end with -->
- Comments may appear
  - anywhere in character data
  - before or after the root element
  - However, NOT inside a tag or another comment
- XML parsers may or may not pass along information found in comments
In HTML, comments are sometimes abused to support nonstandard extensions (e.g., server-side includes)

Unfortunately,

- comments may not survive being passed through several different HTML editors and/or processors
- innocent comments may end up as input to an application

XML uses a special construct to pass information on to applications: a *processing instruction*

It begins with `<?` and ends with `?>`

Immediately following the `<?` is the target (possibly the name of the application)
Processing Instructions (2)

Examples:

- Associating a stylesheet with an XML document:
  ```xml
  <?xml-stylesheet type="text/xsl" href="style.xsl"?>
  ```

- Embedded PHP in (X)HTML:
  ```php
  <?php
      mysql_connect("database...",
                "user",
                "password");
  ...
      mysql_close();
  ?>
  ```
XML Declaration

- The *XML declaration* looks like a processing instruction, but only gives some information about the document:

  ```xml
  <?xml version='1.0'
    encoding='US-ASCII'
    standalone='yes'?>
  ```

- **version**: at the moment 1.0 and 1.1 are available (we focus on 1.0)
- **encoding**: defines the character set used (e.g. ASCII, Latin-1, Unicode UTF-8)
- **standalone**: determines if some other file (e.g. DTD) has to be read to determine proper values for parts of the document
Well-Formedness

A *well-formed* document observes the syntax rules of XML:

- Every start tag must have a matching end tag
- Elements may not overlap
- There must be exactly one root element
- Attribute values must be quoted
- An element may not have two attributes with the same name
- Comments and processing instructions may not appear inside tags
- No unescaped `<` or `&` signs may occur in character data
Well-Formedness (2)

- XML names must be formed in certain ways:
  - May contain standard letters and digits 0 through 9
  - May include the punctuation characters underscore (_), hyphen (-), and period (.)
  - May only start with letters or the underscore character
  - There is no limit to the length

- The above list is not exhaustive; for a complete list look at the W3C specification

- A parser encountering a non-well-formed document will stop its parsing with an error message
MathML is an XML vocabulary for mathematical expressions

SVG (Scalable Vector Graphics) is an XML vocabulary for diagrams

say we want to use XHTML, MathML and SVG in a single XML document

how does a browser know which element is from which vocabulary?

e.g., both SVG and MathML define a set element

we shouldn’t have to worry about potential name clashes

we should be able to specify different namespaces, one for each of XHTML, MathML and SVG
The namespaces solution

- The solution is to *qualify* element names with *URIs*
- A URI (Universal Resource Identifier) is usually used for *identifying* a resource on the Web
- (A Uniform Resource Locator (URL) is a special type of URI)
- A *qualified name* then consists of two parts:
  \[ \text{namespace:local-name} \]
- e.g., `<http://www.w3.org/2000/svg:circle ... />`
- where `http://www.w3.org/2000/svg` is a URI and namespace
- The URI does *not* have to reference a real Web resource
- URIs only disambiguate names; they don’t have to define them
- In this case, the browser knows the SVG namespace and behaves accordingly
Namespace declarations

- using URIs everywhere is very cumbersome
- so namespaces are used indirectly using
  - namespace declarations and
  - associated prefixes (user-specified)

```xml
... xmlns:svg="http://www.w3.org/2000/svg">
  <p>A circle looks like this
  ...
  <svg:circle ... />
  ...
</...>
```

- The `xmlns:svg` attribute
  - declares the namespace `http://www.w3.org/2000/svg`
  - associates it with prefix `svg`
Scope of namespace declarations

- the *scope* of a namespace declaration is
  - the element containing the declaration
  - and all its *descendants* (those elements nested inside the element)
  - can be overridden by *nested* declarations

- both elements and attributes can be qualified with namespaces
- unprefixed element names are assigned a *default* namespace
- default namespace declaration: `xmlns="URI"`
Namespaces example

<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:svg="http://www.w3.org/2000/svg">

...  
<p>A circle looks like this</p>
<svg:svg ... >
  ...
  <svg:circle ... />
  ...
</svg:svg>

and has
...
</p>
</html>

- html and p are in the default namespace
  (http://www.w3.org/1999/xhtml)
Namespaces example (2)

```html
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:svg="http://www.w3.org/2000/svg">
 ...
 <p>A circle looks like this
     <svg:svg ...
         ...
     </svg:svg>
     ...
     <svg:circle ...
         ...
     </svg:circle>
     ...
 </svg:svg>
     and has ...
 ...
</p>
</html>
```

- namespace for `svg` and `circle` is `http://www.w3.org/2000/svg`
- note that `svg` is used both as a prefix and as an element name
Summary

- This chapter gave a brief summary of XML
- Only the most important aspects (which are needed later on) were covered
- More details can be found
  - in the books listed in the Introduction
  - on numerous websites, e.g., World Wide Web Consortium or w3schools.com
Chapter 3

Document Type Definitions
A document type is defined by specifying the constraints which any document which is an instance of the type must satisfy.

For example,
- in an HTML document, one paragraph cannot be nested inside another.
- in an SVG document, every circle element must have an \( r \) (radius) attribute.

Document types are
- useful for restricting authors to use particular representations.
- important for correct processing of documents by software.
Languages for Defining Document Types

There are many languages for defining document types on the Web, e.g.,

- document type definitions (DTDs)
- XML schema definition language (XSDL)
- relaxNG
- schematron

We will consider the first two of these
Document Type Definitions (DTDs)

- A DTD defines a *class* of documents
- The structural constraints are specified using an *extended context-free grammar*
- This defines
  - *element* names and their allowed contents
  - *attribute* names and their allowed values
  - *entity* names and their allowed values
Valid XML

- A valid XML document
  - is well-formed and
  - has been validated against a DTD
  - (the DTD is specified in the document — see later)
The syntax for an element declaration in a DTD is:

```
<!ELEMENT name ( model )>
```

where
- `ELEMENT` is a keyword
- `name` is the element name being declared
- `model` is the element `content model` (the allowed contents of the element)

The content model is specified using a *regular expression* over element names.

The regular expression specifies the permitted *sequences* of element names.
Examples of DTD element declarations

- An **html** element must contain a **head** element followed by a **body** element:

  ```xml
  <!ELEMENT html (head, body) >
  ```

  where "\," is the *sequence* (or concatenation) operator
Examples of DTD element declarations

- An `html` element must contain a `head` element followed by a `body` element:
  ```xml
  <!ELEMENT html (head, body) >
  ```
  where "," is the `sequence` (or concatenation) operator

- A `list` element (not in HTML) must contain either a `ul` element or an `ol` element (but not both):
  ```xml
  <!ELEMENT list (ul | ol) >
  ```
  where "|" is the `alternation` (or "exclusive or") operator
Examples of DTD element declarations

- An `html` element must contain a `head` element followed by a `body` element:
  ```xml
  <!ELEMENT html (head, body) >
  ```
  where "","" is the `sequence` (or concatenation) operator

- A `list` element (not in HTML) must contain either a `ul` element or an `ol` element (but not both):
  ```xml
  <!ELEMENT list (ul | ol) >
  ```
  where "|" is the `alternation` (or "exclusive or") operator

- A `ul` element must contain zero or more `li` elements:
  ```xml
  <!ELEMENT ul (li)* >
  ```
  where "*" is the `repetition` (or "Kleene star") operator
DTD syntax (1)

In the table below:
- $e$ denotes any element name, the simplest regular expression
- $\alpha$ and $\beta$ denote regular expressions

<table>
<thead>
<tr>
<th>DTD Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>element $e$ must occur</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>elements must match $\alpha$</td>
</tr>
<tr>
<td>$(\alpha)$</td>
<td>elements must match $\alpha$</td>
</tr>
<tr>
<td>$\alpha, \beta$</td>
<td>elements must match $\alpha$ followed by $\beta$</td>
</tr>
<tr>
<td>$\alpha \mid \beta$</td>
<td>elements must match either $\alpha$ or $\beta$ (not both)</td>
</tr>
<tr>
<td>$\alpha^*$</td>
<td>elements must match zero or more occurrences of $\alpha$</td>
</tr>
</tbody>
</table>
## DTD syntax (2)

<table>
<thead>
<tr>
<th>DTD Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha+$</td>
<td>one or more sequences matching $\alpha$ must occur</td>
</tr>
<tr>
<td>$\alpha?$</td>
<td>zero or one sequences matching $\alpha$ must occur</td>
</tr>
<tr>
<td>EMPTY</td>
<td>no element content is allowed</td>
</tr>
<tr>
<td>ANY</td>
<td>any content (of declared elements and text) is allowed</td>
</tr>
<tr>
<td>#PCDATA</td>
<td>content is text rather than elements</td>
</tr>
</tbody>
</table>

- $\alpha+$ is short for $\left(\alpha, \alpha^*\right)$
- $\alpha?$ is short for $\left(\alpha | \text{EMPTY}\right)$
- #PCDATA stands for “parsed character data,” meaning an XML parser should parse the text to resolve character and entity references
RSS is a simple XML vocabulary for use in news feeds
RSS stands for *Really Simple Syndication*, among other things
The root (document) element is `rss`
`rss` has a single child called `channel`
`channel` has a `title` child, any number of `item` children (and others)
Each `item` (news story) has a `title`, `description`, `link`, `pubDate`, ...

