

8 XPATH

C'est un langage qui permet de naviguer sur un arbre XML, et c'est la base de XQUERY.

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

Examinons mieux certains de ses operateurs.

En général, l'opérateur / fait descendre d'un niveau. Mais si cet opérateur est le premier symbole d'une expression XPATH, alors cette expression indique un chemin absolu vers un élément donné. Par exemple, si l'élément de départ est :

<AAA>

<BBB>

<CCC>

<FFF>

</FFF>

<DDD>

<EEE>

</EEE>

<DDD/>

</CCC>

</BBB>

</AAA>

Requête /AAA. **Reponse** : ce même élément (arbre).

Un exemple de l'utilisation générale de / :

Même arbre de départ. La requête $AAA/* / CCC$ selectionne tous les petits-enfants de AAA qui sont des CCC

En fait, $*$ indique n'importe quel élément, et l'opérateur $/$ fait descendre d'un niveau.

L'opérateur // permet de sélectionner tous les descendants indiqués.

Arbre interrogé :

<AAA>

<BBB>

<CCC>

<BBB>

<BBB/>

</CCC>

<BBB/>

</AAA>

Requête : //BB. Réponse : deux sous-arbres. Pourquoi ?

Les attributs sont spécifiés par le prefixe @.

Arbre de départ :

```
<AAA>  
<BBB id = "b1"/>  
<CCC id = "b2"/>  
<BBB name = "aaa" />  
<DDD name = "aaa" />  
</AAA>
```

La requête `//@id` sélectionne tous les éléments qui ont une valeur pour l'attribut `id` :

```
<BBB id = "b1"/>  
<CCC id = "b2"/>
```

La requête `//BBB[@name]` sélectionne les éléments `BBB` qui ont une valeur pour l'attribut `name` :

```
<BBB name = "aaa" />
```

La requête `BBB[@*]` sélectionne tous les éléments BBB qui ont un attribut.

La requête `BBB[not (@* )]` sélectionne les éléments BBB qui n'ont pas d'attribut.

9 XQUERY : des notions de base

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

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.

- Déclaratif : en XQUERY tout est une *expression*, qui produit une *valeur* (comparer avec CAML).
- les expressions le plus banales utilisent des *constantes* (et des opérateurs simples) : par ex. $3+4$ est un programme très simple.

Opérateurs arithmétiques : ceux habituels, idem pour les op. Booléens.

$3+4$ est un programme XQUERY qui s'évalue à l'entier 7.

- On a aussi des *variables* : $\$x$, $\$y$, $\$z$ et la possibilité de définir des variables locales grâce au mot-clé `let` :

```
let $x :=5 $y :=6 return 10*$x+$y
```

s'évalue à 56.

Quelques Remarques sur les valeurs

- Une valeur d’une requête XQUERY est une *séquence ordonnée* de 0 ou plusieurs items.
- Ici, un item est un **noeud** ou une valeur atomique.
- Une valeur atomique a un type atomique (mêmes types. at. qu’en XML-schema).
- Dans le jargon XQUERY, il y a 7 sortes de “noeuds” :
 1. Document Node
 2. Element Node
 3. Attribute Node
 4. Text Node
 5. Comment Node
 6. ProcessingInstruction node
 7. Namespace Node

Quelques Remarques sur les valeurs, suite

- Il n'y a pas de distinction entre un item et une séquence de longueur 1
- Il n'y a pas de séquence imbriquée : $(1, (2,3), (4))$ pareil que 1,2,3,4
- Une séquence peut être vide
- Une séquence peut contenir des *données hétérogènes*
- Toutes les séquences sont ordonnées
- *Visualisation* : on peut voir une séquence comme décrivant, en général, une forêt d'arbres.

Qu'est ce qu'une requête XQUERY, en général, ?

- Une requête Q est une expression qui :
- Lit une séquence de fragments XML ou de valeurs atomiques
- Retourne une séquence de fragments XML ou de valeurs atomiques (visualisation : forêt).

Comment construire une requête Q ?

Le formes principales qui peuvent prendre une expression XQUERY sont

- Expressions de chemins, éventuellement filtrée par un prédicat ($\text{XPath} \subset \text{XQuery}$)
- Utilisation de Variables
- Constructeurs
- Expressions FLWOR
- Expressions de listes
- Conditions
- Expressions Quantifiées
- Expressions de types de données
- Fonctions

Expressions de chemins XPATH

Par ex., on interroge un document `recipes.xml` où chaque recette (`recipe`) est une structure complexe ayant (entre autre) un `titre`, une `date` de redaction, des `ingredients` équipées d'un attribut `name` et d'un attribut `amount` (quantité).

Q1 : `doc(''recipes.xml'')//recipe`

donne la suite de tous les éléments `recipes` descendants de la racine.

Expressions de chemins XPATH avec prédicats entre crochets

Supposons que chaque recette ait un fils `title` (le titre du document la décrivant). On peut filtrer les recettes par rapport à la valeur de `title`, par exemple :

```
Q2:doc('recipes.xml')//recipe[title='LinguinePasta']
```

Expressions de chemins XPATH avec prédicats entre crochets

Un autre exemple : on a une BD bibliographique et `titre` est un fils des noeuds `livre`.

