The XML Revolution

Technologies for the future Web

Anders Møller & **Michael I. Schwartzbach**

BRICS, University of Aarhus

http://www.brics.dk/~amoeller/XML/

First published: March 2000
Latest revision: October 2003

This slide collection provides an introduction and overview of **XML**, **Namespaces**, **XInclude**, **XML Base**, **XLink**, **XPointer**, **XPath**, **DTD**, **XML Schema**, **DSD**, **XSLT**, **XQuery**, **DOM**, **SAX**, and **JDOM** including selected links to more information about each topic.

About this tutorial...

This slide collection about XML and related technologies is created by

Anders Møller
http://www.brics.dk/~amoeller

and

Michael I. Schwartzbach
http://www.brics.dk/~mis

at the BRICS research center at University of Aarhus, Denmark.

The slide collection is aimed at computer scientists, software engineers, and others who want to know what this XML thing is all about. It covers both the basic XML concepts and the related technologies for document linking, describing classes of documents, stylesheet transformation, and database-like querying, from a technical but high-level point of view. Based on the essential XML-related specifications, the slides are designed with concrete motivation and technical contents in focus, for the reader who wishes to understand and actually use these technologies.

A PDF version suitable for printing and off-line browsing is available upon request.

Feedback is appreciated! Please send comments and suggestions to **amoeller@brics.dk**.

See also our tutorial **Interactive Web Services with Java** covering Web programming with Java, JSP, Servlets, and JWIG.

Contents

See also the tutorial **Interactive Web Services with Java** covering Web programming with Java, JSP, Servlets, JWIG, SOAP, WSDL, and UDDI.

Markup Languages: HTML and XML

HTML - original motivation, development, and inherent limitations:

- Hyper-Text Markup Language - the Web today
- Original motivation for HTML - some history
- Compact and human readable - alternative document formats
- From logical to physical structure - requirements from users
- Stylesheets - separating logical structure and layout
- A tiny stylesheet - a CSS example
- Different versions of HTML - a decade of development
- Different browsers - obtaining compatibility
- Syntax and validation - HTML as a formal language
- Browsers are forgiving - the real world
- Structuring general information - not everything is hypertext
- Problems with HTML - why HTML is not the solution

XML as the universal format for structuring information:

- What is XML? - the universal data format
- HTML vs. XML - the key differences
- A conceptual view of XML - XML documents as labeled trees
- A concrete view of XML - XML documents as text with markup
- Applications of XML - an XML language for every domain
- The recipe example - designing a small XML language
- From SGML to SML - a word on doc-heads and development
- SGML relics - things to avoid
- XML technologies - generic languages and tools for free
- XML activity - development and applications

Selected links:

- Basic XML tools

- Links to more information

Hyper-Text Markup Language

HTML: **H**yper-**T**ext **M**arkup **L**anguage

What is **hyper-text**?

- a document that contains **links** to other documents (and text, sound, images...)
- links may be **actuated** automatically or on request
- linked documents may **replace**, be **inlined**, or create a **new** window
- most combinations are supported by HTML

What is a **markup language**?

- a notation for writing text with markup **tags**
- the tags indicate the **structure** of the text
- tags have **names** and **attributes**
- tags may **enclose** a part of the text

The start of the HTML for this page, with **`text`**, **`tags`**, and **`attributes`**:

```
<table width="99%">
  <tr>
    <td align=left>
      <a href="../index.html">
        <img src="../home.gif" border=0>
      </a>
      <a href="../info.html">
        <img src="../info.gif" border=0>
      </a>
      <td align=right>
      <a href="index.html">
        <img src="../left.gif" border=0>
      </a>
      <a href="index.html">
        <img src="../up.gif" border=0>
      </a>
      <a href="motivation.html">
        <img src="../right.gif" border=0>
      </a>
```

```
    </td>
  </tr>
</table>
<p>
<h1>Hyper-Text Markup Language</h1>
What is <b>hyper-text</b>?
<ul>
  <li>a document that contains <b>links</b> to other
documents
      (and text, sound, images...)
  <li>links may be <b>actuated</b> automatically or on request
  <li>linked documents may <b>replace</b>, be <b>inlined</b>,
      or create a <b>new</b> window
  <li>most combinations are supported by HTML
</ul>
```

Original motivation for HTML

Exchange data on the Internet:

- documents are published by **servers**
- documents are presented by **clients** (browsers)

HTML was created by Tim Berners-Lee and Robert Caillau at CERN in 1991:

- the motivation was to keep track of experimental data

HTML describes only the **logical structure** of documents:

- browsers are free to interpret markup tags as they please
- the document even makes sense if the tags are ignored

HTML combined well-known ideas:

- hyper-text was known since 1945
- markup languages date back to 1970

Compact and human readable

Many document formats are very **bulky**:

- the author controls the precise layout
- all details, including many font tables, must be stored with the contents

In comparison, HTML is **slim**:

- the author sacrifices control for compactness
- only the actual contents and its logical structure is represented

Sizes of documents containing just the text "**`Hello World!`**":

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

Compactness is good for:

- saving **space** on your server
- lowering network **traffic**

(Don't worry about voluminous markup - specialized compression techniques are emerging.)

Furthermore, HTML documents can be **written and modified with any raw-text editor**.

From logical to physical structure

Originally, HTML tags described logical structure:

- **`h2`**: "this is a **header** at level 2"
- **`em`**: "this text should be **emphasized**"
- **`ul`**: "this is a **list** of items"

Quickly, (non-physicist) users wanted more control:

- "this header is **centered** and written in **Times-Roman** in size **28pt**"
- "this text is **italicized**"
- "these list items are indented **7mm** and use **pink elephants** for bullets"

The early hack for commercial pages was to make everything a huge image:

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

The HTML developers responded with more and more physical layout tags.

Stylesheets

Cascading Style Sheets (CSS):

- specify **physical properties** (layout) of HTML tags
- are (usually) written in **separate** files
- can be **shared** for many HTML documents

There are many advantages:

- **logical** and **physical** properties may be **separated**
- document groups can have **consistent** looks
- the look can easily be changed

A CSS stylesheet works by:

- allowing more than 50 **properties** to be defined for each kind of tag;
- the definitions for a tag may depend on its **context**
- undefined properties are **inherited** from enclosing tags
- normal HTML corresponds to **default** values of properties

Using stylesheets, *all* tags become logical - however, CSS stylesheets only address superficial properties of documents.

A Touch of Style offers good advice on using stylesheets.

A tiny stylesheet

A CSS stylesheet is a collection of **selectors** and **properties**:

```
B {color:red;}
B B {color:blue;}
B.foo {color:green;}
B B.foo {color:yellow;}
B.bar {color:maroon;}
```

In the HTML document, the most **specific** properties are chosen, so:

```
<b class=foo>Hey!</b>
<b>Wow!!
   <b>Amazing!!!</b>
   <b class=foo>Impressive!!!!</b>
   <b class=bar>k00l!!!!!</b>
   <i>Fantastic!!!!!!</i>
</b>
```

gives the result:

Hey! **Wow!!** **Amazing!!!** **Impressive!!!!** **k00l!!!!!**
Fantastic!!!!!!

When properly used, the **physical layout** (a CSS file) is separated from the **logical structure** and the actual contents (a HTML file).

The default layout in a browser corresponds to a default stylesheet.

Different versions of HTML

HTML has been developed extensively over the years:

1992

HTML is first defined

1993

HTML+ (some physical layout, fill-out forms, tables, math)

1994

HTML 2.0 (standard for core features)
HTML 3.0 (an extension of HTML+ submitted as a draft standard)

1995

Netscape-specific non-standard HTML appears

1996

Competing Netscape and Explorer versions of HTML
HTML 3.2 (standard based on current practices)

1997

HTML 4.0 (separates structure and presentation with stylesheets)

1999

HTML 4.01 (slight modifications only)

2000

XHTML 1.0 (XML version of HTML 4.01)

2001

XHTML 1.1 (modularization to allow different subsets)

2002

XHTML 2.0 (simplifying and generalizing several tags)

Different browsers

According to browser statistics the current usage pattern includes:

- 124 versions of Microsoft Internet Explorer (88.0% market share)
- 42 versions of Netscape Navigator (9.2% market share)
- 23 versions of Opera (1.2% market share)
- 204 others (1.6% market share)

The many versions arise from:

- different **releases**
- different **platforms**

Well-written Web pages work well on all reasonably new browsers.

- "*this page is optimized for XYZ*" = "*I only tested this page on XYZ*"
- "*this page is best viewed in 1024x768*" = "*my screen has resolution 1024x768*"
- of course, advanced functionality (JavaScript, XSLT, ...) may require the newest browser releases

Syntax and validation

HTML 4.01 has a precise and formal syntax definition.

- every HTML document should satisfy this definition
- this can be automatically validated
- valid documents get an official seal of approval: 

- invalid documents get a list of error messages

Validating a **concrete page** is easy - ensuring that the **output of a program** is always valid HTML is much more difficult!

Browsers are forgiving

Most HTML documents are in fact *not* valid:

- authors are careless
- documents are "validated" by viewing them in a browser
- autogenerated HTML is often invalid

Even so, most HTML pages look fine:

- the browsers do their best
- no syntax errors are ever reported

```
<h2>Lousy HTML</h1>
<li><a>This is not very</b> good.
<li><i>In fact, it is quite <g>bad</g></em>
</ul>
But the browser does <a naem=suck>something.
```

Lousy HTML

- This is not very good.
- *In fact, it is quite bad But the browser does something.*

This is problematic:

- it promotes bad HTML
- different browsers do different "clever" things
- **it is very hard to use invalid documents for other things than browsing, e.g. for automatic processing by other tools!**

A different approach is HTML Tidy, which attempts to correct errors automatically in HTML documents.

Structuring general information

Consider the following recipe collection published in HTML:

```
<h1>Rhubarb Cobbler</h1>
<h2>Maggie.Herrick@bbs.mhv.net</h2>
<h3>Wed, 14 Jun 95</h3>

Rhubarb Cobbler made with bananas as the main sweetener.
It was delicious.  Basicly it was

  <table>
  <tr><td> 2 1/2 cups <td> diced rhubarb (blanched with boiling water, drain)
  <tr><td> 2 tablespoons <td> sugar
  <tr><td> 2 <td> fairly ripe bananas sliced 1/4" round
  <tr><td> 1/4 teaspoon <td> cinnamon
  <tr><td> dash of <td> nutmeg
  </table>

Combine all and use as cobbler, pie, or crisp.

Related recipes: <a href="#GardenQuiche">Garden Quiche</a>
```

There are many problems with this approach to using HTML:

- the semantics is encoded into **text formatting tags**
- there is no means of checking that a recipe is **encoded correctly**
- it is difficult to change the **layout** of recipes (CSS is not enough)

It would be much better to invent a special "recipe markup language"...

Problems with HTML

- The language is by design **hardwired to describe hypertext**:
  - there is a fixed collection of tags with a fixed semantics
  - but **much information just is not hypertext!**

- **Syntax and semantics is mixed together**:
  - the structuring of data dictates its presentation in browsers
  - stylesheets only provide a weak solution
  - different views are not supported

- The standards have been undermined:
  - most HTML documents are invalid
  - the browsers define sloppy ad-hoc standards

What is XML?

XML: e**X**tensible **M**arkup **L**anguage

XML is a **framework** for defining markup languages:

- there is **no fixed collection of markup tags** - we may define our own tags, tailored for our kind of information
- each XML language is targeted at its own application domain, but the languages will share many features
- there is a common set of **generic tools** for processing documents

XML is *not* a replacement for HTML:

- HTML should ideally be just another XML language
- in fact, XHTML is just that
- XHTML is a (very popular) XML language for hypertext markup

XML is designed to:

- separate **syntax** from **semantics** to provide a common framework for structuring information (browser rendering semantics is completely defined by stylesheets);
- allow **tailor-made markup** for any imaginable application domain
- support **internationalization** (Unicode) and **platform independence**
- be the future of structured information, including **databases**

HTML vs. XML

Consider the HTML recipe collection again:

```
<h1>Rhubarb Cobbler</h1>
<h2>Maggie.Herrick@bbs.mhv.net</h2>
<h3>Wed, 14 Jun 95</h3>

Rhubarb Cobbler made with bananas as the main sweetener.
It was delicious.  Basicly it was

  <table>
  <tr><td> 2 1/2 cups <td> diced rhubarb
  <tr><td> 2 tablespoons <td> sugar
  <tr><td> 2 <td> fairly ripe bananas
  <tr><td> 1/4 teaspoon <td> cinnamon
  <tr><td> dash of <td> nutmeg
  </table>

Combine all and use as cobbler, pie, or crisp.

Related recipes: <a href="#GardenQuiche">Garden Quiche</a>
```

With XML, we can instead define our own "recipe markup language" where the markup tags directly correspond to concepts in the world of recipes:

```
<recipe id="117" category="dessert">
  <title>Rhubarb Cobbler</title>
  <author><email>Maggie.Herrick@bbs.mhv.net</email></author>
  <date>Wed, 14 Jun 95</date>

  <description>
    Rhubarb Cobbler made with bananas as the main sweetener.
    It was delicious.
  </description>

  <ingredients>
    <item><amount>2 1/2 cups</amount><type>diced rhubarb</type></item>
    <item><amount>2 tablespoons</amount><type>sugar</type></item>
    <item><amount>2</amount><type>fairly ripe bananas</type></item>
    <item><amount>1/4 teaspoon</amount><type>cinnamon</type></item>
    <item><amount>dash of</amount><type>nutmeg</type></item>
  </ingredients>

  <preparation>
```

```
    Combine all and use as cobbler, pie, or crisp.
  </preparation>

  <related url="#GardenQuiche">Garden Quiche</related>
</recipe>
```

This example illustrates:

- the markup tags are chosen purely for **logical structure**
- this is just one choice of markup detail level
- we need to define which XML documents we regard as "recipe collections"
- we need a stylesheet to define browser presentation semantics
- we need to express queries in a general way

Later:

- **XML Schema** will later be used to define our class of recipe documents
- **XSLT** will be used to transform the XML document into XHTML (or HTML), including automatic construction of index, references, etc.
- **XLink**, **XPointer**, and **XPath** could be used to create cross-references
- **XQuery** will be used to express queries

A conceptual view of XML

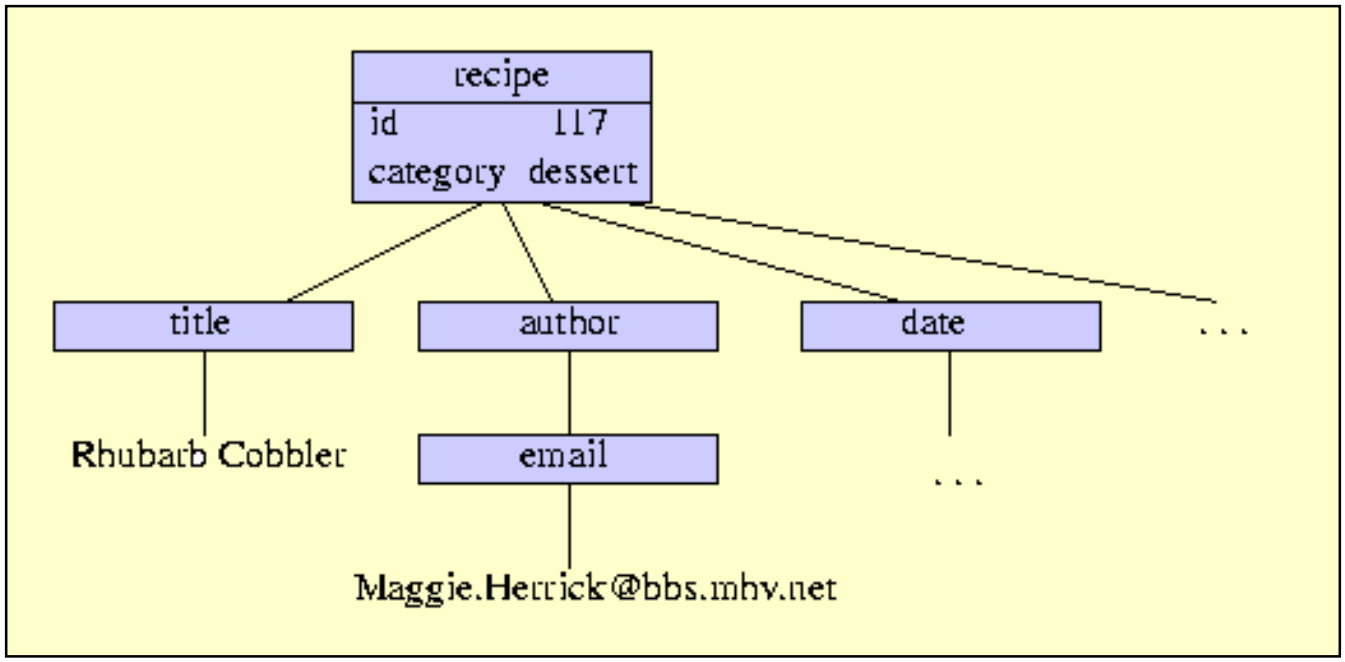
An XML document is an **ordered, labeled tree**:

- **character data** leaf nodes contain the actual data (text strings)
  - usually, character data nodes must be **non-empty** and **non-adjacent to other character data nodes**

- **elements** nodes, are each labeled with
  - a name (often called the element *type*), and
  - a set of **attributes**, each consisting of a name and a value,

  and these nodes can have child nodes

A tree view of the XML recipe collection:



The tree structure of a document can be examined in the Explorer and Mozilla browsers.

In addition, XML trees may contain other kinds of leaf nodes:

- **processing instructions** - annotations for various processors
- **comments** - as in programming languages
- **document type declaration** - described later...

Unfortunately, XML is not as simple as it could be, and there is still no agreement on XML tree terminology :-(

A concrete view of XML

An XML document is a (Unicode) text with **markup tags** and other meta-information.

Markup tags denote elements:

```
  ...<foo attr="val" ...>...</foo>...
     |    |                | |
     |    |                |  a matching element end tag
     |    |                the contents of the element
     |    an attribute with name attr and value val, values enclosed by ' or "
     an element start tag with name foo
```

There is a short-hand notation for empty elements: `...`
`<foo attr="val".../>...`

An XML document must be **well-formed**:

- start and end tags must match
- element tags must be properly nested
- + some more subtle syntactical requirements

Note: XML is *case sensitive*!

Special characters can be escaped using Unicode *character references*:

- `&#60;` and `<` both yield `<`
- `&#38;` and `&` both yield `&`

CDATA Sections are an alternative to escaping many characters:

- `<![CDATA[<greeting>Hello, world!</greeting>]]>`

The strange syntax is a legacy from SGML...

White-space (blanks, newlines, etc.) is used both for indentation and actual contents. (**`xml:space`** attribute provides some control.)

Other meta-information:

`<?target data...?>`
: an instruction for a processor, **`target`** identifies the processor for which it is directed, **`data`** is a string containing the instruction

`<!-- comment -->`
: a comment, will be ignored by all processors

`<!DOCTYPE ...>`
: document type declaration (described later...)

Applications of XML

There are already hundreds of serious applications of XML.

