6 XML et les Données Semi-structurées

– L’apparition de XML (*eXtensible Markup Language*) (plus “évolué” que HTML) a mené au nouveau concept de *données semi-structurées*.
– XML : standard W3C d’échange de données sur le Web. Permet un échange sur un format standard, independemment des formats de stockage de ces données.
– Grande flexibilité.
– Multitude de standards associés :
  - Formats de Schémas : DTD et XML-schéma
  - Langages de Requête : XPATH, XQUERY (extension de XPATH),...
  - XSLT : notation pour transformer un document XML d’un format à un autre.
  etc.
– Lien avec le cours sur XML de la MIAGE ?

Parmi les applications BD :

– Intégration de Données (“mediateur”).
– Bases de Données en Biologie.

6.1 HTML

HTML (*Hyper Text Markup Language*) : un standard d'écriture de documents pour le Web.

HTML est un langage à **balises** ("étiquettes"). Ces balises sont **fixes**, à fonctions prédéfinies.

Les balises de HTML permettent de :

– **Mettre en forme un texte**
  Ex. `<B> </B>, <I> </I>, <CENTER> </CENTER>`,....
– **Créer des liens** (balises "amarres").
  Ex :
  `<A HREF="http://www.univ-evry.fr/">Université d'Evry Val d'Essonne </A>`

Le rôle des balises autres que les “amarres” est celui de *présenter visuellement* du *texte* en un certain format.

Par exemple :

– `<B> bla </B>` sert à écrire **bla** à la place de bla
– `<I> bla </I>` sert à écrire *bla* à la place de bla
– `<CENTER> bla </CENTER>` sert à écrire

  bla

  à la place de :

  bla
– `<H1> bla </H1>`, `<H2> blabla </H2>` `<H3> blablabla </H3>` servent à introduire des titres (des “sections”), par ordre d’importance décroissant.

6.2 Limites de HTML

HTML n’est pas adapté à l’interrogation des données.

Il permet de mettre en FORME un texte.

Il ne permet pas de STRUCTURER “logiquement” un contenu.

Exemple

Une organisation publies des données stockées dans une BD relationnelle. Des pages web sont crées.

Une autre organisation veut une analyse de ces données ; son logiciel a acces seulement aux pages HTML.

– Une petite modification du format d'un élément d'une page web peut casser ce logiciel !
– Même si on a besoin seulement de la valeur moyenne d'une colonne d'une table, on peut avoir besoin de charger une base entière via plusieurs requêtes de pages HTML.

7 XML

XML : nouveau standard adopté par le World Wide Web Consortium (W3C) comme complément de HTML permettant un échange aisé de données de sur le web.

– Le but principal de XML n'est pas de décrire un format de texte, mais de **structurer** logiquement un contenu.
– Les balises ont le rôle de classer des données selon une hiérarchie définie par l'auteur du document XML.

– Avec XML, la mise en forme textuelle est effectué dans une *feuille de style*, un document separé qui associe des formes de présentations (texte en gras, en italique, centré, etc.) aux balises. Des feuilles différentes permettent des formattages différents du même document.
– Des outils permettent de convertir un document XML en HTML, afin de pouvoir afficher une page web.

Example 1 Un petit document XML

```
<?xml version=’’1.0 encoding=’’iso-8859-1’’?>

<communication prior=’’important’’>
<pour> Virginie </pour>
<sujet> Rappel </sujet>
<message> N’oublie pas de lire l’article
<lire> Lutz et al. 2002. </lire>
Il faut bien comprendre
<reflechir> la preuve de terminaison.  </reflechir>
Rendez-vous <date> mercredi </date> <lieu> dans mon bureau </lieu>
</message>
<signature> Serena </signature>
</communication>
```

Suite de l’exemple

Résultat de la mise en forme grâce à une feuille de style :

Priorité : important
Pour : Virginie
Sujet : Rappel

N’oublie pas de lire l’article *Lutz et al. 2002.* Il
faut bien comprendre **la preuve de terminaison.** Rendez-vous **mercredi** *dans mon bureau*

Serena

Suite de l’exemple

Résultat de la mise en forme avec une **autre** feuille de style :

Priorité : IMPORTANT
Pour : VIRGINIE

Sujet : RAPPEL

N’oublie pas de lire l’article *Lutz et al. 2002.*

Il faut bien comprendre *la preuve de terminaison.*

Rendez-vous **mercredi dans mon bureau**

Serena

XML modèlise des informations :

– En organisant les données en un *graphe* d'*objets complexes*
– En les structurant de façon plus *flexible* par rapport au au modèle relationnel ou objet : les données sont dites *semistructurées*

graphe, *objet complexe*, *flexible*, *semistructuré* = ? ? ? ?

$\Rightarrow$ Voir la suite...

7.1 Syntaxe de base de XML

La composante essentielle est l'*élément*, un morceau de document delimité par une balise d'ouverture (ex. `<toto>`) et une de fermeture (ex. `</toto>`.

Un élement peut contenir du texte, des autres élements (→ "objet complexe"), ou un mélange des deux.

– Les balises (leur noms) **sont définies par les utilisateurs**.
– Elles n'ont pas de signification prédéfinie : elles indiquent seulement comment structurer le document sous forme de arbre (ou, plus généralement, de graphe).

Example 2

```
<personne>
<nom> Alan  </nom>
<age> 42 </age>
<email> agb@abc.com </email>
</personne>
```

Ici on a :

– *Un élément complexe de “sorte” (“type”)* `personne`*, qui consiste d’un triplet d’éléments ayant les “sortes”* `nom`,`age`,`email`.
– *Un élément* `Alan` *de “sorte”* `nom`
– *Un élément* `42` *de “sorte”* `age`
– *Un élément* `agb@abc.com` *de “sorte”* `email`

Suite de l’exemple

Le contenu de ce document peut être représenté :

– Soit par un arbre où les *noeuds internes* sont etiquéttés par les baliseses.

– Soit par un arbre où les *arcs* sont etiquéttés par les balises.

FIGURES AU TABLEAU

Example 3

```
<gens>
<personne>
<nom> Alan  </nom>
<age> 42 </age>
<email> agb@abc.com </email>
</personne>
<personne>
<nom> Patricia  </nom>
<age> 36 </age>
<email> ptn@abc.com </email>
</personne>
</gens>
```

Remarque : on peut utiliser plusieurs éléments ayant la même balise pour représenter une collection.

Dans l’exemple 3, une entité de “sorte” `gens` est une collection de personnes...

A nouveau, on peut représenter ces informations sous forme d’un arbre, avec 2 possibilités.

