

Bases de données et Web

Lucas Pastor

L3 Info & L3 MIASHS

2018-2019



Email : lucas.pastor@uca.fr

Page du cours : Moodle -> Base de Données et Web

Bibliographie : Les supports de cours sont largement hérités de :

- Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart. *Web Data Management*, Cambridge University Press, 2011.
- Yves Roos. Cours de Langages Avancés pour les Bases de Données, Université Lille 1, 2016.

Table des matières

- 1 Introduction
- 2 Typage de données XML
- 3 Navigation avec XPath
- 4 Transformations XSLT
- 5 Programmation avec XQuery
- 6 APIs pour la lecture des fichiers XML

Table des matières

1 Introduction

2 Typage de données XML

3 Navigation avec XPath

4 Transformations XSLT

5 Programmation avec XQuery

6 APIs pour la lecture des fichiers XML

Introduction

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

June 12, 2012

Outline

1 Preliminaries

2 XML, a semi-structured data model

3 XML syntax

4 Typing

5 The XML World

6 Use cases

Web data handling

Web data = by far the **largest** information system ever seen, and a fantastic means of sharing information.

- Billions of textual documents, images, PDF, multimedia files, provided and updated by millions of institutions and individuals.
- An anarchical process which results in highly heterogeneous data organization, steadily growing and extending to meet new requirements.
- New usage and applications appear every day: yesterday P2P file sharing, today social networking, tomorrow ?

The challenge, under a data management perspective: master the size and extreme variability of Web information, and make it **usable**.

The role of XML

Web data management has been primarily based on HTML, which describes presentation.

- HTML is appropriate for humans: allows sophisticated output and interaction with textual documents and images;
- HTML falls short when it comes to software exploitation of data.

XML describes content, and promotes machine-to-machine communication and data exchange

- XML is a generic data format, apt to be specialized for a wide range of fields,
⇒ (X)HTML is a specialized XML dialect for data presentation
- XML makes easier data integration, since data from different sources now share a common format;
- XML comes equipped with many software products, APIs and tools.

Perspective of the course

The course develops an XML perspective of the management of heterogeneous data (e.g., Web data) in a distributed environment.

We introduce **models**, **languages**, **architectures** and **techniques** to fulfill the following goals:

- **flexible data representation and retrieval**

XML is viewed as a new **data model**, both more powerful and more flexible than the relational one

- **data exchange and integration**

XML data can be serialized and restructured for better exchange between systems, and integration of multiple data sources

- **efficient distributed computing and storage**

XML can be the glue for high-level description of distributed repositories; this calls for efficient storage, indexing of management.

Outline

1 Preliminaries

2 XML, a semi-structured data model

- Semi-structured data
- XML

3 XML syntax

4 Typing

5 The XML World

6 Use cases

Semi-structured data model

A data model, based on **graphs**, for representing both regular and irregular data.

Basic ideas

Self-describing data. The content comes with its own description;
⇒ contrast with the relational model, where schema and content
are represented separately.

Flexible typing. Data may be typed (i.e., “such nodes are integer values” or “this
part of the graph complies to this description”); often no typing, or
a very flexible one

Serialized form. The graph representation is associated to a serialized form,
convenient for exchanges in an heterogeneous environment.

Self-describing data

Starting point: **association lists**, i.e., records of label-value pairs.

```
{name: "Alan", tel: 2157786, email: "agb@abc.com"}
```

Natural extension: values may themselves be other structures:

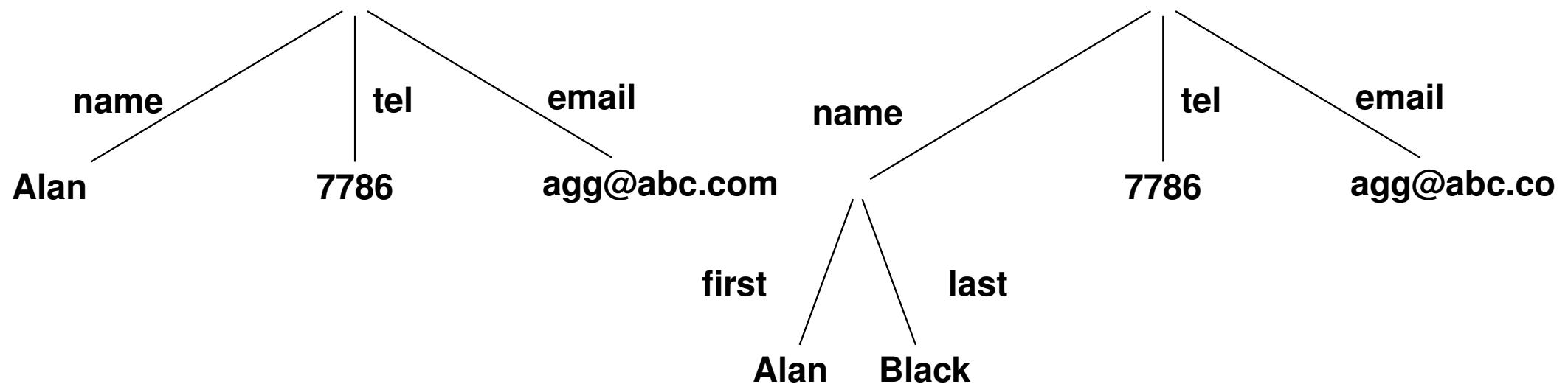
```
{name: {first: "Alan", last: "Black"},  
tel: 2157786,  
email: "agb@abc.com"}
```

Further extension: allow duplicate labels.

```
{name: "alan'", tel: 2157786, tel: 2498762 }
```

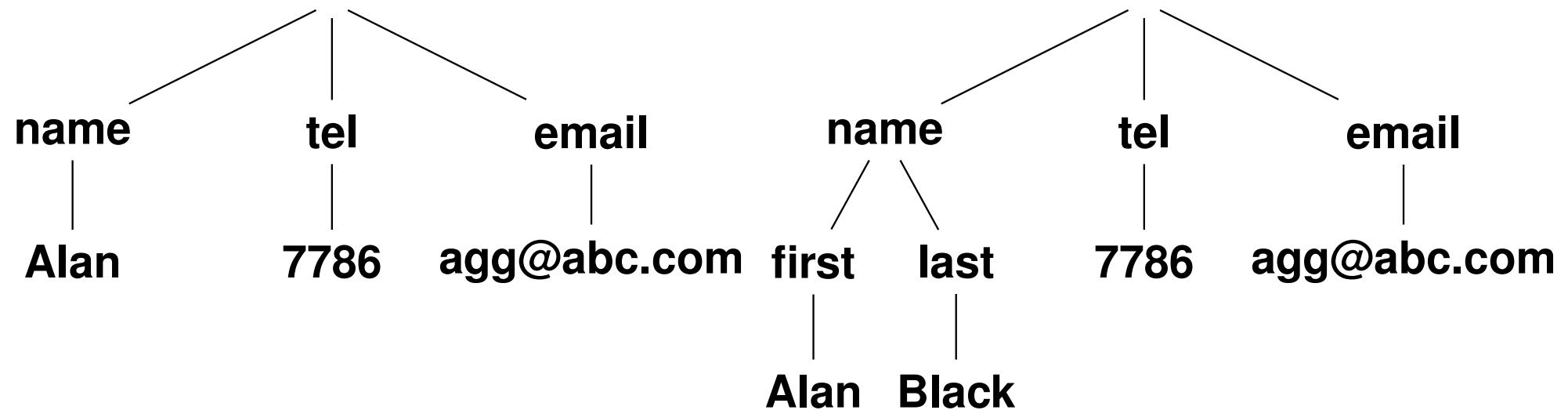
Tree-based representation

Data can be graphically represented as trees: label structure can be captured by tree edges, and values reside at leaves.



Tree-based representation: labels as nodes

Another choice is to represent **both** labels and values as vertices.



Remark

The XML data model adopts this latter representation.

Representation of regular data

The syntax makes it easy to describe sets of tuples as in:

```
{ person: { name: "alan", phone: 3127786, email: "alan@abc.com" },  
  person: { name: "sara", phone: 2136877, email: "sara@xyz.edu" },  
  person: { name: "fred", phone: 7786312, email: "fd@ac.uk" } }
```

Remark

1. relational data can be represented
2. for regular data, the semi-structure representation is highly redundant.

Representation of irregular data

Many possible variations in the structure: missing values, duplicates, changes, etc.

```
{person: {name: "alan", phone: 3127786, email: "agg@abc.com"},  
 person: &314  
     {name: {first: "Sara", last: "Green"},  
      phone: 2136877,  
      email: "sara@math.xyz.edu",  
      spouse: &443},  
 person: &443  
     {name: "fred", Phone: 7786312, Height: 183,  
      spouse: &314}}
```

Node identity

Nodes can be **identified**, and referred to by their identity. Cycles and objects models can be described as well.

XML in brief

XML is the World-Wide-Web Consortium (W3C) standard for Web data exchange.

- XML documents can be serialized in a normalized encoding (typically iso-8859-1, or utf-8), and safely transmitted on the Internet.
- XML is a generic format, which can be specialized in “**dialects**” for specific domain (e.g., XHTML, see further)
- The W3C promotes companion standards: DOM (object model), XSchema (typing), XPath (path expression), XSLT (restructuring), XQuery (query language), and many others.

Remark

1. XML is a simplified version of **SGML**, a long-term used language for technical documents.
2. HTML, up to version 4.0, is **also** a variant of SGML. The successor of HTML 4.0, is **XHTML**, an XML dialect.

XML documents

An XML document is a labeled, unranked, ordered tree:

Labeled means that some annotation, the label, is attached to each node.

Unranked means that there is no a priori bound on the number of children of a node.

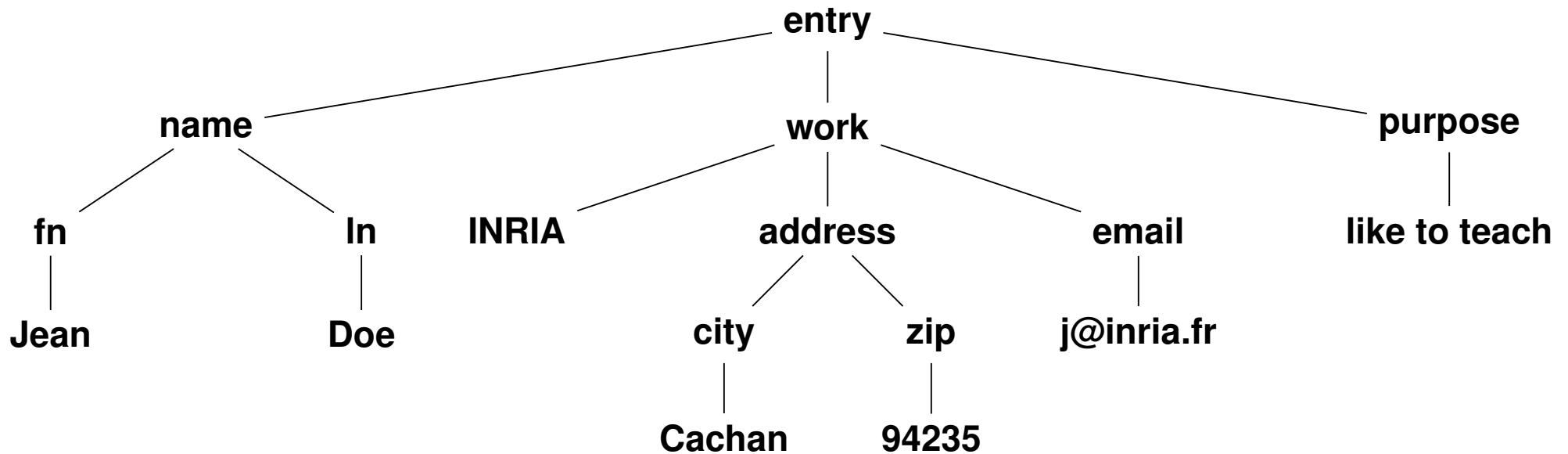
Ordered means that there is an order between the children of each node.

XML specifies nothing more than a syntax: no meaning is attached to the labels.

A dialect, on the other hand, associates a meaning to labels (e.g., `title` in XHTML).

XML documents are trees

Applications view an XML document as a labeled, unranked, ordered tree:



Remark

Some low-level software works on the serialized representation of XML documents, notably SAX (a parser and an API).

Serialized representation of XML document

Documents can be serialized, such as, for instance:

```
<entry><name><fn>Jean</fn><ln>Doe</ln></name><work>INRIA<adress><c  
Cachan</city><zip>94235</zip></adress><email>j@inria.fr</email>  
</work><purpose>like to teach</purpose></entry>
```

or with some beautification as:

```
<entry>  
  <name>  
    <fn>Jean</fn>  
    <ln>Doe</ln>  </name>  
  <work>  
    INRIA  
    <adress>  
      <city>Cachan</city>  
      <zip>94235</zip>  </adress>  
    <email>j@inria.fr</email>  </work>  
    <purpose>like to teach</purpose>  
  </entry>
```

XML describes structured content

Applications cannot interpret unstructured content:

The book ``Fundations of Databases'', written by Serge Abiteboul, Rick Hull and Victor Vianu, published in 1995 by Addison-Wesley

XML provides a means to structure this content:

```
<bibliography>
  <book>
    <title> Foundations of Databases </title>
    <author> Abiteboul </author>
    <author> Hull </author>
    <author> Vianu </author>
    <publisher> Addison Wesley </publisher>
    <year> 1995 </year> </book>
  <book>...</book>
</bibliography>
```

Now, an application can access the XML tree, extract some parts, rename the labels, reorganize the content into another structure, etc.

Applications associate semantics to XML docs

The description of a letter

Letter document

```
<letter>
  <header>
    <author>...</author>
    <date>...</date>
    <recipient>...</recipient>
    <cc>...<cc>
  </header>
  <body>
    <text>...</text>
    <signature>...</signature>
  </body>
</letter>
```

Applications associate semantics to XML docs (2)

Letter style sheet

if *letter* then ...

if *header* then ...

if *author* then ...

if *date* then ...

if *recipient* then ...

if *cc* then ...

if *body* then ...

if *text* then ...

if *signature* then ...

Some software then produces the actual letter to mail or email.

Outline

1 Preliminaries

2 XML, a semi-structured data model

3 XML syntax

- Essential XML Syntax
- XML Syntax: Complements

4 Typing

5 The XML World

6 Use cases

Serialized form, tree form

Typically, an application gets a document in *serialized form*, parse it in *tree form*, and *serializes* it back at the end.



- The serialized form is a textual, linear representation of the tree; it complies to a (sometimes complicated) syntax;
- There exist an object-oriented model for the tree form: the *Document Object Model* (W3C).

Remark

We present here the most significant aspects of both the syntax and the DOM. Details can be found in the W3C documents.

The syntax for serialized document, in brief

Four examples of XML documents (separated by blank lines) are:

```
<document />
```

```
<document> Hello World! </document>
```

```
<document>
  <salutation> Hello World! </salutation>
</document>
```

```
<?xml version="1.0" encoding="utf-8" ?>
<document>
  <salutation color="blue"> Hello World! </salutation>
</document>
```

Last example shows the *prologue*, useful for XML parsers (it gives in particular the document encoding).

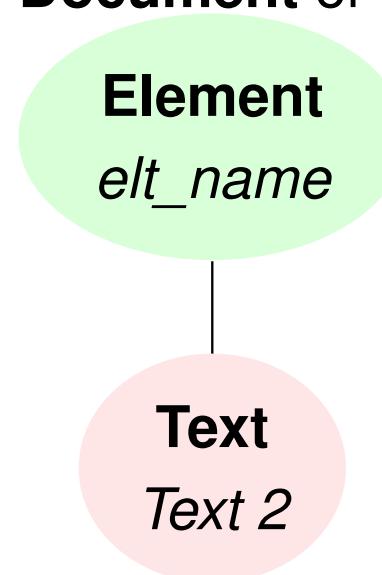
From serialized to tree form: text and elements

The basic components of an XML document are *element* and *text*.

Here is an *element*, whose content is a *text*.

```
<elt_name>  
    Textual content  
</elt_name>
```

The tree form of the document, modeled in DOM: each node has a **type**, either **Document** or **Text**.



From serialized to tree form: nesting elements

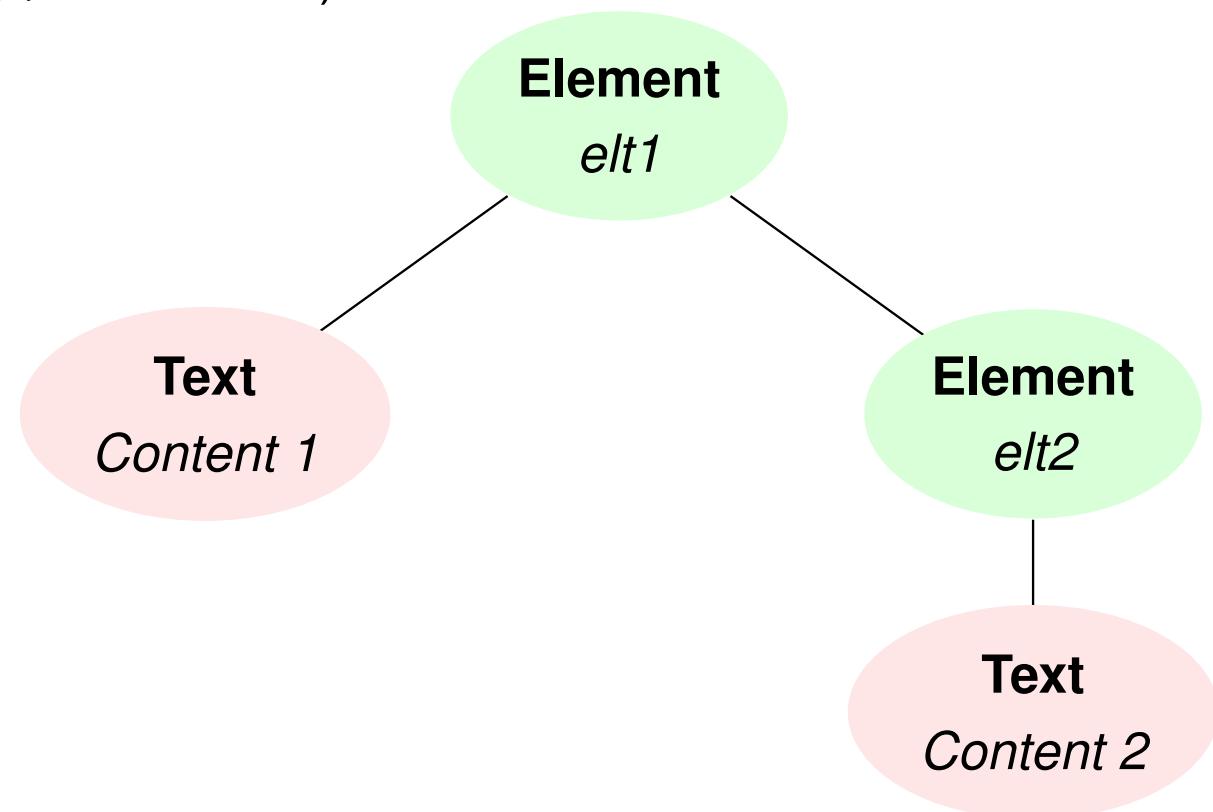
The content of an element is

- ① the part between the opening and ending tags (in serialized form),
- ② the subtree rooted at the corresponding **Element** node (in DOM).

The content may range from atomic text, to any recursive combination of text and elements (and gadgets, e.g., comments).

Example of an element nested
in another element.

```
<elt1>
  Textual content
  <elt2>
    Another content
  </elt2>
</elt1>
```



From serialized to tree form: attributes

Attributes are pairs of name/value attached to an element.

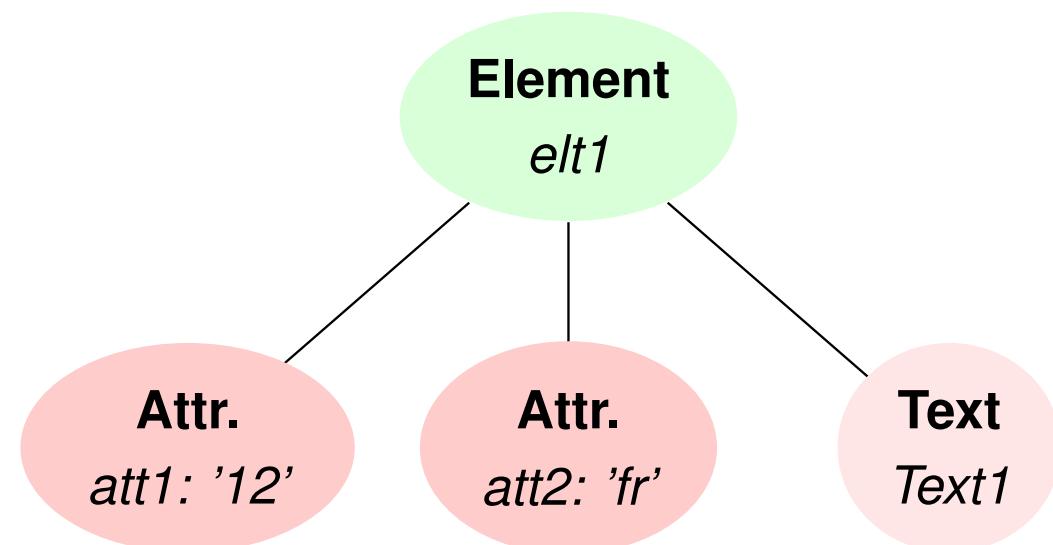
- ① as part of the opening tag in the serialized form,
- ② as special child nodes of the **Element** node (in DOM).

The content of an attribute is always atomic text (no nesting).

An element with two attributes.

```
<elt1 att1='12' att2='fr'>  
    Textual content  
</elt1>
```

Unlike elements, attributes are *not* ordered, and there cannot be two attributes with the same name in an element.



From serialized to tree form: the document root

The first line of the serialized form must *always* be the *prologue* if there is one:

```
<?xml version="1.0" encoding="utf-8"?>
```

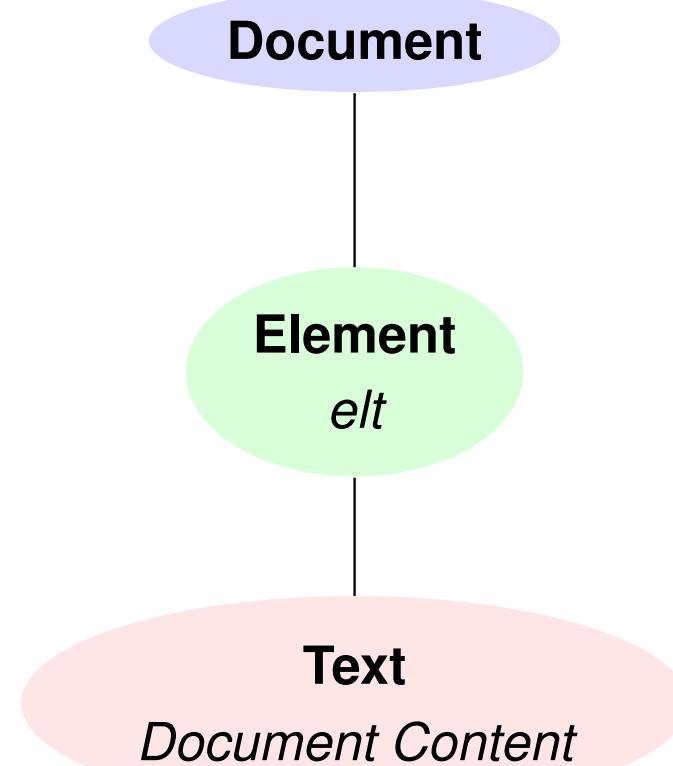
and the document content must *always* be enclosed in a single opening/ending tag, called the *element root*.

A document with its prologue, and element root.

```
<?xml version="1.0"  
       encoding="utf-8" ?>  
<elt>  
  Document content.  
</elt>
```

Note: there may be other syntactic objects after the prologue (processing instructions).

In the DOM representation, the prologue appears as a **Document** node, called the *root node*.



Summary: syntax and vocabulary

Serialized form

- A document begins with a prologue,
- It consists of a single upper-level tag,
- Each *opening tag* `<name>` has a corresponding *closing tag* `</ name>`; everything between is either text or properly enclosed tag content.

Tree form

- A document is a tree with a *root node* (**Document** node in DOM),
- The root node has one and only one element child (**Element** node in DOM), called the *element root*)
- Each element node is the root of a *subtree* which represents its structured *content*

Remark

Other syntactic aspects, not detailed here, pertain to the physical organization of a document. See the lecture notes.

Summary: syntax and semantics

XML provides a syntax

The core of the syntax is the **element name**

Element names have no a priori semantics

Applications assign them a semantics

Entities and references

Entities are used for the physical organization of a document.

An entity is **declared** (in the document type), then **referenced**.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE a [
    <!ENTITY myName "John Doe">
    <!ENTITY mySignature SYSTEM "signature.xml">
] >

<a>
    My name is &myName; .
    &mySignature;
</a>
```

Predefined entities

Several symbols cannot be directly used in an XML document, since they would be misinterpreted by the parser.

They must be introduced as entity references.

Declaration	Reference	Symbol.
<!ENTITY lt "<">	<	<
<!ENTITY gt ">">	>	>
<!ENTITY amp "&">	&	&
<!ENTITY apos "'">	'	'
<!ENTITY quot """>	"	"

Comments and instructions

Comments can be put at any place in the serialized form.

```
<! --This is a comment -->
```

They appear as **Comment** nodes in the DOM tree (they are typically ignored by applications).

Processing instructions: specific commands, useful for some applications, simply ignored by others.

The following instruction requires the transformation of the document by an XSLT stylesheet

```
<?xml-stylesheet href="prog.xslt" type="text/xslt"?>
```

Literal sections

Usually, the content of an element is *parsed* to analyse its structure.

Problem: what if we do *not* want the content to be parsed?

```
<?xml version='1.0'?>
<program>
if ((i < 5) && (j > 6))
    printf("error");
</program>
```

Solution: use entities for all special symbols; or prevent parsing with a *literal section*.

```
<?xml version='1.0'?>
<program>
<! [CDATA[if ((i < 5) && (j > 6))
    printf("error");
]]>
</program>
```

Outline

1 Preliminaries

2 XML, a semi-structured data model

3 XML syntax

4 Typing

5 The XML World

6 Use cases

To type or not to type

What kind of data: very regular one (as in relational databases), less regular (hypertext systems) - all kind of data from very structured to very unstructured.

What kind of typing (unlike in relational systems)

- Possibly irregular, partial, tolerant, flexible
- Possibly evolving
- Possibly very large and complex
- Ignored by some applications such as keyword search.

Typing is not compulsory.

Type declaration

XML documents *may* be typed, although they do not need to. The simplest (and oldest) typing mechanism is based on *Document Type Definitions* (DTD).

A DTD may be specified in the prologue with the keyword **DOCTYPE** using an ad hoc syntax.

A document with proper opening and closing of tags is said to be **well-formed**.

- `<a>...<c>...</c>` is well-formed.
- `<a>...<c>...</c>` is not.
- `<a>...<a>...` is not.

A document that conforms to its DTD is said to be **valid**

Document Type Definition

An example of valid document (the root element matches the DOCTYPE declaration).

```
<?xml version="1.0" standalone="yes" ?>
<!-- Example of a DTD -->
<!DOCTYPE email [
    <!ELEMENT email ( header, body )>
    <!ELEMENT header ( from, to, cc? )>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT cc (#PCDATA)>
    <!ELEMENT body (paragraph*)>
    <!ELEMENT paragraph (#PCDATA)> ]>
<email>
    <header>
        <from> af@abc.com </from>
        <to> zd@ugh.com </to> </header>
    <body> </body> </email>
```

Document Type Definition

A DTD may also be specified externally using an URI.

```
<!DOCTYPE docname SYSTEM "DTD-URI"  
[local-declarations]>
```

- **docname** is the name of the element root
- **DTD-URI** is the URI of the file that contains the DTD
- **local-declarations** are local declarations (mostly for entities.)

Interpreting labels: Namespaces

A particular label, e.g., *job*, may denote different notions in different contexts, e.g., a hiring agency or a computer ASP (application service provider).

The notion of **namespace** is used to distinguish them.

```
<doc xmlns:hire='http://a.hire.com/schema'  
      xmlns:asp='http://b.asp.com/schema' >  
  ...  
  <hire:job> ... </hire:job> ...  
  <asp:job> ... </asp:job> ...  
</doc>
```

DTD vs. XML schema

DTD: old style typing, still very used

XML schema: more modern, used e.g. in Web services

DTD:

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

DTD vs. XML schema

The same structure in XML schema (an XML dialect)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"
          minOccurs='1' maxOccurs='1' />
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Outline

1 Preliminaries

2 XML, a semi-structured data model

3 XML syntax

4 Typing

5 The XML World

6 Use cases

Popular XML dialects

- RSS** is an XML dialect for describing content updates that is heavily used for blog entries, news headlines or podcasts.
- WML** (Wireless Mark-up Language) is used in Web sites by wireless applications based on the Wireless Application Protocol (WAP).
- MathML** (Mathematical Mark-up Language) is an XML dialect for describing mathematical notation and capturing both its structure and content.
- Xlink** (XML Linking Language) is an XML dialect for defining hyperlinks between XML documents. These links are expressed in XML and may be introduced inside XML documents.
- SVG** (Scalable Vector Graphics) is an XML dialect for describing two-dimensional vector graphics, both static and animated. With SVG, images may contain outbound hyperlinks in XLinks.

XML standards

SAX (Simple API for XML) sees an XML document as a sequence of tokens (its serialization).

DOM (Document Object Model) is an object model for representing (HTML and) XML document independently of the programming language.

XPath (XML Path Language) that we will study, is a language for addressing portions of an XML document.

XQuery (that we will study) is a flexible query language for extracting information from collections of XML documents.

XSLT (Extensible Stylesheet Language Transformations), that we will study, is a language for specifying the transformation of XML documents into other XML documents.

Web services provide interoperability between machines based on Web protocols.

Zoom: DOM

Document Object Model - DOM

Document model that is tree-based

Used in APIs for different programming languages (e.g. Java)

Object-oriented access to the content:

- Root of the document
- First child of a node (with a particular label)
- Next one (with a particular label)
- Parent of a node
- Attribute of a node...

Zoom: XPATH

Language for expressing **paths** in an XML document

- Navigation: child, descendant, parent, ancestor
- Tests on the nature of the node
- More complex selection predicates

Means to specify portions of a document

Basic tool for other XML languages: Xlink, XSLT, Xquery

Zoom: XLINK

XML Linking Language

Advanced hypertext primitives

Allows inserting in XML documents descriptions of links to external Web resources

Simple monodirectional links ala (HREF) HTML

Mulridirectional links

XLink relies on XPath for addressing portions of XML documents

Remark

XML trees + XLINK \Rightarrow graph

Zoom: XSLT

Transformation language: “Perl for XML”

An XSLT style sheet includes a set of transformation rules: pattern/template

Pattern: based on XPATH expressions; it specifies a structural context in the tree

Template: specifies what should be produced

Principle: when a pattern is matched in the source document, the corresponding template produces some data

Zoom: XQuery

Query language: “SQL for XML”

Like SQL: select portions of the data and reconstruct a result

Query structure: **FLW** (pronounced "flower")

Exemple

```
FOR $p IN document("bib.xml")//publisher  
LET $b := document("bib.xml")//book[publisher = $p]  
WHERE count($b) > 100  
RETURN $p
```

\$p : scans the sequence of publishers

\$b : scans the sequence of books for a publisher

WHERE filters out some publishers

RETURN constructs the result

Generic XML tools

API

Parsers and type checkers

GUI (Graphical User Interfaces)

Editors

XML diff

XML wiki

Etc.

Facilitate the development of applications specific to particular XML dialects.

Outline

1 Preliminaries

2 XML, a semi-structured data model

3 XML syntax

4 Typing

5 The XML World

6 Use cases

Exploiting XML content

Publishing: an XML document can easily be converted to another XML document (same content, but another structure)

⇒ **Web publishing** is the process of transforming XML documents to XHTML.

Integration: XML documents from many sources can be transformed in a common dialect, and constitute a **collection**.

⇒ **Search engines**, or **portals**, provide browsing and searching services on collections of XML documents.

Distributed Data Processing: many softwares can be adapted to consume/produce XML-represented data.

⇒ **Web services** provide remote services for XML data processing.

Genericity of softwares and APIs

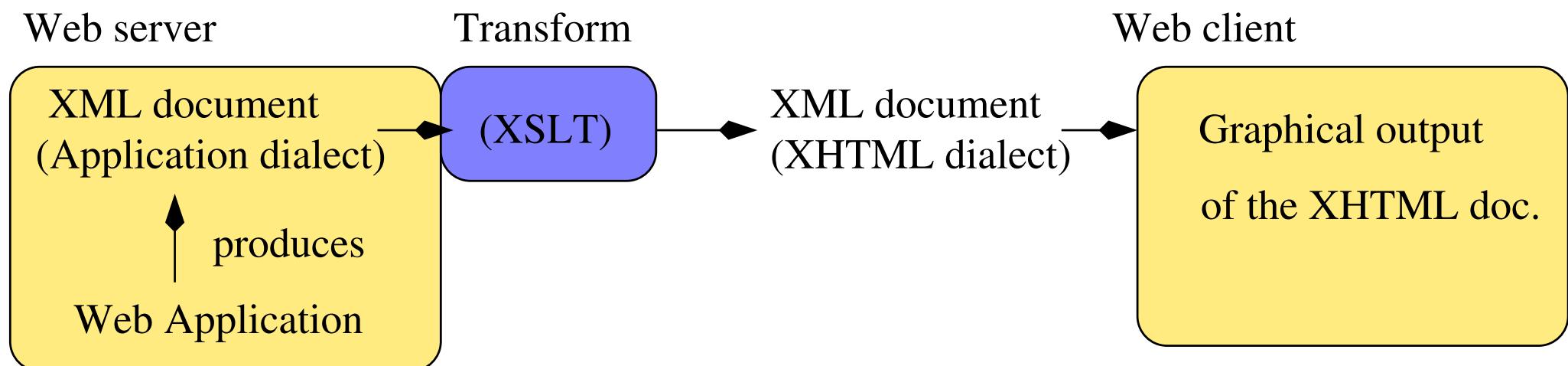
XML comes with many tools that are **generic**: A software or API for XML documents applies to **all** the possible XML dialects.

Web Publishing: restructuring to XHTML

The **Web application** produces some XML content, structured in some application-dependent dialect, on the server.

In a second phase, the XML content is transformed in an XHTML document that can be visualized by humans.

The transformation is typically expressed in XSLT, and can be processed either on the server or on the client.



Web publishing: content + presentation instructions

The following document is an XHTML version of the bibliographic content presented above:

```
<h1> Bibliography </h1>
<p> <i> Foundations of Databases </i>
    Abiteboul, Hull, Vianu
    <br/> Addison Wesley, 1995 </p>
<p> <i> Data on the Web </i>
    Abiteboul, Buneman, Suciu
    <br/> Morgan Kaufmann, 1999 </p>
```

Now the labels belong to the XHTML dialect, and can be interpreted by a Web browser.

[Test: biblio.html](#)

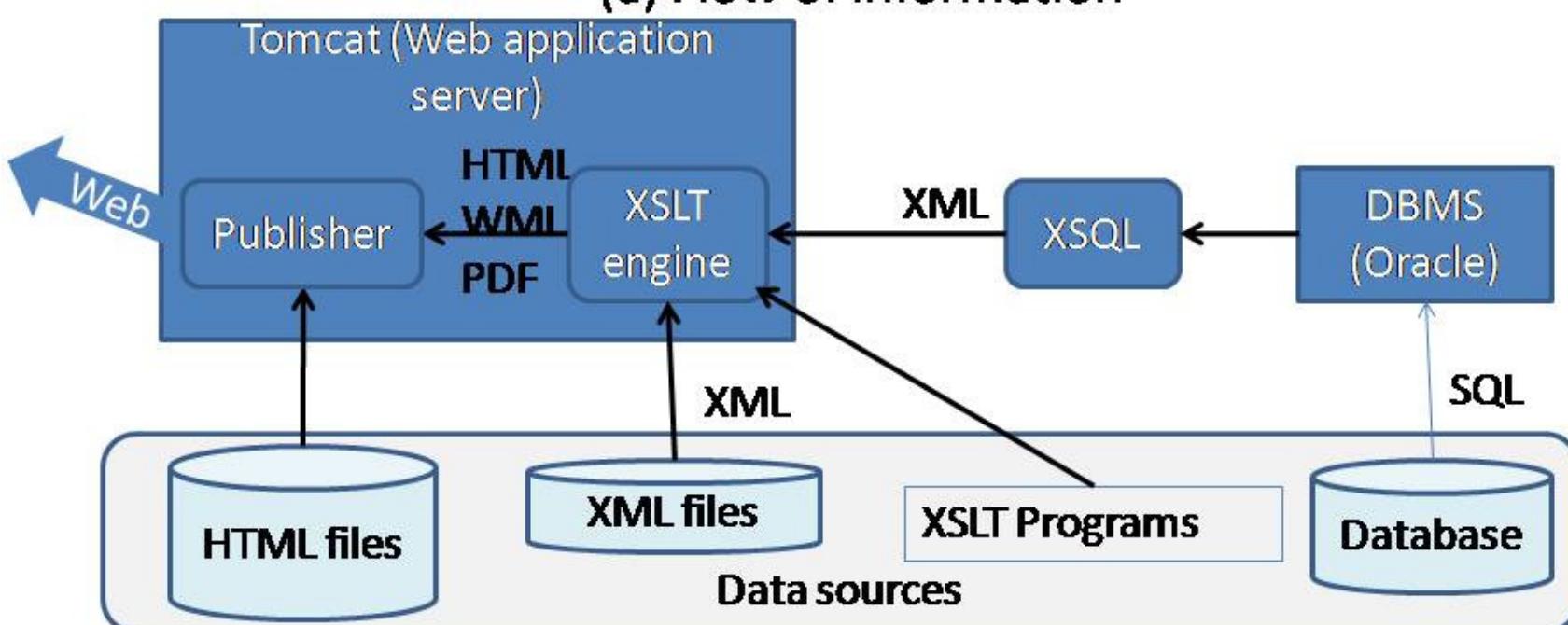
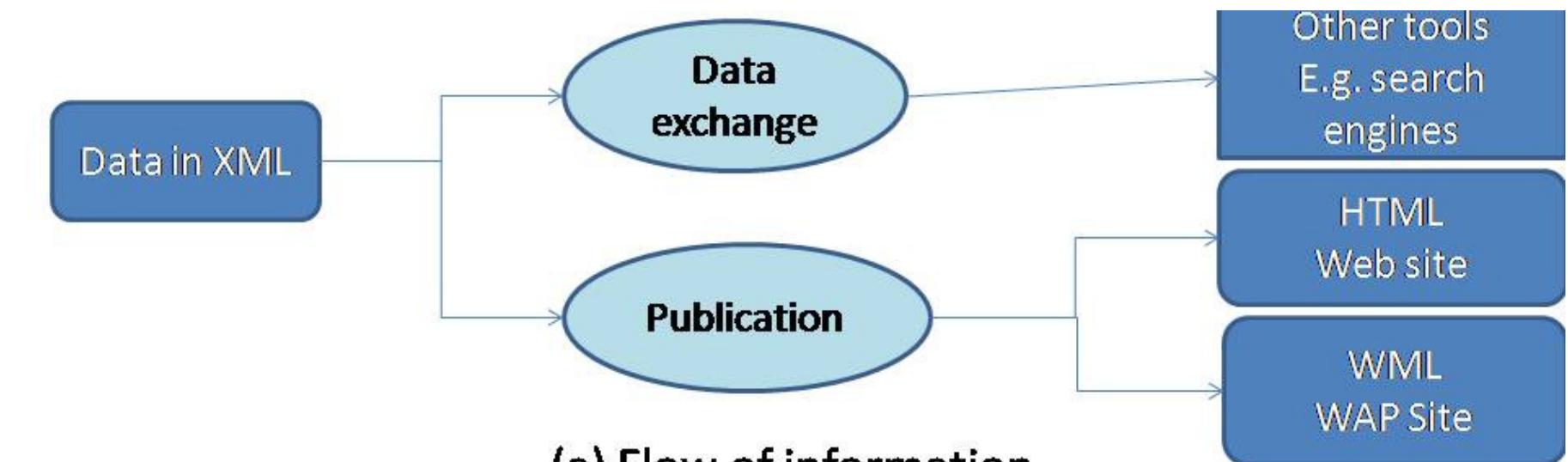
Note that the “meaning” of labels is restricted to presentation purposes. It becomes complicated for a software to distinguish the name of authors.

Web publishing

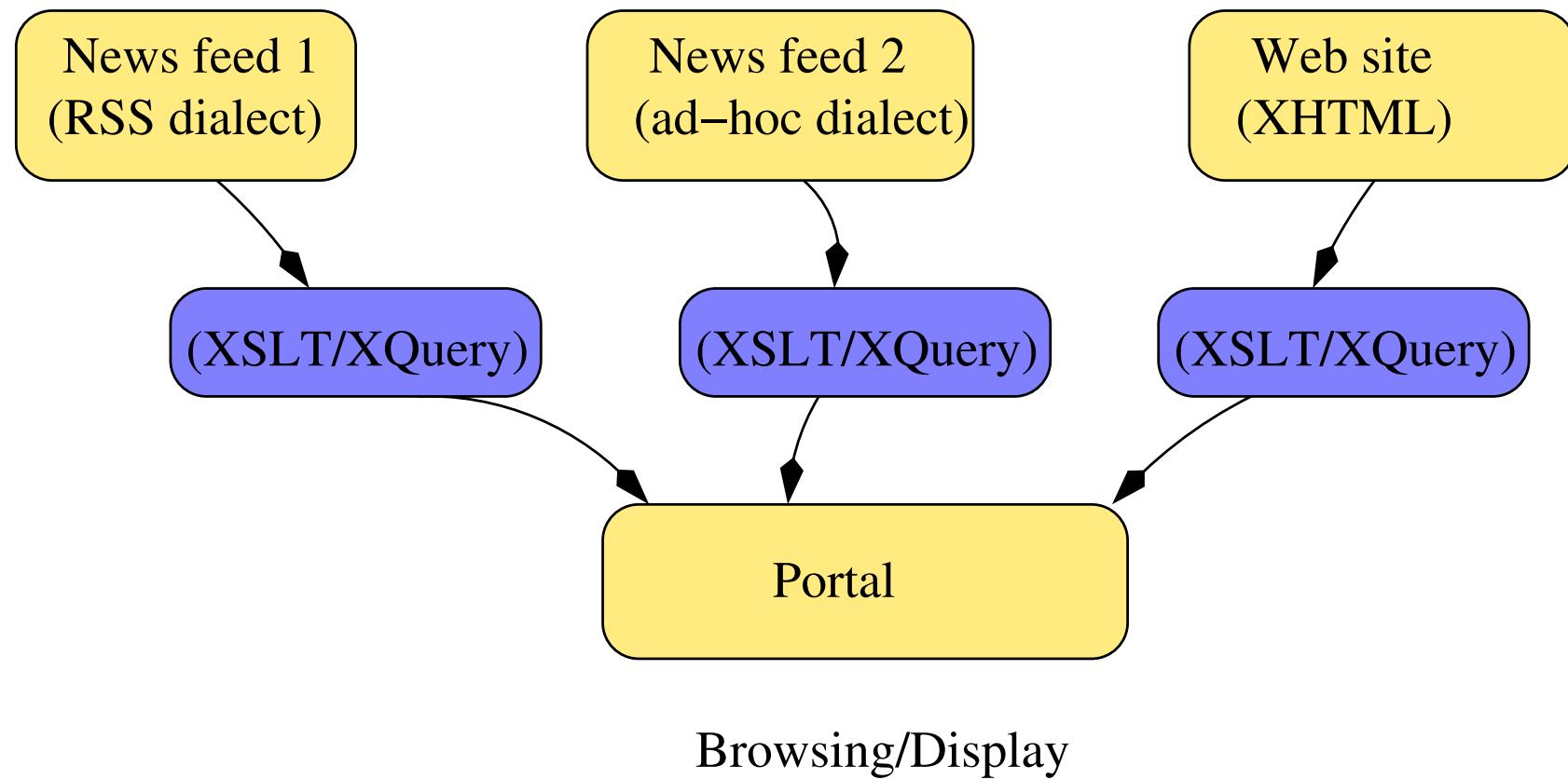
The same content may be published using different means:

- Web publishing: XML \Rightarrow XHTML
- WAP (Wireless Application Protocol): XML \Rightarrow WML

Web publishing, the big picture



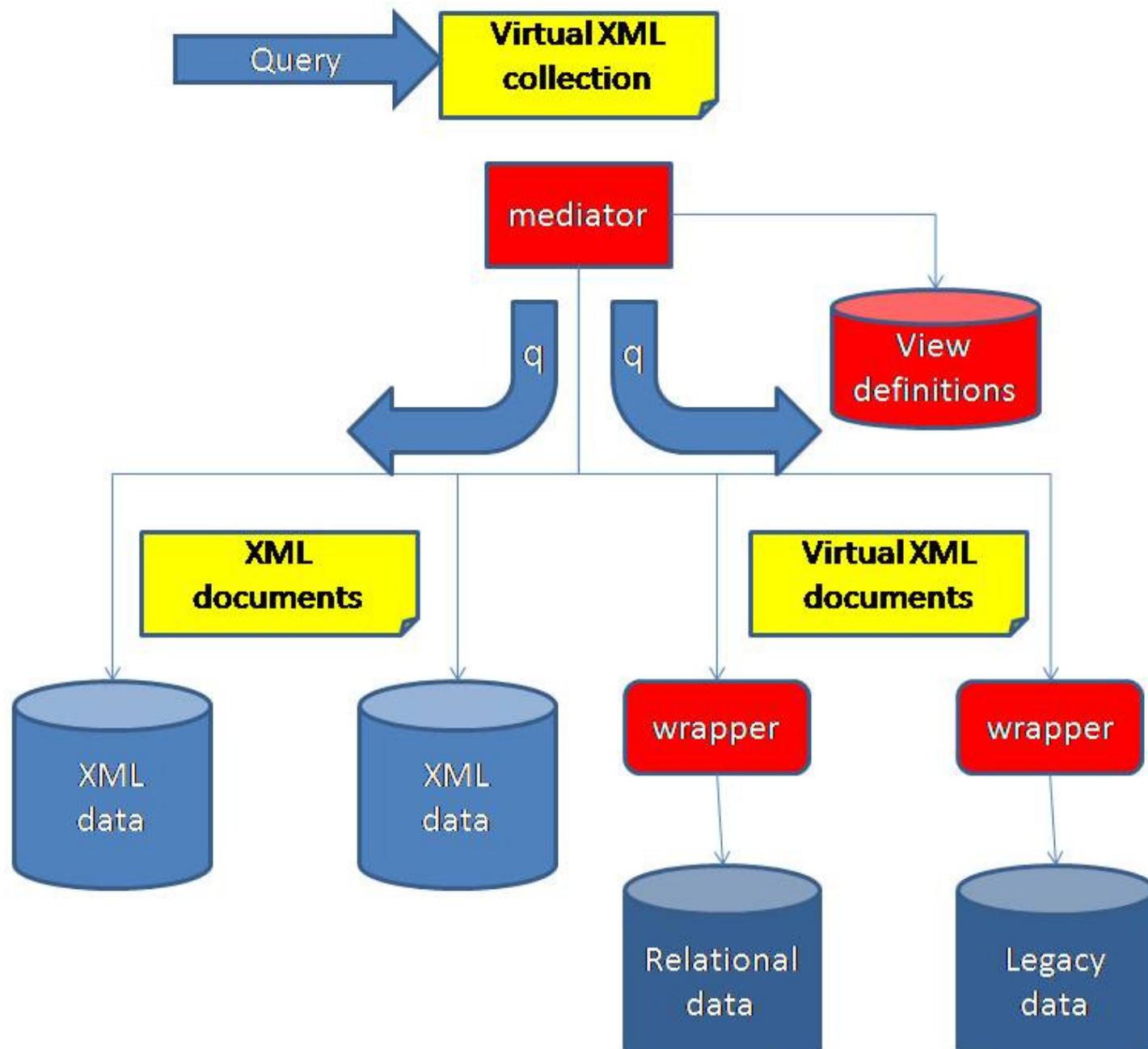
Web Integration: gluing together heterogeneous sources



The **portal** receives (possibly continuously) XML-structured content, each source using its own dialect.

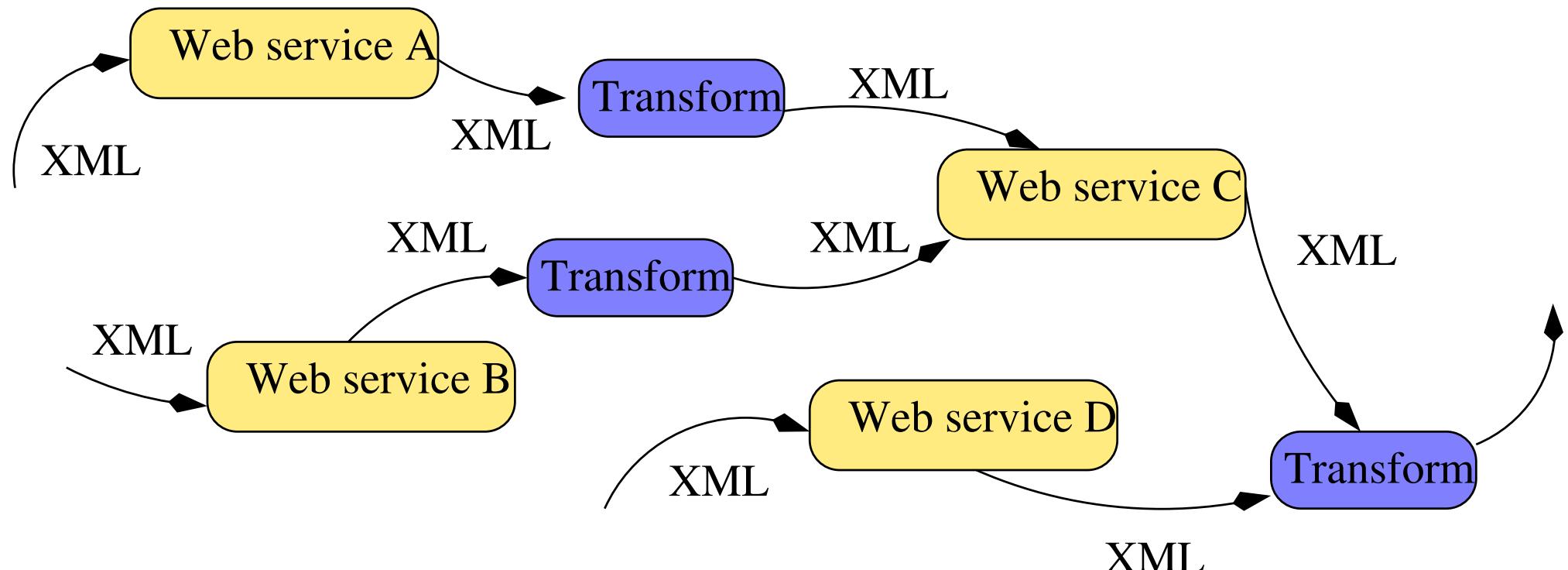
Each feed provides some content, extracted with XSLT or XQuery, or any convenient XML processing tool (e.g., SAX).

Data integration, a larger perspective



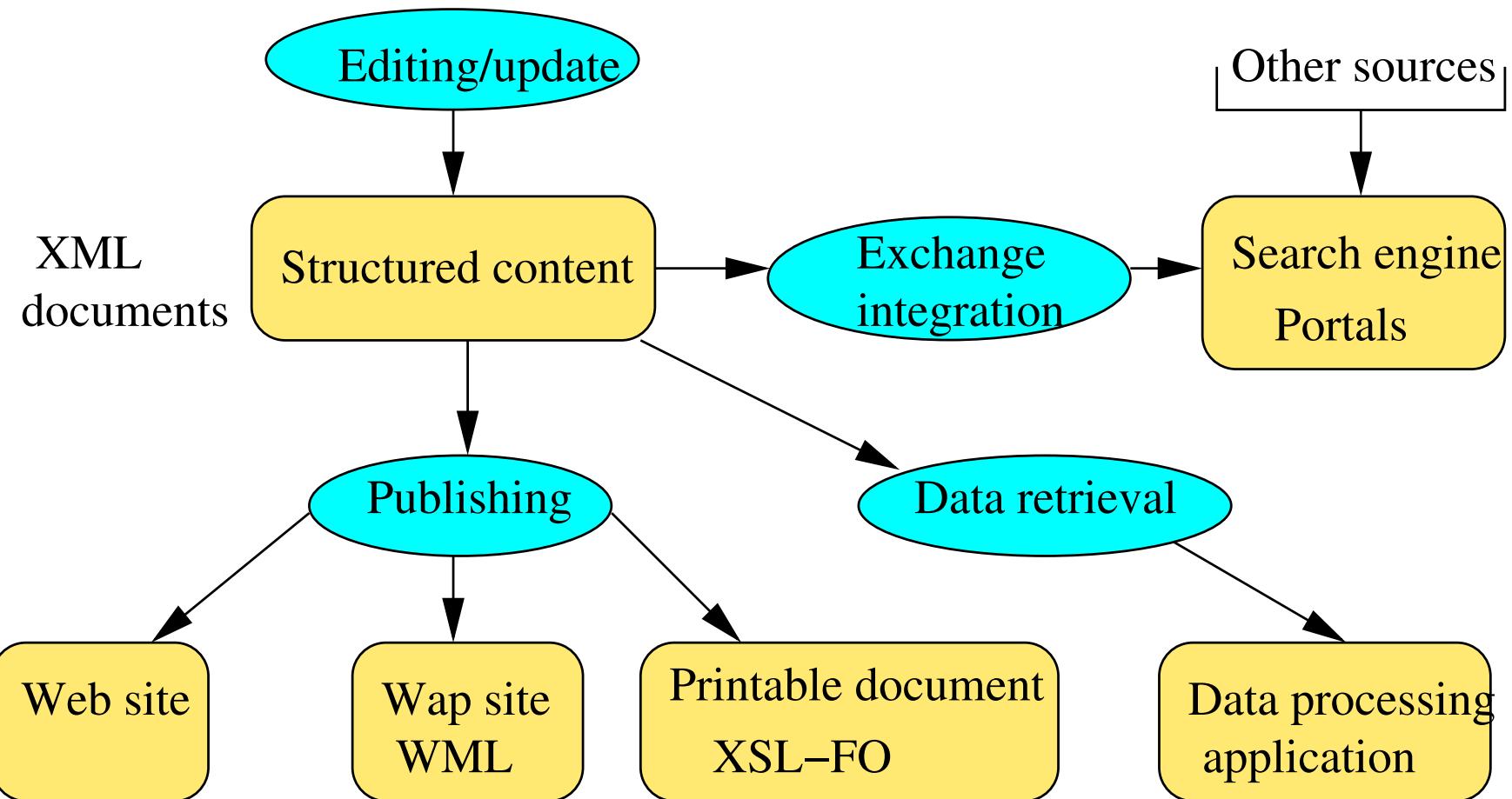
Distributed Data Management with XML

XML encoding is used to *exchange* information between applications.



A specific dialect, WDSL, is used to describe Web Services Interfaces.

Exploiting XML documents: the big picture



... and many other possible use of XML flexibility.

Bibliography

Document Object Model. w3.org/DOM.

World wide web consortium. w3.org/.

Extensible Markup Language. w3.org/XML.

XML Schema. w3.org/XML/Schema.

XML Query (XQuery). w3.org/XML/Query.

Extensible Stylesheet Language. w3.org/Style/XSL.

S. Abiteboul, P. Buneman, and D. Suciu. Data on the Web: From Relations to Semistructured Data and XML. Morgan-Kaufman, New York, 1999.
A. Michard, XML - langage et applications, Eyrolles, 2000.

Table des matières

- 1 Introduction
- 2 Typage de données XML
- 3 Navigation avec XPath
- 4 Transformations XSLT
- 5 Programmation avec XQuery
- 6 APIs pour la lecture des fichiers XML

Définir des DTD

DTD

= **grammaire** pour la structure des documents

= un ensemble de **règles**,
chacune d'entre-elles décrivant le **contenu autorisé** d'un
élément ou l'ensemble des attributs existant pour un
élément.

Comment lier une DTD à un document XML

Une DTD peut être associée de 3 façons à un document XML :

- 1.**DTD interne** : toutes les règles sont dans le fichier XML.
- 2.**DTD mixte** : certaines règles sont décrites dans un fichier spécifique et certaines règles sont dans le fichier XML.
- 3.**DTD externe** : toutes les règles à respecter sont décrites dans un fichier spécifique.

DTD externe

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bonjour SYSTEM "bonjour.dtd">
<bonjour>Hello world!</bonjour>
```

DTD externe

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bonjour SYSTEM
"http://www.chez-moi.fr/dtd/bonjour.dtd">
<bonjour>Hello world!</bonjour>
```

DTD externe

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11-strict.dtd">
<html>
    <head>
        ...
    </head>
    <body>
        ...
    </body>
</html>
```

DTD interne

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bonjour
 [ <!ELEMENT bonjour (#PCDATA)>]>
<bonjour>bonjour tout le monde</bonjour>
```

DTD mixte

```
<?xml version="1.0"?>
<!DOCTYPE bonjour SYSTEM "bonjour.dtd"
[<!ELEMENT bonjour (#PCDATA)>]>
<bonjour>bonjour tout le monde</bonjour>
```

Validation d'un fichier XML

A partir du moment où une DTD est associée au document à valider, on peut **valider** à l'aide :

- d'un **logiciel spécialisé** dans le traitement des documents **XML** (*XMLSpy, Editix, Eclipse, . . .*)
- par **programme** en utilisant les bibliothèques de traitement de **XML** disponibles dans beaucoup de langages (*java, php , perl, ...*)

Le fichier `XMLParser.java`

```
package labd ;

import org.xml.sax.XMLReader ;
import org.xml.sax.helpers.XMLReaderFactory ;

/*
 * Analyseur XML pour une validation par rapport a une DTD.
 * La validation se fait à la volée, en lisant le document.
 * C'est un analyseur SAX -> On ne construit pas l'arbre DOM du document.
 */
public class XMLParser {

    /**
     * Methode de validation : Executer "java XMLParser leDocumentAValider.xml"
     * On vérifie que le document est conforme a la DTD qui lui est liée
     *
     *@param args         ligne de commande = le nom du fichier XML à valider
     *@exception Exception Si probleme lors de la creation des objets.
    */
    public static void main(String[] args) {
        try {
            XMLReader saxReader = XMLReaderFactory.createXMLReader(); //comme avant
            saxReader.setFeature("http://xml.org/sax/features/validation", true); // c'est la la nouveauté
            //saxReader.setContentHandler(new MonHandlerAMoi());// si on veut
            saxReader.parse(args[0]);
        } catch (Exception t) {
            t.printStackTrace();
        }
    }
}
```

Structure d'une DTD

Une DTD contient :

- des déclarations d'**éléments**,
 - des déclarations d'**attributs**,
 - des déclarations d'**entités**,
 - des **commentaires**
- <!-- comme dans les documents XML -->

Déclaration d'élément

<!ELEMENT nom modèle>

- ELEMENT (en **majuscule**) est un mot clef,
- nom est un nom **valide** d'élément,
- modèle est le **modèle de contenu** de l'élément.
 - vide** l'élément n'a pas de contenu (mais peut avoir des attributs)
 - libre** le contenu de l'élément est un contenu quelconque bien formé
 - données** l'élément contient du texte
 - éléments** l'élément est composé d'autre éléments (ses fils)
 - mixte** l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

<!ELEMENT nom EMPTY>

- ELEMENT (en majuscule) est un mot clef,
- nom est un nom valide d'élément,
- modèle est le **modèle de contenu** de l'élément.
 - vide l'élément n'a pas de contenu (mais peut avoir des attributs)
 - libre le contenu de l'élément est un contenu quelconque bien formé
 - données l'élément contient du texte
 - éléments l'élément est composé d'autre éléments (ses fils)
 - mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

<!ELEMENT nom ANY>

- ELEMENT (en majuscule) est un mot clef,
- nom est un nom valide d'élément,
- modèle est le **modèle de contenu** de l'élément.
 - vide l'élément n'a pas de contenu (mais peut avoir des attributs)
 - libre le contenu de l'élément est un contenu quelconque bien formé
 - données l'élément contient du texte
 - éléments l'élément est composé d'autre éléments (ses fils)
 - mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

```
<!ELEMENT nom (#PCDATA)>
```

- ELEMENT (en majuscule) est un mot clef,
- nom est un nom valide d'élément,
- modèle est le **modèle de contenu** de l'élément.
 - vide l'élément n'a pas de contenu (mais peut avoir des attributs)
 - libre le contenu de l'élément est un contenu quelconque bien formé
 - données l'élément contient du texte
 - éléments l'élément est composé d'autre éléments (ses fils)
 - mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

<!ELEMENT nom modèle>

- ELEMENT (en **majuscule**) est un mot clef,
- nom est un nom **valide** d'élément,
- modèle est le **modèle de contenu** de l'élément.
 - vide l'élément n'a pas de contenu (mais peut avoir des attributs)
 - libre le contenu de l'élément est un contenu quelconque bien formé
 - données l'élément contient du texte
 - éléments l'élément est composé d'autre éléments (ses fils)
 - mixte l'élément contient un mélange de texte et de sous-éléments

Modèle de contenu d'élément

On définit le contenu à l'aide d'une **expression régulière** de sous-éléments :

- **séquence**

```
<!ELEMENT chapitre (titre,intro,section)>
```

- **choix**

```
<!ELEMENT chapitre (titre,intro,(section|sections))>
```

- **indicateurs d'occurrence** * (0-n) + (1-n) ? (0-1)

```
<!ELEMENT chapitre (titre,intro?,section+)>
```

```
<!ELEMENT section (titre-section,texte-section)+>
```

```
<!ELEMENT texte-section (p|f)*>
```

Contenu mixte

Une seule façon de mélanger texte #PCDATA et des sous-éléments est acceptée : #PCDATA doit être le premier membre d'un choix placé sous une étoile.

```
<!ELEMENT p (#PCDATA | em | exposant | indice | renvoi ) *>
```

Exemple

```
<!ELEMENT catalogue ( stage )*>
<!ELEMENT stage ( intitule , prerequis ?)>
<!ELEMENT intitule(#PCDATA)>
<!ELEMENT prerequis (#PCDATA | xref )*>
<!ELEMENT xref EMPTY>
```

Exemple

```
<catalogue>
  <stage>
    <intitule>XML et les bases de données</intitule>
    <prerequis>
      connaitre les langages SQL et HTML
    </prerequis>
  </stage>
  <stage>
    <intitule>XML programmation</intitule>
    <prerequis>
      avoir suivi le stage de XML et les bases de données
    </prerequis>
  </stage>
</catalogue>
```

Exemple

```
<catalogue>
  <stage>
    <intitule>XML et les bases de données</intitule>
  </stage>
  <stage>
    <intitule>XML programmation</intitule>
    <prerequis>
      avoir suivi le stage de XML et les bases de données
    </prerequis>
  </stage>
</catalogue>
```

Exemple

```
<catalogue>
</catalogue>
```

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE stage SYSTEM "./dtd1.dtd">
<stage>
  <intitule>T1</intitule>
  <prerequis>T2</prerequis>
</stage>

<!-- dtd1.dtd ci-dessous -->

<!ELEMENT stage ((intitule*| prerequis),(intitule*| prerequis)*)>
<!ELEMENT intitule (#PCDATA)>
<!ELEMENT prerequis (#PCDATA)>
```

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE stage SYSTEM "./dtd2.dtd">
<stage>
  <intitule>T1</intitule>
  <prerequis>T2</prerequis>
</stage>

<!-- dtd2.dtd ci-dessous -->

<!ELEMENT stage (intitule*|prerequis)+>
<!ELEMENT intitule (#PCDATA)>
<!ELEMENT prerequis (#PCDATA)>
```

Déclarations d'attributs

```
<!ATTLIST element nom-attribut1 type1 default1  
          nom-attribut2 type2 default2  
          ...>
```

Le type d'un attribut définit les valeurs qu'il peut prendre

- **CDATA** : valeur chaîne de caractères,
- **ID**, **IDREF**, **IDREFS** permettent de définir des références à l'intérieur du document,
- Une **liste de choix** possibles parmi un ensemble de noms symboliques.

Déclarations d'attributs

```
<!ATTLIST element nom-attribut1 type1 default1  
          nom-attribut2 type2 default2  
          ...>
```

La déclaration par défaut peut prendre quatre formes :

- la valeur par défaut de l'attribut,
- **#REQUIRED** indique que l'attribut est obligatoire,
- **#IMPLIED** indique que l'attribut est optionnel,
- **#FIXED valeur** indique que l'attribut prend toujours la même valeur, dans toute instance de l'élément si l'attribut y apparaît.

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA "1.0">
```

```
<document version="1.0" >  
  ...  
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA "1.0">
```

```
<document version="2.0" >  
  ...  
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA "1.0">
```

```
<document>
  ...
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA #FIXED "1.0">
```

```
<document version="1.0" >  
  ...  
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA #FIXED "1.0">
```

```
<document version="2.0" >  
  ...  
</document>
```

KO

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA #FIXED "1.0">
```

```
<document>
  ...
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom  
          titre (Mlle|Mme|M.) #REQUIRED  
          nom-epouse CDATA #IMPLIED  
>
```

```
<nom titre="Mme" nom-epouse="Lenoir">  
    Martin  
</nom>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom  
          titre (Mlle|Mme|M.) #REQUIRED  
          nom-epouse CDATA #IMPLIED  
>
```

```
<nom titre="M." nom-epouse="Lenoir">  
  Martin  
</nom>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom  
          titre (Mlle|Mme|M.) #REQUIRED  
          nom-epouse CDATA #IMPLIED  
>
```

```
<nom titre="M.">  
  Martin  
</nom>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom  
          titre (Mlle|Mme|M.) #REQUIRED  
          nom-epouse CDATA #IMPLIED  
>
```

```
<nom titre="Madame" nom-epouse="Lenoir">  
    Martin  
</nom>
```

KO

Attributs **ID**, **IDREF**, **IDREFS**

- Un attribut **ID** sert à référencer un élément, la valeur de cette référence pouvant être rappelée dans des attributs **IDREF** ou **IDREFS**.
- Un élément ne peut avoir au plus qu'un attribut **ID** et la valeur associée doit être unique dans le document XML. Cette valeur doit être un **nom XML** (donc pas un nombre).
- La valeur de défaut pour un attribut **ID** est obligatoirement **#REQUIRED** ou **#IMPLIED**
- Une valeur utilisée dans un attribut **IDREF** ou **IDREFS** doit obligatoirement correspondre à celle d'un attribut **ID**.

Exemples ID, IDREF, IDREFS

```
<!ELEMENT document (personne*,livre*)>
<!ELEMENT personne (nom , prenom)>
  <!ATTLIST personne id ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT livre (#PCDATA)>
  <!ATTLIST livre auteur IDREF #IMPLIED>
```

document.dtd

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
  <personne id="id-1">
    <nom>Dupond</nom>
    <prenom>Martin</prenom>
  </personne>
  <personne id="id-2">
    <nom>Durand</nom>
    <prenom>Helmut</prenom>
  </personne>
  <livre auteur="id-1">Ma vie, mon oeuvre</livre>
</document>
```

OK

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
  <personne id="id-1">
    <nom>Dupond</nom>
    <prenom>Martin</prenom>
  </personne>
  <personne id="id-1">
    <nom>Hardailepick</nom>
    <prenom>Helmut</prenom>
  </personne>
  <livre auteur="id-1">Ma vie, mon oeuvre</livre>
</document>
```

KO

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
  <personne id="id-1">
    <nom>Gaisellepick</nom>
    <prenom>Elmer</prenom>
  </personne>
  <personne id="id-2">
    <nom>Hardailepick</nom>
    <prenom>Helmut</prenom>
  </personne>
  <livre auteur="id-3">Ma vie, mon oeuvre</livre>
</document>
```

KO

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
    <personne id="id-1">
        <nom>Gaisellepick</nom>
        <prenom>Elmer</prenom>
    </personne>
    <personne id="id-2">
        <nom>Hardailepick</nom>
        <prenom>Helmut</prenom>
    </personne>
    <livre auteur="id-1 id-2">Ma vie, mon oeuvre</livre>
</document>
```

KO

Exemples ID, IDREF, IDREFS

```
<!ELEMENT document (personne*,livre*)>
<!ELEMENT personne (nom , prenom)>
  <!ATTLIST personne id ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT livre (#PCDATA)>
  <!ATTLIST livre auteur IDREFS #IMPLIED>
```

document.dtd

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
    <personne id="id-1">
        <nom>Gaisellepick</nom>
        <prenom>Elmer</prenom>
    </personne>
    <personne id="id-2">
        <nom>Hardailepick</nom>
        <prenom>Helmut</prenom>
    </personne>
    <livre auteur="id-1 id-2">Ma vie, mon oeuvre</livre>
</document>
```

OK

Déclarations d'entités

- Les **entités internes** : «macros exportées» qui sont utilisées dans le document XML validé par la DTD.

```
<!ENTITY euro "&#8364;">, utilisation &euro;
```

- Les **entités paramétriques** : «macros non exportées» qui sont utilisées ailleurs dans la DTD.

```
<!ENTITY % editeur "O'Reilly">, utilisation %editeur;
```

- Les **entités externes** : «macros importées» définies dans un autre document, utilisables dans la DTD elle-même ou dans tout document XML valide pour la DTD.

```
<!ENTITY % HTMLlat1 PUBLIC  
      "-//W3C//ENTITIES Latin 1 for XHTML//EN"  
      "http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent">
```

Exemple d'entités

```
<!-- entite externe pour importer les entites -->
<!-- representant les caracteres accentues -->
<!ENTITY % HTMLlat1 PUBLIC
  "-//W3C//ENTITIES Latin 1 for XHTML//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent">
%HTMLlat1; <!-- c'est comme un import -->
<!-- entite parametrique -->
<!ENTITY % elt "(#PCDATA|elt1)*" >
<!ELEMENT racine %elt;>
<!ELEMENT elt1 (#PCDATA)>
<!--entites interne -->
<!ENTITY euro "&#8364;">
<!ENTITY LILLE1 "Universit&eacute; Lille 1">
<!--l'utilisation du &eacute; est possible parce que -->
<!--c'est une entité externe importée à l'aide de %HTMLlat1 -->
```

Exemple de document valide pour cette DTD

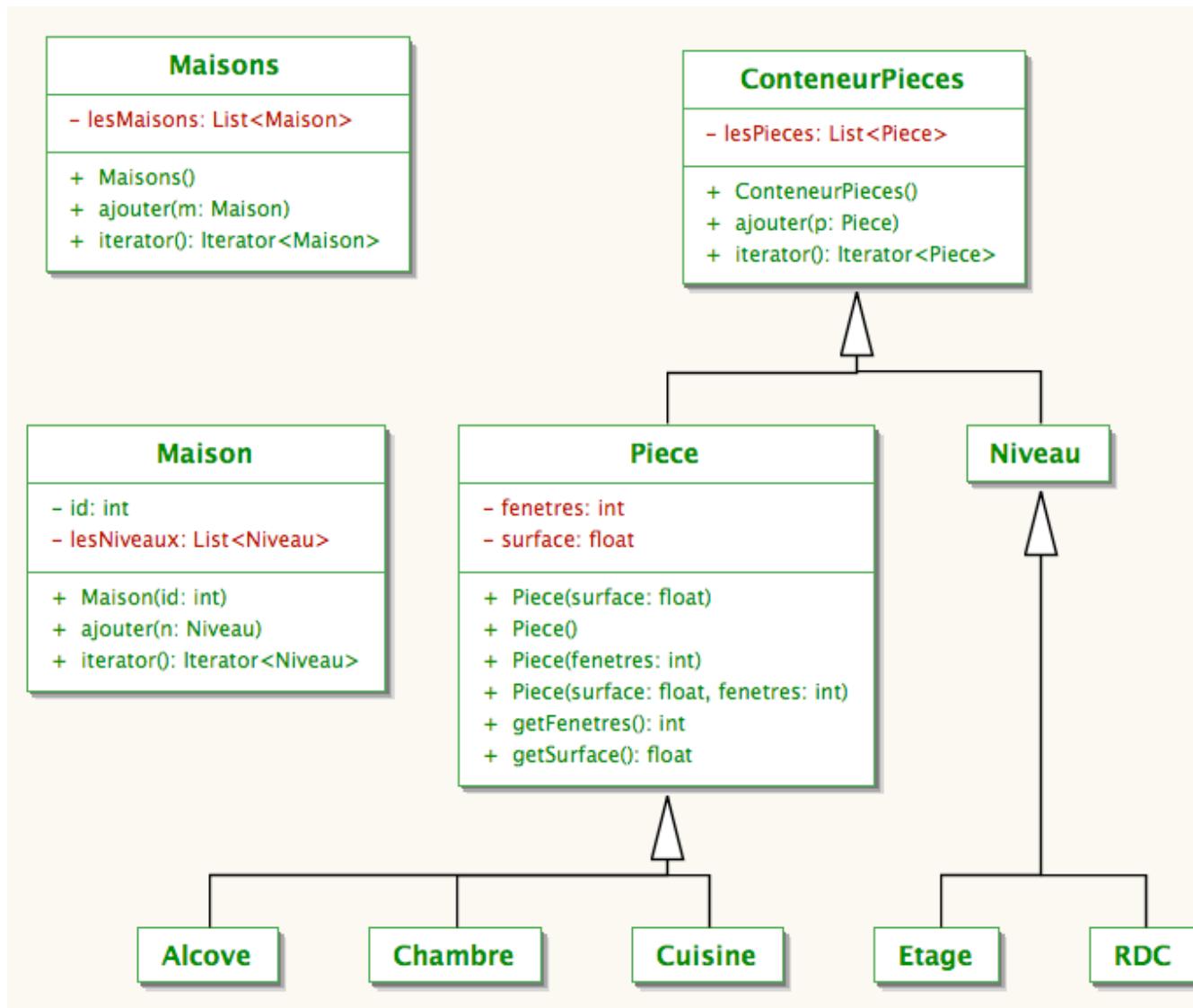
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE racine SYSTEM "./entites.dtd">
<racine>
    blabla
    <elt1>
        Universit&eacute; : &LILLE1;
    </elt1>
    <elt1>
        10000 &euro;;
    </elt1>
    c'est fini !
</racine>
```

XML Schema

maisons.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<maisons>
    <maison id="1">
        <RDC>
            <cuisine surface-m2="12">Evier Inox. Mobilier encastré</cuisine>
            <WC>Lavabo.</WC>
            <séjour surface-m2="38">Cheminée en pierre. Baie vitrée</séjour>
            <bureau surface-m2="14">Bibliothèque</bureau>
            <garage/>
        </RDC>
        <étage>
            <terrasse/>
            <chambre surface-m2="28" fenetre="3">
                <alcove surface-m2="8"/>
            </chambre>
            <chambre surface-m2="18"/>
            <salledeBain surface-m2="15">Douche, baignoire, lavabo</salledeBain>
        </étage>
    </maison>
    <maison id="2">
        <RDC>
            <cuisine surface-m2="12">en ruine</cuisine>
            <garage/>
        </RDC>
        <étage>
            <mirador surface-m2="1">Vue sur la mer</mirador>
            <salledeBain surface-m2="15">Douche</salledeBain>
        </étage>
    </maison>
    <maison id="3">
```

Typage



Comparaison entre DTD et XML Schema

- DTD
 - Essentiellement, définition de l'**imbrication des éléments**, et **définition des attributs**.
 - Types pauvres
 - Pas de gestion des espaces de nom
 - Pas beaucoup de contraintes sur le contenu d'un document
- XML-Schema
 - Notion de **type**, indépendamment de la notion d'élément
 - **Contraintes d'intégrité d'entité et d'intégrité référentielle**, plus précises que les ID/IDREF des DTD.
 - Contraintes de **cardinalité**
 - Gestion des **espaces de noms**
 - **Réutilisation** de mêmes types d'attributs pour des éléments différents
 - Format **XML**

Lier un schéma à un document

Un peu comme pour une DTD

La **balise ouvrante de l'élément racine** du fichier XML contient des informations sur le schéma.

```
<maisons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="maisons.xsd">
```

Lier un schéma à un document

Le schéma `maisons.xsd` est lui-même un fichier XML et doit donc être associé au schéma qui définit ce qu'on peut utiliser dans un schéma !

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="maisons">
    ...
  </xsd:element>
  ...
</xsd:schema>
```

Contenu d'un schéma

Un XML-Schema est composé de

- Définitions de **types**
- Déclaration **d'attributs**
- Déclaration **d'éléments**
- Définitions de **groupes d'attributs**
- Définitions de **groupes de modèles**
- Définitions de **contraintes d'unicité ou de clés**
- ...

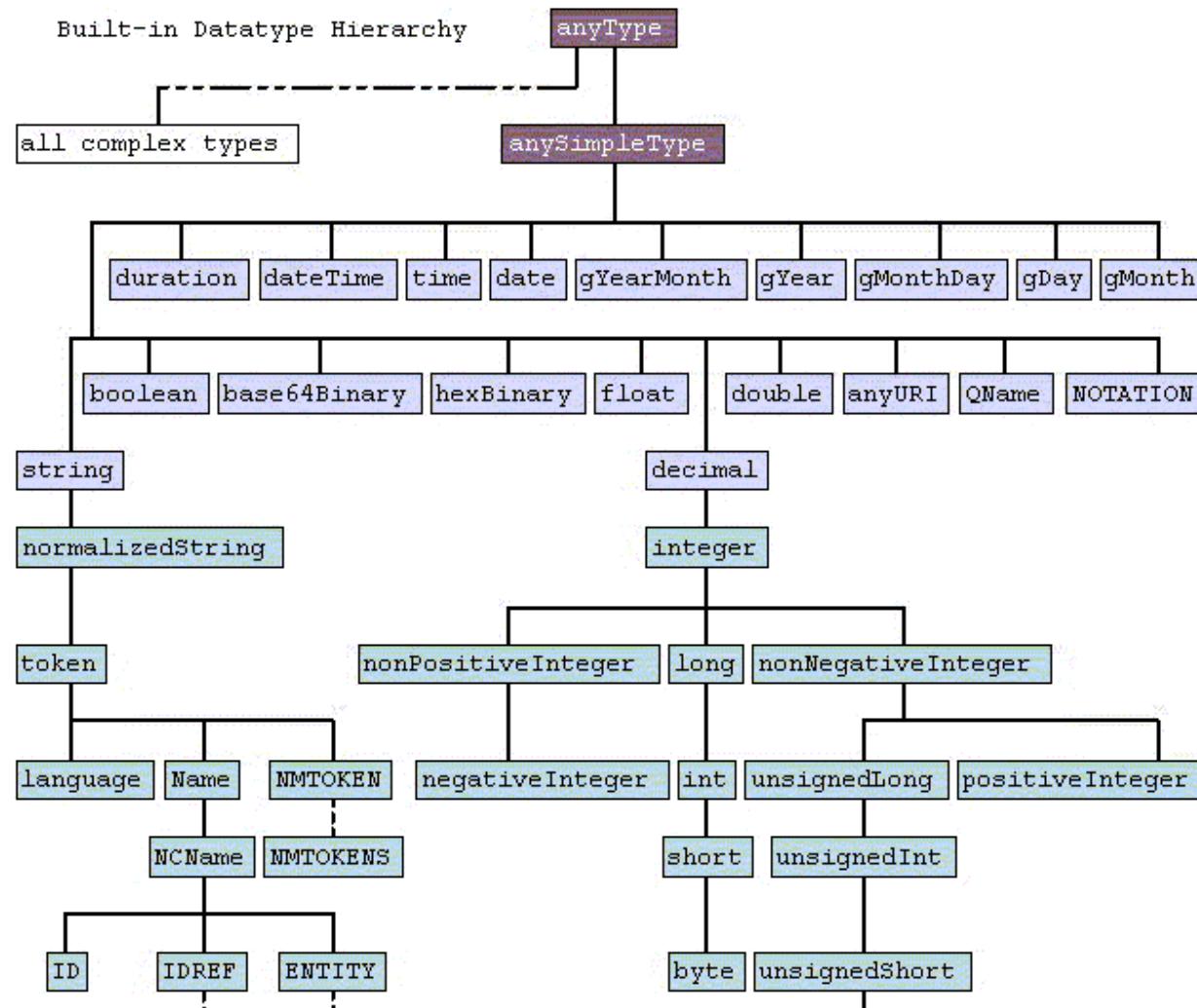
Définitions de types

Typage

- Existence de **types prédéfinis** : hiérarchie de types, dont la racine est le type **anyType**.
- Possibilité de définir de **nouveaux types**
- Distinction **types simples** et **types complexes**
 1. Les types simples sont utilisés pour les déclarations **d'attributs**, les déclarations d'**éléments** dont le **contenu** se limite à des **données atomiques**, et qui n'ont **pas d'attributs**.
 2. Les types complexes s'utilisent dans tous les autres cas.

Les types simples prédéfinis

<http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes#built-in-datatypes>



Exemple d'utilisation d'un type simple prédéfini

```
<xsd:attribute name="surface-m2" type="xsd:decimal"/>  
  
<xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
```

Les types simples

Un type simple est caractérisé par 3 ensembles : son **espace de valeurs**, son **espace lexical**, ses **facettes**.

1.espace de valeurs = les **valeurs autorisées** pour ce type

2.espace lexical = **syntaxe des littéraux**. Par exemple 100 et 1.0E2 sont deux littéraux qui représentent la même valeur, de type **float**.

3.facette = **propriété disponible sur ce type**. Toutes les facettes ne s'appliquent pas à tous les types.

Facettes

Les **facettes fondamentales** définissent le **type de données**. Il en existe 5 disponibles sur tous les types :

1.**equal**

2.**ordered** : **false**, **partial** ou **ordered**, indique si l'espace de valeurs est ordonné

3.**bounded** : valeur booléenne, indique si l'espace de valeurs est borné

4.**cardinality** : indique si l'espace de valeur est **finite** ou **countably infinite**

5.**numeric** : valeur booléenne, indique si l'espace de valeurs est numérique

Facettes de contrainte

Les facettes de contraintes sont optionnelles et permettent de restreindre l'espace des valeurs. Elles ne sont pas toutes disponibles sur tous les types.

- 1.**length** : **longueur**, qui peut être le nombre de caractères pour un type **string**, le nombre d'éléments pour un type **list**, le nombre d'octets pour un type binaire.
- 2.**maxLength** : longueur maximale
- 3.**minLength** : longueur minimale
- 4.**pattern** : expression régulière qui **décrit les littéraux** du type (donc les valeurs)
- 5.**maxExclusive** : valeur **maximale au sens strict** (<)
- 6.**maxInclusive** : valeur **maximale au sens large** (\leq)
- 7.**minExclusive** : valeur **minimale au sens strict**
- 8.**minInclusive** : valeur **minimale au sens large**

Facettes de contrainte

9.`enumeration` : énumération des **valeurs possibles**

10.`fractionDigits` : **nombre maximal de décimales** après le point

11.`totalDigits` : **nombre maximal de chiffres** pour une valeur décimale

12.`whiteSpace` : **règle** pour la **normalisation des espaces** dans une chaîne

Les types simples créés par l'utilisateur

On dérive un type simple **à partir d'un autre type**. Il existe **3** façons de dériver un type simple :

1. par **restriction** : on crée un type dont l'espace de valeurs est inclus dans l'espace de valeurs d'un type existant. Pour cela, **on utilise les facettes** pour restreindre l'espace des valeurs,

2. par **liste**,

3. par **union**.

Exemples (XHTML) de restrictions d'un type atomique

```
<xs:simpleType name="tabindexNumber">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="0" />
    <xs:maxInclusive value="32767" />
  </xs:restriction>
</xs:simpleType>
```

Exemples de restrictions d'un type atomique

```
<xsd:simpleType name="jour-de-la-semaine">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="lundi"/>
    <xsd:enumeration value="mardi"/>
    <xsd:enumeration value="mercredi"/>
    <xsd:enumeration value="jeudi"/>
    <xsd:enumeration value="vendredi"/>
    <xsd:enumeration value="samedi"/>
    <xsd:enumeration value="dimanche"/>
  </xsd:restriction>
</xsd:simpleType>
```

Exemples (XHTML) de restrictions d'un type atomique

```
<!-- single or comma-separated list of media descriptors -->
<xs:simpleType name="MediaDesc">
  <xs:restriction base="xs:string">
    <xs:pattern value="[^, ]+(, \s*[ ^, ]+)*" />
  </xs:restriction>
</xs:simpleType>
```

Exemple de type union

```
<!-- permet de faire <font size="34"> ou <font size="medium">-->

<xsd:simpleType name="fontSize">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base="xsd:positiveInteger">
        <xsd:minInclusive value="8"/>
        <xsd:maxInclusive value="72"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="small"/>
        <xsd:enumeration value="medium"/>
        <xsd:enumeration value="large"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Exemples de types list

```
<xsd:simpleType name="sizes">
  <xsd:list itemType="xsd:decimal"/>
</xsd:simpleType>
```

Permet d'écrire <dimensions>12 5.6 67</dimensions>

```
<xsd:simpleType name='listOfString'>
  <xsd:list itemType='xsd:string'/>
</xsd:simpleType>
```

Attention, la liste ci-dessous, de type `listOfString`, a 18 éléments, pas 3 (l'espace est un séparateur)

```
<someElement>
this is not list item 1
this is not list item 2
this is not list item 3
</someElement>
```

Exemple de restriction d'un type list

```
<xs:simpleType name='myList'>
  <xs:list itemType='xs:integer' />
</xs:simpleType>

<xs:simpleType name='myRestrictedList'>
  <xs:restriction base='myList'>
    <xs:pattern value='123 (\d+\s)*456' />
  </xs:restriction>
</xs:simpleType>

<someElement>123 456</someElement>
<someElement>123 987 456</someElement>
<someElement>123 987 567 456</someElement>
```

Les types complexes

Type complexe

- La définition d'un type complexe est un **ensemble de déclarations d'attributs** et un **type de contenu**.
- Type complexe **à contenu simple** : type d'un élément dont le contenu est de **type simple mais qui possède un attribut** (un type simple n'a pas d'attribut). On le définit par extension d'un type simple par ajout d'un attribut.
- Type complexe **à contenu complexe** : permet de déclarer **des sous-éléments et des attributs**. On le construit à partir de rien ou on le définit par **restriction** d'un type complexe, ou par **extension** d'un type.

Exemple de type complexe à contenu simple

```
<xs:complexType name="TypePiece">
  <xs:simpleContent>
    <xs:extension base="xsd:string">
      <xsd:attribute name="surface-m2" type="xsd:decimal"/>
      <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<!-- exemple de déclaration d'élément chambre
                  de type TypePiece -->
<xs:element name="chambre" type="TypePiece"/>

<!-- exemple d'élément chambre -->
<chambre surface-m2="17.5">Exposition plein sud</chambre>
```

Exemples de types complexes à contenu complexe (1)

Construit à partir de zéro.

```
<xsd:complexType name="type-duree">
  <xsd:sequence>
    <xsd:element name="du" type="xsd:date"/>
    <xsd:element name="au" type="xsd:date"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="duree" type="type-duree"/>

<duree>
  <du>2010-09-13</du>
  <au>2010-09-25</au>
</duree>
```

Exemples de types complexes à contenu complexe (1)

Construit à partir de zéro.

```
<xs:complexType name="personName">
  <xs:sequence>
    <xs:element name="title" minOccurs="0" />
    <xs:element name="forename" minOccurs="0"
                maxOccurs="unbounded" />
    <xs:element name="surname" />
  </xs:sequence>
</xs:complexType>
```

Exemple de type complexe à contenu complexe (2)

Obtenu par extension d'un type simple ou complexe

```
<xs:complexType name="extendedName">
  <xs:complexContent>
    <xs:extension base="personName">
      <xs:sequence>
        <xs:element name="generation" minOccurs="0" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="addressee" type="extendedName"/>

<addressee>
  <forename>Albert</forename>
  <forename>Arnold</forename>
  <surname>Gore</surname>
  <generation>Jr</generation>
</addressee>
```

Exemple de type complexe à contenu complexe (3)

Obtenu également par restriction d'un type complexe

```
<xs:complexType name="simpleName">
  <xs:complexContent>
    <xs:restriction base="personName">
      <xs:sequence>
        <xs:element name="forename" minOccurs="1" maxOccurs="1"/>
        <xs:element name="surname" />
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="who" type="simpleName"/>

<who>
  <forename>Barack</forename>
  <surname>Obama</surname>
</who>
```

Groupe de Modèles

Pour construire un type complexe, on peut utiliser des **constructeurs de groupes de modèles**. On a déjà rencontré le constructeur **sequence**, il existe **3** constructeurs de groupes de modèles :

- **sequence** : les éléments d'une séquence doivent apparaître **dans l'ordre où ils sont déclarés**.
- **choice** : **un seul élément parmi ceux du choice** doit apparaître
- **all** : les éléments contenus dans un **all** peuvent apparaître dans **n'importe quel ordre**.

