XML et outils associés - Cefora/Technofutur TIC

Table des matières succinte

Chapitre 1 - Introduction générale

Table des matières de ce chapitre

Les langages

Trois langages permettent de traiter et de modifier le contenu d'un ou de plusieurs documents XML:

- **XSLT** qui est un langage de transformation
- **XPath** qui est un langage d'interrogation
- **XQuery** qui est un langage de requête

Xpath est la pierre angulaire des deux autres langages: il est utilisé tant par XSLT que par XQuery dès qu'il faut extraire des données d'un document XML ou calculer quelque chose

Le language XPath

XPath est un langage générique d'interrogation. Il permet d'écrire des **expressions** pour aller chercher des données dans un document XML:

```
/carnet/personne

/carnet/personne[1]/adresse/fn:concat(numero,' ',rue,', ',zip,' ',commune)

fn:count(fn:distinct-values(//zip))
```

XPath permet également d'écrire des expressions plus classiques pour calculer divers résultats:

```
1 + 1

$nom='' or $prenom=''
```

XPath en général ne s'utilise pas seul, il est au service d'une autre langage appelé le **langage hôte**: XSLT, XQuery, XML Schema, XForms...

Le language XSLT

XSLT est un langage générique de transformation:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">

  <xsl:template match="/">
    <résultats>
      <xsl:apply-templates select="/carnet/personne"/>
    </résultats>
  </xsl:template>

  <xsl:template match="personne">
    <xsl:copy>
      <xsl:value-of select="concat(nom, ' ',prenom)"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Résultat de la transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<résultats>
  <personne>Bernard John</personne>
  <personne>Durand Elise</personne>
  <personne>Gobbe Nathalie</personne>
  <personne>Gobbe Jean-Luc</personne>
  <personne>Bareel Valérie</personne>
  <personne>Albert Marc</personne>
  <personne>Declerck Serge</personne>
  <personne>Dumont Julie</personne>
  <personne>Jansens Sophie</personne>
  <personne>Pireaux Eric</personne>
  <personne>Da vinci Martine</personne>
</résultats>
```

Le language XQuery

XQuery est un langage qui permet d'écrire des requêtes allant chercher des données dans plusieurs documents XML:

```
<résultat>
  {
  for $p in /carnet/personne
  return <personne>
            {
            $p/adresse/fn:concat(../nom,' ',../prenome)
            }
         </personne>
  }
</résultat>
```

Résultat de la requête

```
<?xml version="1.0" encoding="UTF-8"?>
<résultats>
  <personne>Bernard John</personne>
  <personne>Durand Elise</personne>
  <personne>Gobbe Nathalie</personne>
  <personne>Gobbe Jean-Luc</personne>
  <personne>Bareel Valérie</personne>
  <personne>Albert Marc</personne>
  <personne>Declerck Serge</personne>
  <personne>Dumont Julie</personne>
  <personne>Jansens Sophie</personne>
  <personne>Pireaux Eric</personne>
  <personne>Da vinci Martine</personne>
</résultats>
```

XSLT et XPath

XSLT et Xpath sont indissociables. Une feuille de transformation XSLT fait appel à des instructions qui pour la plupart utilisent des expressions XPath:

```
<xsl:for-each select="/carnet/personne">
  ...
```

```
</xsl:for-each>

<xsl:value-of select="child::nom"/>

<xsl:copy-of select="descendant::billingAddress"/>

<xsl:template match="personne/descendant::text()">
  ...
</xsl:template>
```

XQuery et XPath

XQuery et XPath sont encore plus indissociables: plus de 90% de leur syntaxe est commune.

XQuery est une extension à XPath pour:

- rendre les expressions, appelées ici requêtes, autonomes (sans nécessiter de langage hôte);
- permettre des requêtes sur de multiples documents.

XSLT et XQuery

Depuis la version 2.0 de XSLT, celui-ci peut transformer plusieurs documents XML, formant une base de données de documents XML comme en XQuery.
Le résultat obtenu peut également être très similaire à celui obtenu par XQuery.

La différence principale réside dans l'approche qu'ont ces deux langages:

- XSLT peut s'apparenter à de la programmation procédurale en exécutant des modèles (templates) les uns après les autres;
- XQuery peut s'apparenter à de la programmation déclarative et fonctionnelle, plus naturelle que XSLT, mais moins modulaire et plus restrictive.

Les différentes versions

XSLT 1.0 *Transformation simple (un document en entrée, un document en sortie)*	XPath 1.0 *Typage de données simple (ensemble de noeuds, valeurs numériques, booléennes ou chaînes de caractères)*	-
XSLT 2.0 *Adapté aux expressions XPath 2.0 et aux séquences, plusieurs documents en entrée et en sortie.*	XPath 2.0 *Introduction des séquences et de tous les types de données de XML Schema*	XQuery 1.0 *Language de requêtes dans une base de données de documents XML, basé sur XPath 2.0*
XSLT 3.0 *Introduction du streaming (permet de traiter des documents source*	XPath 3.0 *fonctions dynamiques, nouveaux opérateurs...*	XQuery 3.0

très volumineux), support de JSON, possibilité de merge, de traitements parallèles...		*Introduction des clauses count et group by, des fenêtres glissantes, de nouveaux opérateurs...*
XSLT 3.0	XPath 3.1 *Support de JSON, introduction des maps et des arrays*	XQuery 3.1 *Support de JSON, introduction des maps et des arrays*

Le modèle de données XDM

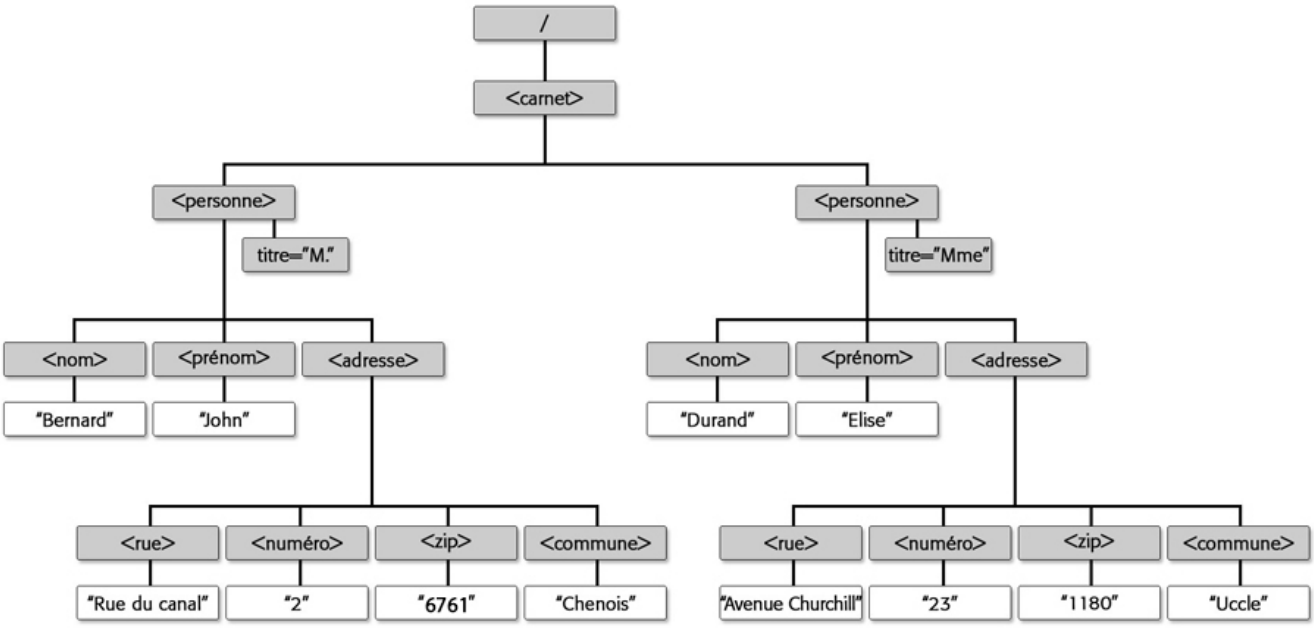
Le modèle de données traitées par XPath (à partir de la version 2), XSLT (à partir de la version 2) et XQuery est appellée: une instance XDM (pour XQuery and XPath Data Model).

Une **instance XDM** est une manière générique de représenter les données en question. Elle est utilisée dans les nouvelles versions des langages XPath, XSLT et XQuery (et le sera dans plusieurs autres langages par la suite).

De manière générale, une instance XDM est une **séquence** d'éléments qui peuvent être soit des noeuds, soit des valeurs atomiques.

Les instances XDM - l'arbre document

La plupart du temps, une instance XDM se résume à une séquence d'un ou plusieurs noeuds de type document. Chacun de ces noeuds est la racine d'un **arbre document** (document tree):



Les instances XDM

Tout ce qui peut ressembler de près ou de loin à du XML peut être généré sous la forme d'une instance XDM: un ensemble de fichiers dans un répertoire, une table dans une base de données relationnelle, etc.

```
<table>
  <row>
    <col>...</col>
    <col>...</col>
    <col>...</col>
  </row>
  <row>
    <col>...</col>
    <col>...</col>
    <col>...</col>
  </row>
  <row>
    <col>...</col>
    <col>...</col>
    <col>...</col>
  </row>
  ...
</table>
```

Chapitre 2 - Le langage XPath

Table des matières de ce chapitre

Le langage Xpath, c'est quoi ?

Le langage XPath permet d'écrire des **expressions** pour accèder à de l'information stockée dans un document XML. Son nom vient des expressions typiques du language: les **chemins d'accès** ou *location paths* (*path expressions* dans les dernières versions).

```
/child::carnet/child::personne
```

```
ancestor::personne/following-sibling::personne[1]/descendant::zip
```

```
1 + 1
```

Les expressions XPath prennent la forme d'une simple chaîne de caractères, utilisables dans la valeur d'un attribut.

XPath est un langage au service d'un autre langage, appelé le **langage hôte**. Celui-ci peut être XSLT, XQuery, XML Schema, XForms, XPointer...

Les nouveautés de la version 2.0 de XPath

Les versions 2.0 et suivantes de XPath sont bien plus riches que la version 1.0. On y a introduit un **typage fort des données**.

```
1 + 1 cast as xs:decimal
```

```
/descendant::element(*, xs:integer)
```

Les concepts de XPath 1.0 ont été totalement généralisés ce qui a renforcé la cohérence et la lisibilité du langage. On y a introduit la notion de **séquence**: une expression XPath retournera toujours comme valeur une séquence.

Les séquences 1/5

Les séquences sont la base du langage XPath (version 2.0 et suivantes). Tout y est séquence: toutes les fonctions, tous les opérateurs, toutes les expressions retournent toujours une séquence.

Une **séquence** (*sequence*) est une collection ordonnée de zéro, un ou plusieurs **éléments** (*item*, à ne pas confondre avec des noeuds éléments).

Un **élément** peut être :

- Une **valeur atomique** (*atomic value*) est une valeur compatible avec l'un des types atomiques défini dans la norme XML Schema (`xs:string`, `xs:integer`, `xs:decimal`, `xs:boolean`, `xs:date`...). Il ne peut s'agir que d'un type simple.
- Un **noeud** (*node*) est un des noeuds d'un arbre document (noeud document, noeud élément, noeud attribut, noeud texte, noeud commentaire ou noeud instruction de traitement).

Les séquences 2/5

Une **séquence** est une généralisation du concept d'**ensemble de noeuds** qui existait dans l'ancienne version de XPath.

Les principales différences sont que :

- les séquences peuvent, en plus des noeuds, contenir des valeurs atomiques;
- les séquences peuvent contenir des doublons;
- les séquences ne sont plus ordonnées dans l'ordre du document XML.

Les séquences 3/5

Le moyen le plus simple pour créer une séquence est d'utiliser l'opérateur **,** (la virgule). Cet exemple crée une séquence de valeurs atomiques comprenant les nombres de 1 à 6 :

```
1, 2, 3, 4, 5, 6
```

On peut également utiliser l'opérateur **to** qui crée une séquence entre deux valeurs atomiques :

```
1 to 6
```

On peut placer des noeuds créé à l'aide d'un **constructeur d'élément** :

```
<item>1</item>, <item>2</item>, <item>3</item>
```

On peut utiliser des **chemins d'accès** (*path expressions*) :

```
/carnet/personne[1], /carnet/personne[2], /carnet/personne[3]
```

On peut tout mélanger :

```
1 to 3, "hello", <item>4</item>, /carnet/personne[1], /carnet/personne[2]
```

Les séquences 4/5

Une séquence qui ne contient qu'un seul élément est appelée un **singleton** :

```
145
```

```
"Hello World"
```

```
<résultat><message>Hello World!</message></résultat>
```

```
/carnet/personne[1]
```

Les séquences 5/5

Une séquence peut être **vide**. Pour en construire une, on utilisera des parenthèses vides :

```
()
```

Une séquence peut être construite à l'aide d'autres séquences. Mais comme une séquence ne peut contenir que des éléments (des valeurs atomiques ou des noeuds), le résultat sera une séquence "aplatie" comme le montre l'expression suivante :

```
1, (2, 3), 4, (), 5, (6, (7 to 9))
```

Ce qui produira le résultat suivant :

```
1, 2, 3, 4, 5, 6, 7, 8, 9
```

Les expressions

Les expressions sont construites à l'aide d'**opérateurs** (+, -, *****, div, and, or...) appliqués à des **opérandes** qui prennent la forme d'expressions de type **chemin d'accès**. Un chemin d'accès aura toujours pour résultat une séquence. Le calcul d'une expression donnera toujours une séquence.

L'expression :

```
1 + 1
```

ne doit pas être vue comme l'addition de deux valeurs entières, dont le résultat est également une valeur entière égale à `2`, mais comme l'addition de deux chemins d'accès, donnant des séquences consitutées d'un seul élément de type atomique `xs:integer`.

Le résultat sera également une séquence (un singleton) ne contenant qu'un seul élément de type atomique `xs:integer` dont la valeur est égale à `2`.

Les expressions chemin d'accès

Un **chemin d'accès** est constitué d'une ou plusieurs **étapes**, qui sont soit des **expressions primaires**, soit des **étapes basées sur un axe**.