XHTML

W3C's XMLization of HTML 4.0. Example XHTML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head><title>Hello world!</title></head>
  <body><p>foobar</p></body>
</html>
```

CML

Chemical Markup Language. Example CML document snippet:

```
<molecule id="METHANOL">
  <atomArray>
    <stringArray builtin="elementType">C O H H H H</stringArray>
    <floatArray builtin="x3" units="pm">
       -0.748 0.558 -1.293 -1.263 -0.699 0.716
    </floatArray>
  </atomArray>
</molecule>
```

WML

Wireless Markup Language for WAP services:

```
<?xml version="1.0"?>
<wml>
   <card id="Card1" title="Wap-UK.
com">
      <p>
         Hello World
      </p>
   </card>
</wml>
```

ThML

Theological Markup Language:

```
<h3 class="s05" id="One.2.p0.2">Having a Humble Opinion of Self</h3>
<p class="First" id="One.2.p0.3">EVERY man naturally desires knowledge
 <note place="foot" id="One.2.p0.4">
  <p class="Footnote" id="One.2.p0.5"><added id="One.2.p0.6">
    <name id="One.2.p0.7">Aristotle</name>, Metaphysics, i. 1.
  </added></p>
 </note>;
 but what good is knowledge without fear of God? Indeed a humble
 rustic who serves God is better than a proud intellectual who
 neglects his soul to study the course of the stars.
 <added id="One.2.p0.8"><note place="foot" id="One.2.p0.9">
   <p class="Footnote" id="One.2.p0.10">
    Augustine, Confessions V. 4.
   </p>
 </note></added>
</p>
```

There is a long list of many other XML applications.

The recipe example

Consider again recipes, such as in this example (raw text file).

We design an XML version of a recipe collection:

- recipes consist of **ingredients**, steps for **preparation**, possibly some **comments**, and a specification of its **nutrition**
- an ingredient can be **simple** or **composite**
- a simple ingredient has a **name**, an **amount** (possibly unspecified), an a **unit** (unless amount is dimensionless)
- a composite ingredient is **recursively** a recipe

This example (formatted XML file) contains five recipes. Abbreviated version:

```
<?xml version="1.0" encoding="UTF-8"?>
<collection>
  <description>
     Some recipes used for the XML tutorial.
  </description>
  <recipe>
    <title>Beef Parmesan with Garlic Angel Hair Pasta</title>
    <ingredient name="beef cube steak" amount="1.5" unit="pound"/>
    ...
    <preparation>
      <step>
        Preheat oven to 350 degrees F (175 degrees C).
      </step>
      ...
    </preparation>
    <comment>
      Make the meat ahead of time, and refrigerate over night, the acid in the
      tomato sauce will tenderize the meat even more. If you do this, save the
      mozzarella till the last minute.
    </comment>
    <nutrition calories="1167" fat="23" carbohydrates="45" protein="32"/>
  </recipe>
  ...
</collection>
```

XML documents (usually) begin with an **XML declaration** (**`<?xml ...?>`**).

From SGML to SML

- DocHeads vs. Simpletons, a process of simplification

SGML (Standard Generalized Markup Language)

- ISO standard, 1985
- huge amount of "document archive" applications in government, military, industry, academia, ...
- a successfull well-known application: HTML is designed as a simple application of SGML.

|
v

XML

- W3C Recommendation 1998
- a simple subset of SGML, targeted for Web applications
- now **de facto standard**

|
v

MinML (Minimal XML, previously known as SML - Simple Markup Language)

- Web community discussions and collaborations, started 1999
- simplifies the XML spec: no DTDs, processing instructions, or comments, UTF-8 and UTF-16 only, considerations on element attributes, white-space,...

Canonical XML

- W3C Recommendation, March 2001
- intended as simplification of general XML documents, *not as a simplified XML spec*
- "canonical" representation
- removes document type declarations, imposes ordering on attributes, UTF-8 encoding, etc.

Occam's razor: "*one should not increase, beyond what is necessary, the number of entities required to explain anything*"

SGML relics

- only a fool does not fear "external general parsed entities"

As an unfortunate heritage from SGML, the header of an XML document may contain a **document type declaration**:

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
  <!ATTLIST greeting style (big|small) "small">
  <!ENTITY hi "Hello">
]>
<greeting> &hi; world! </greeting>
```

This part can contain:

- DTD (Document Type Definition) information:
  - element type declarations (**`ELEMENT`**)
  - attribute-list declarations (**`ATTLIST`**)

  (described later...)
- entity declarations (**`ENTITY`**) - a simple macro mechanism
- notation declarations (**`NOTATION`**) - data format specifications

Avoid all these features whenever possible!

Unfortunately, they cannot always be ignored - all XML processors (even non-validating ones) are required to:

- normalize attribute values (prune white-space etc.)
- handle internal entity references (e.g. expand **`&hi;`** in **`greeting`**)
- insert default attribute values (e.g. insert **`style="small"`** in **`greeting`**)

according to the document type declaration, if a such is present.

XML technologies

XML is:

- hot ($$$)
- the standard for representation of Web information
- by itself, just a notation for hierarchically structured text

But a notation for tree structures is not enough:

- the real force of XML is **generic languages and tools**!
- by building on XML, you get a massive infrastructure for free

The XML vision offers:

- **common extensions to the core XML specification**
  a namespace mechanism, document inclusion, etc.
- **schemas**
  grammars to define classes of documents
- **linking between documents**
  a generalization of HTML anchors and links
- **addressing parts of read-only documents**
  flexible and robust pointers into documents
- **transformation**
  conversion from one document class to another
- **querying**
  extraction of information, generalizing relational databases

To "use XML":

1. define your XML language (use e.g. XML Schema to define its syntax)
2. exploit the generic XML tools (e.g. XSLT and XQuery processors), the generic protocols, and the generic programming frameworks (e.g. DOM or SAX) to build application tools

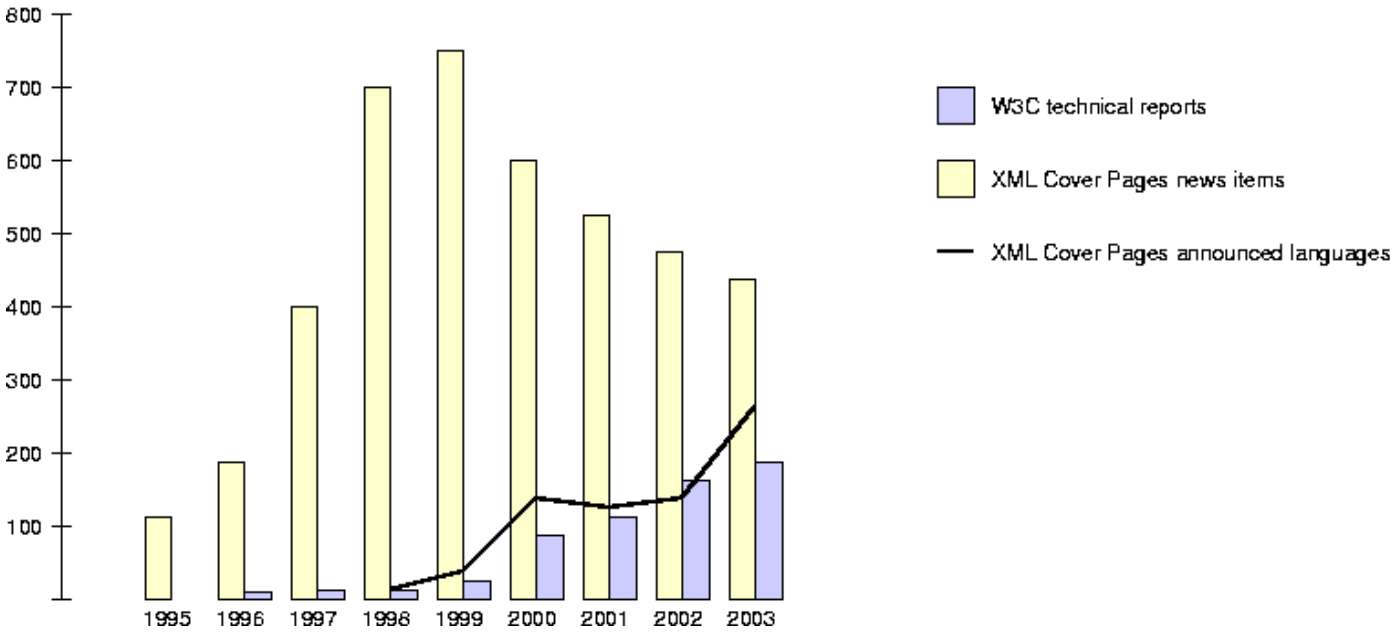
These technologies are described in the following sections.

Other related technologies (not covered here):

- **XML Information Set**
  attempt to define common terminology for XML document concepts ("information set"=tree, "information item"=node, ...)
- **XML-Signature**
  digital signatures of Web resources
- **XML Encryption**
  encryption of Web resources
- **XML Fragment Interchange**
  for dealing with fragments of XML documents
- **XML Protocol** and **SOAP** **(Simple Object Access Protocol)**
  information exchange protocol
- **XForms**
  a common sublanguage for input forms (with XHTML forms as a special case)
- **RDF** **(Resource Description Framework)**
  a framework for *metadata* (statements about properties and relationships)

Activity

Development and application activity in the XML community:



(Rough unscientific measure based on XML Cover Pages news items and W3C reports.)

As the development of core XML technologies is stabilizing, their use is taking off.

The core XML technologies are now mostly Recommendations, and W3C focuses on building higher-level layers.

Basic XML tools

Parsers

- **XML4J / Xerces** (www.alphaworks.ibm.com/tech/xml4j)
  From alphaWorks, in Java, supports DOM and SAX
- **Expat** (expat.sourceforge.net)
  Written in C (ported to other languages), used by LIBWWW, Apache, Netscape, ...
- **libxml2** (xmlsoft.org)
  Written in C (wrappers for many other languages)
- + 1000 others...

Editors

- **Xeena** (www.alphaWorks.ibm.com/tech/xeena)
  From alphaWorks, in Java, with tree-view syntax directed editing
- **XMLSpy** (www.xmlspy.com)
  Popular, but not free :-(
- + 1000 others...

Servers and Browsers

- **Apache XML** (xml.apache.org)
  built in Xerces XML parser, Xalan XSLT processor, ...
- **Netscape Navigator 6** and **Internet Explorer 5**
  XML parsing and validation, rendering with XSL and CSS, script access via DOM, ...
- **Amaya** (www.w3.org/Amaya)
  W3C's editor/browser

More info: www.garshol.priv.no/download/xmltools and www.xmlsoftware.com have comprehensive lists of XML tools.

Links to more information

www.w3.org/TR/REC-xml.html

the XML 1.0 specification

www.w3.org/TR/xml11

XML 1.1 (Candidate Recommendation), minor changes to reflect Unicode revisions

www.w3.org/XML

W3C's XML homepage

www.xml.com

XML information by O'Reilly: articles, software, tutorials

www.oasis-open.org/cover

The XML Cover Pages: comprehensive online reference

www.xmlhack.com

<?xmlhack?>: concise XML news

news:comp.text.xml

XML newsgroup

www.ucc.ie/xml

XML FAQ

www.xml.com/axml/testaxml.htm

the Annotated XML Specification, by Tim Bray

metalab.unc.edu/xml

Cafe con Leche XML News and Resources

inf2.pira.co.uk/top011a.htm

El.pub's markup language section

wdvl.internet.com/Authoring/Languages/XML

links to XML information

www.w3schools.com/xml

XML School: an XML tutorial

www.garshol.priv.no/download/xmltools

a list of free XML tools

Namespaces, XInclude, and XML Base

- common extensions to the core XML specification

Namespaces - mixing XML languages

- Mixing XML languages - name clashes
- Qualifying names - solving the problem with URIs
- Namespace declarations - declarations and prefixes
- Specification controversies - some unclear details in the spec

XInclude - combining XML documents

- Combining XML documents - reuse and modularity
- An XInclude example - an example
- XInclude details - more details

XML Base - resolving relative URIs

- XML Base - another common XML extension

Selected links:

- Links to more information

Mixing XML languages

Consider an XML language WidgetML which uses XHTML as a sublanguage for help messages:

```
<widget type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <head>
      <title>Description of gadget</title>
    </head>
    <body>
      <h1>Gadget</h1>
      A gadget contains a big gizmo
    </body>
  </info>
</widget>
```

A problem: the meaning of **`head`** and **`big`** depends on the context!

This complicates things for processors and might even cause ambiguities.

The root of the problem is: one common **name space**.

Qualifying names

Simple solution: **qualify names with URIs** (Universal Resource Identifiers)

```
<{http://www.w3.org/TR/xhtml1}head>
  \                          / \  /
   -------------------------   --
       qualifying URI        local
name
```

Do not be confused by the use of URIs for namespaces:

- they are **not supposed to point to anything**
- it is simply the cheapest way of getting **unique names**
- we **rely on existing organizations** that control domain names

(just like Java package names!)

This is the idea - the actual solution is less verbose but slightly more complicated...

Namespace declarations

Namespaces are **declared** by special attributes and associated **prefixes**:

```
<... xmlns:foo="http://www.w3.org/TR/xhtml1">
  ...
  <foo:head>...</foo:head>
  ...
</...>
```

`xmlns:`*`prefix`*`="`*`URI`*`"` declares a namespace with a prefix and a URI:

- the scope of declaration is **lexical**, the element containing the declaration and all descendants can be overridden by nested declaration
- both element and attribute names can be qualified with namespaces
- the name of the prefix is irrelevant - applications should use only the URI

For backward compatibility and simplicity, unprefixed element names are assigned a **default** namespace:

- declaration: **`xmlns="`*`URI`*`"`**
- default value: "" (means: treat as unqualified name)
- does *not* affect unprefixed **attribute** names (they belong to the containing elements)

WidgetML with namespaces:

```
<widget xmlns="http://www.widget.org"
        xmlns:xhtml="http://www.w3.org/TR/xhtml1"
        type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <xhtml:head>
      <xhtml:title>Description of gadget</xhtml:title>
    </xhtml:head>
    <xhtml:body>
      <xhtml:h1>Gadget</xhtml:h1>
      A gadget contains a big gizmo
    </xhtml:body>
  </info>
</widget>
```

Specification controversies

How should a **relative** URI be interpreted?

- relative to the base URI?
- relative to the document URI?
- just as a string?

This innocent question spawned a controversy that resulted in leaving the matter undefined (by deprecating such namespaces).

Other controversies:

- does the choice of prefix matter, or is **`<a:widget xmlns:a="www.widget.org"/>`** the same as **`<b:widget xmlns:b="www.widget.org"/>`**?

- is **`<a:widget size="big"/>`** the same as **`<a:widget a:size="big"/>`**?

Unfortunately, according to the spec, the choice of prefix *may* matter, and an unqualified attribute generally does *not* just inherit the namespace from its element.

Combining XML documents

To enhance **reuse** and **modularity**, a technique for **constructing new XML documents from existing ones** is desirable.

XInclude provides a **simple inclusion mechanism**.

Why yet another specification?

- many XML documents and languages can benefit from modularity
- as for the namespace solution, a **generic approach** can be implemented in **generic tools**

Application conformance: Think of XML as if Namespaces, XInclude, and XML Base were parts of the basic XML specification. (Caveat: the latter two are not widely implemented yet.)

An XInclude example

A document containing:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="somewhere.xml"/>
</foo>
```

where **somewhere.xml** contains:

```
<bar>...</
bar>
```

is equivalent to:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
  <bar>...</bar>
</foo>
```

- **http://www.w3.org/2001/XInclude** is the official **XInclude namespace**
- the **include** element name in that namespace is an **inclusion directive**
- right after parsing and before other processing, an **XInclude processor** performs the inclusion (tree substitution)
- the original and the resulting document should be considered **equivalent**
- it is an error to have cyclic includes

XInclude details

How is the included resource denoted?

- with **XPointer** (described later...) - an extension of URLs that can address document nodes, node sets, or character data ranges

Other issues:

- with **`parse="text"`** and **`encoding="..."`** attributes, a resource can be transformed into a **character data** node before inclusion
- XInclude processors may need to create namespace declaration attributes to ensure equivalence

Many XInclude processors support only whole-document URIs, not full XPointer.

XML Base

A URI identifies a resource:

- **`http://somewhere/somefile.xml`** is an **absolute** URI
- **`somefile.xml`** is a **relative** URI

Inspired by the **`<base href="...">`** mechanism in HTML, **XML Base** provides a **uniform way of resolving relative URIs**.

In the following example:

```
<... xml:base="http://www.daimi.au.dk/">
  <... href="~mis/mn/index.html" .../>
</...>
```

the value of **`href`** attribute can be interpreted as the absolute URI **`http://www.daimi.au.dk/~mis/mn/index.html`**.

- the **`xml`** namespace prefix is hardwired by the Namespace specification
- **`xml:base`** has **lexical scope** (as namespace declarations)
- the URI used to access the document is used as **default** URI base

Examples of applications:

- XLink (requires XML Base support)
- XHTML (will use XML Base)
- Namespaces (does not conform to XML Base, but it ought to...)
- your future XML language

Future XML parsers will support Namespaces, XInclude, and XML Base.

Links to more information

Namespaces:

www.w3.org/TR/REC-xml-names

the W3C XML Namespace Recommendation

www.jclark.com/xml/xmlns.htm

an explanation of the recommendation by James Clark

www.xml.com/xml/pub/1999/01/namespaces.html

an XML.com article on Namespaces

www.rpbourret.com/xml/NamespacesFAQ.htm

comprehensive Namespace FAQ

XInclude:

www.w3.org/TR/xinclude

XInclude, W3C Candidate Recommendation

www.ibiblio.org/xml/XInclude

a Java XInclude processor

XML Base:

www.w3.org/TR/xmlbase

the W3C XML Base Recommendation

DTD, XML Schema, and DSD

- defining language syntax with schemas

Overview:

- Schemas and schema languages - defining the syntax of your own XML language
- Choosing a schema language - lots of alternatives

DTD - the insufficient schema language defined in the XML 1.0 spec:

- DTD - Document Type Definition - an overview
- Example DTD - the recipe example
- Problems with DTD - top 15 reasons for not using DTD

XML Schema - W3C's recent proposal:

- Design requirements - how to design a schema language in W3C
- XML Schema - the design
- A small example - the business-card example
- Overview - the central constructs and ideas
- Constructing complex types - requirements for attribute and content presence
- Constructing simple types - requirements for attribute values and character data
- Local definitions - inlined declarations, anonymous types, and overloading
- Type derivation and substitution groups - the type system
- Annotations - self-documentation
- Schema inclusion and redefinition - modularity and reuse
- Namespaces - constraining the use of namespaces
- Attribute and element defaults - side-effects of validation
- Identity constraints - uniqueness and keys
- A larger example - the recipe example

- Problems with XML Schema - 15 reasons why we haven't seen the last schema language

DSD - the next generation of schema languages:

- Document Structure Description 2.0 - central aspects
- Example - the recipe example
- Rules - describing elements
- Boolean expressions - expressing element properties
- Regular expressions - describing attribute values and chardata
- Normalization - whitespace, upper/lower-case, defaults
- Inclusion and extension - modular descriptions

Selected links:

- Links to more information

Schemas and schema languages

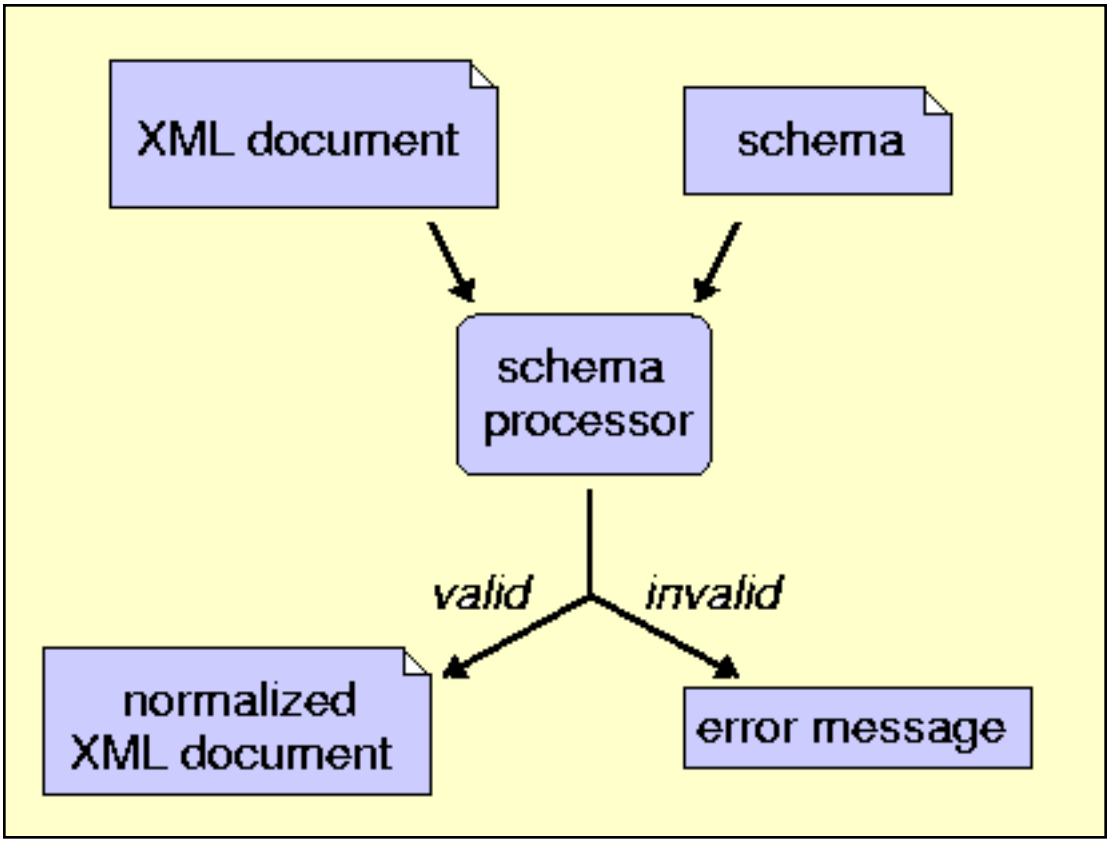
A **schema** is a definition of the syntax of an XML-based language (i.e., it defines a class of XML documents).