Exemple d'utilisation de choice

```
<xsd:complexType name="SurfaceOuVolume">
  <xsd:choice>
    <xsd:element name="surface" type="length0"/>
    <xsd:element name="volume" type="length0"/>
  </xsd:choice>
</xsd:complexType>

<!-- element occupation du type SurfaceouVolume -->
<occupation>
  <surface>453</surface>
</occupation>
```

Exemple d'utilisation de all

```
<xsd:complexType name="Identite">
  <xsd:all>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string"/>
    <xsd:element name="datenaiss" type="xsd:date"/>
  </xsd:all>
</xsd:complexType>
```

Exemple d'utilisation de all

```
<!-- elements de type Identite -->
<identite>
    <nom>meurisse</nom>
    <prenom>paul</prenom>
    <datenaiss>1912-12-21</datenaiss>
</identite>

<identite>
    <datenaiss>1948-05-31</datenaiss>
    <nom>bonham</nom>
    <prenom>john</prenom>
</identite>
```

Déclaration d'attribut

- Un **attribut** est de **type simple**, par exemple un type prédéfini, une énumération, une liste ...
- On peut préciser le caractère **obligatoire ou facultatif** de l'attribut (**attribut use**)
- On peut lui donner une **valeur par défaut** (**attribut default**) **ou une valeur fixe** (**attribut fixed**)

Exemples de déclarations d'attributs

```
<xsd:attribute name="size"
                type="fontSize"
                use="required"/>

<xs:attribute name="jour" default="lundi"
               type="jour-de-la-semaine"/>

<xs:attribute name="version"
               type="xs:number"
               fixed="1.0"/>
```

Déclaration d'élément

- On définit le contenu de l'élément grâce aux types
- Lorsqu'on n'attribue **pas de type** à un élément, il est considéré comme de type **xs:anyType** et peut donc contenir n'importe quoi
- Pour un **élément de contenu mixte** (texte et sous-éléments), on utilise l'attribut **mixed** de **xs:complexType**.
- Pour un **élément vide**, on définit un type complexe **qui n'a pas de sous-élément**

Exemple 1 : éléments de contenu simple

```
<xsd:element name="long" type="length0"/>

<xsd:element name="nom" type="xsd:string"/>

<xsd:element name="font">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="size" type="fontSize"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

Exemple 1 : éléments de contenu simple

Permet d'écrire dans le document XML

```
<long unit="cm">45</long>  
  
<nom>durand</nom>  
  
<font size="medium">machin</font>
```

Exemple 2 : élément de contenu mixte

```
<xsd:element name="toto">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element ref="font" minOccurs="1" maxOccurs="2"/>
      <xsd:element ref="long"/>
      <xsd:element name="long" type="length1"/>
      <xsd:element name="media" type="MediaDesc"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Exemple 2 : instance du schéma précédent

```
<toto>
  <font size="45">blabla</font>
  contenu mixte donc on peut avoir du texte
  <font size="medium">machin</font>
  <!-- les 2 éléments long qui suivent -->
  <!-- ne sont pas du même type -->
  <!-- c'est impossible pour une DTD -->
  <long unit="cm">45</long>
  <long><size>34</size><unit>cm</unit></long>
  et patati et patata
  <media>screen</media>
</toto>
```

Exemple 3 : élément vide

L'élément `br` en XHTML contient uniquement des attributs. Les types utilisés sont définis dans le schéma de XHTML.

```
<xs:element name="br">
  <xs:complexType>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute name="class" type="xs:NMTOKENS" />
    <xs:attribute name="style" type="StyleSheet" />
    <xs:attribute name="title" type="Text" />
  </xs:complexType>
</xs:element>
```

```
<xsd:element name="vide">
  <xsd:complexType/>
</xsd:element>
```

Eléments, Attributs et Types

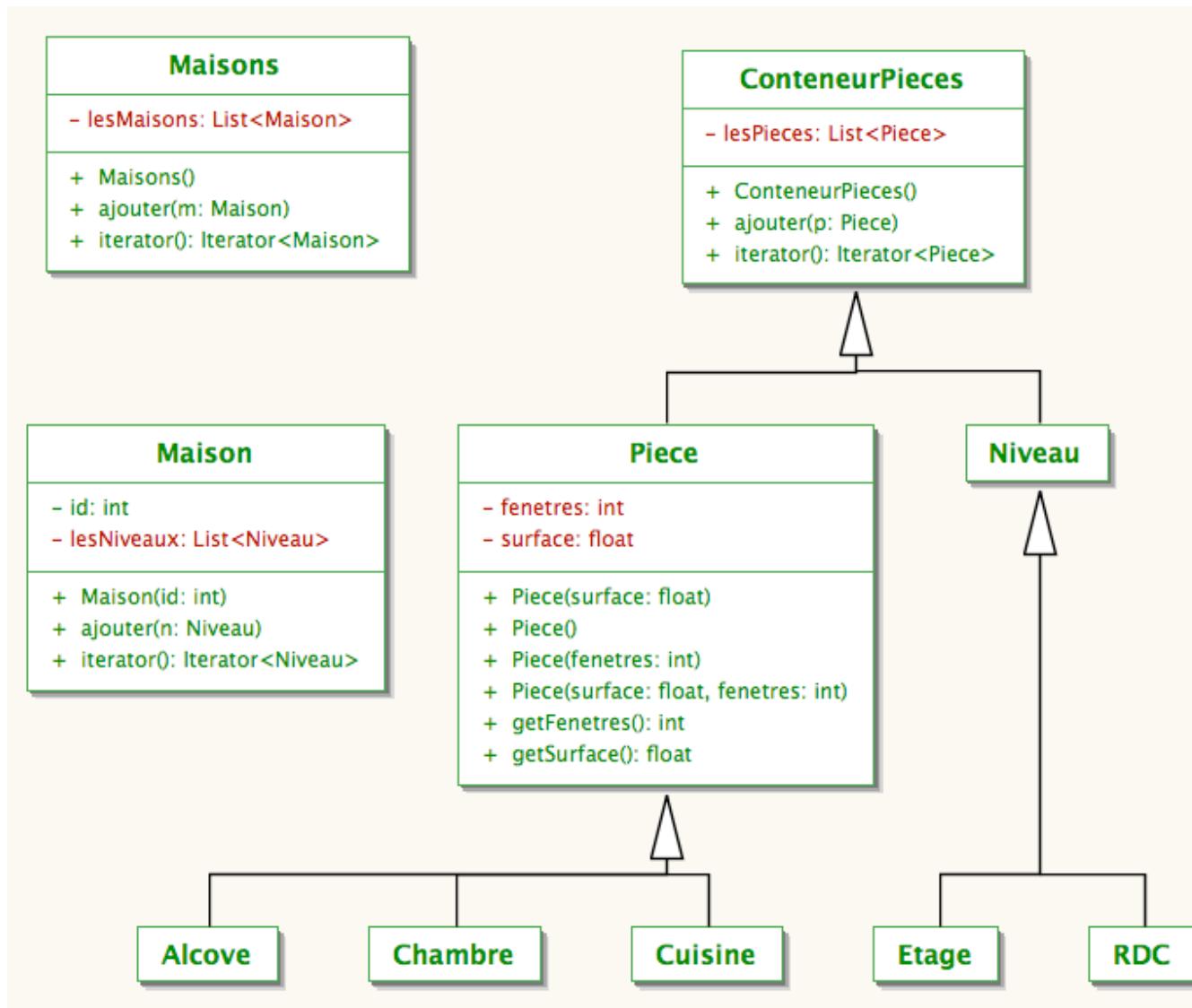
- On doit associer les types et les noms d'éléments ou d'attributs.
- Un type peut contenir des déclarations d'éléments ou d'attributs
- Un élément ou attribut peut contenir une déclaration de type
- On distingue : déclaration locale (dans une autre déclaration) ou globale (fils de la racine `xs:schema`)
- On peut faire référence à un type, élément, attribut déjà défini grâce à l'attribut `ref`

Eléments, Attributs et Types

```
<xsd:element name="trimestre">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="type-duree">
        <xsd:attribute name="num" type="type-trimestre"
                      use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name="trimestres">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="trimestre" minOccurs="4" maxOccurs="4" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Typage



maisons.xsd

```
<xsd:element name="maison">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="étage" type="TypeNiveau"/>
        <xsd:element name="RDC" type="TypeNiveau"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:positiveInteger"
      use="required" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="maisons">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="maison"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

maisons.xsd

```
<xsd:complexType name="TypeNiveau">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="alcove" type="TypePiece"/>
      <xsd:element name="cuisine" type="TypePiece"/>
      <xsd:element name="séjour" type="TypePiece"/>
      <xsd:element name="bureau" type="TypePiece"/>
      <xsd:element name="garage" type="TypePiece"/>
      <xsd:element name="terrasse" type="TypePiece"/>
      <xsd:element name="chambre" type="TypePiece"/>
      <xsd:element name="salledeBain" type="TypePiece"/>
      <xsd:element name="mirador" type="TypePiece"/>
      <xsd:element name="WC" type="TypePiece"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

maisons.xsd

```
<xsd:complexType name="TypePiece" mixed="true">
  <xsd:complexContent>
    <xsd:extension base="TypeNiveau">
      <xsd:attribute name="surface-m2" type="xsd:decimal"/>
      <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Impossible car un type dont l'attribut **mixed** vaut **true**
ne peut pas étendre un type dont l'attribut **mixed** vaut
false

maisons.xsd

```
<xsd:complexType name="TypePiece" mixed="true">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="alcove" type="TypePiece"/>
      <xsd:element name="cuisine" type="TypePiece"/>
      <xsd:element name="séjour" type="TypePiece"/>
      <xsd:element name="bureau" type="TypePiece"/>
      <xsd:element name="garage" type="TypePiece"/>
      <xsd:element name="terrasse" type="TypePiece"/>
      <xsd:element name="chambre" type="TypePiece"/>
      <xsd:element name="salledeBain" type="TypePiece"/>
      <xsd:element name="mirador" type="TypePiece"/>
      <xsd:element name="WC" type="TypePiece"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="surface-m2" type="xsd:decimal"/>
  <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
</xsd:complexType>
```

Lourd et redondant avec le type TypeNiveau

xsd:group

```
<xsd:group name="TypeConteneurPieces">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="alcove" type="TypePiece"/>
      <xsd:element name="cuisine" type="TypePiece"/>
      <xsd:element name="séjour" type="TypePiece"/>
      <xsd:element name="bureau" type="TypePiece"/>
      <xsd:element name="garage" type="TypePiece"/>
      <xsd:element name="terrasse" type="TypePiece"/>
      <xsd:element name="chambre" type="TypePiece"/>
      <xsd:element name="salledeBain" type="TypePiece"/>
      <xsd:element name="mirador" type="TypePiece"/>
      <xsd:element name="WC" type="TypePiece"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:group>
```

Utilisation d'un groupe de modèle nommé

xsd:group

```
<xsd:complexType name="TypeNiveau">
    <xsd:group ref="TypeConteneurPieces" />
</xsd:complexType>

<xsd:complexType name="TypePiece" mixed="true">
    <xsd:group ref="TypeConteneurPieces" />
    <xsd:attribute name="surface-m2" type="xsd:decimal" />
    <xsd:attribute name="fenetre" type="xsd:positiveInteger" />
</xsd:complexType>
```

Utilisation d'un groupe de **modèle nommé**

maisons.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:group name="TypeConteneurPieces">
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="alcove" type="TypePiece"/>
        <xsd:element name="cuisine" type="TypePiece"/>
        <xsd:element name="séjour" type="TypePiece"/>
        <xsd:element name="bureau" type="TypePiece"/>
        <xsd:element name="garage" type="TypePiece"/>
        <xsd:element name="terrasse" type="TypePiece"/>
        <xsd:element name="chambre" type="TypePiece"/>
        <xsd:element name="salledeBain" type="TypePiece"/>
        <xsd:element name="mirador" type="TypePiece"/>
        <xsd:element name="WC" type="TypePiece"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:group>
```

maisons.xsd

```
<xsd:complexType name="TypeNiveau">
    <xsd:group ref="TypeConteneurPieces" />
</xsd:complexType>

<xsd:complexType name="TypePiece" mixed="true">
    <xsd:group ref="TypeConteneurPieces" />
    <xsd:attribute name="surface-m2" type="xsd:decimal" />
    <xsd:attribute name="fenetre" type="xsd:positiveInteger" />
</xsd:complexType>

<xsd:element name="maisons">
    <xsd:complexType>
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="maison" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

maisons.xsd

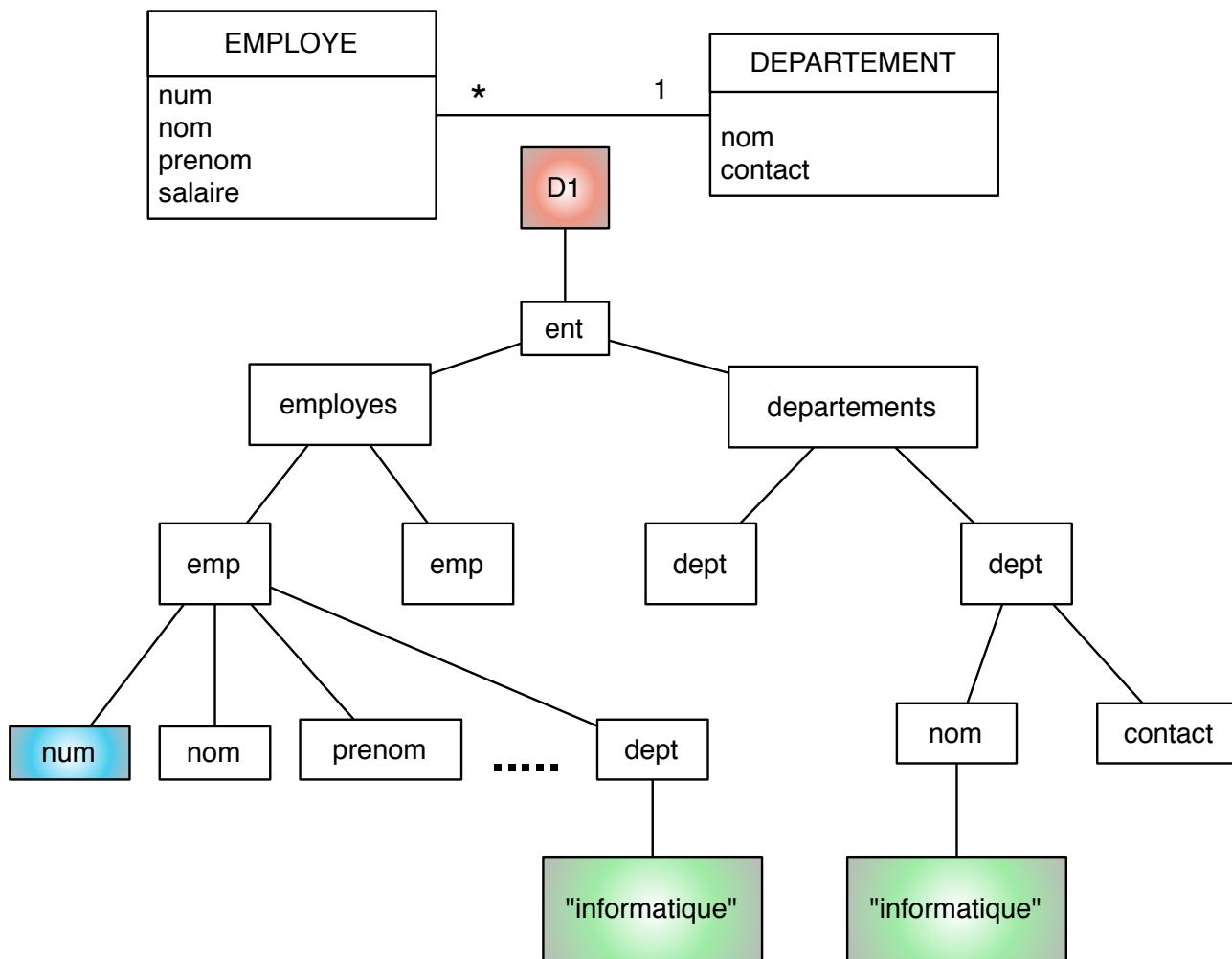
```
<xsd:element name="maison">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="étage" type="TypeNiveau"/>
        <xsd:element name="RDC" type="TypeNiveau"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:positiveInteger"
      use="required"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Contraintes d'intégrité en XML-Schema

- Modèle relationnel :
 - Contrainte d'unicité **UNIQUE**, de **clef primaire** (unicité et existence) **PRIMARY KEY**, de **clef étrangère FOREIGN KEY ... REFERENCES**
 - Une clef primaire est définie pour une relation, et elle composée d'un ou plusieurs attributs.
 - Une clef étrangère fait référence à des attributs d'une relation précise.
- Avec une **DTD**, on peut définir des identifiants (attribut de type **ID**), et des références d'identifiant (de type **IDREF**). L'existence ou non est définie en choisissant **IMPLIED** ou **REQUIRED**. Mais
 - les références ne sont pas typées (on ne sait pas à quel type de nœud on fait référence)
 - un **identifiant est global** au document
- **XML-Schema** : on se rapproche du modèle relationnel

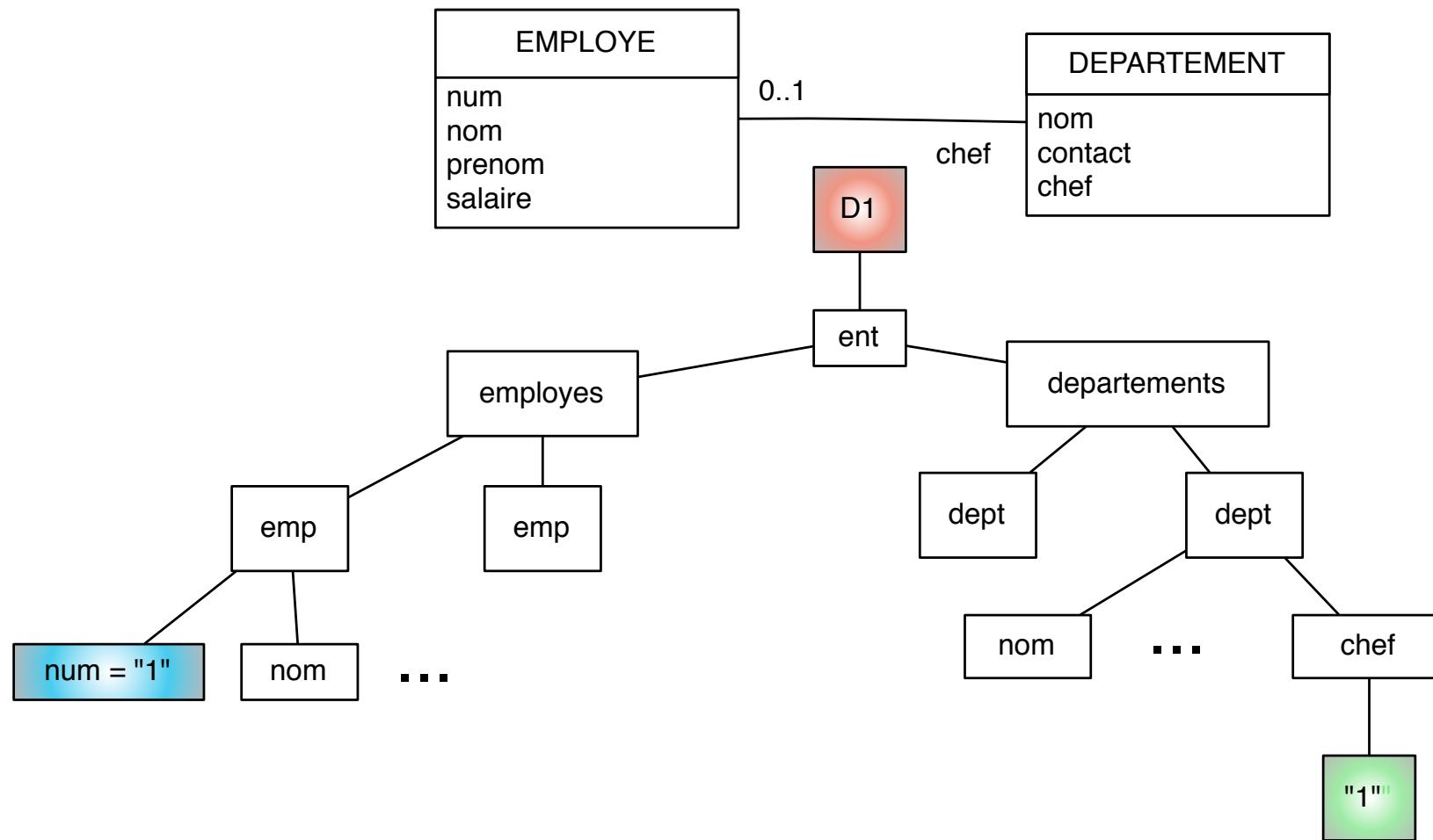
Exemple

Tout employé appartient à un département :



Exemple (suite)

Un département a au plus un chef qui est employé par l'entreprise



Exemple (suite)

On déclare un type pour un département, et pour une séquence de départements

```
<xsd:complexType name="TypeDept">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="contact" type="xsd:string"/>
    <xsd:element name="chef" type="xsd:int" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="SeqDept">
  <xsd:sequence>
    <xsd:element name="dept" type="TypeDept" maxOccurs="unbounded"
      minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Exemple (suite)

Même chose pour les employés.

```
<xsd:complexType name="TypeEmploye">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string"/>
    <xsd:element name="salaire" type="xsd:decimal"/>
    <xsd:element name="dept" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="num" type="xsd:int"/>
</xsd:complexType>
```

Exemple (suite)

```
<xsd:complexType name="SeqEmploye">
  <xsd:sequence>
    <xsd:element name="emp"
      type="TypeEmploye"
      maxOccurs="unbounded"
      minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
```

Exemple (suite)

Déclaration des éléments `employes` et `departements` avec les clefs primaires pour les éléments `emp` et `dept`.

```
<xsd:element name="employes" type="SeqEmploye">
  <xsd:key name="clefEmp">
    <xsd:selector xpath="emp"/>
    <xsd:field xpath="@num"/>
  </xsd:key>
</xsd:element>

<xsd:element name="departements" type="SeqDept">
  <xsd:key name="clefDept">
    <xsd:selector xpath="dept"/>
    <xsd:field xpath="nom"/>
  </xsd:key>
</xsd:element>
```

Exemple (suite)

L'élément racine du document avec les clefs étrangères.

```
<xsd:element name="ent">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="employes" />
      <xsd:element ref="departements" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:keyref name="refDept" refer="clefDept">
    <xsd:selector xpath="employes/emp" />
    <xsd:field xpath="dept" />
  </xsd:keyref>
  <xsd:keyref name="refEmp" refer="clefEmp">
    <xsd:selector xpath="departements/dept" />
    <xsd:field xpath="chef" />
  </xsd:keyref>
</xsd:element>
```

Explications

- L'endroit où l'on déclare une contrainte **détermine la zone où elle doit être vérifiée**. En effet, la contrainte s'applique à l'intérieur de **l'élément "contexte"** de cette contrainte.
- Le **selector** détermine **pour quel élément c'est une clef**. C'est un chemin **XPath** qui désigne un ensemble de noeuds (appelés noeuds cibles) contenus dans l'élément contexte de la contrainte.
- Les **field** qui suivent donnent **les composants** (attributs ou éléments) **de la clef**. Chaque composant **désigne un noeud unique** (élément ou attribut), par rapport à 1 noeud cible désigné par le **selector**. De plus, chaque composant doit être **de type simple**.

Explications

- La syntaxe des chemins XPath est réduite, voir la norme pour plus de précision.
- Dans une clef étrangère, l'attribut `refer` indique à quelle clef primaire elle fait référence.
- Pour définir une **contrainte d'unicité**, remplacer `xsd:key` par `xsd:unique`.

Espaces de noms

Motivations

- mélanger différents **vocabulaires** et éviter les conflits de nom
 - modularité, **réutilisation**
 - **exporter les définitions** d'un schéma
- ➡ utilisation de **noms qualifiés** (= préfixés) pour les éléments et les attributs.

Identification d'un espace de noms

Un **espace de noms** est identifié par une adresse **IRI**, (Internationalized Resource Identifier)

- **IRI** = extension des **URI** permettant l'utilisation de caractères internationaux (par ex. **UTF-8**) dans l'adresse elle-même.

`http://www.exemple.org/clés`

- **URI** = URN ou URL
 - **URN** : `urn:isbn-0-395-36341-1`
 - **URL** : `ftp://ftpperso.free.fr/xml`

Remarques

- Le W3C **déconseille l'usage d'une IRI relative** comme identifiant d'espace de nom
- L'analyse d'un identifiant d'espace de nom **tient compte de la casse et ne prend pas en compte la résolution de l'IRI**. Tous les exemples suivants représentent des identifiants différents :

`http://www.Example.org/wine`

`http://www.example.org/Wine`

`http://www.example.org/rosé`

`http://www.example.org/ros%c3%a9`

`http://www.example.org/ros%c3%A9`

Finalement en pratique

Identifiant d'espace de nom = **URL absolue avec des caractères ASCII.**

`http://www.w3.org/XML/1998/namespace`

`http://www.w3.org/1999/xhtml`

`http://www.w3.org/2001/XMLSchema`

`http://www.w3.org/2001/XMLSchema-instance`

`http://www.w3.org/1999/XSL/Transform`

`http://xml.insee.fr/schema/`

`http://fil.univ-lille1.fr/miage-fa-fc`

Déclaration d'un espace de noms

Le **préfixe** qui désigne un espace de noms doit avoir été déclaré, grâce à un **pseudo attribut** qui commence par **xmlns** :

- Les préfixes **xml** et **xmlns** sont réservés.
- La déclaration se fait **dans la balise ouvrante** d'un élément
- Lorsqu'on déclare un espace de noms, le préfixe est **applicable dès la balise ouvrante** où se fait la déclaration, **et pour tout le contenu de cet élément**, sauf si le même préfixe est utilisé plus bas pour un autre espace de noms.
- L'utilisation du préfixe pour un élément (ou attribut) indique que cet élément (ou attribut) appartient à l'espace de noms associé au préfixe. (nom qualifié)
- On peut déclarer **plusieurs espaces de noms** dans une même balise ouvrante.

Déclaration d'un espace de noms

- La déclaration se fait dans la balise ouvrante d'un élément
- Lorsqu'on déclare un espace de noms, le préfixe est applicable dès la balise ouvrante où se fait la déclaration, et pour tout le contenu de cet élément.

```
<x xmlns:edi="http://ecom.example.org/schema">  
    ....  
</x>
```

Déclaration d'un espace de noms

- La déclaration se fait dans la balise ouvrante d'un élément
- Lorsqu'on déclare un espace de noms, le préfixe est applicable dès la balise ouvrante où se fait la déclaration, et pour tout le contenu de cet élément.

```
<edi:price xmlns:edi="http://ecom.exple.org/sch" units="Euro">  
  32.18  
</edi:price>
```

Déclaration d'un espace de noms

- La déclaration se fait dans la balise ouvrante d'un élément
- Lorsqu'on déclare un espace de noms, le préfixe est applicable dès la balise ouvrante où se fait la déclaration, et pour tout le contenu de cet élément.

```
<x xmlns:edi="http://ecom.exple.org/schema">
    <lineItem edi:taxClass="exempt">
        Baby food
    </lineItem>
</x>
```

Déclaration d'un espace de noms

- La déclaration se fait dans la balise ouvrante d'un élément
- Lorsqu'on déclare un espace de noms, le préfixe est applicable dès la balise ouvrante où se fait la déclaration, et pour tout le contenu de cet élément, **sauf si le même préfixe est utilisé plus bas pour un autre espace de noms.**

```
<x xmlns:edi="http://ecom.exple.org/sch1">
  <edi:a>
    <edi:b xmlns:edi="http://ecom.exple.org/sch2">
      <edi:a>
        <edi:c/>
      </edi:a>
    </edi:b>
    <edi:b/>
  </edi:a>
</x>
```

Déclaration d'un espace de noms

On peut déclarer **plusieurs espaces de noms** dans une même balise ouvrante.

```
<bk:book  xmlns:bk="urn:loc.gov:books"
            xmlns:isbn="urn:ISBN:0-395-36341-6">
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
</bk:book>
```

Espace de noms par défaut

- Si on utilise l'attribut `xmlns` (sans `:`), on définit alors un **espace de noms par défaut**, pour lequel il n'existe **pas de préfixe associé**. L'espace de nom par défaut **ne s'applique pas aux attributs**.

```
<?xml version="1.1"?>
<!-- elements are in the HTML namespace, by default --&gt;
&lt;html xmlns="http://www.w3.org/1999/xhtml"&gt;
  &lt;head&gt;&lt;title&gt;Frobnotication&lt;/title&gt;&lt;/head&gt;
  &lt;body&gt;
    &lt;p&gt;
      Moved to &lt;a href="http://frob.example.com"&gt;here&lt;/a&gt;.
    &lt;/p&gt;
  &lt;/body&gt;
&lt;/html&gt;</pre>
```

Espace de noms par défaut

- Si on utilise l'attribut `xmlns` (sans `:`), on définit alors un espace de noms par défaut, pour lequel il n'existe pas de préfixe associé.

```
<?xml version="1.1"?>
<!-- initially, the default namespace is "books" -->
<book xmlns="urn:loc.gov:books"
      xmlns:isbn="urn:ISBN:0-395-36341-6">
    <title>Cheaper by the Dozen </title>
    <isbn:number>1568491379</isbn:number>
    <notes>
      <!-- make HTML the default namespace for some commentary -->
      <p xmlns="http://www.w3.org/1999/xhtml">
        This is a <i>funny</i> book!
      </p>
    </notes>
</book>
```

Espace de noms par défaut

- Il faut, en général, résERVER l'espace de noms par défaut à l'espace de noms le plus utilisé.
- Tant que l'espace de noms par défaut n'a pas été spécifié, les éléments dont le nom n'est pas qualifié ne font partie d'aucun espace de noms. Leur propriété espace de noms n'a pas de valeur.
- Il est possible de revenir à l'espace de noms par défaut non spécifié en affectant la chaîne vide à l'attribut `xmlns`.
- Les attributs peuvent également avoir des noms qualifiés formés d'un préfixe et d'un nom local. Ils font alors partie de l'espace de noms auquel est associé le préfixe.
- Les attributs dont le nom n'est pas qualifié ne font jamais partie de l'espace de noms par défaut. Cette règle s'applique que l'espace de noms par défaut soit spécifié ou non.

Exercice

```
<?xml version="1.0" encoding="utf-8"?>

<exercice xmlns:pre="http://A">
  <pre:niveau xmlns:pre="http://D" xmlns="http://A">
    <out:garage xmlns:out="http://C">
      <pre:alcove/>
      <entree xmlns:pre="http://E">
        <pre:cuisine/>
      </entree>
    </out:garage>
  </pre:niveau>
  <def xmlns="http://B">
    <pre:figue>
      <levain/>
    </pre:figue>
  </def>
</exercice>
```

exercice ->
niveau ->
garage ->
alcove ->
entree ->
cuisine ->
def ->
figue ->
levain ->

Exporter un espace de noms

attribut targetNameSpace

- Pour **exporter** (créer) **un espace de nom** dans un schéma, on utilise l'attribut **targetNameSpace** de la balise **xsd:schema**

```
<?xml version="1.0"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.w3.org/2001/XMLSchema
         http://www.w3.org/2001/XMLSchema.xsd"
    targetNamespace="http://fil.univ-lille1.fr/miage-fa-fc">
    <!-- description de la miage fa fc -->
    ...

```

Exporter un espace de noms

```
<?xml version="1.0" encoding="utf-8"?>
<miage-fa-fc
    xmlns="http://fil.univ-lille1.fr/miage-fa-fc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://fil.univ-lille1.fr/miage-fa-fc http://
www.fil.univ-lille1.fr/FORMATIONS/MIAGE-FC-FA/schemas/miage-
fa-fc.xsd"
    annee="2011">
    <creneaux>
        <creneau>
            <trimestre>T1</trimestre>
            <jour>lundi</jour>
            <de>09:00:00</de>
            <a>12:00:00</a>
            <salle>M5-A2</salle>
            ...
        </creneau>
    </creneaux>
</miage-fa-fc>
```

Espaces de noms et schémas

attribut `elementFormDefault`

Quand on exporte un espace de noms,

- Les déclarations globales appartiennent à l'espace de nom
- Les déclarations locales n'appartiennent pas à l'espace de nom, sauf si on ajoute l'attribut `elementFormDefault="qualified"`
- on dispose aussi de l'attribut `attributeFormDefault`

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:schemaLocation="http://www.w3.org/2001/XMLSchema
                      http://www.w3.org/2001/XMLSchema.xsd"
  xmlns="http://fil.univ-lille1.fr/miage-fa-fc"
  targetNamespace="http://fil.univ-lille1.fr/miage-fa-fc"
  elementFormDefault="qualified">
  <!-- description de la miage fa fc -->
```

Inclusions de schémas

On peut **assembler** plusieurs composants de schémas (définitions de types, déclarations d'éléments, ...), provenant de plusieurs documents.

- élément **include** qui permet d'inclure les définitions provenant d'autres schémas mais pas de plusieurs espaces de noms.
- Les schémas inclus doivent avoir
 1. soit **le même espace de noms cible** que le document qui les inclut
 2. soit **pas d'espace de noms**, dans ce cas, c'est l'espace de noms du schéma qui inclut tous les autres qui est pris en compte.

Exemple d'inclusion sans espace de noms cible

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:include schemaLocation="dept.xsd"/>
    <xsd:include schemaLocation="emp.xsd"/>
    <xsd:element name="ent">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="employes"/>
                <xsd:element ref="departements"/>
            </xsd:sequence>
        </xsd:complexType>
        ... les clefs étrangères ...
    </xsd:element>
</xsd:schema>
```

- `dept.xsd` et `emp.xsd` sont des fichiers dans le même répertoire.
- `dept.xsd` (resp. `emp.xsd`) contient les déclarations de l'élément `departements`(resp. `employes`) et de tous ses sous-éléments.

Exemple d'instance du schéma précédent

```
<?xml version="1.0" encoding="utf-8"?>
<ent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="entreprise.xsd">
  <employes>
    <emp num="1">
      <nom>toto</nom><prenom>jules</prenom>
      <salaire>3452</salaire>
      <dept>informatique</dept>
    </emp>
  </employes>
  <departements>
    <dept>
      <nom>informatique</nom>
      <contact>Mme Machin 45-76-77-09-54</contact>
      <chef>1</chef>
    </dept>
  </departements>
</ent>
```

Exemple d'inclusion avec espace de noms cible

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.lifl.fr/~yroos/schema"
    targetNamespace="http://www.lifl.fr/~yroos/schema"
    elementFormDefault="qualified"
>
    <xsd:include schemaLocation="dept.xsd"/>
    <xsd:include schemaLocation="emp.xsd"/>

    <xsd:element name="ent">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="employes"/>
                <xsd:element ref="departements"/>
            </xsd:sequence>
        </xsd:complexType>
        ... et les clefs étrangères ...
    </xsd:element>
</xsd:schema>
```

Instance du schéma précédent

```
<?xml version="1.0" encoding="utf-8"?>
<ent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.lifl.fr/~yroos/schema"
      xsi:schemaLocation="http://www.lifl.fr/~yroos/schema
http://saxo.lifl.fr/~yroos/schema/entreprise.xsd">
  <employees>
    <emp num="1">
      <nom>toto</nom> <prenom>jules</prenom>
      <salaire>3452</salaire> <dept>informatique</dept>
    </emp>
  </employees>
  <departements>
    <dept>
      <nom>informatique</nom>
      <contact>Mme Machin 45-76-77-09-54</contact>
      <chef>1</chef>
    </dept>
  </departements>
</ent>
```

Importation de schémas

- Un schéma est associé à un espace de noms cible
- L'élément **import** permet de faire référence à des composants d'un schéma qui appartient à un autre espace de noms que le schéma dans lequel on fait référence à ces composants.
- Dans l'exemple qui suit, on utilise un composant du schéma de **XHTML** pour notre propre schéma.

Exemple d'importation de schémas

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.lifl.fr/~yroos/schema"
  xmlns:art="http://www.lifl.fr/~yroos/schema"
  xmlns:html="http://www.w3.org/1999/xhtml"
  elementFormDefault="qualified"
>

<xsd:import namespace="http://www.w3.org/1999/xhtml"
  schemaLocation=
  "http://www.w3.org/2002/08/xhtml/xhtml1-strict.xsd"/>
...

```

Exemple d'importation de schémas

```
...
<xsd:element name="dansRevue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="auteur" maxOccurs="unbounded"
                    type="xsd:string"/>
      <xsd:element name="revue" type="xsd:string"/>
      <xsd:element name="titre" type="xsd:string"/>
      <xsd:element minOccurs="0" name="resume"
                    type="html:Block"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Exemple d'instance de ce schéma

```
<?xml version="1.0" encoding="utf-8"?>
<bibliographie xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.lifl.fr/~yroos/schema"
    xmlns:html="http://www.w3.org/1999/xhtml"
    xsi:schemaLocation="http://www.lifl.fr/~yroos/schema
    http://saxo.lifl.fr/~yroos/schema/articles.xsd"
>
    <dansRevue>
        <auteur>Tryphon Tournesol</auteur>
        <revue>Revue de Physique</revue>
        <titre>Ma machine à voyager dans le temps</titre>
        <resume><html:div>et patati
            <html:br/>et patata</html:div></resume>
    </dansRevue>
</bibliographie>
```

En conclusion

Bonne pratique

L'utilisation des espaces de noms peut parfois être un peu compliquée. Quand un espace de nom est très clairement majoritaire dans le document XML (que ce soit une instance ou un schéma), on le définit comme espace de nom par défaut, sinon une bonne pratique est de lier (dans un premier temps) tous les espaces de noms à des préfixes et de ne définir un espace de nom par défaut que lorsqu'il y en a un qui se détache (c.a.d. quand on tape majoritairement toujours le même préfixe)

Table des matières

- 1 Introduction
- 2 Typage de données XML
- 3 Navigation avec XPath
- 4 Transformations XSLT
- 5 Programmation avec XQuery
- 6 APIs pour la lecture des fichiers XML

XPath

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

May 30, 2013

Outline

1 Introduction

2 Path Expressions

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

7 Exercise

XPath

- An **expression language** to be used in another host language (e.g., XSLT, XQuery).
- Allows the description of **paths** in an XML tree, and the retrieval of nodes that match these paths.
- Can also be used for performing some (limited) operations on XML data.

Example

`2 * 3` is an XPath **literal expression**.

`// * [@msg="Hello world"]` is an XPath **path expression**, retrieving all elements with a msg attribute set to “Hello world”.

Content of this presentation

Mostly XPath 1.0: a W3C recommendation published in 1999, widely used. Also a *basic* introduction to XPath 2.0, published in 2007.

XPath Data Model

XPath expressions operate over **XML trees**, which consist of the following **node types**:

- **Document**: the **root node** of the XML document;
- **Element**: element nodes;
- **Attribute**: attribute nodes, represented as children of an **Element** node;
- **Text**: text nodes, i.e., leaves of the XML tree.

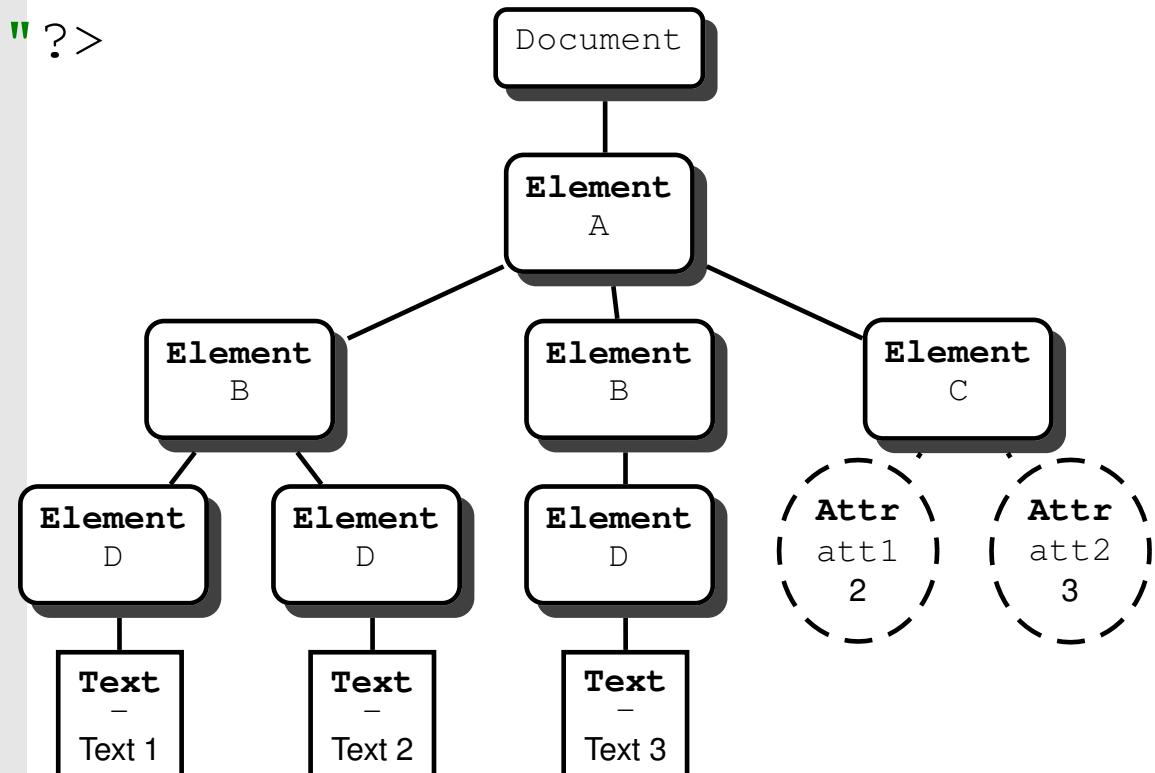
Remark

Remark 1 The XPath data model features also **ProcessingInstruction** and **Comment** node types.

Remark 2 Syntactic features specific to serialized representation (e.g., entities, literal section) are ignored by XPath.

From serialized representation to XML trees