```
/carnet/personne/nom

ancestor::carnet/personne[adresse/zip = '1050']/concat(prénom, ' ', nom)

"Hello"

/child::carnet/child::personne/"Hello"

$prix
```

Un chemin d'accès donnera toujours comme résultat une séquence.

Le contexte

Le calcul d'une expression XPath se fera toujours par rapport à un **contexte** donné. Ce contexte est déterminé par le langage hôte qui fait appel à XPath (XSLT, XQuery, XForms...). Le contexte comprend toutes les informations qui peuvent influencer le résultat du calcul d'une expression.

Il est composé d'un **contexte dynamique** et d'un **contexte statique**

Le contexte dynamique comprend :

- un **élément contexte** (*context item*), également appelé élément courant (il s'agit souvent d'un noeud)
- une **taille de contexte** (entier >= 1)
- une **position de contexte** (entier >=1 et <= taille de contexte)

Le contexte statique comprend (dépendant du langage hôte) :

- un ensemble de **variables** (ensemble de paires nom = valeur)
- un ensemble de **types**, de **déclarations** (éléments, attributs...) définis par les schémas éventuels
- un ensemble de **fonctions**
- un ensemble d'**espaces de noms**
- un ensemble de **documents** ou de **collections** accessibles (la base de données !)

Le contexte statique et le contexte dynamique

Le **contexte statique** est déterminé par le langage hôte (chacun à ses règles) avant le calcul d'une expression et ne change pas lors de ce calcul.
Le **contexte dynamique** est également intialement déterminé par le langage hôte, mais change de valeurs au fur et à mesure du calcul d'un chemin d'accès.
Ce changement de valeurs sera opéré à chaque calcul d'une étape ou d'un prédicat:

```
étape1/étape2
```

```
étape[prédicat]
```

Une fois le calcul terminé, ce contexte dynamique revient aux valeurs initiales.

Les étapes constituées d'une expression primaire

Une étape peut prendre la forme d'une **expression primaire**, qui peut être:

- un littéral (constante)
- une variable (le nom commence par un $)
- le '.' (représente l'élément contexte)
- un appel à une fonction
- une expression entre parenthèses

```
"dupont"
```

```
438
```

```
$a1

.

fn:count(child::*)

(1 to 10)
```

Les étapes basées sur un axe 1/3

Une étape peut être une **étape basée sur un axe** :

```
child::nom

descendant::*

ancestor::personne

node()
```

L'étape sera dans ce cas toujours constituée :

- d'un **axe de recherche** (s'il n'est pas présent, `child::` est utilisé par défaut)
- d'un **noeud test**

Une étape basée sur un axe permet de sélectionner une séquence de noeuds.

Les étapes basées sur un axe 2/3

Les axes de recherche peuvent valoir :

```
child::
descendant::
descendant-or-self::
following-sibling::
following::
attribute::
namespace::

self::

parent::
ancestor::
ancestor-or-self::
preceding-sibling::
preceding::
```

L'axe de recherche indique, par rapport au noeud contexte, où il faut regarder dans l'arbre pour sélectionner des noeuds. L'axe de recherche par défaut est `child::`.

L'axe `attribute::` peut également s'écrire `@` (`attribute::titre` et `@titre` sont équivalents)

Les étapes basées sur un axe 3/3

Les noeuds tests peuvent valoir :

```
<_low>nom</_low>, <_low>prefix</_low>:<_low>nom</_low>, *, <_low>prefix</_low>:*, *:<_low>nom</
_low>, Q{<_low>uri</_low>}<_low>nom</_low>

text()
comment()
processing-instruction(), processing-instruction(<_low>nom</_low>)
namespace-node()
node()

element(), element(<_low>nom</_low>), element(<_low>nom</_low>, <_low>type</_low>), element(*, <_low>type</
_low>)
attribute(), attribute(<_low>nom</_low>), attribute(<_low>nom</_low>, <_low>type</
_low>), attribute(*, <_low>type</_low>)
```

L'axe de recherche indique, par rapport à l'axe de recherche, quels noeuds il faut sélectionner.

Les chemins d'accès constitués de plusieurs étapes

Lorsque plusieurs étapes existent, elles doivent toutes retourner une séquence ne contenant que des noeuds sauf la dernière qui peut retourner une séquence normale, composée de noeuds et/ou de valeurs atomiques.

```
/child::carnet/child::personne/child::nom

/carnet/personne/nom

/carnet/personne/fn:concat(prénom, ' ', nom)
```

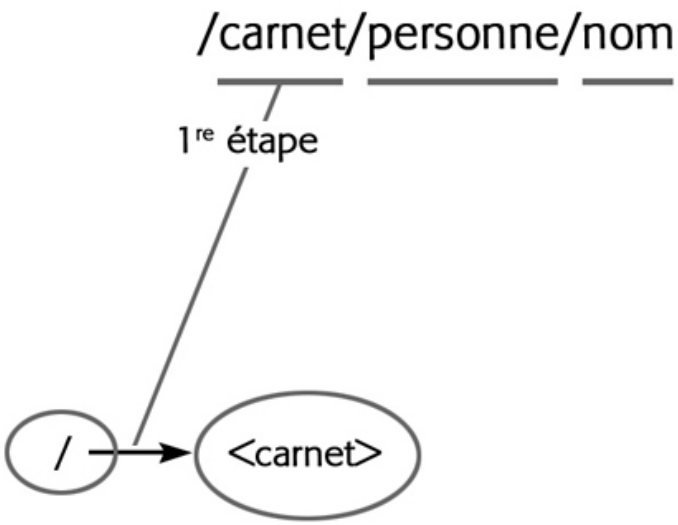
Cette restriction est imposée par la manière de calculer l'expression, étape par étape.

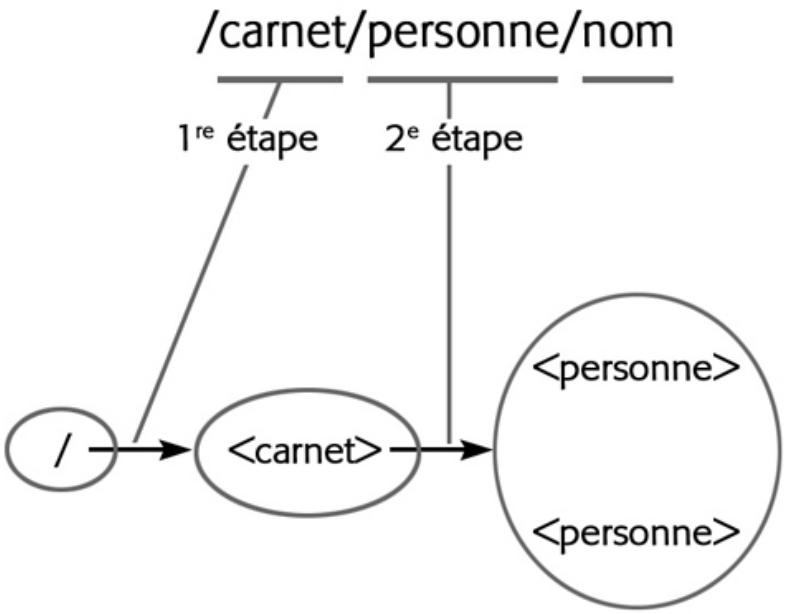
Les étapes dans les chemins d'accès

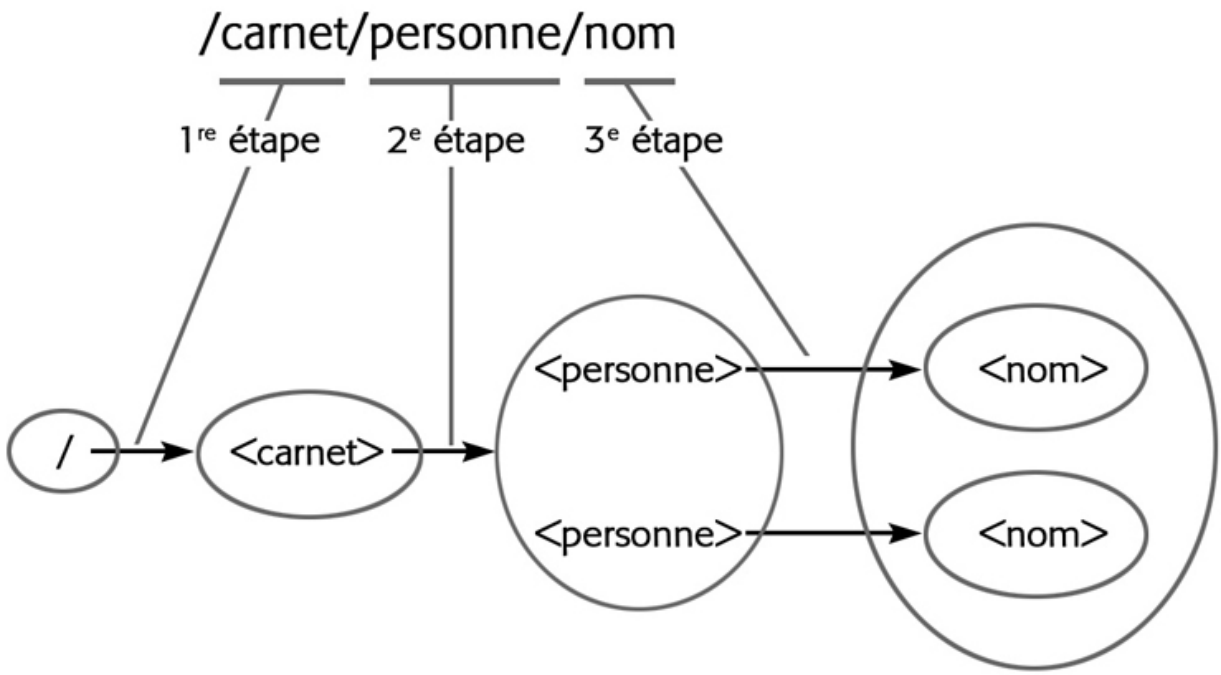
Chaque étape, sauf la dernière, sert à sélectionner une séquence de noeuds. Cette séquence est utilisée pour calculer l'étape suivante, qui sera **calculée pour chaque noeud** de la séquence en adaptant à chaque fois le contexte dynamique :

- l'élément contexte est le noeud en question
- la taille de contexte est égal au nombre de noeuds de la séquence
- la position de contexte est la position de ce noeud dans la séquence

La séquence obtenue au cours de la dernière étape, qui peut contenir autre chose que des noeuds, sera le résultat final de l'expression

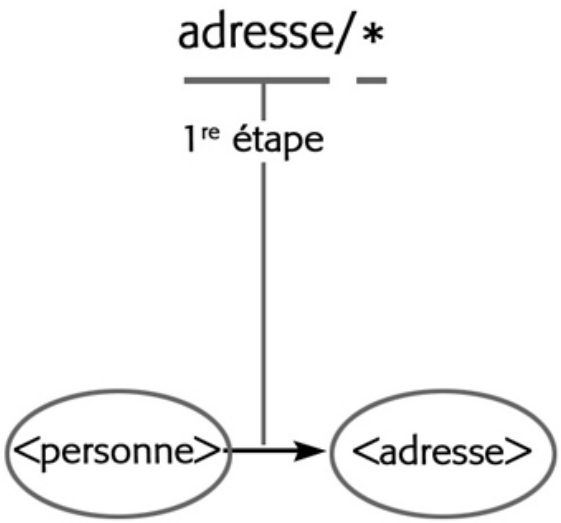

/carnet/personne/nom
1re étape
/
<carnet>

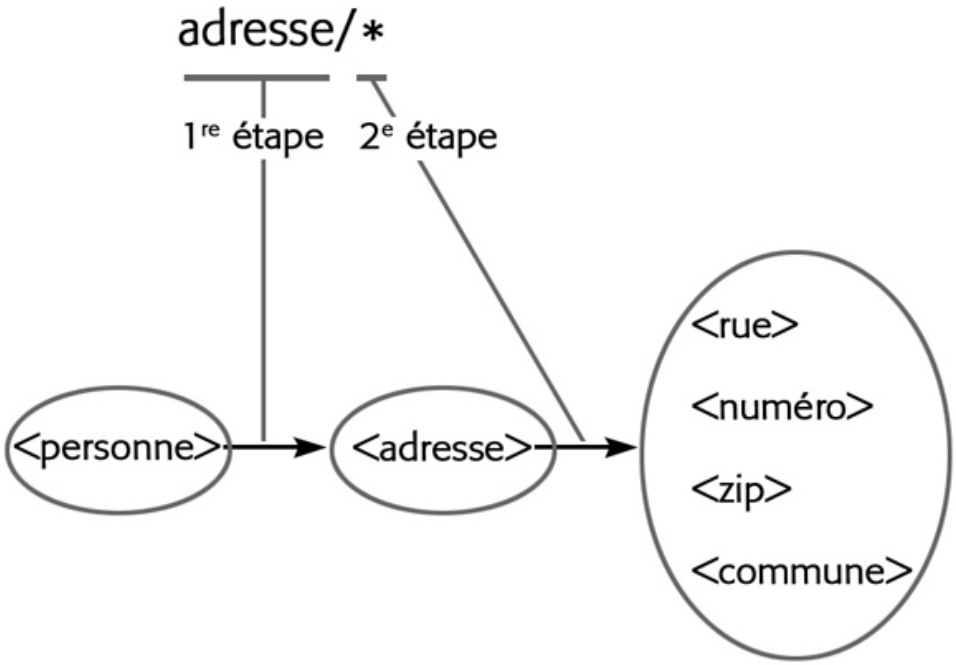

/carnet/personne/nom
1re étape
2e étape
/
<carnet>
<personne>
<personne>


/carnet/personne/nom
1re étape
2e étape
3e étape
/
<carnet>
<personne>
<personne>
<nom>
<nom>

Les chemin d'accès absolus et relatifs

- Dans le cas d'un **chemin d'accès absolu**, le point de départ est la racine du document (un / au début de l'expression)
- Dans le cas d'un **chemin d'accès relatif**, le point de départ est l'**élément contexte** (qui doit être un noeud)

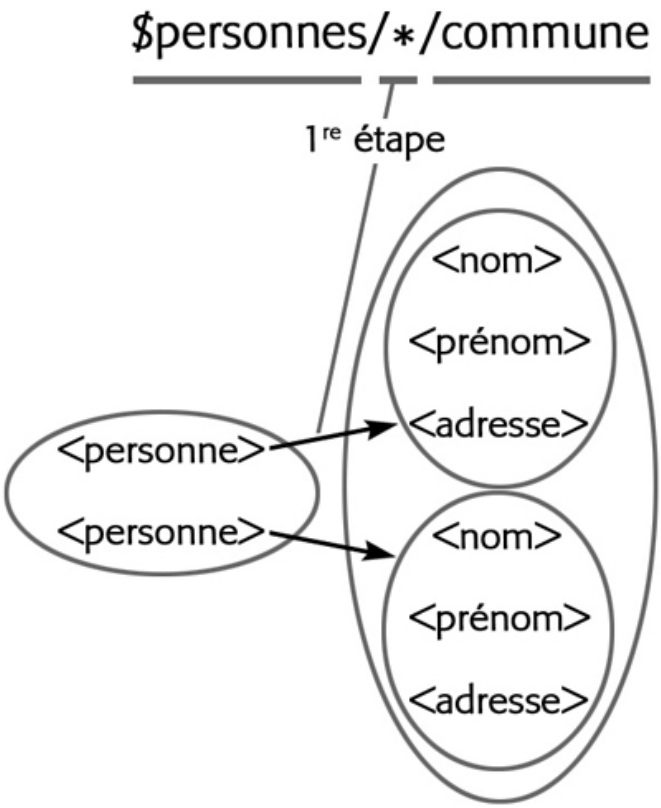


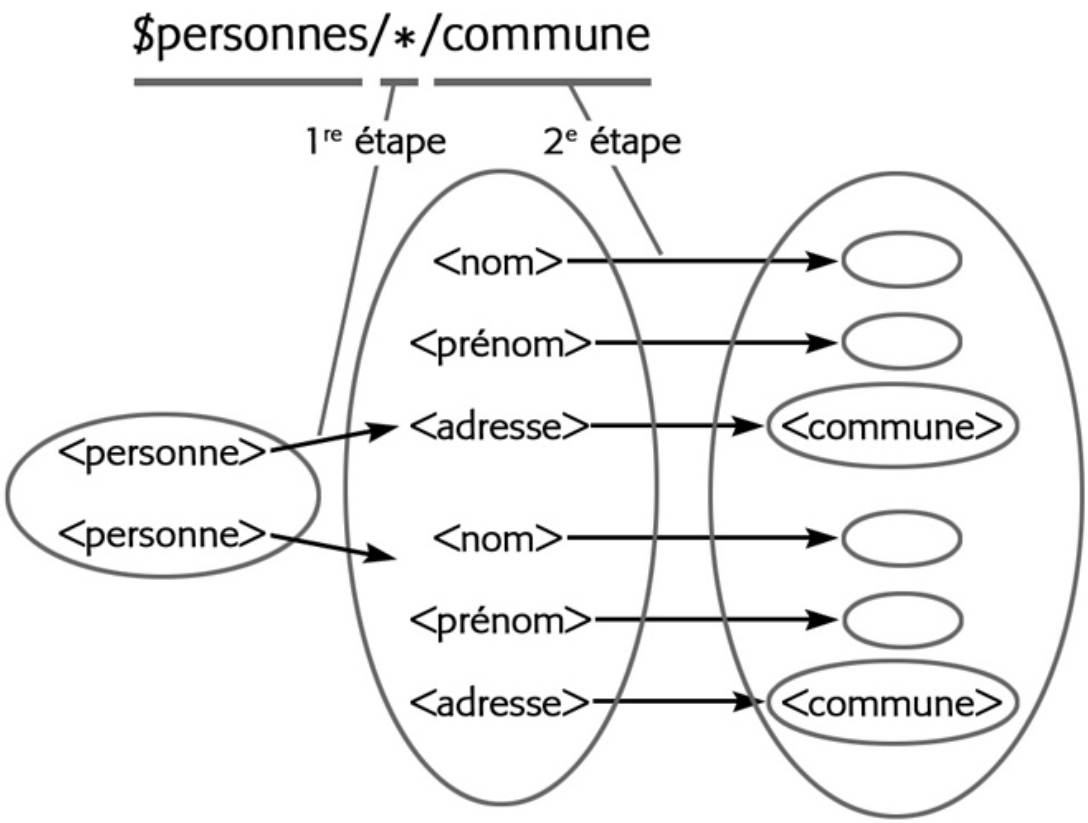


Le point de départ du calcul d'un chemin d'accès

De manière plus générale, n'importe quelle expression XPath qui retourne un séquence de noeuds peut servir de point de départ, par exemple :

- une variable qui contient une séquence de noeuds
- une fonction qui retourne une séquence de noeuds
- une expression entre parenthèses qui retourne une séquence de noeuds





Les prédicats

Chaque étape, que ce soit une expression primaire ou une étape basée sur un arbre, peut être suivie d'un ou plusieurs **prédicats** :