Example 4

```
<biblio>
<livre>
<auteurs>
<nom> Abiteboul </nom>
<nom> Bunemann </nom>
<nom> Suciu </nom>
</auteurs>
<titre> Data on the Web </titre>
<edition>
<nom-ed> Morgan Kaufman </nom-ed>
<adresse-edition>
<rue-ed> 340, Pine Street </rue-ed>
<ville-ed> San Francisco </ville-ed>
<etat-ed> California </etat-ed>
<pays-ed> USA </pays-ed>
</adresse-edition>
</edition>
</livre>
<livre>
.
.
.
</livre>
</biblio>
```

7.2 Pourquoi “objet complex” ?

Comparer avec les BD relationnelles (en première forme normale), où le domaine de tout attribut contient seulement des valeurs atomiques, et toute “entité est plate” :

nom	titre	nom-ed	rue-ed	ville-ed	etat-ed	pays-ed
Abit	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
Bune	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
Suciu	Data on the Web	Kauf	340, Pine St.	S. Fr.	Cal	USA
⋮						

Dans le document XML de l’exemple 4, un livre est un objet **complex**, composé d’une séquence d’auteurs, d’un titre et d’une adresse (comme dans le BD à objet). La première et la troisième composantes sont elles mêmes des objets complexes.

7.3 Pourquoi “semistructuré” ?

Un premier élément de réponse :
un objet complexe peut avoir des composantes optionnelles, le “schéma” de la base n’est pas rigide.

Différence par rapport au modèle relationnel, où le nombre d’attributs du schéma d’une relation est fixé en avance.

Example 5 *Un livre peut* **éventuellement** *être stocké avec son prix ; le champ* `pays-ed` *est aussi optionnel :*

```
<biblio>
<livre>
<auteurs>
<nom> Abiteboul </nom>
<nom> Bunemann </nom>
<nom> Suciu </nom>
</auteurs>
<titre> Data on the Web </titre>
<edition>
<nom-ed> Morgan Kaufman </nom-ed>
<adresse-edition>
<rue-ed> 340, Pine Street </rue-ed>
<ville-ed> San Francisco </ville-ed>
<etat-ed> California </etat-ed>
<pays-ed> USA </pays-ed>
</adresse-edition>
</edition>
<prix-en-euros> 44 <prix-en-euros>
</livre>
<livre>
<auteurs>
<nom> Gardarin </nom>
</auteurs>
<titre> Internet/Intranet et Bases de Donn\’ees </titre>
<edition>
<nom-ed> Eyrolles </nom-ed>
<adresse-edition>
<rue-ed> 61, Bld Saint Germain </rue-ed>
<ville-ed> Paris </ville-ed>
```

```
<etat-ed> France </etat-ed>
</adresse-edition>
</edition>
</livre>
</biblio>
```

Dans les exemples vus jusqu'à ici, les données sont organisées en arbres. Les réferences produisent des **graphes**.

On peut faire référence à un sommet déjà existant dans le graphe car on peut associer à un *idéntificateur* à chaque élément.

Pour "pointer" vers un élément ayant identificateur, disons `cle` (nom choisi par l'auteur), on exploite l'existence des *attributs* XML.

En général, un attribut XML sert à dèfinir une **proprieté des données** ; sa valeur est une chaîne de caractéres. (Dans l'exemple 1, `prior` était un attribut).

La syntaxe générale de la déclaration d'attributs est :

```
<balise attribut1=valeur1 ... attributN = valeurN> ...
</balise>
```

Par ex. :

```
<nom langue=français> Abiteboul </nom>
```

```
<nom langue=anglais> Bunemann </nom>
```

Pour identifier un élément il suffit d'utiliser un attribut dont le type déclaré (où ?) est ID :

`<balise attribut=valeur> </balise>` marche, à condition que `attribut` soit de type ID.

La syntaxe abrégée `<balise attribut=valeur/>`

est aussi possible.

Par ex. :

```
<Livre ISBN="isbn-95456255"/>
```

C'est dans le schéma du document XML (voir après) que l'on on déclare l'attribut ISBN comme ayant le type ID.

Pour faire **référence** à un élément on utilise un attribut de type IDREF :

```
<balise attributRef= identificateur> </balise>
```

La syntaxe abrégée

```
<balise attributRef= identificateur /balise>
```

est aussi possible.

Par ex :

```
<Livre NOM=''Tout sur Linux'' EditeurRef= "LFE" />
```

(on pointe à l'élément identifié par `LFE` et décrivant l'édieur Linux French Edition).

Ici, on déclarera l'attribut `EditeurRef` comme ayant le type IDREF.

Un élément de la forme :

```
<balise attributRef= identificateur /balise>
```

où `attributRef` est de type IDREF n'a pas de contenu. Il est dit dit *élément vide*.

Example 6

```
<geographie-USA>
</etats>
<etat>
<etat cle = ’’e1’’>
<code-etat> IDA </code-etat>
<nom-etat> Idaho </nom-etat>
<capitale ref-cap = ’’v1’ />
<villes-dans  ref-a-villes = ’’v1’’ />
<villes-dans  ref-a-villes = ’’v3’’ />
...
</etat>
<etat>
...
</etat>
</etats>
<villes>
<ville>
<ville iden = ’’v1’’>
<code-ville> BOI </code-ville>
<nom-ville> Boise </nom-ville>
<etat-de-la-ville ref-a-etat = ’’e1’’ />
</ville>
<ville>
<ville iden = ’’v2’’>
<code-ville> CCN </code-ville>
<nom-ville> Carson City </nom-ville>
<etat-de-la-ville ref-a-etat = ’’e2’’ />
</ville>
<ville>
<ville iden = ’’v3’’>
<code-ville> MO </code-ville>
```

```
<nom-ville> Moscow </nom-ville>
<etat-de-la-ville ref = ’’e1’’ />
</ville>
...
</villes>
</geographie-USA>
```

La possibilité de faire des références fait passer de la structure de *arbre* à celle plus générale de *graphe* orienté avec une racine.

Figure au tableau.

XML permet de mélanger des données textuelles et des sous-éléments au sein d’un élément :

Example 7 `<personne>`

```
Voici mon meilleur ami
<nom> Alan  </nom>
<age> 42 </age>
Je ne suis pas sure de l’adresse e-mail suivante˜:
<email> agb@abc.com </email>
</personne>
```

Pas naturel du point de vue BD, mais du à l’origine de XML comme langage de documents hyper-texte.

Dans la suite, on aura pas de mélange, et les données (texte) seront toujours aux feuilles de l’arbre qu’on a si on ignore les réf.

7.4 Schémas pour des documents XML

Deux formats :

1. Un *DTD* (**D***ocument* **T***ype* **D***efinition*)
2. Un *XML-schema*, qui a une structure de typage plus riche.

7.4.1 Les DTD

Un DTD peut être vu comme une sorte de schéma pour les données XML. Il est **optionnel** $\Rightarrow$ données **semistructurées**.

Une document XML qui, en outre d'être syntaxiquement correct, a un DTD, et le respecte, et dit *valide*.