```
<?xml version="1.0"
      encoding="utf-8"?>
<A>
  <B att1='1'>
    <D>Text 1</D>
    <D>Text 2</D>
  </B>
  <B att1='2'>
    <D>Text 3</D>
  </B>
  <C att2="a"
      att3="b"/>
</A>
```



(Some attributes not shown.)

XPath Data Model (cont.)

- The **root node** of an XML tree is the (unique) **Document** node;
- The **root element** is the (unique) **Element** child of the root node;
- A node has a **name**, or a **value**, or both
 - ▶ an **Element** node has a name, but no value;
 - ▶ a **Text** node has a value (a character string), but no name;
 - ▶ an **Attribute** node has both a name and a value.
- *Attributes are special!* Attributes are not considered as first-class nodes in an XML tree. They must be addressed specifically, when needed.

Remark

The expression “*textual value of an Element N*” denotes the concatenation of all the **Text** node values which are descendant of *N*, taken in the **document order**.

Outline

1 Introduction

2 Path Expressions

- Steps and expressions
- Axes and node tests
- Predicates

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

7 Exercises

XPath Context

A step is evaluated in a specific **context** $[N_1, N_2, \dots, N_n, N_c]$ which consists of:

- a **context list** N_1, N_2, \dots, N_n of nodes from the XML tree;
- a **context node** N_c belonging to the context list.

Information on the context

- The **context length** n is a positive integer indicating the **size** of a contextual list of nodes; it can be known by using the function `last()`;
- The **context node position** $c \in [1, n]$ is a positive integer indicating the **position** of the context node in the context list of nodes; it can be known by using the function `position()`.

XPath steps

The basic component of XPath expression are **steps**, of the form:

$$\text{axis} : : \text{node-test} [P_1] [P_2] \dots [P_n]$$

axis is an **axis name** indicating what the direction of the step in the XML tree is (**child** is the default).

node-test is a **node test**, indicating the kind of nodes to select.

P_i is a **predicate**, that is, any XPath expression, evaluated as a boolean, indicating an additional condition. There may be no predicates at all.

Interpretation of a step

A step is evaluated with respect to a **context**, and returns a **node list**.

Example

`descendant::C[@att1='1']` is a step which denotes all the **Element** nodes named C, descendant of the context node, having an **Attribute** node att1 with value 1

Path Expressions

A path expression is of the form: $[/]step_1/step_2/\dots/step_n$

A path that begins with $/$ is an **absolute** path expression;

A path that does not begin with $/$ is a **relative** path expression.

Example

`/A/B` is an **absolute** path expression denoting the **Element** nodes with name B, children of the root named A

`./B/descendant::text()` is a **relative** path expression which denotes all the **Text** nodes descendant of an **Element** B, itself child of the context node

`/A/B/@att1[.>2]` denotes all the **Attribute** nodes @att1 (of a B node child of the A root element) whose value is greater than 2

`.` is a special step, which refers to the context node. Thus, `. /toto` means the same thing as `toto`.

Evaluation of Path Expressions

Each step step_i is interpreted with respect to a **context**; its result is a **node list**.

A step step_i is evaluated with respect to the context of step_{i-1} . More precisely:

For $i = 1$ (first step) if the path is **absolute**, the context is a singleton, the root of the XML tree; else (**relative paths**) the context is defined by the environment;

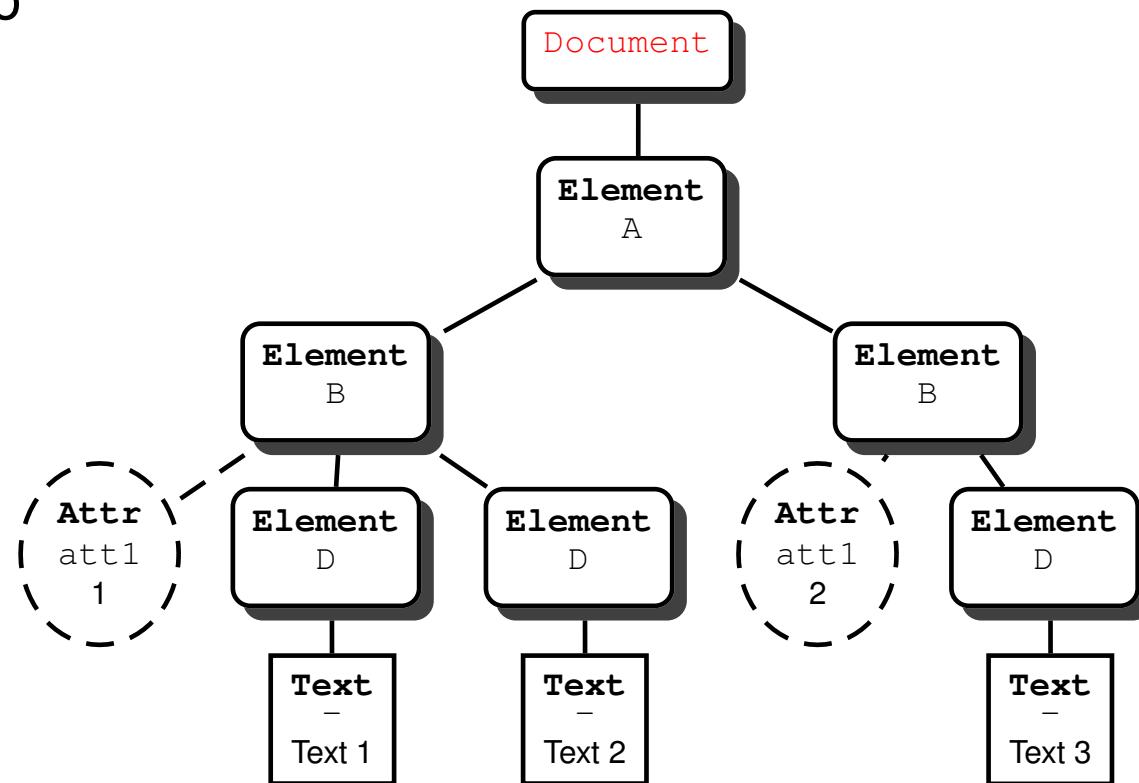
For $i > 1$ if $\mathcal{N} = \langle N_1, N_2, \dots, N_n \rangle$ is the result of step step_{i-1} , step_i is successively evaluated with respect to the context $[\mathcal{N}, N_j]$, for each $j \in [1, n]$.

The result of the path expression is the node set obtained after evaluating the last step.

Evaluation of /A/B/@att1

The path expression is absolute: the context consists of the root node of the tree.

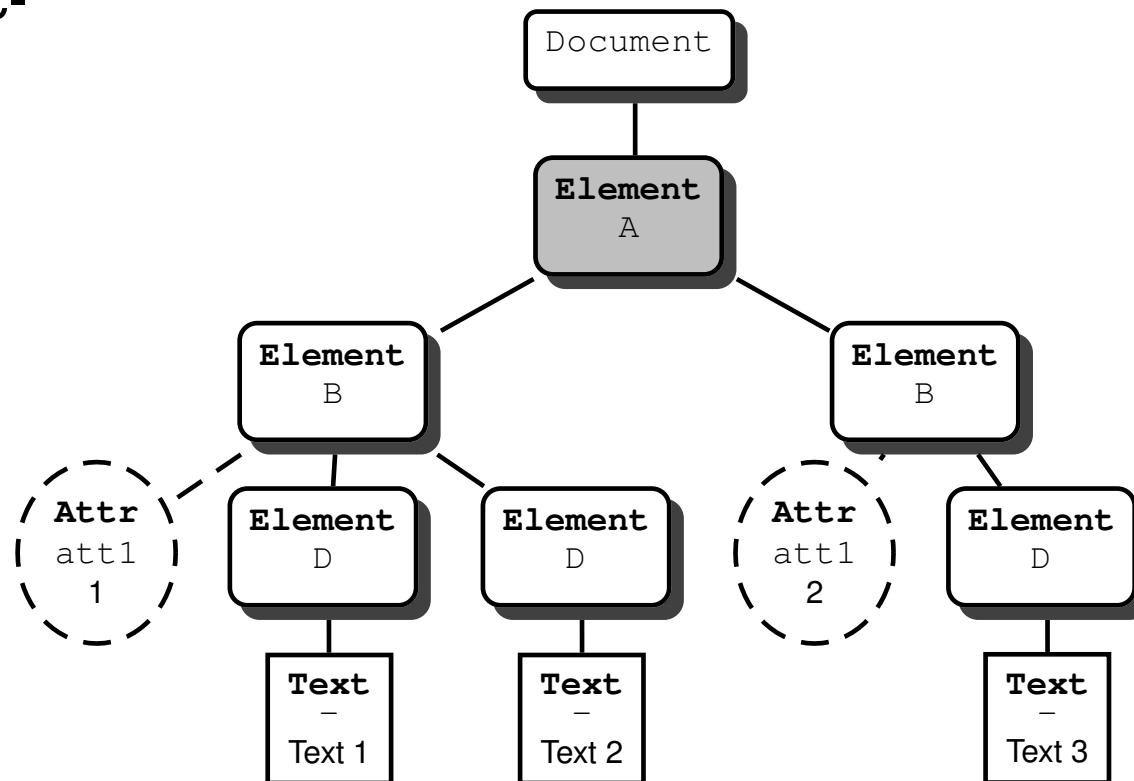
The first step, A, is evaluated with respect to this context.



Evaluation of /A/B/@att1

The result is **A**, the root element.

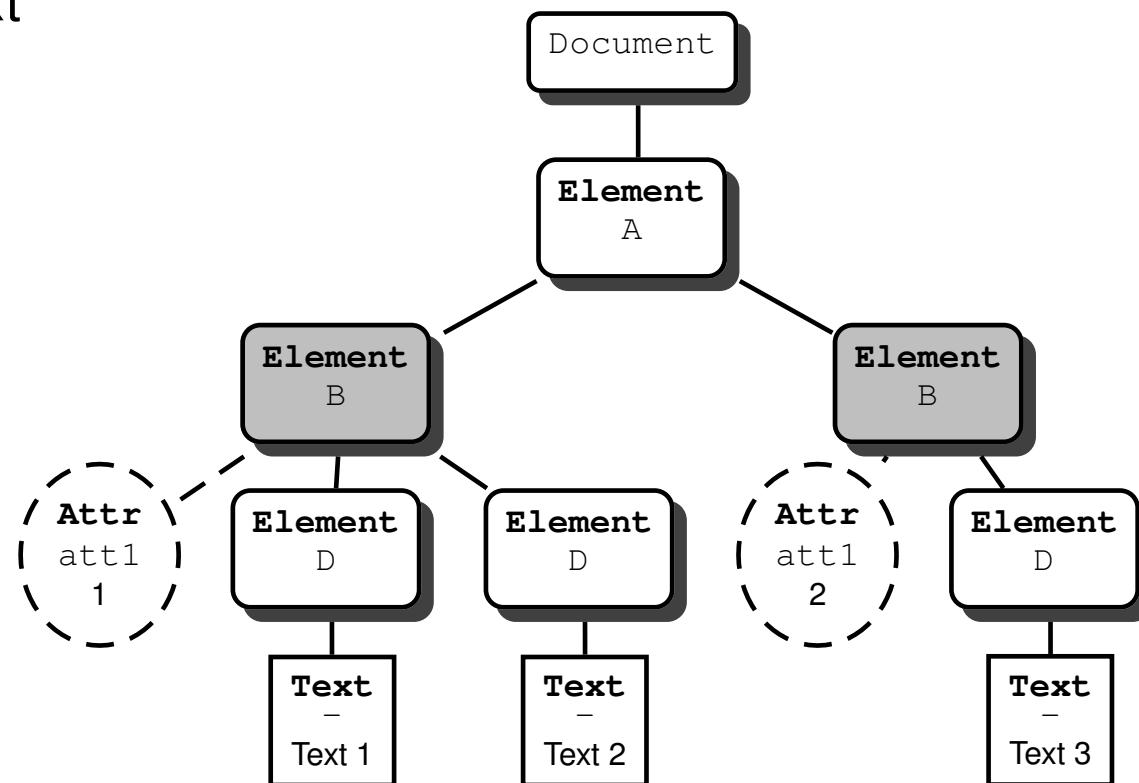
A is the context for the evaluation of the second step, **B**.



Evaluation of /A/B/@att1

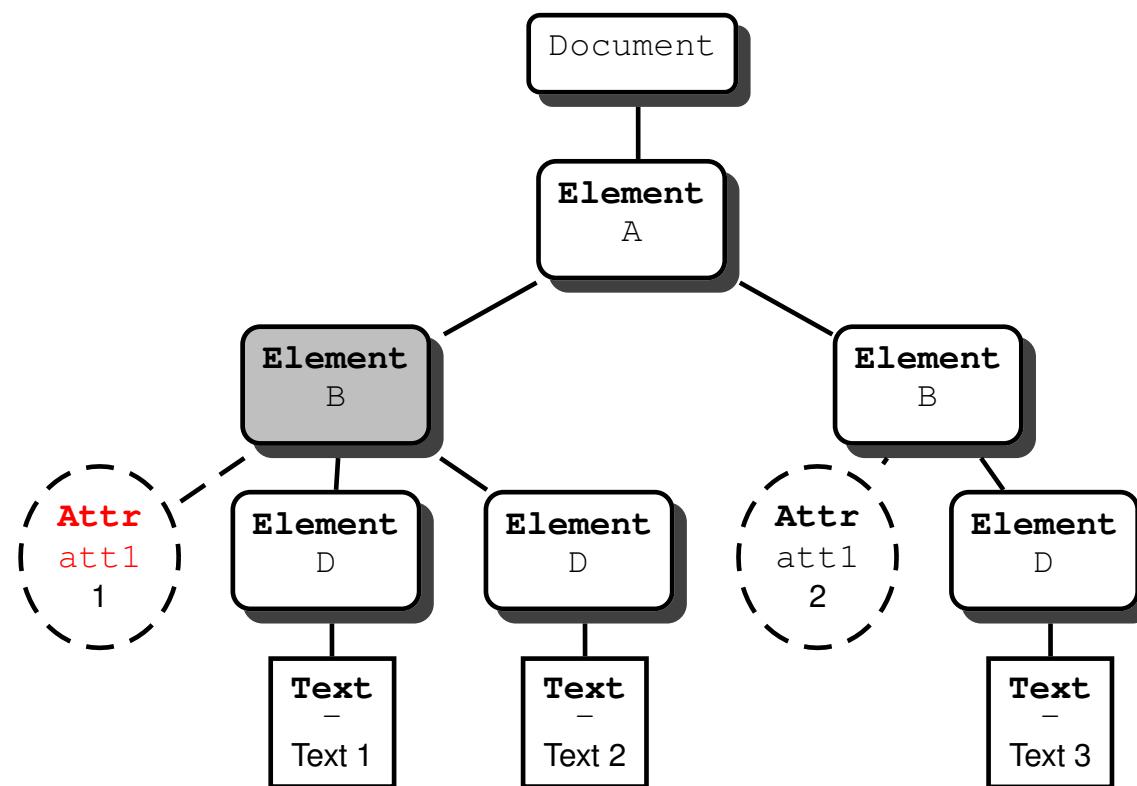
The result is a node list with two nodes B[1], B[2].

@att1 is first evaluated with the context node B[1].



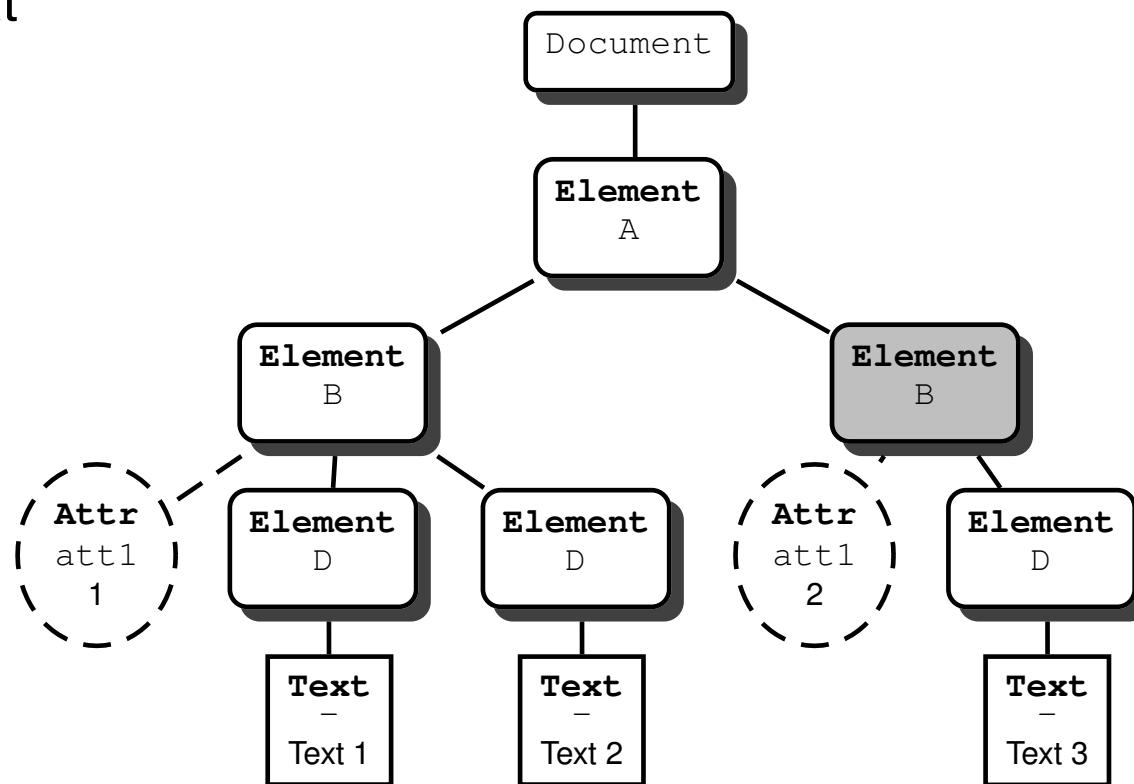
Evaluation of /A/B/@att1

The result is the attribute node of **B[1]**.



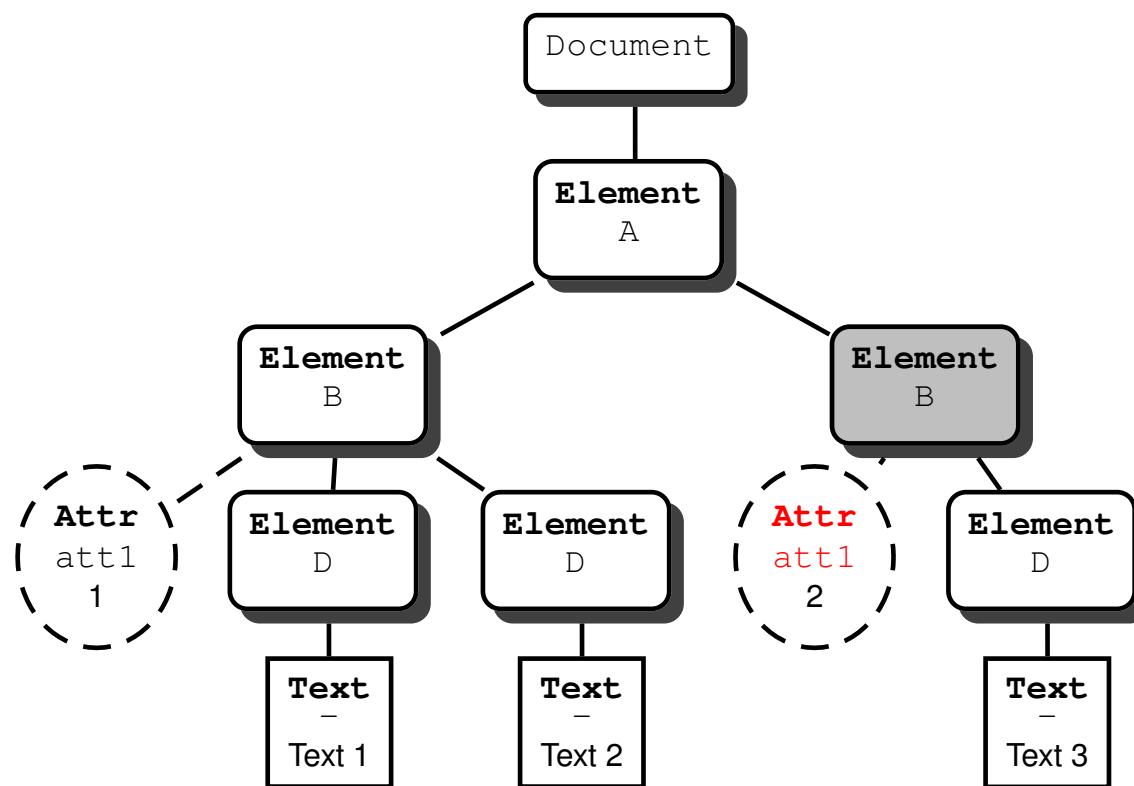
Evaluation of /A/B/@att1

@att1 is also evaluated with the context node **B[2]**.



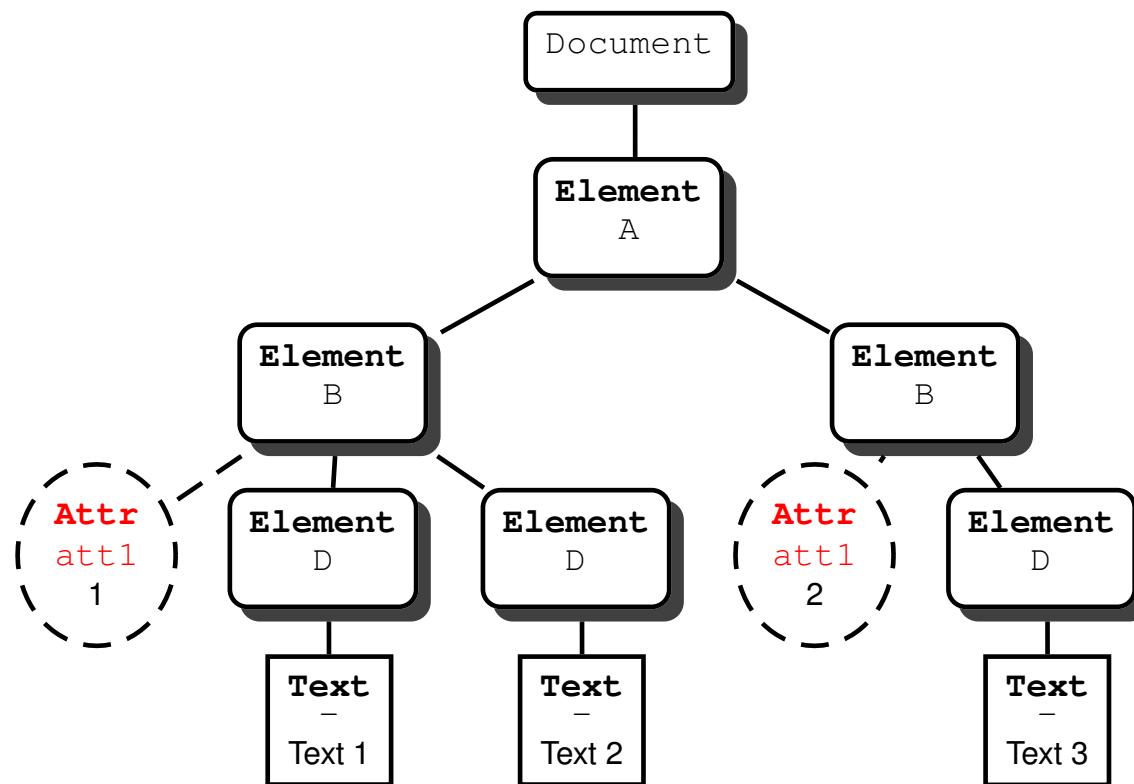
Evaluation of /A/B/@att1

The result is the attribute node of **B[2]**.



Evaluation of /A/B/@att1

Final result: the node set union of all the results of the last step, @att1.



Axes

An axis = a set of nodes determined from the context node, **and** an ordering of the sequence.

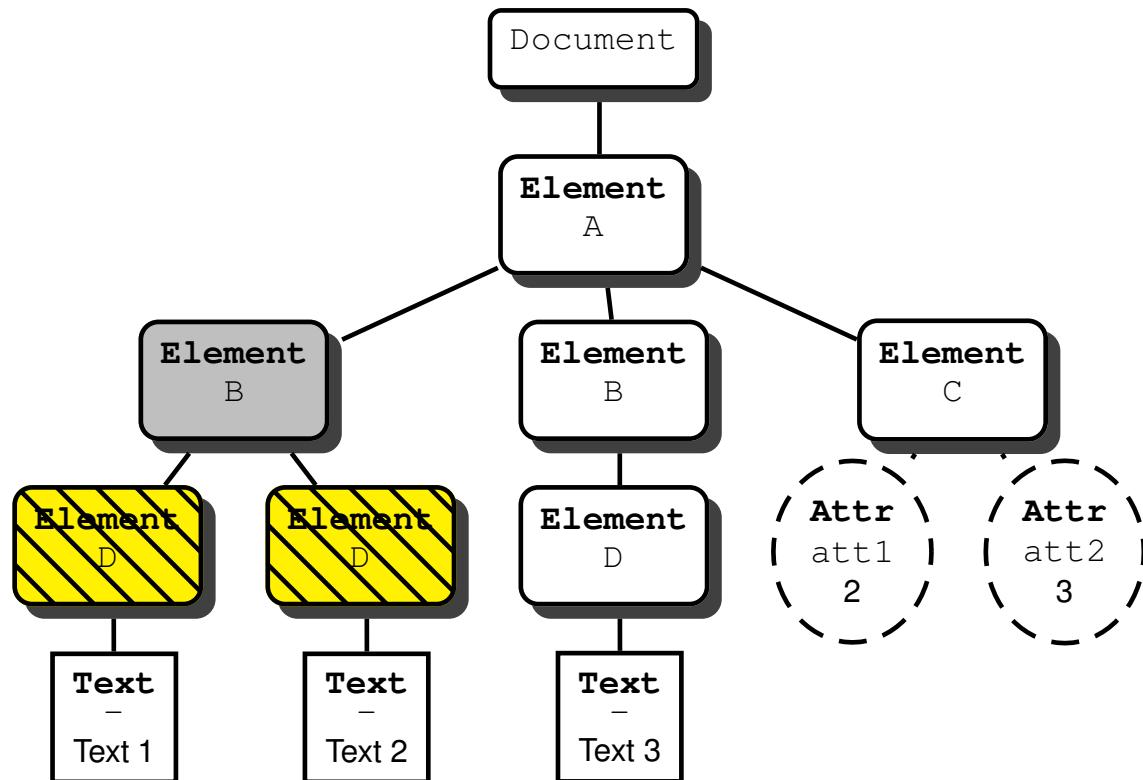
- `child` (**default axis**).
- `parent` Parent node.
- `attribute` Attribute nodes.
- `descendant` Descendants, excluding the node itself.
- `descendant-or-self` Descendants, including the node itself.
- `ancestor` Ancestors, excluding the node itself.
- `ancestor-or-self` Ancestors, including the node itself.
- `following` Following nodes in **document order**.
- `following-sibling` Following siblings in **document order**.
- `preceding` Preceding nodes in **document order**.
- `preceding-sibling` Preceding siblings in **document order**.
- `self` The context node itself.

Examples of axis interpretation

Child axis: denotes the **Element** or **Text** children of the context node.

Important: An **Attribute** node has a parent (the element on which it is located), but an attribute node is *not* one of the children of its parent.

Result of child::D (equivalent to D)



Examples of axis interpretation

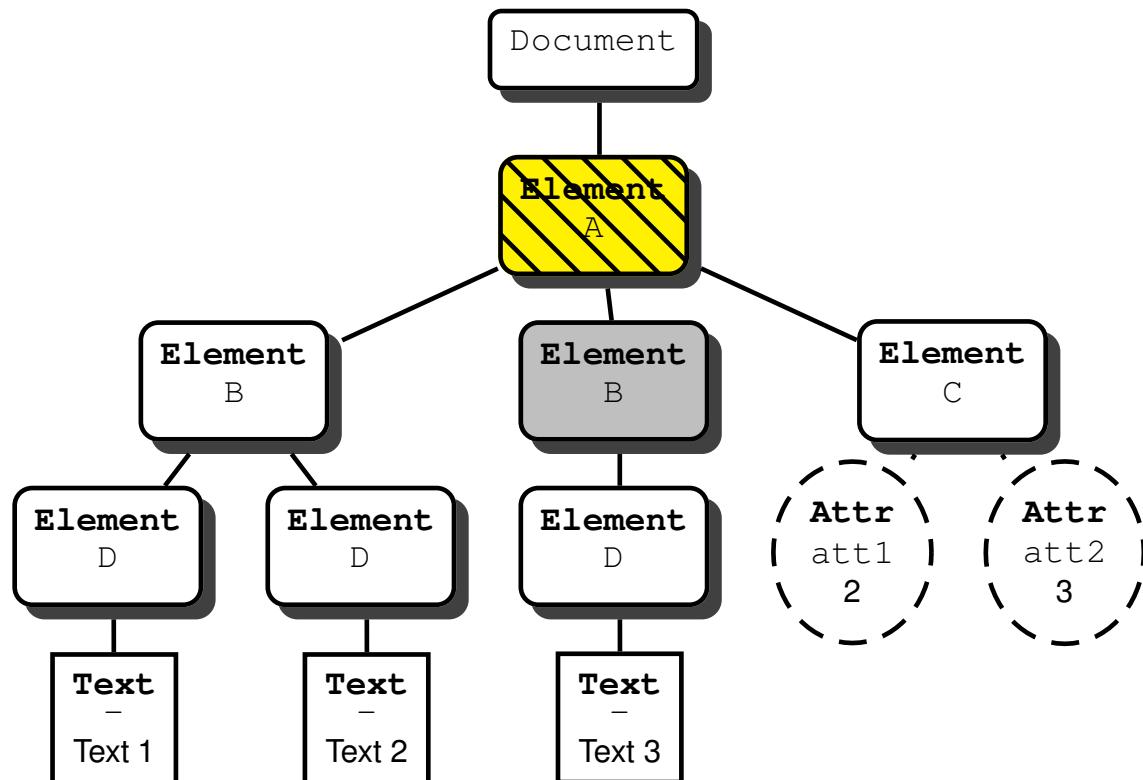
Parent axis: denotes the parent of the context node.

The node test is either an element name, or `*` which matches all names, `node()` which matches all node types.

Always a **Element** or **Document** node, or an empty node-set (if the parent does not match the node test or does not satisfy a predicate).

`..` is an abbreviation for `parent::node()`: the parent of the context node, whatever its type.

Result of `parent::node()` (may be abbreviated to `..`)

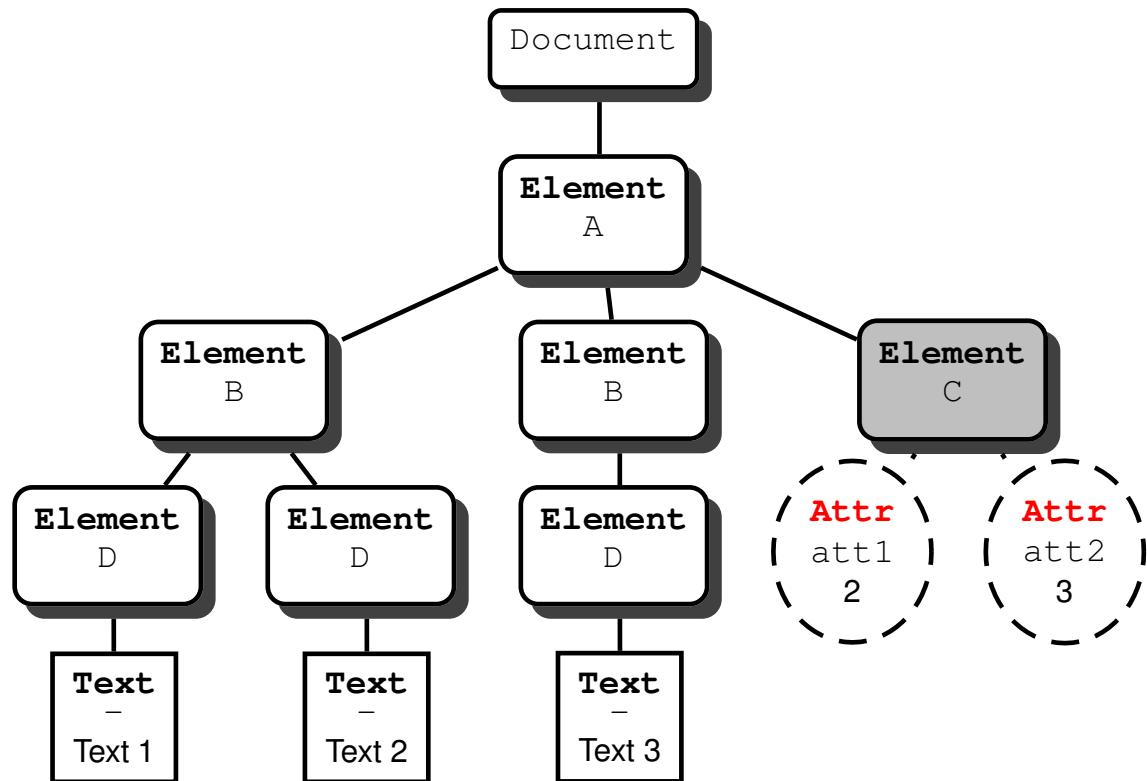


Examples of axis interpretation

Attribute axis: denotes the attributes of the context node.

The node test is either the attribute name, or $*$ which matches all the names.

Result of `attribute::*` (equiv. to `@*`)



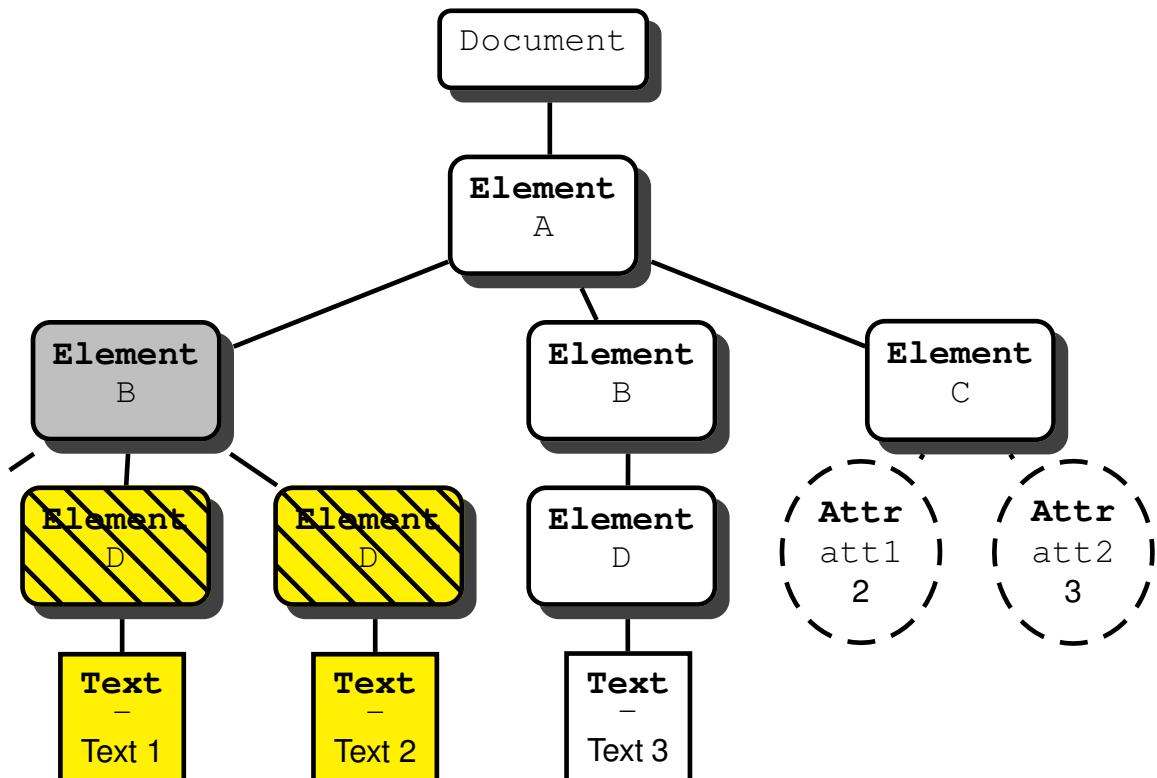
Examples of axis interpretation

Descendant axis: all the descendant nodes, **except** the **Attribute** nodes.

The node test is either the node name (for **Element** nodes), or `*` (any **Element** node) or `text()` (any **Text** node) or `node()` (all nodes).

The context node does **not** belong to the result: use `descendant-or-self` instead.

Result of `descendant::node()`



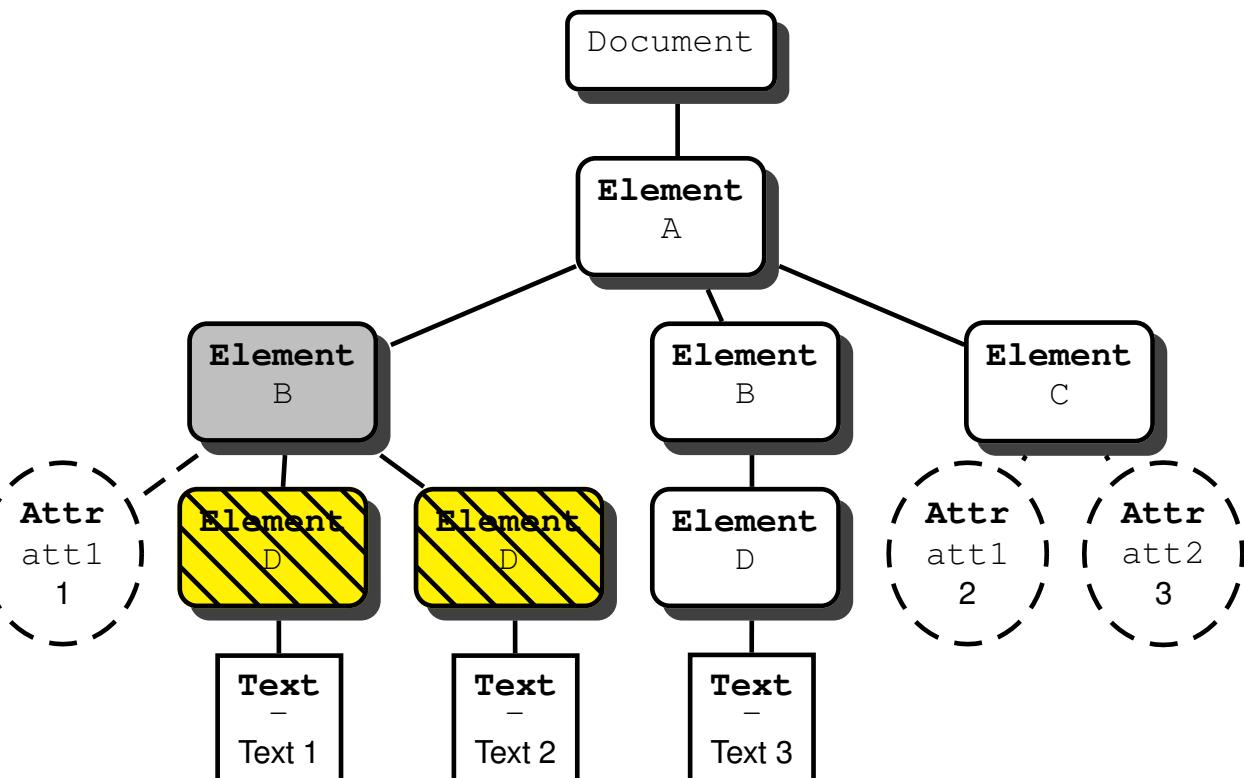
Examples of axis interpretation

Descendant axis: all the descendant nodes, **except** the **Attribute** nodes.

The node test is either the node name (for **Element** nodes), or `*` (any **Element** node) or `text()` (any **Text** node) or `node()` (all nodes).

The context node does **not** belong to the result: use `descendant-or-self` instead.

Result of descendant :: *



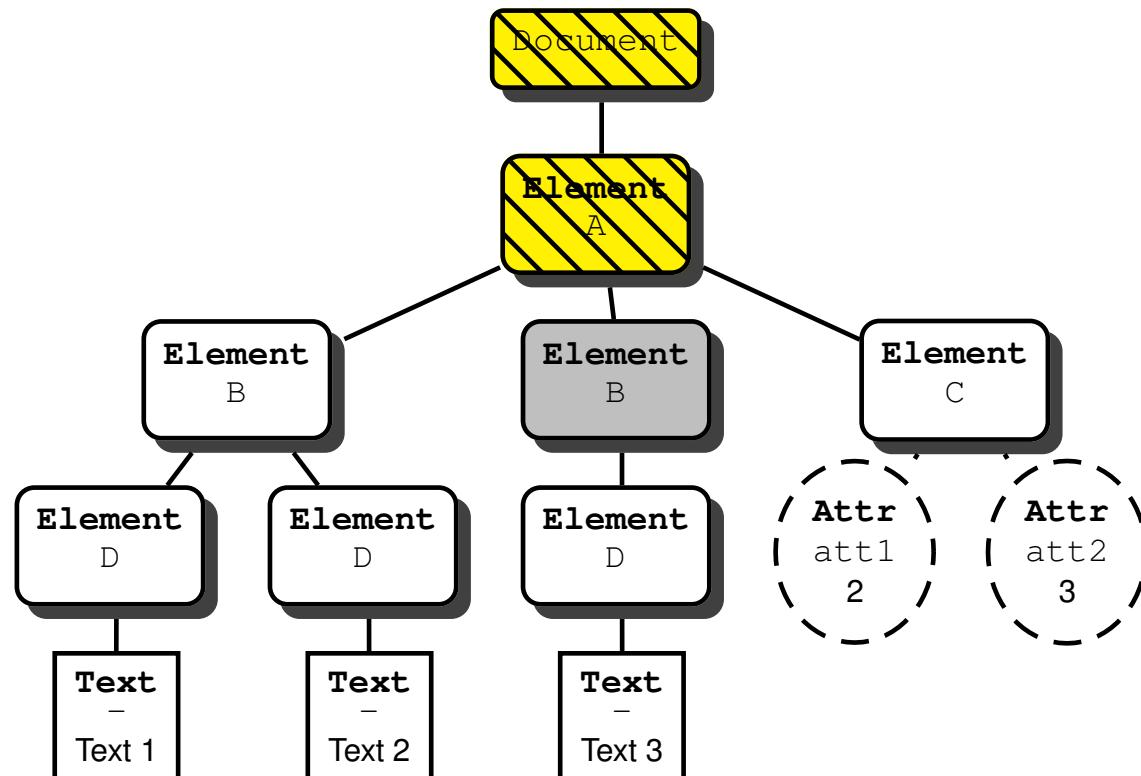
Examples of axis interpretation

Ancestor axis: all the ancestor nodes.

The node test is either the node name (for **Element** nodes), or `node()` (any **Element** node, and the **Document** root node).

The context node does **not** belong to the result: use `ancestor-or-self` instead.

Result of `ancestor::node()`



Examples of axis interpretation

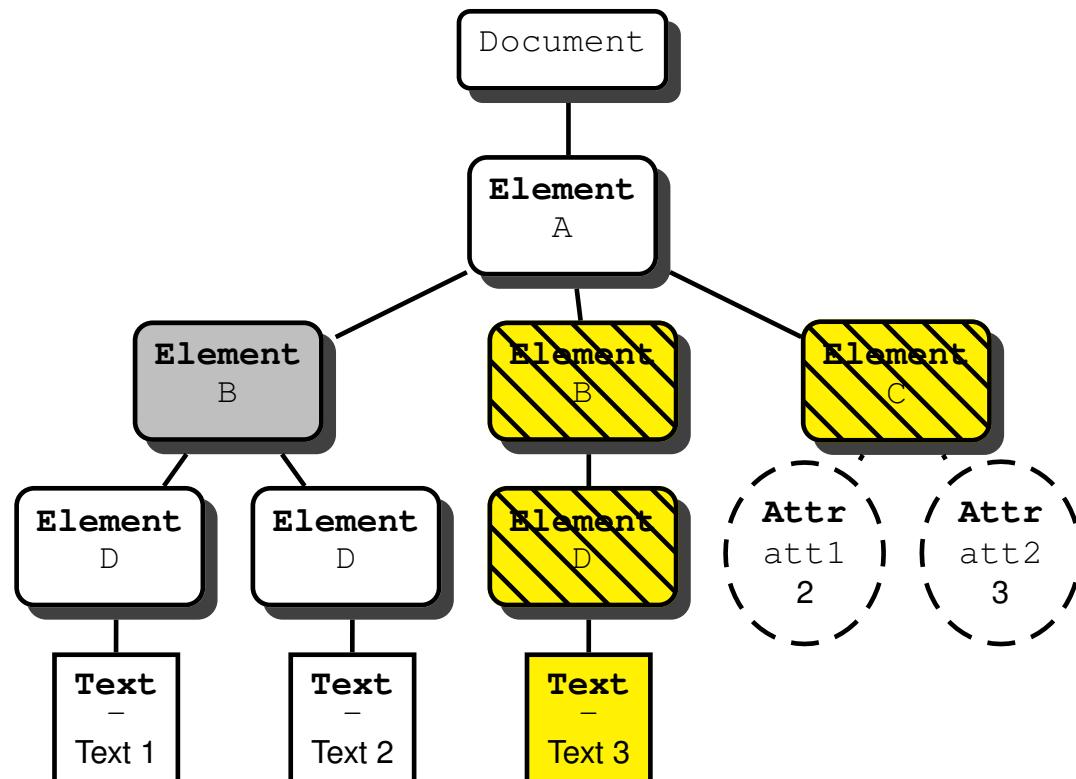
Following axis: all the nodes that follows the context node in the document order.

Attribute nodes are *not* selected.

The node test is either the node name, `* text()` or `node()`.

The axis `preceding` denotes all the nodes that precede the context node.

Result of `following::node()`



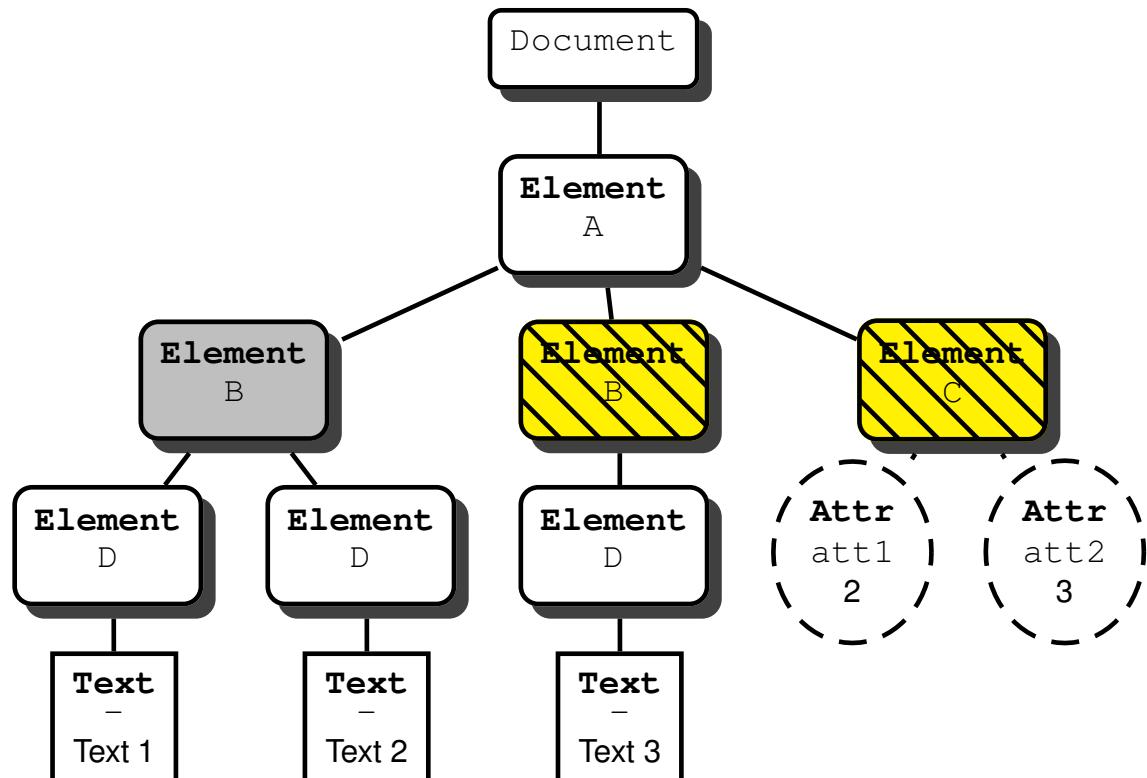
Examples of axis interpretation

Following sibling axis: all the nodes that follows the context node, and share the same parent node.

Same node tests as descendant or following.

The axis preceding-sibling denotes all the nodes that precede the context node.

Result of `following-sibling::node()`



Abbreviations (summary)

Summary of abbreviations:

somename	child::somename
.	self::node()
..	parent::node()
@someattr	attribute::someattr
a//b	a/descendant-or-self::node() / b
//a	/descendant-or-self::node() / a
/	/self::node()

Examples

- `@b` selects the `b` attribute of the context node
- `.. / *` selects all element siblings of the context node, itself included (if it is an element node)

`//@someattr` selects all `someattr` attributes wherever their position in the document

Node Tests (summary)

A node test has one of the following forms:

`node ()` any node

`text ()` any text node

`*` any element (or any attribute for the `attribute` axis)

`ns : *` any element or attribute in the namespace bound to the prefix
`ns`

`ns : toto` any element or attribute in the namespace bound to the prefix
`ns` and whose name is `toto`

Examples

`a / node ()` selects all nodes which are children of a `a` node, itself child of the context node

`xsl : *` selects all elements whose namespace is bound to the prefix `xsl` and that are children of the context node

`/ *` selects the top-level element node

XPath Predicates

- Boolean expression, built with **tests** and the Boolean connectors `and` and `or` (negation is expressed with the `not()` function);
- a **test** is
 - ▶ either an XPath expression, whose result is converted to a Boolean;
 - ▶ a comparison or a call to a Boolean function.

Important: predicate evaluation requires several rules for converting nodes and node sets to the appropriate type.

Example

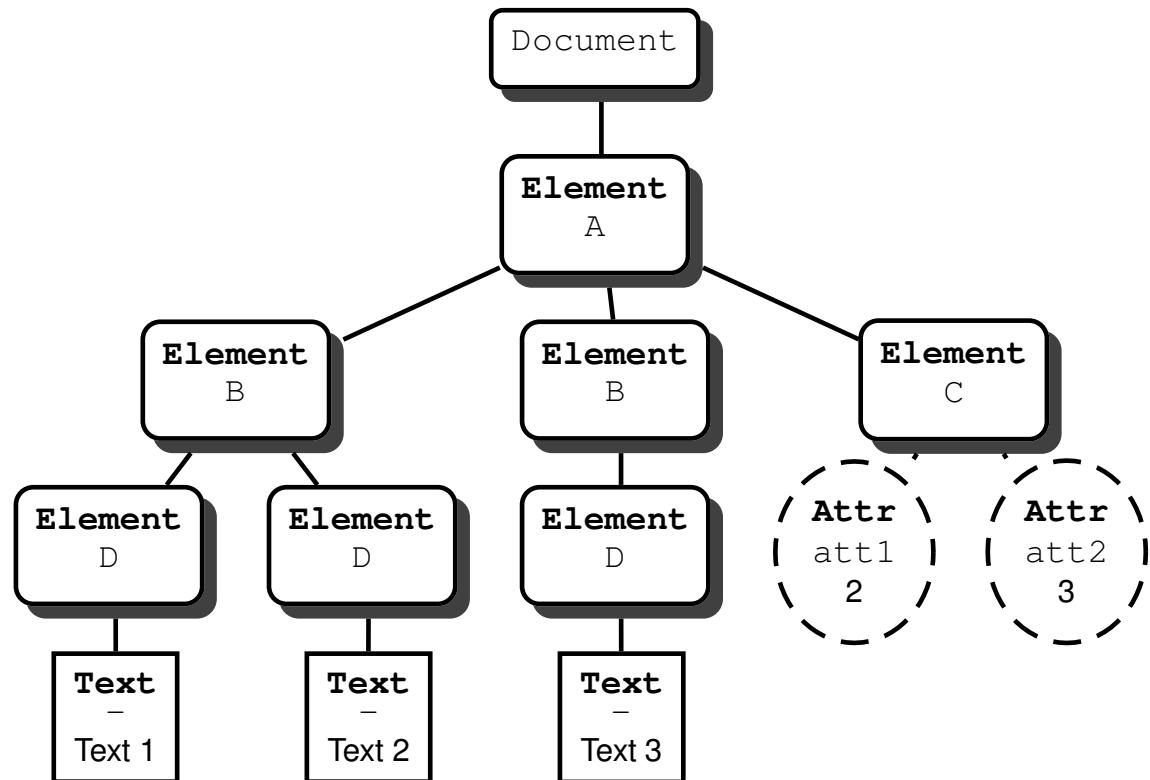
- `//B[@att1=1]`: nodes **B** having an attribute `att1` with value 1;
- `//B[@att1]`: all nodes **B** having an attributes named `att1`!
⇒ `@att1` is an XPath expression whose result (a node set) is converted to a Boolean.
- `//B/descendant::text() [position()=1]`: the first **Text** node descendant of each node **B**.
Can be abbreviated to `//B/descendant::text() [1]`.

Predicate evaluation

A step is of the form
`axis::node-test [P]`.

Ex.: `/A/B/descendant::text () [1]`

- First
`axis::node-test`
is evaluated: one
obtains an
intermediate result I
- Second, for each
node in I , P is
evaluated: the step
result consists of
those nodes in I for
which P is true.

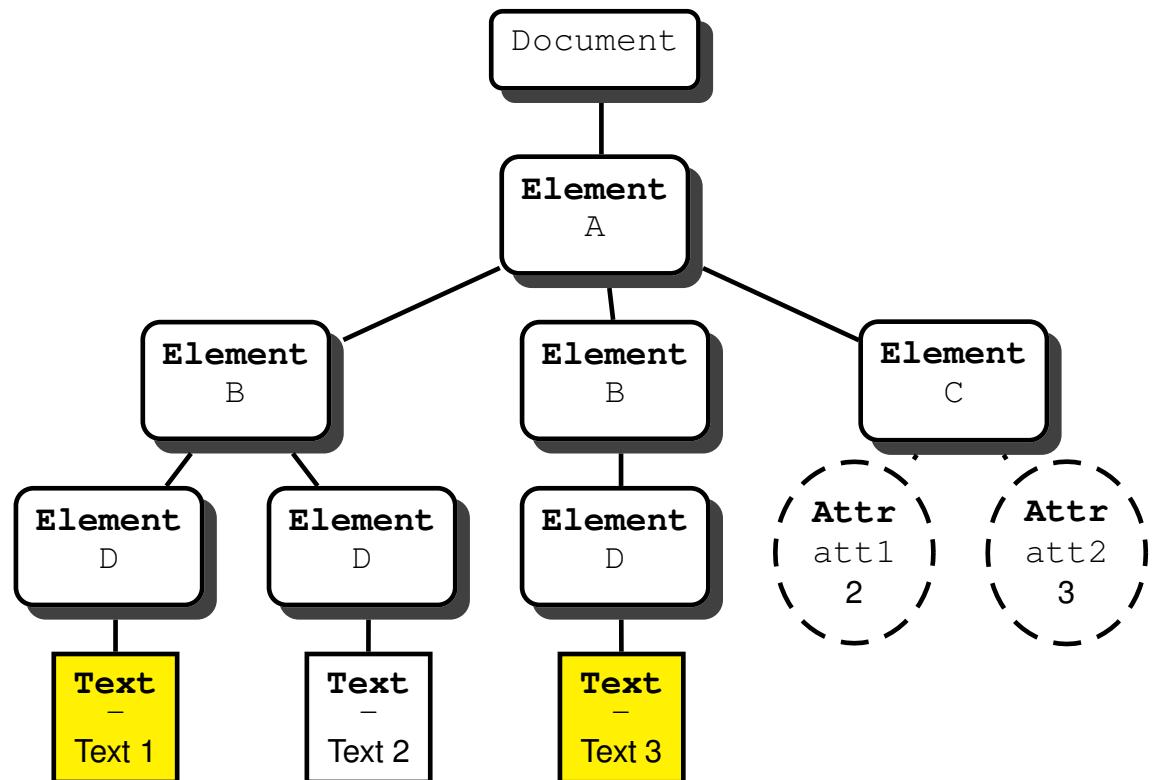


Predicate evaluation

Beware: an XPath step is **always** evaluated with respect to the context of the previous step.

Here the result consists of those **Text** nodes, first descendant (in the document order) of a node **B**.

Result
of
`/A/B/descendant::text() [1]`



XPath 1.0 Type System

Four primitive types:

Type	Description	Literals	Examples
boolean	Boolean values	<i>none</i>	<code>true()</code> , <code>not (\$a=3)</code>
number	Floating-point	12, 12.5	<code>1 div 33</code>
string	Ch. strings	"to", 'ti'	<code>concat('Hello', '!')</code>
nodeset	Node set	<i>none</i>	<code>/a/b[c=1 or @e]/d</code>

The `boolean()`, `number()`, `string()` functions **convert** types into each other (no conversion to nodesets is defined), but this conversion is done in an **implicit** way most of the time.

Rules for **converting to a boolean**:

- A number is true if it is neither 0 nor NaN.
- A string is true if its length is not 0.
- A nodeset is true if it is not empty.

Rules for converting a nodeset to a string:

- The string value of a nodeset is the string value of its first item in document order.
- The string value of an element or document node is the concatenation of the character data in all text nodes below.
- The string value of a text node is its character data.
- The string value of an attribute node is the attribute value.

Examples (Whitespace-only text nodes removed)