```
personne[2]

personne[adresse/zip eq '1050']

ancestor::personne[@titre='M.'][.//codePays = 'BE']

$prix[@quantité >= 10]

(1 to 10)[. mod 2 eq 1]
```

Les prédicats vont servir à filter la séquence d'éléments obtenue lors du calcul de l'étape.

Les prédicats de position

Si un prédicat retourne une valeur qui est une séquence ne contenant qu'une valeur numérique entière, il s'agit d'un prédicat de position.
Seul l'élément à la position donnée par le prédicat sera conservé (la numérotation de la position commence à 1).

```
/carnet/personne[2]/adresse/rue
```

```
following-sibling::*[1]

following-sibling::*[last()]
```

Dans ces exemples, le calcul se fait comme si le prédicat valait :

```
/carnet/personne[position() = 2]/adresse/rue

following-sibling::*[position() = 1]

following-sibling::*[position() = last()]
```

Les prédicats autres que ceux de position

Le prédicat est calculé **individuellement** pour chaque noeud de la séquence obtenue dans l'étape, en modifiant le contexte dynamique de la manière suivante :

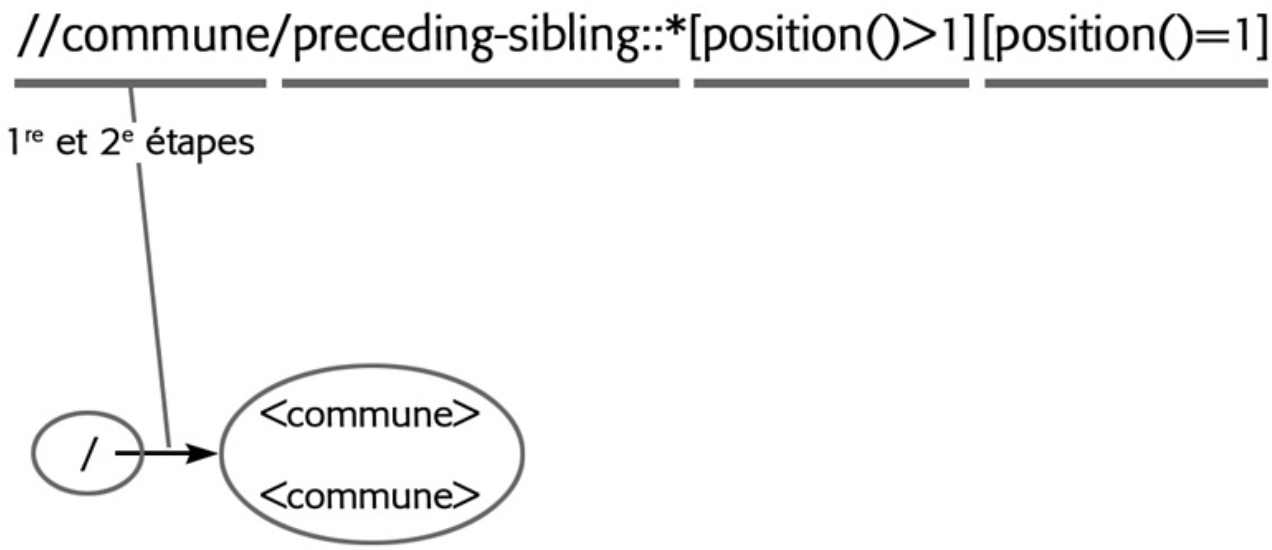
- l'élément contexte est le noeud en question
- la taille de contexte est égal au nombre de noeuds de la séquence
- la position de contexte est la position de ce noeud dans la séquence

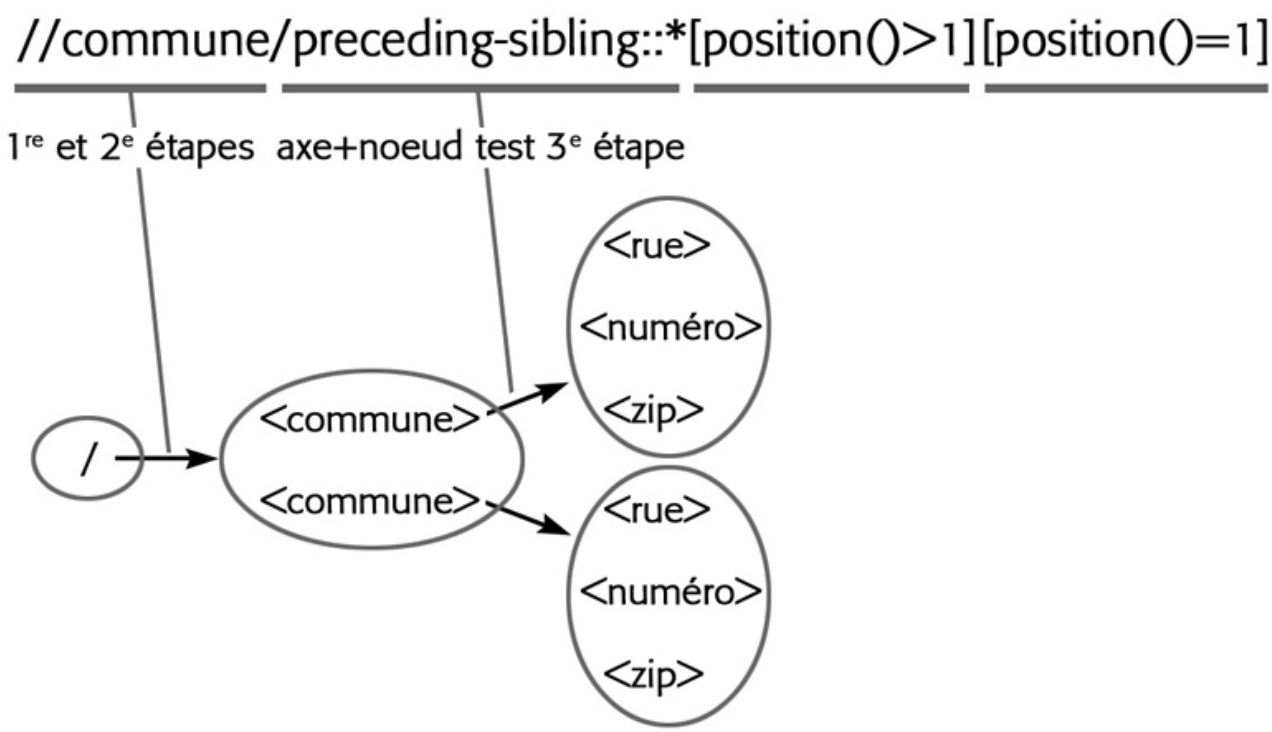
Les séquences sont ordonnées dans l'ordre du document s'il s'agit d'un axe avant, ou dans l'ordre inverse s'il s'agit d'un axe arrière. En d'autres termes, le noeud en position 1 est toujours celui le plus proche du noeud testé.

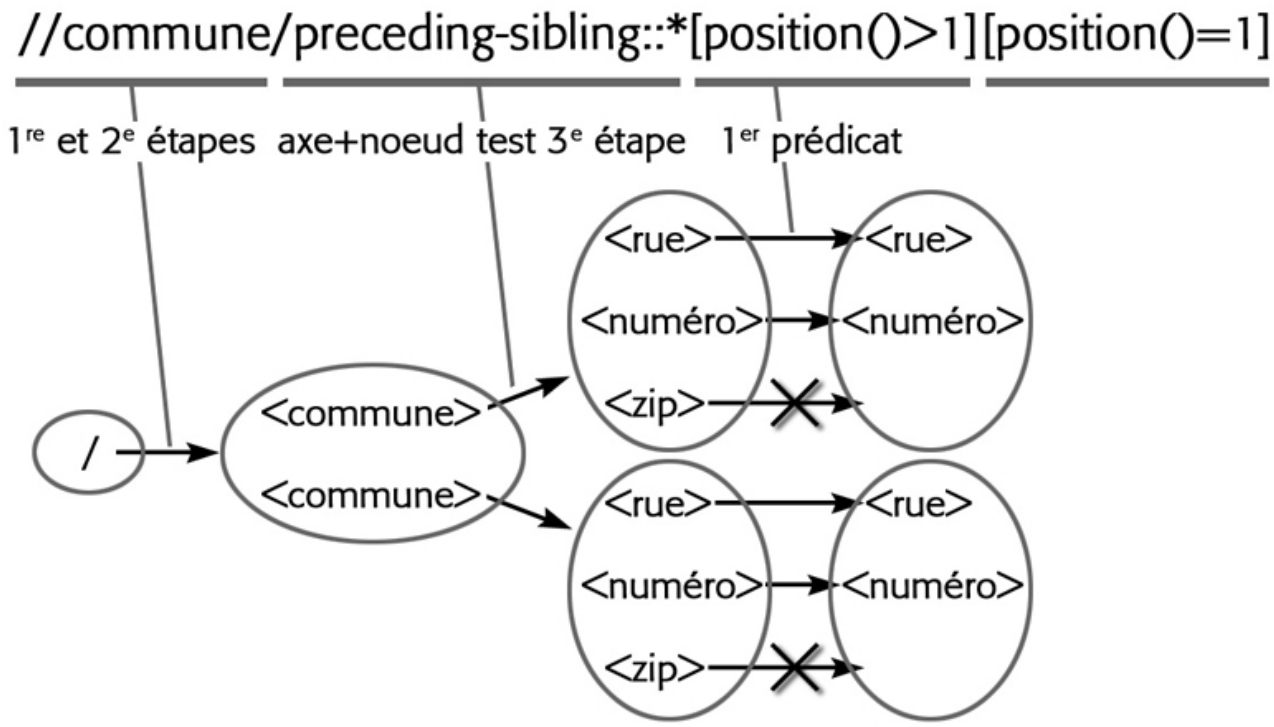
```
/carnet/personne[adresse/zip = '6041']/adresse/rue