P. Atzeni (heavily from Peter Wood)
RSS Example Outline

```xml
<rss version="2.0">
  <channel>
    <title> BBC News - World </title>
    ... 
    <item>
      <title> Hollande becomes French president </title>
      ... 
    </item>
    <item>
      <title> New Greece poll due as talks fail </title>
      ... 
    </item>
    <item>
      <title> EU forces attack Somalia pirates </title>
    </item>
  </channel>
</rss>
```
RSS Example Fragment (channel)

```
<channel>
  <title> BBC News - World </title>
  <link>http://www.bbc.co.uk/news/world/...</link>
  <description>The latest stories from the World section of the BBC News web site.</description>
  <lastBuildDate>Tue, 15 May 2012 13:42:05 GMT</lastBuildDate>
  <ttl>15</ttl>
  ...
</channel>
```
RSS Example Fragment (first item)

<item>
  <title>Hollande becomes French president</title>
  <description>Francois Hollande says he is fully aware of the challenges facing France after being sworn in as the country’s new president.</description>
  <link>http://www.bbc.co.uk/news/world-europe-...</link>
  <pubDate>Tue, 15 May 2012 11:44:17 GMT</pubDate>
  ...
</item>
RSS Example Fragment (second item)

<item>
  <title>New Greece poll due as talks fail</title>
  <description>Greece is set to go to the polls again after parties failed to agree on a government for the debt-stricken country, says Socialist leader Evangelos Venizelos.</description>
  <link>http://www.bbc.co.uk/news/world-europe-...</link>
  <pubDate>Tue, 15 May 2012 13:52:38 GMT</pubDate>
  ...
</item>
<item>
  <title>EU forces attack Somalia pirates</title>
  <description>EU naval forces conduct their first raid on pirate bases on the Somali mainland, saying they have destroyed several boats.</description>
  <link>http://www.bbc.co.uk/news/world-africa-...</link>
  <pubDate>Tue, 15 May 2012 13:19:51 GMT</pubDate>
  ...
</item>
Simplified DTD for RSS

```xml
<!ELEMENT rss (channel)>
<!ELEMENT channel (title, link, description, lastBuildDate?, ttl?, item+)> 
<!ELEMENT item (title, description, link?, pubDate?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT link (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT lastBuildDate (#PCDATA)>
<!ELEMENT ttl (#PCDATA)>
<!ELEMENT pubDate (#PCDATA)>
```
Validation of XML Documents

- Recall that an XML document is called valid if it is well-formed and has been validated against a DTD.
- Validation is essentially checking that the XML document, viewed as a tree, is a parse tree in the language specified by the DTD.
- We can use the W3C validator service (suggestion, pass the URI; use two files, one for the XML document and the other for the DTD).
- Each of the following files has the same DTD specified (as on the previous slide):
  - `rss-invalid.xml` giving results
  - `rss-valid.xml` giving results
Referencing a DTD

- The DTD to be used to validate a document can be specified
  - internally (as part of the document)
  - externally (in another file)
- done using a *document type declaration*
- *declare* document to be of type given in DTD
- e.g., `<!DOCTYPE rss ...>`
Declaring an Internal DTD

<?xml version="1.0"?>
<!DOCTYPE rss [
    <!-- all declarations for rss DTD go here -->
    ...
    <!ELEMENT rss ... >
    ...
]>
<rss>
    <!-- This is an instance of a document of type rss -->
    ...
</rss>

- element `rss` must be defined in the DTD
- name after `DOCTYPE` (i.e., `rss`) must match root element of document
Declaring an External DTD (1)

```xml
<?xml version="1.0"?>
<!DOCTYPE rss SYSTEM "rss.dtd">
<rss>
    <!-- This is an instance of a document of type rss -->
    ...
</rss>
```

- what follows SYSTEM is a **URI**
- rss.dtd is a relative URI, assumed to be in same directory as source document
Declaring an External DTD (2)

```xml
<?xml version="1.0"?>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
 "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
<math>
  <!-- This is an instance of a mathML document type -->
  ...
</math>
```

- **PUBLIC** means what follows is a *formal public identifier* with 4 fields:
  1. **ISO** for ISO standard, + for approval by other standards body, and - for everything else
  2. **owner** of the DTD: e.g., W3C
  3. **title** of the DTD: e.g., DTD MathML 2.0
  4. **language** abbreviation: e.g., EN

- **URI** gives location of DTD
More on RSS

- The RSS 2.0 specification actually states that, for each item, at least one of title or description must be present.
- How can we modify our previous DTD to specify this?
More on RSS

- The RSS 2.0 specification actually states that, for each item, *at least one of* title or description must be present.
- How can we modify our previous DTD to specify this?
- The allowed sequences are:
  1. title
  2. title description
  3. description
The RSS 2.0 specification actually states that, for each item, at least one of title or description must be present.

How can we modify our previous DTD to specify this?

The allowed sequences are:

1. title
2. title description
3. description

So what about the following regular expression?

title | (title, description) | description
Non-Deterministic Regular Expressions

- The regular expression
  \[\text{title} \mid (\text{title, description}) \mid \text{description}\]
  is non-deterministic

- This means that a parser must read ahead to find out which part of the regular expression to match

- e.g., given a \text{title} element in the input, which of the following expressions should a parser try to match?
  - title or
  - title description
Non-Deterministic Regular Expressions

- The regular expression
  \[\text{title} \mid (\text{title, description}) \mid \text{description}\]
  is non-deterministic

- This means that a parser must read ahead to find out which part of the regular expression to match

- e.g., given a \text{title} element in the input, which of the following expressions should a parser try to match?
  - \text{title} or
  - \text{title description}

- It needs to read the next element to check whether or not it is \text{description}
Non-Deterministic vs Deterministic Regular Expressions

- Non-deterministic regular expressions are *forbidden* by DTDs and XSDL
- They are allowed by RelaxNG
- A non-deterministic regular expression can always be rewritten to be deterministic
Non-Deterministic vs Deterministic Regular Expressions

- Non-deterministic regular expressions are *forbidden* by DTDs and XSDL.
- They are allowed by RelaxNG.
- A non-deterministic regular expression can always be rewritten to be deterministic.
- E.g.,
  
  \[ \text{title} \mid (\text{title}, \text{description}) \mid \text{description} \]
  
  can be rewritten as
  
  \[(\text{title}, \text{description}?) \mid \text{description}\]
Non-Deterministic vs Deterministic Regular Expressions

- Non-deterministic regular expressions are *forbidden* by DTDs and XSDL
- They are allowed by RelaxNG
- A non-deterministic regular expression can always be rewritten to be deterministic
- e.g.,
  
  title | (title, description) | description
  
  can be rewritten as
  
  (title, description?) | description

- The rewriting may cause an exponential increase in size
Attributes

- Recall that attribute name-value pairs are allowed in start tags, e.g., `version="2.0"` in the `rss` start tag.

- Allowed attributes for an element are defined in an *attribute list declaration*: e.g., for `rss` and `guid` elements:

  ```xml
  <!ATTLIST rss
      version CDATA #FIXED "2.0" >
  <!ATTLIST guid
      isPermaLink (true|false) "true" >
  
  attribute definition comprises
  ```
  - *attribute name*, e.g., `version`
  - *type*, e.g., `CDATA`
  - *default*, e.g., "true"
Some Attribute Types

- **CDATA**: any valid character data
- **ID**: an identifier unique within the document
- **IDREF**: a reference to a unique identifier
- **IDREFS**: a reference to several unique identifiers (separated by white-space)
- **(a|b|c)**, e.g.: *(enumerated attribute type)* possible values are one of a, b or c
- ...
Attribute Defaults

- #IMPLIED: attribute may be omitted (optional)
- #REQUIRED: attribute must be present
- #FIXED "x", e.g.: attribute optional; if present, value must be x
- "x", e.g.: value will be x if attribute is omitted
Mixed Content

- In **RSS**, the content of each element comprised either only other elements or only text.
- In **HTML**, on the other hand, paragraph elements allow text interleaved with various in-line elements, such as `<em>`, `<img>`, `<b>`, etc.
- Elements like HTML paragraphs are said to have *mixed content*.
- How do we define mixed content models in a DTD?
Mixed Content Models

- Say we want to mix text with elements `em`, `img` and `b` as the allowed contents of a `p` element.
- The DTD content model would be as follows:
  ```xml
  <!ELEMENT p (#PCDATA | em | img | b)* >
  ```
  - #PCDATA must be first (in the definition)
  - It must be followed by the other elements separated by `|`
  - The subexpression must have `*` applied to it
- These restrictions limit our ability to constrain the content model (see XSDL later).
Entities

- An entity is a physical unit such as a character, string or file — essentially, they are “macros”
- Entities allow
  - references to non-keyboard characters
  - abbreviations for frequently used strings
  - documents to be broken up into multiple parts
- An entity declaration in a DTD associates a name with an entity, e.g.,
  ```xml
  <!ENTITY BBK "Birkbeck, University of London">
  ```
- An entity reference, e.g., &BBK; substitutes value of entity for its name in document
- An entity must be declared before it is referenced
General Entities

- BBK is an example of a *general entity*
- In XML, only 5 general entity declarations are built-in
  - `&`, `&lt;`, `&gt;`, `&quot;`, `&apos;`,
- All other entities must be declared in a DTD
- The values of *internal* entities are defined in the same document as references to them
- The values of *external* entities are defined elsewhere, e.g.,
  - `<!ENTITY HTML-chapter SYSTEM "html.xml" >`
    - then `&HTML-chapter;` includes the contents of file `html.xml` at the point of reference
    - `standalone="no"` must be included in the XML declaration
Parameter Entities

Parameter entities are

- used only within XML markup declarations
- declared by inserting % between ENTITY and name, e.g.,

  ```xml
  <!ENTITY % list "OL | UL" >
  <!ENTITY % heading "H1 | H2 | H3 | H4 | H5 | H6" >
  ```

- referenced using % and ; delimiters, e.g.,

  ```xml
  <!ENTITY % block "P | %list; | %heading; | ..." >
  ```