```
<a toto="3">
  <b titi='tutu'><c /></b>
  <d>tata</d>
</a>
```

string(/)	"tata"
string(/a/@toto)	"3"
boolean(/a/b)	true()
boolean(/a/e)	false()

Outline

1 Introduction

2 Path Expressions

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

7 Exercise

Operators

The following operators can be used in XPath.

`+, -, *, div, mod` standard arithmetic operators (Example: `1+2*-3`).

Warning! `div` is used instead of the usual `/`.

`or, and` boolean operators (Example: `@a and c=3`)

`=, !=` equality operators. Can be used for strings, booleans or numbers. **Warning!** `//a!=3` means: there is an `a` element in the document whose string value is different from 3.

`<, <=, >=, >` relational operators (Example: `($a<2) and ($a>0)`).

Warning! Can only be used to compare numbers, not strings. If an XPath expression is embedded in an XML document, `<` must be escaped as `<`.

| union of nodesets (Example: `node() | @*`)

Remark

`$a` is a **reference** to the variable `a`. Variables can not be defined in XPath, they can only be referred to.

Node Functions

`count ($s)` returns the **number of items** in the nodeset `$s`

`local-name ($s)` returns the **name** of the first item of the nodeset `$s` in document order, **without** the namespace prefix; if `$s` is omitted, it is taken to be the context item

`namespace-uri ($s)` returns the **namespace URI** bound to the prefix of the name of the first item of the nodeset `$s` in document order; if `$s` is omitted, it is taken to be the context item

`name ($s)` returns the **name** of the first item of the nodeset `$s` in document order, **including** its namespace prefix; if `$s` is omitted, it is taken to be the context item

String Functions

`concat($s1, ..., $sn)` concatenates the strings \$s₁, ..., \$s_n

`starts-with($a, $b)` returns true () if the string \$a starts with \$b

`contains($a, $b)` returns true () if the string \$a contains \$b

`substring-before($a, $b)` returns the substring of \$a before the first occurrence of \$b

`substring-after($a, $b)` returns the substring of \$a after the first occurrence of \$b

`substring($a, $n, $l)` returns the substring of \$a of length \$l starting at index \$n (indexes start from 1). \$l may be omitted.

`string-length($a)` returns the length of the string \$a

`normalize-space($a)` removes all leading and trailing whitespace from \$a, and collapse all whitespace to a single character

`translate($a, $b, $c)` returns the string \$a, where all occurrences of a character from \$b has been replaced by the character at the same place in \$c.

Boolean and Number Functions

`not ($b)` returns the **logical negation** of the boolean `$b`

`sum ($s)` returns the **sum** of the values of the nodes in the nodeset `$s`

`floor ($n)` rounds the number `$n` to the **next lowest** integer

`ceiling ($n)` rounds the number `$n` to the **next greatest** integer

`round ($n)` rounds the number `$n` to the **closest** integer

Examples

`count (//*)` returns the number of elements in the document

`normalize-space('titi toto')` returns the string “titi
toto”

`translate('baba,'abcdef','ABCDEF')` returns the string “BABA”

`round(3.457)` returns the number 3

Outline

1 Introduction

2 Path Expressions

3 Operators and Functions

4 XPath examples

5 XPath 2.0

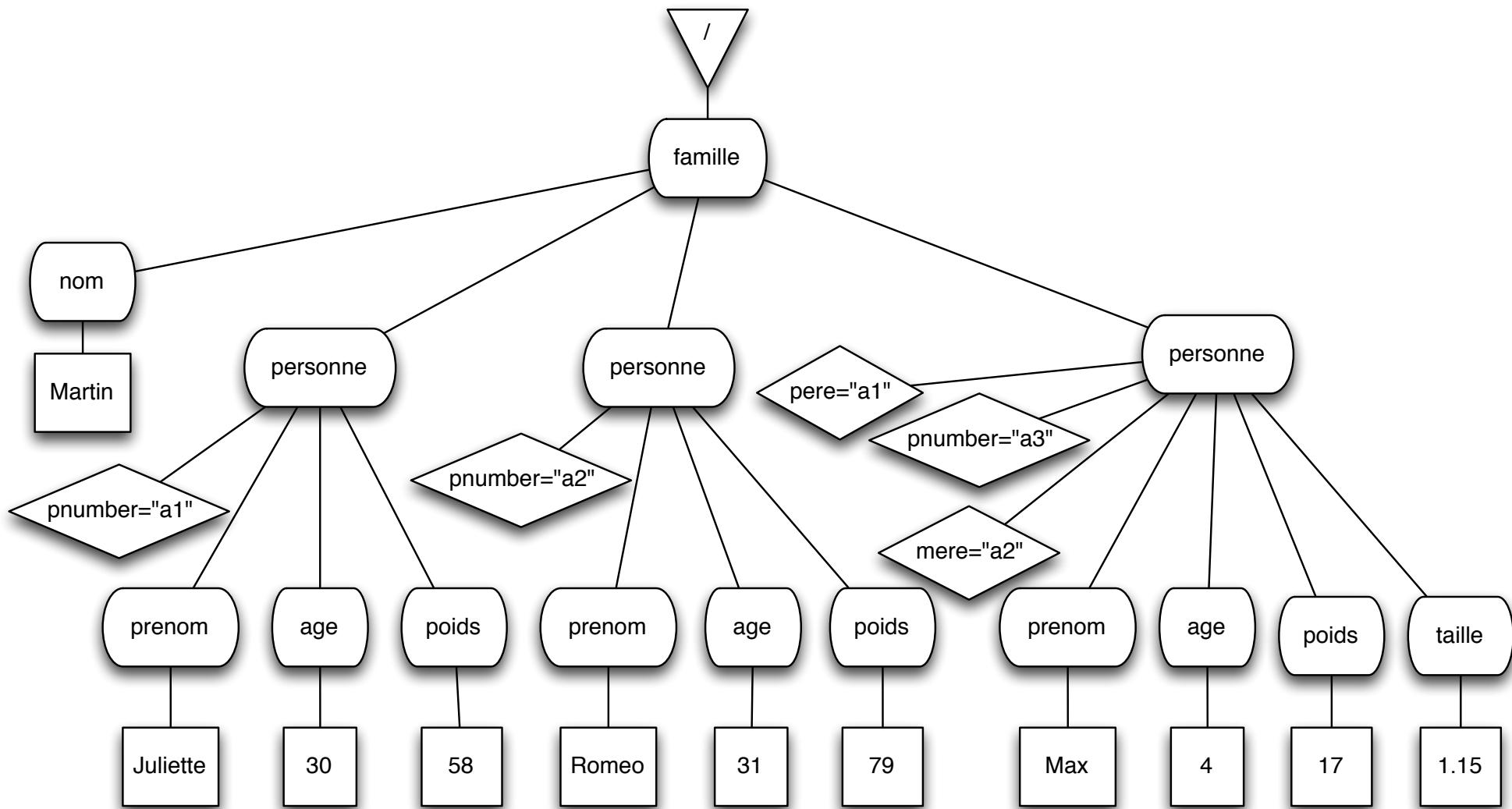
6 Reference Information

7 Exercise

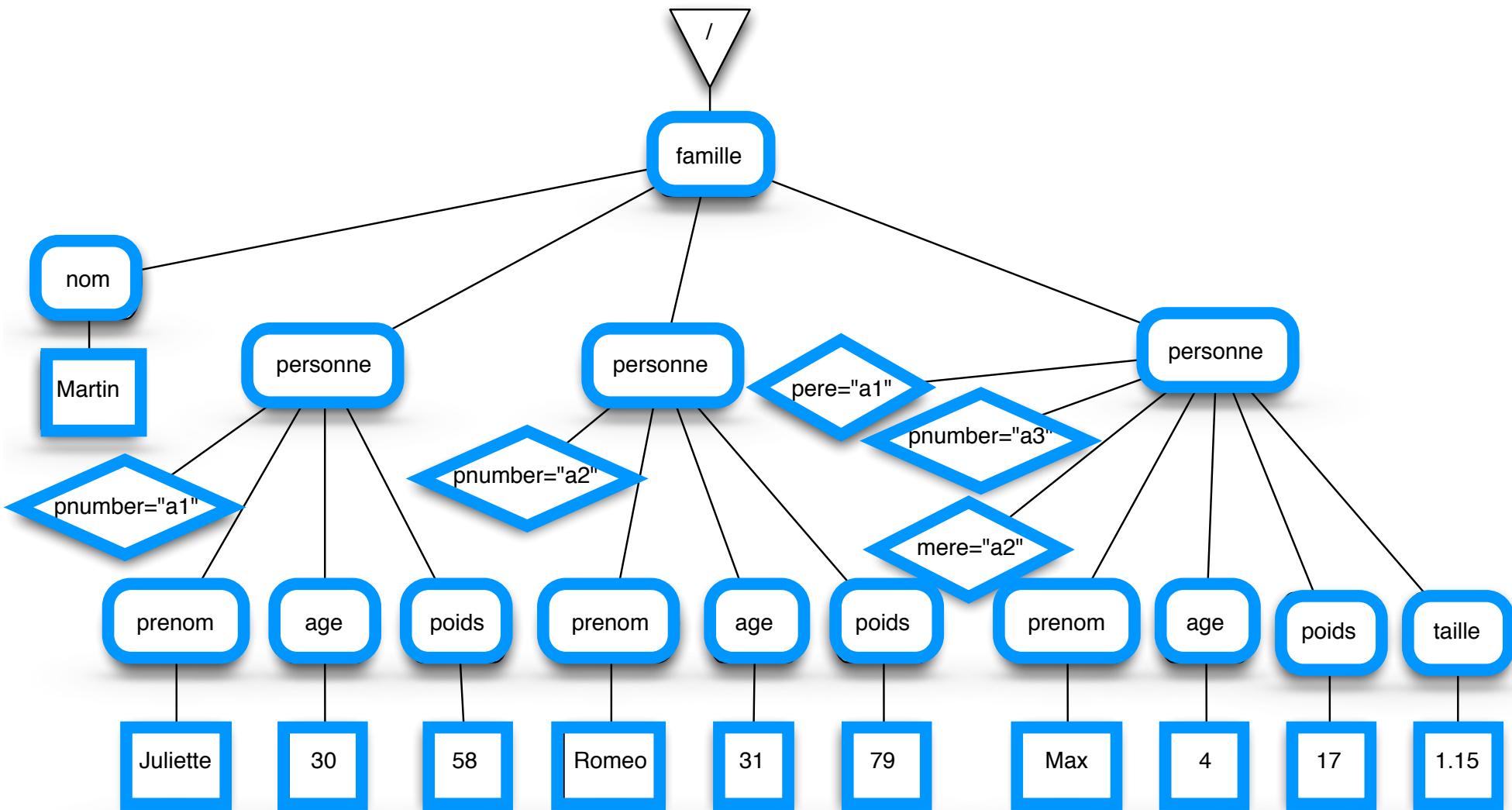
Exemple de document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE famille SYSTEM "famille.dtd">
<famille>
    <nom>Martin</nom>
    <personne pnumber="a1">
        <prenom>Juliette</prenom>
        <age>30</age>
        <poids>58</poids>
    </personne>
    <personne pnumber="a2">
        <prenom>Romeo</prenom>
        <age>31</age>
        <poids>79</poids>
    </personne>
    <personne pnumber="a3" mere="a1" pere="a2">
        <prenom>Max</prenom>
        <age>4</age>
        <poids>17</poids>
        <taille>1.15</taille>
    </personne>
</famille>
```

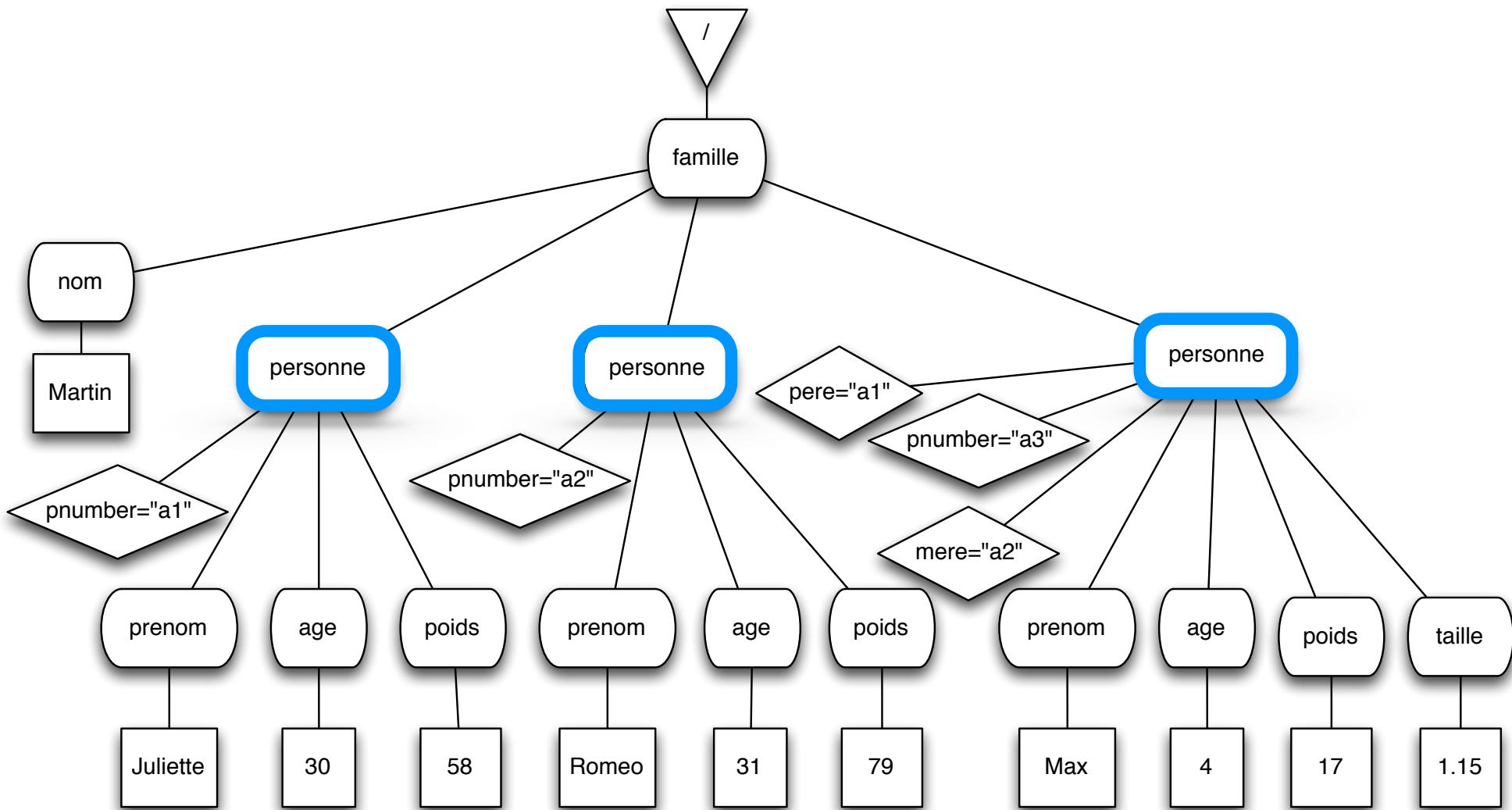
/descendant::personne/child::age/child::text()



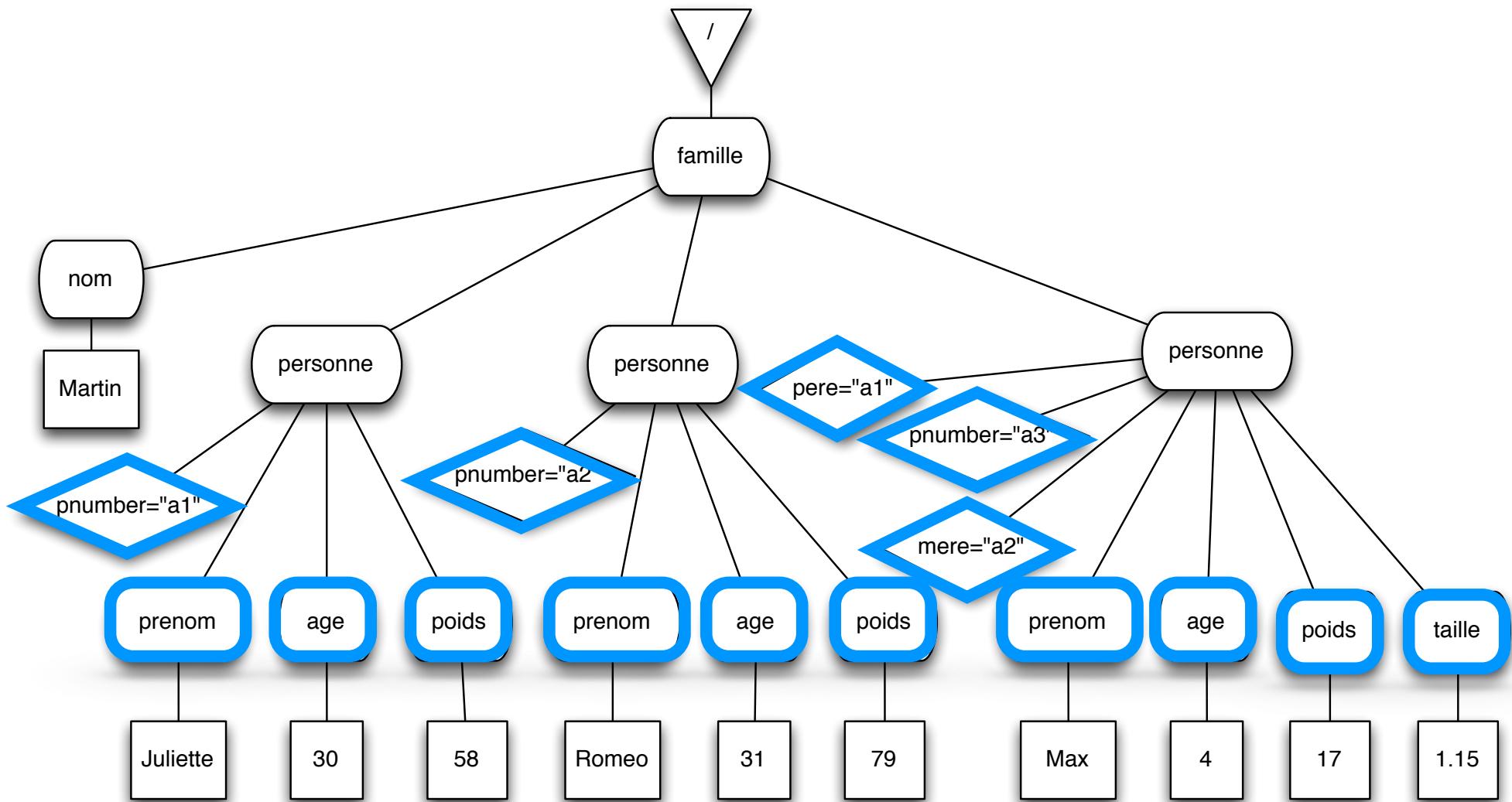
`/descendant::personne/child::age/child::text()`



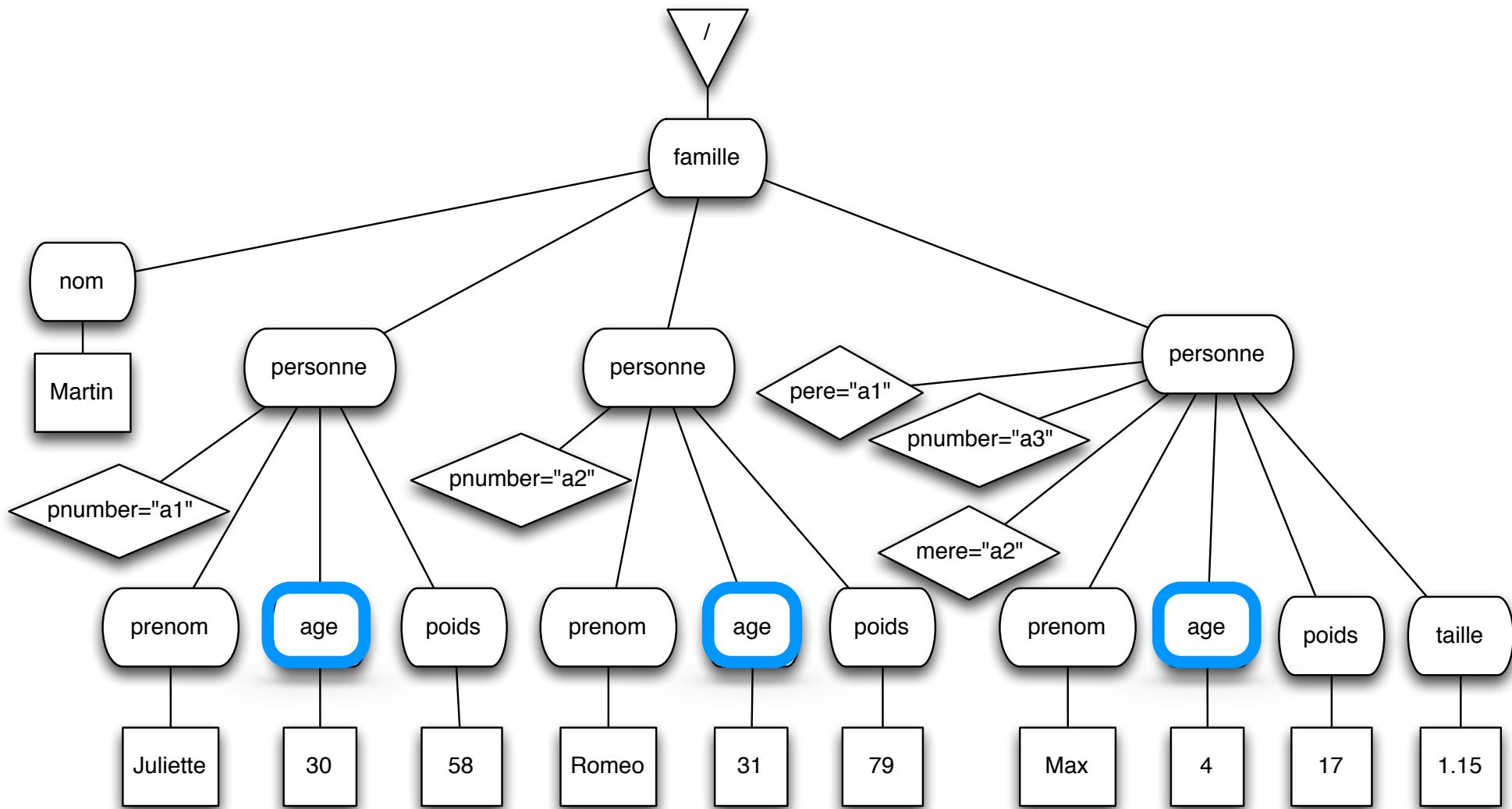
/descendant::**personne**/child::age/child::text()



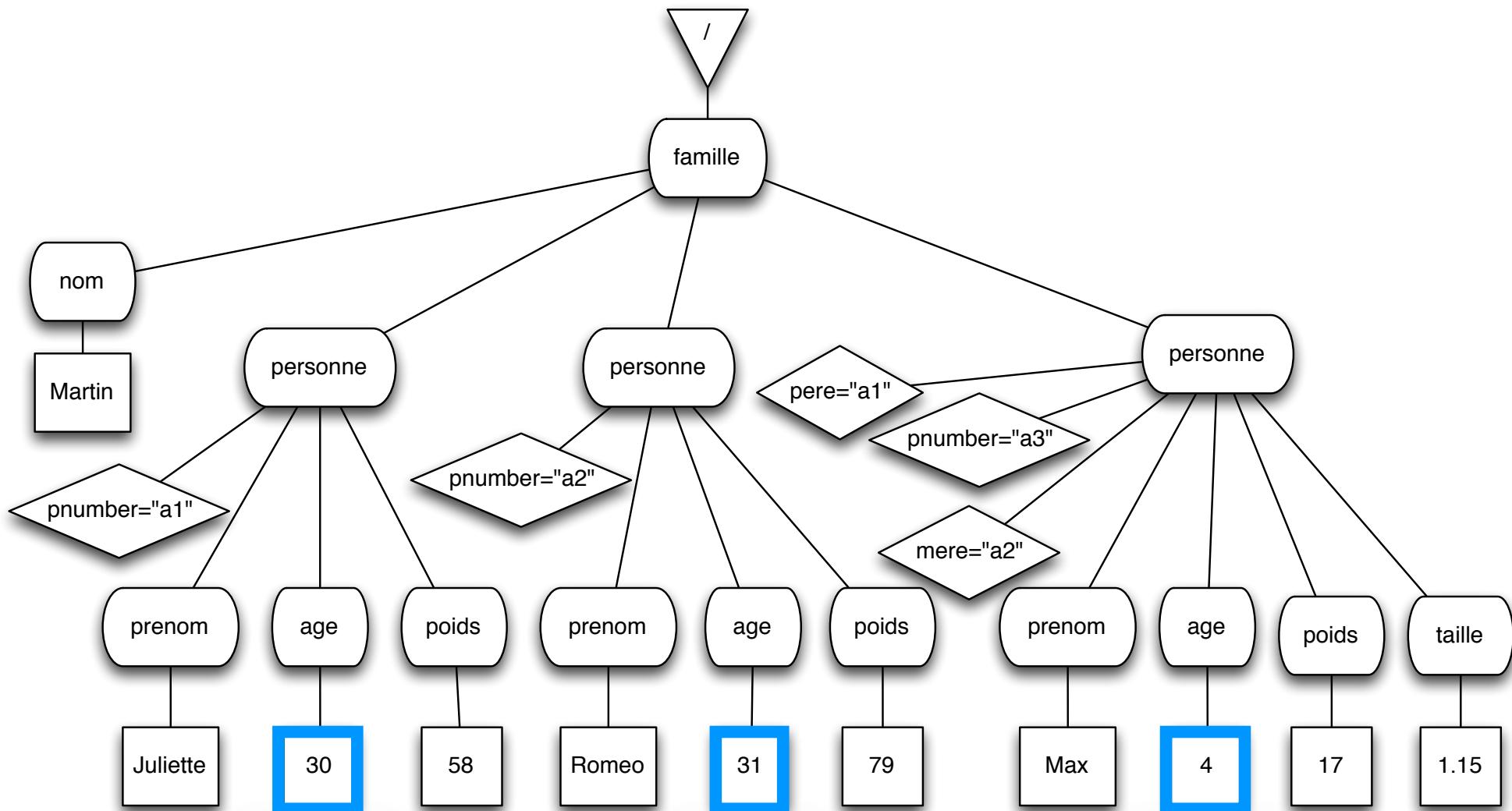
/descendant::personne/child::age/child::text()



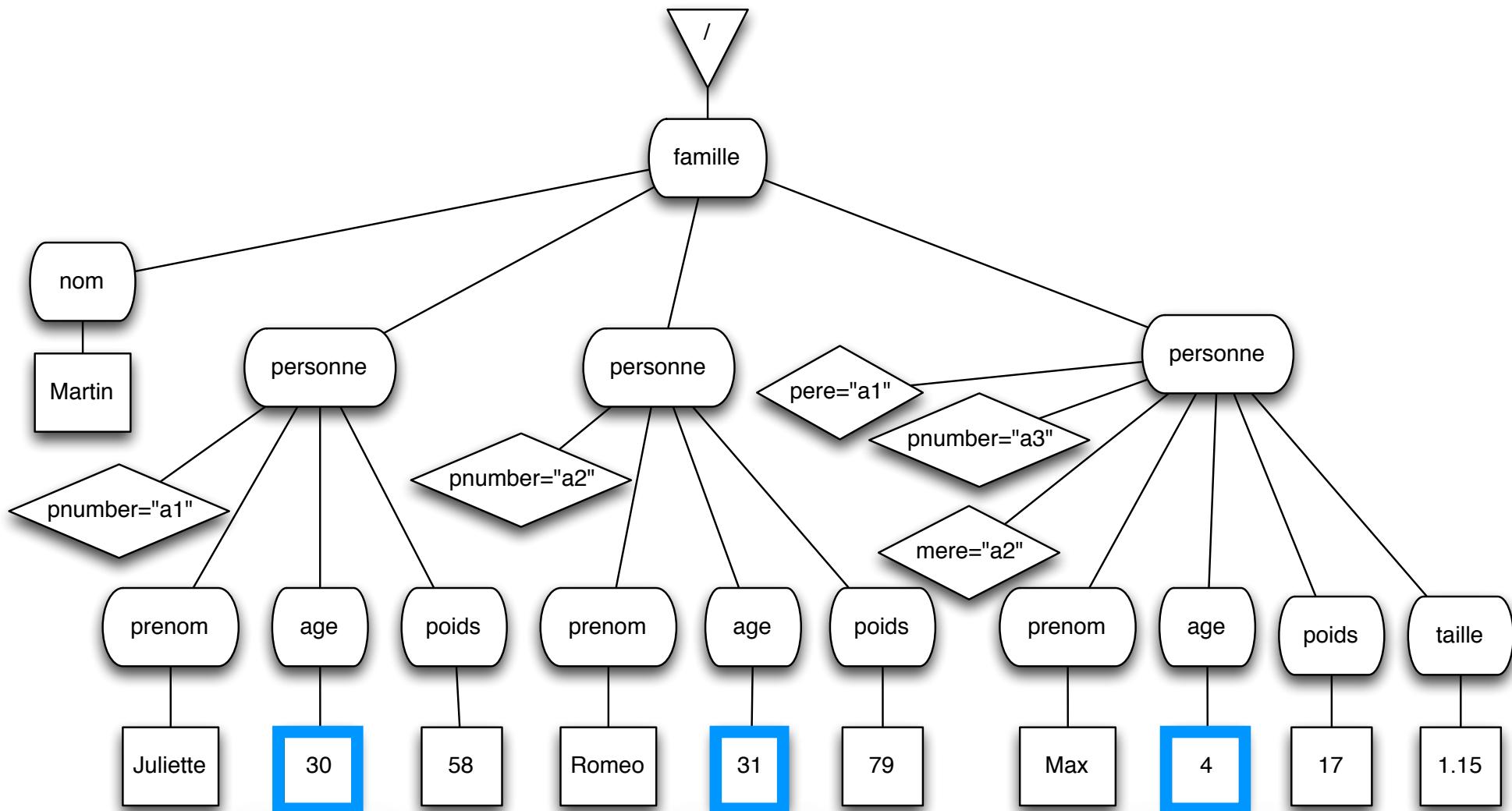
/descendant::personne/child::age/child::text()



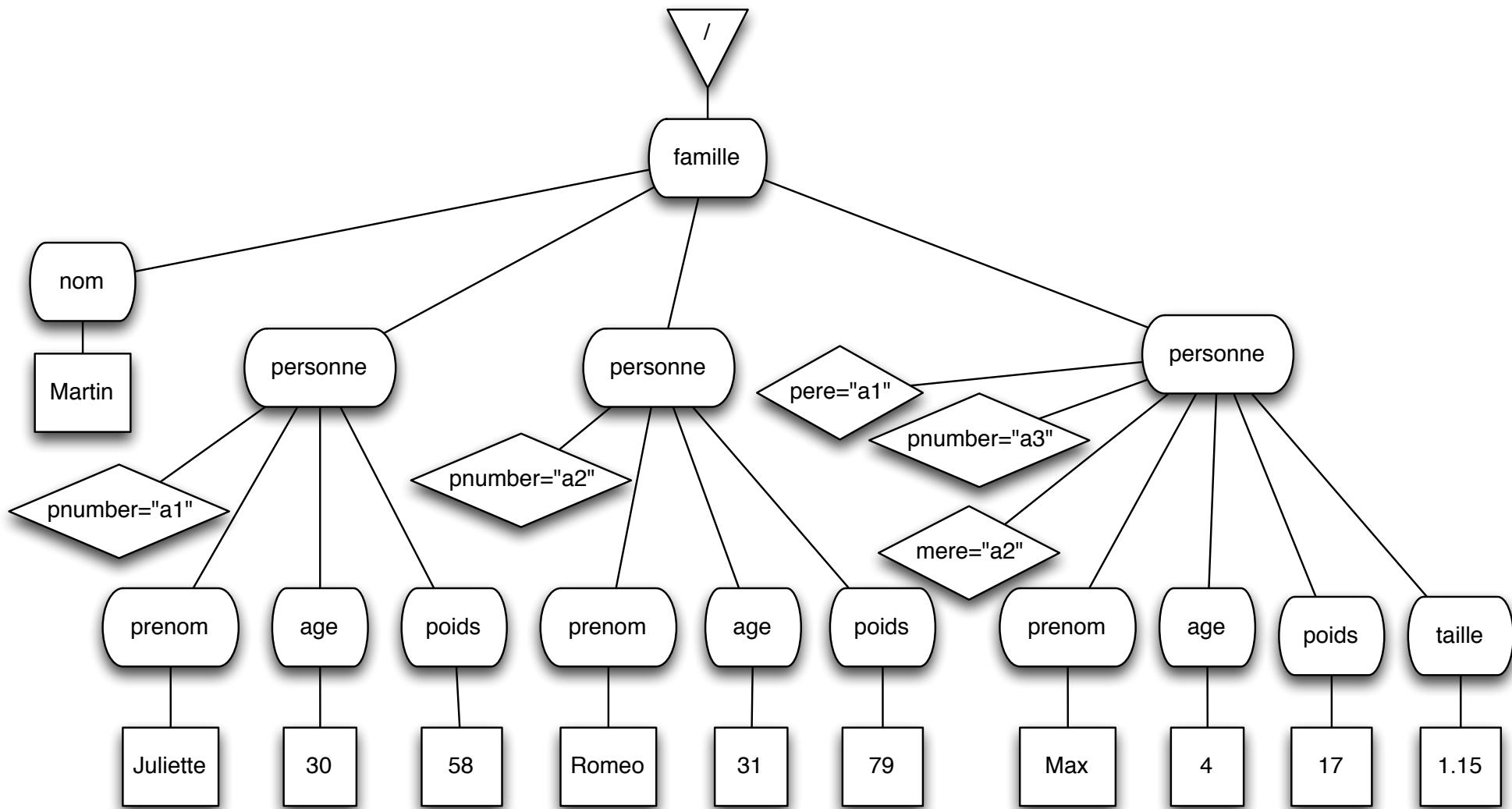
/descendant::personne/child::age/child::text()