Syntaxe des DTD

Structure d'un DTD :

```
<? xml version=''1.0''?>
<?DOCTYPE nom [Declarations-de-Type]>
```

La première ligne, optionnelle, indique la version de XML utilisée, la deuxième contient le DTD proprement dit.

La balise `nom` est la balise racine. `Declarations-de-Type` est une suite $Déclaration_1$,...,$Déclaration_n$
où toute déclaration introduit le nom d'un élément et sa "sorte", c.à.d une déscription de la "forme de son contenu", ou bien introduit les attributs d'un élément donné, et leur types.

Syntaxe d'UNE déclaration de la suite de déclarations du DTD

Pour le moment, ignorons les déclarations pour les attributs.

Chaque déclaration d'élément est constituée du symbole <, puis de la chaîne de caractères `!ELEMENT`, puis d'une balise, puis d'un modèle de contenu, et, enfin, le délimiteur de fin > :

< `!ELEMENT` balise modèle_contenu>

Il y a cinq sortes différents de modèles de contenu.

Modèles de contenu dans une déclaration d'un DTD

1. Contenu vide : `<!ELEMENT balise EMPTY >`
2. Pas de contraintes sur le contenu : `<!ELEMENT` balise `ANY>`.
   (NB : données "semistructurées" !)
3. Elément ne contenant que des données textuelles :
   `<!ELEMENT balise #PCDATA>`
4. Elément ne contenant que d'autre éléments : `<!ELEMENT` balise motif `>`
   où `motif` est une expression regulière sur l'alphabet des noms des balises.
5. Eléments de contenu "mixte" : mélange à la fois d'éléments et de données textuelles.

Les opérateurs des expressions regulières utilisés dans un DTD

– Le symbole , indique la concaténation.
  Exemple : `chat,chien` signifie qu'un chien doit suivre un chat. L'ordre compte dans les documents XML (arbres ordonnés). (**pourquoi ? ?**)
– Le symbole | est le XOR logique.
  Exemple : `chat` | `tortue` | `chien` signifie que soit un chat soit une tortue soit un chien est acceptable (mais un seul de ces animaux).
– Le symbole ? rend l'expression immédiatement précédente optionnelle.
  Exemple : (`chat,chien`) ? signifie que une suite d'un chat puis d'un chien peut être placée à cet endroit, ou omise.

Les opérateurs des expression regulières utilisés dans un DTD, suite

– Le symbole $+$ signifie qu'une suite non vide d'éléments conformes à l'expression immédiatement précédente est réquise.
Exemple : $(\texttt{chat} \mid \texttt{chien})^{+}$ signifie qu'il doit y avoir un nombre non nul de chats et de chiens.
– Le symbole $*$ signifie qu'une suite éventuellement vide d'éléments conformes à l'expression immédiatement précédente est réquise.
Exemple : $(\texttt{chat},\texttt{chien})*$ signifie que, à cet endroit, ou bien il n'y a rien du tout, ou alors il y a une suite de chats et chiens telle que tout chat est immédiatement suivi par un chien et tout chien est immédiatement précédé par un chat.

En effet, un DTD est une grammaire, qui spécifie un arbre.

C'est une grammaire *context free* élargie en permettant des expressions regulières dans la droite des règles de production.

Les opérateurs regulières sont ceux habituels, car :

ϵ = EMPTY
ab = a,b
a+b = a | b
a*=a*
aa*=a+
ϵ + a = a ?

Un document D valide par rapport à un DTD S est arbre qui appartient au langage d'arbres accepté par l'"automate d'arbres associés à la grammaire d'arbres S.

Example 8

```
<!ELEMENT article
(titre, sous-titre?, auteur*,
(paragraphe|table|figures)+, bibliographie?)>
```

Cette déclaration décrit le contenu d'un article comme étant composé d'un titre suivi éventuellement d'un sous-titre, puis de 0 ou plusieurs auteurs, puis d'une combinaison (non-vide) de paragraphes, tables et figures, puis, éventuellement, d'une bibliographie.

Example 9 *Un exemple simple de DDT.*

```
<!DOCTYPE gens [
<!ELEMENT gens (personne)*>
<!ELEMENT personne (nom, age, e-mail)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
]>
```

Le document de l’exemple 3 (p. 104) est valide par rapport à ce DTD.

En résumant, il y a plusieurs raisons pour lesquelles on peut qualifier des données représentées dans un document XML comme étant *semi-structurées* :

1. Le document n'a pas de schéma (DTD ou XML-schéma). Dans ce cas, on a juste du texte, que l'on ne sait pas comme interroger ! (Sauf par recherche de mot clé, comme pour les documents HTML)
2. Le document a un schéma. Mais :
   (a) Une déclaration de la forme `<!ELEMENT balise ANY>` ne donne aucune information sur la structure.
   (b) Une déclaration comportant ? prévoit l'optionalité d'une balise B dans le motif associé à une balise A. (Mais : penser aux valeurs nulles dans les SGBD relationnels...)

A la place d’inclure le DTD dans le document, on peut aussi le sauver dans un fichier separé, qui peut être placé à une URL différente. Ceci permet à différents sites web de partager un unique schéma.

Déclaration d'attributs dans un DTD

Un DTD permet aussi de déclarer des attributs, et leur type.

`ID` est le type des attributs permettant de donner des idéntificateurs aux éléments.
(Voir l'exemple 6, p. 115, dans le DTD correspondant : `cle` est de type `ID`).

`IDREF` (ou IDREFS) est le type des attributs de référence.
Si un attribut A est déclaré comme ayant le type IDREF, ceci indique que la valeur de A est un idéntificateur d'un élément (l'élément pointé").
(Voir l'exemple 6, p. 115 : `ref-cap` est de type `IDREF`, et `ref-à-villes` est de type `IDREFS`).

Déclaration d'attributs dans un DTD, suite

Le mot-clé ATTLIST est utilisé pour déclarer une liste d'attributs (pouvant contenir un seul attribut).

En outre de déclarer le type des attributs, on décrit aussi leur "comportement" ; le mots-clés `#REQUIRED, #IMPLIED` indiquent, respectivement, si un attribut est obligatoire ou optionnel.