/carnet/personne[position() > 2]/adresse/rue

/carnet/personne[position() >= last() - 1]/adresse/rue

following-sibling::*[position() <= 2]

preceding-sibling::*[position() <= 2]
```

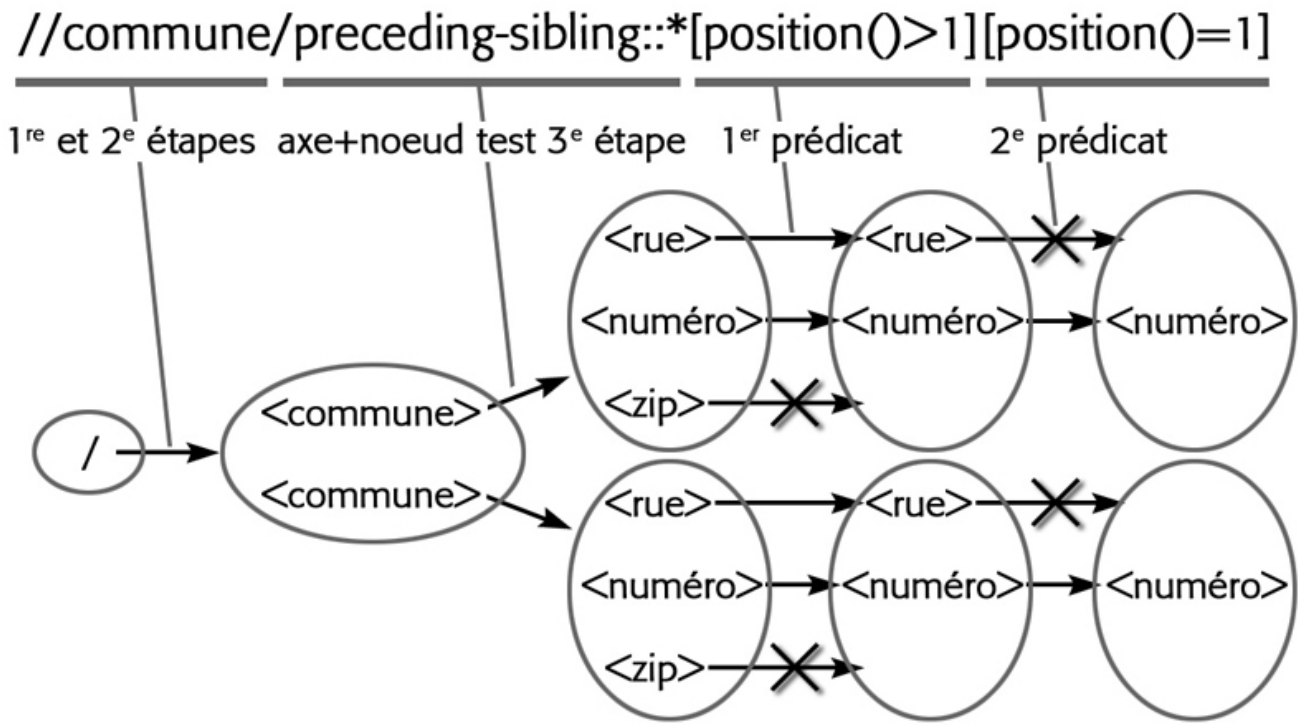
Méthode de calcul des prédicats

Cet exemple montre précisément comment se calculent les prédicats.









Les prédicats pour effectuer une jointure entre documents

Les prédicats permettent de réaliser une jointure entre deux documents XML ou au sein du même document.

```
fn:concat( ./zip, ': ', fn:doc("codesPostaux.xml")//city[@zip = current()/zip]/cityName )
```

```
/filmographie/réalisateurs/réalisateur[@id = current()/@réalisé-par]/nom
```

Simplifier la syntaxe avec les abréviations

- `.` est l'abréviation de `self::node()`

  Le `.` retourne l'élément contexte dans le contexte dynamique, tandis que la fonction `current()` (en XSLT) retourne celui du contexte statique

  Exemples: `.` `current()` `./zip` `current()/zip`

- `..` est l'abréviation de `parent::node()`

  Exemples: `..` `../zip`

- `//` est l'abréviation de `/descendant-or-self::node()/`

  Exemples: `//zip` `.//zip` `/carnet/personne//zip`

- `@` est l'abréviation de `attribute::`

  Exemples: `@titre` `@*` `/carnet/personne/@titre`

- Pour rappel, `child::` est l'axe de recherche par défaut

  Exemples: `carnet` `child::carnet` `node()` `child::node()`

Les fonctions les plus importantes

XPath contient un nombre important de fonctions. Parmi elles :

```
fn:string(<_low>seq</_low>)
fn:normalize-space(<_low>string</_low>)
fn:concat(<_low>val1</_low>, <_low>val2</_low>, <_low>val3</_low>...)
fn:string-length(<_low>string</_low>)
fn:upper-case(<_low>string</_low>)
fn:lower-case(<_low>string</_low>)
fn:string-join(<_low>seq1</_low>, <_low>sep</_low>)

fn:contains(<_low>string</_low>, <_low>substring</_low>)
fn:starts-with(<_low>string</_low>, <_low>substring</_low>)
fn:ends-with(<_low>string</_low>, <_low>substring</_low>)
fn:substring-before(<_low>string</_low>, <_low>substring</_low>)
fn:substring-after(<_low>string</_low>, <_low>substring</_low>)

fn:distinct-values(<_low>seq</_low>)

fn:true()
fn:false()

fn:name()
fn:name(<_low>seq</_low>)
fn:local-name()
fn:local-name(<_low>seq</_low>)
fn:root()
```

```
fn:position()
fn:last()

fn:doc(<_low>string</_low>)
fn:collection(<_low>string</_low>)

fn:exists(<_low>seq</_low>)
fn:count(<_low>seq</_low>)
```

Les commentaires

Une expression peut contenir des commentaires entourés de **(:** et **:)**

Les commentaires peuvent être imbriqués

```
//city[@zip = current()/zip]/cityName (: get the city's name :)
```

L'instruction for

Une expression **for** permet de faire varier une (ou plusieurs) variable(s) dans une séquence et de calculer un résultat pour chaque élément de cette séquence. Le résultat sera une séquence reprenant l'ensemble de ces résultats.

```
for $<_low>variable</_low> in <_low>expression</_low> return <_low>expression</_low>

for $<_low>variable1</_low> in <_low>expression1</_low>, $<_low>variable2</_low> in <_low>expression2</
_low> return <_low>expression</_low>
```

```
for $a in ("a", "b", "c") return concat($a,$a)

aa, bb, cc



for $a in ("a", "b", "c"),$b in ("1", "2", "3") return concat($a,$b)

a1, a2, a3, b1, b2, b3, c1, c2, c3



for $z in distinct-values(//zip) return concat($z,': ', string-join(//commune[../zip=$z],', '))

6761: Chenois, Chenois
1180: Uccle, Uccle, Uccle
6041: Gosselies, Gosselies, Gosselies
1040: Etterbeek
1050: Ixelles, Ixelles
```

L'instruction if then else

Une expression **if then else** permet de retourner l'une ou l'autre séquence en fonction d'une condition.

```
if (<_low>condition</_low>) then <_low>expression</_low> else <_low>expression</_low>
```

Le **then** et le **else** sont obligatoires.

```
for $p in //personne return concat (if ($p/@titre='M.') then "Monsieur" else "Madame",' ',$p/nom)
```

```
Monsieur Bernard
Madame Durand
Madame Gobbe
Monsieur Gobbe
Madame Bareel
Monsieur Albert
Monsieur Declerck
Madame Dumont
Madame Jansens
Monsieur Pireaux
Madame Da vinci
```

Les expressions quantifiées

Une **expression quantifié** permet de savoir si tous les éléments (**every**) ou si une partie des éléments (**some**) d'une séquence respectent une condition donnée.

```
every $<_low>variable</_low> in <_low>expression</_low> satisfies <_low>condition</_low>

every $<_low>variable1</_low> in <_low>expression1</_low>, $<_low>variable2</_low> in <_low>expression2</
_low> satisfies <_low>condition</_low>

some $<_low>variable</_low> in <_low>expression</_low> satisfies <_low>condition</_low>

some $<_low>variable1</_low> in <_low>expression1</_low>, $<_low>variable2</_low> in <_low>expression2</
_low> satisfies <_low>condition</_low>
```

```
if (every $a in ("a", "b", "c") satisfies $a="a") then "ok" else "not ok"
```

```
not ok
```

```
if (some $a in ("a", "b", "c") satisfies $a="a") then "ok" else "not ok"
```

```
ok
```

```
if (some $a in ("a", "b", "c"), $b in ("d", "a", "f") satisfies $a=$b) then "ok" else "not ok"
```

```
ok
```

Chapitre 3 - Le langage XSLT

Table des matières de ce chapitre

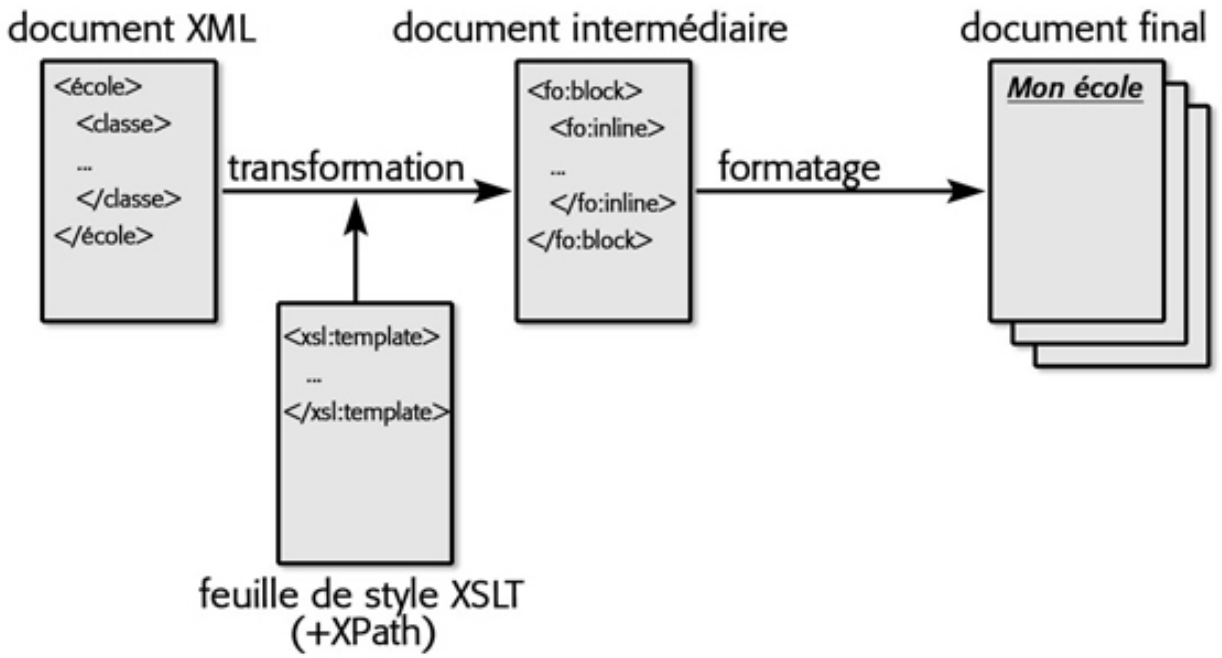
XSLT c'est quoi ?

XSLT est un langage destiné à effectuer une transformation d'un (ou plusieurs) document(s) XML. XSLT est l'acronyme de **Extensible Stylesheet Language for Transformation**.

Cette transformation est décrite dans un document appelé traditionellement une **feuille de styles**, mais qui en toute logique devrait s'appeler une **feuille de transformation**

La transformation se déroulera en deux phases :

- une phase de transformation
- une phase de formatage



Les langages utilisés dans une feuille de styles

Trois langages différents vont intervenir dans une feuille de styles :

- le langage de transformation **XSLT**
- le langage d'interrogation **XPath**
- un langage de sortie, ou de formatage, à choisir librement (**HTML**, **XHTML**, **XML**, **CSV**, **JSON**, **XSL**...)

La structure de la feuille de styles est définie par le langage XSLT. Celle-ci utilise des instructions faisant appel à des expressions XPath pour produire un document en sortie, respectant la structure du langage de formatage.

Structure générale d'une feuille de styles

L'élément racine doit s'appeller `<xsl:stylesheet>` dans l'espace de noms du langage. Il doit posséder un attribut `version` donnant la version du langage et contenir:

- Un élément `<xsl:output>` qui définit la méthode de sortie: `xml`, `html`, `xhtml`, `text`, `json` ou `adaptive`
- Un ou plusieurs éléments `<template>`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"/>

  <xsl:template match="/">
      ...
  </xsl:template>

  <xsl:template match="personne">
    ...
  </xsl:template>

</xsl:stylesheet>
```