On peut filtrer, avec un prédicat, par rapport à la valeur de `titre` :

Q3 : `doc(biblio.xml)//livre[titre='TCP Illustrated']`

On peut filtrer aussi par rapports aux attributs. Par exemple, toujours pour les recettes, supposons que `amount` soit un attribut de l'élément `ingredient`, qui est un descendant de `recipe` :

```
Q4:doc(''recipes.xml'')//recipe[title=''LinguinePasta'']//ingredient
[@amount]
```

on obtient la suite d'ingredients et le quantités correspondantes pour la recette dont la balise `title` prend la valeur `''LinguinePasta''` :

```
<ingredient amount=''2'' unit = ''tablespoon'' > olive oil
</ingredient>
```

```
<ingredient amount=''2'' > garlic gloves </ingredient>
```

⋮

Pourquoi ?

Puisque les arbres XML sont ordonnés, le prédicat peut aussi filtrer selon l'ordre d'occurrence de l'élément cherché dans l'arbre (de gauche vers droite) :

```
Q5 .5 doc(biblio.xml)//bib/book[1]/
```

Cette requête retourne une séquence d'éléments qui sont des enfants du premier élément de type book

Autre exemple :

```
Q6 : doc(biblio.xml)//bib/book[3]
```

On peut faire de la concatenation de séquences :

Q7 : <MonLivre>

```
doc('biblio.xml')//book[1]/(price, author)
</MonLivre>
```

NB : dans le document, les prix n'étaient pas forcément avant les auteurs : Dans le résultat de Q7, si.

Et puis : pourquoi on a utilisé la nouvelle balise MonLivre ?

Soient $n1$ et $n2$ des noeuds. L'expression $n1 = n2$ veut dire que la valeur de $n1$ est égale à celle de $n2$. Ainsi, la requête

```
Q8 : doc(biblio.xml)//bib/book[author[1]/firstname =  
author[last()]/firstname]
```

s'évalue à la séquence des éléments `book` tels que :

le prénom du premier auteur est le même que celui du dernier auteur.

NB : mot-clé `last()`.

Constructeurs Une expression XQuery peut construire un nouvel élément XML

```
Q9 : <employee empid=' '12345' '>  
<name> John Dole </name>  
<job> XML Specialists </job>  
</employee>
```

Le résultat de Q9 est elle même. **NB** : Cette donnée pouvait être *absente de la base de départ* ; les balises l'ont construite.

Avec les Variables

```
Q10 : <employee empid={$id}>  
<name> {$name} </name>  
<job> XML Specialists </job>  
</employee>
```

Pour pouvoir évaluer, les variables doivent être liées à des fragments XML (voir après).
Les accolades : parenthèses.

Comparaison de la position de deux noeud : $n1 \ll n2$ (resp. $n1 \gg n2$) est vrai ssi $n1$ apparaît avant (après) $n2$ dans le document.

Q11 : <livre>

```
doc('bib.xml'//book[author[name = 'Abiteboul']] <<
author[name = 'Suciu']]/@title } </livre>
```

On cherche les titres des books dont l'auteur qui s'appelle Abiteboul apparaît avant l'auteur Suciu.

Par ex. on obtient :

```
<livre title = 'Data on the Web' />
```

NB : C'est le @ à indiquer que title est un attribut (de book).

FLWOR

- **Syntaxe**

`for var dans expr let var := expr where expr order by expr return expr.`

- **Signification opérationnelle**

`for` et `let` génèrent une liste de tuples de variables liées, en preservant l'ordre du document.

`where` applique un prédicat, et filtre les tuples.

`order by` impose un ordre sur les tuples qui restent.

`return` est exécuté pour chaque tuple choisie $\Rightarrow$ liste ordonnée de outputs.

Exemple : count, avg, descending sont de fonctions *built-in*, analogues à celles de SQL.

Trouver la description et le prix moyen de toute partie rouge qui a été ordonnée au moins 10 fois.

```
Q5: for $p in doc(''parts.xml')//part[color='Red']
let $o := doc(''orders.xml'')//order[partno= $p/partno]
where count($o) > 10
order by count($o) descending
return
<important_red_part>
{$p/description}
<avg_price> avg($o/price) </avg_price>
</important_red_part>
```

NB : important_red_part, avg_price sont des balises nouvelles au document interrogé : on a fait de l'extraction et aussi de la reorganisation de données !

Différence entre le for et le let

Intuition : Analogie avec l'instruction d'un pseudo-langage procédural :

```
for x in [1...5] let d := 2*x
```

x indique une séquence de valeurs (ceux de la boucle) : 1,2,3,4,5.

d est affecté, chaque fois à une seule valeur : d'abord 2, puis 4, puis 6, puis 9, puis 10.

Bref : le `for x` donne l'espace de recherche de valeurs possibles pour d, donc l'espace de la boucle.