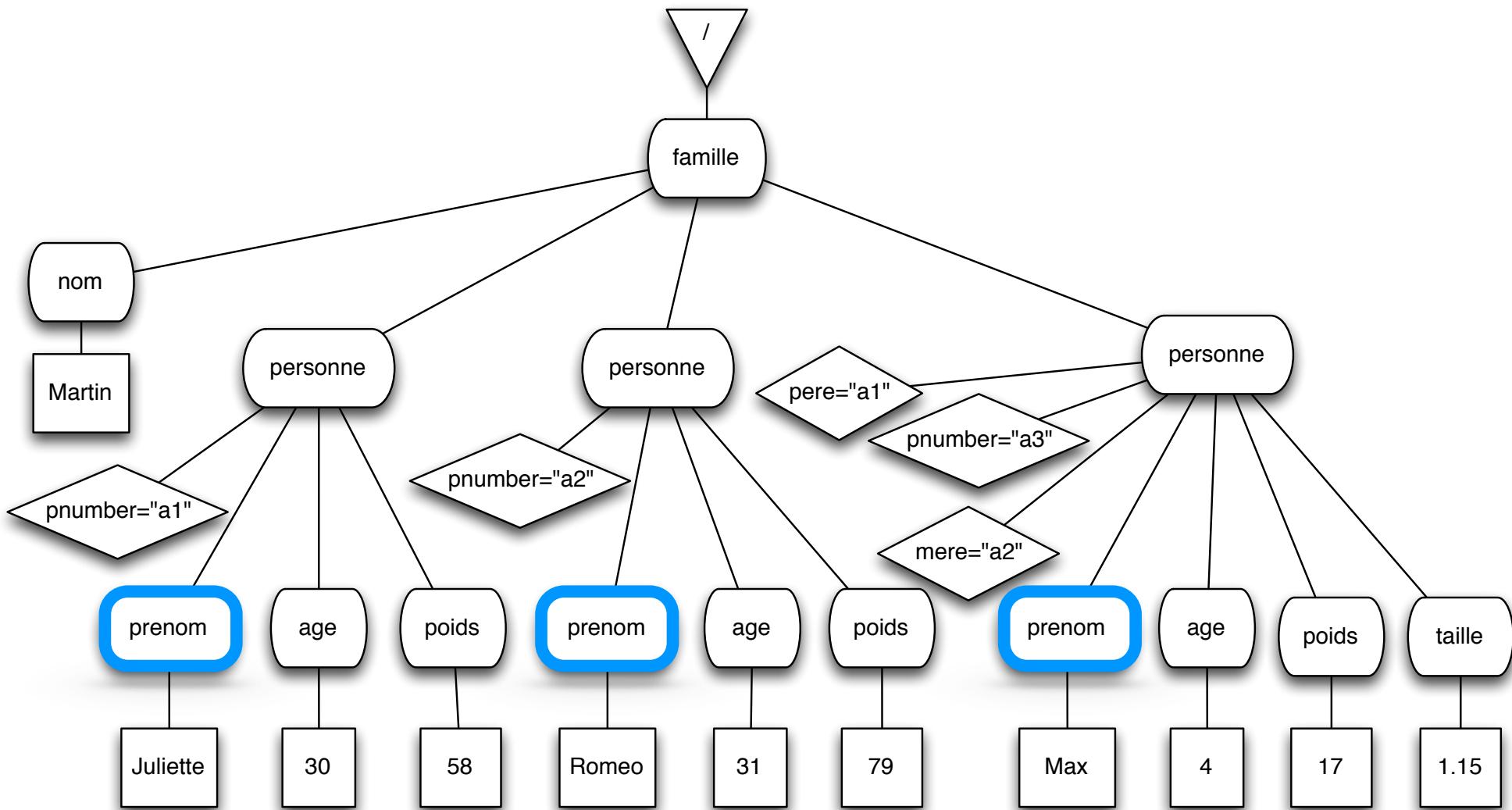
/descendant::personne/child::age/child::text()



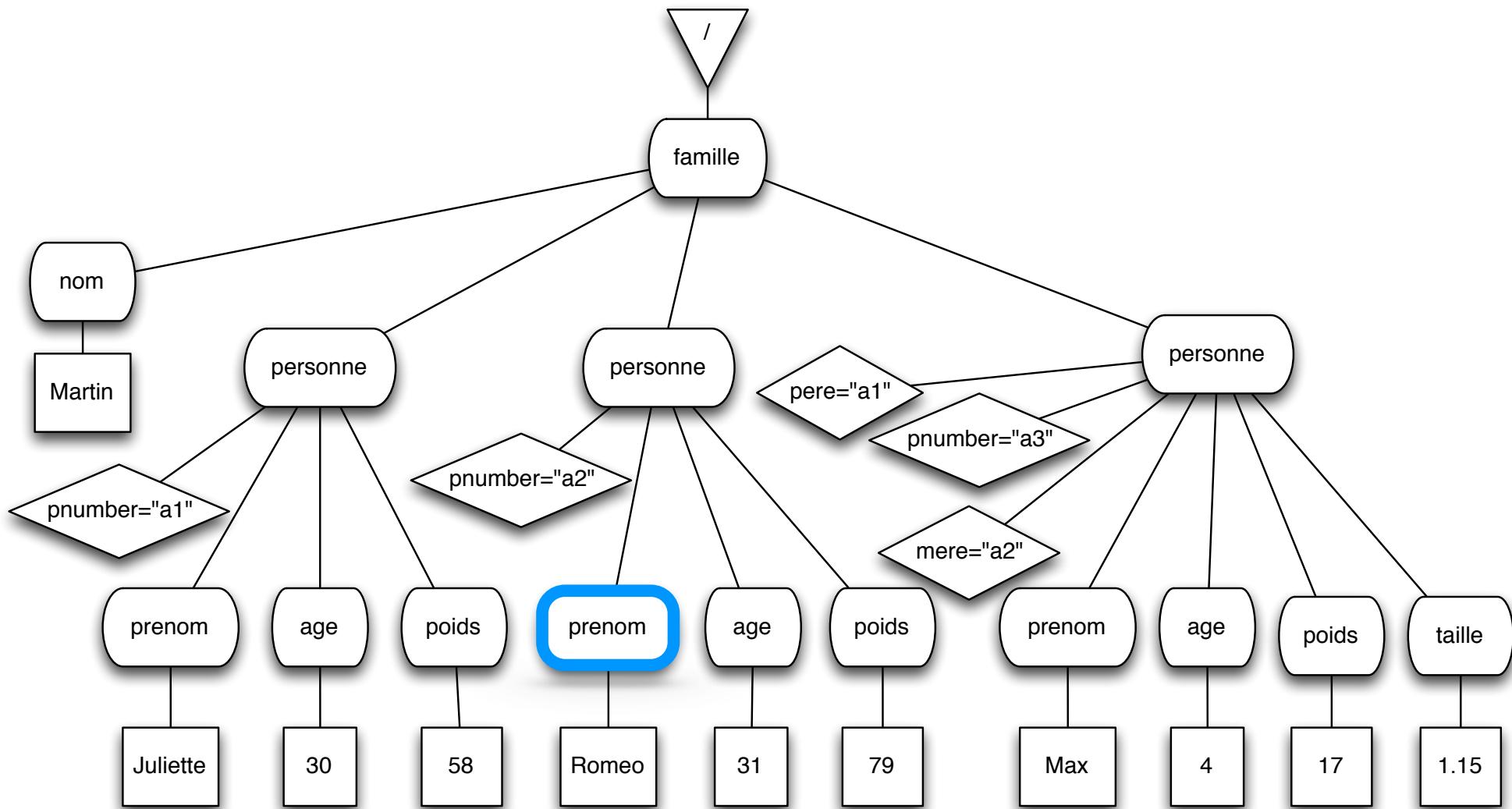
```
/descendant::prenom[child::text()="Romeo"]/*[parent::*/attribute::pnumber
```



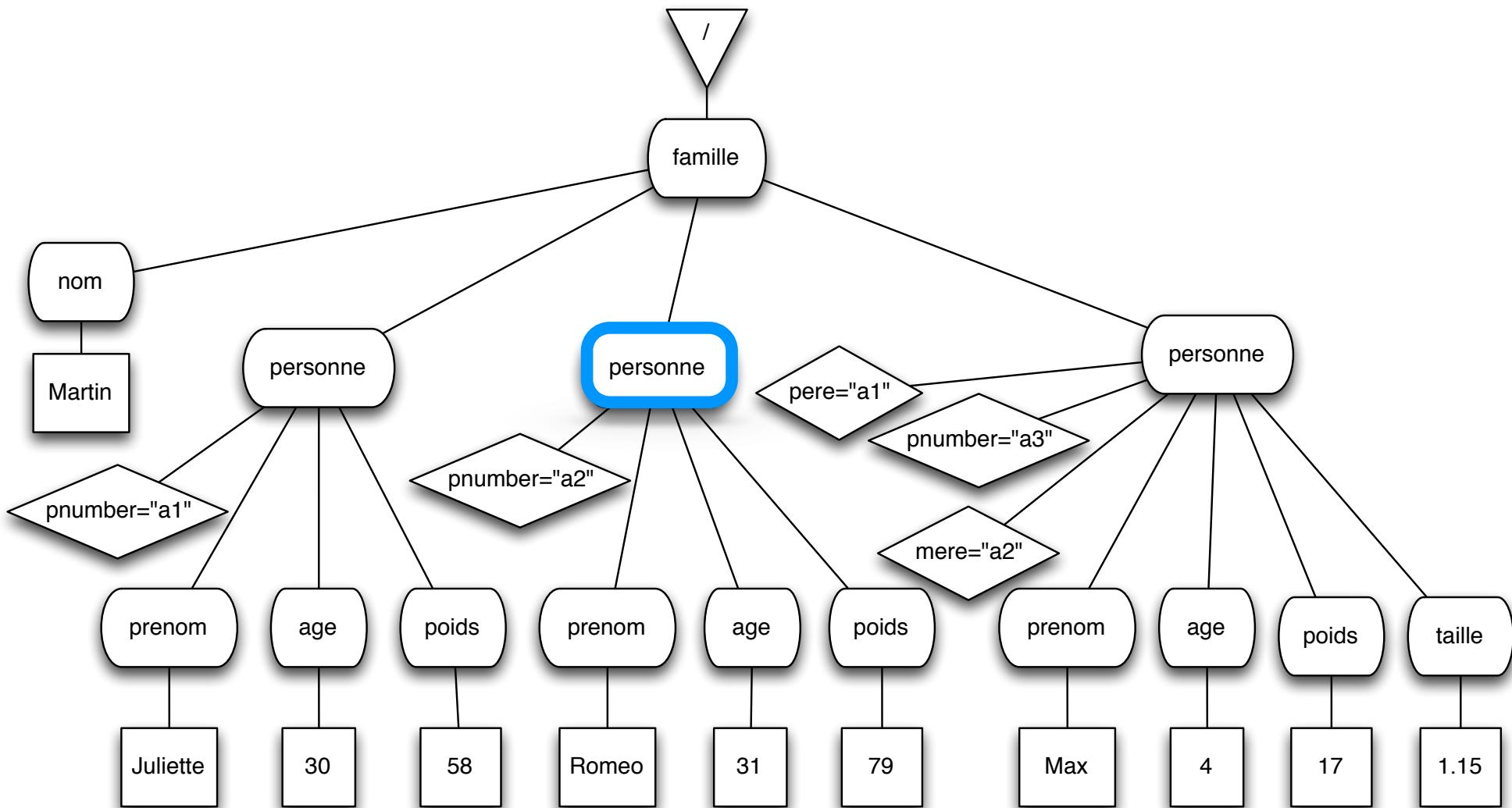
```
/descendant::prenom[child::text()="Romeo"]/*/parent::*/attribute::pnumber
```



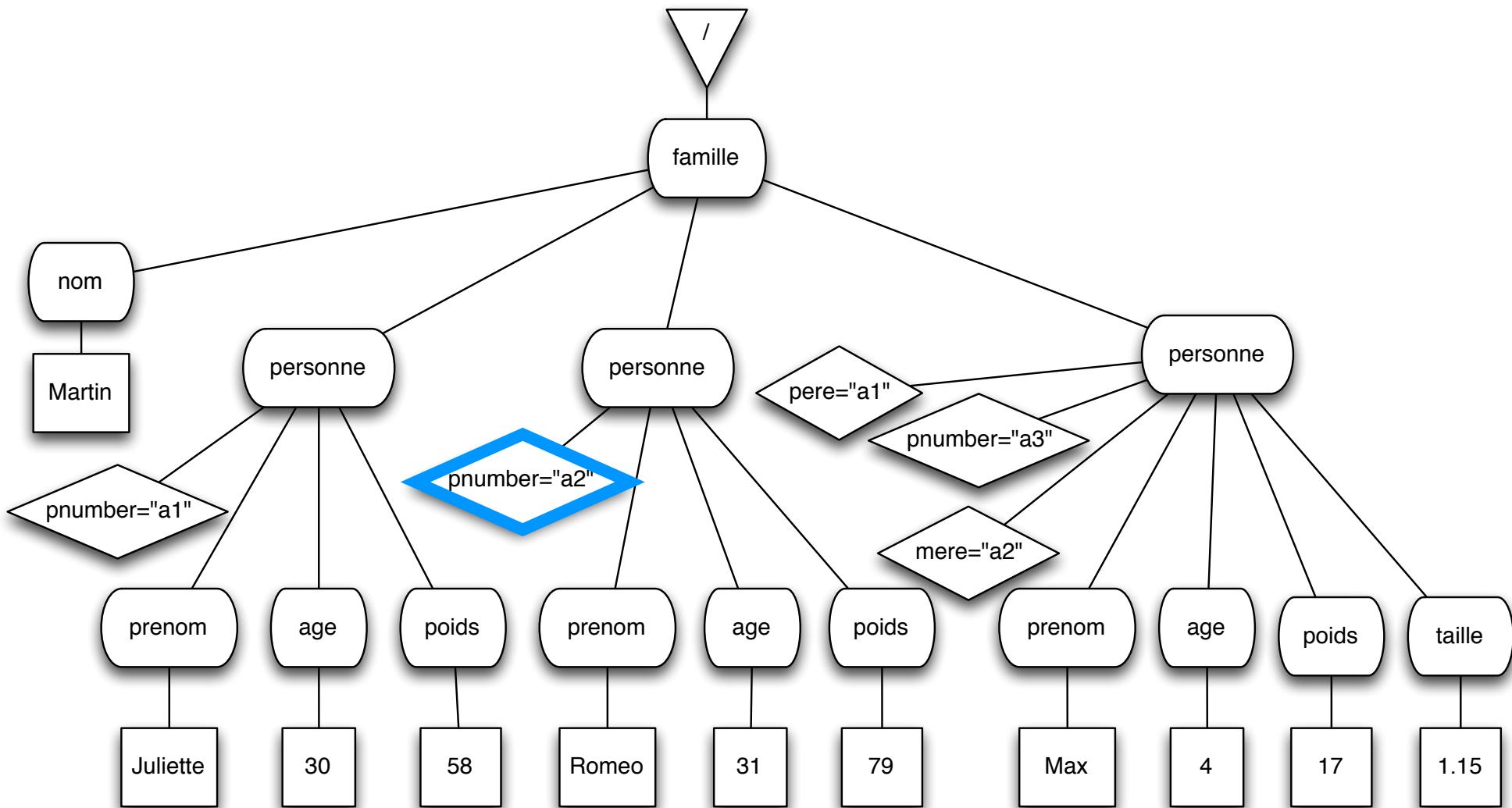
`/descendant::prenom[child::text()="Romeo"] /parent::* /attribute::pnumber`



`/descendant::prenom[child::text()="Romeo"]/parent::*[attribute::pnumber`



`/descendant::prenom[child::text()="Romeo"]/parent::*[attribute::pnumber]`



Sémantique existentielle des comparaisons

```
//livre[auteur = editeur]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<livres>
  <livre>
    <titre>Bonjour chez vous</titre>
    <auteur>Patrick McGoohan</auteur>
    <editeur>Patrick McGoohan</editeur>
  </livre>
</livres>
```

Sémantique existentielle des comparaisons

```
//livre[auteur = editeur]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<livres>
  <livre>
    <titre>Bonjour chez vous</titre>
    <auteur>John Smith</auteur>
    <auteur>Patrick McGoohan</auteur>
    <editeur>James Kaith</editeur>
    <editeur>Patrick McGoohan</editeur>
  </livre>
</livres>
```

Sémantique existentielle des comparaisons

```
//livre[auteur = editeur]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<livres>
  <livre>
    <titre>Bonjour chez vous</titre>
    <auteur>John Smith</auteur>
    <auteur>Will Stuart</auteur>
    <editeur>James Kaith</editeur>
    <editeur>Patrick McGoohan</editeur>
  </livre>
</livres>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<famille id="MARTIN">
    <femme id="1">
        <prenom>Juliette</prenom>
        <age>63</age>
        <poids>58</poids>
    </femme>
    <homme id="2">
        <prenom>Romeo</prenom>
        <age>65</age>
        <poids>97</poids>
    </homme>
    <homme id="3">
        <prenom>Max</prenom>
        <age>25</age>
        <poids>73</poids>
        <pere>2</pere>
        <mere>1</mere>
    </homme>
    <femme id="4">
        <prenom>Marie</prenom>
        <age>18</age>
        <poids>54</poids>
        <pere>2</pere>
        <mere>1</mere>
    </femme>
    <homme id="5">
        <prenom>Paul</prenom>
        <age>5</age>
        <poids>10</poids>
        <pere>3</pere>
    </homme>
</famille>
```

le prénom des personnes les plus lourdes

~~/famille/*[not(poids < //poids)]/prenom/text()~~

Romeo

~~/famille/*[poids >= //poids]/prenom/text()~~

Juliette

Romeo

Max

Marie

Paul

```

<?xml version="1.0" encoding="UTF-8"?>
<famille id="MARTIN">
    <femme id="1">
        <prenom>Juliette</prenom>
        <age>63</age>
        <poids>58</poids>
    </femme>
    <homme id="2">
        <prenom>Romeo</prenom>
        <age>65</age>
        <poids>97</poids>
    </homme>
    <homme id="3">
        <prenom>Max</prenom>
        <age>25</age>
        <poids>73</poids>
        <pere>2</pere>
        <mere>1</mere>
    </homme>
    <femme id="4">
        <prenom>Marie</prenom>
        <age>18</age>
        <poids>54</poids>
        <pere>2</pere>
        <mere>1</mere>
    </femme>
    <homme id="5">
        <prenom>Paul</prenom>
        <age>5</age>
        <poids>10</poids>
        <pere>3</pere>
    </homme>
</famille>

```

les id des hommes qui ne sont père d'aucune personne

~~//homme[@id != //pere]/@id~~

2
3
5

~~//homme[not(@id=//pere)]/@id~~

5

```
<?xml version="1.0" encoding="UTF-8"?>
<famille id="MARTIN">
    <femme id="1">
        <prenom>Juliette</prenom>
        <age>30</age>
        <poids>58</poids>
    </femme>
    <homme id="2">
        <prenom>Romeo</prenom>
        <age>31</age>
        <poids>97</poids>
    </homme>
    <homme id="3">
        <prenom>Max</prenom>
        <age>4</age>
        <poids>12</poids>
        <taille>1.25</taille>
        <pere>2</pere>
        <mere>1</mere>
    </homme>
    <femme id="4">
        <prenom>Marie</prenom>
        <age>3</age>
        <poids>18</poids>
        <taille>1.10</taille>
        <pere>2</pere>
        <mere>1</mere>
    </femme>
</famille>
```

//age union //poids

<age>30</age>
<poids>58</poids>
<age>31</age>
<poids>97</poids>
<age>4</age>
<poids>12</poids>
<age>3</age>
<poids>18</poids>

```
<?xml version="1.0" encoding="UTF-8"?>
<famille id="MARTIN">
    <femme id="1">
        <prenom>Juliette</prenom>
        <age>63</age>
        <poids>58</poids>
    </femme>
    <homme id="2">
        <prenom>Romeo</prenom>
        <age>65</age>
        <poids>97</poids>
    </homme>
    <homme id="3">
        <prenom>Max</prenom>
        <age>25</age>
        <poids>73</poids>
        <pere>2</pere>
        <mere>1</mere>
    </homme>
    <femme id="4">
        <prenom>Marie</prenom>
        <age>18</age>
        <poids>54</poids>
        <pere>2</pere>
        <mere>1</mere>
    </femme>
    <homme id="5">
        <prenom>Paul</prenom>
        <age>5</age>
        <poids>10</poids>
        <pere>3</pere>
    </homme>
</famille>
```

tous les prénoms sans doublon

distinct-values(doc("famille.xml")//prenom)

Examples (1)

`child::A/descendant::B` : `B` elements, descendant of an `A` element, itself child of the context node;
Can be abbreviated to `A//B`.

`child::* / child::B` : all the `B` grand-children of the context node;
`descendant-or-self::B` : elements `B` descendants of the context node, **plus** the context node itself if its name is `B`.

`child::B [position () = last ()]` : the last child named `B` of the context node.
Abbreviated to `B [last ()]`.

`following-sibling::B [1]` : the first sibling of type `B` (in the document order) of the context node,

Examples (2)

/descendant::B[10] : the tenth element of type B in the document.

Not: the tenth element of the document, if its type is B!

child::B[child::C] : child elements B that have a child element C.

Abbreviated to B[C].

/descendant::B[@att1 or @att2] : elements B that have an attribute att1 or an attribute att2;

Abbreviated to //B[@att1 or @att2]

* [self::B or self::C] : children elements named B or C

Outline

1 Introduction

2 Path Expressions

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

7 Exercise

XPath 2.0

An extension of XPath 1.0, backward compatible with XPath 1.0. Main differences:

Improved data model tightly associated with XML Schema.

- ⇒ a new **sequence** type, representing ordered set of nodes and/or values, with duplicates allowed.
- ⇒ XSD types can be used for node tests.

More powerful new operators (loops) and better control of the output (limited tree restructuring capabilities)

Extensible Many new built-in functions; possibility to add user-defined functions.

XPath 2.0 is **also** a subset of XQuery 1.0.

Path expressions in XPath 2.0

New node tests in XPath 2.0:

`item()` any node or atomic value

`element()` any element (eq. to `child::*` in XPath 1.0)

`element(author)` any element named `author`

`element(*, xs:person)` any element of type `xs:person`

`attribute()` any attribute

Nested paths expressions:

Any expression that returns a sequence of nodes can be used as a step.

`/book / (author | editor) /name`

Outline

1 Introduction

2 Path Expressions

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

7 Exercise

XPath 1.0 Implementations

Large number of implementations.

[libxml2](#) Free **C** library for parsing XML documents, supporting XPath.

[java.xml.xpath](#) **Java** package, included with JDK versions starting from 1.5.

[System.Xml.XPath](#) **.NET** classes for XPath.

[XML::XPath](#) Free **Perl** module, includes a command-line tool.

[DOMXPath](#) **PHP** class for XPath, included in PHP5.

[PyXML](#) Free **Python** library for parsing XML documents, supporting XPath.

References

- <http://www.w3.org/TR/xpath>
- *XML in a nutshell*, Elliotte Rusty Harold & W. Scott Means, O'Reilly
- *XPath 2.0 Programmer's Reference*, Michael Kay, Wrox

Outline

1 Introduction

2 Path Expressions

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

7 Exercise

Exercise

```
<a>
  <b><c /></b>
  <b id="3" di="7">bli <c /><c><e>bla</e></c></b>
  <d>bou</d>
</a>
```

We suppose that all text nodes containing only whitespace are removed from the tree.

- Give the result of the following XPath expressions:
 - ▶ //e/preceding::text ()
 - ▶ count (/c | /b/node ())
- Give an XPath expression for the following problems, and the corresponding result:
 - ▶ Sum of all attribute values
 - ▶ Text content of the document, where every “b” is replaced by a “c”
 - ▶ Name of the child of the last “c” element in the tree

Table des matières

- 1 Introduction
- 2 Typage de données XML
- 3 Navigation avec XPath
- 4 Transformations XSLT
- 5 Programmation avec XQuery
- 6 APIs pour la lecture des fichiers XML

XSLT

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

March 20, 2013

What is XSLT?

XSLT = a specialized language for transforming an XML document into another XML document.

Main principles:

- An XSLT program, or **stylesheet**, consists of rules, or **templates**.
- A template applies to a specific kind of node of the input document, and produces a fragment of the output document.
 - ▶ by creating **literal nodes**,
 - ▶ by **copying values** and **fragments** from the input document,
 - ▶ by **instantiating** (= calling) other templates.
- Execution model: initially, a template is applied to the **root node** of the input document
⇒ this first template may initiate a traversal of the input document.

Remark

An XSLT stylesheet is an XML document! XSLT element names are prefixed by (typically) `xsl:` that refers to the XSLT namespace.

A Hello World! Stylesheet

```
<?xml version="1.0" encoding="utf-8"?>

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

  <xsl:output method="xml" encoding="utf-8" />

  <xsl:template match="/">
    <hello>world</hello>
  </xsl:template>

</xsl:stylesheet>
```

General structure of a stylesheet:

- A top-level `<xsl:stylesheet>` element
- Some **declarations** (all elements except `<xsl:template>` ones)
- Some **template rules**, in this case a template that applies to the root node.

Invocation of an XSLT Stylesheet

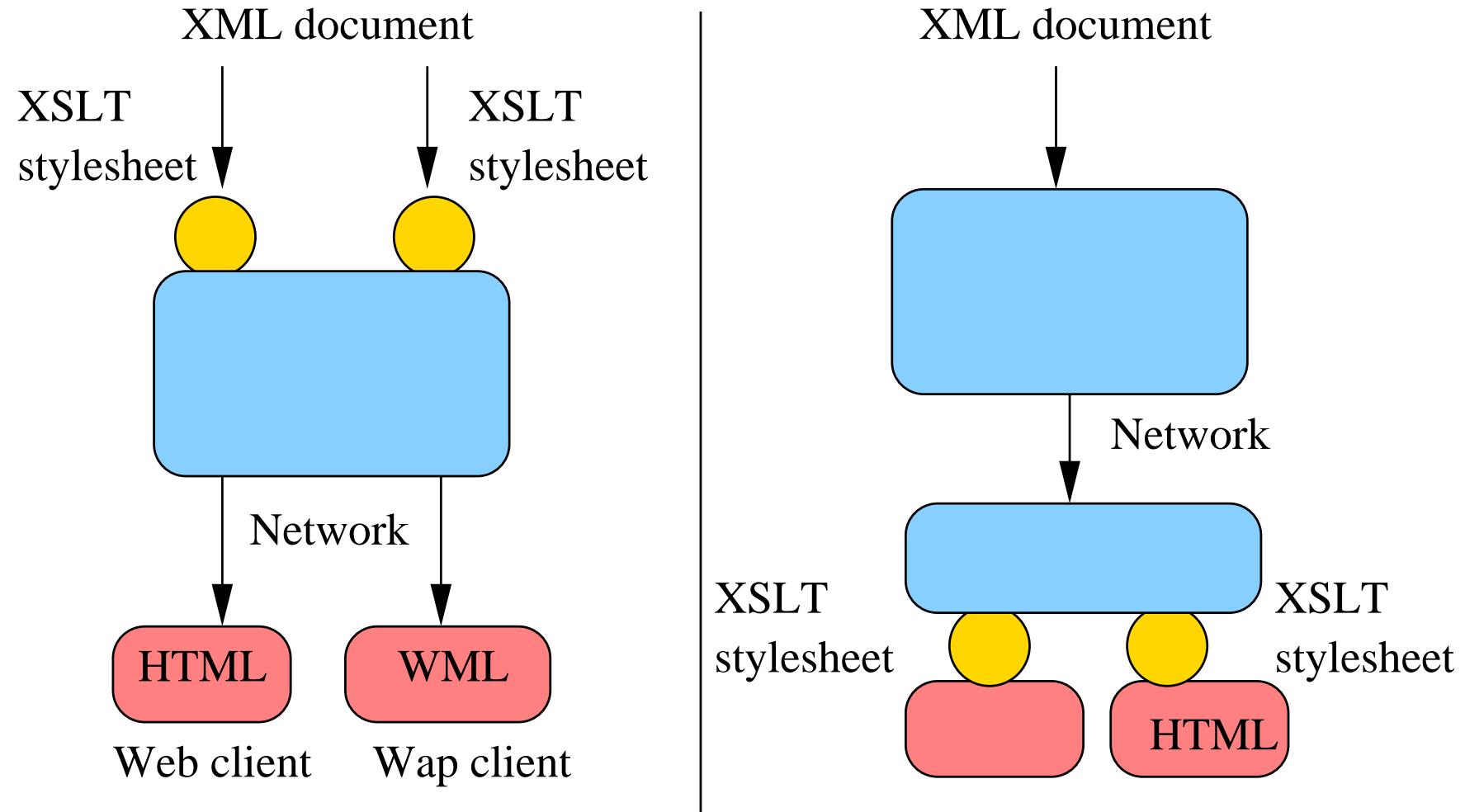
An XSLT stylesheet may be invoked:

- **Programmatically**, through one of the various XSLT libraries.
- Through a **command line** interface.
- In a Web Publishing context, by including a styling processing instruction in the XML document

```
<?xml-stylesheet  
    href="toto.xsl" type="text/xsl" ?>  
  
<doc>  
    <titi />  
</doc>
```

- ▶ the transformation can be processed on the **server side** by a CGI, PHP, ASP, JSP... script
- ▶ or on the **client side** through the XSLT engines integrated to most browsers.

Web Publishing with XSLT



The `<xsl:template>` Element

```
<xsl:template match="book">
    The book title is:
        <xsl:value-of select="title" />

    <h2>Authors list</h2>
    <ul>
        <xsl:apply-templates select="authors/name" />
    </ul>
</xsl:template>
```

A template consists of

A pattern an XPath expression (restricted) which determines the node to which the template applies.

The pattern is the value of the `match` attribute.

A body an XML fragment (well-formed!) which is inserted in the output document when the template is instantiated.

XPath patterns in XSLT

The role of the XPath expression of the `match` attribute is quite specific: it describes the nodes which can be the target of a template instantiation. Those expressions are called **patterns**. They must comply to the following requirements

- A pattern always denotes a node set.
Example: `<xsl:template match='1'>` is incorrect.
- It must be easy to decide whether a node is denoted or not by a pattern.
Example: `<xsl:template match='preceding::*[12]'>` is meaningful, but quite difficult to evaluate.

Patterns syntax

A pattern is a valid XPath expression which uses only the `child` and `@` axes, and the abbreviation `//`. Predicates are allowed.

Pattern examples

Recall: a pattern is interpreted as the nodes to which a template applies.

- `<xsl:template match='B'>`
applies to any **B** element.
- `<xsl:template match='A/B'>`
applies to any **B** element, child of an **A** element.
- `<xsl:template match='@att1'>`
applies to any **att1** attribute, whatever its parent element.
- `<xsl:template match='A//@att1'>`
applies to any **att1** attribute, if its parent element is a descendant of an **A** element.

General rule

Given an XML tree T , a pattern P **matches** a node N if there exists a node C (the **context node**) in T such that $N \in P(T, C)$.

Content of a template body

Basically, the content of `<xsl:template>` may consist of:

- Literal elements and text.

Example: `<h2>Authors</h2>`. This creates in the output document an element **h2**, with a **Text** child node 'Authors'.

- Values and elements from the input document.

Example: `<xsl:value-of select='title' />`. This inserts in the output document a node set, result of the XPath expression **title**.

- Call to other templates.

Example: `<xsl:apply-templates select='authors' />`. Applies a template to each node in the node set result of the XPath expression **authors**.

Remark

Only the basic of XSLT programming! Many advanced features (modes, priorities, loops and tests) beyond this core description.

Instantiation of a `<xsl:template>`

Main principles:

- **Literal elements** (those that don't belong to the XSLT namespace) and **text** are simply copied to the output document.
- **Context node**: A template is always instantiated in the context of a node from the input document.
- **XPath expressions**: all the (relative) XPath expression found in the template are evaluated with respect to the context node (an exception: `<xsl:for-each>`).
- **Calls with `xsl:apply-templates`**: find and instantiate a template for each node selected by the XPath expression `select`.
- **Template call substitution**: any call to other templates is eventually replaced by the instantiation of these templates.

The `xsl:apply-templates` element

```
<xsl:apply-templates select="authors/name" />
```

select an XPath expression which, if relative, is interpreted with respect to the context node,

Note: the default value is `child::node()`, i.e., select all the children of the context node

mode a label which can be used to specify which kind of template is required.

The `xsl:apply-templates` mechanism

```
<xsl:template match="book">
  <ul><xsl:apply-templates
    select="authors/name" /></ul>
</xsl:template>

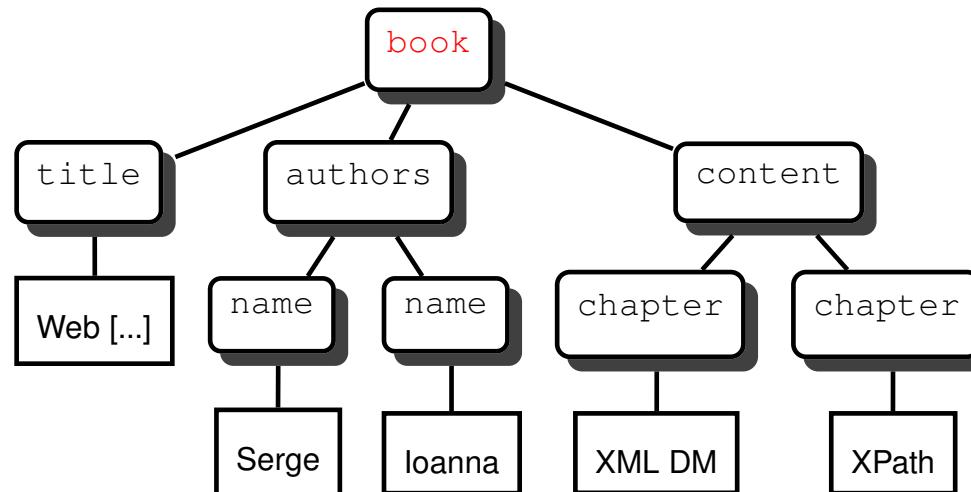
<xsl:template match="name">
  <li><xsl:value-of select=".." /></li>
</xsl:template>
```

```
<book>
  ...
  <authors>
    <name>Serge</name>
    <name>Ioana</name>
  </authors>
</book>
```



```
<ul>
  <li>Serge</li>
  <li>Ioana</li>
</ul>
```

Combined templates instantiation



```

<xsl:template match="book">
  Title: <xsl:value-of
    select="title" />

  <h2>Authors</h2>
  <ul><xsl:apply-templates
    select="authors/name" />
  </ul>
</xsl:template>
  
```



```

  Title: Web[...]

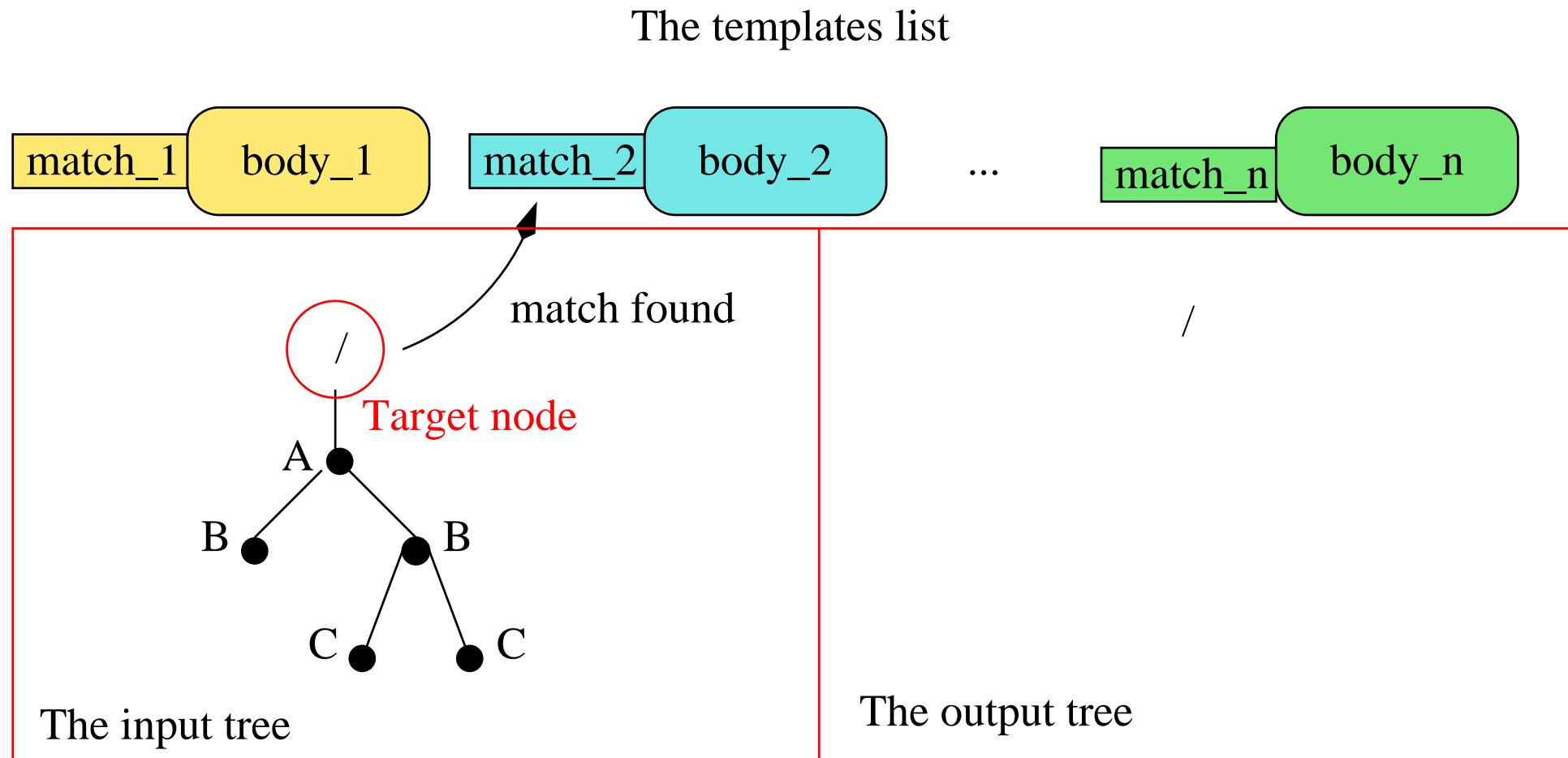
  <h2>Authors</h2>
  <ul>
    <li>Serge</li>
    <li>Ioanna</li>
  </ul>
  
```

The execution model of XSLT

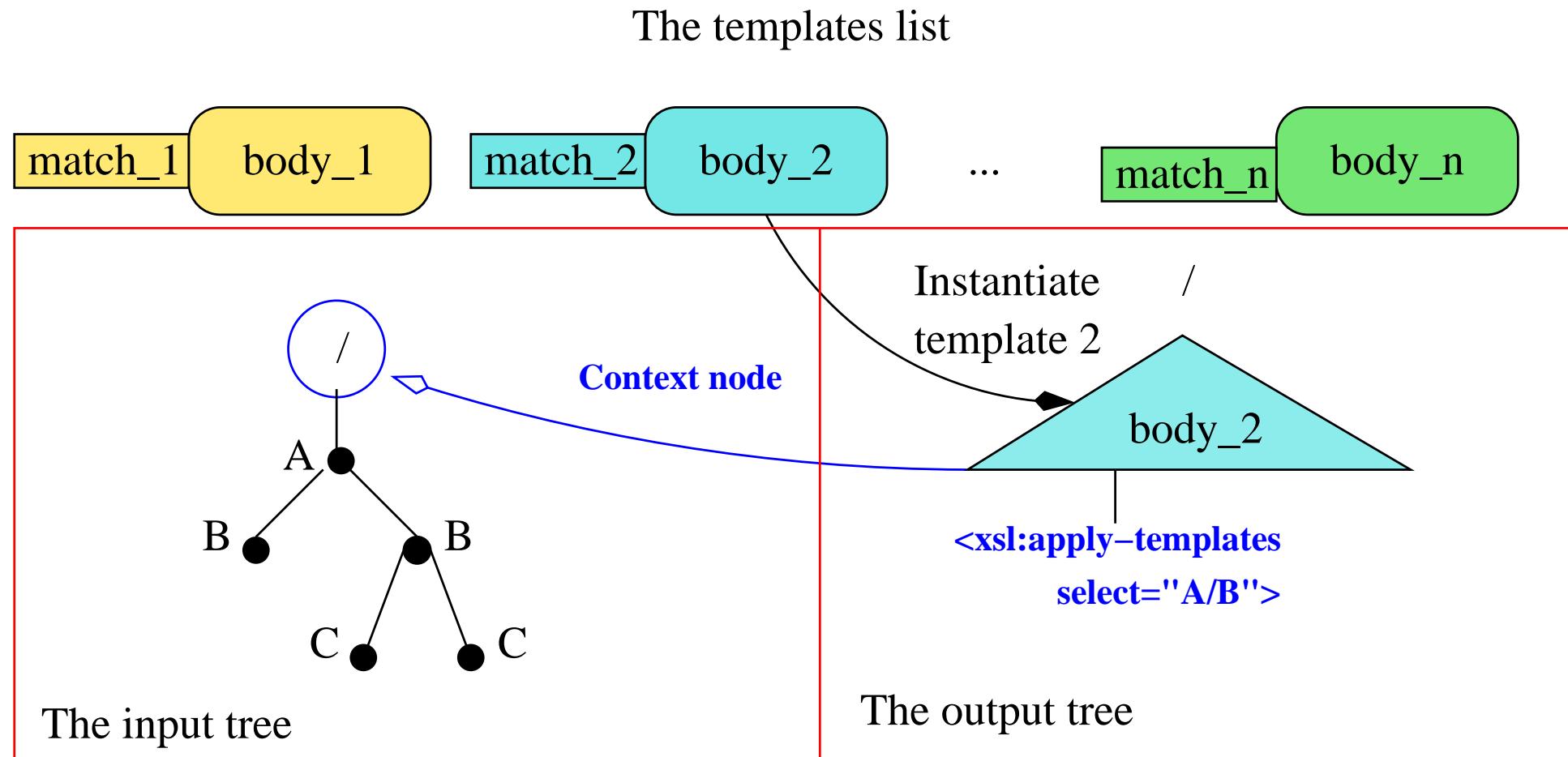
An XSLT stylesheet consists of a set of templates. The transformation of an input document I proceeds as follows:

- ➊ The engine considers the **root node** R of I , and selects the template that applies to R .
- ➋ The template body is copied in the output document O .
- ➌ Next, the engine considers all the `<xsl:apply-templates>` that have been copied in O , and evaluate the **select** XPath expression, taking R as context node.
- ➍ For each node result of the XPath evaluation, a template is selected, and its body replaces the `<xsl:apply-templates>` call.
- ➎ The process iterates, as new `<xsl:apply-templates>` may have been inserted in O .
- ➏ The transform terminates when O is free of **xsl:** instructions.