Structure générale d'une feuille de styles

Les éléments `<xsl:template>` sont appelées des **règles modèles** (*template rules*). Elles possèdent un **motif** donné par l'attribut `pattern` et contiennent un **constructeur de séquence**, anciennement appelé **modèle**, entre la balise de début et la balise de fin.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" indent="yes"/>

  <xsl:template match="/">
     <html>
       <head>
         <title>Liste des personnes</title>
       </head>
       <body>
         <ul>
           <xsl:apply-templates select="/carnet/personne"/>
         </ul>
       </body>
     </html>
  </xsl:template>

  <xsl:template match="personne">
     <li>
       <xsl:value-of select="nom"/>
       <xsl:text> </xsl:text>
       <xsl:value-of select="prénom"/>
     </li>
  </xsl:template>

</xsl:stylesheet>
```

Résultat obtenu :

```
<html>
  <head>
      <title>Liste des personnes</title>
```

```
  </head>
  <body>
    <ul>
      <li>Bernard John</li>
      <li>Durand Elise</li>
      <li>Gobbe Nathalie</li>
      <li>Gobbe Jean-Luc</li>
      <li>Bareel Valérie</li>
      <li>Albert Marc</li>
      <li>Declerck Serge</li>
      <li>Dumont Julie</li>
      <li>Jansens Sophie</li>
      <li>Pireaux Eric</li>
      <li>Da vinci Martine</li>
    </ul>
  </body>
</html>
```

Le contexte d'exécution d'une feuille de styles

L'exécution d'une feuille de styles se fait dans le cadre d'un contexte donné. Au démarrage, le contexte initial sera:

- / comme élément contexte
- 1 comme taille de contexte
- 1 comme position de contexte

Lors de cette exécution, le contexte sera modifié par les instructions suivantes :

- `<xsl:apply-templates select="..."/>`
- `<xsl:for-each select="..."/>`

Par exemple, si on rencontre `<xsl:apply-templates select="/carnet/personne"/>`, deux contextes seront créés:

1re`<personne>` comme élément contexte	2e`<personne>` comme élément contexte
2 comme taille de contexte	2 comme taille de contexte
1 comme position de contexte	2 comme position de contexte

Choix de la règle modèle à exécuter

Pour chaque contexte, il n'y aura qu'un seul constructeur de séquence qui sera exécuté. Le choix se fera parmi toutes les règles modèles présentes dans la feuille de styles.
Ce choix sera déterminé grâce au motif: seuls les motifs qui permettent de sélectionner l'élément contexte seront déterminants. Des règles de priorités seront établies pour effectuer le choix

Si le contexte est le suivant :	Les motifs suivants seront déterminants:
	`<xsl:template match="personne">`

- 1re<personne> comme élément contexte
- 2 comme taille de contexte
- 1 comme position de contexte

```
  ...
</xsl:template>

<xsl:template match="*">
  ...
</xsl:template>

<xsl:template match="/carnet/personne[1]">
  ...
</xsl:template>
```

Par contre, ceux-ci ne seront pas déterminants:

```
<xsl:template match="/">
  ...
</xsl:template>

<xsl:template match="adresse">
  ...
</xsl:template>

<xsl:template match="/carnet/personne[2]">
  ...
</xsl:template>
```

Exécution d'un constructeur de séquence

L'exécution d'un constructeur de séquence est très simple :

- On recopie tout ce qui ne fait pas partie de l'espace de noms du langage
- On exécute les instructions mentionnées dans l'espace de noms du langage

```
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <data>
      <genDate>Date: <xsl:value-of select="current-dateTime()"/></genDate>
      <xsl:for-each select="/carnet/personne">
        <item>
          <xsl:value-of select="nom"/>
          <xsl:text> </xsl:text>
          <xsl:value-of select="prénom"/>
        </item>
      </xsl:for-each>
    </data>
  </xsl:template>

</xsl:stylesheet>
```

Ce qui donne :

```
<data>
  <genDate>Date: 2017-06-12T16:29:48+02:00</genDate>
  <item>Bernard John</item>
  <item>Durand Elise</item>
```

```
  <item>Gobbe Nathalie</item>
  <item>Gobbe Jean-Luc</item>
  <item>Bareel Valérie</item>
  <item>Albert Marc</item>
  <item>Declerck Serge</item>
  <item>Dumont Julie</item>
  <item>Jansens Sophie</item>
  <item>Pireaux Eric</item>
  <item>Da vinci Martine</item>
</data>
```

Les instructions principales

Les instructions principales, qui peuvent prendre place dans un constructeur de séquence, sont:

`<xsl:value-of select="..."/>`	Calcule l'expression XPath donnée par l'attribut `select` et écrit en sortie le résultat sous la forme d'une chaîne de caractères
`<xsl:text>...</xsl:text>`	Recopie en sortie la chaîne de caractères donnée par l'instruction
`<xsl:apply-templates select="..."/>`	Calcule une séquence donnée par l'attribut `select`. Crée des contextes pour chaque élément de cette séquence et exécute la règle modèle la plus appropriée pour ces contextes

`<xsl:for-each select="...">`
`  ...`
`</xsl:for-each>`
	Calcule une séquence donnée par l'attribut `select`. Crée des contextes pour chaque élément de cette séquence et exécute le constructeur de séquence contenu dans l'instruction

`<xsl:sort select="..."/>`	Placée dans un `<xsl:apply-templates>` ou un `<xsl:for-each>`, cette instruction permet de trier les éléments dans la séquence grâce à la valeur donnée par l'attribut `select`

`<xsl:if test="...">`
`  ...`
`</xsl:if>`
	Évalue une condition. Si elle vaut `true`, exécute le constructeur de séquence contenu dans l'instruction

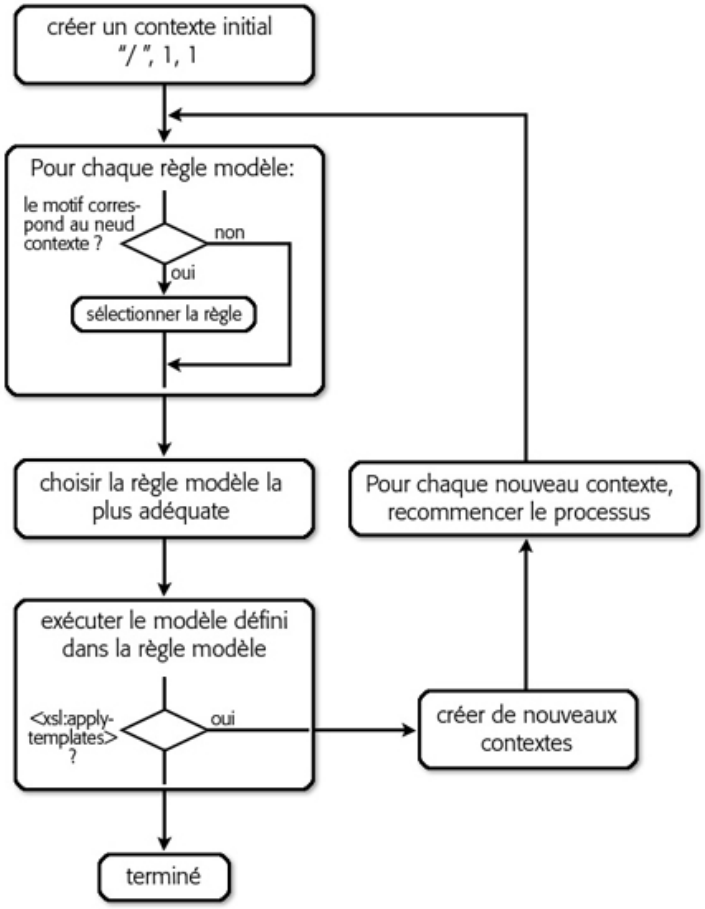
`<xsl:choose>`
`  <xsl:when test="...">...</xsl:when>`
`  <xsl:when test="...">...</xsl:when>`
`  <xsl:otherwise>...</xsl:otherwise>`
`</xsl:choose>`
	Évalue les conditions les unes après les autres. La première qui donne `true` détermine le constructeur de séquence à exécuter. Si aucune ne donne `true`, exécute le constructeur de séquence du `<xsl:otherwise>` éventuel

`<xsl:copy>`
`  ...`
`</xsl:copy>`
	Copie le noeud courant, sans ses attributs et sans son contenu. Ajoute à ce noeud le résultat de l'exécution du constructeur de séquence contenu dans l'instruction

`<xsl:copy-of select="..."/>`	Calcule une séquence donnée par l'attribut `select` et genère en sortie une copie de tous les éléments de cette séquence
`<xsl:variable name="..." select="..."/>`	Crée un variable et lui donne la valeur calculée par l'attribut `select`

Fonctionnement de la feuille de styles

Le processus itératif est toujours le même. Sélectionner un contexte, chercher la règle modèle appropriée et exécuter le modèle qu'elle contient. Si cette règle modèle contient une instruction `<xsl:apply-templates>`, celle-ci crée de nouveaux contextes et le processus recommence.



Exemple n°1 de transformation XML

Cet exemple remplace le contenu de l'élément `<adresse>` par une chaine de caractères :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

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

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

```
  <xsl:template match="@*">
    <xsl:copy/>
  </xsl:template>

  <xsl:template match="adresse">
    <xsl:copy>
      <xsl:value-of select="concat(numéro,' ',rue,', ',zip,' ',commune)"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

document XML obtenu après transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<carnet>
  <personne titre="M.">
    <nom>Bernard</nom>
    <prénom>John</prénom>
    <adresse>2 Rue du canal, 6761 Chenois</adresse>
  </personne>
  <personne titre="Mme">
    <nom>Durand</nom>
    <prénom>Elise</prénom>
    <adresse>23 Avenue Churchill, 1180 Uccle</adresse>
  </personne>
  <personne titre="Mme">
    <nom>Gobbe</nom>
    <prénom>Nathalie</prénom>
    <adresse>42 Rue de la gare, 6041 Gosselies</adresse>
  </personne>
  <personne titre="M.">
    <nom>Gobbe</nom>
    <prénom>Jean-Luc</prénom>
    <adresse>42 Rue de la gare, 1040 Etterbeek</adresse>
  </personne>
  <personne titre="Mme">
    <nom>Bareel</nom>
    <prénom>Valérie</prénom>
    <adresse>23 Avenue des pinsons, 1180 Uccle</adresse>
  </personne>
  <personne titre="M.">
    <nom>Albert</nom>
    <prénom>Marc</prénom>
    <adresse>122 Rue longue, 1050 Ixelles</adresse>
  </personne>
  <personne titre="M.">
    <nom>Declerck</nom>
    <prénom>Serge</prénom>
    <adresse>134 Rue basse, 1050 Ixelles</adresse>
  </personne>
  <personne titre="Mlle">
    <nom>Dumont</nom>
    <prénom>Julie</prénom>
    <adresse>6 Boulevard du triomphe, 6761 Chenois</adresse>
  </personne>
  <personne titre="Mlle">
    <nom>Jansens</nom>
    <prénom>Sophie</prénom>
    <adresse>6 Avenue F.D. Roosevelt, 1180 Uccle</adresse>
```