As an example, see the HTML 4.01 DTD
Limitations of DTDs

- There is no data typing, especially for element content.
- They are only marginally compatible with namespaces.
- We cannot use mixed content and enforce the order and number of child elements.
- It is clumsy to enforce the presence of child elements without also enforcing an order for them (i.e. no & operator from SGML).
- Element names in a DTD are global (see later).
- They use non-XML syntax.
- The XML Schema Definition Language, e.g., addresses these limitations.
Chapter 4

XML Schema Definition Language (XSDL)
XML Schema

- XML Schema is a W3C Recommendation
  - XML Schema Part 0: Primer
  - XML Schema Part 1: Structures
  - XML Schema Part 2: Datatypes
- describes permissible contents of XML documents
- uses XML syntax
- sometimes referred to as XSDL: XML Schema Definition Language
- addresses a number of limitations of DTDs
Simple example

- file `greeting.xml` contains:
  ```xml
  <?xml version="1.0"?>
  <greet>Hello World!</greet>
  ```

- file `greeting.xsd` contains:
  ```xml
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="greet" type="xsd:string"/>
  </xsd:schema>
  ```

  declares element with name `greet` to be of built-in type `string`

  `xsd` is prefix for the namespace for the "schema of schemas"
## DTDs vs. schemas

<table>
<thead>
<tr>
<th>DTD</th>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;!ELEMENT&gt;</code> declaration</td>
<td><code>xsd:element element</code></td>
</tr>
<tr>
<td><code>&lt;!ATTLIST&gt;</code> declaration</td>
<td><code>xsd:attribute element</code></td>
</tr>
<tr>
<td><code>&lt;!ENTITY&gt;</code> declaration</td>
<td>(not available)</td>
</tr>
<tr>
<td>#PCDATA content</td>
<td><code>xsd:string type</code></td>
</tr>
<tr>
<td>(not available)</td>
<td>other data types</td>
</tr>
</tbody>
</table>
Schemas and namespaces

- Schemas are designed to be compatible with namespaces.
- A schema can define structures for a particular namespace.
  - This is called the *target* namespace.
- A document using this namespace can refer to the schema for validation.
- Schemas can also be defined for document types which do not use namespaces.
  - In this case, there is no target namespace.
Schemas and namespaces

- Schemas are designed to be compatible with namespaces.
- A schema can define structures for a particular namespace.
  - This is called the target namespace.
- A document using this namespace can refer to the schema for validation.
- Schemas can also be defined for document types which do not use namespaces.
  - In this case, there is no target namespace.
- We will consider only the case without namespaces.
Linking a schema to a document (no namespaces)

- `xsi:noNamespaceSchemaLocation` attribute on root element
- this says no target namespace is declared in the schema
- `xsi` prefix is mapped to the URI: `http://www.w3.org/2001/XMLSchema-instance`
- this namespace defines global attributes that relate to schemas and can occur in instance documents
- for example:

```xml
<greet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="greeting.xsd">
    Hello World!
</greet>
```
Validating a document

- a validator (found yesterday — it seems ok):
More complex document example

```xml
<cd xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="cd.xsd">
    <composer>Johannes Brahms</composer>
    <performance>
        <composition>Piano Concerto No. 2</composition>
        <soloist>Emil Gilels</soloist>
        <orchestra>Berlin Philharmonic</orchestra>
        <conductor>Eugen Jochum</conductor>
        <recorded>1972</recorded>
    </performance>
    <performance>
        <composition>Fantasias Op. 116</composition>
        <soloist>Emil Gilels</soloist>
        <recorded>1976</recorded>
    </performance>
    <length>PT1H13M37S</length>
</cd>
```
Simple and complex data types

- XSDL allows the definition of *data types* as well as declarations of elements and attributes
- simple data types can contain only text (i.e., no markup)
  - e.g., values of attributes
  - e.g., elements without children or attributes
- complex data types can contain
  - child elements (i.e., markup) or
  - attributes
More complex schema example

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="cd" type="CDType"/>
</xsd:schema>
```
More complex schema example

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="cd" type="CDType"/>
  <xsd:complexType name="CDType">
    <xsd:sequence>
      <xsd:element name="composer" type="xsd:string"/>
      <xsd:element name="performance" type="PerfType" maxOccurs="unbounded"/>
      <xsd:element name="length" type="xsd:duration" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="cd" type="CDType"/>
    <xsd:complexType name="CDType">
        <xsd:sequence>
            <xsd:element name="composer" type="xsd:string"/>
            <xsd:element name="performance" type="PerfType" maxOccurs="unbounded"/>
            <xsd:element name="length" type="xsd:duration" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
More complex schema example

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="cd" type="CDType"/>
  <xsd:complexType name="CDType">
    <xsd:sequence>
      <xsd:element name="composer" type="xsd:string"/>
      <xsd:element name="performance" type="PerfType"
                   maxOccurs="unbounded"/>
      <xsd:element name="length" type="xsd:duration"
                   minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
More complex schema example

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="cd" type="CDType"/>
  <xsd:complexType name="CDType">
    <xsd:sequence>
      <xsd:element name="composer" type="xsd:string"/>
      <xsd:element name="performance" type="PerfType" maxOccurs="unbounded"/>
      <xsd:element name="length" type="xsd:duration" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  ...
</xsd:schema>
```
Main schema components

- **xsd:element** declares an element and assigns it a type, e.g.,
  ```xml
  <xsd:element name="composer" type="xsd:string"/>
  ```
  using a built-in, simple data type, or
  ```xml
  <xsd:element name="cd" type="CDType"/>
  ```
  using a user-defined, complex data type
Main schema components

- `xsd:element` *declares* an element and assigns it a type, e.g.,
  ```xml
  <xsd:element name="composer" type="xsd:string"/>
  ```
  using a built-in, simple data type, or
  ```xml
  <xsd:element name="cd" type="CDType"/>
  ```
  using a user-defined, complex data type

- `xsd:complexType` *defines* a new type, e.g.,
  ```xml
  <xsd:complexType name="CDType">
  ...
  </xsd:complexType>
  ```

- defining named types allows reuse (and may help readability)
Main schema components

- **xsd:element** *declares* an element and assigns it a type, e.g.,
  
  <xsd:element name="composer" type="xsd:string"/>
  
  using a built-in, simple data type, or
  
  <xsd:element name="cd" type="CDType"/>
  
  using a user-defined, complex data type

- **xsd:complexType** *defines* a new type, e.g.,
  
  <xsd:complexType name="CDType">
    ...
  </xsd:complexType>

- defining named types allows reuse (and may help readability)

- **xsd:attribute** *declares* an attribute and assigns it a type (see later)
Structuring element declarations

- **xsd:sequence**
  - requires elements to occur in order given
  - analogous to `, in DTDs

- **xsd:choice**
  - allows only one of the given elements to occur
  - analogous to `| in DTDs

- **xsd:all**
  - all elements must occur, but in any order
  - analogous to `& in SGML DTDs
Defining number of element occurrences

- `minOccurs` and `maxOccurs` attributes control the number of occurrences of an element, sequence or choice
- `minOccurs` must be a non-negative integer
- `maxOccurs` must be a non-negative integer or unbounded
- Default value for each of `minOccurs` and `maxOccurs` is 1
Another complex type example

```xml
<xsd:complexType name="PerfType">
  <xsd:sequence>
    <xsd:element name="composition" type="xsd:string"/>
    <xsd:element name="soloist" type="xsd:string" minOccurs="0"/>
  </xsd:sequence minOccurs="0">
    <xsd:element name="orchestra" type="xsd:string"/>
    <xsd:element name="conductor" type="xsd:string"/>
  </xsd:sequence>
  <xsd:element name="recorded" type="xsd:gYear"/>
</xsd:complexType>
```
An equivalent DTD

```xml
<!ELEMENT CD (composer, (performance)+, (length)?)>  
<!ELEMENT performance (composition, (soloist)?,  
                      (orchestra, conductor)?, recorded)>  
<!ELEMENT composer (#PCDATA)>  
<!ELEMENT length (#PCDATA)>    <!-- duration -->  
<!ELEMENT composition (#PCDATA)>  
<!ELEMENT soloist (#PCDATA)>  
<!ELEMENT orchestra (#PCDATA)>  
<!ELEMENT conductor (#PCDATA)>  
<!ELEMENT recorded (#PCDATA)>    <!-- gYear -->
```
Declaring attributes

- **Use** `xsd:attribute` element inside an `xsd:complexType`
  - has attributes `name`, `type`, e.g.,
    ```xml
    <xsd:attribute name="version" type="xsd:number"/>
    ```
  - attribute `use` is optional
    - if omitted means attribute is optional (like `#IMPLIED`)
    - for required attributes, say `use="required"` (like `#REQUIRED`)
  - for fixed attributes, say `fixed="..."` (like `#FIXED`), e.g.,
    ```xml
    <xs:attribute name="version" type="xs:number" fixed="2.0"/>
    ```
  - for attributes with default value, say `default="..."`
  - for enumeration, use `xsd:simpleType`
- Attributes must be declared at the end of an `xsd:complexType`
Locally-scoped element names

- in DTDs, all element names are *global*
- XML schema allows element types to be local to a context, e.g.,

  `<xsd:element name="book">
    <xsd:element name="title"> ... </xsd:element>
    ...
  </xsd:element>`

  `<xsd:element name="employee">
    <xsd:element name="title"> ... </xsd:element>
    ...
  </xsd:element>`

- content models for two occurrences of `title` can be different
Simple data types

- form a type hierarchy; the root is called *anyType*
  - all complex types
  - *anySimpleType*
    - string
    - boolean, e.g., true
    - anyUri, e.g., http://www.dcs.bbk.ac.uk/~ptw/home.html
    - duration, e.g., P1Y2M3DT10H5M49.3S
    - gYear, e.g., 1972
    - float, e.g., 123E99
    - decimal, e.g., 123456.789
    - ...