A **schema language** is a formal language for expressing schemas.

Schema **processing**: Given an XML document and a schema, a schema processor

- checks for **validity**, i.e. that the document conforms to the schema requirements
- if the document is valid, a **normalized** version is output: default attributes and elements are inserted, parsing information may be added, etc.

The document being validated is called an **instance document** or **application document**.



Why bother formalizing the syntax with a schema?

- a formal definition can provide a **precise but human-readable** reference
- schema processing can be done with **existing implementations**
- **your own tools for your languages can benefit**: by piping input documents through a schema processor, you can assume that the input is valid and defaults have been inserted

Schemas are similar to **grammars** for programming languages, however, context-free grammars are not expressive enough for XML.

The term "schema" comes from the database community.

Choosing a schema language

There have been many schema language proposals.

W3C proposals:

- **DTD**
- XML-Data, January 1998
- DCD (Document Content Description), July 1998
- DDML (Document Definition Markup Language), January 1999
- SOX (Schema for Object-oriented XML), July 1999
- **XML Schema**

Non-W3C proposals:

- Assertion Grammars by Dave Raggett
- Schematron by Rick Jellife
- TREX (Tree Regular Expressions for XML) by James Clark
- Examplotron by Eric van der Vlist
- RELAX by Makoto Murara / RELAX NG by Murata and Clark
- **DSD** (Document Structure Description)

Unlike for many other XML technologies, it has proved difficult to reach a consensus - probably because:

- it is an inherently difficult problem
- people have different needs from a schema language
- the official (W3C) proposals are not very good

however, most schema languages have many similarities.

We shall look at W3C's **DTD** and **XML Schema** proposals and at the **DSD** proposal developed by BRICS and AT&T.

DTD - Document Type Definition

Recall from earlier that XML 1.0 contains a built-in schema language:
Document Type Definition

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

- **`<!ELEMENT element-name content-model>`**
  associates a *content model* to all elements of the given name

  content models:

  - **`EMPTY`**: no content is allowed
  - **`ANY`**: any content is allowed
  - **`(#PCDATA|element-name|...)`**: "mixed content", arbitrary sequence of character data and listed elements
  - *deterministic regular expression over element names*: sequence of elements matching the expression
    - choice: **`(...|...|...)`**
    - sequence: **`(...,...,...)`**
    - optional: **`...?`**
    - zero or more: **`...*`**
    - one or more: **`...+`**

- **`<!ATTLIST element-name attr-name attr-type attr-default ...>`**
  declares which attributes are allowed or required in which elements

  attribute types:

  - **`CDATA`**: any value is allowed (the default)
  - **`(value|...)`**: enumeration of allowed values
  - **`ID`**, **`IDREF`**, **`IDREFS`**: ID attribute values must be unique (contain "element identity"), IDREF attribute values must match some ID

(reference to an element)

- **ENTITY**, **ENTITIES**, **NMTOKEN**, **NMTOKENS**, **NOTATION**: just forget these... (consider them deprecated)

attribute defaults:

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

This is a simple subset of SGML DTD.

Validity can be checked by a simple top-down traversal of the XML document (followed by a check of IDREF requirements).

Example DTD

A DTD for our recipe collections, **recipes.dtd**:

```
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation,comment?,nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step*)>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition protein CDATA #REQUIRED
                    carbohydrates CDATA #REQUIRED
                    fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>
```

By inserting:

```
<!DOCTYPE collection SYSTEM "recipes.dtd">
```

in the headers of recipe collection documents, we state that they are intended to conform to **recipes.dtd**.

Alternatively, the DTD can be given locally with **<!DOCTYPE collection [...]>**.

This grammatical description has some obvious shortcomings:

- we cannot express that, e.g. **protein**, must contain a non-negative number

- **`unit`** should only be allowed when **`amount`** is present
- the **`comment`** element should be allowed to appear anywhere
- nested **`ingredient`** elements should only be allowed when **`amount`** is absent

Problems with DTD

Top 15 reasons for avoiding DTD:

1. **not itself using XML syntax** (the SGML heritage can be very unintuitive + if using XML, DTDs could potentially themselves be syntax checked with a "meta DTD")

2. **mixed into the XML 1.0 spec** (would be much less confusing if specified separately + even non-validating processors must look at the DTD)

3. **no constraints on character data** (if character data is allowed, *any* character data is allowed)

4. **too simple attribute value models** (enumerations are clearly insufficient)

5. **cannot mix character data and regexp content models** (and the content models are generally hard to use for complex requirements)

6. **no support for Namespaces** (of course, XML 1.0 was defined before Namespaces)

7. **very limited support for modularity and reuse** (the entity mechanism is too low-level)

8. **no support for schema evolution, extension, or inheritance of declarations** (difficult to write, maintain, and read large DTDs, and to define families of related schemas)

9. **limited white-space control** (**`xml:space`** is rarely used)

10. **no embedded, structured self-documentation** (**`<!-- comments -->`** are not enough)

11. **content and attribute declarations cannot depend on attributes or**

element context (*many* XML languages use that, but their DTDs have to "allow too much")

12. **too simple ID attribute mechanism** (no points-to requirements, uniqueness scope, etc.)

13. **only defaults for attributes, not for elements** (but that would often be convenient)

14. **cannot specify "any element" or "any attribute"** (useful for partial specifications and during schema development)

15. **defaults cannot be specified separate from the declarations** (would be convenient to have defaults in separate modules)

Design requirements

Quotes from the W3C Note "XML Schema Requirements" (Feb. 1999):

Design principles:

The XML schema language shall be

1. **more expressive than XML DTDs**
2. **expressed in XML**
3. **self-describing**
4. usable by a wide variety of applications that employ XML
5. straightforwardly usable on the Internet
6. optimized for interoperability
7. **simple** enough to be implemented with modest design and runtime resources
8. **coordinated with relevant W3C specs**

The XML schema language specification shall

1. be prepared **quickly**
2. be precise, **concise**, human-readable, and illustrated with examples

Structural requirements:

The XML schema language must define

1. mechanisms for **constraining document structure** (namespaces, elements, attributes) and **content** (datatypes, entities, notations)
2. mechanisms to enable **inheritance** for element, attribute, and datatype definitions
3. mechanism for **URI reference to standard semantic understanding** of a construct
4. mechanism for **embedded documentation**
5. mechanism for application-specific constraints and descriptions

6. mechanisms for addressing the **evolution** of schemata
7. mechanisms to enable integration of structural schemas with **primitive data types**

Unfortunately, their own XML Schema Recommendation does *not* fulfil all requirements (self-describing, simple, concise, human-readable, ...)

XML Schema

W3C Recommendation, May 2001.

Consists of two parts:

1. *Structures*
2. *Datatypes*

Main features:

- **XML syntax** (there is a Schema for Schemas)
- uses and supports **Namespaces**
- **object-oriented-like type system** for declarations (with inheritance, subsumption, abstract types, and finals)
- **global** (=top-level) and **local** (=inlined) type definitions
- **modularization** (schema inclusion and redefinitions)
- structured **self-documentation**
- **cardinality constraints** for sub-elements
- nil values (missing content)
- attribute and element **defaults**
- any-element, any-attribute
- **uniqueness** constraints and ID/IDREF attribute scope
- **regular expressions** for specifying valid chardata and attribute values
- lots of **built-in data types** for chardata and attribute values

Yes, it is big and complicated! (Part 1 of the spec alone is more than 150 dense pages...)

A small example

Assume we want to create an XML-based language for **business cards**.

An example document **`john_doe.xml`**:

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
  <logo url="widget.gif"/>
</card>
```

To describe the syntax of our new language, we write a schema **`business_card.xsd`**:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="title" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
  <element name="logo" type="b:logo_type"/>

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>
```

```
  <complexType name="logo_type">
    <attribute name="url" type="anyURI"/>
  </complexType>

</schema>
```

The XML Schema language is recognized by the namespace **http://www.w3.org/2001/XMLSchema**.

A document may refer to a schema with the **schemaLocation** (or the **noNamespaceSchemaLocation**) attribute:

```
<card xmlns="http://businesscard.org"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://businesscard.org
                          business_card.xsd">
  ...
</card>
```

By inserting this, the author claims that the document is intended to be valid with respect to the schema.

Overview of XML Schema

The most central top-level constructs:

- a (global) **element declaration** associates an element name with a *type*
- a ***complex* type definition** defines requirements for attributes, sub-elements, and character data in elements of that type
  - **attribute declarations**: describe which attributes that may or must appear
  - **element references**: describe which sub-elements that may or must appear, how many, and in which order
- a ***simple* type definition** defines a set of strings to be used as attribute values or character data

(Two types or two elements cannot be defined with the same name, but an element declaration and a type definition may use the same name.)

An *element* in an XML document is **valid** according to a given schema if the associated element type rules are satisfied.

If all elements are valid, the whole *document* is called **valid**. (Unlike DTD, there is no way to require a specific root element.)

Constructing complex types

A **`complexType`** can contain:

- **attribute declarations**:
  - **`<attribute name="..." type=".." use=".."/>`**
    where **`type`** refers to a simple type definition and **`use`** is either **`optional`** (the default), **`required`**, or **`prohibited`** (used with type derivations)
  - **`<anyAttribute ... />`**
- one of the following **content model** kinds:
  - *empty content* (the default)
  - *simple content*: **`<simpleContent> ... </simpleContent>`** (only character data is allowed)
  - *regexp content*: a (restricted) combination of
    - **`<sequence> ... </sequence>`**
    - **`<choice> ... </choice>`**
    - **`<all> ... </all>`**

    containing element references of the form
    - **`<element ref="..." minOccurs=".." maxOccurs=".."/>`**
      where **`ref`** refers to an element definition, and **`minOccurs`** and **`maxOccurs`** constrain the number of occurences (default: 1)
    - **`<any .../>`**

    (if **`complexType`** has the attribute **`mixed="true"`**, arbitrary character data is also allowed)

Example:

```
<complexType name="order_type" mixed="true">

  <attribute name="id" type="unsignedInt" use="required"/>

  <choice>
    <element ref="n:address"/>
    <sequence>
      <element ref="n:email" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="n:phone"/>
    </sequence>
  </choice>

</complexType>
```

- grouping of definitions:
    - **attribute groups**: groups of attribute declarations can be defined with **`<attributeGroup name="..."> ... </...>`** and used with **`<attributeGroup use="..."/>`**.
    - **element groups**: similarly, groups of regexp content model descriptions can be defined and used with the **`group`** construct.
- **`anyType`** is a built-in type that allows anything
- ***global* attribute declarations** apply to all elements

Constructing simple types

Simple types can be:

- **primitive** (hardwired meaning)

- **derived** from existing simple types:
  - by a **`list`**: white-space separated sequence of other simple types
  - by a **`union`**: union of other simple types
  - by a **`restriction`**:
    - **`length`**, **`minLength`**, **`maxLength`** (list lengths)
    - **`enumeration`** (intersection with list of values)
    - **`pattern`** (intersection with **Perl-like regexp**)
    - **`whiteSpace`** (preserve/replace/collapse white-space)
    - **`minInclusive`**, **`maxInclusive`** (bounds on numbers)

A lot of often-used simple types (all the primitive and some derived) are **predefined**:

- **`integer`**
- **`date`**
- **`anyURI`**
- **`unsignedLong`**
- **`language`**
- ...

In addition, **white space normalization** of attribute values and chardata can be controlled at the simple-type definitions.

Example definition of a derived simple type:

```
<simpleType name="may_date">
  <restriction base="date">
    <pattern value="\d{4}-05-\d{2}"/>
  </restriction>
</simpleType>
```

All this is specified in Part 2 of the spec.

Local definitions

Instead of writing all element declarations and type definitions at top-level (globally), they may be **inlined** (locally):

Example:

```
<element name="card" type="b:card_type"/>
<element name="name" type="string"/>

<complexType name="card_type">
  <sequence>
    <element ref="b:name"/>
    <element ref="b:title"/>
    <element ref="b:email" maxOccurs="unbounded"/>
    <element ref="b:phone" minOccurs="0"/>
    <element ref="b:background" minOccurs="0"/>
  </sequence>
</complexType>
```

means the same as

```
<element name="card">
  <complexType>
    <sequence>
      <element name="name" type="string"/>
      <element ref="b:title"/>
      <element ref="b:email" maxOccurs="unbounded"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:background" minOccurs="0"/>
    </sequence>
  </complexType>
</element>
```

(where the complex type **`card_type`** and the description of **`name`** have been inlined)

except that:

- inlined *type definitions* are **anonymous**, so they cannot be referred to for reuse
- inlined *element declarations* can be **overloaded**, i.e. they need not have unique names

- otherwise, it is mostly a matter of authoring style.

Type derivation and substitution groups

XML Schema contains a complicated **type system**.

Types can be derived from other types:

- Derivation by **extension**:

```
<complexType name="car">
  <complexContent>
    <extension base="n:vehicle">
      <element name="wheel" minOccurs="3" maxOccurs="4"/>
    </extension>
  </complexContent>
</complexType>
```

creates a **car** type from a **vehicle** type by extending it with 3 or 4 **wheel** sub-elements. (Components from the base type are inherited as in an implicit **sequence**.)

- Derivation by **restriction**:

```
<complexType name="small_car">
  <complexContent>
    <restriction base="n:car">
      ...
      <element name="wheel" maxOccurs="3"/>
    </extension>
  </complexContent>
</complexType>
```

creates a **small_car** type from the **car** type by restricting it to 3 **wheel** sub-elements. (All components from the base type must be repeated, as indicated by "**...**".)

Subsumption:

Assume that we declare an element:

```
<element name="myVehicle" type="n:vehicle"/>
```

meaning that **myVehicle** elements are valid if they match the **vehicle** type.

Since **car** is derived from **vehicle**, **myVehicle** elements are also valid if they match **car**
- provided that we add **xsi:type="n:car"** to the elements (where **xsi** refers to

`http://www.w3.org/2001/XMLSchema-instance`) in the instance document.

Substitution groups - an alternative to type derivation:

If we declare another element as:

```
<element name="myCar" type="n:car" substitutionGroup="n:vehicle"/>
```

then we may always use **`myCar`** elements whenever **`myVehicle`** elements are required (without using **`xsi:type`**).

This is independent of the extension/restriction inheritance hierarchy! - **`car`** is not required to be declared as a sub-type of **`vehicle`**.

Controlling derivation and subsumption:

In addition to all this,

- derivation of types can be forbidden (e.g. using **`final="restriction"`**)
- subsumption can be forbidden (e.g. using **`block="#all"`**)
- use of elements and types can be forbidden (using **`abstract="true"`**)

Annotations

Schemas can be annotated with human or machine readable **documentation** and other information:

```
<xsd:element name="author">
  <xsd:annotation>
    <xsd:documentation xmlns:xhtml="http://www.w3.org/1999/xhtml">
       the author of the recipe,
       see <xhtml:a href="authors.xml">this list</xhtml:a> of authors
    </xsd:documentation>
    <xsd:appinfo xmlns:fp="http://foodprocessor.org">
       <fp:process type="117"/>
    </xsd:appinfo>
  </xsd:annotation>
  ...
</xsd:element>
```

Note that annotations can be **structured**, as opposed to simple **<!-- ... -->** XML comments.

Schema inclusion and redefinition

No less than 3 mechanisms are available:

- **`<include schemaLocation="..."/>`** - compose with schema having *same* target namespace
- **`<import namespace="..." schemaLocation="..."/>`** - compose with schema having *different* target namespace
- **`<redefine schemaLocation="..."> ... </redefine>`** - compose with schema having same target namespace, *allowing redefinitions*

XInclude is not used...

Example:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <import namespace="http://www.w3.org/1999/xhtml" schemaLocation="xhtml.xsd"/>

  <redefine schemaLocation="phone.xsd">
    <element name="phone"/>
      ...
    </element>
  </redefine>

  ...
</schema>
```

Here, a schema for XHTML is imported together with **`phone.xsd`** (which is assumed to contain a description of phone numbers) and its description of **phone** is redefined.

Namespaces

When defining a new XML-based language, we usually want to assign it a namespace.

XML Schema

- *uses namespaces itself* - to distinguish schema instructions from the language we are describing
- *supports namespace assigning* - by associating a **target namespace** to the language we are describing

Example:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  ...

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      ...
    </sequence>
  </complexType>

  ...
</schema>
```

- the **default** namespace is that of XML Schema (such that e.g. **complexType** is considered an XML Schema element)
- the **target** namespace is our business card namespace
- the **b** prefix also denotes our business card namespace (such that we can refer to target language constructs from within the schema)

Unfortunately, XML Schema has a rather unconventional use of namespaces:

- prefixes in attribute values (e.g. **`ref="b:name"`**) - the namespace spec does not tell how to resolve this
- a notion of "unqualified locals" (which is even a default) - allowing prefixes to be omitted from locally declared elements in instance documents
  (for a reasonable semantics, always use **`elementFormDefault="qualified"`** if using local definitions)

This precludes the use of standard namespace-compliant XML parsers for reading XML Schema documents :-(

Attribute and element defaults

Side-effect of validation: insertion of **default** values

Each attribute and element declarations can contain a **`default="..."`** attribute.

- **attribute defaults:** are inserted (before validation) if the attribute is absent (in elements of the type containing the declaration)
- **element defaults:** are inserted **as character data** in **empty** elements (of the type of the declaration)

(For some strange design reason, element defaults cannot contain markup.)

Example:

With a schema containing:

```
<element name="widget" default="no content explicitly provided">
  <complexType mixed="true">
    <attribute name="size" default="big"/>
  </complexType>
</element>
```

a schema processor will validate and transform:

```
<widget/>
```

into:

```
<widget size="big">no content explicitly provided</widget>
```

Identity constraints

XPath can be used to specify **uniqueness** requirements.

Example:

```
<unique name="uniqueness-requirement-87">
  <selector xpath=".//personlist"/>
  <field xpath="person/@ssn"/>
</unique>
```

occurring in an element declaration, means that: within each **`personlist`**, every **`ssn`** attribute of a **`person`** element must have a unique value.

Similarly, we can define **keys** (with **`key`**) and **references** (with **`keyref`**) which generalizes the ID/IDREF mechanism from DTD in a straightforward way.