```
  </personne>
  <personne titre="M.">
    <nom>Pireaux</nom>
    <prénom>Eric</prénom>
    <adresse>432 Rue Jules Blase, 6041 Gosselies</adresse>
  </personne>
  <personne titre="Mme">
    <nom>Da vinci</nom>
    <prénom>Martine</prénom>
    <adresse>32 Rue Ferdinand Bracke, 6041 Gosselies</adresse>
  </personne>
</carnet>
```

Exemple n°2 de transformation XML

L'exemple précédent peut être réécrit de manière plus compacte, en utilisant l'opérateur | d'union de deux séquences :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

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

  <xsl:template match="adresse">
    <xsl:copy>
      <xsl:value-of select="concat(numéro,' ',rue,', ',zip,' ',commune)"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Exemple n°3 de transformation XML

Dans cet exemple, le document est totalement transformé :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <customers>
      <xsl:for-each select="carnet/personne">
        <customer>
          <xsl:apply-templates select="nom|adresse"/>
        </customer>
      </xsl:for-each>
    </customers>
  </xsl:template>

  <xsl:template match="nom">
    <name>
```

```
      <xsl:value-of select="../@titre"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="upper-case(.)"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="upper-case(following-sibling::prénom)"/>
    </name>
  </xsl:template>

  <xsl:template match="adresse">
    <address>
      <xsl:value-of select="concat(numéro,' ',rue,', ',zip,' ',commune)"/>
    </address>
  </xsl:template>

</xsl:stylesheet>
```

document XML obtenu après transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
  <customer>
    <name>M. BERNARD JOHN</name>
    <address>2 Rue du canal, 6761 Chenois</address>
  </customer>
  <customer>
    <name>Mme DURAND ELISE</name>
    <address>23 Avenue Churchill, 1180 Uccle</address>
  </customer>
  <customer>
    <name>Mme GOBBE NATHALIE</name>
    <address>42 Rue de la gare, 6041 Gosselies</address>
  </customer>
  <customer>
    <name>M. GOBBE JEAN-LUC</name>
    <address>42 Rue de la gare, 1040 Etterbeek</address>
  </customer>
  <customer>
    <name>Mme BAREEL VALÉRIE</name>
    <address>23 Avenue des pinsons, 1180 Uccle</address>
  </customer>
  <customer>
    <name>M. ALBERT MARC</name>
    <address>122 Rue longue, 1050 Ixelles</address>
  </customer>
  <customer>
    <name>M. DECLERCK SERGE</name>
    <address>134 Rue basse, 1050 Ixelles</address>
  </customer>
  <customer>
    <name>Mlle DUMONT JULIE</name>
    <address>6 Boulevard du triomphe, 6761 Chenois</address>
  </customer>
  <customer>
    <name>Mlle JANSENS SOPHIE</name>
    <address>6 Avenue F.D. Roosevelt, 1180 Uccle</address>
  </customer>
  <customer>
    <name>M. PIREAUX ERIC</name>
    <address>432 Rue Jules Blase, 6041 Gosselies</address>
  </customer>
```

```
  <customer>
    <name>Mme DA VINCI MARTINE</name>
    <address>32 Rue Ferdinand Bracke, 6041 Gosselies</address>
  </customer>
</customers>
```

Exemple n°4 de transformation XML

Cet exemple regroupe les clients par codes postaux :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <customers>
      <xsl:iterate select="distinct-values(//zip)">
        <area zip="{.}">
          <xsl:apply-templates select="$root/carnet/personne[adresse/zip = current()]"/>
        </area>
      </xsl:iterate>
    </customers>
  </xsl:template>

  <xsl:template match="personne">
    <customer>
      <name>
        <xsl:value-of select="nom"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="prénom"/>
      </name>
      <address>
        <xsl:for-each select="adresse">
          <xsl:value-of select="concat(numéro,' ',rue,', ',zip,' ',commune)"/>
        </xsl:for-each>
      </address>
    </customer>
  </xsl:template>

</xsl:stylesheet>
```

Une autre façon d'obtenir le même résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/">
    <customers>
      <xsl:for-each-group select="carnet/personne" group-by="adresse/zip">
        <area zip="{current-grouping-key()}">
          <xsl:apply-templates select="current-group()"/>
        </area>
      </xsl:for-each-group>
    </customers>
  </xsl:template>
```

```
  <xsl:template match="personne">
    <customer>
      <name>
        <xsl:value-of select="nom"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="prénom"/>
      </name>
      <address>
        <xsl:for-each select="adresse">
          <xsl:value-of select="concat(numéro,' ',rue,', ',zip,' ',commune)"/>
        </xsl:for-each>
      </address>
    </customer>
  </xsl:template>

</xsl:stylesheet>
```

document XML obtenu après transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
  <area zip="6761">
    <customer>
      <name>Bernard John</name>
      <address>2 Rue du canal, 6761 Chenois</address>
    </customer>
    <customer>
      <name>Dumont Julie</name>
      <address>6 Boulevard du triomphe, 6761 Chenois</address>
    </customer>
  </area>
  <area zip="1180">
    <customer>
      <name>Durand Elise</name>
      <address>23 Avenue Churchill, 1180 Uccle</address>
    </customer>
    <customer>
      <name>Bareel Valérie</name>
      <address>23 Avenue des pinsons, 1180 Uccle</address>
    </customer>
    <customer>
      <name>Jansens Sophie</name>
      <address>6 Avenue F.D. Roosevelt, 1180 Uccle</address>
    </customer>
  </area>
  <area zip="6041">
    <customer>
      <name>Gobbe Nathalie</name>
      <address>42 Rue de la gare, 6041 Gosselies</address>
    </customer>
    <customer>
      <name>Pireaux Eric</name>
      <address>432 Rue Jules Blase, 6041 Gosselies</address>
    </customer>
    <customer>
      <name>Da vinci Martine</name>
      <address>32 Rue Ferdinand Bracke, 6041 Gosselies</address>
    </customer>
  </area>
  <area zip="1040">
```

```
    <customer>
      <name>Gobbe Jean-Luc</name>
      <address>42 Rue de la gare, 1040 Etterbeek</address>
    </customer>
  </area>
  <area zip="1050">
    <customer>
      <name>Albert Marc</name>
      <address>122 Rue longue, 1050 Ixelles</address>
    </customer>
    <customer>
      <name>Declerck Serge</name>
      <address>134 Rue basse, 1050 Ixelles</address>
    </customer>
  </area>
</customers>
```

Exemple n°5 de transformation XML

Cet exemple découpe le document en plusieurs documents (`carnet-part1.xml`, `carnet-part2.xml`, `carnet-part3.xml`...) contenant chacun un nombre donné de personnes (3 dans cet exemple), en utilisant les fonctionnalités de streaming permettant de traiter des documents très volumineux. :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/" name="xsl:initial-template">
    <xsl:source-document href="carnet.xml" streamable="yes">
      <xsl:for-each-group select="/carnet/personne" group-starting-with="*[position() mod 3 eq 1]">
        <xsl:result-document href="carnet-part{position()}.xml">
          <carnet>
            <xsl:copy-of select="current-group()"/>
          </carnet>
        </xsl:result-document>
      </xsl:for-each-group>
    </xsl:source-document>
  </xsl:template>

</xsl:stylesheet>
```

document XML obtenu après transformation

Contenu du document carnet-part1.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<carnet>
  <personne titre="M.">
    <nom>Bernard</nom>
    <prénom>John</prénom>
    <adresse>
      <rue>Rue du canal</rue>
      <numéro>2</numéro>
      <zip>6761</zip>
      <commune>Chenois</commune>
    </adresse>
  </personne>
```

```
  <personne titre="Mme">
    <nom>Durand</nom>
    <prénom>Elise</prénom>
    <adresse>
      <rue>Avenue Churchill</rue>
      <numéro>23</numéro>
      <zip>1180</zip>
      <commune>Uccle</commune>
    </adresse>
  </personne>
  <personne titre="Mme">
    <nom>Gobbe</nom>
    <prénom>Nathalie</prénom>
    <adresse>
      <rue>Rue de la gare</rue>
      <numéro>42</numéro>
      <zip>6041</zip>
      <commune>Gosselies</commune>
    </adresse>
  </personne>
</carnet>
```

Contenu du document carnet-part2.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<carnet>
  <personne titre="M.">
    <nom>Gobbe</nom>
    <prénom>Jean-Luc</prénom>
    <adresse>
      <rue>Rue de la gare</rue>
      <numéro>42</numéro>
      <zip>1040</zip>
      <commune>Etterbeek</commune>
    </adresse>
  </personne>
  <personne titre="Mme">
    <nom>Bareel</nom>
    <prénom>Valérie</prénom>
    <adresse>
      <rue>Avenue des pinsons</rue>
      <numéro>23</numéro>
      <zip>1180</zip>
      <commune>Uccle</commune>
    </adresse>
  </personne>
  <personne titre="M.">
    <nom>Albert</nom>
    <prénom>Marc</prénom>
    <adresse>
      <rue>Rue longue</rue>
      <numéro>122</numéro>
      <zip>1050</zip>
      <commune>Ixelles</commune>
    </adresse>
  </personne>
</carnet>
```

Et ainsi de suite pour les autres documents.

Exemple de transformation en texte (format JSON)

Cet exemple génère un contenu formaté en JSON :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet exclude-result-prefixes="fn" version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fn="http://www.w3.org/2005/xpath-functions">

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>[&#x0A;</xsl:text>
    <xsl:apply-templates select="//personne"/>
    <xsl:text>]&#x0A;</xsl:text>
  </xsl:template>

  <xsl:template match="personne">
    <xsl:text>   {&#x0A;</xsl:text>
    <xsl:text>   "nom": "</xsl:text>
    <xsl:value-of select="nom"/>
    <xsl:text>",&#x0A;</xsl:text>
    <xsl:text>   "prenom": "</xsl:text>
    <xsl:value-of select="prénom"/>
    <xsl:text>",&#x0A;</xsl:text>
    <xsl:text>   "adresse": "</xsl:text>
    <xsl:for-each select="adresse">
      <xsl:value-of select="concat(numéro,' ',rue,', ',zip,' ',commune)"/>
    </xsl:for-each>
    <xsl:text>"&#x0A;</xsl:text>
    <xsl:text>   }</xsl:text>
    <xsl:if test="position()!=last()">,</xsl:if>
    <xsl:text>&#x0A;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

document texte obtenu après transformation

```
[
  {
  "nom": "Bernard",
  "prenom": "John",
  "adresse": "2 Rue du canal, 6761 Chenois"
  },
  {
  "nom": "Durand",
  "prenom": "Elise",
  "adresse": "23 Avenue Churchill, 1180 Uccle"
  },
  {
  "nom": "Gobbe",
  "prenom": "Nathalie",
  "adresse": "42 Rue de la gare, 6041 Gosselies"
  },
  {
  "nom": "Gobbe",
  "prenom": "Jean-Luc",
  "adresse": "42 Rue de la gare, 1040 Etterbeek"
  },
```

```
  {
  "nom": "Bareel",
  "prenom": "ValÃ©rie",
  "adresse": "23 Avenue des pinsons, 1180 Uccle"
  },
  {
  "nom": "Albert",
  "prenom": "Marc",
  "adresse": "122 Rue longue, 1050 Ixelles"
  },
  {
  "nom": "Declerck",
  "prenom": "Serge",
  "adresse": "134 Rue basse, 1050 Ixelles"
  },
  {
  "nom": "Dumont",
  "prenom": "Julie",
  "adresse": "6 Boulevard du triomphe, 6761 Chenois"
  },
  {
  "nom": "Jansens",
  "prenom": "Sophie",
  "adresse": "6 Avenue F.D. Roosevelt, 1180 Uccle"
  },
  {
  "nom": "Pireaux",
  "prenom": "Eric",
  "adresse": "432 Rue Jules Blase, 6041 Gosselies"
  },
  {
  "nom": "Da vinci",
  "prenom": "Martine",
  "adresse": "32 Rue Ferdinand Bracke, 6041 Gosselies"
  }
]
```

Chapitre 4 - Le langage XQuery

Table des matières de ce chapitre

XQuery c'est quoi ?

La norme **XQuery** permet d'écrire des **requêtes** pour accèder à de l'information stockée dans une **base de données XML**.

Une base de données XML est une notion que le W3C a volontairement gardée la plus vague possible: n'importe quelle structure qui s'apparente à du XML peut être considérée comme une **base de données XML**.

- Certaines implémentations vont utiliser un véritable moteur de base de données, avec toutes les fonctionnalités requises pour ce type d'engin (rapidité, robustesse, efficacité, intégrité, backups, droits d'accès...)
- Certaines implémentations vont stocker l'information dans de simples documents XML placés dans des répertoires
- Certaines implémentations vont utiliser des bases de données relationnelles classiques dont les tables seront présentées sous la forme de structures XML

Les bases de XQuery

XQuery est une extension du langage XPath qui se distingue essentiellement par trois points :

- la partie déclarative d'une requête, ou **prologue**, qui rend la requête autonome
- les **constructeurs** d'éléments, d'attributs...
- les instructions **FLWOR** (qui se prononce **flower**)

```
xquery version "1.0";
declare variable $personnes := /carnet/personne;

for $z in distinct-values($personnes//zip)
order by $z
return <codePostal> { $z } </codePostal>
```

Les deux syntaxes de XQuery

Les **requêtes XQuery** peuvent être écrites de deux manières différentes :

- soit en utilisant un **langage naturel non-XML**, qui facilite la compréhension
- soit en utilisant un **langage construit en XML**, qui facilite la manipulation informatique

On peut, bien entendu, passer d'un langage à l'autre (les requêtes sont les mêmes, elles retournent un résultat identique).

Le langage naturel d'XQuery (non-XML)

Exemple de **requête** qui utilise le **langage naturel de XQuery** (non-XML) :

```
<personnes>
```

```
  {
  for $p in doc("carnet/carnet.xml")/carnet/personne
  return <personne>{ $p/nom }</personne>
  }
</personnes>
```

La syntaxe XML de XQuery (XQueryX)

Même requête que l'exemple précédent, mais qui utilise la **syntaxe XML de XQuery** :