- lowest level above are the *primitive data types*

- for a full list, see Simple Types in the Primer
Primitive date and time types

- **date**, e.g., 1994-04-27
- **time**, e.g., 16:50:00+01:00 or 15:50:00Z if in Co-ordinated Universal Time (UTC)
- **dateTime**, e.g., 1918-11-11T11:00:00.000+01:00
- **duration**, e.g., P2Y1M3DT20H30M31.4159S
- "Gregorian" calendar dates (introduced in 1582 by Pope Gregory XIII):
  - **gYear**, e.g., 2001
  - **gYearMonth**, e.g., 2001-01
  - **gMonthDay**, e.g., --12-25 (note hyphen for missing year)
  - **gMonth**, e.g., --12-- (note hyphens for missing year and day)
  - **gDay**, e.g., ---25 (note only 3 hyphens)
Built-in derived string types

Derived from `string`:

- `normalizedString` (**newline, tab, carriage-return are converted to spaces**)
  - `token` (**adjacent spaces collapsed to a single space; leading and trailing spaces removed**)
    - language, e.g., `en`
    - name, e.g., `my:name`
Built-in derived string types

Derived from string:
- normalizedString (newline, tab, carriage-return are converted to spaces)
  - token (adjacent spaces collapsed to a single space; leading and trailing spaces removed)
    - language, e.g., en
    - name, e.g., my:name

Derived from name:
- NCNAME ("non-colonized" name), e.g., myName
  - ID
  - IDREF
  - ENTITY
Built-in derived numeric types

Derived from decimal:

- **integer** (decimal with no fractional part), e.g., -123456
  - **nonPositiveInteger**, e.g., 0, -1
    - **negativeInteger**, e.g., -1
  - **nonNegativeInteger**, e.g., 0, 1
    - **positiveInteger**, e.g., 1
    - **...**
  - **...**
User-derived simple data types

- complex data types can be created "from scratch"
- new simple data types must be derived from existing simple data types

- derivation can be by one of:
  - extension
  - list: a list of values of an existing data type
  - union: allows values from two or more data types
  - restriction: limits the values allowed using, e.g.,
    - maximum value (e.g., 100)
    - minimum value (e.g., 50)
    - length (e.g., of string or list)
    - number of digits
    - enumeration (list of values)
    - pattern

The above constraints are known as facets.
User-derived simple data types

- complex data types can be created "from scratch"
- new simple data types must be derived from existing simple data types
- derivation can be by one of
  - **extension**
    - list: a list of values of an existing data type
    - union: allows values from two or more data types
  - **restriction**: limits the values allowed using, e.g.,
    - maximum value (e.g., 100)
    - minimum value (e.g., 50)
    - length (e.g., of string or list)
    - number of digits
    - enumeration (list of values)
    - pattern
  - above constraints are known as facets
Restriction by enumeration

```xml
<xsd:element name="MScResult">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="distinction"/>
      <xsd:enumeration value="merit"/>
      <xsd:enumeration value="pass"/>
      <xsd:enumeration value="fail"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

- contents of `MScResult` element is a restriction of the `xsd:string` type
- must be one of the 4 values given
- e.g., `<MScResult>pass</MScResult>`
Restriction by values

- examMark can be from 0 to 100

```xml
<xsd:element name="examMark">
    <xsd:simpleType>
        <xsd:restriction base="xsd:nonNegativeInteger">
            <xsd:maxInclusive value="100"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```
Restriction by values

- **examMark** can be from 0 to 100
  
  ```xml
  <xsd:element name="examMark">
    <xsd:simpleType>
      <xsd:restriction base="xsd:nonNegativeInteger">
        <xsd:maxInclusive value="100"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  
  or, equivalently

  ```xml
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="100"/>
  </xsd:restriction>
  ```
Restriction by pattern

```xml
<xsd:element name="zipcode">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{5}(-\d{4})?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

- The `value` attribute contains a *regular expression*
- `\d` means any digit
- `()` used for grouping
- `x{5}` means exactly 5 x’s (in a row)
- `x?` indicates zero or one x
- Zipcode examples: 90720-1314 and 22043
Document with mixed content

- We may want to mix elements and text, e.g.:

  <letter>
  Dear Mr <name>Smith</name>,
  Your order of <quantity>1</quantity> <product>Baby Monitor</product> was shipped on <date>1999-05-21</date>. ....
  </letter>

- A DTD would have to contain:

  <!ELEMENT letter (#PCDATA|name|quantity|product|date)*)>

  which cannot enforce the order of subelements
Schema fragment declaring mixed content

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="letter">
        <xsd:complexType mixed="true">
            <xsd:sequence>
                <xsd:element name="name" type="xsd:string"/>
                <xsd:element name="quantity" type="xsd:positiveInteger"/>
                <xsd:element name="product" type="xsd:string"/>
                <xsd:element name="date" type="xsd:date" minOccurs="0"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```
Summary

XSDL provides, e.g.:

- compatibility with namespaces
- many built-in data types
- user-defined (derived) data types
- locally-scoped element declarations
- more control over mixed content models
Chapter 6

XPath
Introduction

- XPath is a language that lets you identify particular parts of XML documents.
- XPath interprets XML documents as nodes (with content) organised in a tree structure.
- XPath indicates nodes by (relative) position, type, content, and several other criteria.
- Basic syntax is somewhat similar to that used for navigating file hierarchies.
- **XPath 1.0** (1999) and **2.0** (2010) are W3C recommendations.
Some Tools for XPath

- **Saxon** (specifically Saxon-HE which implements XPath 2.0, XQuery 1.0 and XSLT 2.0)
- **eXist-db** (a native XML database system supporting XPath 2.0, most of XQuery 1.0 and 3.0, and XSLT 1.0)
- **XPath Checker** (add-on for Firefox)
- **XPath Expression Testbed** (available online)
- **http://videlibri.sourceforge.net/cgi-bin/xidelcgi** (also available online)
XPath’s data model has some non-obvious features:

- The tree’s root node is not the same as the document’s root (document) element
- The tree’s root node contains the entire document including the root element (and comments and processing instructions that appear before it)
- XPath’s data model does not include everything in the document: XML declaration and DTD are not addressable
- `xmlns` attributes are reported as namespace nodes
Data Model (2)

- There are 6 types of node:
  - root
  - element
  - attribute
  - text
  - comment
  - processing instruction

- Element nodes have an associated set of attribute nodes
- Attribute nodes are not children of element nodes
- The order of child element nodes is significant
- We will only consider the first 4 types of node
Example (1)

Recall our CD library example

```
<CD-library>
  <CD number="724356690424">
    <performance>
      <composer>Frederic Chopin</composer>
      <composition>Waltzes</composition>
      <soloist>Dinu Lipatti</soloist>
      <date>1950</date>
    </performance>
  </CD>
  ...
```
Example (2)

...<CD number="419160-2">
   <composer>Johannes Brahms</composer>
   <soloist>Emil Gilels</soloist>
   <performance>
      <composition>Piano Concerto No. 2</composition>
      <orchestra>Berlin Philharmonic</orchestra>
      <conductor>Eugen Jochum</conductor>
      <date>1972</date>
   </performance>
   <performance>
      <composition>Fantasias Op. 116</composition>
      <date>1976</date>
   </performance>
</CD>
...

P. Atzeni (heavily from Peter Wood)
Example (3)

...<CD number="449719-2">
  <soloist>Martha Argerich</soloist>
  <orchestra>London Symphony Orchestra</orchestra>
  <conductor>Claudio Abbado</conductor>
  <date>1968</date>
  <performance>
    <composer>Frederic Chopin</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
  <performance>
    <composer>Frances Liszt</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
</CD>
...

P. Atzeni (heavily from Peter Wood)
Example (4)

...  
<CD number="430702-2">  
  <composer>Antonin Dvorak</composer>  
  <performance>  
    <composition>Symphony No. 9</composition>  
    <orchestra>Vienna Philharmonic</orchestra>  
    <conductor>Kirill Kondrashin</conductor>  
    <date>1980</date>  
  </performance>  
<performance>  
  <composition>American Suite</composition>  
  <orchestra>Royal Philharmonic</orchestra>  
  <conductor>Antal Dorati</conductor>  
  <date>1984</date>  
</performance>  
</CD>  
</CD-library>
Example — Tree Structure
Example — Tree Structure
Example — Tree Structure
Example — Tree Structure
Comments on example tree structure

- *attribute* nodes are not shown (for *number* attribute)
- the *root* node is shown as solid black
- all nodes with labels (*C, p, ...*) are *element* nodes
- white nodes without labels are *text* nodes
- not all of the tree is shown
The most useful XPath expression is a *location path*: e.g., `/CD-library/CD/performance`

A location path consists of at least one *location step*: e.g., `CD-library`, `CD` and `performance` are location steps

A location step takes as input a set of nodes, also called the *context* (to be defined more precisely later)

The location step expression is applied to this node set and results in an output node set

This output node set is used as input for the next location step
Location Path (2)

- There are two different kinds of location paths:
  - *Absolute* location paths
  - *Relative* location paths

An absolute location path
- starts with `/`
- is followed by a relative location path
- is evaluated at the root (context) node of a document
- e.g., `/CD-library/CD/performance`

A relative location path
- is a sequence of location steps
- each separated by `/`
- evaluated with respect to some other context nodes
- e.g., `CD/performance`
Location Path (2)

There are two different kinds of location paths:

- *Absolute* location paths
  - starts with `/`
  - is followed by a relative location path
  - is evaluated at the root (context) node of a document
  - e.g., `/CD-library/CD/performance`

- *Relative* location paths
  - a sequence of location steps
  - each separated by `/`
  - evaluated with respect to some other context nodes
  - e.g., `CD/performance`
Location Path (2)