Syntaxe d'une déclaration d'une liste d'attributs de l'élément ayant balise B :

`<!ATTLIST` $B\ nom_{att1}\ type_{att1}\ descr_{att1}, ..., nom_{attN}\ type_{attN}\ descr_{attN}$`>`

Example 10 *Un DDT pour les données de l'exemple 6 :*

```
<!DOCTYPE  geographie-USA [
<!ELEMENT geographie-USA (etats|villes)*>
<!ELEMENT etats (etat)*>
<!ELEMENT etat (code-etat,nom-etat,capitale,villes-dans*)>
<!ATTLIST etat cle ID #REQUIRED>
<!ELEMENT code-etat (#PCDATA)>
<!ELEMENT nom-etat (#PCDATA)>
<!ELEMENT capitale EMPTY>
<!ATTLIST capitale ref-cap IDREF #REQUIRED>
<!ELEMENT villes-dans EMPTY>
<!ATTLIST villes-dans ref-a-villes IDREFS #REQUIRED>
<!ELEMENT villes (ville)*>
<!ELEMENT ville (code-ville,nom-ville,etat-de-la-ville)>
<!ATTLIST ville iden ID #REQUIRED>
<!ELEMENT code-ville (#PCDATA)>
<!ELEMENT nom-ville (#PCDATA)>
<!ELEMENT etat-de-la-ville EMPTY>
<!ATTLIST etat-de-la-ville ref-a-etat IDREF #REQUIRED>
]>
```

Un défaut des DTD : on ne peut pas déclarer que les sortes des valeurs de l'attribut `ref-cap`, par exemple, sont des villes (et pas des états, par exemple). Les DTD n'offrent pas un typage adéquat des données semistructurées.

Pourquoi ce défaut ?

7.4.2 XML Schema : C'est quoi ?

Ce sujet est traité dans un autre cours de la MIAGE, et ici on se limite à le mentionner.

Un fichier XML-Schema est lui-même un document XML ! Mais ce document fixe le format d'autres documents XML.

XML a un système de types riches, et ces types sont ceux utilisés par XQuery aussi.

Types Simples de XML Schema : Il en a beaucoup. Par ex., STRING, BOOl, INTEGER, POSITIVEINTEGER, CDATA, DATE, ID, IDREF, IDREFS, NMTOKEN (une lettre, une chiffre, un point, un tiré, une virgule..)

Types Complexes : On peut construire des types complexe à l'aide de constructeurs. Par exemple, grâce au constructeur `<xsd :sequence>`, on peut déclarer qu'un élément `personne` a un type complexe qui est défini comme étant une *séquence* d'un `nom`, de type `string`, d'un `prénom`, de type `string`, d'une `date_de_naissance`, de type `date`, d'une `adresse`, de type `string`, et, enfin, d'un `e-mail`, de type string.

8 Un langage de requête pour les données XML : XQUERY

XQUERY : langage d'interrogation de documents (bases de données) XML.

Ces diapositives sont fortement inspirées par le document :

Katz, Xquery : A guided Tour

disponible en ligne avec les supports de cours.

Historique

– 1998 : W3C organise un workshop sur XML Query
– 1999 : W3C lance le *XML Query Working Group* (39 membres, 25 companies)
– 2000 : publication des objectifs, des cases d'utilisation et du modèle de données.
– 2001 : draft de la spécification du langage
– 2002 : mises à jour périodiques de cette draft
– 2003 : redaction complete des objectifs et des cases d'utilisation
– Version finale de XQuery Version 1.

– XQUERY est déclaratif : toute expression XQUERY produit une *valeur* ; c'est un langage de style *fonctionnel* (comme CAML).
– les expressions le plus banales utilisent des *constantes* (et des opérateurs simples) :
  3+4
  est un programme XQUERY qui s'évalue à l'entier 7.
– On a aussi des *variables* : $x, $y, $z. On verra comment utiliser et lier les variables.

XQUERY utilise une partie du langage XPATH.

XPATH permet de naviguer sur un arbre XML.

La base de la syntaxe XPATH est semblable à celle de l'adressage du systéme de fichiers en Unix ou Linux. On peut descendre d'un niveau, etc.

Ses expressions sont dites expressions de chemin (*path expressions*).

Si l’expression de chemin commence par ’/’, alors elle représente un chemin absolu vers l’element requis.

Elément (sous-arbre) interrogé :

```
<AAA>
    <BBB>
        <CCC>
            <FFF>
            </FFF>
            <DDD>
                <EEE>
                </EEE>
            </DDD>
        </CCC>
    </BBB>
</AAA>
```

Requête `/AAA`. **Reponse** : L'élément (arbre) interrogé lui-même.

Requête : `AAA/*/CCC`. Elle calcule selectionne tous les petits-enfants de `AAA` qui sont des `CCC`

L’operateur // permet de sélectionner tous les descendants indiqués.

Elément (arbre) interrogé :

```
<AAA>
   <BBB>
        <CCC>
             <BBB>
             </BBB>
        </CCC>
   </BBB>
</AAA>
```

Requête : `//BB`. Réponse : deux éléments (sous-arbres) :

```
   <BBB>
        <CCC>
             <BBB>
             </BBB>
        </CCC>
   </BBB>
```

et

```
             <BBB>
             </BBB>
```

On va illustrer XQUERY en étudiant des requêtes sur des données bibliographiques, contenues dans le document books.xml :

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book year="1992">
        <title>Advanced Programming in the UNIX Environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
        </book>
    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>65.95</price>
    </book>
    <book year="1999">
        <title>The Economics of Technology and Content
               for Digital TV</title>
        <editor>
           <last>Gerbarg</last>
           <first>Darcy</first>
           <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
</bib>
```

Ce document valide le DTD :

```
<!ELEMENT bib (book* )>
<!ELEMENT book (title, (author+ | editor+ ), publisher, price )>
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author (last, first )>
<!ELEMENT editor (last, first, affiliation )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT affiliation (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

Modèle de données sous-jacent à XQUERY (et XML) :

– Un document est un arbre (ordonné).
– Sept sortes de noeuds :

1. *Document Node* (le tout premier noeud, représentant le document même)
2. *Element Node*
3. *Attribute Node*[a]
4. *Text Node*
5. *Comment Node*
6. *ProcessingInstruction node*
7. *Namespace Node*

– Même si deux noeuds de l'arbre contiennent les mêmes informations, ils sont vus comme 2 individus distincts (ils ont des positions différentes dans l'arbre global !)

[a] bien que nous avons représenté un attribut autrement. Pourquoi ?

Les valeurs des expressions

- Valeur Atomique = une valeur de type atomique : un booléen, un entier, un flottant,.. (Les types atomiques de XQuery sont les mêmes que pour XML-schema).
- Un *item* est ou bien un noeud ou bien une valeur atomique.
- Une valeur d'une expression (requête) XQUERY est une *séquence ordonnée* de zero ou plusieur items.
  - Il n'y a pas de distinction entre un item et une séquence de longueur 1 : (2) = 2.
  - Il n'y a pas de séquence imbriquée : (1, (2,3),(4)) = (1,2,3,4)
  - Une séquence peut être vide (notée ())
  - Une séquence peut conténir des *données hétéreogènes*

La fonction d'entrée `doc()` renvoie un document (le *document node*).

Exemple : `doc(''books.xml'')`

Elle sert à indiquer le document XML interrogé.

En XQUERY des expressions de chemins (XPATH) sont utilisées pour localiser des noeuds :

```
doc(''books.xml'')/bib/book
```

renvoie la séquence des noeuds qui sont des livres dans l'ordre du document. Pour le document de l'exemple, elle est équivalente à :

```
doc(''books.xml'')//book
```

Pourquoi ?