```
<?xml version="1.0" encoding="UTF-8"?>
<xqx:module xsi:schemaLocation="http://www.w3.org/2005/XQueryX http://www.w3.org/2005/XQueryX/xqueryx.xsd"
xmlns:xqx="http://www.w3.org/2005/XQueryX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xqx:mainModule>
    <xqx:queryBody>
      <xqx:elementConstructor>
        <xqx:tagName>personnes</xqx:tagName>
        <xqx:elementContent>
          <xqx:flworExpr>
            <xqx:forClause>
              <xqx:forClauseItem>
                <xqx:typedVariableBinding>
                  <xqx:varName>p</xqx:varName>
                </xqx:typedVariableBinding>
                <xqx:forExpr>
                  <xqx:pathExpr>
                    <xqx:stepExpr>
                      <xqx:filterExpr>
                        <xqx:functionCallExpr>
                          <xqx:functionName>doc</xqx:functionName>
                          <xqx:arguments>
                            <xqx:stringConstantExpr>
                              <xqx:value>carnet/carnet.xml</xqx:value>
                            </xqx:stringConstantExpr>
                          </xqx:arguments>
                        </xqx:functionCallExpr>
                      </xqx:filterExpr>
                    </xqx:stepExpr>
                    <xqx:stepExpr>
                      <xqx:xpathAxis>child</xqx:xpathAxis>
                      <xqx:nameTest>carnet</xqx:nameTest>
                    </xqx:stepExpr>
                    <xqx:stepExpr>
                      <xqx:xpathAxis>child</xqx:xpathAxis>
                      <xqx:nameTest>personne</xqx:nameTest>
                    </xqx:stepExpr>

                  </xqx:pathExpr>
                </xqx:forExpr>
              </xqx:forClauseItem>
            </xqx:forClause>
            <xqx:returnClause>
              <xqx:elementConstructor>
                <xqx:tagName>personne</xqx:tagName>
                <xqx:elementContent>
                  <xqx:pathExpr>
                    <xqx:stepExpr>
                      <xqx:filterExpr>
                        <xqx:varRef>
                          <xqx:name>p</xqx:name>
```

```
                    </xqx:varRef>
                  </xqx:filterExpr>
                </xqx:stepExpr>
                <xqx:stepExpr>
                  <xqx:xpathAxis>child</xqx:xpathAxis>
                  <xqx:nameTest>nom</xqx:nameTest>
                </xqx:stepExpr>
              </xqx:pathExpr>
            </xqx:elementContent>
          </xqx:elementConstructor>
        </xqx:returnClause>
      </xqx:flworExpr>
    </xqx:elementContent>
  </xqx:elementConstructor>
 </xqx:queryBody>
</xqx:mainModule>
</xqx:module>
```

Les commentaires

Une expression XQuery peut contenir des commentaires entourés de **(:** et **:)**

Les commentaires peuvent être imbriqués

```
<personnes>
  {
  (: boucle sur toutes les personnes de la DB :)

  for $p in doc("carnet/carnet.xml")/carnet/personne
  return <personne>{ $p/nom/text() }</personne> (: retourne un élément "personne" :)
  }
</personnes>
```

Les modules XQuery

Une requête XQuery est construite en un ou plusieurs **modules**. Chaque requête contient un **module principal** qui peut, éventuellement, faire appel à des **modules de librairie**.

Le rôle des modules de librairie est de définir des variables ou des fonctions utilisables par le module principal.

Un **module principal** est constitué:

- d'une déclaration éventuelle du numéro de version du langage et du type d'encodage
- d'un **prologue** (importation de modules, déclarations d'espaces de noms, de variables, de fonctions...)
- d'un **corps de requête** (**query body** en anglais), qui est une **expression** destinée à fournir un résultat

Un **module de librairie** est constitué:

- d'une déclaration éventuelle du numéro de version du langage et du type d'encodage
- d'une déclaration de module (qui définit l'espace de noms de toutes les variables et fonctions déclarées dans le module)
- d'un **prologue** (déclarations des variables et des fonctions qui seront utilisées par d'autres modules)

Exemple d'un module principal

Ce qui suit est un exemple d'une requête uniquement constituée d'un module principal contenant:

- une déclaration de numéro de version et de type d'encodage
- suivie d'un prologue
- suivi d'un corps de requête (= **expression** à exécuter pour fournir un résultat)

```
xquery version "1.0" encoding "iso-8859-1";

declare namespace cli = "http://www.samples.com/clients";
declare variable $type := "société";

<cli:clients>
  {
  for $client in doc("commandes/clients.xml")/cli:clients/cli:client
  where $client/@type = $type
  return <cli:client id="{$client/@id}">
    {
    <cli:nom>{ $client/cli:nom/text() }</cli:nom>
    }
  </cli:client>
  }
</cli:clients>
```

Exemple d'un module de librairie

Ce qui suit est un exemple de **module de librairie** avec:

- une déclaration de numéro de version et de type d'encodage;
- suivie d'un déclaration de module;
- suivie d'un prologue (ne contenant qu'une déclaration de variable).

Module de librairie:

```
xquery version "1.0" encoding "iso-8859-1";
module namespace cmd = "http://www.samples.com/commandes";

declare variable $cmd:type := "société";
```

Ce module peut être importé dans le **module principal** d'une requête XQuery, comme dans l'exemple suivant:

```
xquery version "1.0" encoding "iso-8859-1";
import module namespace cmd = "http://www.samples.com/commandes";
declare namespace cli = "http://www.samples.com/clients";

<cli:clients>
  {
  for $client in doc("commandes/clients.xml")/cli:clients/cli:client
  where $client/@type = $cmd:type
  return <cli:client id="{$client/@id}">
    {
```

```
    <cli:nom>{ $client/cli:nom/text() }</cli:nom>
    }
  </cli:client>
  }
</cli:clients>
```

Prologue - déclaration du numéro de version et du type d'encodage

Le numéro de version de XQuery, ainsi que le type d'encodage, peuvent être déclarés de la manière suivante :

```
xquery version "<_low>version</_low>" encoding "<_low>encoding</_low>";
```

```
xquery version "1.0";

xquery version "1.0" encoding "iso-8859-1";

xquery version "3.0" encoding "utf-8";

xquery encoding "utf-16";
```

Par défaut, la version utilisée est **3.1** (la dernière) et le type d'encodage est spécifique à l'application utilisée.

Prologue - déclaration d'un espace de noms

Un espace de noms utilisé dans une requête doit être déclaré de la manière suivante :

```
declare namespace <_low>prefix</_low> = "<_low>namespace_uri</_low>";
```

Une série d'espace de noms sont déclarés par défaut :

```
declare namespace xml   = "http://www.w3.org/XML/1998/namespace";
declare namespace xs    = "http://www.w3.org/2001/XMLSchema";
declare namespace xsi   = "http://www.w3.org/2001/XMLSchema-instance";
declare namespace fn    = "http://www.w3.org/2005/xpath-functions";
declare namespace local = "http://www.w3.org/2005/xquery-local-functions";
```

Exemple de requête qui utilise des espace de noms :

```
declare namespace cli = "http://www.samples.com/clients";
declare namespace pr  = "http://www.samples.com/produits";
declare namespace cmd = "http://www.samples.com/commandes";

<cmd:clients>
  {
  for $cl in /cli:clients/cli:client
  return <cmd:client id="{$cl/@id}">
    {
    for $cmd in /cmd:commandes/cmd:commande[cmd:client/@ref = $cl/@id]/cmd:produit,
    $pro in /pr:catalogue/pr:produit
    where $pro/@id = $cmd/@ref
```

```
    return <cmd:produit quantité="{$cmd/@quantité}" id="{$cmd/@ref}">{ $pro/pr:nom/text() }</cmd:produit>
    }
  </cmd:client>
  }
</cmd:clients>
```

Prologue - déclaration d'une variable

Une variable peut être déclarée d'une des manières suivantes :

```
declare variable $<_low>nom</_low> := <_low>expression</_low>;

declare variable $<_low>nom</_low> as <_low>type</_low> := <_low>expression</_low>;

declare variable $<_low>nom</_low> external;

declare variable $<_low>nom</_low> as <_low>type</_low> external;
```

Exemples:

```
declare variable $x := 7.5;
declare variable $y as xs:decimal := 7.5;



declare namespace cli = "http://www.samples.com/clients";

declare variable $cli:x := 7.5;
declare variable $cli:y as xs:decimal := 7.5;



declare variable $x external;
declare variable $y as xs:decimal external;
```

Prologue - déclaration d'une fonction

Une fonction doit être déclarée dans un espace de noms de la manière suivante (on utilise ici l'espace de noms **local** qui est pré-défini dans le langage) :

```
declare function <_low>nom_de_la_fonction</_low>(<_low>par1</_low>, <_low>par2</_low>, <_low>par3</_low>...)
   {
   <_low>expression</_low>
   };
```

```
declare function local:addition($a, $b)
   {
   $a + $b
   };

<result> { local:addition(5, 9) } </result>
```

La fonction et les paramètres peuvent être typés :

```
declare function local:addition($a as xs:integer, $b as xs:integer) as xs:integer
   {
   $a + $b
   };


declare function local:getZipList($root as element()) as xs:string+
   {
   for $z in distinct-values($root//zip) order by $z return $z
   };

for $zip in local:getZipList(/carnet)
return <zip>{$zip}</zip>
```

Les constructeurs

Un **constructeur** est une instruction XQuery qui permet de générer un noeud XML quelconque (noeud élément, noeud attribut, noeud commentaire...)

Il existe deux types de constructeurs :

- les **constructeurs directs** permettent de créer un noeud en donnant un exemple en XML :

```
<personne titre="M.">
  <nom>Declerck</nom>
  <prénom>Christophe</prénom>
</personne>
```

- les **constructeurs calculés** permettent de créer un noeud à l'aide d'une instruction :

```
element personne {
    attribute titre { "M." },
    element nom { "Declerck" },
    element prénom { "Christophe" }
    }
```

Les constructeurs directs

Au sein d'un constructeur direct, on peut utiliser des accolades pour insérer le résultat d'une autre expression XQuery :

```
for $z in distinct-values(//zip) where number($z)>=6000 and number($z)<=6999
return <area zip="{$z}">
          {
          count(//personne[.//zip = $z])," ",<!-- nbr de personnes -->
          }
       </area>
```

Ce qui donne:

```
<area zip="6761">2 <!-- nbr de personnes --></area>
<area zip="6041">3 <!-- nbr de personnes --></area>
```

remarque: pour générer des accolades dans la réponse, on écrira **{{** et **}}**

Les constructeurs calculés

Les constructeurs calculés ont l'avantage de pouvoir construire le nom de l'élément ou de l'attribut par une expression :

```
for $z in distinct-values(//zip) where number($z)>=6000 and number($z)<=6999
return element { concat('area', $z) }
  {
  count(//personne[.//zip = $z])
  }
```

Ce qui donne:

```
<area6761>2</area6761>
<area6041>3</area6041>
```

L'instruction FLWOR

Une des plus importante instruction de XQuery est l'instruction **FLWOR** (prononcez "**flower**"). Cette instruction étend les possibilités de l'instruction `for` de XPath. Cette instruction est l'équivalent du **SELECT** en **SQL**.

FLWOR est l'acronyme de **for**, **let**, **where**, **order by** et **return**.

```
for $p in doc("carnet/carnet.xml")/carnet/personne
let $nom:=fn:concat($p/nom/text(), ' ', $p/prenom/text())
where $p/adresse/zip='6041'
order by $nom
return <personne> { $nom } </personne>
```

L'instruction FLWOR

L'instruction **FLWOR** doit :

- commencer par une clause **for** ou une clause **let** obligatoire
- contenir un nombre quelconque de clauses **for**, **let**, **where**, **group by**, **order by** ou **count**
- se terminer par une clause **return** obligatoire

Dans la première version de XQuery, les clauses **for** et **let** devaient commencer l'instruction. Elles pouvaient éventuellement être suivies d'un seul **where** et/ou un seul **order by**

```
for $p in /carnet/personne
return <personne> { concat($p/nom, " ", $p/prénom) } </personne>
```

```
let $p := /carnet/personne/concat(nom,' ',prénom)
return <personnes> { string-join($p, ', ') } </personnes>
```

```
for $p in /carnet/personne
let $nom:=fn:concat($p/nom, ' ', $p/prénom)
```

```
return <personne> { $nom } </personne>


for $z in distinct-values(//zip)
order by $z
for $p in /carnet/personne
where $p/adresse/zip = $z
let $nom:=fn:concat($p/nom, ' ', $p/prénom, ' (',$z,')')
return <personne> { $nom } </personne>


for $p in /carnet/personne
group by $z := $p//zip
return <habitants zip="{$z}"> { count($p) } </habitants>
```

L'instruction FLWOR - la clause for

Une clause **for** fait varier une variable, appelée **variable liée** (*bound variable*), parmi les éléments d'une séquence. La clause **return** sera exécutée pour chaque élément de la séquence (la variable liée contiendra l'élément en question).

```
for $p in /carnet/personne
return <personne> { concat($p/nom, " ", $p/prénom) } </personne>


<personne>Bernard John</personne>
<personne>Durand Elise</personne>
<personne>Gobbe Nathalie</personne>
<personne>Gobbe Jean-Luc</personne>
<personne>Bareel Valérie</personne>
<personne>Albert Marc</personne>
<personne>Declerck Serge</personne>
<personne>Dumont Julie</personne>
<personne>Jansens Sophie</personne>
<personne>Pireaux Eric</personne>
<personne>Da vinci Martine</personne>
```

L'instruction FLWOR - la clause let

Un clause **let** donne à une variable liée une seule valeur, qui bien entendu peut être une séquence. La clause **return** ne sera exécutée qu'une seule fois.