- There are two different kinds of location paths:
  - *Absolute* location paths
  - *Relative* location paths

- An absolute location path
  - starts with `/`
  - is followed by a relative location path
  - is evaluated at the root (context) node of a document
  - e.g., `/CD-library/CD/performance`

- A relative location path
  - is a sequence of location steps
  - each separated by `/`
  - evaluated with respect to some other context nodes
  - e.g., `CD/performance`
Evaluation of absolute location path
Evaluation of absolute location path

/
Evaluation of absolute location path

/CD-library
Evaluation of absolute location path

/CD-library/CD
Evaluation of absolute location path

/CD-library/CD/performance
### Location Step

- In general, a location step consists of 3 parts:
  - (navigation) axis
  - node test
  - (optional) predicate(s)

- Full syntax is `axis :: node test [ predicate ] ... [ predicate ]`
- *(We used the *abbreviated* syntax in previous examples)*
Location Step

- In general, a location step consists of 3 parts:
  - (navigation) axis
  - node test
  - (optional) predicate(s)

- Full syntax is `axis :: node test [ predicate ] ... [ predicate ]`

- (We used the *abbreviated* syntax in previous examples)

- e.g., `child::CD[composer='Johannes Brahms']`
  - `child` is the axis
  - `CD` is the node test
  - `composer='Johannes Brahms'` is the predicate
### Location Step

- In general, a location step consists of 3 parts:
  - (navigation) axis
  - node test
  - (optional) predicate(s)

- Full syntax is `axis :: node test [ predicate ] . . . [ predicate ]`
- (We used the *abbreviated* syntax in previous examples)
- e.g., `child::CD[composer='Johannes Brahms']`
  - `child` is the axis
  - `CD` is the node test
  - `composer='Johannes Brahms'` is the predicate

- A location step is applied to each node in the context (i.e., each node becomes the context node)
- All resulting nodes are added to the output set of this location step
Evaluation of predicate

/child::CD-library/child::CD
Evaluation of predicate

/child::CD-library/child::CD[composer='Johannes Brahms']
Axioms

- An axis specifies what nodes, relative to the current context node, to consider.
- There are 13 different axes (some can be abbreviated):
  - `self`, abbreviated by `.`
  - `child`, abbreviated by `empty axis`
  - `parent`, abbreviated by `..`
  - `descendant-or-self`, abbreviated by `empty location step`
  - `descendant`, `ancestor`, `ancestor-or-self`
  - `following`, `following-sibling`, `preceding`, `preceding-sibling`
  - `attribute`, abbreviated by `@`
  - `namespace`
Axes

The following slides show (graphical) examples of the axes, assuming the node in bold typeface is the context node.
Self-Axis

- The self-axis just contains the context node itself
Child-Axis

- The child-axis contains the children (direct descendants) of the context node
Parent-Axis

- The parent-axis contains the parent (direct ancestor) of the context node
Descendant-Axis

- The descendant-axis contains all direct and indirect descendants of the context node
The descendant-or-self-axis contains all direct and indirect descendants of the context node + the context node itself.
Ancestor-Axis

- The ancestor-axis contains all direct and indirect ancestors of the context node
Ancestor-Or-Self-Axis

The ancestor-or-self-axis contains all direct and indirect ancestors of the context node + the context node itself.
Following-Axis

- The following-axis contains all nodes that begin after the context node ends
Preceding-Axis

The preceding-axis contains all nodes that end before the context node begins.
Following-Sibling-Axes

- The following-sibling-axis contains all following nodes that have the same parent as the context node.
Preceding-Sibling-Axis

- The preceding-sibling-axis contains all preceding nodes that have the same parent as the context node.
Partitioning

- The axes self, ancestor, descendant, following and preceding partition a document into five disjoint subtrees:
Attribute-Axis

- Attributes are handled in a special way in XPath.
- The attribute-axis contains all the attribute nodes of the context node.
- This axis is empty if the context node is not an element node.
- Does not contain xmlns attributes used to declare namespaces.
Namespace-Axis

- The namespace-axis contains all namespaces in scope of the context node
- This axis is empty if the context node is not an element node
Node Tests

- Once the correct relative position of a node has been identified the type of a node can be tested.
- A *node test* further refines the nodes selected by the location step.
- A double colon `::` separates the axis from the node test.
- There are seven different kinds of node tests:
  
  ```
  name
  prefix::*
  node()
  text()
  comment()
  processing-instruction()
  *
  ```
The *name* node test selects all elements with a matching name

- e.g., if our context is a set of 4 *CD* elements and the location step uses the *child* axis, then we get element nodes with different names
- we can use the *name* node test to return, e.g., only *soloist* elements

Along the attribute-axis it matches all attributes with the same name
Prefix:* 

- Along an element axis, all nodes whose namespace URIs are the same as the prefix are matched.
- This node test is also available for attribute nodes.
Comment, Text, Processing-Instruction

- `comment()` matches all comment nodes
- `text()` matches all text nodes
- `processing-instruction()` matches all processing instructions
Node and *

- `node()` selects all nodes, regardless of type: attribute, element, text, comment, namespace, processing instruction, and root
- usually `*` selects all `element` nodes, regardless of name
  - If the axis is the attribute axis, then it selects all attribute nodes
  - If the axis is the namespace axis, then it selects all namespace nodes
### Key for full CD library example

<table>
<thead>
<tr>
<th>Element name</th>
<th>Abbreviation</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>black</td>
</tr>
<tr>
<td>library</td>
<td>L</td>
<td>white</td>
</tr>
<tr>
<td>cd</td>
<td>C</td>
<td>grey</td>
</tr>
<tr>
<td>performance</td>
<td>p</td>
<td>pink</td>
</tr>
<tr>
<td>composer</td>
<td>c</td>
<td>blue</td>
</tr>
<tr>
<td>composition</td>
<td></td>
<td>green</td>
</tr>
<tr>
<td>soloist</td>
<td>s</td>
<td>yellow</td>
</tr>
<tr>
<td>conductor</td>
<td>t</td>
<td>red</td>
</tr>
<tr>
<td>orchestra</td>
<td>o</td>
<td>brown</td>
</tr>
<tr>
<td>date</td>
<td>d</td>
<td>orange</td>
</tr>
</tbody>
</table>
Full CD library example
Example using * and node()

/CD-library/CD/*/node()
Example showing difference between * and node()

/CD-library/CD/*/*
Example using descendant

//composer (abbreviated syntax) or /descendant-or-self::node()/child::composer (full syntax)
Another example using descendant

//performance/composer or
/descendant-or-self::performance/child::composer
Predicates

- A node set can be reduced further with *predicates*
- While each location step must have an axis and a node test (which may be empty), a predicate is optional
- A predicate contains a Boolean expression which is tested for each node in the resulting node set
- A predicate is enclosed in square brackets `[ ]`
Predicates (2)

- XPath supports a full complement of relational operators, including =, >, <, >=, <=, !=
- XPath also provides Boolean \texttt{and} and \texttt{or} operators to combine expressions logically
- In some cases a predicate may not be a Boolean; then XPath will convert it to one implicitly (if that is possible):
  - an empty node set is interpreted as false
  - a non-empty node set is interpreted as true
Example using a predicate

//performance[composer]
Another example using a predicate

//CD[performance/orchestra]
Example using multiple predicates

`//performance[conductor][date]`
Further examples with predicates

\[
//\text{performance}[\text{composer}='\text{Frederic Chopin}']/\text{composition}
\]
returns

1. \text{<composition>Waltzes</composition>}
2. \text{<composition>Piano Concerto No. 1</composition>}

The two composition nodes have the same value, but they are different nodes.
Further examples with predicates

- \[//\text{performance}[\text{composer}=’\text{Frederic Chopin’}]/\text{composition}\]
  returns
  1. <composition>Waltzes</composition>
  2. <composition>Piano Concerto No. 1</composition>

- \[//\text{CD[@number=’449719-2’]}/\text{composition}\]
  returns
  1. <composition>Piano Concerto No. 1</composition>
  2. <composition>Piano Concerto No. 1</composition>

The two composition nodes have the same value, but they are different nodes.
Functions

- XPath provides many functions that may be useful in predicates.
- Each XPath function takes as input or returns one of these four types:
  - node set
  - string
  - Boolean
  - number
More about Context

Each location step and predicate is evaluated with respect to a given context.

A specific context is defined as \((\langle N_1, N_2, \ldots, N_m \rangle, N_c)\) with

- a context list \(\langle N_1, N_2, \ldots, N_m \rangle\) of nodes in the tree
- a context node \(N_c\) belonging to the list

The context length \(m\) is the size of the context list.