Only a simple subset of XPath is allowed:

- only the **child axis** and the **attribute axis**
- only **node set expressions**

A larger example

A XML Schema description of our recipe collections, **recipes.xsd**:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://recipes.org"
        targetNamespace="http://recipes.org"
        elementFormDefault="qualified">

  <element name="collection">
    <complexType>
      <sequence>
        <element name="description" type="r:anycontent"/>
        <element ref="r:recipe" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>

  <complexType name="anycontent" mixed="true">
    <sequence>
      <any minOccurs="0" maxOccurs="unbounded"
           processContents="skip" namespace="##other"/>
    </sequence>
  </complexType>

  <element name="recipe">
    <complexType>
      <sequence>
        <element name="title" type="string"/>
        <element ref="r:ingredient" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="r:preparation"/>
        <element name="comment" minOccurs="0" type="string"/>
        <element name="nutrition">
          <complexType>
            <attribute name="protein" type="r:nonNegativeDecimal" use="required"/>
            <attribute name="carbohydrates" type="r:nonNegativeDecimal"
use="required"/>
            <attribute name="fat" type="r:nonNegativeDecimal" use="required"/>
            <attribute name="calories" type="r:nonNegativeDecimal" use="required"/>
            <attribute name="alcohol" type="r:nonNegativeDecimal" use="optional"/>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>

  <element name="preparation">
    <complexType>
      <sequence>
        <element name="step" type="string" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
 </element>
```

```
  <element name="ingredient">
    <complexType>
      <sequence>
        <element ref="r:ingredient" minOccurs="0" maxOccurs="unbounded"/>
        <element ref="r:preparation" minOccurs="0"/>
      </sequence>
      <attribute name="name" use="required"/>
      <attribute name="amount" use="optional">
        <simpleType>
          <union>
            <simpleType>
              <restriction base="string">
                <enumeration value="*"/>
              </restriction>
            </simpleType>
            <simpleType>
              <restriction base="r:nonNegativeDecimal"/>
            </simpleType>
          </union>
        </simpleType>
      </attribute>
      <attribute name="unit" use="optional"/>
    </complexType>
  </element>

  <simpleType name="nonNegativeDecimal">
    <restriction base="decimal">
      <minInclusive value="0"/>
    </restriction>
  </simpleType>

</schema>
```

Note that:

- we need to set **`elementFormDefault="qualified"`** to use the standard Namespace semantics
- the **`nonNegativeDecimal`** and **`anycontent`** definitions were not possible with DTD
- we choose to use a mix of global and local definitions
- as with the DTD version, we still cannot express that:
  - **`unit`** should only be allowed when **`amount`** is present
  - the **`comment`** element should be allowed to appear anywhere
  - nested **`ingredient`** elements should only be allowed when **`amount`** is absent

By inserting the following reference into the root element:

```
<collection xmlns="http://recipes.org"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://recipes.org recipes.xsd">
  ...
</collection>
```

in our recipe collection **`recipes.xml`**, we state that the document is intended to be valid according to **`recipes.xsd`**.

Problems with XML Schema

The general problem:

- it is generally **too complicated** (the spec is several hundred pages in a very technical language), so it is hard to use by non-experts - but many non-experts need schemas to describe data formats

  also, the complicated design necessitates an incomprehensible specification style (example from Part 1, Section 3.3.1: "*{value constraint} establishes a default or fixed value for an element. If default is specified, and if the element being ·validated· is empty, then the canonical form of the supplied constraint value becomes the [schema normalized value] of the ·validated· element in the post-schema-validation infoset. If fixed is specified, then the element's content must either be empty, in which case fixed behaves as default, or its value must match the supplied constraint value.*", or from Section 3.3.4: "*If the item cannot be ·strictly assessed·, because neither clause 1.1 nor clause 1.2 above are satisfied, [Definition:] an element information item's schema validity may be laxly assessed if its ·context-determined declaration· is not skip by ·validating· with respect to the ·ur-type definition· as per Element Locally Valid (Type) (§3.3.4).*")

Practical limitations of expressibility:

- cannot require specific **root element** (so extra information is required to validate even the simplest documents)

- when describing **mixed content**, the character data cannot be constrained in any way (not even a set of valid characters can be specified)

- **content and attribute declarations cannot depend on attributes or element context** (this was also listed as a central problem of DTD)
  - a typical example that cannot be expressed (actually from the XML Schema spec which is packed with examples): *"'default' and 'fixed' may not both be present, and [...] if 'ref' is present, then all*

of <simpleType>, 'form' and 'type' must be absent"
    - a solution to this would also eliminate the need for "nil values"
    - local definitions allows some degree of context sensitivity, but make it impossible to combine this context sensitivity with e.g. reuse of definitions

- it is **not 100% self-describing** (as a trivial example, see the previous point), even though that was an initial design requirement

- **defaults** cannot be specified separate from the declarations (this makes it hard to make families of schemas that only differ in the default values)

- **element defaults** can only be character data (not containing markup)

Technical problems:

- although it technically is namespace conformant, it does not seem to follow the **namespace** spirit (e.g. "unqualified locals")

The major source of complexity:

- the notion of "**type**" adds an extra layer of confusing complexity:
    - in instance documents, we have "elements" which have "element names"
    - in schemas, elements are described by "element definitions" which associate "element names" with "type names",
    - type definitions associate "type names" with "element descriptions" which describe the elements in the instance documents

  (and to cause further confusion, the XML 1.0 spec uses the term "element type" for the name of an element)

- **`xsi:type`** attributes are required in instance documents when derived types are being used in place of base types

- substitution groups and local declarations (with non-unique names) make it **difficult to look up the description of a given element**

Non-minimalistic design:

- the mechanisms for type derivation and substitution groups seem to be different attempts to solve the same problems

- incorporation of **XPath** to express uniqueness and keys (neither uniqueness or keys are fundamental concepts for schemas, so dragging in a big language as XPath seems overkill)

- the set of **built-in data types** is **not minimalistic** (a minimalistic set + some data type libraries would lower the learning burden)

- the use of **Perl-style regular expressions** violates the principle of using XML syntax to describe XML syntax

For other comments about the design of XML Schema, see for instance www.xml.com/pub/a/2000/07/05/specs/lastword.html, www.ibiblio.org/xql/tally.html , and www.xml.com/pub/a/2002/07/31/wxstypes.html.

Document Structure Description 2.0

- a successor to DSD 1.0, a schema language developed in cooperation by BRICS and AT&T Labs Research.

DSD is designed to:

- contain **few** and **simple** language constructs (based on familiar concepts, such as boolean logic and regular expressions)
- be **easy to understand**, also by non-XML-experts
- have **more expressive power** than other schema languages for most practical purposes

The central ideas in DSD 2.0:

- a schema consists of a list of **rules**
- for every element in the instance document, all rules are processed
- rules can conditionally depend on the name, attributes, and context of the current element
- rules contain *declare* and *require* sections
- **declare** sections specify which content (sub-elements and character data) and attributes that are allowed for the current element
- **require** sections specify extra restrictions on content and attributes
- attribute values and element contents are described by **regular expressions**
- rule conditions and extra restrictions are described by **boolean logic**

Main benefits, compared to XML Schema:

- the complete language specification is only 15 pages (excluding examples)
- no notion of type, rules are directly tied to element names (and no subtyping, substitution groups, or local definitions)
- rules can be hierarchical by depending on attribute values and element context

- DSD is 100% self-describing (so there is a *complete* "DSD for DSDs")
- lots of non-essential features are removed or reduced to more basic and general constructs

DSD 1.0 was announced in November 1999. The spec for DSD 2.0 is now available!

Example

A DSD 2.0 description of our recipe collections:

```
<dsd xmlns="http://www.brics.dk/DSD/2.0"
     xmlns:r="http://recipes.org"
     root="r:collection">

  <if><element name="r:collection"/>
    <declare><contents>
      <element name="r:description"/>
      <repeat><element name="r:recipe"/></repeat>
    </contents></declare>
  </if>

  <if><element name="r:description"/>
    <rule ref="r:ANYCONTENT"/>
  </if>

  <if><element name="r:recipe"/>
    <declare><contents>
      <sequence>
        <element name="r:title"/>
        <repeat><element name="r:ingredient"/></repeat>
        <element name="r:preparation"/>
        <element name="r:nutrition"/>
      </sequence>
      <optional><element name="r:comment"/></optional>
    </contents></declare>
  </if>

  <if><element name="r:ingredient"/>
    <declare>
      <required><attribute name="name"/></required>
      <attribute name="amount">
        <union>
          <string value="*"/>
          <stringtype ref="r:NUMBER"/>
        </union>
      </attribute>
      <attribute name="unit"/>
    </declare>
    <if><not><attribute name="amount"/></not>
      <require><not><attribute name="unit"/></not></require>
      <declare><contents>
        <repeat min="1"><element name="r:ingredient"/></repeat>
        <element name="r:preparation"/>
      </contents></declare>
    </if>
  </if>

  <if><element name="r:preparation"/>
```

```
    <declare><contents>
      <repeat><element name="r:step"/></repeat>
    </contents></declare>
  </if>

  <if>
    <or>
      <element name="r:step"/>
      <element name="r:comment"/>
      <element name="r:title"/>
    </or>
    <declare><contents>
      <string/>
    </contents></declare>
  </if>

  <if><element name="r:nutrition"/>
    <declare>
      <required>
        <attribute name="protein"><stringtype ref="r:NUMBER"/></attribute>
        <attribute name="carbohydrates"><stringtype ref="r:NUMBER"/></attribute>
        <attribute name="fat"><stringtype ref="r:NUMBER"/></attribute>
        <attribute name="calories"><stringtype ref="r:NUMBER"/></attribute>
      </required>
      <attribute name="alcohol"><stringtype ref="r:NUMBER"/></attribute>
    </declare>
  </if>

  <stringtype id="r:DIGITS">
    <repeat min="1">
      <char min="0" max="9"/>
    </repeat>
  </stringtype>

  <stringtype id="r:NUMBER">
    <sequence>
      <stringtype ref="r:DIGITS"/>
      <optional>
        <sequence>
          <string value="."/>
          <stringtype ref="r:DIGITS"/>
        </sequence>
      </optional>
    </sequence>
  </stringtype>

  <rule id="r:ANYCONTENT">
    <declare><contents>
      <repeat><union><element/><string/></union></repeat>
    </contents></declare>
  </rule>

</dsd>
```

Notice in particular:

- the **hierarchical rule** in the description of **`ingredient`**
- the modular definitions of two stringtypes and a rule
- the simple use of namespaces (following the Namespaces spec but also using prefixes in attribute values)
- it is intuitive and human-readable (if you are used to looking at XML documents :-)

This DSD is **more precise** than the DTD and the XML Schema descriptions:

- **`unit`** is only allowed when **`amount`** is present
- the **`comment`** element is allowed to appear anywhere
- nested **`ingredient`** elements are only allowed when **`amount`** is absent

One can check that this is indeed a DSD by validating it with the meta-DSD.

Rules

- a closer look at the central DSD 2.0 construct

Example:

```
<if><element name="r:preparation"/>
  <declare><contents>
    <repeat><element name="r:step"/></repeat>
  </contents></declare>
</if>
```

Rules can be:

- **`if`** rules, conditional rules guarded by boolean expressions over element properties (permitting context sensitive constraints!)
- **`declare`** rules, declaring which attributes and contents an element may have
- **`require`** rules, containing boolean expressions over element properties that are required to hold

(In addition, there are **`unique`** and **`pointer`** rules for generalized IDs/ IDREFs.)

Rules can be **defined** (given an ID for reference) to support modularity, as e.g. **`ANYCONTENT`** in the full example.

A document is validated through a top-down process, checking each element in turn. For each element,

- all its attributes and contents must be declared by some applicable **`declare`** rule
- all applicable additional requirements (**`require`** rules) must be satisfied

The root element must match the **`root`** attribute of the **`schema`** element.

Boolean expressions

Boolean logic for expressing properties of elements:

- element names and contents
- attribute presence and values
- context properties: **`parent`**, **`ancestor`**, **`child`**, and **`descendant`**

combined with **`and`**, **`or`**, **`not`**, **`impl`**, etc.

Example:

```
<or>
  <and>
    <attribute name="a"><stringtype ref="b"/></attribute>
    <ancestor><element name="p:c"><attribute name="d"/></element></ancestor>
  </and>
  <contents>
    <sequence><element name="p:e"/><element name="p:f"/></sequence>
  </contents>
</or>
```

means: "either the current element has an **a** attribute with a **b** value and also a **c** ancestor element with a **d** attribute, or - if only looking at **e** and **`f`** elements - its contents consist of one **e** element followed by one **`f`** element (where the elements use the namespace selected by the **p** prefix)."

Boolean expressions are used both as **conditions** in conditional constraints (**`if`**) and as **requirements** (in **`require`**).

As with the other syntactic categories, boolean expressions can be **defined** for modularity.

Regular expressions

Both **attribute values** and element **content models** are described by **regular expressions**.

Regular expressions can be built from:

- constant strings, character sets (over the Unicode alphabet)
- boolean expressions (which describe elements)
- sequencing, union, repetition
- complement, intersection (!)
- ...

Example:

```
...
  <union>
    <string value="*"/>
    <stringtype ref="r:NUMBER"/>
  </union>
...

<stringtype id="r:DIGITS">
  <repeat min="1">
    <char min="0" max="9"/>
  </repeat>
</stringtype>

<stringtype id="r:NUMBER">
  <sequence>
    <stringtype ref="r:DIGITS"/>
    <optional>
      <sequence>
        <string value="."/>
        <stringtype ref="r:DIGITS"/>
      </sequence>
    </optional>
  </sequence>
```

```
</stringtype>
```

Libraries of common expressions can be made with the import feature described later...

If multiple regular expressions are declared for the contents of an element, they are implicitly combined by projection:

```
<contents>
  <sequence>
    <element name="p:a"/>
    <element name="p:b"/>
  </sequence>
</contents>

<contents>
  <sequence>
    <element name="p:a"/>
    <optional><element name="p:c"/></optional>
  </sequence>
</contents>
```

Here, an **a** element must be followed by a **b** element and optionally by a **c** element, but the **b** and **c** elements can appear in any order.

Normalization

Normalization declarations define how schema processors will modify **whitespace** and **character cases** and insert **default attributes and contents**.

Attribute declarations and contents declarations may contain

```
<normalize whitespace="..." case="..."/>
```

where

- **`whitespace`** has value
  - **`preserve`** (the default)
  - **`compress`** (compress consecutive whitespace characters)
  - **`trim`** (as **`compress`**, but also remove leading and trailing whitespace)
- **`case`** has value
  - **`preserve`** (the default)
  - **`upper`**
  - **`lower`**

Default attributes:

Attribute declarations may contain
`<default value="..."/>`

Default contents:

Contents declarations may contain
`<default> ... </default>` (may contain elements!)

Normalization occurs before validity checking.

Inclusion and extension

DSD descriptions can consist of **several XML documents** that can be combined using **`import`**.

This makes it easy to

- **modularize** specifications for easier readability and maintainability
- **reuse and extend** existing schemas
- create **families of related schemas**

Example:

Assuming **`basic.dsd`** contains:

```
<dsd xmlns="http://www.brics.dk/DSD/2.0"
     xmlns:p="http://www.example.org/basic">

  <if><element name="p:widget"/>
    <declare>
      <contents>
        <repeat><element name="p:thingy"/></repeat>
      </contents>
      <attribute name="type"/>
    </declare>
  </if>
  ...
</dsd>
```

then we may use this schema in another schema:

```
...
<import href="basic.dsd"/>

<if>
  <and>
    <element name="p:widget"/>
    <attribute name="type"><string value="small"/></attribute>
  </and>

  <require><contents>
    <repeat max="7"><element name="p:thingy"/></repeat>
  </contents></require>
  <declare>
    <attribute name="style"/>
  </declare>
</if>
...
```

which means that a **`widget`** with **`type="small"`** can at most have 7 **`thingy`** child elements (restricting the basic definition), but additionally, it may have a **`style`** attribute (extending the basic definition).

Links to more information

www.w3.org/TR/xmlschema-0

XML Schema Part 0: Primer (a non-normative introduction)

www.w3.org/TR/xmlschema-1

XML Schema Part 1: Structures

www.w3.org/TR/xmlschema-2

XML Schema Part 2: Datatypes

www.brics.dk/DSD

the DSD homepage

www.brics.dk/DSD/dsd2.html

Document Structure Description 2.0

www.brics.dk/DSD/dsd2

DSD 2.0 implementation in Java

www.oasis-open.org/cover/schemas.html

Robin Cover's XML schema information

www.xml.com/pub/1999/12/dtd

XML.com article on schema languages

www.xml.com/pub/a/2000/11/29/schemas/part1.html

XML.com introduction to XML Schema

www.xfront.com/BestPracticesHomepage.html

"best practices" of XML Schema

www.ibiblio.org/xml/books/bible2/chapters/ch24.html

chapter from "XML Bible" on XML Schema

www.xmlhack.com/read.php?item=1097

"W3C XML Schema still has big problems", article on <?xmlhack?>

www.alphaworks.ibm.com/tech/xmlsqc

XML Schema quality checker, from IBM alphaWorks

www.cobase.cs.ucla.edu/tech-docs/dongwon/ucla-200008.html

"Comparative Analysis of Six XML Schema Languages"

www.redrice.com/schemavalid/faq/xml-schema.html

XML Schema FAQ

xml.apache.org

Apache's Xerces parser and validator (for DTD and XML Schema)

XLink, XPointer, and XPath

- linking and addressing

Overview:

- XLink, XPointer, and XPath - three layers of languages

XLink:

- Problems with HTML links - why do we need something new?
- The XLink linking model - a generalization of HTML links
- An example - a link between two remote resources
- Linking elements - defining links
- Behavior - **`show`** and **`actuate`**
- Simple vs. Extended links - compatibility issues
- HLink vs. XLink - a new alternative?

XPointer, Part I - using XPointer in XLink:

- XPointer: Why, what, and how? - introduction
- XPointer vs. XPath - what is the difference
- XPointer fragment identifiers - the structure of an XPointer

XPath:

- Location paths - the central construct
- Location steps - expressing node-sets
  - Axes - selecting candidates
  - Node tests - initial filtration
  - Predicates - fine-grained filtration
- Expressions - a little expression language
- Core function library - the built-in functions
- Abbreviations - convenient notation

- XPath visualization - a useful tool
- XPath examples - continuing the recipe example
- XPath 2.0 - the next version

XPointer, Part II - how XPointer uses XPath:

- Context initialization - filling out the gap between XPath and XLink
- Extra XPointer features - generalizing XPath

Selected links:

- Tools
- Links to more information

XLink, XPointer, and XPath

- imagine a Web without links...

Three layers:

- **XLink**
  - a **generalization of the HTML link concept**
  - higher abstraction level (intended for **general XML** - not just hypertext)
  - more **expressive power** (multiple destinations, special behaviors, linkbases, ...)
  - uses XPointer to locate resources

- **XPointer**
  - an **extension of XPath** suited for linking
  - specifies connection between XPath expressions and URIs

- **XPath**
  - a declarative language for **locating nodes and fragments** in XML trees
  - used in both XPointer (for **addressing**), XSL (for **pattern matching**), XML Schema (for **uniqueness and scope descriptions**), and XQuery (for **selection and iteration**)

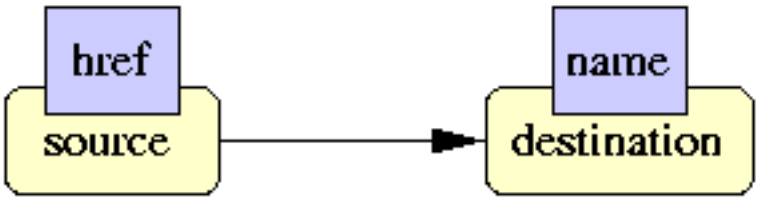
These technologies are standardized but not all widely implemented yet.

XQuery vs. XPointer/XPath? Reminiscent, but different goals:

- XQuery: SQL-like database queries
- XPointer/XPath: robust addressing into known information

Problems with HTML links

The HTML link model:



Construction of a hyperlink:

- **`<a name="here">`** is placed at the destination
- **`<a href="URL#here">`** is placed at the source

Problems when using the HTML model for general XML:

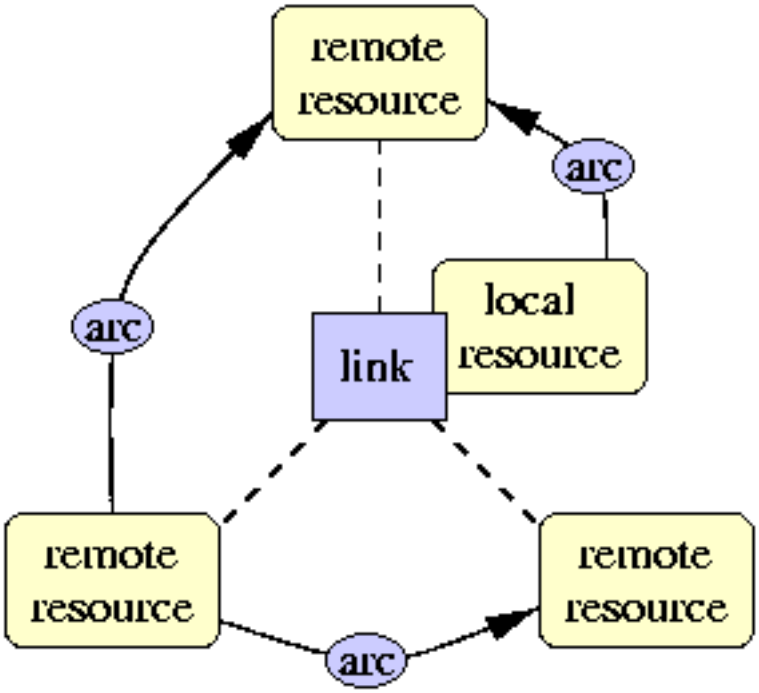
- Link recognition:
  - in HTML, links are recognized by element names (**`a`**, **`img`**, ..)
    - we want a *generic XML solution*
  - the "semantics" of a link is defined in the HTML specification
    - we want to identify abstract semantic features, e.g. *link actuation*
- Limitations:
  - an anchor must be placed at every link destination (problem with read-only documents)
    - we want to express *relative locations* (XPointer!)
  - the link definition must be at the same location as the link source (outbound)
    - we want *inbound* and *third-party* links
  - only individual nodes can be linked to
    - we want links to whole *tree fragments*
  - a link always has one source and one destination
    - we want links with *multiple sources and destinations*

The usual point: *generic solutions allow generic tools!*

The XLink linking model

Basic XLink terminology:

Link: explicit relationship between two or more resources.
Linking element: an XML element that asserts the existence and describes the characteristics of a link.
Locator: an identification of a remote resource that is participating in the link.



One **linking element** defines a set of traversable **arcs** between some **resources**.

A **local resource** comes from the linking element's own content.

Outbound: the *source* is a local resource
Inbound: the *destination* is a local resource
Third-party: none of the resources are local

Third-party links can be used to construct shared link bases for browsers.

An example

A linking element defining a third-party "extended" link involving two remote resources:

```
<mylink xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
  <myresource xlink:type="locator"
              xlink:href="students.xml#Fred" xlink:label="student"/>
  <myresource xlink:type="locator"
              xlink:href="teachers.xml#Joe" xlink:label="teacher"/>
  <myarc xlink:type="arc"
         xlink:from="student" xlink:to="teacher"/>
</mylink>
```

- the namespace **`http://www.w3.org/1999/xlink`** is used to recognize XLink information in general XML documents
  - the namespace often (but not necessarily) uses namespace prefix **`xlink`**
  - **host language**: elements and attributes *not* belonging to this namespace are ignored by XLink processors
  - all XLink information is defined in attributes (in host language elements)
- **`xlink:type="extended"`** indicates a linking element
- **`xlink:type="locator"`** locates a remote resource
- **`xlink:type="arc"`** defines traversal rules

A powerful example application of general XLinks:

> Using third-party links and a smart browser, a group of people can annotate Web pages with "post-it notes" for discussion - without having write access to the pages. They simply need to agree on a set of URIs to XLink link bases defining the annotations. The smart XLink-aware browser lets them select parts of the Web pages (as XPointer ranges), comment the parts by creating XLinks to a small XHTML documents, view each other's comments, place comments on comments, and perhaps also aid in structuring the comments.

Linking elements

- defining links

All elements with XLink information contain an **`xlink:type`** attribute.

- a general **linking element** is defined using an **`xlink:type="extended"`** attribute; this element can contain the following:
- a **local resource** is defined with **`xlink:type="resource"`**
- a **remote resource** is defined with **`xlink:type="locator"`** and with an **`xlink:href`** attribute (an XPointer expression locating the resource)
- **arcs** (traversal rules) are defined with **`xlink:type="arc"`**:
    - both **`"resource"`** and **`"locator"`** elements can have **`xlink:label`** attributes
    - an arc element has an **`xlink:from`** and an **`xlink:to`** attribute
    - the **`"arc"`** element defines a set of arcs: from each resource having the **`from`** label to each resource having the **`to`** label

(Note the confusing terminology: a *resource* is defined either by a **`"resource"`** element or by a **`"locator"`** element.)

XPointer is described later - just think of XPointer expression as URIs for now...

Behavior

- link semantics

Arcs can be annotated with abstract behavior information using the following attributes:

`xlink:show` - what happens when the link is activated?

Possible values:

`embed`

insert the presentation of the target resource (the one at the end of the arc) in place of the source resource (the one at the beginning of the arc, where traversal was initiated) (example: as images in HTML)

`new`

display the target resource some other place without affecting the presentation of the source resource (example: as **`target="_blank"`** in an HTML link)

`replace`

replace the presentation of the resource containing the source with a presentation of the destination (example: as normal HTML links)

`other`

behavior specified elsewhere

`none`

no behavior is specified

`xlink:actuate` - when is the link activated?

Possible values:

`onLoad`

traverse the link immediately when recognized (example: as HTML images)

`onRequest`

traverse when explicitly requested (example: as normal HTML links)

`other`

behavior specified elsewhere

none

no behavior is specified

Note: these notions of link behavior are rather abstract and do not make sense for all applications.

Semantic attributes: describe the meaning of link resources and arcs

`xlink:title`

provide human readable descriptions (also available as **`xlink:type="title"`** to allow markup)

`xlink:role` and **`xlink:arcrole`**

URI references to descriptions

Simple vs. Extended links

- for compatibility and simplicity

Two kinds of links:

- **extended** - the general ones we have seen so far
- **simple** - a restricted version of extended links: only for **two-ended outbound** links (enough for HTML-style links)

Convenient shorthand notation for simple links:

```
<mylink xlink:type="simple" xlink:href="..." xlink:show="..." .../>
```

is equivalent to:

```
<mylink xlink:type="extended">
  <myresource xlink:type="resource"
              xlink:role="local"/>
  <myresource xlink:type="locator"
              xlink:role="remote" xlink:href="..."/>
  <myarc xlink:type="arc"
         xlink:from="local" xlink:to="remote" xlink:show="..." .../>
</mylink>
```

Many XLink properties (e.g. **`xlink:type`** and **`xlink:show`**) can conveniently be specified as defaults in the schema definition!

When should I use XLink? Tim Berners-Lee: *only for **hypertext** linking* (Not everybody agree...)

HLink vs. XLink

Why is XHTML not using XLink?

The problem:

- we want a **general** mechanism for **identifying** links
- ...but we want full control when designing the syntax of the **host languages**

When integrating XLink in a host language, the use of the XLink namespace makes a mess.

HLink: a recent alternative to XLink

- same underlying ideas
- different **syntax**

Example HLink: Definition of the link semantics of **<a href="...">** elements in XHTML.

```
<hlink namespace="http://www.w3.org/1999/xhtml"
       element="a"
       locator="@href"
       effect="replace"
       actuate="onRequest"
       replacement="@target"/>
```

13 September 2002: W3C's HTML Working Group publishes HLink draft for intended use in XHTML 2.0

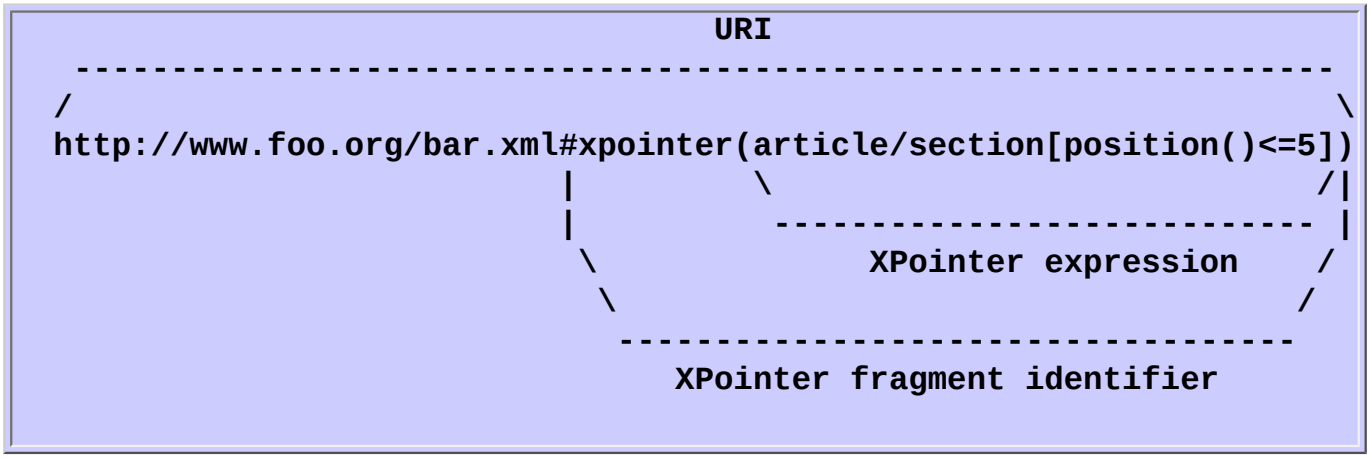
24 September 2002: W3C's Technical Architecture Group **rejects** HLink in favor of XLink for the design of XHTML 2.0

- what's next?

XPointer: Why, what, and how?

- an **extension of XPath** which is **used by XLink** to locate remote link resources
- **relative addressing**: allows links to places with no anchors
- **flexible** and **robust**: XPointer/XPath expressions often survive changes in the target document
- can point to **substrings in character data** and to whole **tree fragments**

Example of an XPointer:



(points to the first five section elements in the article root element.)

In HTML, fragment identifiers may denote anchor IDs - XPointer generalizes that.

XPointer vs. XPath

XPointer is based upon XPath:

- an XPointer expression is **basically the same** as an XPath expression
- XPath says nothing about **URIs**; XPointer specifies that connection
- an XPath expression is evaluated wrt. a **context**; XPointer specifies this context
- XPointer **adds some features** not available in XPath

XPointer fragment identifiers

An *XPointer fragment identifier* (the substring to the right of # in the URI) is either

- the value of some **ID** attribute in the document (ID attributes are specified by the schema),
- a sequence of element numbers denoting the **path** from the root to an element (e.g. **`/1/27/3`**), or
- a sequence of the form

  ```
  xpointer(...) xpointer(...) ...
  ```

  containing a list (typically of length 1) of **XPointer expressions**.

  Each expression is evaluated in turn, and the first where evaluation succeeds is used. (This allows alternative pointers to be specified thereby increasing robustness.)

Recently, the XPointer spec has been split into four (tiny) parts:

- framework
- xmlns() scheme
- element() scheme
- xpointer() scheme

Next: We will now look into **XPath** and then later describe what additional features XPointer adds to XPath...

XPath: Location paths

XPath is a declarative language for:

- **addressing** (used in **XLink/XPointer** and in **XSLT**)
- **pattern matching** (used in **XSLT** and in **XQuery**)

The central construct is the **location path**, which is a sequence of **location steps** separated by `/`, e.g.:

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

selects all **`href`** attributes in **`cite`** elements in the first 5 **`section`**s of an article document.

- a **location step** is evaluated wrt. some context resulting in a set of nodes
- a **location path** is evaluated *compositionally*, *left-to-right*, starting with some initial context
  - location paths resemble operating system *directory paths*
  - each node resulting from evaluation of one step is used as context for evaluation of the next, and the results are unioned together

A **context** consists of:

- a context **node**
- a context **position** and **size** (two integers)
- **variable bindings**, a **function library**, and a set of **namespace declarations**

Initial context: defined externally (e.g. by XPointer, XSLT, or XQuery).
Location paths can be prefixed with `/` to use the document root as initial context node!

Note: in the XPath data model, the XML document tree has a special **root node** above the root element.

There is a strong analogy to directory paths (in UNIX). As an example, the directory path **`/*/d/*.txt`** selects a set of files, and the location path **`/*/d/*[@ext="txt"]`** select a set of XML elements.

Location steps

A single **location step** has the form

axis **::** *node-test* **[** *predicate* **]**

- The **axis** selects a rough set of *candidate nodes* (e.g. the child nodes of the context node).
- The **node-test** performs an **initial filtration** of the candidates based on their
  - *types* (chardata node, processing instruction, etc.), or
  - *names* (e.g. element name).
- The **predicates** (zero or more) cause a further, potentially more complex, filtration. Only candidates for which the predicates evaluate to *true* are kept.

The candidates that survive the filtration constitute the *result*.

This structure of location steps makes implementation rather easy and **efficient**, since the complex predicates are only evaluated on relatively few nodes.

The example from before:

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

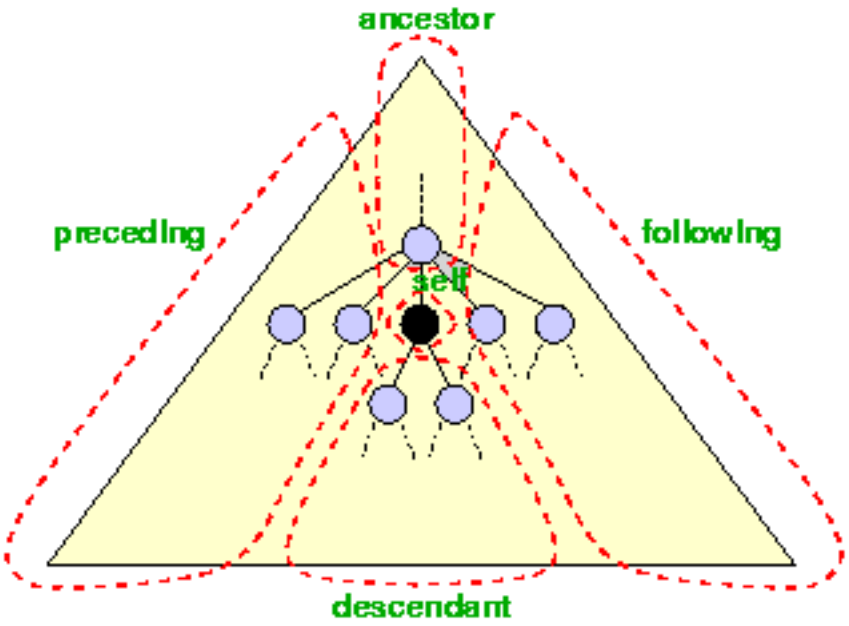
selects all **`href`** attributes in **`cite`** elements in the first 5 **`section`**s of an article document.

Axes

Available axes:

`child`	the children of the context node
`descendant`	all descendants (children, childrens children, ...)
`parent`	the parent (empty if at the root)
`ancestor`	all ancestors from the parent to the root
`following-sibling`	siblings to the right
`preceding-sibling`	siblings to the left
`following`	all following nodes in the document, excluding descendants
`preceding`	all preceding nodes in the document, excluding ancestors
`attribute`	the attributes of the context node
`namespace`	namespace declarations in the context node
`self`	the context node itself
`descendant-or-self`	the union of **`descendant`** and **`self`**
`ancestor-or-self`	the union of **`ancestor`** and **`self`**

Note that **attributes** and **namespace declarations** are considered a special kind of nodes here.



Some of these axes assume a **document ordering** of the tree nodes. The ordering is the **left-to-right preorder** traversal of the document tree - which is the same as the order in the textual representation.

The resulting sets are **ordered** intuitively, either **forward** (in document order) or **reverse** (reverse document order).
For instance, **`following`** is a forward axis, and **`ancestor`** is a reverse axis.

(Frustratingly, each technology uses a slightly different tree model...)

Node tests

Testing by node **type**:

`text()`	chardata nodes
`comment()`	comment nodes
`processing-instruction()`	processing instruction nodes
`node()`	all nodes (not including attributes and namespace declarations)

Testing by node **name**:

`name`	nodes with that name
`*`	any node

Warning: There is a **bug in the XPath 1.0 spec**! Default namespaces are required to be handled incorrectly, so, if using Namespaces together with XPath (or XSLT), all elements must have an explicit prefix.

Predicates

- expressions coerced to type *boolean*

A predicate filters a node-set by evaluating the predicate expression on each node in the set with

- that node as the **context node**,
- the size of the node-set as the **context size**, and
- the position of the node in the node-set wrt. the axis ordering as the **context position**.

Example:

```
child::section[position()<6] / descendant::cite[attribute::href="there"]
```

selects all **`cite`** elements with **`href="there"`** attributes in the first 5 **`section`**s of an article document.

(Compare with the earlier example.)

The XPath predicate language is very large, but these are the **essential** ones to know

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

An entire **location path** may be used as a predicate

- start at the current node
- the predicate is true if the location path hits some result positions
- it is false otherwise

This is very useful to **look ahead**:

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

Expressions

Available types:

- **node-set** (set of nodes)
- **boolean** (true or false)
- **number** (floating point)
- **string** (Unicode text)

An **expression** can be:

- a constant, e.g. **`"..."`**
- a variable: **`$variable`**
- a function call: **`function ( arguments )`**
- a boolean expression: **`or`**, **`and`**, **`=`**, **`!=`**, **`<`**, **`>`**, **`<=`**, **`>=`** (standard precedence, all left associative)
- a numerical expression: **`+`**, **`-`**, **`*`**, **`div`**, **`mod`**
- a node-set expression (using **location paths**!): **`|`** (set union)

Coercion may occur at function arguments and when expressions are used as predicates.

Variables and functions are evaluated using the context.

Core function library

Node-set functions:

last()	returns the context size
position()	returns the context position
count(*node-set*)	number of nodes in node-set
name(*node-set*)	string representation of first node in node-set
...	...

String functions:

string(*value*)	type cast to string
concat(*string*, *string*, ...)	string concatenation
...	...

Boolean functions:

boolean(*value*)	type cast to boolean
not(*boolean*)	boolean negation
...	...

Number functions:

number(*value*)	type cast to number
sum(*node-set*)	sum of number value of each node in node-set
...	...

- see the XPath specification for the complete list.

Abbreviations

Syntactic sugar: convenient notation for common situations

Normal syntax	***Abbreviation***
`child::`	*nothing* (so **`child`** is the default axis)
`attribute::`	`@`
`/descendant-or-self::node()/`	`//`
`self::node()`	`.` (useful because location paths starting with `/` begin evaluation at the root)
`parent::node()`	`..`

Example:

```
.//@href
```

selects all **`href`** attributes in descendants of the context node.

Furthermore, the **coercion rules** often allow compact notation, e.g.