```
let $p := /carnet/personne[adresse/zip='6041']/concat(nom,' ',prénom)
return <personnes> { string-join($p, ', ') } </personnes>


<personnes>Gobbe Nathalie, Pireaux Eric, Da vinci Martine</personnes>
```

L'instruction FLWOR - combiner plusieurs for

Deux **for** combinés vont créer un jeu de paires de valeurs (chaque valeur de la première variable liée sera combinée avec chaque valeur de la deuxième variable liée). La clause **return** sera exécutée pour chacune de ces paires de valeurs.

De manière générale, on va parler de **tuples de valeurs** pour désigner les différentes combinaisons de valeurs des variables liées. Dans cet exemple, les tuples seront `($a1=1, $a2=a)`, `($a1=1, $a2=b)`, `($a1=1, $a2=c)`, `($a1=2, $a2=a)`, `($a1=2, $a2=b)`, `($a1=2, $a2=c)`, `($a1=3, $a2=a)`, `($a1=3, $a2=b)` et `($a1=3, $a2=c)`.

```
for $a1 in ("1", "2", "3")
for $a2 in ("a", "b", "c")
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>1 a</valeur>
<valeur>1 b</valeur>
<valeur>1 c</valeur>
<valeur>2 a</valeur>
<valeur>2 b</valeur>
<valeur>2 c</valeur>
<valeur>3 a</valeur>
<valeur>3 b</valeur>
<valeur>3 c</valeur>
```

On peut simplifier l'écriture, en utilisant une seule clause **for** et en séparant par une virgule les deux séquences.

```
for $a1 in ("1", "2", "3"), $a2 in ("a", "b", "c")
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>1 a</valeur>
<valeur>1 b</valeur>
<valeur>1 c</valeur>
<valeur>2 a</valeur>
<valeur>2 b</valeur>
<valeur>2 c</valeur>
<valeur>3 a</valeur>
<valeur>3 b</valeur>
<valeur>3 c</valeur>
```

L'instruction FLWOR - combiner un for et un let

Un **for** combiné avec un **let** va également créer une séquence de **tuples** de valeurs. Mais on combinera les valeurs du **for** avec la seule valeur du **let** (même si celle-ci est une séquence).

Dans cet exemple, les tuples de valeurs seront `($a1=1, $a2=(a,b,c))`, `($a1=1, $a2=(a,b,c))` et `($a1=3, $a2=(a,b,c))`.

```
for $a1 in ("1", "2", "3")
let $a2 := ("a", "b", "c")
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>1 a b c</valeur>
<valeur>2 a b c</valeur>
<valeur>3 a b c</valeur>
```

```
for $p in /carnet/personne
let $tagName:="people"
return element {$tagName} { concat($p/nom, ' ', $p/prénom) }
```

```
<people>Bernard John</people>
<people>Durand Elise</people>
<people>Gobbe Nathalie</people>
<people>Gobbe Jean-Luc</people>
<people>Bareel Valérie</people>
<people>Albert Marc</people>
<people>Declerck Serge</people>
<people>Dumont Julie</people>
<people>Jansens Sophie</people>
<people>Pireaux Eric</people>
<people>Da vinci Martine</people>
```

L'instruction FLWOR - utilisation des variables liées dans les for et les let

La valeur d'un **for** ou d'un **let** peut faire appel à la variable d'un **for** ou d'un **let** précédent :

```
for $a1 in (1, 2, 3)
let $a2 := $a1
return <valeur> { $a1, $a2 } </valeur>

<valeur>1 1</valeur>
<valeur>2 2</valeur>
<valeur>3 3</valeur>
```

```
for $a1 in (1, 2, 3)
for $a2 in 1 to $a1
return <valeur> { $a1, $a2 } </valeur>

<valeur>1 1</valeur>
<valeur>2 1</valeur>
<valeur>2 2</valeur>
<valeur>3 1</valeur>
<valeur>3 2</valeur>
<valeur>3 3</valeur>
```

```
for $p in /carnet/personne
let $nom:=fn:concat($p/nom, ' ', $p/prénom)
return <personne> { $nom } </personne>

<personne>Bernard John</personne>
<personne>Durand Elise</personne>
<personne>Gobbe Nathalie</personne>
<personne>Gobbe Jean-Luc</personne>
<personne>Bareel Valérie</personne>
<personne>Albert Marc</personne>
<personne>Declerck Serge</personne>
<personne>Dumont Julie</personne>
<personne>Jansens Sophie</personne>
<personne>Pireaux Eric</personne>
<personne>Da vinci Martine</personne>
```

L'instruction FLWOR - variable de position dans un for

Un **for** peut utiliser une variable de position grâce au mot-clé **at** :

```
for $a1 at $p1 in ("a", "b", "c"), $a2 at $p2 in ("i", "j", "k")
return <valeur> { concat($p1, "=", $a1, ", ", $p2, "=",$a2) } </valeur>
```

```
<valeur>1=a, 1=i</valeur>
<valeur>1=a, 2=j</valeur>
<valeur>1=a, 3=k</valeur>
<valeur>2=b, 1=i</valeur>
<valeur>2=b, 2=j</valeur>
<valeur>2=b, 3=k</valeur>
<valeur>3=c, 1=i</valeur>
<valeur>3=c, 2=j</valeur>
<valeur>3=c, 3=k</valeur>
```

```
for $z at $zCount in distinct-values(//zip)
return <area zip="{$z}">
  {
  for $p at $pCount in /carnet/personne[adresse/zip = $z]
  return <personne id="{$zCount}.{$pCount}">
    {
    concat($p/nom,' ',$p/prénom)
    }
  </personne>
  }
</area>
```

```
<area zip="6761">
  <personne id="1.1">Bernard John</personne>
  <personne id="1.2">Dumont Julie</personne>
</area>
<area zip="1180">
  <personne id="2.1">Durand Elise</personne>
  <personne id="2.2">Bareel Valérie</personne>
  <personne id="2.3">Jansens Sophie</personne>
</area>
<area zip="6041">
  <personne id="3.1">Gobbe Nathalie</personne>
  <personne id="3.2">Pireaux Eric</personne>
  <personne id="3.3">Da vinci Martine</personne>
</area>
<area zip="1040">
  <personne id="4.1">Gobbe Jean-Luc</personne>
</area>
<area zip="1050">
  <personne id="5.1">Albert Marc</personne>
  <personne id="5.2">Declerck Serge</personne>
</area>
```

L'instruction FLWOR - la clause where

La clause **where** sert à éliminer des **tuples** grâce à une condition booléenne qui est calculée individuellement pour chaque tuple (la condition peut, bien sûr, faire appel aux variables liées) :

```
for $a1 in ("a", "b", "c"), $a2 in ("a", "b", "c")
where $a1!=$a2
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>a b</valeur>
<valeur>a c</valeur>
<valeur>b a</valeur>
<valeur>b c</valeur>
<valeur>c a</valeur>
<valeur>c b</valeur>
```

```
for $z in distinct-values(//zip)
for $p in /carnet/personne
where $p/adresse/zip = $z
return <personne> { fn:concat($p/nom, ' ', $p/prénom, ' (',$z,')') } </personne>
```

```
<personne>Bernard John (6761)</personne>
<personne>Dumont Julie (6761)</personne>
<personne>Durand Elise (1180)</personne>
<personne>Bareel Valérie (1180)</personne>
<personne>Jansens Sophie (1180)</personne>
<personne>Gobbe Nathalie (6041)</personne>
<personne>Pireaux Eric (6041)</personne>
<personne>Da vinci Martine (6041)</personne>
<personne>Gobbe Jean-Luc (1040)</personne>
<personne>Albert Marc (1050)</personne>
<personne>Declerck Serge (1050)</personne>
```

L'instruction FLWOR - la clause order by

La clause **order by** sert comme en SQL à trier les **tuples** avant d'exécuter la clause **return** :

```
for $a1 in ("3", "1", "2")
order by $a1
return <valeur> { $a1 } </valeur>
```

```
<valeur>1</valeur>
<valeur>2</valeur>
<valeur>3</valeur>
```

On peut utiliser les mots-clés **ascending** ou **descending** pour trier par ordre croissant ou décroissant :

```
for $a1 in ("3", "1", "2")
order by $a1 descending
return <valeur> { $a1 } </valeur>
```

```
<valeur>3</valeur>
<valeur>2</valeur>
<valeur>1</valeur>
```

L'instruction FLWOR - la clause order by avec plusieurs critères de tri

```
for $a1 in ("3", "1", "2")
for $a2 in ("c", "a", "b")
order by $a1, $a2
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>1 a</valeur>
```

```
<valeur>1 b</valeur>
<valeur>1 c</valeur>
<valeur>2 a</valeur>
<valeur>2 b</valeur>
<valeur>2 c</valeur>
<valeur>3 a</valeur>
<valeur>3 b</valeur>
<valeur>3 c</valeur>
```

```
for $a1 in ("3", "1", "2")
for $a2 in ("c", "a", "b")
order by $a2, $a1
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>1 a</valeur>
<valeur>2 a</valeur>
<valeur>3 a</valeur>
<valeur>1 b</valeur>
<valeur>2 b</valeur>
<valeur>3 b</valeur>
<valeur>1 c</valeur>
<valeur>2 c</valeur>
<valeur>3 c</valeur>
```

```
for $z in distinct-values(//zip)
for $p in /carnet/personne
where $p/adresse/zip = $z
let $name := fn:concat($p/nom, ' ', $p/prénom)
order by $z, $name
return <personne> { fn:concat($name, ' (',$z,')') } </personne>
```

```
<personne>Gobbe Jean-Luc (1040)</personne>
<personne>Albert Marc (1050)</personne>
<personne>Declerck Serge (1050)</personne>
<personne>Bareel Valérie (1180)</personne>
<personne>Durand Elise (1180)</personne>
<personne>Jansens Sophie (1180)</personne>
<personne>Da vinci Martine (6041)</personne>
<personne>Gobbe Nathalie (6041)</personne>
<personne>Pireaux Eric (6041)</personne>
<personne>Bernard John (6761)</personne>
<personne>Dumont Julie (6761)</personne>
```

L'instruction FLWOR - la clause group by

La clause `group by` va générer des groupes de tuples sur base d'un clé donnée dans l'instruction (tous les tuples ayant la même valeur de clé seront regroupés)

```
for $p in /carnet/personne
group by $z := $p//zip
return <habitants zip="{$z}"> { count($p) } </habitants>
```

```
<habitants zip="1040">1</habitants>
<habitants zip="1050">2</habitants>
<habitants zip="1180">3</habitants>
<habitants zip="6041">3</habitants>
<habitants zip="6761">2</habitants>
```

Les variables liées des **for** précédents ne vont pas contenir des éléments individuels, mais des séquences d'éléments qui résultent du regroupement :

```
for $p in /carnet/personne
group by $z := $p//zip
return <habitants zip="{$z}">
           {
           for $p1 in $p
           return <personne>{ concat($p1/nom,' ',$p1/prénom) }</personne>
           }
       </habitants>
```

```
<habitants zip="1040">
  <personne>Gobbe Jean-Luc</personne>
</habitants>
<habitants zip="1050">
  <personne>Albert Marc</personne>
  <personne>Declerck Serge</personne>
</habitants>
<habitants zip="1180">
  <personne>Durand Elise</personne>
  <personne>Bareel Valérie</personne>
  <personne>Jansens Sophie</personne>
</habitants>
<habitants zip="6041">
  <personne>Gobbe Nathalie</personne>
  <personne>Pireaux Eric</personne>
  <personne>Da vinci Martine</personne>
</habitants>
<habitants zip="6761">
  <personne>Bernard John</personne>
  <personne>Dumont Julie</personne>
</habitants>
```

L'instruction FLWOR - la clause count

La clause **count** permet de rajouter une variable liée qui va simplement compter les tuples (pour réaliser une pagination, par exemple)

```
for $p in /carnet/personne
count $num
where $num<=5
return <personne> { concat($p/nom,' ',$p/prénom) } </personne>
```

```
<personne>Bernard John</personne>
<personne>Durand Elise</personne>
<personne>Gobbe Nathalie</personne>
<personne>Gobbe Jean-Luc</personne>
<personne>Bareel Valérie</personne>
```

L'instruction FLWOR - la clause return

La clause **return** est une expression qui sera évaluée pour chaque **tuples** de valeurs crées. Le résultat sera ajouté à la séquence retournée par l'instruction FLOWR.

Très souvent, cette expression fera appel à un constructeur d'élément :

```
for $a1 at $p1 in ("a", "b", "c"), $a2 at $p2 in ("a", "b", "c")
where $a1=$a2
return <valeur> { $a1, $a2 } </valeur>
```

```
<valeur>a a</valeur>
<valeur>b b</valeur>
<valeur>c c</valeur>
```

Ce n'est pas une obligation, n'importe quelle expression peut être utilisée :

```
for $a1 at $p1 in ("a", "b", "c"), $a2 at $p2 in ("a", "b", "c")
where $a1=$a2
return fn:concat($a1,", ",$a2)
```

```
a, a
b, b
c, c
```

Chapitre 5 - Les facilités d'update XQuery

Table des matières de ce chapitre

Modification dans la base de données

Depuis peu, XQuery est complèté par la norme **XQuery Update Facility 1.0** afin de pouvoir modifier en XQuery le contenu de la base de données.

Elle introduit de nouvelles instructions :

- **insert**
- **delete**
- **replace**
- **rename**
- **copy**

Exemples de de modifications dans la base de données

Exemple d'utilisation de **insert**:

```
for $p in /carnet/personne
return insert node <fullname> {concat($p/nom, ' ', $p/prenom)} </fullname> as last into $p
```

Exemple d'utilisation de **delete**:

```
for $p in /carnet/personne return delete node $p/fullname
```

Exemple d'(utilisation de **rename**:

```
for $p in /carnet/personne return rename node $p/nom as "name"
```

Exemple d'utilisation de **replace**:

```
for $p in /carnet/personne
return replace node $p/name with <nom> {$p/name/text()} </nom>
```

Table des matières complète