The context node position \(c \in [1, m]\) gives the position of the context node in the list.
More about XPath Evaluation

- Each step $s_i$ is interpreted with respect to a context; its result is a node list.
- A step $s_i$ is evaluated with respect to the context of step $s_{i-1}$.
- More precisely:
  - For $i = 1$ (first step)
    - if the path is absolute, the context is the root of the XML tree;
    - else (relative paths) the context is defined by the environment;
  - For $i > 1$
    - if $\mathcal{N} = \langle N_1, N_2, \ldots N_m \rangle$ is the result of step $s_{i-1}$,
      - step $s_i$ is successively evaluated with respect to the context $(\mathcal{N}, N_j)$, for each $j \in [1, m]$.
- The result of the path expression is the node list obtained after evaluating the last step.
Node-set Functions

- **Node-set functions** operate on or return information about node sets

- **Examples:**
  - `position()`: returns a number equal to the position of the current node in the context list
    - *[position()=i]* can be abbreviated as *[i]*
  - `last()`: returns the size (i.e. the number of nodes in) the context list
  - `count(set)`: returns the size of the argument node set
  - `id(idrefs)`: returns a node set containing all elements in the document with any of the IDs specified by *idrefs*
Example about context

- The expression //CD/performance[2] returns the second performance of each CD, not the second of all performances.
- The result of the step CD is the list of the 4 CD nodes.
- The step performance[2] is evaluated once for each of 4 CD nodes in the context.
Example about context (2)

- The result is the list comprising the second performance element child of each CD:

1. `<performance>`
   - `<composition>`Fantasias Op. 116`</composition>`
   - `<date>`1976`</date>`
   `</performance>`

2. `<performance>`
   - `<composer>`Franz Liszt`</composer>`
   - `<composition>`Piano Concerto No. 1`</composition>`
   `</performance>`

3. `<performance>`
   - `<composition>`American Suite`</composition>`
   - `<orchestra>`Royal Philharmonic`</orchestra>`
   - `<conductor>`Antal Dorati`</conductor>`
   - `<date>`1984`</date>`
   `</performance>`
Problems with XPath processors

- Say we want those *performance* children of *CD* elements that are both the second *performance* and have a *date*

- The following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`

- But different processors give different results!
Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions
Problems with XPath processors

- Say we want those performance children of CD elements that are both the second performance and have a date.
- The following 4 expressions should all be equivalent:
  1. //CD/performance[2][date]
  2. //CD/performance[date][2]
  3. //CD/performance[date and position()=2]
  4. //CD/performance[position()=2 and date]

- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions.
- But, for //CD/performance[date][2], eXist seems to return the second of all performance elements with a date.
Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions
- But, for `//CD/performance[date][2]`, eXist seems to return the second of all `performance` elements with a `date`
- An earlier tool returned, for each `CD`, the second of its `performance` elements that had a `date` (if any)
More about the position() function

- `position()` is a function that returns the position of the current node in the context node set.
- For most axes it counts forward from the context node.
- For the “backward” axes it counts backwards from the context node.
- The “backward” axes are: ancestor, ancestor-or-self, preceding, and preceding-sibling.
Examples using position()

- To get the CD immediately before the one where the composer was Dvorak:
  ```xml
  //CD[composer='Antonin Dvorak']/preceding::CD[1]
  ```
- This selects the third CD
Examples using position() 

- To get the CD immediately before the one where the composer was Dvorak:
  \[
  //CD[\text{composer}='\text{Antonin Dvorak'}]/\text{preceding::CD}[1]
  \]

- This selects the third CD

- To get the last CD (without having to know how many there are), use
  \[
  //CD[\text{position}()=\text{last}()]
  \]
Example using a different axis

//date/following-sibling::* returns the following:

1. <performance>
   <composer>Frederic Chopin</composer>
   <composition>Piano Concerto No. 1</composition>
</performance>

2. <performance>
   <composer>Franz Liszt</composer>
   <composition>Piano Concerto No. 1</composition>
</performance>

only one date element in the document has any following siblings
Examples using count

- //CD[count(performance)=2] returns CD elements with exactly two performance elements as children: the last 3 CDs
Examples using count

- //CD[count(performance)=2] returns CD elements with exactly two performance elements as children: the last 3 CDs
- //CD[performance][performance] of course does not do this:
  - it is equivalent to //CD[performance]
  - which returns CD elements with at least one performance child
More examples using count

- Assume we want the CDs containing only one orchestra element.
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”.
- This is because we are counting the orchestra children of CD elements.
- But orchestras are also represented below performance elements.
More examples using count

- Assume we want the CDs containing only one `orchestra` element
  
  `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”

- This is because we are counting the orchestra `children` of `CD` elements

- But orchestras are also represented below `performance` elements

- What about `//CD[count(//orchestra)=1]`?
  
  - But `//orchestra` is an absolute expression evaluated at the root
  
  - So the answer to `count(//orchestra)is` 4, not 1
More examples using count

- Assume we want the CDs containing only one `orchestra` element
  - `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra `children` of `CD` elements
- But orchestras are also represented below `performance` elements
- What about `//CD[count(//orchestra)=1]`?
  - But `//orchestra` is an absolute expression evaluated at the root
  - So the answer to `count(//orchestra)` is 4, not 1
- What we need is `/CD[count(.//orchestra)=1]`, where “.” represents the current context node
  - This gives us the CDs with the “Berlin Philharmonic” and “London Symphony Orchestra”
String Functions

- **String functions** include basic string operations

Examples:

- `string-length()`: returns the length of a string
- `concat()`: concatenates its arguments in order from left to right and returns the combined string
- `contains(s1, s2)`: returns true if \( s2 \) is a substring of \( s1 \)
- `normalize-space()`: strips all leading and trailing whitespace from its argument
Boolean Functions

*Boolean functions* always return a Boolean with the value true or false:

- `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
- `false()`: returns false
- `not()`: inverts its argument (i.e., true becomes false and vice versa)

Examples:

- `//performance[orchestra][not(conductor)]` returns performance elements which have an orchestra child but no conductor child
- `//CD[not(.//soloist)]` returns CDs containing no soloists
**Boolean Functions**

- **Boolean functions** always return a Boolean with the value true or false:
  - `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
  - `false()`: returns false
  - `not()`: inverts its argument (i.e., true becomes false and vice versa)

- **Examples:**
  - `//performance[orchestra][not(conductor)]` returns performance elements which have an orchestra child but no conductor child
  - `//CD[not(.//soloist)]` returns CDs containing no soloists
**Boolean Functions (2)**

- **boolean()**: converts its argument to a Boolean and returns the result
  - Numbers are false if they are zero or NaN (not a number)
  - Node sets are false if they are empty
  - Strings are false if they have zero length
Number Functions

- **Number functions** include a few simple numeric functions

**Examples:**

- `sum(set)`: converts each node in a node set to a number and returns the sum of these numbers
- `round()`, `floor()`, `ceiling()`: round numbers to integer values
Summary

- XPath is used to navigate through elements and attributes in an XML document.
- XPath is a major element in many W3C standards: XQuery, XSLT, XLink, XPointer.
- It is also used to navigate XML trees represented in Java or JavaScript, e.g.
- So an understanding of XPath is fundamental to much advanced XML usage.
Chapter 9

XQuery
Motivation

- Now that we have XPath, what do we need XQuery for?
- XPath was designed for addressing parts of existing XML documents
- XPath cannot
  - create new XML nodes
  - perform joins between parts of a document (or many documents)
  - re-order the output it produces
  - ...

- Furthermore, XPath
  - has a very simple type system
  - can be hard to read and understand (due to its conciseness)
XQuery closely follows the XML Schema data model
The most general data type is an *item*
An item is either a (single) node or an atomic value
Data Model (2)

- XQuery works on *sequences*, which are series of items
- In XQuery every value is a sequence
  - There is no distinction between a single item and a sequence of length one
- Sequences can only contain items; they cannot contain other sequences
Document Representation

- Every document is represented as a tree of nodes
- Every node has a unique node identity that distinguishes it from other nodes (independent of any ID attributes)
- The first node in any document is the document node (which contains the whole document)
- The order in which the nodes occur in an XML document is called the *document order* (which corresponds to the pre-order traversal of the nodes)
Document Representation (2)

- Attributes are not considered children of an element
  - They occur after their element and before its first child
  - The relative order within the attributes of an element is implementation-dependent
We are now going to look at the query language itself

- Basics
- Creating nodes/documents
- FLWOR expressions
- Advanced topics
Comments

- XQuery uses “smileys” to begin and end comments:
  (: This is a comment :) 
- These are comments found in a query (to comment the query)
  - Not to be confused with comments in XML documents
XQuery supports numeric and string literals

There are three kinds of numeric literals

- Integers (e.g. 3)
- Decimals (e.g. -1.23)
- Doubles (e.g. 1.2e5)

String literals are delimited by quotation marks or apostrophes

- “a string”
- ’a string’
- ’This is a “string”’
Input Functions

- XQuery uses input functions to identify the data to be queried
- There are two different input functions, each taking a single argument
  - doc()
    - Returns an entire document (i.e. the document node)
    - Document is identified by a Universal Resource Identifier (URI)
  - collection()
    - Returns any sequence of nodes that is associated with a URI
    - How the sequence is identified is implementation-dependant
    - For example, eXist allows a database administrator to define collections, each containing a number of documents
Sample Data

In order to illustrate XQuery queries, we use a sample data file `books.xml` which is based on bibliography data.

```xml
<bib>

  <book year='1994'>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison Wesley</publisher>
    <price>65.95</price>
  </book>

</bib>
```
Sample Data (cont’d)

<book year='1992'>
  <title>
    Advanced Programming in the UNIX environment
  </title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison Wesley</publisher>
  <price>65.95</price>
</book>
Sample Data (cont’d)

<book year='2000'>
  <title>Data on the Web</title>
  <author>
    <last>Abiteboul</last>  <first>Serge</first>
  </author>
  <author>
    <last>Buneman</last>  <first>Peter</first>
  </author>
  <author>
    <last>Suciu</last>  <first>Dan</first>
  </author>
  <publisher>Morgan Kaufmann</publisher>
  <price>39.95</price>
</book>
Sample Data (cont’d)

<book year='1999'>
  <title>
    The Economics of Technology and Content for Digital TV
  </title>
  <editor>
    <last>Gerbarg</last>
    <first>Darcy</first>
    <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic</publisher>
  <price>129.95</price>
</book>

</bib>
Input Functions (2)

- `doc("books.xml")` returns the entire document
- A run-time error is raised if the `doc` function is unable to locate the document
XQuery uses XPath to locate nodes in XML data

An XPath expression can be appended to a `doc` (or `collection`) function to select specific nodes

For example, `doc("books.xml")//book` returns all book nodes of `books.xml`
Creating Nodes

- So far, XQuery does not look much more powerful than XPath
- We only located nodes in XML documents
- Now we take a look at how to create nodes
- Note that this creates nodes in the *output* of a query; it does *not* update the document being queried
Creating Nodes (2)

Elements, attributes, text nodes, processing instructions, and comment nodes can all be created using the same syntax as XML.