```
foo[3]
```

refers to the third **`foo`** child element of the context node (because **`3`** is coerced to **`position()=3`**).

XPath visualization

Using Explorer 6 (or an updated version of Explorer 5) it is easy to experiment with XPath expressions.

The XPath Visualizer provides an interactive XPath evaluator that additionally visualizes the resulting node set (online installation).

This tool is implemented as an ordinary HTML page that makes heavy use of XSLT and JavaScript.

XPath examples

The following XPath expressions point to sets of nodes in the recipe collection:

"The amounts of flour being used":

```
//ingredient[@name="flour"]/
@amount
```

```
4
0.5
3
0.25
```

"The ingredients of which half a cup are used":

```
//ingredient[@amount='0.5' and @unit='cup']/
@name
```

```
grated Parmesan cheese
shredded mozzarella cheese
shortening
flour
orange juice
```

"The second step in preparing *stock* for *Cailles en Sarcophages*":

```
//ingredient[@name="stock"]/preparation/step[position()=2]/text()
```

```
When the liquid is relatively clear, add the carrots, celery, whole
onion,
bay leaf, parsley, peppercorns and salt. Reduce the heat, cover and let
simmer at least 2 hours to make a hearty stock.
```

XPath 2.0

- currently a Working Draft, developed to capture the common subset of XSLT 2.0 and XQuery 1.0

Major changes from 1.0:

- now using **XML Schema primitive types** instead of the four in 1.0
  - new type operators: **`cast`**, **`treat`**, **`assert`**, **`instance of`**
- now using **sequences** instead of node-sets
  - also allow non-node types
  - new operators: **`for`**, **`if`**, **`some`**, **`every`**, **`intersect`**, **`except`**
- many(!) new **functions**
  - regular expression match/replace/tokenize
  - date formats
  - ...

XPointer: Context initialization

An XPointer is basically an XPath expression occurring in a URI.

When evaluated, the **initial context** is defined as follows:

- the **context node** is the root node of the document
- the **context position** and **size** are both 1 (because the root has no siblings)
- the **variable bindings** are empty (variables are not used by XPointer)
- the **function library** consists of the core XPath functions + a few extra functions
- the **namespace declarations** are set as follows:
  `xmlns(myprefix=http://mynamespace.org) xpointer(...)`

Warning: several levels of **character escaping** occur when using XPointer in XML documents

- in XPointer, unbalanced parentheses must be escaped, e.g. **`^)`**
- in URIs, many characters must be escaped, e.g. **`%20`**
- in XML attribute values, quotes, ampersand, etc. must be escaped, e.g. **`<`**

Extra XPointer features

XPointer provides a more **fine-grained addressing** than XPath.

- Instead of just *nodes*, XPointers address **locations**, which can be *nodes*, *points*, or *ranges*.

- A **point** can represent the location preceding or following any *individual character* in e.g. chardata nodes.
  The special node test
  `point()`
  selects the set of points of a node.

- A **range** consists of two points in the same document, and is specified using a special **`range-to`** location step construct.

- XPointer provides some **extra functions**:

`here()`	get location of element containing current XPointer
`origin()`	get location where user initiated link traversal
`start-point(`*`location-set`*`)`	get start point of location set
`string-range(...)`	find matching substrings
`...`	

Example:

```
/descendant::text()/point()[position()=0]
```

selects the locations right before the first character of all character data nodes in the document.

Example:

```
/section[1] / range-to(/section[3])
```

selects everything from the beginning of the first **`section`** to the end of the third.

Tools

Kinds of tools supporting XLink/XPointer:

- browsers
- parsers
- link bases

but XLink is still not widely implemented.

XPath is primarily implemented as part of XSLT processors.

Links to more information

www.w3.org/TR/xlink

W3C's XLink Recommendation

www.w3.org/TR/xptr-framework/

W3C's XPointer Framework (Recommendation)

www.w3.org/TR/xptr-xmlns/

W3C's XPointer xmlns() Scheme (Recommendation)

www.w3.org/TR/xptr-element/

W3C's XPointer element() Scheme (Recommendation)

www.w3.org/TR/xptr-xpointer/

W3C's XPointer xpointer() Scheme (Working Draft)

www.w3.org/TR/xpath

W3C's XPath 1.0 Recommendation

www.w3.org/TR/xpath20/

W3C's XPath 2.0 (Working Draft)

www.stg.brown.edu/~sjd/xlinkintro.html

a brief introduction to XML linking

www.ibiblio.org/xml/books/bible2/chapters/ch19.html

a chapter from "The XML Bible" on XLink

www.ibiblio.org/xml/books/bible2/chapters/ch20.html

a chapter from "The XML Bible" on XPointer (and XPath)

XSL and XSLT

- stylesheets and document transformation

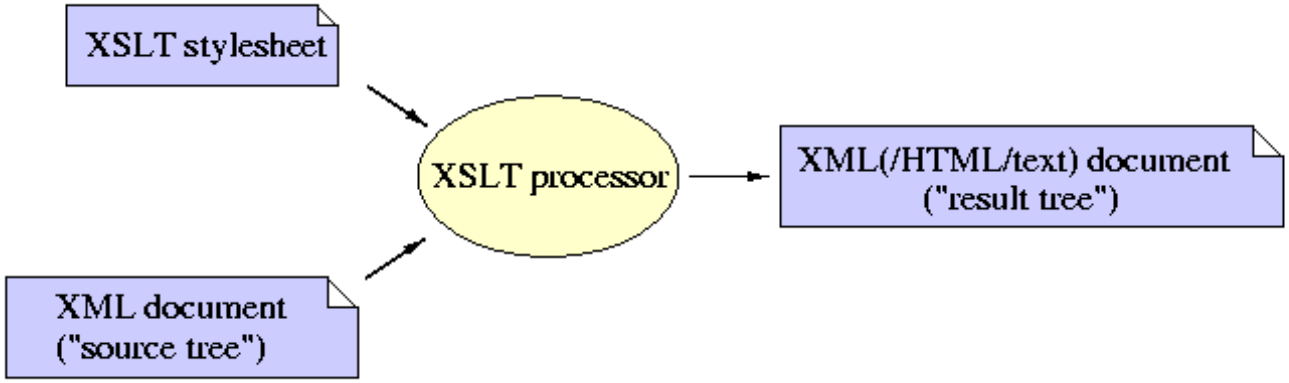
XSLT - XSL Transformations

XSL (e**X**tensible **S**tylesheet **L**anguage) consists of two parts:

1. *XSL Transformations* (**XSLT**), and
2. *XSL Formatting Objects* (**XSL-FO**).

- a stylesheet separates **contents and logical structure** from **presentation** (as with CSS)
- an **XSLT stylesheet** is an XML document defining a **transformation** from one class of XML documents into another
- XSLT is **not** intended as a completely general-purpose XML transformation language - it is designed for **XSL Formatting Objects** as transformation target language - nevertheless: XSLT is generally useful
- **XSL-FO** is an XML language for specifying formatting in a more low-level and detailed way than possible with HTML+CSS

The basic idea of XSLT:



An XSLT stylesheet:

- is **declarative** and uses **pattern matching** and **templates** to specify the transformation
- is vastly more **expressive** than a CSS stylesheet
- may perform arbitrary **computations** (it is theoretically Turing complete!)

Tools:

- on the Web, XSLT transformation can be done either on the **client** (e.g. Explorer or Mozilla), or on the **server** (e.g. Apache Xalan) - either as **pre-processing** or **on-the-fly**
- in the future, Web browsers **only need to understand XSLT and XSL-FO** (rendering HTML/XHTML can be done using a standard stylesheet)
- today, the target language is typically **XHTML** which is understood by current browsers
- XSLT is widely implemented - XSL-FO is not yet...

Processing model

An XSLT stylesheet consists of a number of **template rules**:

template rule = pattern + template

For a given input XML document, the output is obtained as follows:

- the **source tree** is processed by processing the root node (implicitly, the root node of a document is a special *document node* which has the root element as its only child)

- a single **node** is processed by:
  1. **finding** the template rule with the best matching pattern
  2. **instantiating** its template (creates result fragment + continues processing recursively)

- a **node list** is processed by processing each node in order and concatenating the results

This cartoon illustrates the processing model.

Structure of a stylesheet

An XSLT stylesheet is itself an XML document:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" version="1.0"
                xmlns="...">
   .
   .
   .
  <xsl:template match="pattern">   \
    template                        > a template rule
  </xsl:template>                   /
   .
   .                    <- other top-level elements
   .
</xsl:stylesheet>
```

The namespace **`http://www.w3.org/1999/XSL/Transform`** is used to recognize the XSLT elements; elements from other namespaces constitute *literal result fragments*.

A document may refer to a stylesheet using the processing instruction:

```
<?xml-stylesheet type="text/xsl" href="foo.xsl"?
>
```

Newer browsers contain an XSLT processor.

A tiny example

The following XSLT stylesheet transforms XML business cards into XHTML:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="card">
    <html>
      <head>
        <title><xsl:value-of select="name/text()"/></title>
      </head>
      <body bgcolor="#ffffff">
        <table border="3">
          <tr>
            <td>
              <xsl:apply-templates select="name"/><br/>
              <xsl:apply-templates select="title"/><p/>
              <tt><xsl:apply-templates select="email"/></tt><br/>
              <xsl:if test="phone">
                Phone: <xsl:apply-templates select="phone"/><br/>
              </xsl:if>
            </td>
            <td>
              <xsl:if test="logo">
                <img src="{logo/@url}"/>
              </xsl:if>
            </td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="name">
    <xsl:value-of select="text()"/>
  </xsl:template>

  <xsl:template match="title">
    <xsl:value-of select="text()"/>
  </xsl:template>

  <xsl:template match="email">
    <xsl:value-of select="text()"/>
  </xsl:template>

  <xsl:template match="phone">
    <xsl:value-of select="text()"/>
  </xsl:template>
</xsl:stylesheet>
```

The transformation applied to the business card:

```
<card>
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo url="widget.gif"/>
</card>
```

looks like:

John Doe
CEO, Widget Inc.

john.doe@widget.com
Phone: (202) 555-1414

A CSS example

The CSS2 language extends the CSS ideas from HTML to XML.

The following CSS2 stylesheet also makes business cards visible in the browser:

```
card  { background-color: #cccccc; border: none; width: 300;}
name  { display: block; font-size: 20pt; margin-left: 0; }
title { display: block; margin-left: 20pt;}
email { display: block; font-family: monospace; margin-left: 20pt;}
phone { display: block; margin-left: 20pt;}
```

The transformation applied to the business card:

```
<card>
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo url="widget.gif"/>
</card>
```

looks like:

John Doe

CEO, Widget Inc.
john.doe@widget.com
(202) 555-1414

CSS2 is very limited compared to XSLT:

- information cannot be rearranged
- no real computation is possible
- the target cannot be another XML language

Patterns

Patterns are simple XPath expressions evaluating to **node-sets**.

A given node **matches** a given pattern if:

> the node is member of the result of **evaluating** the pattern with respect to **some** context.

Operationally (and intuitively), a pattern matching is probably best evaluated backwards (from right to left).

Recall the structure of XPath node-set expressions:

pattern:	*location path* **|** ... **|** *location path*
location path:	*/step/* ... **//** ... */step*
step:	*axis nodetest predicate*

- a pattern is a set of XPath location paths separated by **|** (union)
- restrictions: only the **`child`** (default) and **`attribute`** (@) axes are allowed here
- extensions: the location paths may start with **`id(..)`** or **`key(..)`**

A simple example is:

```
match="section/subsection | appendix//subsection"
```

which matches **`subsection`** elements occurring either as child elements of **`section`** elements or as descendants of **`appendix`** elements.

These figures illustrate the pattern matching mechanism.

Templates

There are many different kinds of template constructs:

- literal result fragments
- recursive processing
- computed result fragments
- conditional processing
- sorting
- numbering
- variables and parameters
- keys

Literal result fragments

A literal result fragment is:

- a text constant (character data)
- an element not belonging to the XSL namespace
- **<xsl:text ...> ... </...>** (as raw text, but with white-space and character escaping control)
- **<xsl:comment> ... </...>** (inserts a comment **<!-...->**)

Since literal fragments are part of the stylesheet XML document, only well-formed XML will be generated.

Example:

```
<xsl:template match="...">
  this text is written directly to output
  when this template is instantiated
</xsl:template>
```

Recursive processing

Recursive processing instructions:

- **`<xsl:apply-templates select="`*`node-set expression`*`" .../>`**
  apply pattern matching and template instantiation on selected nodes (default: all children)

- **`<xsl:call-template name="..."/>`**
  invoke template by name (where **`xsl:template`** has **`name="..."`** attribute)

- **`<xsl:for-each select="`*`node-set expression`*`">`** ***`template`*** **`</...>`**
  instantiate inlined template for each node in node-set (document order by default)

- **`<xsl:copy>`** ***`template`*** **`</...>`**
  copy current node to output and apply template (shallow copy)

- **`<xsl:copy-of select="..."/>`**
  copy selected nodes to output (deep copy, includes descendants)

The **`select`** attribute contains an **XPath expression**, which is evaluated in the current context.

Example:

```
<xsl:template match="article">
  <h1><xsl:apply-templates select="title"/></h1>
</xsl:template>
```

Processing modes: **`mode="..."`** on **`xsl:template`** and **`xsl:apply-templates`** allows an element to be processed multiple times in different ways.

Computed result fragments

Result fragments can be computed using XPath expressions:

- **`<xsl:element name="..." namespace="..."> ... </...>`**
  construct an element with the given name, attributes, and contents
- **`<xsl:attribute name="..." namespace="..."> ... </...>`**
  construct an attribute (inside **`xsl:element`**)
- **`<xsl:value-of select="..."/>`**
  construct character data or attribute value (expression converted to string)
- **`<xsl:processing-instruction name="..."> ... </...>`**
  construct a processing instruction

The attributes may contain ***{expression}***: XPath expressions which are **evaluated** (and coerced to string) on instantiation.

Example:

```
<xsl:template match="section">
  <xsl:element name="sec{@level}">
    <xsl:attribute name="kind">
      <xsl:value-of select="kind"/>
    </xsl:attribute>
  </xsl:element>
</xsl:template>
```

This template rule converts
`<section level="3"><kind>intro</kind></section>`
into
`<sec3 kind="intro"/>`.

Conditional processing

Processing can be conditional:

- **<xsl:if test="*expression*"> ... </...>**
  apply template if expression (coerced to boolean) evaluates to true

- **<xsl:choose>**
  <xsl:when test="*expression*"> ... </...>
  ...
  <xsl:otherwise> ... </...>
  </...>
  test conditions in turn, apply template for the first that is true

Example:

```
<xsl:template match="nutrition">
  <xsl:if test="@alcohol">
    <td align="right"><xsl:value-of select="@alcohol"/>%</td>
  </xsl:if>
</xsl:template>
```

Sorting

Sorting chooses an order for **`xsl:apply-templates`** and **`xsl:for-each`** (default: document order):

- **`<xsl:sort select="`*`expression`*`" .../>`**;
  a sequence of **`xsl:sort`** elements placed in **`xsl:apply-templates`** or **`xsl:for-each`** defines a lexicographic order

Some extra attributes:

- **`order="ascending/descending"`**
- **`lang="..."`**
- **`data-type="text/number"`**
- **`case-order="upper-first/lower-first"`**

Example:

```
<xsl:template match="personlist">
  <xsl:apply-templates select="person">
    <xsl:sort select="name/family"/>
    <xsl:sort select="name/given"/>
  </xsl:apply-templates>
</xsl:template>
```

This template rule processes a list of persons, sorted with family name as primary key and given name as secondary key.

Numbering

- for automatic numbering of sections, item lists, footnotes, etc.

<xsl: number	**value="*expression*"**	converted to number
	format="..."	(as **ol** in HTML, default: "**1.** ")
	level="..."	**any/single/multiple**
	count="..."	select what to count
	from="..."	select where to start counting
	lang="..."	
	letter-value="..."	
	grouping-separator="..."	
	grouping-size="..."/>	

- If **value** is specified, that value is used.
- Otherwise, the action is determined by **level**:
  - **level="any"**: number of preceding **count** nodes occurring after **from**
    (example use: numbering footnotes)
  - **level="single"** (the default): as **any** but only considers ancestors and their siblings
    (example use: numbering ordered list items)
  - **level="multiple"**: generates whole list of numbers
    (example use: numbering sections and subsections at the same time)

Example:

```
<xsl:template match="footnote">
  (<xsl:number level="any" count="footnote" from="chapter" format="1"/>)
</xsl:template>
```

- much of the functionality of **xsl:number** can also be achieved using **xsl:value-of**
- an example of a non-simplistic design :-(

Variables and parameters

- for reusing results of computations and parameterizing templates and whole stylesheets

- static scope rules
- can hold any XPath value (string, number, boolean, node-set) + **result-tree fragment**
- purely declarative: variables cannot be updated
- can be global or local to a template rule

Declaration:

- **`<xsl:variable name="..." select="`*`expression`*`"/>`**
  variable declaration, value given by XPath expression
- **`<xsl:variable name="...">`** ***`template`*** **`</..>`**
  variable declaration, template is instantiated as result tree fragment to give value

- similarly for **`xsl:param`** parameter declarations (where the specified values act as defaults).

Use:

- **`$`*`name`***
  returns XPath value in expressions, e.g. attribute value templates
- **`xsl:with-param`**
  passes parameters in **`xsl:call-template`** and **`xsl:apply-templates`**

Example 1:

```
<xsl:template match="foo">

  <xsl:variable name="X" select="42"/>
  <xsl:variable name="Y">
    <sometag><xsl:value-of select="@bar"/></sometag>
  </xsl:variable>

  <first>
    <xsl:value-of select="$X"/>
    <xsl:copy-of select="$Y"/>
  </first>
  <second>
    <xsl:value-of select="$X"/>
    <xsl:copy-of select="$Y"/>
  </second>

</xsl:template>
```

Example 2:

```
  ...
  <xsl:apply-templates select="item">
    <xsl:with-param name="COLOR" select="blue"/>
  </xsl:apply-templates>
  ...

<xsl:template match="item">
  <xsl:param name="COLOR" select="black"/>
  <font color="{$COLOR}">
    <xsl:apply-templates/>
  </font>
</xsl:template>
```

- the mechanism for assigning values to top-level (global) parameters is implementation dependent

- unfortunately, result tree fragments in variables cannot be used as source for pattern matching and template instantiation - so general **composition** of transformations is not possible within a single

stylesheet :-(

Keys

- advanced node IDs for automatic construction of links

A key is a triple (**node**, **name**, **value**) associating a name-value pair to a tree node.

<xsl:key match="*pattern*" name="..." use="*node set expression*"/>

declares set of keys - one for each node matching the pattern and for each node in the node set

Extra XPath key function:

key(*name expression, value expression*)

returns nodes with given key name and value

This is often used together with:

generate-id(*singleton node-set expression*)

returns unique string identifying the given node

Example:

```
<xsl:key name="mykeys" match="section[@id]" use="@id"/>

<xsl:template match="section">
  <h1>
    <a name="{generate-id()}">
      <xsl:number/>
      <xsl:apply-templates select="title"/>
    </a>
  </h1>
  <xsl:apply-templates select="body"/>
</xsl:template>

<xsl:template match="ref[@section]">
```

```
  <a href="#{generate-id(key('mykeys',@section))}">
    <xsl:for-each select="key('mykeys',@section)">
      Section <xsl:number/>
    </xsl:for-each>
  </a>
</xsl:template>
```

- a key is declared for each **`section`** element with an **`id`** attribute
- at each section title, a link anchor with a unique name is inserted
- at each **`ref`** element with a **`section`** attribute, a link to the appropriate section is inserted using the key to locate the destination node
- at the same time, both the section titles and the references are numbered

Comparison to DTD IDs:

- keys are declared in the stylesheet (not in the DTD)
- keys allow different "name spaces"
- key values can be placed anywhere (not just as attributes)
- one node may have several keys
- keys need not be unique

Other issues

Things not covered here:

- conflict resolution (**`priority`**) - choosing a template rule when multiple patterns match
- output modes (**`xml`**, **`html`**, **`text`**) - constructing HTML or non-formatted text instead of XML
- white-space handling (**`strip-space`**, **`preserve-space`**) and output escaping (**`disable-output-escaping`**)
- **`attribute-set`** - grouping attribute declarations
- additional XPath functions (**`document`**, **`format-number`**, **`current`**, ...) - allow multiple input documents, etc.
- stylesheet **`import`**/**`include`** - modularity
- built-in template rules - convenient, but confusing for beginners

XSL Formatting Objects

- XSL-FO provides exact and detailed layout control
- it resembles e.g. LaTeX, but is XML based
- recall that HTML/XHTML has different goals: the exact look of an HTML/XHTML page is decided by the browser, not by the author

A small example:

```
<?xml version="1.0"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <fo:layout-master-set>
    <fo:simple-page-master master-name="my-page">
      <fo:region-body margin="1in"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <fo:page-sequence master-name="my-page">
    <fo:flow flow-name="xsl-region-body">
      <fo:block font-family="Times" font-size="14pt">
        <fo:inline font-weight="bold">Hello</fo:inline>, world!
      </fo:block>
    </fo:flow>
  </fo:page-sequence>

</fo:root>
```

- **layout masters** define the page layout
- pages are grouped into **page sequences**
- **flow objects** bind contents to page regions
- the actual contents is grouped in **blocks**
- inside blocks, content fragments can be assigned **inline properties**

- XSL-FO documents are almost always created using XSLT!

XSL-FO is not supported by existing browsers, but can be tried out using FOP that translates into PDF.

Examples

The following XSLT stylesheet produces an XHTML version of the recipe XML example and illustrates many XSLT features:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns="http://www.w3.org/1999/xhtml">

  <xsl:template match="collection">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title><xsl:apply-templates select="description"/></title>
        <link href="../style.css" rel="stylesheet" type="text/css"/>
      </head>
      <body>
        <table border="1">
          <xsl:apply-templates select="recipe"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="description">
     <xsl:value-of select="text()"/>
  </xsl:template>

  <xsl:template match="recipe">
    <tr>
      <td>
        <h1>
          <xsl:apply-templates select="title"/>
        </h1>
        <ul>
          <xsl:apply-templates select="ingredient"/>
        </ul>
        <xsl:apply-templates select="preparation"/>
        <xsl:apply-templates select="comment"/>
        <xsl:apply-templates select="nutrition"/>
      </td>
    </tr>
  </xsl:template>

  <xsl:template match="ingredient">
    <xsl:choose>
      <xsl:when test="@amount">
        <li>
          <xsl:if test="@amount!='*'">
            <xsl:value-of select="@amount"/>
            <xsl:text> </xsl:text>
            <xsl:if test="@unit">
              <xsl:value-of select="@unit"/>
```

```
            <xsl:if test="number(@amount)&gt;number(1)">
               <xsl:text>s</xsl:text>
            </xsl:if>
            <xsl:text> of </xsl:text>
          </xsl:if>
          <xsl:text> </xsl:text>
        </xsl:if>
        <xsl:value-of select="@name"/>
      </li>
    </xsl:when>
    <xsl:otherwise>
      <li><xsl:value-of select="@name"/>
        <ul>
          <xsl:apply-templates select="ingredient"/>
        </ul>
        <xsl:apply-templates select="preparation"/>
      </li>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="preparation">
  <ol><xsl:apply-templates select="step"/></ol>
</xsl:template>

<xsl:template match="step">
  <li><xsl:value-of select="node()"/></li>
</xsl:template>

<xsl:template match="comment">
  <ul>
    <li type="square"><xsl:value-of select="node()"/></li>
  </ul>
</xsl:template>

<xsl:template match="nutrition">
  <table border="2">
    <tr>
      <th>Calories</th><th>Fat</th><th>Carbohydrates</th><th>Protein</th>
      <xsl:if test="@alcohol">
        <th>Alcohol</th>
      </xsl:if>
    </tr>
    <tr>
      <td align="right"><xsl:value-of select="@calories"/></td>
      <td align="right"><xsl:value-of select="@fat"/>%</td>
      <td align="right"><xsl:value-of select="@carbohydrates"/>%</td>
      <td align="right"><xsl:value-of select="@protein"/>%</td>
      <xsl:if test="@alcohol">
        <td align="right"><xsl:value-of select="@alcohol"/>%</td>
      </xsl:if>
    </tr>
  </table>
</xsl:template>
```

```
</xsl:stylesheet>
```

- **apply-templates** is the most often used XSLT instruction
- most **select** expressions select elements among the children of the current element (so processing is mostly top-down)
- **<xsl:text> </xsl:text>** is the right way of producing pure-white-space nodes
- **choose** and **if** are useful for branching based on complex decisions
- the resulting XHTML document contains a reference to a CSS stylesheet (XSLT and CSS work well together!)
- it is difficult to ensure that the output is always valid!