On peut filtrer des noeuds, calculés par une expression de chemin, en utilisant un *prédicat*, c.à.d. une expression booléenne entre crochets :

```
doc("books.xml")/bib/book/author[last="Stevens"]
```

est évaluée à la sequence des noeuds (éléments) n tels que :
- n est un `author`
- le fils `last` de n vaut “Stevens” (c.à.d. l’expression booléenne `last="Stevens"` s’évalue à Vrai).

Cette expression calcule donc les auteurs dont le nom de famille est “Stevens”.

Cas particulier : on a une valeur numérique N entre crochets. C'est interpreté comme le prédicat : "position = N" :

– L'expression suivante renvoie le premier auteur de chaque livre :
  `doc("books.xml")/bib/book/author[1]`
  N.B. La sous-expression `author[1]` est évaluée pour chaque livre.
– Si, par contre, on veut le premier auteur dans le document, il faut écrire :
  `(doc("books.xml")/bib/book/author)[1]`
  Remarquer l'utilisation des parenthèses ici.

Attention :

– Un chaîne de caractéres juste après l'opérateur de descente d'un niveau, est interpretée comme le nom d'un élément fils du noeud courant :
`doc("books.xml")/bib/book/author`
calcule les auteurs des livres.
Remarquer : un noeud `author` est un fils d'un noeud `book`.
– Mais si ce mot commence par le caractére spécial @, c'est compris comme indiquant un attribut du noeud courant :
`doc("books.xml")/bib/book/@year`
calcule les valeurs de l'attribut `year` des livres.
Les noeuds attributs ne sont pas des fils de l'élément dont ils parlent !

Une requête XQUERY peut être utilisée pour créer un nouveau document XML, ou une partie de document.

```
<mon_example>
   <mon_text> J'utilise le document : </mon_text>
   <ma_ref>{doc("books.xml")//book[1]/title}</ma_ref>
</mon_example>
```

s'évalue à l'élement XML (la réponse) :

```
<mon_example>
   <mon_text> J'utilise le document : </mon_text>
   <ma_ref> <title>TCP/IP Illustrated</title></ma_ref>
</mon_example>
```

car :

1) `<mot> ... </mot>` est un constructor qui crée un élément XML dont la balise est "mot"

2) une expression entre { } est une expression qui va être évaluée.

Requête qui crée un document :

```
 document {
  <book year="1977">
    <title>Harold and the Purple Crayon</title>
    <author><last>Johnson</last><first>Crockett</first></author>
    <publisher>HarperCollins Juvenile Books</publisher>
    <price>14.95</price>
  </book>
}
```

Le constructeur `document` crée un noeud document.

Une requête peut aussi restructurer des valeurs existants. La requête suivante liste les titres des livres du document interrogé, et les compte.

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

Réponse :

```
<LesTitres nombre = "4">
   <title>TCP/IP Illustrated</title>
   <title>Advanced Programming in the Unix Environment</title>
   <title>Data on the Web</title>
   <title>The Economics of Technology and Content for
   Digital TV</title>
</LesTitres>
```

NB. Dans la requête précédente :

On construit l'attribut `nombre` de l'élément `LesTitres`.

On pose la valeur de cet attribut comme étant égale à l'évaluation de l'instruction `count` (pre-définie), qui compte.

Un élément de balise bla et contenu strumpf peut être créé en écrivant la requête `<bla> Strumpf </bla>`, comme on a vu. Mais il existe aussi une syntaxe alternative.

La requête suivant crée un élément (qui a un attribut) grâce aux constructeurs : `element` et `attribute` (mots predéfinis !) :

```
element book
{
    attribute year { 1977 },
    element author
    {
        element first { "Crockett" },
        element last { "Johnson" }
    },
        element publisher {"HarperCollins Juvenile Books"},
        element price { 14.95 }
}
```

Expressions FLWOR

Une expression FLWOR :

1) Lie des variables à des valeurs qui sont dans l'espace de valeurs indiqué par le `for` et le `let`.

2) Utilise ces liasons pour créer des valeurs nouveaux.

Une combinaison de liasons de variables crée par le `let` et le `for` est appellée *tuple*.

F : For, **L** : Let, **O** : Order by, **W** : Where, **R** : Return.

On commencera par étudier l'exemple de requête :

```
for $b in doc("books.xml")//book
where $b/@year = "2000"
return $b/title
```

qui calcule les titres des livres publiés en 2000.

```
for $b in doc("books.xml")//book
where $b/@year = "2000"
return $b/title
```

Cette requête lie la variable `$b` à chaque book, un à la fois, pour créer une séquence de tuples. Chaque tuple contient une liaison dans laquelle `$b` est lié à un seul livre.
Le `where` teste chaque tuple, pour voir si `$b/@year` est égal à “2000”.
Avec le `return` on renvoie les valeurs de `$b/title` tels que la valeur de `$b` a satisfait la condition du `where`.

Réponse :

```
<title>Data on the Web</title>
```

car on a un seul livre qui passe le test, dans notre base.

NB. Pas de `let`, dans cette requête. On n’est pas obligé à avoir les 2, `for` et `let`. Ici pas de `order by` non plus.

En général :

- `for` : on associe une variable $x à une expression E, et on crée des tuples où chaque tuple lie la variable $x à un élément de la séquence d'objets qui est la valeur de E Ex : `for $i in (1, 2)` associe la variable $i à l'expression (1,2), qui est une séquence d'entiers, et crée : une liaison de $i à 1 et une liaison de $i à 2 (2 tuples).
- `let` : lie une variable $y au résultat d'une expression E, et ajoute ces liaisons aux tuples générés par le `for`. Ex : `for $i in (1, 2) let $j :=(i, i+1)`. On lie $j à (1,2) (pour le tuple qui dit que $i est 1), puis on lie $j à (2,3) (pour le tuple qui dit que $i est 2).
- `where` : filtrage des tuples selon une condition ;
- `order by` : tri des tuples selon un ordre (croissant, par défaut) ;
- `return` : construction du résultat.

Une expression FLOWR commmence par des `for` et/ou des `let`, après on a un `where optionnel`, puis un `order by` optionnel, puis un `return` obligatoire.

Cet exemple de requête crée des éléments qui s'appellent `tuple` (ma balise, pas prédéfinie !)