The following element constructor creates a book element:

```xml
<book year='1977'>
  <title>Harold and the Purple Crayon</title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>
    Harper Collins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
```
Creating Nodes (3)

- Document nodes do not have an explicit syntax in XML.
- XQuery provides a special document node constructor.
- The query
  ```xml
document {}
  ```
- creates an empty document node.
Creating Nodes (4)

Document node constructor can be combined with other constructors to create entire documents

document {
   <!-- I love this book -->
   <book year='1977'>
       <title>Harold and the Purple Crayon</title>
       <author>
           <last>Johnson</last>
           <first>Crockett</first>
       </author>
       <publisher>
           Harper Collins Juvenile Books
       </publisher>
       <price>14.95</price>
   </book>
}

P. Atzeni (heavily from Peter Wood)
Constructors can be combined with other XQuery expressions to generate content dynamically.

In element constructors, curly braces \{\} delimit enclosed expressions which are evaluated to create content.

Enclosed expressions may occur in the content of an element or the value of an attribute.
Creating Nodes (6)

This query creates a list of book titles from books.xml

```xml
<titles count='{ count(doc("books.xml")//title) }'>
  { doc("books.xml")//title }
</titles>
```

The result is:

```
<titles count="4">
  <title>TCP/IP Illustrated</title>
  <title>Advanced Programming ...</title>
  <title>Data on the Web</title>
  <title>The Economics of ...</title>
</titles>
```
Creating Nodes (6)

- This query creates a list of book titles from `books.xml`

```xml
<titles count = '{ count(doc("books.xml")//title) }'>
  { 
    doc("books.xml")//title 
  }
</titles>
```

- The result is:

```xml
<titles count="4">
  <title>TCP/IP Illustrated</title>
  <title>Advanced Programming ...</title>
  <title>Data on the Web</title>
  <title>The Economics of ...</title>
</titles>
```
Whitespace

- Implementations may discard boundary whitespace (whitespace between tags with no intervening non-whitespace)
- This whitespace can be preserved by an `boundary-space` declaration in the `prolog` of a query
- The prolog of a query is an optional section setting up the compile-time context for the rest of the query
Whitespace (2)

- The following query declares that all whitespace in element constructors must be preserved (which will output the element in exactly the same format)

  declare boundary-space preserve;

  
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>

- Omitting this declaration (or setting the mode to strip) will give:

  <author><last>Stevens</last><first>W.</first></author>
Combining and Restructuring

- The expressiveness of XQuery goes beyond just creating nodes
- Information from one or more sources can be combined and restructured to create new results
- We are going to have a look at the most important expressions and functions
FLWOR

- FLWOR expressions (pronounced “flower”) are one of the most powerful and common expressions in XQuery.
- Syntactically, they show similarity to the select-from-where statements in SQL.
- However, FLWOR expressions do not operate on tables, rows, and columns.
The name FLWOR is an acronym standing for the first letter of the clauses that may appear

- For
- Let
- Where
- Order by
- Return
The acronym FLWOR roughly follows the order in which the clauses occur

A FLWOR expression
- starts with one or more `for` or `let` clauses (in any order)
- followed by an optional `where` clause,
- an optional `order by` clause,
- and a required `return` clause
For and Let Clauses

- Every clause in a FLWOR expression is defined in terms of tuples.
- The `for` and `let` clauses produce these tuples.
- Therefore, every FLWOR expression must have at least one `for` or `let` clause.
- We will start with artificial-looking queries to illustrate the inner workings of `for` and `let` clauses.
The following query creates an element named `tuple` in its return clause:

```xquery
for $i in (1, 2, 3)
return
    <tuple><i> { $i } </i></tuple>
```

We bind the variable `$i` to the expression `(1, 2, 3)`, which constructs a sequence of integers.
For and Let Clauses (2)

- The following query creates an element named `tuple` in its return clause

```
for $i in (1, 2, 3)
return
    <tuple><i> { $i } </i></tuple>
```

- We bind the variable `$i` to the expression `(1, 2, 3)`, which constructs a sequence of integers

- The above query results in:

```
<tuple><i> 1 </i></tuple>
<tuple><i> 2 </i></tuple>
<tuple><i> 3 </i></tuple>
```

(a `for` clause preserves order when it creates tuples)
A `let` clause binds a variable to the entire result of an expression.

If there are no `for` clauses, then a single tuple is created.

So the query:

```xml
let $i := (1, 2, 3)
return
  <tuple><i> { $i } </i></tuple>
```

Yields:

```xml
<tuple><i> 1 2 3 </i></tuple>
```
For and Let Clauses (3)

- A `let` clause binds a variable to the entire result of an expression.
- If there are no `for` clauses, then a single tuple is created.
- So the query:

```xquery
let $i := (1, 2, 3)
return
  <tuple><i> { $i } </i></tuple>
```

gives the answer:

```xml
<tuple><i> 1 2 3 </i></tuple>
```
For and Let Clauses (4)

- Variable bindings of let clauses are added to the tuples generated by for clauses.
- So the query:

```xquery
for $i in (1, 2, 3)
let $j := ('a', 'b', 'c')
return
  <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

gives the answer:

```
<tuple><i>1</i><j>a b c</j></tuple>
<tuple><i>2</i><j>a b c</j></tuple>
<tuple><i>3</i><j>a b c</j></tuple>
```
Variable bindings of let clauses are added to the tuples generated by for clauses

So the query:

```xml
for $i in (1, 2, 3)
    let $j := ('a', 'b', 'c')
    return <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

gives the answer:

```xml
<tuple><i>1</i><j>a b c</j></tuple>
<tuple><i>2</i><j>a b c</j></tuple>
<tuple><i>3</i><j>a b c</j></tuple>
```
For and Let Clauses (5)

- **for** and **let** clauses can be bound to any XQuery expression
- Let us do a more realistic example
- List the title of each book in *books.xml* together with the numbers of authors:

```xquery
for $b in doc("books.xml")//book
let $a := $b/author
return
  <book> { $b/title,
    <count> { count($a) } </count> }
</book>
```
For and Let Clauses (6)

This results in:

```xml
<book>
  <title>TCP/IP Illustrated</title>
  <count>1</count>
</book>
<book>
  <title>Advanced Programming ...</title>
  <count>1</count>
</book>
<book>
  <title>Data on the Web</title>
  <count>3</count>
</book>
<book>
  <title>The Economics of Technology ...</title>
  <count>0</count>
</book>
```
Where Clauses

- A `where` clause eliminates tuples that do not satisfy a particular condition.
- A `return` clause is only evaluated for tuples that “survive” the `where` clause.
- The following query returns only books whose prices are less than 50.00:

```xml
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```
Where Clauses

- A `where` clause eliminates tuples that do not satisfy a particular condition.
- A `return` clause is only evaluated for tuples that “survive” the `where` clause.
- The following query returns only books whose prices are less than 50.00:

```xml
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```

- The answer is

```
<title>Data on the Web</title>
```
Order By Clauses

- An order by clause sorts the tuples before the return clause is evaluated.
- If there is no order by clause, then the results are returned in document order.
- The following example lists the titles of books in alphabetical order:

  ```xquery
  for $t in doc("books.xml")//title
  order by $t
  return $t
  ```

- An order spec may also specify whether to sort in ascending or descending order (using ascending or descending).
Return Clauses

- Any XQuery expression may occur in a `return` clause.
- Element constructors are very common in `return` clauses.
- The following query represents an author’s name as a string in a single element:

  ```xquery
  for $a in doc("books.xml")//author
  return
      <author> { string($a/first),
                  string($a/last) } </author>
  ```

  The result is:
  ```xml
  <author> W. Stevens </author>
  <author> W. Stevens </author>
  <author> Serge Abiteboul </author>
  <author> Peter Buneman </author>
  <author> Dan Suciu </author>
  ```
Return Clauses

- Any XQuery expression may occur in a `return` clause
- Element constructors are very common in `return` clauses
- The following query represents an author’s name as a string in a single element

```xml
for $a in doc("books.xml")//author
return
    <author> { string($a/first),
                string($a/last) } </author>
```

- The result is

```
<author> W. Stevens </author>
<author> W. Stevens </author>
<author> Serge Abiteboul </author>
<author> Peter Buneman </author>
<author> Dan Suciu </author>
```
The following query adds another level to the hierarchy:

```xquery
for $a in doc("books.xml")//author
return
  <author>
    <name> { $a/first, $a/last } </name>
  </author>
```

The result is:

```
<author>
  <name>
    <first>W.</first>
    <last>Stevens</last>
  </name>
</author>
...
Return Clauses (2)

The following query adds another level to the hierarchy:

```xquery
for $a in doc("books.xml")//author
return
  <author>
    <name> { $a/first, $a/last } </name>
  </author>
```

The result is

```xml
<author>
  <name>
    <first>W.</first>
    <last>Stevens</last>
  </name>
</author>
...
Formatting XQuery Output

- Standard XQuery parameters can be set to
  - omit the XML declaration in the output (`omit-xml-declaration`)
  - have nested elements in the output indented (`indent`)

- However, it seems that new lines have to be added to the output explicitly using the new line character obtained through the entity reference `&#10;`;

- As an example, see the query on the next slide
Nested Expressions

This query outputs book titles and authors, each on a new line:

```
declare namespace saxon="http://saxon.sf.net/";
declare option saxon:output "omit-xml-declaration=yes";
declare option saxon:output "indent=yes";

let $nl := "&#10;"
for $b in doc("books.xml")//book
return ($b/title,
    for $a in $b/author return ($a, $nl),
    $nl)
```
Nested Expressions

This query outputs book titles and authors, each on a new line:

```xml
declare namespace saxon="http://saxon.sf.net/";
declare option saxon:output "omit-xml-declaration=yes";
declare option saxon:output "indent=yes";

let $nl := "&#10;"
for $b in doc("books.xml")//book
return ($b/title,
    for $a in $b/author return ($a, $nl),
    $nl)
```

- use of the namespace declaration for the software tool Saxon
- character entity reference for the new line character
- `for` clause nested in the `return` clause
- sequences returned by using (and)
Operators

- We have seen a few examples of operators in queries
- Let's consider operators in more detail now
- XQuery has three different kinds of operators
  - Arithmetic operators
  - Comparison operators
  - Sequence operators
Arithmetic Operators

- XQuery supports the arithmetic operators +, -, *, div, idiv, and mod.
- The idiv and mod operators require integer arguments, returning the quotient and the remainder, respectively.
- If an operand is a node, atomization is applied (casting the content to an atomic type).
- If an operand is an empty sequence, the result is an empty sequence.
- If an operand is untyped, it is cast to a double (raising an error if the cast fails).
Comparison Operators

- XQuery has different sets of comparison operators: value comparisons, general comparisons and node (order) comparisons.