Different views

The following XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="collection">
    <nutrition>
      <xsl:apply-templates select="recipe"/>
    </nutrition>
  </xsl:template>

  <xsl:template match="recipe">
    <dish name="{title/text()}"
          calories="{nutrition/@calories}"
          fat="{nutrition/@fat}"
          carbohydrates="{nutrition/@carbohydrates}"
          protein="{nutrition/@protein}"
          alcohol="{number(concat(0,nutrition/@alcohol))}"/>
  </xsl:template>
</xsl:stylesheet>
```

produces a different view of the recipes:

```
<nutrition>
  <dish alcohol="0"
        protein="32"
        carbohydrates="45"
        fat="23" calories="1167"
        name="Beef Parmesan with Garlic Angel Hair Pasta"/>
  <dish alcohol="0"
        protein="18"
        carbohydrates="64"
        fat="18"
        calories="349"
        name="Ricotta Pie"/>
  <dish alcohol="0"
        protein="29"
        carbohydrates="59"
        fat="12"
        calories="532"
        name="Linguine Pescadoro"/>
  <dish alcohol="2"
        protein="4"
        carbohydrates="45"
        fat="49"
        calories="612"
        name="Zuppa Inglese"/>
  <dish alcohol="0"
        protein="39"
        carbohydrates="28"
        fat="33"
```

```
        calories="8892"
        name="Cailles en Sarcophages"/>
</nutrition>
```

which validates according to the DSD2 schema:

```
<dsd:dsd xmlns:dsd="http://www.brics.dk/DSD/2.0"
         root="nutrition">

  <dsd:if><dsd:element name="nutrition"/>
    <dsd:declare><dsd:contents>
      <dsd:repeat><dsd:element name="dish"/></dsd:repeat>
    </dsd:contents></dsd:declare>
  </dsd:if>

  <dsd:if><dsd:element name="dish"/>
    <dsd:declare>
      <dsd:required><dsd:attribute name="name"/></dsd:required>
      <dsd:attribute name="calories"/>
      <dsd:attribute name="carbohydrates"/>
      <dsd:attribute name="protein"/>
      <dsd:attribute name="alcohol"/>
      <dsd:attribute name="fat"/>
    </dsd:declare>
  </dsd:if>

</dsd:dsd>
```

and using the XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns="http://www.w3.org/1999/xhtml">

  <xsl:template match="nutrition">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <link href="../style.css" rel="stylesheet" type="text/css"/>
      </head>
      <body>
        <table border="1">
          <tr>
            <th>Dish</th>
            <th>Calories</th>
            <th>Fat</th>
          <th>Carbohydrates</th>
          <th>Protein</th>
         </tr>
         <xsl:apply-templates select="dish"/>
      </table>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="dish">
    <tr>
      <td><xsl:value-of select="@name"/></td>
      <td><xsl:value-of select="@calories"/></td>
      <td><xsl:value-of select="@fat"/>%</td>
      <td><xsl:value-of select="@carbohydrates"/>%</td>
      <td><xsl:value-of select="@protein"/>%</td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

then produces the following XHTML table:

Dish	Calories	Fat	Carbohydrates	Protein
Beef Parmesan with Garlic Angel Hair Pasta	1167	23%	45%	32%
Ricotta Pie	349	18%	64%	18%
Linguine Pescadoro	532	12%	59%	29%
Zuppa Inglese	612	49%	45%	4%
Cailles en Sarcophages	8892	33%	28%	39%

XSLT 2.0

- currently a Working Draft

Major changes from 1.0:

- using **XPath 2.0** (which implies: using **sequences** and **XML Schema primitive types**)
  - sequences replace the notion of "result tree fragments"
  - sequences are first-class citizens that can be subjected to further processing

- **multiple output documents** (**`<xsl:result-document href="`*`file`*`">`** *`template`* **`</xsl:result-document>`**)

- better support for **grouping** (**`for-each-group`**)

- user-defined functions (allow reuse of XPath expressions)

- XML Base support

- XHTML output method

- XML Schema validation
  - optional type annotations at variable/parameter declarations and templates
  - dynamic validation
  - static validation ("It is implementation-defined whether type errors are signaled statically.")

- no complete implementations yet (in particular, XML Schema support is missing)

Links to more information

www.w3.org/Style/XSL/

W3C's XSL homepage, contains lots of links

www.w3.org/TR/xslt

the XSLT 1.0 specification

www.w3.org/TR/xslt20/

working draft for XSLT 2.0

www.w3.org/TR/xsl

the XSL 1.0 (defines the Formatting Objects XML language)

www.mulberrytech.com/xsl/xsl-list/

XSL-List - mailing list

www.ibiblio.org/xml/books/bible2/chapters/ch17.html

a chapter from "The XML Bible" on XSL Transformations

www.ibiblio.org/xml/books/bible2/chapters/ch18.html

a chapter from "The XML Bible" on XSL Formatting Objects

nwalsh.com/docs/tutorials/xsl/

an XSL tutorial by Paul Grosso and Norman Walsh

xml.apache.org/xalan-j

Xalan, a Java/C++ XSLT implementation

saxon.sourceforge.net

SAXON, another Java implementation

xmlsoft.org/XSLT

libxslt, C implementation from the Gnome project

www.jclark.com/xml/xt.html

XT, an early Java implementation by the editor of the XSLT spec

xml.apache.org/fop

an XSL Formatting Objects to PDF converter

XQuery

- information extraction and transformation

- Queries on XML documents - generalizing relational data
- Usage scenarios - why do we need it?
- Query languages requirements - the W3C specification
- The XQuery language - the canonical notation
- XQuery concepts - writing queries
  - Path expressions
  - Element constructors
  - FLWOR expressions
  - List expressions
  - Conditional expressions
  - Quantified expressions
  - Datatype expressions
- Other issues - things not covered here
- Examples - continuing the recipe example
- XML databases - implemenating XQuery
- XML shredding - mapping XML into relational data
- From XQuery to SQL - mapping queries
- Mixed processing - a spectrum of possibilities
- Links to more information

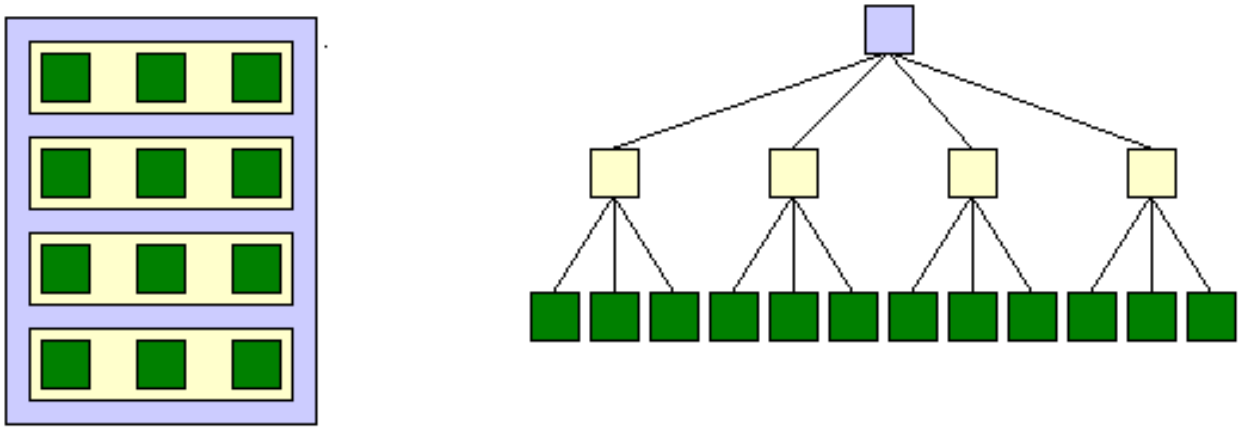
Queries on XML documents

The database community has long been looking for a richer data model:

- hierarchical databases
- object-oriented databases
- multi-dimensional databases

but no consensus has emerged yet.

XML documents **generalize** relational data in a very straightforward manner:



Here, we see:

relations (tables)
tuples (records)
attributes (entries)

A relation is a tree of height two with:

- *unbounded* fanout at the first level
- *fixed* fanout at the second level

In contrast, an XML document is an **arbitrary** tree.

How should query languages like SQL be similarly generalized?

Usage scenarios

XML querying is relevant for:

- **human-readable documents**
  to retrieve individual documents, to provide dynamic indexes, to perform context-sensitive searching, and to generate new documents

- **data-oriented documents**
  to query (virtual) XML representations of databases, to transform data into new XML representations, and to integrate data from multiple heterogeneous data sources

- **mixed-model documents**
  to perform queries on documents with embedded data, such as catalogs, patient health records, employment records, or business analysis documents

- in short, *information retrieval*.

Query language requirements

The W3C Query Working Group has identified many technical requirements:

- at least one **XML syntax** (at least one human-readable syntax)
- must be **declarative**
- must be protocol independent
- must respect XML data model
- must be **namespace aware**
- must **coordinate with XML Schema**
- must work even if schemas are unavailable
- must support simple and complex datatypes
- must support **universal and existential quantifiers**
- must support operations on hierarchy and sequence of document structures
- must **combine information from multiple documents**
- must support aggregation
- must be **able to transform and to create XML structures**
- must be able to traverse ID references

In short, it must be SQL generalized to XML!

The XQuery language

The query language developed by W3C is called **XQuery** and is currently at the level of a Working Draft.

It is derived from several previous proposals:

- XML-QL
- YATL
- Lorel
- Quilt

which happily all **agreed** on the fundamental principles.

XQuery relies on XPath and XML Schema datatypes.

XQuery is **not** an XML language - a version in XML syntax is called XQueryX.

XQuery concepts

A **query** in XQuery is an expression that:

- reads a sequence of XML fragments or atomic values
- returns a sequence of XML fragments or atomic values

The **principal forms** of XQuery expressions are:

- path expressions
- element constructors
- FLWOR ("flower") expressions
- list expressions
- conditional expressions
- quantified expressions
- datatype expressions

Expressions are evaluated relative to a **context**:

- namespaces
- variables
- functions
- date and time
- context item (current node or atomic value)
- context position (in the sequence being processed)
- context size (of the sequence being processed)

Path expressions

The simplest kind of query is just an **XPath 2.0** expression.

A simple path expression looks like:

```
document("recipes.xml")//recipe[title="Ricotta Pie"]//ingredient[@amount]
```

- the result is all simple ingredients used to prepare Ricotta Pie in the recipe collection
- the result is given as a list of XML fragments, each rooted with an **`ingredient`** element
- the **order** of the fragments respects the document order (order matters! - as opposed to SQL)

The initial context for the path expression is given by **`document("recipes.xml")`** (similarly to **XPointer**).

Some XQuery specific extension of XPath:

- location steps may follow a new **IDREF axis**
- an arbitrary XQuery expression may be used as a location step

Element constructors

An XQuery expression may **construct** a new XML element:

```
<employee empid="12345">
  <name>John Doe</name>
  <job>XML specialist</job>
  <deptno>187>/deptno>
  <salary>125000>/salary>
</employee>
```

which just evaluates to itself.

In the XQuery syntax this is **unambiguous** - XQueryX must use namespaces!

More interestingly, an expression may use values bound to **variables**:

```
<employee empid="{$id}">
  <name>{$name}</name>
  {$job}
  <deptno>{$deptno}>/deptno>
  <salary>{$SGMLspecialist+100000}>/salary>
</employee>
```

Here the variables **`$id`**, **`$name`**, and **`$job`** must be bound to appropriate XML fragments or strings.

In general, **`{...}`** may contain full XQuery expressions returning sequences.

FLWOR expressions

The **main engine** of XQuery is the FLWOR expression:

- **F**or-**L**et-**W**here-**O**rder-**R**eturn
- pronounced "flower"
- generalizes SELECT-FROM-HAVING-WHERE from SQL

A complete example is:

```
for $d in document("depts.xml")//deptno
let $e := document("emps.xml")//employee[deptno = $d]
where count($e) >= 10
order by avg($e/salary) descending
return
  <big-dept>
     { $d,
        <headcount>{count($e)}</headcount>,
        <avgsal>{avg($e/salary)}</avgsal>
     }
  </big-dept>
```

- **`for`** generates an ordered list of bindings of **`deptno`** values to **`$d`**
- **`let`** associates to each binding a further binding of the list of **`emp`** elements with that department number to **`$e`**
- at this stage, we have an ordered list of tuples of bindings: **`($d,$e)`**
- **`where`** filters that list to retain only the desired tuples
- **`order`** sorts that list by the given criteria
- **`return`** constructs for each tuple a resulting value

The combined result is in this case a list of departments with at least 10 employees, sorted by average salaries.

General rules:

- **`for`** and **`let`** may be used many times in any order
- only one **`where`** is allowed
- many different sorting criteria can be specified

Note the difference between **`for`** and **`let`**:

```
for $x in /company/employee
```

generates a list of bindings of **`$x`** to each **`employee`** element in the **`company`**, but:

```
let $x := /company/employee
```

generates a single binding of **`$x`** to the list of **`employee`** elements in the **`company`**.

This is also sufficient to compute joins of documents:

```
for $p IN document("www.irs.gov/taxpayers.xml")//person
for $n IN document("neighbors.xml")//neighbor[ssn = $p/ssn]
return
  <person>
    <ssn> { $p/ssn } </ssn>
    { $n/name }
    <income> { $p/income } </income>
  </person>
```

List expressions

XQuery expressions manipulate **lists** of values, for which many **operators** are supported:

- constant lists: **`(7, 9, <thirteen/>)`**
- integer ranges: ***`i`*** **`to`** ***`j`***
- XPath expressions, like all named children of the context node: **`name`**
- concatenation: **`,`**
- set operators: **`|`** (or **`union`**), **`intersect`**, **`except`**
- functions: **`remove`**, **`index-of`**, **`count`**, **`avg`**, **`max`**, **`min`**, **`sum`**, **`distinct-values`** ...

When lists are viewed as sets:

- XML nodes are compared on node identity
- duplicates are removed
- the order is preserved

The following query lists each publisher and the average price of their books:

```
for $p in distinct-values(document("bib.xml")//publisher)
let $a := avg(document("bib.xml")//book[publisher = $p]/price)
return
  <publisher>
     <name>{ $p/text() }</name>
     <avgprice>{ $a }</avgprice>
  </publisher>
```

Compare this with the verbose XQueryX syntax.

Conditional expressions

XQuery supports a general **`if-then-else`** construction.

The example query:

```
for $h in document("library.xml")//holding
return
  <holding>
    { $h/title,
      if ($h/@type = "Journal")
      then $h/editor
      else $h/author
    }
  </holding>
```

extracts from the holdings of a library the titles and either editors or authors.

Quantified expressions

XQuery allows **quantified** expressions, which decide properties for all elements in a list:

- **some-in-satisfies**
- **every-in-satisfies**

The following example finds the titles of all books which mention both sailing and windsurfing in the same paragraph:

```
for $b in document("bib.xml")//book
where some $p in $b//paragraph satisfies
  (contains($p,"sailing") AND contains($p,"windsurfing"))
return $b/title
```

The next example finds the titles of all books which mention sailing in every paragraph:

```
for $b in document("bib.xml")//book
where every $p in $b//paragraph satisfies
  contains($p,"sailing")
return $b/title
```

Datatype expressions

XQuery supports all **datatypes** from XML Schema, both primitive and complex types.

Constant values can be written:

- as literals (like **`string`**, **`integer`**, **`float`**)
- as constructor functions (**`true()`**, **`date("2001-06-07")`**)
- as explicit casts (**`cast as xsd:positiveInteger(47)`**)

Arbitrary XML Schema documents can be imported into a query.

An **`instance of`** operator allows runtime **validation** of any value relative to a datatype or a schema.

A **`typeswitch`** operator allows **branching** based on types.

Other issues

Things not covered here:

- hundreds of built-in operators and functions - contains anything you might think of
- computed element and attribute names - allow more flexible queries
- user-defined functions - allow general-purpose computations
- views and updates are still under development

Examples

The following XQuery expressions extract information from the recipe collection:

"The titles of all recipes":

```
for $t in document("recipes.xml")//title
return $t
```

```
<title>Beef Parmesan with Garlic Angel Hair Pasta</title>,
<title>Ricotta Pie</title>,
<title>Linguine Pescadoro</title>,
<title>Zuppa Inglese</title>,
<title>Cailles en Sarcophages</title>
```

"The dishes that contain flour":

```
<floury>
  { for $r in document("recipes.xml")//recipe[.//ingredient[@name="flour"]]
    return <dish>{$r/title/text()}</dish>
  }
</floury>
```

```
<floury>
  <dish>Ricotta Pie</dish>
  <dish>Zuppa Inglese</dish>
  <dish>Cailles en Sarcophages</dish>
</floury>
```

"For each ingredient, the recipes that it is used in":

```
for $i in distinct-values(document("recipes.xml")//ingredient/@name)
return <ingredient name="{$i}">
            { for $r in document("recipes.xml")//recipe
              where $r//ingredient[@name=$i]
              return $r/title
            }
       </ingredient>
```

```
<ingredient name="beef cube steak">
  <title>Beef Parmesan with Garlic Angel Hair Pasta</title>
</ingredient>,
<ingredient name="onion, sliced into thin rings">
  <title>Beef Parmesan with Garlic Angel Hair Pasta</title>
</ingredient>,
  ...
<ingredient name="butter">
  <title>Beef Parmesan with Garlic Angel Hair Pasta</title>
  <title>Cailles en Sarcophages</title>
</ingredient>,
  ...
```

"The recipes that use some of the stuff in our refrigerator":

```
distinct-values(
  for $r in document("recipes.xml")//recipe
  for $i in $r//ingredient/@name
  for $j in document("fridge.xml")//stuff[text()=$i]
  return $r/title
)
```