Le `for` permet, entre autre, d'effectuer une **jointure** entre 2 documents.

SITUATION

On a un document `www.irs.gov/taxpayers.xml` avec des infos sur les personnes potentiels contribuents aux impôts d'une nation donnée, en particulier sur leur imposable annuel. Les personnes sont identifiées par leur SNN.

On a aussi un document `regions.xml` avec plein d'infos sur les habitants d'une region donnée, region par region ; entre autre on associe un nom à tout SNN.

Auant les personnes du premier document que les habitants du second ont donc un attribut partagé `ssn`.

On veut connaître l'imposable des personnes, leur nom et leur `ssn`.

Interrogation $\Rightarrow$

```
Q6: for $p IN doc(''www.irs.gov/taxpayers.xml''//person
for $n IN doc(''regions.xml''//habitant[ssn=$p/ssn]
return
```

```
<person>
<ssn> $p/ssn </ssn>
$n/name
<imposable> $p/imposable </imposable>
</person>
```

Q6 s'évalue à une séquence d'éléments de la forme :

```
<person>
<ssn> {123456} </ssn>
Dupont
<imposable> 42300 </imposable>
</person>
```

Un autre exemple : *SITUATION*

On a un document concernant des départements d'une institution, des infos sur le nombre des employés pour chaque dept.

On a un autre document concernant les employés de l'institution, leur salaire etc.

Q7 :

```
for $d in doc(''depts.xml'')//dept_no
let $e := doc(''emps.xml'')//employee[dept_no=$d]
where count($e)>=10
order by avg($e/salary) descending
return <big_dept> {$d,
<headcount> { count($e)} </headcount>,
<avgsal> { avg($e/salary)} </avgsal>
}
big_dept
```

Résultat ?

XPATH et XQuery gèrent des listes de valeurs.

Q8 :

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

Résultat : liste des éditeurs avec le prix moyen de leur livres.

NB : Le mot-clé `text` permet de n'extraire que le contenu textuel d'un élément donné : plus d'exemples de son utilité en TD.

Conditions

Le `if ... then ... else ....`

Exemple

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

Le `else` est obligatoire, mais `else( )` est admis.

Expressions Quantifiées

Quantification existentielle. Mots clés : some, satisfies

Exemple

```
for $b in doc('bib.xml')//book
where some $p in $b//paragraph satisfies
<contains($p, 'informatique') and (contains($p,
'thriller'))
return $b/title
```

NB : contains est *built-in*.

Expressions Quantifiées

Quantification universelle. Mots clés : every, satisfies

Exemple

```
for $b in doc('bib.xml')//book
where every $p in $b//paragraph satisfies
<contains($p, 'informatique')
return $b/title
```

Encore quelques exemples d'interrogation.

Document “Biblio.xml”

```
<livres>
<livre annee=''1994''>
<titre> TCP/IP Illustrated </titre>
<auteur> <nom> Stevens </nom> <prenom> W. </prenom></auteur>
<maison.edition> Addison Wesley </maison.edition>
<prix> 65.95 </prix>
</livre>
<livre annee=''1992''>
<titre> Advanced Programming in the Unix Environment </titre>
<auteur> <nom> Stevens </nom> <prenom> W. </prenom></auteur>
<maison.edition> Addison Wesley </maison.edition>
<prix> 65.95 </prix>
</livre>
<livre annee=''2000''>
<titre> Data on the Web</titre>
<auteur> <nom> Ger </nom> <prenom> S. </prenom></auteur>
<auteur> <nom> Bunemann </nom> <prenom> P. </prenom></auteur>
<auteur> <nom> Suciu </nom> <prenom> D. </prenom></auteur>
<maison.edition> Morgan Kauffmann </maison.edition>
<prix> 129.95 </prix>
</livre>
<livre annee=''1999''>
<titre> The Economics of Technology and Content for Digital TV </titre>
<editeur> <nom> Gerbarg </nom> <prenom> D. </prenom> <affiliation> CITI </affiliation> </editeur>
<maison.edition > Academic maisons.editions </maison.edition>
</livre>
</livres>
```


R1 : *Lister les prix des tous les livres*

```
doc( ' 'Biblio.xml' ' )//prix
```

R2 : *Lister les prix des livres dont le titre est “TCP/IP Illustrated”*

```
doc('Biblio.xml')/*livre[titre = 'TCP/IP  
Illustrated']/prix
```

ou bien

```
doc('Biblio.xml')//livre[titre = 'TCP/IP  
Illustrated']/prix
```

Ici, un seul résultat : la valeur 65.95.

R3 : Construire un nouveau document XML dont la balise racine est livres_chers et qui contient les titres des livres dont le prix est supérieur à 65.00.

R3 :

```
for $l in doc('Biblio.xml')//livre
let $t := $l/titre]
where $l/price > 65.00
return
<livres_chers> $t </livres_chers>
```

XQuery est un langage riche, avec beaucoup de fonctionnalités.

Certaines (pas toutes !) seront vues en TD, sur machine, en interrogeant une base XML réelle : mondial.xml.