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

On associe la variable $i à l'expression E=(1,2,3), et on crée 3 liasons : de $i à 1, de $i à 2 et de $i à 3 (autant de liaisons que d'éléments dans la valeur de E). La réponse est donc :

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

Remarquer la différence avec la requête :

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

dont la réponse contient 1 seul tuple (Pourquoi ?) :

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

Quand le `for` et le `let` sont combinés, les liaisons générées par le `let` sont ajoutés aux tuples générés par le `for` :

```
for $i in (1, 2, 3)  let $j := (1, 2, 3)
return
   <tuple>
      <valeur_de_i>{ $i }</valeur_de_i>  <valeur_de_j>{ $j }</valeur_de_j
   </tuple>
```

dont la réponse est :

```
<tuple>
  <valeur_de_i>1</valeur_de_i><valeur_de_j>1 2 3</valeur_de_j>
</tuple>
<tuple>
  <valeur_de_i>2</valeur_de_i><valeur_de_j>1 2 3</valeur_de_j>
</tuple>
<tuple>
  <valeur_de_i>3</valeur_de_i><valeur_de_j>1 2 3</valeur_de_j>
</tuple>
```

On veut lister les titres des livres :

```
for $b in doc("books.xml")//book
let $c := $b/title
return <book> {$c} </book>
```

qui s’évalue :

```
<book> <title>TCP/IP Illustrated</title> </book>
<book>
       <title>Advanced Programming in the UNIX Environment</title>
</book>
<book> <title>Data on the Web</title> </book>
<book>
       <title>The Economics of Technology and Content for Digital TV</tit
</book>
```

On veut lister les titres des livres et leur nombre d’auteurs.

```
for $b in doc("books.xml")//book
let $c := $b/author
return <book> {$b/title, <nombre> { count($c) }</nombre>}
</book>
```

Réponse

```
<book>
  <title>TCP/IP Illustrated</title>
  <nombre>1</nombre> </book>
<book>
  <title>Advanced Programming in the UNIX Environment</title>
  <nombre>1</nombre>
 </book>
<book>
 <title>Data on the Web</title>
 <nombre>3</nombre> </book>
<book>
  <title>The Economics of Technology and Content for Digital TV</title>
  <nombre>0</nombre>
</book>
```

On peut liér plus qu'une variable avec le `for` :

```
 for $i in (1, 2, 3),
     $j in (4, 5, 6)
return
   <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

Réponse : le produit cartésien :

```
<tuple><i>1</i><j>4</j></tuple>
<tuple><i>1</i><j>5</j></tuple>
<tuple><i>1</i><j>6</j></tuple>
<tuple><i>2</i><j>4</j></tuple>
<tuple><i>2</i><j>5</j></tuple>
<tuple><i>2</i><j>6</j></tuple>
<tuple><i>3</i><j>4</j></tuple>
<tuple><i>3</i><j>5</j></tuple>
<tuple><i>3</i><j>6</j></tuple>
```

On veut seulement les titres des livres qui coûtent moins que 50 dollars :

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

Réponse : Il n’y en a pas. La réponse est la séquence vide.

On veut seulement les titres des livres qui ont plus que 2 auteurs :

```
for $b in doc("books.xml")//book
let $c := $b//author
where count($c) > 2
return $b/title
```

Réponse :

```
  <title>Data on the Web</title>
```

N.B. : let `let` a lié $c à la séquence des auteurs, pour chaque livre valeur de $b, et cette séquence a plus que 2 éléments, pour le livre “Data on the Web”

Lest titres des livres, triés en ordre croissant :

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

Le `for` génére une sequence de tuples, avec un titre par tuple. Le `order by` reorganize cette séquence, en la triant selon la valeur des titres, et le `return` renvoie les titres ainsi triés :

```
<title>Advanced Programming in the Unix Environment</title>
<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
<title>
The Economics of Technology and Content for Digital TV
</title>
```

Toutes les informations sur les auteurs (nom de famille et prénom), mais triées en ordre decroissant, selon le noms de famille, d'abord, puis le prénom :

```
for $a in doc("books.xml")//author
order by $a/last descending, $a/first descending
return $a
```

Réponse :

```
 author>
    <last>Suciu</last>
    <first>Dan</first>
</author>
<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
<author>
    <last>Stevens</last>
    <first>W.</first>
```

```
</author>
<author>
    <last>Buneman</last>
    <first>Peter</first>
</author>
<author>
    <last>Abiteboul</last>
    <first>Serge</first>
</author>
```

Pourquoi on obtient 2 fois W. Stevens ?

Une expression complexe peut apparaître dans un `return`.

Afficher des éléments `info-livre`, contenant le titre et le prix.

```
for $b in doc("books.xml")//book
return
   <info-livre>{ $b/title, $b/price }</info-livre>
```

Réponse :

```
<info-livre>
   <title>TCP/IP Illustrated</title>
   <price>65.95</price> </info-livre>
<info-livre>
   <title>Advanced Programming in the UNIX Environment</title>
   <price>65.95</price>
</info-livre>
<info-livre>
   <title>Data on the Web</title>
   <price>39.95</price> </info-livre>
<info-livre>
  <title>The Economics of Technology and Content for Digital TV</title>
  <price>129.95</price>
</info-livre>
```

Des constructeurs d'éléments peuvent être utilisés pour présenter les données autrement. Par exemple, on veut changer la presentation du nom et du prénom d'un auteur :

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

Réponse :

```
<author>W. Stevens</author>
<author>W. Stevens</author>
<author>Serge Abiteboul</author>
<author>Peter Buneman</author>
<author>Dan Suciu</author>
```

N.B. L'instruction `string`(argument) transforme son argument (ici : un élément de balise `first`, puis un élément de balise `last`), en chaîne de caractéres. Les balises `first` et `last` ne sont pas dans la réponse, qui contient juste les chaînes de caractéres correspondant aux VALEURS des 2 éléments.

Dans le `for` $x `in` *E*, on peut utiliser une variable de position qui donne la position d'un individu dans la sequence valeur de *E* ; il faut utiliser le mot-clé `at`.

Les titres des livres, avec un attribut `pos` qui en donne la position dans la séquence des livres *selon le document* :

```
for $t at $i in doc("books.xml")//title
return <title pos="{$i}">{string($t)}</title>
```

```
<title pos="1">TCP/IP Illustrated</title>
<title pos="2">
Advanced Programming in the Unix Environment</title>
<title pos="3">Data on the Web
</title>
<title pos="4">
The Economics of Technology and Content for Digital
TV
</title>
```

La requête `doc("books.xml")//author/last)` produit :

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

On veut éliminer les répetitions. Si on écrit :
`distinct-values(doc("books.xml")//author/last)`
on obtient : `Stevens Abiteboul Buneman Suciu`.

Mais la balise `last` a disparu, aussi. Pourquoi ? La fonction `distinct-values()` extrait les valeurs d'une séquence de noeuds et crée une séquence de valeurs sans répétitions.

Comparer la requête précédente avec :

```
for $l in distinct-values(doc("books.xml")//author/last)
return <last>{ $l }</last>
```

qui donne la réponse :

```
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

Une requête peut lier plusieurs variables avec le `for` afin de combiner des informations provenant d'expressions différentes, voir de fichiers différents.