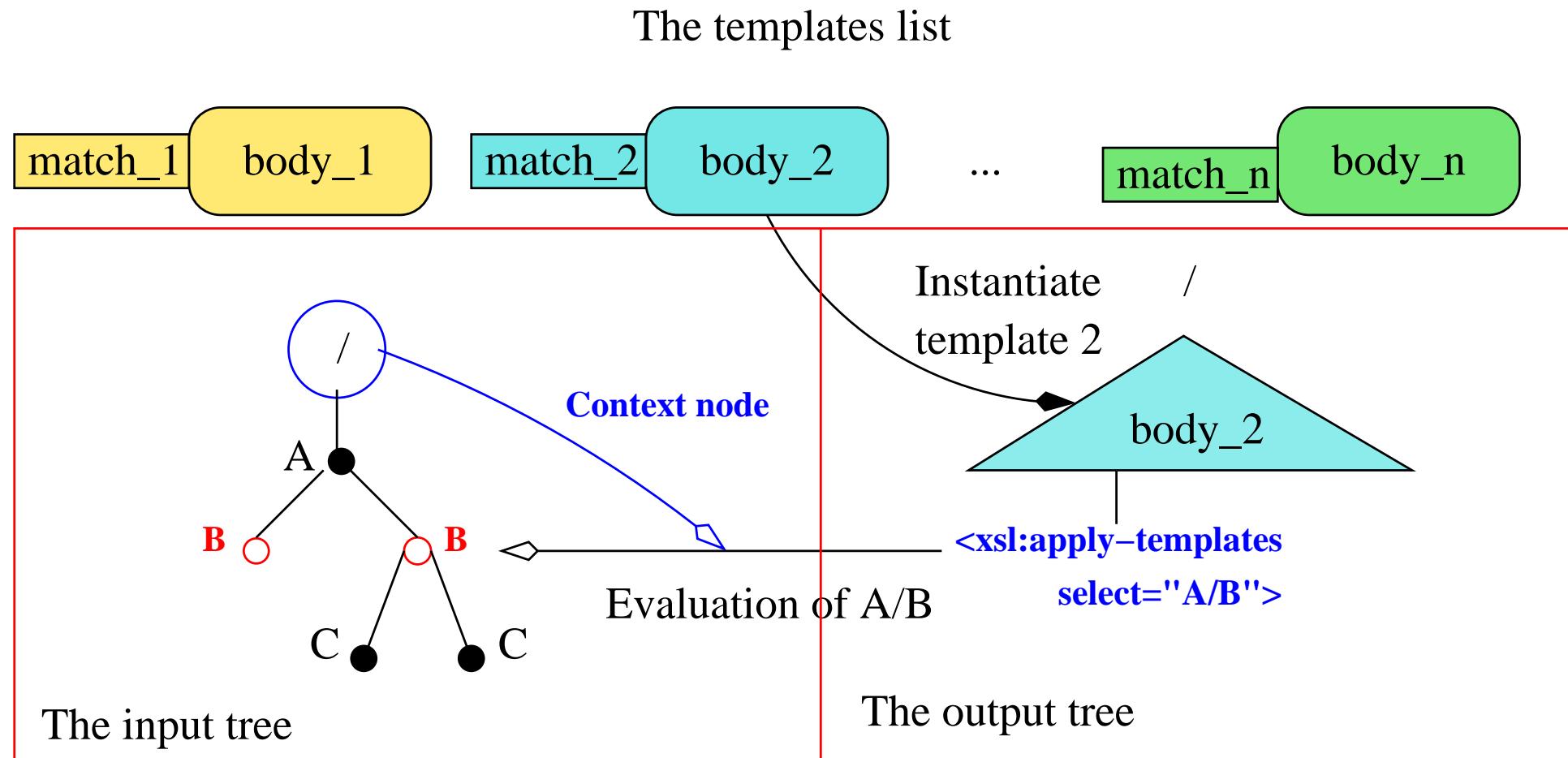
The execution model: illustration



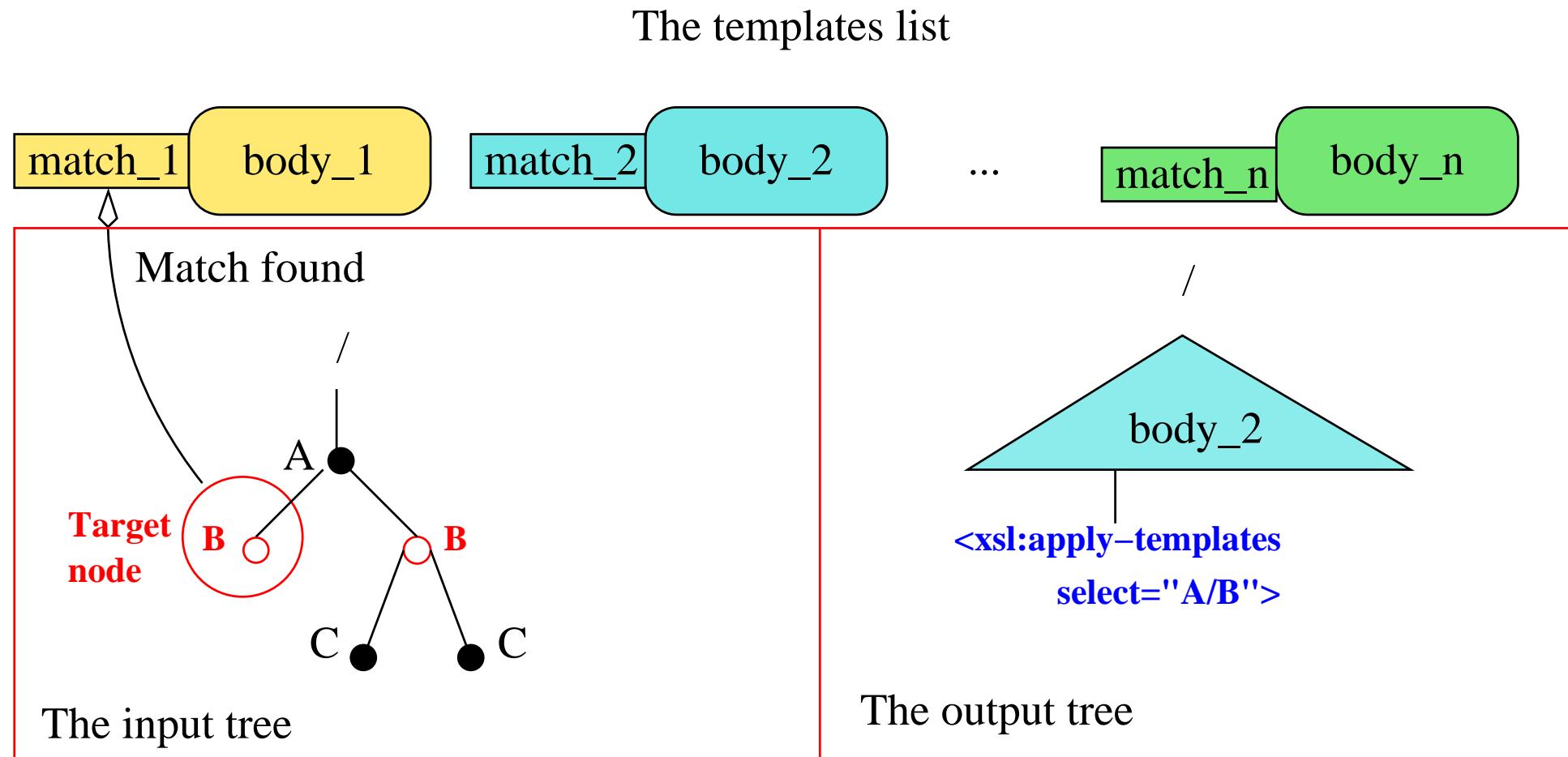
The execution model: illustration



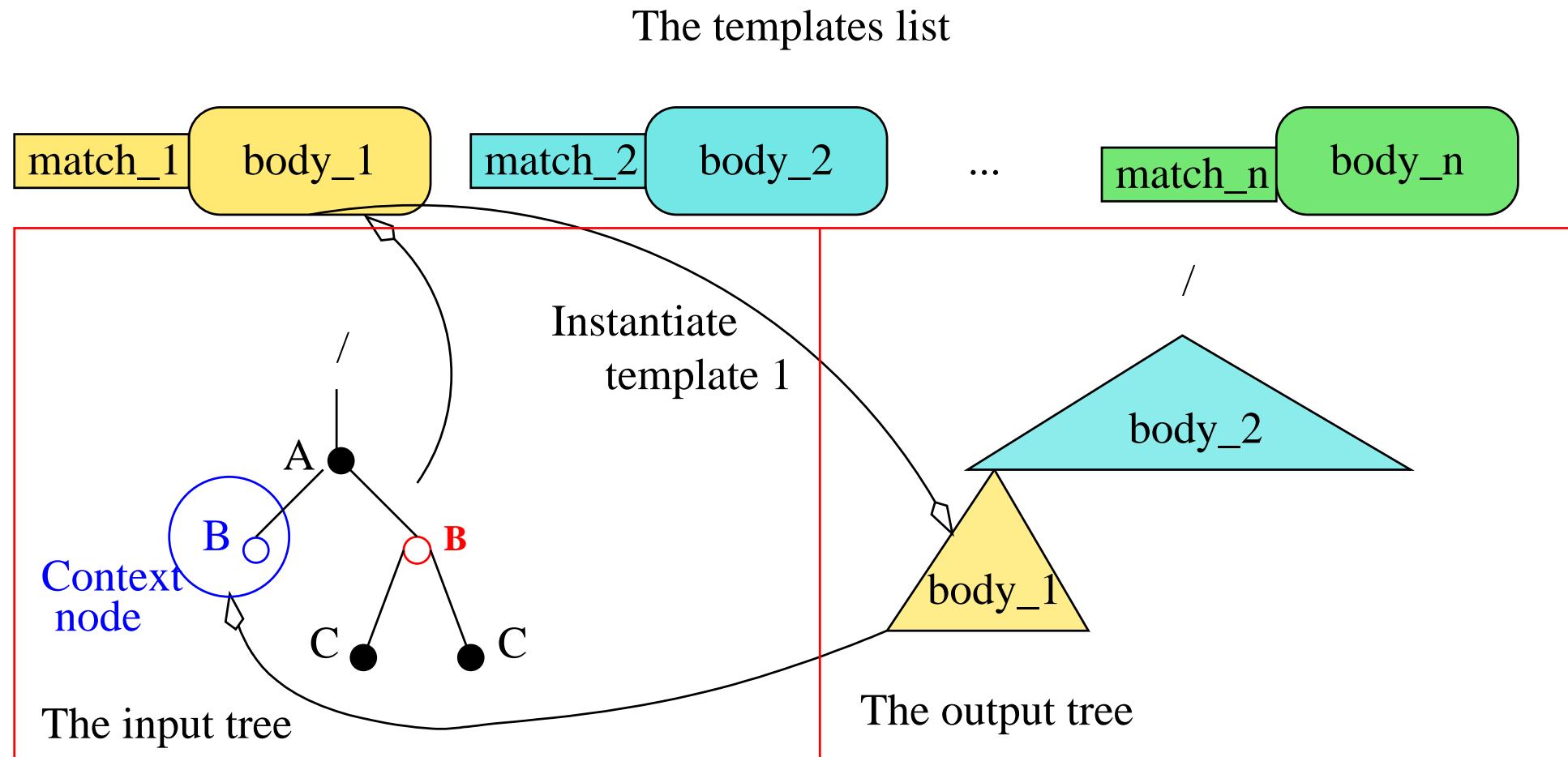
The execution model: illustration



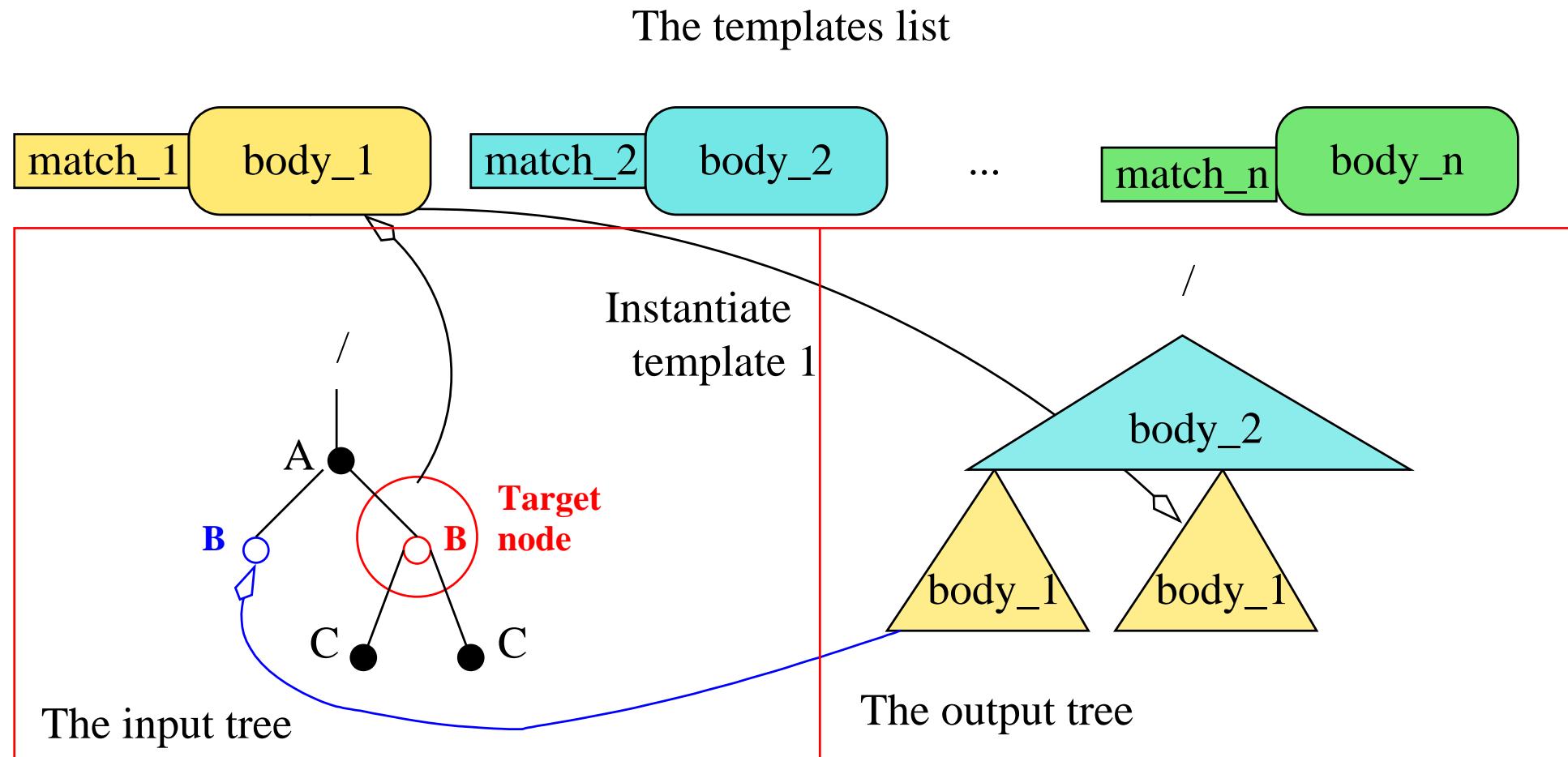
The execution model: illustration



The execution model: illustration

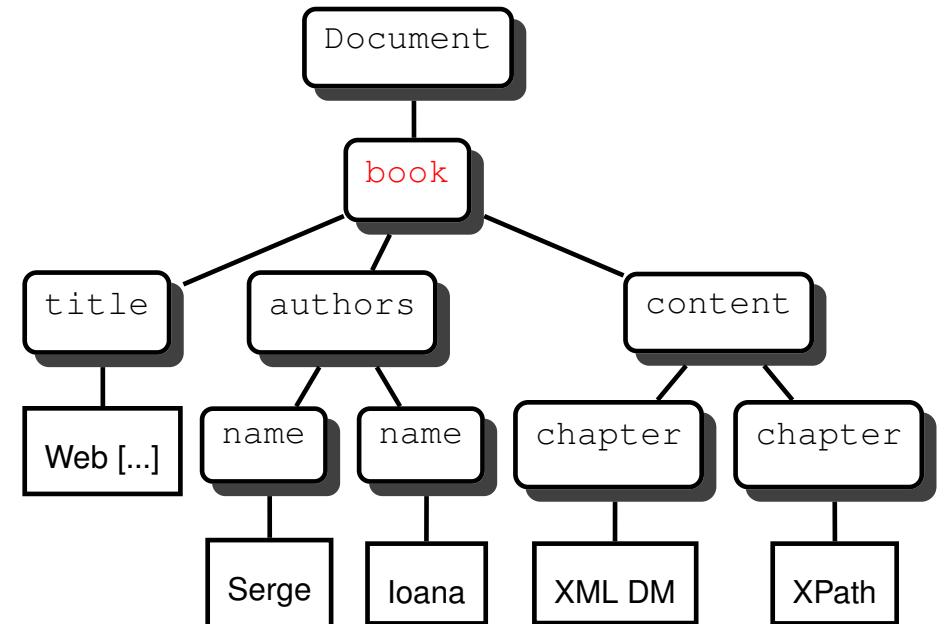


The execution model: illustration



The input document in serialized and tree forms

```
<?xml version="1.0"
      encoding="utf-8"?>
<book>
  <title>Web [...]</title>
  <authors>
    <name>Serge</name>
    <name>Ioana</name>
  </authors>
  <content>
    <chapter id="1">
      XML data model
    </chapter>
    <chapter id="2">
      XPath
    </chapter>
  </content>
</book>
```



The XSLT template that matches the root node

```
<xsl:template match="/">
  <html>
    <head>
      <title>
        <xsl:value-of select="/book/title"/>
      </title>
    </head>
    <body bgcolor="white">

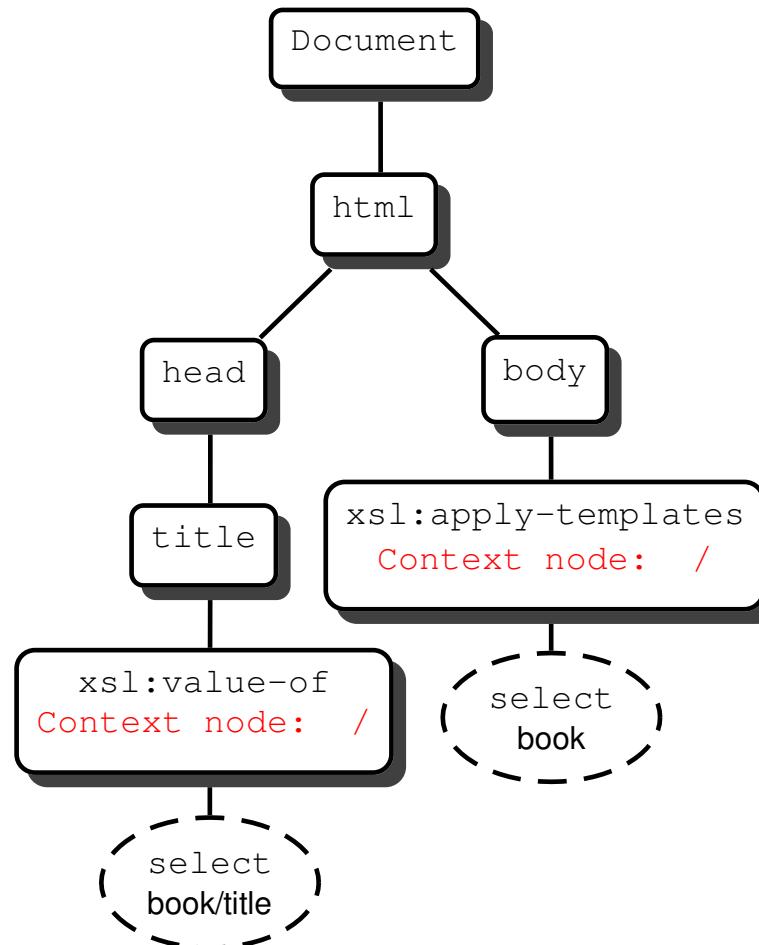
      <xsl:apply-templates select="book"/>

    </body>
  </html>
</xsl:template>
```

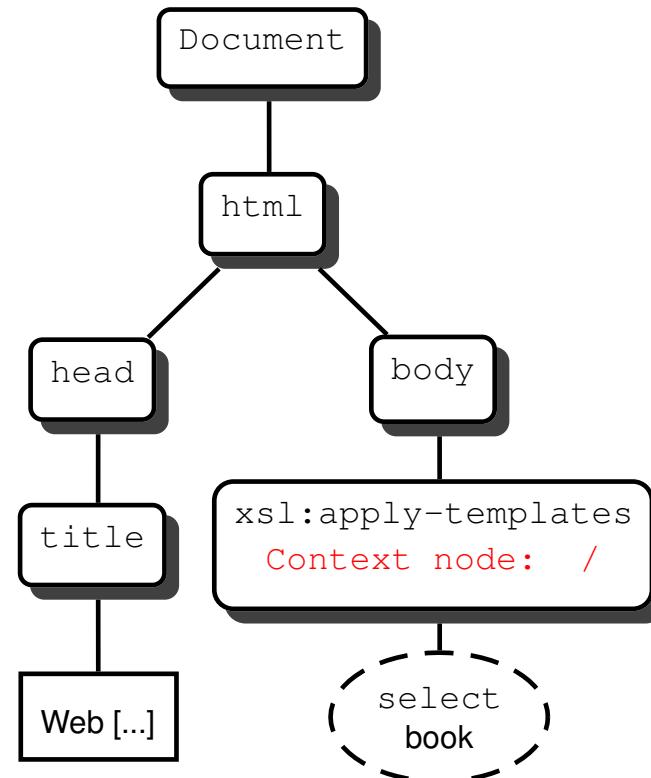
Remark

Typical of Web publishing templates.

The output document after instantiating the template



The output document after evaluation of `<xsl:value-of>`



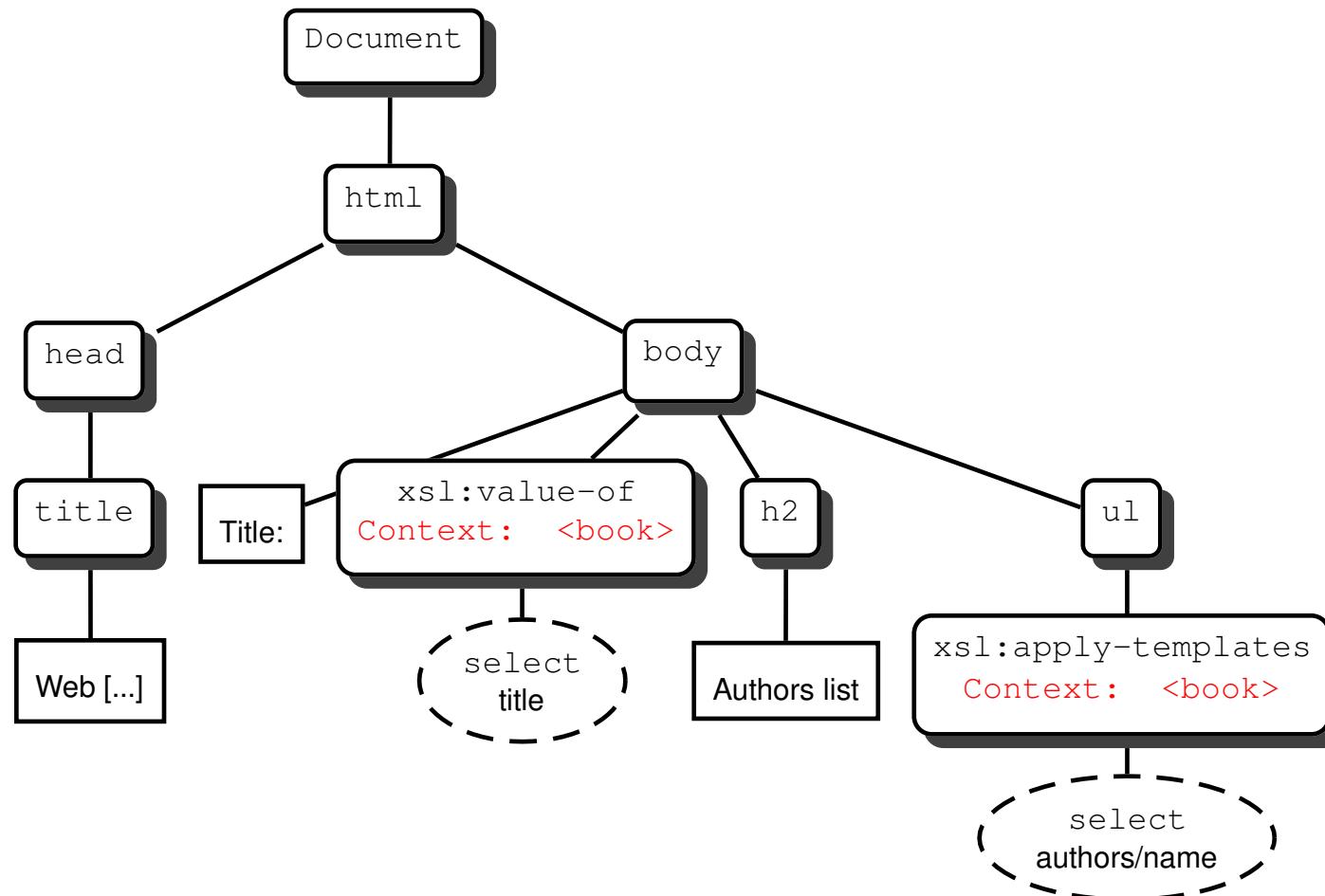
The remaining templates

```
<xsl:template match="book">
    Title:
        <xsl:value-of select="title" />

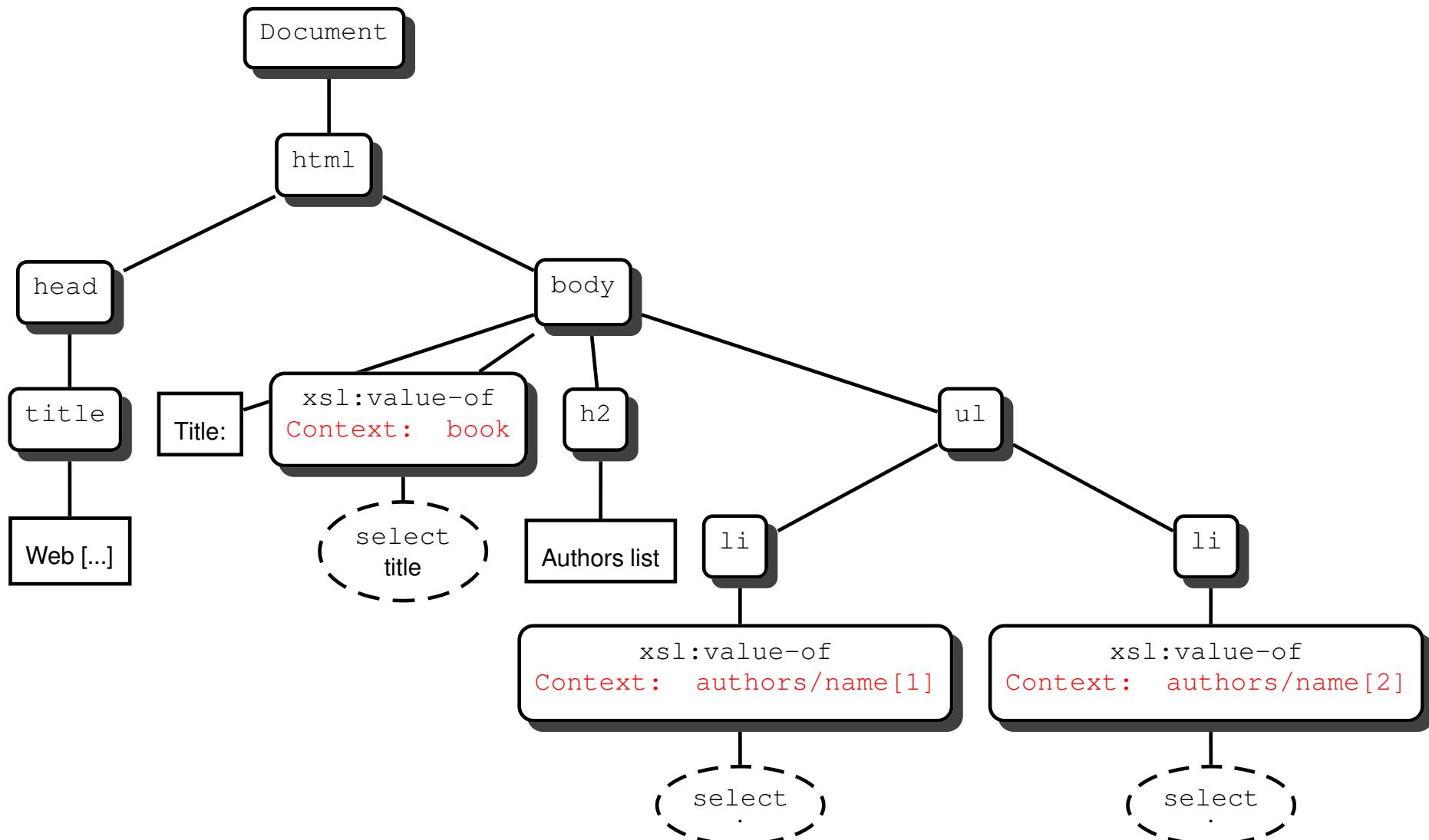
    <h2>Authors list</h2>
    <ul>
        <xsl:apply-templates select="authors/name" />
    </ul>
</xsl:template>

<xsl:template match="authors/name">
    <li><xsl:value-of select="." /></li>
</xsl:template>
```

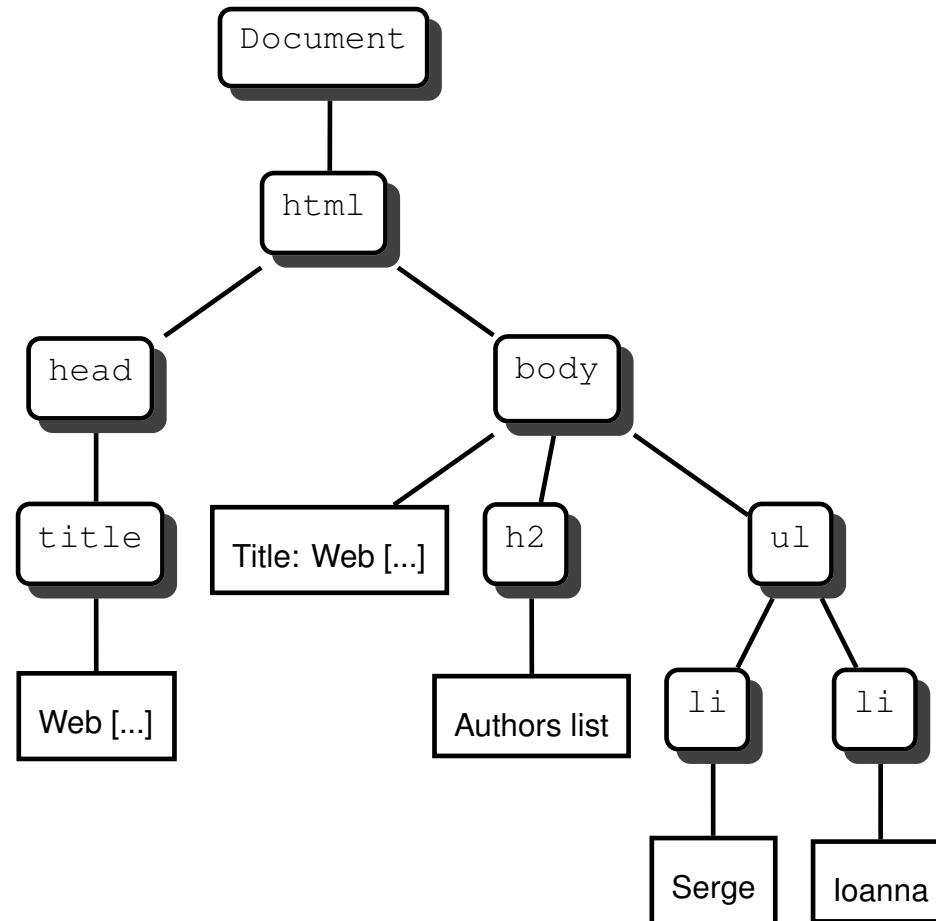
The `<book>` element is selected \Rightarrow the book template is instantiated



The `authors/name` template is instantiated twice, one for each author



The final output document



Advanced XSLT

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

March 20, 2013

Outline

1 Stylesheets revisited

2 XSLT programming

3 Complements

4 Reference Information

5 Beyond XSLT 1.0

The children elements of `xsl:stylesheet`

- `import` Import the templates of an XSLT program, with low priorities
- `include` Same as before, but no priority level
- `output` Gives the output format (default: `xml`)
- `param` Defines or imports a parameter
- `variable` Defines a variable.
- `template` Defines a template.

Also: `strip-space` and `preserve-space`, for, resp., removal or preservation of blank text nodes. Not presented here.

Stylesheet Output

```
<xsl:output  
    method="html"  
    encoding="iso-8859-1"  
    doctype-public="-//W3C//DTD HTML 4.01//EN"  
    doctype-system="http://www.w3.org/TR/html4/strict.dtd"  
    indent="yes"  
/>
```

- `method` is either `xml` (default), `html` or `text`.
- `encoding` is the desired encoding of the result.
- `doctype-public` and `doctype-system` make it possible to add a document type declaration in the resulting document.
- `indent` specifies whether the resulting XML document will be indented (default is `no`).

Importing Stylesheets

```
<xsl:import href="lib_templates.xsl" />
```

Templates are imported this way from another stylesheet. Their **precedence** is less than that of the local templates.

`<xsl:import>` must be the **first** declaration of the stylesheet.

```
<xsl:include href="lib_templates.xsl" />
```

Templates are included from another stylesheet. No precedence rule: works as if the included templates were local ones.

Template Rule Conflicts

- Rules from imported stylesheets are **overridden** by rules of the stylesheet which imports them.
- Rules with **highest priority** (as specified by the `priority` attribute of `<xsl:template>`) prevail. If no priority is specified on a template rule, a default priority is assigned according to the **specificity** of the XPath expression (the more specific, the highest).
- If there are still conflicts, it is an error.
- If no rules apply for the node currently processed (the document node at the start, or the nodes selected by a `<xsl:apply-templates>` instruction), **built-in** rules are applied.

Built-in templates

- A first built-in rule applies to the **Element** nodes and to the root node:

```
<xsl:template match="* | /">
  <xsl:apply-templates select="node() " />
</xsl:template>
```

Interpretation: recursive call to the children of the context node.

- Second built-in rule: applies to **Attribute** and **Text** nodes.

```
<xsl:template match="@* | text() ">
  <xsl:value-of select=". " />
</xsl:template>
```

Interpretation: copy the textual value of the context node to the output document.

Exercise: what happens when an empty stylesheet is applied to an XML document?

Global Variables and Parameters

```
<xsl:param name="nom" select="' John Doe' />  
  
<xsl:variable name="pi" select="3.14159" />
```

Global parameters are passed to the stylesheet through some **implementation-defined** way. The `select` attribute gives the default value, in case the parameter is not passed, as an XPath expression.

Global variables, as well as local variables which are defined in the same way inside template rules, are immutable in XSLT, since it is a side-effect-free language.

The `select` content may be replaced in both cases by the content of the `<xsl:param>` or `<xsl:variable>` elements.

Outline

1 Stylesheets revisited

2 XSLT programming

3 Complements

4 Reference Information

5 Beyond XSLT 1.0

Named templates

```
<xsl:template name="print">
  <xsl:value-of select="position()"/>:
    <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="*">
  <xsl:call-template name="print"/>
</xsl:template>

<xsl:template match="text()">
  <xsl:call-template name="print"/>
</xsl:template>
```

Named templates play a role analogous to functions in traditional programming languages.

Remark

A call to a named template does not change the context node.

Parameters

```
<xsl:template name="print">
  <xsl:param name="message" select="'nothing'"/>

  <xsl:value-of select="position()"/>:
    <xsl:value-of select="$message"/>
</xsl:template>

<xsl:template match="*">
  <xsl:call-template name="print">
    <xsl:with-param name="message"
      select="'Element node'"/>
  </xsl:call-template>
</xsl:template>
```

Same mechanism for `<xsl:apply-templates>`.

`param` describes the parameter received from a template

`with-param` defines the parameter sent to a template

Example: computing $n!$ with XSLT

```
<xsl:template name="factorial">
  <xsl:param name="n" />

  <xsl:choose>
    <xsl:when test="$n<1">1</xsl:when>
    <xsl:otherwise>
      <xsl:variable name="fact">
        <xsl:call-template name="factorial">
          <xsl:with-param name="n" select="$n - 1" />
        </xsl:call-template>
      </xsl:variable>
      <xsl:value-of select="$fact * $n" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Conditional Constructs: `<xsl:if>`

```
<xsl:template match="Movie">
  <xsl:if test="year < 1970">
    <xsl:copy-of select="." />
  </xsl:if>
</xsl:template>
```

`<xsl:copy-of>` makes a deep copy of a node set; `<xsl:copy>` copies the nodes without their descendant.

Remark

- An XSLT program is an XML document: we must use entities for < and &.
- XSLT is closely associated to XPath (node select, node matching, and here data manipulation)

Conditional Constructs: `<xsl:choose>`

```
<xsl:choose>
  <xsl:when test="$year mod 4">no</xsl:when>
  <xsl:when test="$year mod 100">yes</xsl:when>
  <xsl:when test="$year mod 400">no</xsl:when>
  <xsl:otherwise>yes</xsl:otherwise>
</xsl:choose>

<xsl:value-of select="count(a)"/>
<xsl:text> item</xsl:text>
<xsl:if test="count(a)>1">s</xsl:if>
```

`<xsl:otherwise>` is optional. There can be any number of `<xsl:when>`, only the content of the first matching one will be processed.

Loops

`<xsl:for-each>` is an instruction for looping over a set of nodes.

It is more or less an alternative to the use of

`<xsl:template> / <xsl:apply-templates>`.

- The set of nodes is obtained with an XPath expression (attribute `select`);
- Each node of the set becomes in turn the **context node** (which temporarily replaces the template context node).
- The body of `<xsl:for-each>` is instantiated for each context node.

⇒ no need to call another template: somewhat simpler to read, and likely to be more efficient.

The `<xsl:for-each>` element

Example (`<xsl:sort>` is optional):

```
<xsl:template match="person">
[ . . . ]
  <xsl:for-each select="child">
    <xsl:sort select="@age" order="ascending"
              data-type="number"/>

    <xsl:value-of select="name" />
    <xsl:text> is </xsl:text>
    <xsl:value-of select="@age" />
  </xsl:for-each>
[ . . . ]
</xsl:template>
```

`<xsl:sort>` may also be used as a direct child of an
`<xsl:apply-templates>` element.

Variables in XSLT

A variable is a (*name, value*) pair. It may be defined

- Either as the result of an XPath expression

```
<xsl:variable name='pi' select='3.14116' />
```

```
<xsl:variable name='children' select='//child' />
```

- or as the content of the `<xsl:variable>` element.

```
<xsl:variable name="Signature">  
    Franck Sampori<br/>  
    Institution: INRIA<br/>  
    Email: <i>franck.sampori@inria.fr</i>  
</xsl:variable>
```

Remark

A variable has a **scope** (all its siblings, and their descendants) and **cannot** be redefined within this scope.

Outline

1 Stylesheets revisited

2 XSLT programming

3 Complements

4 Reference Information

5 Beyond XSLT 1.0

Other XSLT features

Many other instructions and functionalities. In brief:

Control of text output `<xsl:text>` , `<xsl:strip-space>` ,
`<xsl:preserve-space>` , and `normalize-space` function;

Dynamic creation of elements and attributes `<xsl:element>` and
`<xsl:attribute>` .

Multiple document input, and multiple document output document function,
`<xsl:document>` element (XSLT 2.0, but widely implemented in XSLT 1.0 as an extension function)

Generation of hypertext documents with links and anchors `generate-id` function

⇒ See the Exercises and projects.

Handling Whitespace and blank nodes

Main rules

All the whitespace is kept in the input document, **including** blank nodes.

All the whitespace is kept in the XSLT program document, **except** blank nodes.

Handling Whitespace explicitly:

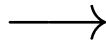
```
<xsl:strip-space elements="*" />  
<xsl:preserve-space elements="para poem" />
```

`<xsl:strip-space>` specifies the set of nodes whose whitespace-only text child nodes will be removed, and `<xsl:preserve-space>` allows for exceptions to this list.

Dynamic Elements and dynamic attributes

```
<xsl:element name="{concat('p', @age)}"
  namespace="http://ns">
  <xsl:attribute name="name">
    <xsl:value-of select="name" />
  </xsl:attribute>
</xsl:element>
```

```
<person age="12">
  <name>titi</name>
</person>
```



```
<p12
  name="titi"
  xmlns="http://ns" />
```

Remark

The value of the `name` attribute is here an **attribute template**: this attribute normally requires a string, not an XPath expression, but XPath expressions between curly braces are evaluated. This is often used with literal result elements: `<toto titi="${$var + 1}" />`.

Working with multiple documents

`document ($s)` returns the **document node** of the document at the URL `$s`

Example: `document ("toto.xml") /*`

Note: `$s` can be computed dynamically (e.g., by an XPath expression). The result can be manipulated as the root node of the returned document.

Unique identifiers

```
<xsl:template match="Person">
  <h2 id="{generate-id(.)}">
    <xsl:value-of
      select="concat(first_name, ' ', last_name)"/>
  </h2>
</xsl:template>
```

`generate-id($s)` returns a **unique identifier string** for the first node of the nodeset `$s` in document order.

Useful for testing the identity of two different nodes, or to generate HTML anchor names.

Outline

- 1 Stylesheets revisited
- 2 XSLT programming
- 3 Complements
- 4 Reference Information
- 5 Beyond XSLT 1.0

XSLT 1.0 Implementations

- Browsers** All modern browsers (Internet Explorer, Firefox, Opera, Safari) include XSLT engines, used to process `xmlstylesheet` references. Also available via **JavaScript**, with various interfaces.
- libxslt** Free **C** library for XSLT transformations. Includes `xsltproc` command-line tool. **Perl** and **Python** wrappers exist.
- Sablotron** Free **C++** XSLT engine.
- Xalan-C++** Free **C++** XSLT engine.
- JAXP** **Java** API for Transformation. Common interface for various JAVA XSLT engines (e.g., SAXON, Xalan, Oracle). Starting from JDK 1.4, a version of Xalan is bundled with Java.
- System.Xml** **.NET** XML and XSLT library.
- php-xslt** XSLT extension for **PHP**, based on Sablotron.
- 4XSLT** Free XSLT **Python** library.

References

- <http://www.w3.org/TR/xslt>
- *XML in a nutshell*, Eliotte Rusty Harold & W. Scott Means, O'Reilly
- *Comprendre XSLT*, Bernd Amman & Philippe Rigaux, O'Reilly

Outline

1 Stylesheets revisited

2 XSLT programming

3 Complements

4 Reference Information

5 Beyond XSLT 1.0

Limitations of XSLT 1.0

- Impossible to **process a temporary tree** stored into a variable (with
`<xsl:variable name="t"><toto a="3"/></xsl:variable>`).
Sometimes indispensable!
- **Manipulation of strings** is not very easy.
- **Manipulation of sequences** of nodes (for instance, for extracting all nodes with a distinct value) is awkward.
- Impossible to define in a portable way **new functions** to be used in XPath expressions. Using named templates for the same purpose is often verbose, since something equivalent to $y = f(2)$ needs to be written as:

```
<xsl:variable name="y">
  <xsl:call-template name="f">
    <xsl:with-param name="x" select="2" />
  </xsl:call-template>
</xsl:variable>
```

Extension Functions

XSLT allows for **extension functions**, defined in specific namespaces. These functions are typically written in a classical programming language, but the mechanism depends on the precise XSLT engine used. **Extension elements** also exist.

Once they are defined, such extension functions can be used in XSLT in the following way:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:math="http://exslt.org/math"
  version="1.0"
  extension-element-prefixes="math">

  ...

<xsl:value-of select="math:cos($angle)" />
```

EXSLT

EXSLT (<http://www.exslt.org/>) is a collection of extensions to XSLT which are portable across some XSLT implementations. See the website for the description of the extensions, and which XSLT engines support them (varies greatly). Includes:

`exsl:node-set` solves one of the main limitation of XSLT, by allowing to **process temporary trees** stored in a variable.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:exsl="http://exslt.org/common"
  version="1.0" extension-element-prefixes="exsl">

  ...

  <xsl:variable name="t"><toto a="3" /></xsl:variable>

  <xsl:value-of select="exsl:node-set($t) /*/@a" />
```

date library for formatting **dates** and **times**
math library of **mathematical** (in particular, **trigonometric**) functions
regexp library for **regular expressions**
strings library for manipulating **strings**

...

Other extension functions outside EXSLT may be provided by each XSLT engine.

XSLT 2.0

- W3C **Recommendation** (2007)
- Like XQuery 1.0, uses **XPath 2.0**, a much more powerful language than XPath 1.0:
 - ▶ **Strong typing**, in relation with XML Schemas
 - ▶ **Regular expressions**
 - ▶ **Loop** and **conditional** expressions
 - ▶ Manipulation of **sequences** of nodes and values
 - ▶ ...
- New functionalities in XSLT 2.0:
 - ▶ Native processing of **temporary trees**
 - ▶ **Multiple** output documents
 - ▶ **Grouping** functionalities
 - ▶ **User-defined** functions
 - ▶ ...
- All in all, XSLT 2.0 stylesheets tend to be much more **concise** and **readable** than XSLT 1.0 stylesheets.

Example XSLT 2.0 Stylesheet

Produces a list of each word appearing in a document, with their frequency.

(from *XSLT 2.0 Programmer's Reference*)

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <wordcount>
    <xsl:for-each-group group-by=". " select=
      "for $w in tokenize(string(.), '\W+')
       return lower-case($w)">
      <word word="{current-grouping-key() }"
           frequency="{count(current-group()) }"/>
    </xsl:for-each-group>
  </wordcount>
</xsl:template>

</xsl:stylesheet>
```

XSLT 2.0 Implementations

A few implementations.