```
<title>Beef Parmesan with Garlic Angel Hair Pasta</title>,
<title>Ricotta Pie</title>,
<title>Linguine Pescadoro</title>,
```

XML databases

XML databases running XQuery may be:

- **native** - specialized engines evaluating queries on XML documents
  examples are Galax and Qexo
- **relational** - built on top of existing database engines
  most commercial database products support some version of this with a subset of XQuery or XPath

Native XML processing is:

- lightweight and easy to extend with new XML features
- unable to scale and weak on security, concurrency, transactions, recovery, ...

Relational XML processing is:

- an attempt to get the best from both worlds

XML shredding

With relational processing, the XML document must be stored in a relation.

Mapping from XML to relations is called **shredding**, which for the recipes could look like:

collection

id	description
147	Some recipes used in the XML tutorial.

recipe

id	parent	title
231	147	Beef Parmesan with Garlic Angel Hair Pasta
237	147	Ricotta Pie
242	147	Linguine Pescadoro
247	147	Zuppa Inglese
253	147	Cailles en Sarcophages

ingredient

id	parent	name	amount	unit
411	231	beef cube steak	1.5	pound
462	237	ricotta cheese	3	pound
535	247	egg yolks	4	
612	147	pastry		
789	612	flour	3	cup
...	...	...	...	...

preparation

id	parent
376	231
...	...

step

id	parent	text
423	376	Preheat oven to 350 degrees F (175 degrees C).
...	...	...

comment

id	parent	text
...	...	...

nutrition

id	parent	calories	fat	carbohydrates	protein	alcohol
...	...	...	...	...	...	...

In this construction, we have:

- one relation for each element type
- a unique id for each occurrence of an element
- identifications of the parent nodes
- the ids of siblings are ordered

From XQuery to SQL

Using a tool like SilkRoute, XQuery queries can be translated into equivalent SQL queries.

As a simple example, the following XQuery query:

```
//ingredient[@name="butter"]/@amount
```

corresponds to the simple SQL query:

```
select ingredient.amount
from ingredient
where ingredient.name='butter'
```

This becomes much more complicated for general queries.

The performance of a translated query depends on the shredding.

In response, the LegoDB system performs **adaptive** shredding.

Mixed processing

There is a spectrum of possibilities between native and relational XML databases.

Shredding with **fine** XML fragments:

- small irregular XML fragments are stored as character data in tuples (**`VARCHAR`**)
- simplifies the mapping
- may improve performance
- example: XHTML help texts

Shredding with **coarse** XML fragments:

- larger XML fragments are stored as external character data (**`CLOB`**)
- necessary if the schema is unknown or some XML features are not supported
- decreases performance
- example: SOAP message contents

Links to more information

www.w3.org/TR/xquery

XQuery 1.0 Working Draft

www.w3.org/TR/xquery-requirements

W3C XML Query Requirements

www.w3.org/TR/xmlquery-use-cases

XML Query Use Cases

www.w3.org/TR/query-semantics

XQuery 1.0 Formal Semantics

db.bell-labs.com/galax/

XQuery prototype implementation

DOM, SAX, and JDOM

- XML support in programming languages

XML and programming

XSLT, XPath and XQuery provide tools for **specialized** tasks.

But many applications are not covered:

- **domain-specific** tools for **concrete** XML languages
- **general** tools that nobody has thought of yet

To work with XML in general-purpose programming languages we need to:

- **parse** XML documents into XML trees
- **navigate** through XML trees
- **construct** XML trees
- **output** XML trees as XML documents

DOM and SAX are corresponding APIs that are language independent and supported by numerous languages.

JDOM is an API that is tailored to Java.

Typical examples: domain-specific editors and browsers.

The DOM API

DOM is the official W3C proposal.

It views an XML tree as a data structure, similar to the DOM from Javascript.

It is quite large and complex...

- Level 1 Core: W3C Recommendation, October 1998
  - primitive navigation and manipulation of XML trees
  - other Level 1 parts: HTML
- Level 2 Core: W3C Recommendation, November 2000
  - adds Namespace support and minor new features
  - other Level 2 parts: Events, Views, Style, Traversal and Range
- Level 3 Core: W3C Working Draft, April 2002
  - adds minor new features
  - other Level 3 parts: Schemas, XPath, Load/Save

The DOM API is specified in OMG IDL (Interface Definition Language).

A simple DOM example

The following Java program uses DOM to read the recipe collection and cut it down to the first recipe:

```
import java.io.*;
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;

public class FirstRecipeDOM {

  public static void main(String[] args) {
    try {
      DOMParser p = new DOMParser();
      p.parse(args[0]);
      Document doc = p.getDocument();
      Node n = doc.getDocumentElement().getFirstChild();
      while (n!=null && !n.getNodeName().equals("recipe"))
        n = n.getNextSibling();
      PrintStream out = System.out;
      out.println("<?xml version=\"1.0\"?>");
      out.println("<collection>");
      if (n!=null)
        print(n, out);
      out.println("</collection>");
    } catch (Exception e) {e.printStackTrace();}
  }

  static void print(Node node, PrintStream out) {
    int type = node.getNodeType();
    switch (type) {
      case Node.ELEMENT_NODE:
        out.print("<" + node.getNodeName());
        NamedNodeMap attrs = node.getAttributes();
        int len = attrs.getLength();
        for (int i=0; i<len; i++) {
            Attr attr = (Attr)attrs.item(i);
            out.print(" " + attr.getNodeName() + "=\"" +
                      escapeXML(attr.getNodeValue()) + "\"");
        }
```

```
        out.print('>');
        NodeList children = node.getChildNodes();
        len = children.getLength();
        for (int i=0; i<len; i++)
          print(children.item(i), out);
        out.print("</" + node.getNodeName() + ">");
        break;
      case Node.ENTITY_REFERENCE_NODE:
        out.print("&" + node.getNodeName() + ";");
        break;
      case Node.CDATA_SECTION_NODE:
        out.print("<![CDATA[" + node.getNodeValue() + "]]>");
        break;
      case Node.TEXT_NODE:
        out.print(escapeXML(node.getNodeValue()));
        break;
      case Node.PROCESSING_INSTRUCTION_NODE:
        out.print("<?" + node.getNodeName());
        String data = node.getNodeValue();
        if (data!=null && data.length()>0)
           out.print(" " + data);
        out.println("?>");
        break;
    }
  }

  static String escapeXML(String s) {
    StringBuffer str = new StringBuffer();
    int len = (s != null) ? s.length() : 0;
    for (int i=0; i<len; i++) {
       char ch = s.charAt(i);
       switch (ch) {
       case '<': str.append("&lt;"); break;
       case '>': str.append("&gt;"); break;
       case '&': str.append("&amp;"); break;
       case '"': str.append("&quot;"); break;
       case '\'': str.append("&apos;"); break;
       default: str.append(ch);
     }
    }
    return str.toString();
  }
}
```

Note that:

- we need to make our own **`print`** method
- when using DOM in Java, one actually uses the Java language binding

The SAX API

SAX (Simple API for XML) started as a grassroots movement, but has gained an official standing.

An XML tree is not viewed as a data structure, but as a stream of **events** generated by the parser.

The kinds of events are:

- the **start** of the document is encountered
- the **end** of the document is encountered
- the **start tag** of an element is encountered
- the **end tag** of an element is encountered
- **character data** is encountered
- a **processing instruction** is encountered

Scanning the XML file from start to end, each event invokes a corresponding **callback** method that the programmer writes.

An XML tree can be built in response, but it is not required to construct a data structure.

This is sometimes much more efficient:

- the document can be piped through the application
- the only real option for very large documents
- good for local processing, not for random access

A simple SAX example

The following Java programs reads the recipe collection and outputs the total amount of flour being used (assuming the unit is always **cup**):

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;

public class Flour extends DefaultHandler {

  float amount = 0;

  public void startElement(String namespaceURI, String localName,
                           String qName, Attributes atts) {
    if (namespaceURI.equals("http://recipes.org") && localName.equals("ingredient"))
{
       String n = atts.getValue("","name");
       if (n.equals("flour")) {
         String a = atts.getValue("","amount"); // assume 'amount' exists
         amount = amount + Float.valueOf(a).floatValue();
       }
    }
  }

  public static void main(String[] args) {
    Flour f = new Flour();
    SAXParser p = new SAXParser();
    p.setContentHandler(f);
    try { p.parse(args[0]); }
    catch (Exception e) {e.printStackTrace();}
    System.out.println(f.amount);
  }
}
```

The output for our recipe collection is:

```
7.75
```

Only a tiny amount of the XML document is stored at any time.

SAX events

The following Java program traces all SAX events generated by parsing the recipe collection:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;

public class Trace extends DefaultHandler {

  int indent;

  void printIndent() {
    for (int i=0; i<indent; i++) System.out.print("-");
  }

  public void startDocument() {
    System.out.println("start document");
  }

  public void endDocument() {
    System.out.println("end document");
  }

  public void startElement(String uri, String localName,
                           String qName, Attributes attributes) {
    printIndent();
    System.out.println("starting element: " + qName);
    indent++;
  }

  public void endElement(String uri, String localName,
                         String qName) {
    indent--;
    printIndent();
    System.out.println("end element: " + qName);
  }

  public void ignorableWhitespace(char[] ch, int start, int length) {
    printIndent();
    System.out.println("whitespace, length " + length);
  }

  public void processingInstruction(String target, String data) {
```

```
    printIndent();
    System.out.println("processing instruction: " + target);
  }

  public void characters(char[] ch, int start, int length){
    printIndent();
    System.out.println("character data, length " + length);
  }

  public static void main(String[] args) {
    Trace t = new Trace();
    SAXParser p = new SAXParser();
    p.setContentHandler(t);
    try { p.parse(args[0]); }
    catch (Exception e) {e.printStackTrace();}
  }
}
```

The output is (abbreviated with `...`):

```
start document
processing instruction: dsd
starting element: collection
-character data, length 3
-starting element: description
--character data, length 47
-end element: description
-character data, length 3
-starting element: recipe
--character data, length 5
...
-end element: recipe
-character data, length 1
end element: collection
end document
```

The JDOM API

DOM is too complicated to suit many programmers.

Since it is a **general** API, it does not use **special** Java features - for example, existing collection classes are ignored

JDOM is designed to be **simple** and **Java-specific**.

JDOM is a **small** (124K) library, since it is used on top of either DOM or SAX.
- a full XML parser is complex, dealing with encodings, namespaces, and entities

It shares the simple view of XML trees presented in these slides.

A simple JDOM example

The following Java program uses JDOM to read the recipe collection and cut it down to the first recipe:

```
import java.io.*;
import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;

public class FirstRecipeJDOM {
  public static void main(String[] args) {
    try {
      Document d = new SAXBuilder().build(new File(args[0]));
      Namespace ns = Namespace.getNamespace("http://recipes.org");
      Element r = d.getRootElement().getChild("recipe", ns).detach();
      Document n = new Document((new Element("collection")).addContent(r));
      new XMLOutputter().output(n, System.out);
    } catch (Exception e) {e.printStackTrace();}
  }
}
```

Compare this with the DOM version.

The JDOM packages

JDOM contains five Java packages:

- **`org.jdom`** - defines the basic model of an XML tree
- **`org.jdom.adapters`** - defines wrappers for various DOM implementations
- **`org.jdom.input`** - defines means for reading XML documents
- **`org.jdom.output`** - defines means for writing XML documents
- **`org.jdom.transform`** - defines an interface to JAXP XSLT

The JDOM tree model

JDOM has a class for every kind of XML tree node (in the general sense):

- **`Document`**
- **`Element`**
- **`Attribute`**
- **`Namespace`**
- **`Text`**
- **`CDATA`**
- **`Comment`**
- **`DocType`**
- **`EntityRef`**
- **`ProcessingInstruction`**

Each node has a parent pointer. There are no sibling pointers but several methods for accessing the child nodes.

Content is given as a **`java.util.List`** - modifications are reflected in the tree.

Surprisingly, the common behavior of nodes is not modelled by interfaces or abstract superclasses.

JDOM input and output

Parsing XML documents into JDOM:

- this can be done with basically any DOM or SAX parser
- SAX is preferable, since it avoids the construction of a DOM tree
- parsing errors are reported as exceptions

Output from JDOM can be generated in different ways:

- as the corresponding sequence of SAX events
- as a standard DOM tree
- as an (indented) XML document represented as a stream of characters (to a file or another application)

JAXP

JAXP is the official API for XML processing from Sun.

It supports DOM, SAX, and XSLT (which may be run inside Java applications) - and also a number of other Java/XML technologies.

JDOM is rumored to become integrated into a future version.

A Business Card editor

A typical Java application is a **domain-specific XML editor**:

- nobody wants to write the markup by hand
- general-purpose XML editors are too clunky

We generalize the business card language to allow collections of business cards:

```
<cards>
  <card>
    <name>John Doe</name>
    <title>CEO, Widget Inc.</title>
    <email>john.doe@widget.com</email>
    <phone>(202) 456-1414</phone>
    <logo url="widget.gif" />
  </card>
  <card>
    <name>Michael Schwartzbach</name>
    <title>Associate Professor</title>
    <email>mis@brics.dk</email>
    <phone>+45 8610 8790</phone>
    <logo url="http://www.brics.dk/~mis/portrait.gif" />
  </card>
  <card>
    <name>Anders Møller</name>
    <title>Research Assistant Professor</title>
    <email>amoeller@brics.dk</email>
    <phone>+45 8942 3475</phone>
    <logo url="http://www.brics.dk/~amoeller/am.jpg"/>
  </card>
</cards>
```

We then write a Java program to edit such collections.

First, we need a high-level representation of a business card:

```
class Card {
  public String name, title, email, phone, logo;

  public Card(String name, String title, String email, String phone, String logo) {
    this.name = name;
    this.title = title;
    this.email = email;
    this.phone = phone;
    this.logo = logo;
  }
}
```

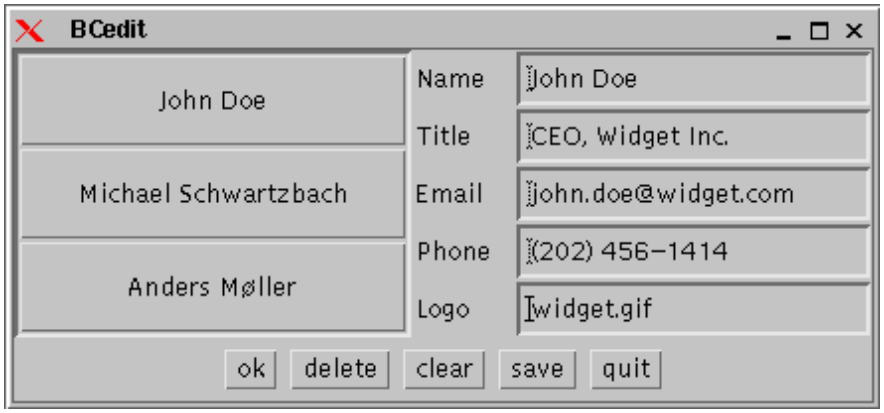
An XML document must then be translated into a vector of such objects:

```
Vector doc2vector(Document d) {
  Vector v = new Vector();
  Iterator i = d.getRootElement().getChildren().iterator();
  while (i.hasNext()) {
    Element e = (Element)i.next();
    String phone = e.getChildText("phone");
    if (phone==null) phone="";
    Element logo = e.getChild("logo");
    String url;
    if (logo==null) url = ""; else url = logo.getAttributeValue("url");
    Card c = new Card(e.getChildText("name"),  // exploit schema,
                      e.getChildText("title"), // assume validity
                      e.getChildText("email"),
                      phone,
                      url);
    v.add(c);
  }
  return v;
}
```

And back into an XML document:

```
Document vector2doc() {
  Element cards = new Element("cards");
  for (int i=0; i<cardvector.size(); i++) {
    Card c = (Card)cardvector.elementAt(i);
    if (c!=null) {
      Element card = new Element("card");
      Element name = new Element("name");
      name.addContent(c.name);
      card.addContent(name);
      Element title = new Element("title");
      title.addContent(c.title);
      card.addContent(title);
      Element email = new Element("email");
      email.addContent(c.email);
      card.addContent(email);
      if (!c.phone.equals("")) {
        Element phone = new Element("phone");
        phone.addContent(c.phone);
        card.addContent(phone);
      }
      if (!c.logo.equals("")) {
        Element logo = new Element("logo");
        logo.setAttribute("url",c.logo);
        card.addContent(logo);
      }
      cards.addContent(card);
    }
  }
  return new Document(cards);
}
```

A little logic and some GUI then completes the editor:



Compile with: **`javac -classpath xerces.jar:jdom.jar BCedit.java`**

This example contains some general observations:

- XML documents are parsed via JDOM into **domain-specific data structures**
- if the input is known to **validate** according to some schema, then many runtime errors can be assumed never to occur
- how do we ensure that the output of **`vector2doc`** is valid according to the schema? (well-formedness is for free)
  - that's a current research challenge!

Problems with JDOM

JDOM is not (yet) perfect:

- it is still a beta version under development
- shared functionality is not collected in an interface or common superclass (so many casts are necessary)
- the documentation is insufficient (for instance, **`Element.getContent`** returns a **`List`**, but it is not specified which of the optional methods that are implemented, so none of them can be used safely)

XML Binding

For the business card editor we had to:

- write Java classes corresponding to the XML schema (bind)
- write code to import an XML document (unmarshal)
- write code to export an XML document (marshal)

This is necessary for **every** XML application in Java, so **automatic support** would be useful.

Things to consider when designing a data binding framework:

- which XML schema language is supported?
- can the binding into classes be customized?
- does the generated classes extend some standard framework?
- are imported documents validated?
- are exported documents validated?
- are documents (partially) validated when they are modified?

JAXB

JAXB is the official API for XML binding from Sun:

- supports binding of DTD and XML Schema
- allows customization of the binding
- standardizes the interfaces for the generated classes
- allows different JAXB implementations to use different implementations of these interfaces
- supports validation of unmarshalled documents
- supports validation of content trees
- is unrelated to DOM and JDOM

Other XML data binding frameworks:

- Castor
- XGen
- Breeze

A comparison is available.

Links to more information

www.w3.org/DOM/

DOM homepage

sax.sourceforge.net

SAX project website

java.sun.com/xml/

SUN's Java/XML page

java.sun.com/xml/jaxp/

JAXP, Sun's Java APIs for XML Processing

java.sun.com/xml/jaxb/

JAXB, Sun's Java APIs for XML Binding

xmlsoft.org

libxml, Gnome project's XML C library

xml.apache.org

Apache's XML page

xml.apache.org/xerces2-j/

Apache's XML parser

www.cafeconleche.org/books/xmljava/

"Processing XML with Java"

www.brics.dk/~amoeller/WWW/

the tutorial *Interactive Web Services with Java*

Background: W3C

- a look at the organization behind most of the XML-related specifications

- W3C - The World Wide Web Consortium
- Organization
- Activities
- Technical Reports
- Policies

W3C - The World Wide Web Consortium

www.w3.org

- the de facto leader in defining Web standards

Consists of around 380 companies and organizations, led by Tim Berners-Lee, creator of the World Wide Web.

W3C's Mission Statement:

> "To lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability."

Competitors:

- ISO (standardization of anything from dentistry to nuclear energy)
- OASIS (e-business),
- ECMA (information and communication systems)

Coming up: an overview of the W3C Process Document...

See also the Outsider's Guide to the W3C.

Organization

W3C's organizational structure:

- the **Members** (who pay $50,000 a year!) - companies and organizations (carry out activities)

- the **Team** - Chairman, Director, Staff, Fellows (technical leadership, coordinates activities, hosted by MIT, INRIA, and Keio)
- the **Advisory Board** (elected) - provides guidance of strategy
- the **Technical Architecture Group** - build consensus around the principles of the Web architecture

(Compare this with ISO!)

Activities

Activities carried out by groups:

- Working Groups - produce specifications and prototypes
- Interest Groups - explore and evaluate technologies
- Coordination Groups - ensure consistency and integrety between other groups

Current XML groups:

- query
- stylesheet
- schema
- linking
- core
- coordination

Other (current or former) W3C activities: HTML, HTTP, PNG, Amaya, ...

Organization of events:

- workshops - short expert meeting
- symposia - education
- conferences - the International World Wide Web Conference

Technical Reports

- the central activity of W3C

Member submissions and Working Group publications:

- Notes - acknowledged submissions by Members (members only!), Working Group notes, etc.;

Recommendation track:

- Working Drafts - Working Group reports (work in progress)
- Candidate Recommendations - stable Working Drafts
- Proposed Recommendations - being reviewed by the Advisory Committee
- Recommendations - standards recommended by W3C (although they don't call them "standards")

Be aware of the publication status when considering W3C publications!

Policies

Guiding the development work:

- consensus - reach "substantial agreement"
- dissemination - limit intellectual property rights, ensure availability

+ the unofficial:

- better too soon than too late - otherwise someone else will take over
- greatest common denominator - every interested member is allowed one favourite feature in each spec

Designing by committees does not necessarily produce the best solutions - but often the most widely used!