Exemple. En plus de books.xml, on va utiliser le fichier suivant, reviews.xml, qui donne des critiques de livres :

```
<reviews>
   <entry>
    <title>TCP/IP Illustrated</title>
    <rating>5</rating>
    <remarks>
    Excellent technical content. Not much plot.
    </remarks>
   </entry>
</reviews>
```

Suite de l'exemple $\Rightarrow$

Requête : Pour chaque livre in books.xml, en donner les remarques qu'on trouve dans review.xml :

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

Réponse :

```
<review>
     <title>TCP/IP Illustrated</title>
     <remarks>
     Excellent technical content. Not much plot.
     </remarks>
</review>
```

C'est une sorte de jointure !

Les deux requêtes :

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

et

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

calculent la même chose : les titres des livres et les remarques, en donnant exclusivement ces livres pour les quels une critique existe.

Par contre, la requête :

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

donne tout livre, accompagné de ses critiques quand elles existent.

D'où vient la différence entre cette requête et les deux précédentes ?

Lister les livres publiés par chaque maison d’édition (*publisher*), en donnant d’abord la maison d’édition, puis tous les livres qu’elle publie.

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

On obtient :

```
<listings>
 <result>
     <publisher>Addison-Wesley</publisher>
   <title>Advanced Programming in the Unix Environment</title>
   <title>TCP/IP Illustrated</title>
    </result>
    <result>
   <publisher>Kluwer Academic Publishers</publisher>
   <title>The Economics of Technology and Content for
   Digital TV</title>
    </result>
    <result>
   <publisher>Morgan Kaufmann Publishers</publisher>
   <title>Data on the Web</title>
    </result>
</listings>
```

Quantificateurs

Quantificateur Existentiel.

L'expression `some` $x `in` E `satisfies` (C($x)) permet de tester si au moins un individu dans la séquence valeur de E satisfait la condition C.

Quels livres ont au moins un auteur qui s'appelle W. Stevens ?

```
for $b in doc("books.xml")//book
where some $a in $b/author
      satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

On obtient :

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
```

Quantificateurs

Quantificateur Universel.

L'expression `every` $x `in satisfies` (C($x)) teste si tout individu dans la séquence valeur de E satisfait la condition C.

Quels livres sont tels que tous leur auteurs s'appellent W. Stevens ?

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

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix Environment</title>
<title>The Economics of Technology and Content for Digital TV</title>
```

Pourquoi on obtient aussi le dernier titre ? Ce livre n'a pas d'auteurs. Donc, pour ce livre, `b/author` s'évalue à une séquence **vide**. Tout élément de la séquence vide satisfait la condition C, c'est une question de logique !

Exemple d'utilisation des quantificateurs pour reorganizer les données.

Lister les livres par auteur (on remarquera aussi les `return` imbriqués).

```
<author-list>
  { let $a := doc("books.xml")//author
    for $l in distinct-values($a/last),
        $f in distinct-values($a[last=$l]/first)
    order by $l, $f
    return
        <author>
          <name>{ $l, ", ", $f }</name>
           { for $b in doc("books.xml")/bib/book
             where some $ba in $b/author satisfies
                   ($ba/last=$l and $ba/first=$f)
             order by $b/title
             return $b/title }
        </author> }
</author-list>
```

On obtient :

```
 author-list>
    <author>
        <name>Stevens, W.</name>
        <title>Advanced Programming in the Unix Environment</title>
        <title>TCP/IP Illustrated</title>
    </author>
    <author>
        <name>Abiteboul, Serge</name>
        <title>Data on the Web</title>
    </author>
    <author>
        <name>Buneman, Peter</name>
        <title>Data on the Web</title>
    </author>
    <author>
        <name>Suciu, Dan</name>
        <title>Data on the Web</title>
    </author>
</author-list>
```

Expressions Conditionnelles

Pour chaque livre, donner ses 2 premiers auteurs, et “et. al” s’il en a d’autres.

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

Réponse :

```
 book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens, W.</author>
</book>
<book>
    <title>Advanced Programming in the Unix Environment</title>
    <author>Stevens, W.</author>
</book>
<book>
    <title>Data on the Web</title>
    <author>Abiteboul, Serge</author>
    <author>Buneman, Peter</author>
    <author>et al.</author>
</book>
<book>
```

```
    <title>The Economics of Technology and Content for
    Digital TV</title>
</book>
```

N.B. Pour la liaison de $b au dernier livre, qui n’a pas d’auteur, l’évaluation de

```
{
      for $a at $i in $b/author
      where $i <= 2
      return <author>{string($a/last), ", ",
                    string($a/first)}</author>
}
```

a donné le résultat vide, pas un message d’erreur.

Operateurs

En XQUERY :

Operateurs Arithmetiques

Operateurs de Comparaison (de plusieurs sortes, voir après)

Operateurs sur les séquences de noeuds.

Operateurs Arithmetiques

+, -, *, `div` (division pour des types numériques qques), `idiv` (division sur les entiers), `mod` (modulo).

NB : 2 + (), 2 * (),...etc s'évaluent à la séquence vide (), qui se comporte comme le `NULL` de SQL.

Operateurs de Comparaison

Les ***value comparaison operators*** ne sont pas la même chose que les ***general comparison operators***, même s'il y a une correspondance entre eux :

Value Comparison Operators	General Comparison Operators
`eq`	`=`
`ne`	`!=`
`lt`	`<`
`le`	`<=`
`gt`	`>`
`ge`	`>=`

Quelles sont les differences entre le premier groupe (VCO) et le second (GCO) ?

1. Les VCO sont stricts par rapport au types : on peut comparer des strings avec des strings, des nombre decimales avec des decimales, etc.
   Les GCO sont plus souples.
2. Si un VCO op est appliqué à un argument qui est une séquence S de longueur supérieure à 1, on a un message d'erreur.
   Ce n'est pas le cas pour un GCO : une expression $S\ op\ a$ est lue : $\exists s \in S\ :\ s\ op\ a$?

Illustration de la première différence.

– Avec le VCO `gt` (*greater than*) :

```
for $b in doc("books.xml")//book
where $b/price gt 100.00
return $b/title
```

Le DTD n'a pas défini le type de `price` comme étant un décimal (avec XML-schéma on aurait pu le faire). La comparaison `gt` avec un décimal donne une erreur.

– Avec le GCO `>` :

```
for $b in doc("books.xml")//book
where $b/price > 100.00
return $b/title
```

Puisque 100.00 est un décimal, le `>` converti le prix à un décimal.

Illustration de la seconde différence.

– Avec le VCO `eq` (*equal to*) :

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

Puisque un livre peut avoir plusieurs auteurs, l'expression `$b/author/last` peut s'évaluer à une séquence d'auteurs qui contient plus qu'un élément : erreur !

– Avec le GCO `=` :

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

Pas d'erreur ! En fait, `$b/author/last eq "Stevens"` est lue comme :

Existe-t-il un auteur du livre $b tel que son nom de famille (`last`) vaut "Stevens" ?

Mais, attention, le comportement du GCO = vis à vis d'une séquence peut reserver des surprises !