SAXON **Java** and **.NET** implementation of XSLT 2.0 and XQuery 1.0.

The full version is commercial, but a **GPL version** is available without support of external XML Schemas.

Oracle XML Developer's Kit **Java** implementation of various XML technologies, including XSLT 2.0, XQuery 1.0, with full support of XML Schema. Commercial.

AltovaXML **Windows** implementation, with Java, .NET, COM interfaces. Commercial, schema-aware.

Gestalt **Eiffel open-source** XSLT 2.0 processor. Not schema-aware.
... a few others (Intel SOA Expressway, IBM WebSphere XML)

References

- <http://www.w3.org/TR/xpath20/>
- <http://www.w3.org/TR/xslt20/>
- *XPath 2.0 Programmer's Reference*, Michael Kay, Wrox
- *XSLT 2.0 Programmer's Reference*, Michael Kay, Wrox

Table des matières

- 1 Introduction
- 2 Typage de données XML
- 3 Navigation avec XPath
- 4 Transformations XSLT
- 5 Programmation avec XQuery
- 6 APIs pour la lecture des fichiers XML

XQuery

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

March 20, 2013

Why XQuery?

XQuery, the XML query language promoted by the W3C. See:

<http://www.w3.org/XML/Query>

Check your queries online (syntactic analysis):

<http://www.w3.org/2005/qt-applets/xqueryApplet.html>

Sample queries:

<http://www.w3.org/TR/xquery-use-cases/>

XQuery vs XSLT

- XSLT is a **procedural** language, good at transforming XML documents
- XQuery is a **declarative** language, good at efficiently retrieving some content from large (collections of) documents

Remark

In many cases, XSLT and XQuery can be used interchangeably. The choice is a matter of context and/or taste.

Main principles

The design of XQuery satisfies the following rules:

Closed-form evaluation. XQuery relies on a **data model**, and each query maps an instance of the model to another instance of the model.

Composition. XQuery relies on **expressions** which can be composed to form arbitrarily rich queries.

Type awareness. XQuery may associate an XSD schema to query interpretation. But XQuery also operates on schema-free documents.

XPath compatibility. XQuery is an extension of XPath 2.0 (thus, any XPath expression is **also** an XQuery expression).

Static analysis. Type inference, rewriting, optimisation: the goal is to exploit the declarative nature of XQuery for clever evaluation.

At a syntactic level, XQuery aims at remaining both concise and simple.

A simple model for document collections

A **value** is a **sequence** of 0 to n **items**.

An **item** is either a node or an atomic value.

There exist 7 kinds of nodes:

- **Document**, the document root;
- **Element**, named, mark the structure of the document;
- **Attributes**, named and valued, associated to an **Element**;
- **Text**, unnamed and valued;
- **Comment**;
- **ProcessingInstruction**;
- **Namespace**.

The model is quite general: everything is a **sequence of items**. This covers anything from a single integer value to wide collections of large XML documents.

Examples of values

The following are example of values

- `47` : a sequence with a single item (atomic value);
- `<a/>` : a sequence with a single item (**Element** node);
- `(1, 2, 3)` : a sequence with 3 atomic values.
- `(47, <a/>, "Hello")` : a sequence with 3 items, each of different kinds.
- `()` the empty sequence;
- an XML document;
- several XML documents (a **collection**).

Sequences: details

There is no distinction between an item and a sequence of length 1 ⇒ everything is a sequence.

Sequence **cannot** be nested (a sequence never contains another sequence)

The notion of “null value” does not exist in the XQuery model: a value is there, or not.

A sequence may be empty

A sequence may contain **heterogeneous** items (see previous examples).

Sequences are ordered: two sequences with the same set of items, but ordered differently, are different.

Items: details

Nodes have an **identity**; values do not.

Element and **Attribute** have **type annotations**, which may be inferred from the XSD schema (or unknown if the schema is not provided).

Nodes appear in a given order in their document. Attribute order is undefined.

Syntactic aspects of XQuery

XQuery is a case-sensitive language (keywords must be written in lowercase).

XQuery builds queries as **composition** of **expressions**.

An expression produces a **value**, and is side-effect free (no modification of the context, in particular variable values).

XQuery comments can be put anywhere. Syntax:

```
(:This is a comment :)
```

Evaluation context

An expression is always evaluated with respect to a **context**. It is a slight generalization of XPath and XSLT contexts, and includes:

- Bindings of namespace prefixes with namespaces URIs
- Bindings for variables
- In-scope functions
- A set of available collections and a default collection
- Date and time
- Context (current) node
- Position of the context node in the context sequence
- Size of the sequence

XQuery expressions

An expression takes a value (a sequence of items) and returns a value.

Expressions may take several forms

- path expressions;
- constructors;
- FLWOR expressions;
- list expressions;
- conditions;
- quantified expressions;
- data types expressions;
- functions.

Simple expressions

Values *are* expressions:

Literals: 'Hello', 47, 4.7, 4.7E+2

Built values: date ('2008-03-15'), true(), false()

Variables: \$x

Built sequences: (1, (2, 3), (), (4, 5)), equiv. to (1, 2, 3, 4, 5), equiv. to 1 to 5.

An XML document is *also* an expression.

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

The result of these expressions is the expression itself!

Retrieving documents and collections

A query takes in general as input one or several sequences of XML documents, called *collections*.

XQuery identifies its input(s) with the following functions:

doc() takes the URI of an XML document and returns a singleton document tree;

collection() takes a URI and returns a sequence.

The result of the *doc()* function is the *root node* of the document tree, and its type is **Document**.

XPath and beyond

Any XPath expression is a query. The following retrieves all the movies titles in the movies collection (for movies published in 2005).

```
collection('movies')/movie[year=2005]/title
```

The result is a sequence of title nodes:

```
<title>A History of Violence</title>
<title>Match Point</title>
```

Remark

The XPath expression is evaluated for each item (document) in the sequence delivered by `collection('movies')`.

Constructors

XQuery allows the construction of new elements, whose content may freely mix literal tags, literal values, and results of XQuery expressions.

```
<titles>
  {collection('movies') //title}
</titles>
```

Expressions can be used at any level of a query, and a constructor may include many expressions.

Remark

An expression e must be surrounded by curly braces {} in order to be recognized and processed.

Constructors

Other element constructors

```
<chapter ref="[{1 to 5, 7, 9}]">
```

same as:

```
<chapter ref="[1 2 3 4 5 7 9]">
```

```
<chapter ref="[1 to 5, 7, 9]">
```

same as

```
<chapter ref="[1 to 5, 7, 9]">
```

The constructor:

```
<paper>{ $myPaper/@id }</paper>
```

will create an element of the form:

```
<paper id="271"></paper>
```

Variables

A **variable** is a name that refers to a value. It can be used in any expression (including identity) in its scope.

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

Variables \$id, \$name, \$job, \$deptno and \$SGMLspecialist must be bound to values.

FLWOR expressions

The most powerful expressions in XQuery. A FLWOR (“flower”) exp.:

- iterates over sequences (**for**);
- defines and binds variables (**let**);
- apply predicates (**where**);
- sort the result (**order**);
- construct a result (**return**).

An example (without **let**):

```
for $m in collection('movies')/movie
where $m/year >= 2005
return
<film>{ $m/title/text() } ,
    (director: {$m/director/last_name/text() } )
</film>
```

FLWOR expressions and XPath

In its simplest form, a FLWR expression provides just an alternative to XPath expressions. For instance:

```
let $year:=1960
for $a in doc('SpiderMan.xml')//actor
where $a/birth_date >= $year
return $a/last_name
```

is equivalent to the XPath expression

```
//actor[birth_date>=1960]/last_name
```

Not all FLWR expressions can be rewritten with XPath.

A complex FLWOR example

"Find the description and average price of each red part that has at least 10 orders" (assume collections *parts.xml* and *orders.xml*):

```
for $p in doc("parts.xml")//part[color = "Red"]
let $o := doc("orders.xml")//order[partno = $p/partno]
where count($o) >= 10
order by count($o) descending
return
<important_red_part>
{ $p/description }
<avg_price> {avg($o/price)} </avg_price>
</important_red_part>
```

for and let

Both clauses bind variables. However:

`for` successively binds each item from the input sequence.

`for $x in /company/employee` binds each employee to
\$x, for each item in the *company* sequence.

`let` binds the whole input sequence.

`let $x := /company/employee` binds \$x to **all** the
employees in *company*.

Note the `for` may range over an heterogeneous sequence:

```
for $a in doc("Spider-Man.xml") //*
where $a/birth_date >= 1960
return $a/last_name
```

Here, \$a is bound in turn to all the elements of the document! (Does it work?
Yes!)

for + return = an expression!

The combination `for` and `return` defines an expression: `for` defines the input sequence, `return` the output sequence.

- A simple loop:

```
for $i in (1 to 10) return $i
```

- Nested loops:

```
for $i in (1 to 10) return
  for $j in (1 to 2) return $i * $j
```

- Syntactic variant:

```
for $i in (1 to 10),
  $j in (1 to 2) return $i * $j
```

- Combination of loops:

```
for $i in (for $j in (1 to 10) return $j * 2)
  return $i * 3
```

Defining variables with `let`

`let` binds a name to a value, i.e., a sequence obtained by any convenient mean, ranging from literals to complex queries:

```
let $m := doc("movies/Spider-Man.xml")/movie
return $m/director/last_name
```

A variable is just a synonym for its value:

```
let $m := doc("movies/Spider-Man.xml")/movie
for $a in $m/actor
return $a/last_name
```

The scope of a variable is that of the FLWR expression where it is defined. Variables cannot be redefined or updated within their scope.

The `where` clause

`where` is quite similar to its SQL synonym. The difference lies in the much more flexible structure of XML documents.

“Find the movies directed by M. Allen”

```
for $m in collection("movies")/movie
where $m/director/last_name="Allen"
return $m/title
```

Looks like a SQL query? Yes but predicates are interpreted according to the XPath rules:

- ① if a path does not exists, the result is `false`, no typing error!
- ② if a path expression returns several nodes: the result is true if there is at least one match.

“Find movies with Kirsten Dunst” (note: many actors in a movie!)

```
for $m in collection("movies")/movie
where $m/actor/last_name="Dunst"
return $m/title
```

The `return` clause

`return` is a mandatory part of a FLWR expression. It is instantiated once for each binding of the variable in the `for` clause.

```
for $m in collection("movies")/movie
let $d := $m/director
where $m/actor/last_name="Dunst"
return
  <div>
    {$m/title/text(), "directed by",
     $d/first_name/text(), $d/last_name/text()},
    with
    <ol>
      {for $a in $m/actor
       return <li>{$a/first_name, $a/last_name,
                    " as ", $a/role}</li>
      }
    </ol>
  </div>
```

Joins

Nested FLWOR expressions makes it easy to express joins on document, à la SQL:

```
for $p in doc("taxpayers.xml")//person
  for $n in doc("neighbors.xml")//neighbor
    where $n/ssn = $p/ssn
    return
<person>
  <ssn> { $p/ssn } </ssn>
    { $n/name }
  <income> { $p/income } </income>
</person>
```

Remark

The join condition can be expressed either as an XPath predicate in the second **for**, or as a **where** clause.

Join and grouping

“Get the list of departments with more than 10 employees, sorted on the average salary”

```
for $d in doc("depts.xml") //deptno
  let $e := doc("emps.xml") //employee[deptno=$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>
```

Operations on lists

XQuery proposes operators to manipulate lists:

- ① concatenation
- ② set operations: (union, intersection, difference)
- ③ Functions (*remove()*, *index-of()*, *count()*, *avg()*, *min()*, *max()*, etc.)

The distinct *values* from a list can be gathered in another list. (This loses identity and order.)

“Give each publisher with their average book price”

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

if-then-else expressions

“Give the holding of published documents”

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

some expressions

some expresses the existential quantifier:

“Get the document that mention sailing and windsurfing activities”

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

every expressions

every expresses the universal quantifier:

“Get the document where each paragraph talks about sailing“

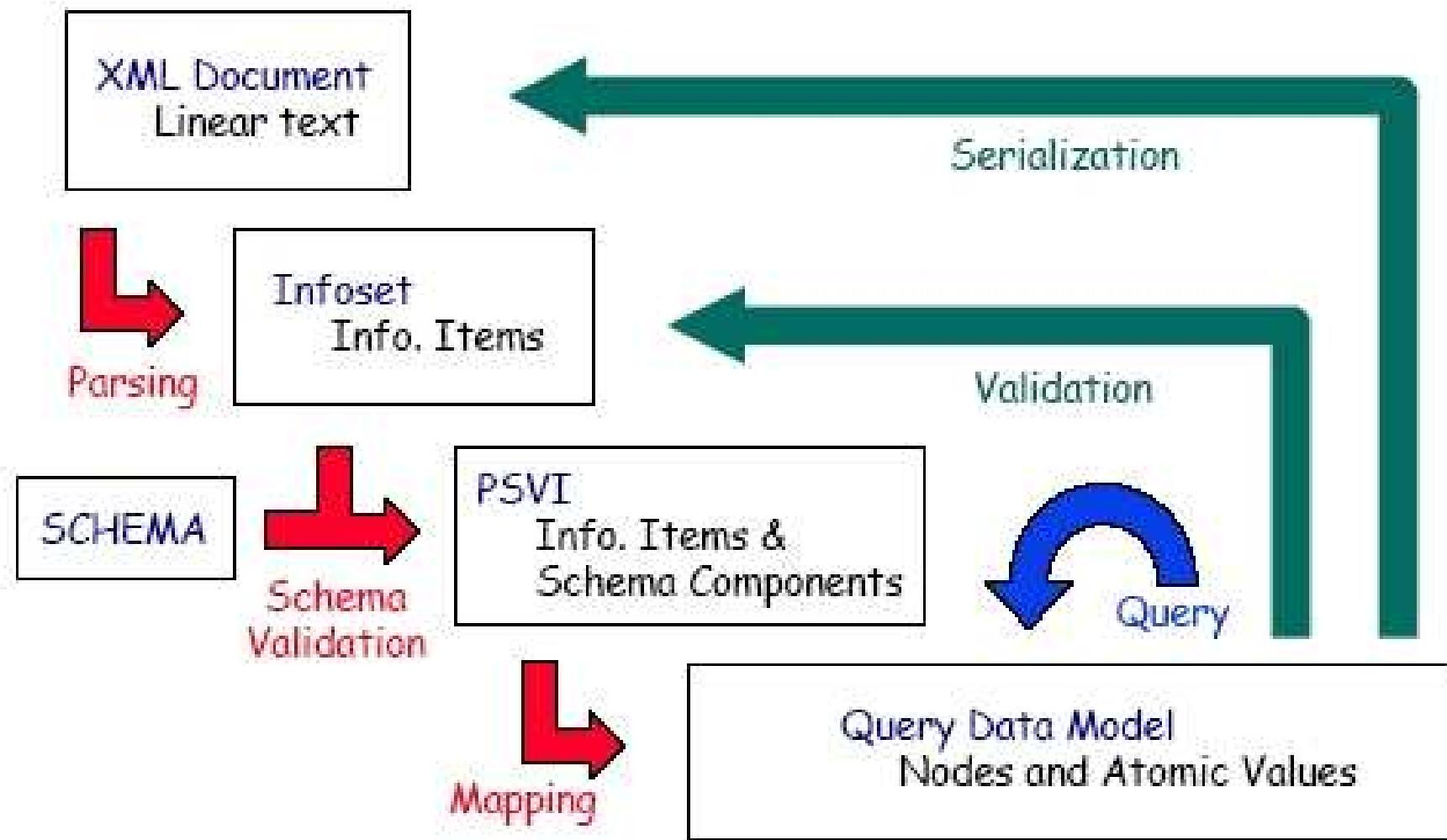
```
for $b in doc("bib.xml")//book
where every $p in $b//paragraph
      satisfies contains($p, "sailing")
return $b/title
```

Function definitions

```
declare namespace my="urn:my";  
  
declare function my:mccarthy91($x as xs:integer)  
  as xs:integer  
{  
  let $result:=  
    if ($x>100) then  
      $x - 10  
    else  
      my:mccarthy91(my:mccarthy91($x + 11))  
  return $result  
}
```

- **Function definitions** similar to other (functional) programming languages
- **Recursion** possible
- **Any** XQuery data type can be used as function argument or return value

XQuery processing model



When XQuery doesn't behave as expected

- ➊ The query does not parse (applet grammar check page) \Rightarrow reformulate it.
You may start from the XQuery use cases.
- ➋ The query parses, but does not work.
- ➌ The query works, but the results are unexpected \Rightarrow figure out what the parser understood.

When XQuery doesn't behave as expected

Sometimes the query parses but will not work (the engine will refuse it).

The parser only checks that the production is well-formed. It does not check that the context provides sufficient information to run the query:

- the functions called in the query are defined
- the variables referred in the query are defined
- the numeric operations are legal etc.

This query parses but it does not work:

```
for $x in doc("bib.xml") //book
return <res1>{$x/title}</res1>,
           <res2>{$x/author}</res2>
```

```
org.exist.xquery.XPathException:
variable $x is not bound.
```

When XQuery doesn't behave as expected

Sometimes the query parses but will not work (the engine will refuse it).

This query parses but it does not work:

```
for $x in doc("bib.xml") //book  
return <res1>{$x/title}</res1>,  
           <res2>{$x/author}</res2>
```

```
org.exist.xquery.XPathException:  
variable $x is not bound.
```

The parser saw this as a sequence formed of:

- a **for-return** expression
- a path expression

You probably meant:

```
for $x in doc("bib.xml") //book  
return (<res1>{$x/title}</res1>,  
           <res2>{$x/author}</res2>)
```

When XQuery doesn't behave as expected

The query gives unexpected results:

Query

```
//book[price<"39.95"]
```

Result

```
<book year="1999">
  <title>The Economics of Technology...</title>
  <editor>
    <last>Gerbarg</last>
    <first>Darcy</first>
    <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
</book>
```

When XQuery doesn't behave as expected

The query gives unexpected results:

Query

```
//book[price<"39.95"]
```

Parsing tree (partial):

```
Predicate
Expr
ComparisonExpr <
PathExpr
StepExpr
AbbrevForwardStep
NodeTest
NameTest
QName price
PathExpr
StringLiteral "35.99"
```

The comparison is done in the string domain.

When XQuery doesn't behave as expected

The query gives unexpected results:

This query has the desired meaning:

Query

```
//book[price<39.95]
```

Parsing tree (partial):

```
Expr
  ComparisonExpr <
    PathExpr
      StepExpr
        AbbrevForwardStep
          NodeTest
            NameTest
              QName price
    PathExpr
      DecimalLiteral 35.99
```

This time, the comparison is done in the numeric domain.

When XQuery doesn't behave as expected

```
for $b in doc("bib.xml") //book return bla
```

```
IXPath2
| QueryList
|   Module
|     MainModule
|     Prolog
|     QueryBody
|       Expr
|         FLWORExpr
|           ForClause
|             VarName
|               QName t
|             PathExpr
|               SlashSlash //
|               StepExpr
|                 AbbrevForwardStep
|                   NodeTest
|                     NameTest
|                       QName title
|             PathExpr
|               StepExpr
|                 AbbrevForwardStep
|                     NodeTest
|                       NameTest
|                         QName bla
```

When XQuery doesn't behave as expected

The query gives unexpected results

```
for $b in doc("bib.xml") //book return bla
```

The last part of the expression is a *path expression testing if the context node is named bla*.

If the context is empty, the query has an empty result.

Maybe you meant:

```
for $b in doc("bib.xml") //book return "bla"
```

More on comparisons

- ➊ Two atomic values:
 - ▶ determine the types of both operands
 - ▶ cast them to a common type
 - ▶ compare the values according to the rules of that type
- ➋ One atomic value and a node:
 - ▶ Cast the node to a string, then proceed as above.
- ➌ Two lists (one list may be of length one):
 - ▶ Compare all list item pairs, return true if the predicate is satisfied at least for one item pair.

Casting is described in the XQuery Functions and Operators document.

Going in depth: W3C specifications

Web documents found under <http://www.w3.org>. *Not* articles! Typically very long *but* navigable. The Introduction clarifies the document role, then go directly to the interesting (sub)sections.

XML specification:

- XML and DTDs
- Namespaces in XML
- XML Schema

XQuery specification:

- XQuery 1.0 specification (syntax)
- XPath functions and operators (`op:equal`, `fn:text`,
`fn:distinct-values`, `fn:document`, `op:gt`, ...)
- XQuery data model

XQuery implementations

Among those that are free and/or open-source:

[Galax](#) : complete, not very efficient

[Saxon](#) : in memory; by Michael Kay, XSL guru

[MonetDB](#) : based on in-memory column-oriented engine; among the fastest

[eXist](#) : very user-friendly interface

[QizX](#) : Xavier Franc. Nice but not great

[BerkeleyDB XML](#) : now belongs to Oracle

SQL/XML: bridging the two worlds

Recent SQL versions (2003) include:

- a native **XML** atomic type, which can be queried in XQuery style
- a set of **XML publishing functions**: extracting XML elements out of relational data by querying
- mapping rules: exporting relational tables in XML

Advantages:

- Unified manipulation of relational and XML data
- Efficient relational query engine well exploited
- Ease of transformation from one format to another

Disadvantage:

- Complexity

SQL/XML: bridging the two worlds

XML publishing functions:

```
select xmlelement(name Customer,
                  xmlattributes(c.city as city),
                  xmlforest(c.CustID,
                            c.Name as CustName))
from customer c
```

Mixed querying:

```
select customer, XMLExtract(order, '/order/@date')
from orders
where XMLExists(order,
                  '/order[//desc/text ()="Shoes"]' )
      =1
```

The precise SQL/XML syntax sometimes depends on the vendor.

Updating XML with XQuery

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

September 29, 2011

Why XQuery Update

XQuery is a *read-only* language: it can *return (compute)* an instance of the XQuery Data Model, but it cannot *modify* an existing instance.

SQL parallel:

```
select... from... where...
```

without

```
insert into table... update table...
```

Applications require reading *and* updating XML data.

XQuery Update Facility: a working draft, not yet a specification

<http://www.w3.org/TR/xquery-update-10/>

Requirements for the XQuery Update language

Expressive power:

- Insert
- Delete
- Update
- Copy with new identity

Extension of XQuery itself:

- Simplifies understanding and learning the language
- Difficulty to introduce side effects...

Well-defined semantics

Conciseness

Amenable to efficient implementation...

XQuery Update concepts

All XQuery expressions can be classified into:

- Updating expressions
- Non-updating expressions

XQuery Update introduces five new kind of expressions:

- **insert, delete, replace, rename**: updating expressions
- **transform**: non-updating expression

XQuery Update specifies:

- how all XQuery expressions are classified into updating and non-updating
- places where each type of expression can appear
- syntax and semantics of each new expression

XQuery Update processing model

The evaluation of an expression produces:

- an instance of the XQuery Data Model and
- a **pending update list**: set of update primitives, i.e. node stat changes that have to be applied.

In the current specification, one of the two has to be empty. This may change in the future.

(The evaluation of a simple XQuery produces an instance of the XQuery Data Model.)

Each update primitive has a target node.

Update primitives are checked for conflicts, and if no conflict appears, they are applied.

Insert expressions

Insert is an updating expression.

General form:

```
insert (constructor!({expr})) (as (first|last))? into  
          (after|before) expr
```

The first expression is called the *source*, and the second the *target*.

The source and target expressions must not be updating.

```
insert <year>2005</year>  
after doc("bib.xml")/books/book[1]/published
```

```
insert $article/author  
as last into doc("bib.xml")/books/book[3]
```

Insert expressions

The pending update list is obtained as follows:

- evaluate the update target (which are the nodes that should get new children)
- for each such node, add to the **pul** the corresponding add-child operation

```
insert {$new-police-report} as last
into doc("insurance.xml")//policies/policy[id=$pid]
/driver[licence=$licence]/accident[date=$dateacc]
/police-reports
```

- locate the proper police-reports element
- for each element in \$new-police-report, add an add-last-child operation to the **pul**

Delete expressions

Delete is an updating expressions. It produces a non-empty pending update list.

General form:

delete expr

```
delete doc("bib.xml")/books/book[1]/author[last()]
```

```
delete /email/message[fn:currentDate() - date >
xdt:dayTimeDuration(P365D)]
```

Replace expressions

Replace is an updating expression. It produces a non-empty pending update list.

General form:

replace expr with expression

```
replace doc("bib.xml")/books/book[1]/publisher  
with doc("bib.xml")/books/book[2]/publisher
```

```
replace value of doc("bib.xml")/books/book[1]/price  
with doc("bib.xml")/books/book[1]/price*1.1
```

Rename expression

Rename is an updating expression.

General form:

```
rename expr to expr
```

```
rename doc("bib.xml")/books/book[1]/author[1]
to main-author
```

```
rename doc("bib.xml")/books/book[1]/author[1]
to $newname
```

Transform expressions (1)

Transform is a **non-updating** expression.

General form:

```
copy $varName := expr (, $varName := expr )*
```

```
modify expr return expr
```

Example: return all managers, omitting their salaries and replacing them with an attribute **xsi:nil**.

Remark

It can be done with XQuery. But it's painful!

Transform returns a *modified copy*, without impacting the original database (it is a non-updating expression).

Transform expressions (2)

Document

```
<employees>
  <employee mgr="true" dept="Toys">
    <name>Smith</name>
    <salary>100000</salary>
  </employee>
  <employee dept="Toys">
    <name>Jones</name>
    <salary>60000</salary>  </employee>
  <employee mgr="true" dept="Shoes">
    <name>Roberts</name>
    <salary>150000</salary>
  </employee>
</employees>
```

Desired result

```
<employee mgr="true" dept="Toys">
  <name>Smith</name>
  <salary xsi:nil="true"/>
</employee>
<employee mgr="true" dept="Shoes">
  <name>Roberts</name>
  <salary xsi:nil="true"/>
</employee>
```

It can be done with XQuery. But it is difficult! Exercise...

Transform expressions (3)

Return all managers, omitting their salaries and replacing them with an attribute **xsi:nil**.

```
for $e in doc("employees.xml")//employee
where $e/@manager = true()
return
  copy $emp := $e
  modify (
    replace value of node $emp/salary with "" ,
    insert nodes (attribute xsi:nil {"true"})
      into $emp/salary
  )
return $em
```

Programming with XQuery Update

Address book synchronization:

- One archive version and two copies
- $c_1 = a$ and $c_2 \neq a \Rightarrow$ propagate c_2 to a and c_1
- $c_1 \neq a, c_2 \neq a \Rightarrow$
 - ▶ If possible, merge differences and propagate them to a , then to c_1, c_2
 - ▶ Otherwise, raise an error.

Agenda entries are of the form:

```
<entry>
    <name>Benjamin</name>
    <contact>benjamin@inria.fr</contact>
</entry>
<entry>
    <name>Anthony</name>
    <contact>tony@uni-toulon.fr</contact>
</entry>
```

Programming with XQuery Update

```
for $a in doc("archive.xml")//entry,
  $v1 in doc("copy1.xml")/version/entry,
  $v2 in doc("copy2.xml")/version/entry
where $a/name=$v1/name and $v1/name=$v2/name
return
  if ($a/contact=$v1/contact and $v1/contact=$v2/contact)
  then ()
  else
    if ($v1/contact=$v2/contact) then
      replace value of node $a/contact with $v1/contact
    else
      if ($a/contact=$v1/contact)
      then (
        replace value of $a/contact
          with $v2/contact,
        replace value of $v1/contact
          with $v2/contact ...
```

Programming with XQueryUpdate

```
...
if ($a/contact = $v1/contact)
then ...
else
  if ($a/contact = $v2/contact)
  then ( replace value of $a/contact
          with $v1/contact,
          replace value of $v2/contact
          with $v1/contact )
  else ( insert node
          <fail> <arch>{$a}</arch>
                  <v1>{$v1}</v1> <v2>{$v2}</v2>
          </fail>
          into doc("log.xml")/log ),
replace value of node doc("archive.xml")
 /*/last-synch-time with current-dateTime()
```

XQuery - SQL comparison

Function

Query (read-only)
Update
Full-text
Scripting

Relational

SQL select
SQL update
SQL MMS
PL/SQL

XML

XQuery
XQuery Update
XQuery Full-Text
XQuery Scripting Extension

XQuery update is not a programming language. Missing:

- Control over the *scope of snapshots*, i.e. **when do my updates become visible to another query?** XQuery Update: after the current query has finished executing.
- Control over atomicity, i.e. **which expressions must be executed atomically?**
- The possibility to **both return a result and have side effects**. XQuery Update: one or the other is empty.
- Error handling.

An XQuery scripting language: XQuery-P

D.Chamberlin, M.Carey, D.Florescu, D.Kossmann: "Programming with XQuery",
XIME-P 2006

- ① Define a **sequential execution mode**: the statements must be evaluated in order, and each statement sees the side effect of the previous one
- ② Define **blocks**, which are units of code to be executed sequentially. **New variables** can be defined inside a block. The returned result is that of the last expression.
- ③ Introduce **assignments** to bind variables to new values.

```
for $item in /catalog/item[price < 100]
return
{replace value of $item/price with $item/price * 1.1;
$item}
```

An XQuery scripting language: XQuery-P

Forces to define **evaluation order** on an XQuery expression:

- **for, let, where, order by** executed in the order of their appearance; then, **return**
- **if** evaluated first, then evaluate **then** or **else**
- **,:** evaluate from left to right, apply all the updates after each item
- **function call**: evaluate the arguments before the body

Specifying evaluation order is a **big** departure from traditional query language style. (Which of **select**, **from** and **where** is evaluated first?)

Programming with XQuery-P

```
declare updating function local:transfer
  ($from-acctno as xs:string, $to-acctno as xs:string,
   $amount as xs:decimal) as xs:integer
{ declare $from-acct as element(account)
  := /bank/account[acctno eq $from-acctno],
      $to-acct as element(account)
  := /bank/account[acctno eq $to-acctno];
  if ($from-acct/balance > $amount)
  then atomic {
    do replace value of $from-acct/balance
       with $from-acct/balance - $amount;
    do replace value of $to-acct/balance
       with $to-acct/balance + $amount;
  } (: end of atomic region :)
  else -1
};
```

Implementations

XQuery Update:

- eXist
- MonetDB

XQuery-P and similar proposals: preliminary prototypes

Table des matières

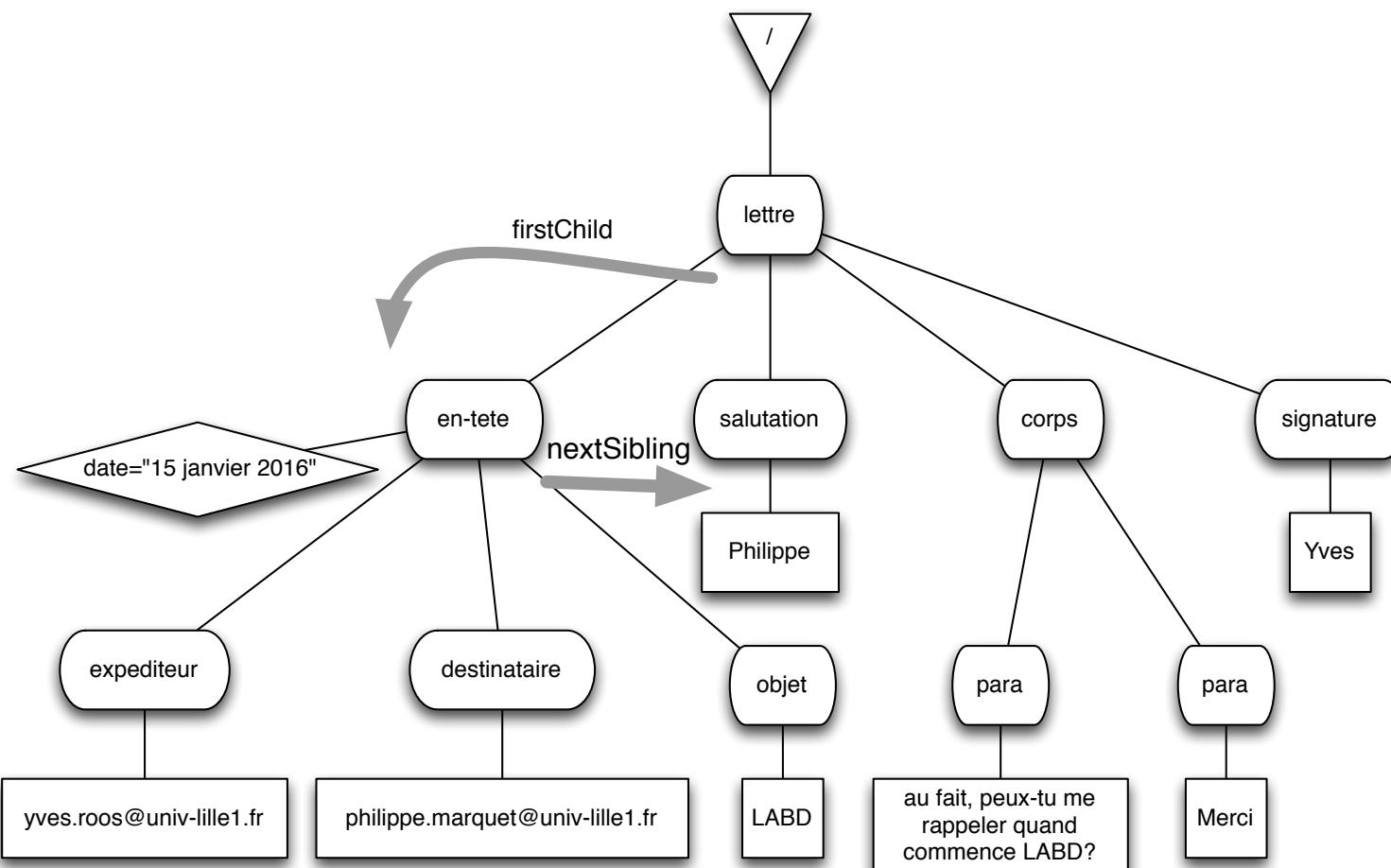
- 1 Introduction
- 2 Typage de données XML
- 3 Navigation avec XPath
- 4 Transformations XSLT
- 5 Programmation avec XQuery
- 6 APIs pour la lecture des fichiers XML

Lecture de fichier XML : APIs java

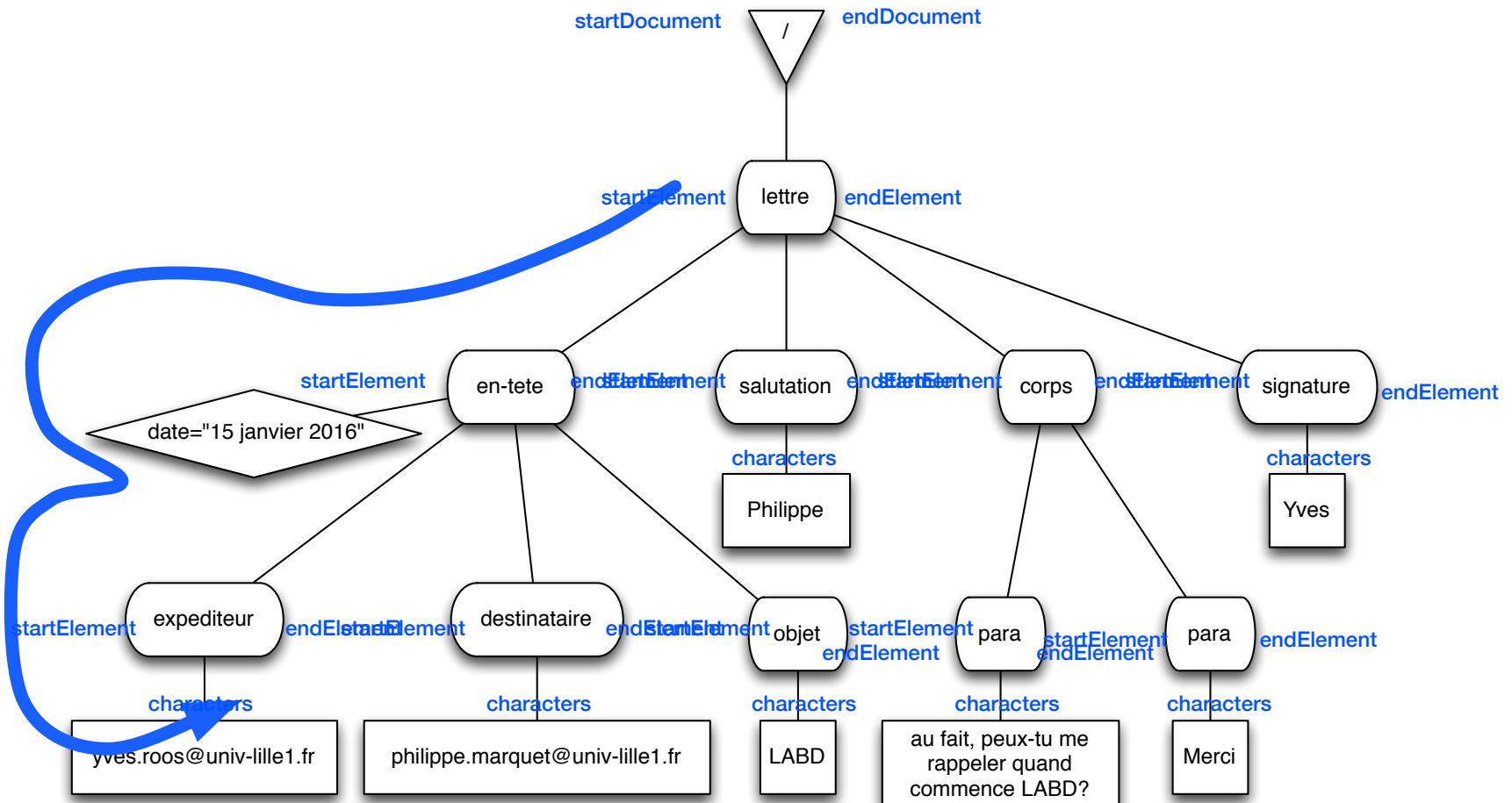
- DOM

- SAX

Document Object Model



Simple Api for XML



Simple API for XML

-
[org.w3c.dom.bootstrap](#)
[org.w3c.dom.events](#)
[org.w3c.dom.ls](#)
[org.xml.sax](#)
[org.xml.sax.ext](#)
[org.xml.sax.helpers](#)


[org.xml.sax](#)
Interfaces
[AttributeList](#)
[Attributes](#)

[ContentHandler](#)
[DocumentHandler](#)
[DTDHandler](#)
[EntityResolver](#)
[ErrorHandler](#)
[Locator](#)
[Parser](#)
[XMLFilter](#)
[XMLReader](#)

Classes
[HandlerBase](#)
[InputSource](#)

Exceptions
[SAXException](#)
[SAXNotRecognizedException](#)
[SAXNotSupportedException](#)
[SAXParseException](#)

Method Summary

void	characters(char[] ch, int start, int length) Receive notification of character data.
void	endDocument() Receive notification of the end of a document.
void	endElement(String uri, String localName, String qName) Receive notification of the end of an element.
void	endPrefixMapping(String prefix) End the scope of a prefix-URI mapping.
void	ignorableWhitespace(char[] ch, int start, int length) Receive notification of ignorable whitespace in element content.
void	processingInstruction(String target, String data) Receive notification of a processing instruction.
void	setDocumentLocator(Locator locator) Receive an object for locating the origin of SAX document events.
void	skippedEntity(String name) Receive notification of a skipped entity.
void	startDocument() Receive notification of the beginning of a document.
void	startElement(String uri, String localName, String qName, Attributes atts) Receive notification of the beginning of an element.
void	startPrefixMapping(String prefix, String uri) Begin the scope of a prefix-URI Namespace mapping.

Simple Api for XML

```
package labd ;

import org.xml.sax.*;
import org.xml.sax.helpers.* ;
import java.io.IOException;

/**
 * @author yves.roos
 *
 * Exemple d'implementation d'un ContentHandler.
 */
public class LABDHandler extends DefaultHandler {

    /**
     * Evenement envoye au demarrage du parse du flux xml.
     * @throws SAXException en cas de probleme quelconque ne permettant pas de
     * se lancer dans l'analyse du document.
     * @see org.xml.sax.ContentHandler#startDocument()
     */
    public void startDocument() throws SAXException {
        System.out.println("Debut du document");
    }
}
```

Simple Api for XML

```
/**  
 * Evenement reçu a chaque fois que l'analyseur rencontre une balise xml ouvrante.  
 * @param nameSpaceURI l'url de l'espace de nommage.  
 * @param localName le nom local de la balise.  
 * @param rawName nom de la balise en version 1.0 <code>nameSpaceURI + ":" + localName</code>  
 * @throws SAXException si la balise ne correspond pas a ce qui est attendu,  
 * comme par exemple non respect d'une dtd.  
 */  
public void startElement(String nameSpaceURI, String localName, String rawName, Attributes attributs)  
throws SAXException {  
    System.out.println("Ouverture de la balise : " + localName) ;  
    if (attributs.getLength() != 0) System.out.println(" Attributs de la balise : ") ;  
    for (int index = 0; index < attributs.getLength(); index++) { // on parcourt la liste des attributs  
        System.out.println("      - " + attributs.getLocalName(index) + " = " + attributs.getValue(index));  
    }  
}
```

Simple Api for XML

```
public static void main(String[] args) {
    try {
        XMLReader saxReader = XMLReaderFactory.createXMLReader();
        saxReader.setContentHandler(new LABDHandler());
        saxReader.parse(args[0]);
    } catch (Exception t) {
        t.printStackTrace();
    }
}
```

Simple Api for XML : exemple

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- exemple de fichier xml -->
<lettre>
    <en-tete date="15 janvier 2016">
        <expediteur>
            yves.roos@univ-lille1.fr
        </expediteur>
        <destinataire>
            philippe.marquet@univ-lille1.fr
        </destinataire>
        <objet>BDA</objet>
    </en-tete>
    <salutation>Philippe</salutation>
    <corps>
        <para>
            au fait, peux-tu me rappeler quand commence LABD ?
        </para>
        <para>Merci.</para>
    </corps>
    <signature>Yves</signature>
</lettre>
```

Simple Api for XML : exemple

```
[macbook-pro-de-yves-roos-/Users/yroos/Documents/fil/labd/labd-1] java labd.LABDHandler courrier.xml
Debut du document
Ouverture de la balise : lettre
Ouverture de la balise : en-tete
    Attributs de la balise :
        - date = 15 janvier 2016
Ouverture de la balise : expediteur
    Contenu : |yves.roos@univ-lille1.fr|
Fermeture de la balise : expediteur
Ouverture de la balise : destinataire
    Contenu : |philippe.marquet@univ-lille1.fr|
Fermeture de la balise : destinataire
Ouverture de la balise : objet
    Contenu : |LABD|
Fermeture de la balise : objet
Fermeture de la balise : en-tete
Ouverture de la balise : salutation
    Contenu : |Philippe|
Fermeture de la balise : salutation
Ouverture de la balise : corps
Ouverture de la balise : para
    Contenu : |au fait, peux-tu me rappeler quand commence LABD ?|
Fermeture de la balise : para
Ouverture de la balise : para
    Contenu : |Merci.|
Fermeture de la balise : para
Fermeture de la balise : corps
Ouverture de la balise : signature
    Contenu : |Yves|
Fermeture de la balise : signature
Fermeture de la balise : lettre
Fin du document
[macbook-pro-de-yves-roos-/Users/yroos/Documents/fil/labd/labd-1]
```

Simple Api for XML : exercice

```
...
public class InterroHandler extends DefaultHandler {
    private int n ;
    private boolean fermante ;

    public void startDocument() {this.n = 0 ; this.fermante = false ;}

    public void endDocument() {System.out.println() ;}

    public void startElement(String nameSpaceURI, ...){
        if (this.fermante) System.out.print(" , ");
        System.out.print(localName + "-" + this.n + "( ");
        this.n++ ; this.fermante = false ;
    }

    public void endElement(String nameSpaceURI, ...){
        System.out.print(" )");
        this.n++ ; this.fermante = true ;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<a><b/><c><d/><e><f/></e></c><g/></a>
```