- *Value* comparison operators compare atomic values:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>equals</td>
</tr>
<tr>
<td>ne</td>
<td>not equals</td>
</tr>
<tr>
<td>lt</td>
<td>less than</td>
</tr>
<tr>
<td>le</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>gt</td>
<td>greater than</td>
</tr>
<tr>
<td>ge</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>
General Comparisons

- The following query raises an error

```
for $b in doc("books.xml")//book
where $b/author/last eq 'Stevens'
return $b/title
```

because we try to compare several author names to 'Stevens' (books may have more than one author)

- We need a *general* comparison operator for this to work

- A general comparison returns true if *any* value in a sequence of atomic values matches
The following table shows the corresponding general comparison operator for each value comparison operator:

<table>
<thead>
<tr>
<th>value comparison</th>
<th>general comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>=</td>
</tr>
<tr>
<td>ne</td>
<td>!=</td>
</tr>
<tr>
<td>lt</td>
<td>&lt;</td>
</tr>
<tr>
<td>le</td>
<td>&lt;=</td>
</tr>
<tr>
<td>gt</td>
<td>&gt;</td>
</tr>
<tr>
<td>ge</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>
Node (Order) Comparisons

These operators expect each of their operands to be a single node. If not, an error is raised.

The operator `is` tests whether two expressions return the same node.

The operators `«` and `»` test whether one node precedes (`«`) or succeeds (`»`) another, in document order.
Built-in Functions

- XQuery also offers a set of built-in functions and operators
- We focus only on the most common ones here
- SQL users will be familiar with the \( \text{min}() \), \( \text{max}() \), \( \text{count}() \), \( \text{sum}() \), and \( \text{avg}() \) functions
- Other familiar functions include
  - Numeric functions like \( \text{round}() \), \( \text{floor}() \), and \( \text{ceiling}() \)
  - String functions like \( \text{concat}() \), \( \text{string-length}() \), \( \text{substring}() \), \( \text{upper-case}() \), \( \text{lower-case}() \)
  - Cast functions for the various atomic types
User-Defined Functions and Library Modules

- When a query becomes large and complex, it becomes easier to understand if it is split up into functions.
- A function is declared in the XQuery prolog.
- Functions can be put into library modules, which can be imported by any query.
- Every module in XQuery is either a main module (which contains a query body) or a library module (which has no query body).
- We will not cover the details of user-defined functions or library modules.

P. Atzeni (heavily from Peter Wood)
Positional Variables

- The `for` clause supports positional variables using `at`
- This identifies the position of a given item in the sequence generated by an expression
- The following query returns the titles of books with an attribute that numbers the books:

```xquery
for $t at $i in doc("books.xml")//title
return
    <title pos=' { $i } '>
    { string($t) }
    </title>
```
Positional Variables (2)

- The output of the previous query is as follows:

  `<title pos="1">TCP/IP Illustrated</title>
  <title pos="2">Advanced Programming in ...</title>
  <title pos="3">Data on the Web</title>
  <title pos="4">The Economics of Technology ...</title>`
Combining Data Sources

- A query may bind multiple variables in a `for` clause to combine data from different expressions.
- Suppose we have a file named `reviews.xml` that contains book reviews:

```xml
<reviews>
  <entry>
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>
      A very good discussion of semi-structured databases ...
    </review>
  </entry>
  ...
</reviews>
```
A FLWOR expression can bind one variable to the bibliography data and another to the review data.

In the following query we join data from the two files:

```
for $t in doc("books.xml")//title,
    $e in doc("reviews.xml")//entry
where $t = $e/title
return
  <review>
    { $t, $e/review }
  </review>
```
Combining Data Sources (3)

This returns the following answer:

```xml
<review>
  <title>TCP/IP Illustrated</title>
  <review>
    One of the best books on TCP/IP.
  </review>
</review>
<review>
  <title>Advanced Programming in the ...</title>
  <review>
    A clear and detailed discussion of ...
  </review>
</review>
...
Eliminating Duplicates

- Data (or intermediate query results) often contain duplicate values
- Consider a query returning the last names of authors:
  
  \[
  \text{doc("books.xml")//author/last}
  \]
Eliminating Duplicates

- Data (or intermediate query results) often contain duplicate values.
- Consider a query returning the last names of authors:

  \[
  \text{doc("books.xml")//author/last}
  \]

- This returns one of the authors twice:

  <last>Stevens</last>
  <last>Stevens</last>
  <last>Abiteboul</last>
  <last>Buneman</last>
  <last>Suciu</last>
The `distinct-values()` function is used to remove duplicate values.

It extracts values from a sequence of nodes and creates a sequence of unique values.

Example:

```
distinct-values(doc("books.xml")//author/last)
```

which outputs

Stevens Abiteboul Buneman Suciu
Inverting Hierarchies

- XQuery can be used to do general transformations
- In the `books.xml` file, books are sorted by title
- If we want to group books by publisher, we have to “pull up” the publisher element (i.e., invert the hierarchy of the document)
- The next slide shows a query to do this
Inverting Hierarchies — Example Query

```xml
<listings> {
  for $p in
    distinct-values(doc("books.xml")//publisher)
  order by $p
  return
  <result>
    <publisher>{ $p }</publisher>
    { for $b in doc("books.xml")//book
      where $b/publisher = $p
      order by $b/title
      return $b/title
    }
  </result>
} 
</listings>
```
Inverting Hierarchies — Query Result

<listings>
  <result>
    <publisher>Addison-Wesley</publisher>
    <title>Advanced Programming ...</title>
    <title>TCP/IP Illustrated</title>
  </result>
  <result>
    <publisher>Kluwer Academic Publishers</publisher>
    <title>The Economics of ...</title>
  </result>
  <result>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <title>Data on the Web</title>
  </result>
</listings>
Quantifiers

Some queries need to determine whether
- at least one item in a sequence satisfies a condition
- every item in sequence satisfies a condition

This is done using quantifiers:
- \texttt{some} is an existential quantifier
- \texttt{every} is a universal quantifier
Quantifiers (2)

- The following query shows an existential quantifier
- We are looking for a book where *at least one* of the authors has the last name ‘Buneman’:

```xquery
for $b in doc("books.xml")//book
where some $a in $b/author
    satisfies ($a/last = 'Buneman')
return $b/title
```

which returns:

```xml
<title>Data on the Web</title>
```
The following query shows a universal quantifier:

We are looking for a book where *all* of the authors have the last name ‘Stevens’:

```xquery
for $b in doc("books.xml")//book
where every $a in $b/author
  satisfies ($a/last = 'Stevens')
return $b/title
```

which returns:

```xml
<title>TCP/IP Illustrated</title>
<title>Advanced Programming ...</title>
<title>The Economics of Technology ...</title>
```
Quantifiers (4)

- A universal quantifier applied to an empty sequence always yields true (there is no item violating the condition)
- An existential quantifier applied to an empty sequence always yields false (there is no item satisfying the condition)
Conditional Expressions

- XQuery’s conditional expressions (if - then - else) are used in the same way as in other languages.
- In XQuery, both the then and the else clause are required.
- The empty sequence () can be used to specify that a clause should return nothing.
- The following query returns all authors for books with up to two authors and “et al.” for any remaining authors.
Conditional Expressions — Example Query

for $b in doc("books.xml") // book
return
  <book> { $b/title } {
    for $a at $i in $b/author
    where $i <= 2
    return <author> { string($a/last), ", " ,
      string($a/first) }
  </author>
}
{ if (count($b/author) > 2)
  then <author> et al. </author>
  else ()
}
</book>
Conditional Expressions — Query Result

<book>
  <title>TCP/IP Illustrated</title>
  <author>Stevens, W.</author>
</book>
<book>
  <title>Advanced Programming in ...</title>
  <author>Stevens, W.</author>
</book>
<book>
  <title>Data on the Web</title>
  <author>Abiteboul, Serge</author>
  <author>Buneman, Peter</author>
  <author>et al.</author>
</book>
<book>
  <title>The Economics of Technology ...</title>
</book>
Summary

- XQuery was designed to be compact and compositional
- It is a powerful declarative language
- It is well-suited to XML-processing tasks like data integration and data transformation (including tasks for which XSLT might be used)
Summary

- XQuery was designed to be compact and compositional
- It is a powerful declarative language
- It is well-suited to XML-processing tasks like data integration and data transformation (including tasks for which XSLT might be used)
- But what if most of your data is stored in a relational database?