```
for $b in doc("books.xml")//book
where $b/author/first = "Serge"
  and $b/author/last = "Suciu"
return $b
```

At-on démandé quels sont livres dont un auteur s'appelle "Serge Suciu" ? (Remarquer qu'il n'y en a pas).

Non ! On obtient la réponse :

```
<book year = "2000">
  <title>Data on the Web</title>
  <author>
    <last>Abiteboul</last> <first>Serge</first>
  </author>
  <author>
    <last>Buneman</last> <first>Peter</first>
  </author>
  <author>
    <last>Suciu</last> <first>Dan</first>
  </author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
</book>
```

Pourquoi ?

La condition C après le `where` :

```
      $b/author/first = "Serge"
  and $b/author/last = "Suciu"
```

a été lue :
$\exists$ un auteur $a1$ du livre $b tel que le `first` de $a1$ vaut ‘‘`Serge`’’ et $\exists$ un auteur $a2$ du livre $b tel que le `last` de $a2$ vaut ‘‘`Suciu`’’.
Le livre Data on the Web satisfait C !

Si on veut demander quels sont livres dont un auteur s’appelle “Serge Suciu’, il faut écrire :

```
for $b in doc("books.xml")//book,
    $a in $b/author
where $a/first="Serge"
  and $a/last="Suciu"
return $b
```

Ici, on raisonne sur un auteur donné, valeur de $a.

Operateurs de Comparaison, Suite : **Operateurs de Comparaison de Noeuds**

Rappel : chaque noeud est identifié de façon unique par sa position dans l'arbre, son contenu ne suffit pas !

n `is` m teste si n et m sont le même noeud, et n `is not` m teste s'ils ne le sont pas.

Est-il vrai que le livre les plus cher coincide avec le livre qui a le nombre le plus grand d'auteurs et éditeurs ?

```
let $b1 := for $b in doc("books.xml")//book
           order by count($b/author) + count($b/editor)
           return $b
let $b2 := for $b in doc("books.xml")//book
           order by $b/price
           return $b
return $b1[last()] is $b2[last()]
```

où la fonction pre-définie `last()` calcule le dernier élément d'une séquence.

Observer :

n = m, en révanche, teste si la valeur (contenu) des deux noeuds est le même.

n = m et n `is not` m peuvent être vrais au même temps !

Un autre operateur de comparaison de nodes est << qui teste si un noeud n précéde un noeud m dans l'ordre du document.

Quels sont les livre dont Abiteboul est bien un auteur, mais pas le premier auteur ?

```
for $b in doc("books.xml")//book
let $a := ($b/author)[1],
    $sa := ($b/author)[last="Abiteboul"]
where $a << $sa
return $b
```

Réponse : il n'y en a pas.

Opérateurs sur les Séquences de noeuds

`union` (ou `|`), `intersect` et `except` combinent 2 séquences de noeuds, et la séquence résultat suit l'ordre du document.

`union` (ou `|`) : union

`intersect` : intersection

`except` : différence.

Donner la liste triée des noms de famille des auteurs et des editeurs :

```
let $l :=
  distinct-values(doc("books.xml")//(author | editor)/last)
order by $l
return <last>{ $l }</last>
```

Réponse :

```
<last>Abiteboul</last>
<last>Buneman</last>
<last>Gerbarg</last>
<last>Stevens</last>
<last>Suciu</last>
```

Donner toutes les informations du livre TCP/IP Illustrated, sauf son prix.

```
for $b in doc("books.xml")//book
where $b/title = "TCP/IP Illustrated"
return
   <book>
       { $b/@* } { $b/* except $b/price }
   </book>
```

Réponse :

```
<book year = "1994">
<title>TCP/IP Illustrated</title>
<author>
    <last>Stevens</last>
    <first>W.</first>
</author>
<publisher>Addison-Wesley</publisher>
</book>
```

Fonctions Pre-définies

En XQUERY on a des fonctions `min`, `max`, `count`, `sum` et `avg` analogues à celles de SQL. On a déjà vu des exemples avec `count`.

Quels livres sont plus chers que la moyenne ?

```
let $b := doc("books.xml")//book
let $avg := average( $b//price )
return $b[price > $avg]
```

On obtient :

$\Rightarrow$

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

N.B : `price` est le nom d'un élément, pas un type tel que on puisse calculer le maximum de $< valeur_1(price), ..., valeur_n(price) >$, mais `average` extrait la séquence des valeurs de `price`, et fait la conversion réquise.

On a aussi des fonctions numériques comme `round`, `floor` et `ceiling`.

On a des fonctions des chaînes de caractères comme :
`concat`,
`string-length`,
`starts-with`,
`end-with`,
`substring`,
`upper-case`,
`lower-case`.

Puis : `distinct-values` et `doc` (déjà vues).

Et encore : `not`.

Quels sont les livres dont aucun auteur s'appelle Stevens ?

```
for $b in doc("books.xml")//book
where not(some $a in $b/author satisfies $a/last="Stevens")
return $b
```

La fonction `empty` teste si une séquence est vide.

Quels sont les livres qui ont au moins un auteur ?

```
for $b in doc("books.xml")//book
where not(empty($b/author))
return $b
```

On aurait pu aussi écrire :

```
for $b in doc("books.xml")//book
where exists($b/author)
return $b
```

La fonction `string`, appliqué à un noeud, renvoie la réprésentation sous forme de chaîne de caractères du texte trouvé dans le noeud . Par exemple, la requête :

```
string((doc("books.xml")//author)[1])
```

calcule la chaîne `‘‘Stevens W.’’` (le nom du premier auteur dans le document).

La fonction `data`, appliqué à un noeud, renvoie la valeur typée d’un noeud ; exemples d’utilisation de cette fonction dans le TD.

Fonctions définies par l'utilisateur

On peut définir une fonction qui calcule une requête donnée, et la réutiliser. Par exemple :

```
define function books-by-author($last, $first)
  as element()*
{
  for $b in doc("books.xml")/bib/book
  where some $ba in $b/author satisfies
         ($ba/last=$last and $ba/first=$first)
  order by $b/title
  return $b/title
}
```

définie une fonction qui crée une fonction qui liste les livres par auteur, et la nomme `books-by-author`.

XQUERY supporte la définition de fonctions récursives.

Suppoons qu'un chapitre de livre est constitué de sections, qui peuvent être imbriquées. La requête suivante crée une table des matières, containant les sections et leur titres, selon la structure des sections du livre.

```
define function toc($book-or-section as element())
  as element()*
{for $section in $book-or-section/section
  return
    <section>
      { $section/@* , $section/title , toc($section) }
    </section>}
<toc>
   {
     for $s in doc("xquery-book.xml")/book
     return toc($s)
   }
</toc>
```