XML et Programmation Internet

Cours 3

kn@lri.fr

Plan

UNIVERSITE PARIS SUD
Comprendre le monde, construire l'avenir®

Rappels de syntaxe

```
p ::=   p or p
    |   p and p
    |   not (p)
    |   count(…), contains(…), position(), …
    |   chemin XPath
    |   e1 op e2
```

On évalue le prédicat et on converti son résultat en valeur de vérité. Si la valeur vaut vrai, on garde le nœud courant, si elle vaut faux, on ne le garde pas

XPath connait 4 types de données pour les prédicats :

- Les booléens, valeur de vérité : vrai ou faux
- Les nombres (flottants), valeur de vérité compliquée...
- Les chaînes de caractères, chaîne vide = faux, sinon vrai
- Les ensembles de nœuds, ensemble vide = faux, sinon vrai

UNIVERSITE PARIS SUD
Comprendre le monde, construire l'avenir®

Comparaisons (e_1 op e_2)

Les opérateurs de comparaisons sont : `=`, `!=`, `<`, `<=`, `>`, `>=` .
La manière de calculer de vérité de e_1 op e_2 dépend du typed de e_1 et e_2 :

- Si e_1 et e_2 représente des ensembles de nœuds, alors la comparaison est vraie ssi il existe un élément `x` dans e_1 et un élément `y` dans e_2 tels que `x op y`
- Si e_1 représente un ensembles de nœuds et e_2 une valeur scalaire `y`, alors la comparaison est vraie ssi il existe un élément `x` dans e_1 tel que `x op y`
- Si e_1 et e_2 sont des valeurs scalaires, alors ont les compare en utilisant les règles de comparaison des valeurs scalaires (voir page suivante).

UNIVERSITE PARIS SUD
Comprendre le monde, construire l'avenir®

Comparaisons des valeurs scalaires

v_1 op v_2

Si op est `!=` ou `=`, on applique les règles dans cet ordre:

1. **Si v_1 (resp. v_2) est un booléen, alors v_2 (resp. v_1) est converti en booléen et les deux booléens sont comparés**
2. **Sinon si v_1 (resp. v_2) est un nombre, alors v_2 (resp. v_1) est converti en nombre et les deux nombres sont comparés**
3. **Sinon, v_1 et v_2 sont des chaines de caractères, on les compares**

Si op est `<`, `<=`, `>` ou `>=`, on convertit v_1 et v_2 en nombres et on les compare.

Conversions

Conversion en booléen

- **0 et NaN sont converti en `false`, le reste en `true`**
- **Un ensemble de nœud vaut `true` ssi il est non vide**
- **Une chaine vaut `true` ssi elle est non vide**

Conversion en nombre

- **Une chaine de caractère représentant un flottant au format IEEE-754 est convertie en ce nombre, sinon elle est convertie en NaN**
- **Booléen: `true` est converti à 1, `false` est converti à 0**
- **Un ensemble de nœud est d'abord converti en chaine de caractères puis la chaine est convertie en nombre**

Conversions (suite)

Conversion en chaine de caractères

- **Un booléen est traduit en `"true"` ou `"false"` selon sa valeur**
- **Nombres:**
  - **NaN est converti en la chaine `"NaN"`**
  - **Si le nombre n'a pas de partie décimale, sa représentation *entière* est convertie en chaine (ex: 1.000e10 ≡ `"10"`)**
  - **Sinon une représentation IEE-754 du nombre est utilisé (ex: `"123.10e-34"`)**
- **Ensemble de noœud :**
  - **L'ensemble vide est converti en la chaine vide**
  - **Sinon le premier élément dans l'ordre du document est converti en chaine en concatenant tous les nœuds textes dans ces descendants (y compris lui-même)**

Appels de fonction

Il existe des fonctions prédéfinies (voir la spec XPath)

- `contains(str1,str2)`
- `starts-with(str1,str2)`
- `count(node_set1)`
- `last()`
- `position()`
- **...**

Si les arguments ne sont pas du bon type, ils sont convertis en utilisant les règles de conversion

Exemples

Dans la suite, on se donne un document de test ayant une racine `<a> ... </a>` et une expression XPath qui sélectionne la racine si et seulement si le prédicat vaut vrai

Exemples

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

1. `/child::a[ child::*/child::text() ]` **sélectionne la racine (l'ensemble de nœud renvoyé par** `child::*/cild::text()` **est non vide, donc il est converti en** `true`**)**
2. `/child::a[ child::*/child::text() = "2" ]` **sélectionne la racine (l'ensemble de nœud texte renvoyé par** `child::*/child::text()` **est comparé à "2", et il existe un élément dans cet ensemble pour lequel le test réussi)**
3. `/child::a[ child::*/child::text() != "2" ]` **sélectionne la racine (l'ensemble de nœud texte renvoyé par** `child::*/child::text()` **est comparé à "2", et il existe un élément dans cet ensemble pour lequel le test réussi)**
4. `/child::a[ not(child::*/child::text() = "2") ]` **ne sélectionne pas la racine (on prend la négation du deuxième cas ci-dessus)**

Exemples

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

1. `/child::a[ child::*/child::text() > 1.5 ]` **sélectionne la racine (l'ensemble de nœud texte renvoyé par** `child::*/child::text()` **est comparé à 1.5, et il existe un élément dans cet ensemble pour lequel le test réussi)**
2. `/child::a[ child::b/child::text() >= child::c/child::text() ]` **sélectionne la racine (les deux ensembles de nœuds sont convertis en ensmbles de nombres, car on utilise** `>=` **et on a bien que** `2 >= 2` **)**
3. `/child::a[ child::b/child::text() = child::c/child::text() ]` **sélectionne la racine (les deux ensembles de nœuds comportent un élément commun)**
4. `/child::a[ child::b/child::text() != child::c/child::text() ]` **sélectionne la racine (les deux comportent des éléments différents)**

Exemples

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

1. `/child::a[ contains(self::*, "22") ]` **sélectionne la racine(l'ensemble de nœud séléctionné par** `self::*`**, i.e. la racine est converti en chaine. Pour ce faire, on colle toutes éléments textes descendants et on obtient la chaine "1223" qui contient bien "22")**
2. `/child::a[ self::* > 442.38 ]` **sélectionne la racine(l'ensemble de nœud séléctionné par** `self::*`**, est converti en chaine puis en nombre pour comparer 1223 à 442.38)**
3. `/child::a[ sum(child::*) >= 7.5 ]` **sélectionne la racine(la fonction** `sum` **converti la liste de valeurs passée en argument en liste de nombre et fait la somme de ces derniers)**

Exemples

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

1. `/child::a[ sum(child::*) >= 7.5 ]` **ne sélectionne pas la racine** (la fonction `sum` converti la liste de valeurs passée en argument en liste de nombres, toto n'étant pas un nombre valide, il est remplacé par NaN. La somme totale fait donc NaN, et une comparaison avec NaN renvoie toujours faux)

Prédicat imbriqués

On peut imbriquer des prédicats de manière arbitraire:

$Q_1 \equiv$ `/child::a[ child::b[ count(descendant::c) > 4 ] ]`

Quelle différence avec :

$Q_2 \equiv$ `/child::a[ count(child::b/descendant::c) > 4 ]`

Il suffit de considérer le document :

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

Q_1 **ne séléctionne rien car il n'y a aucun** `b` **ayant plus de 4 descendants** `c`.
Q_2 **séléctionne la racine car le nombre de descendants** `c` **de nœuds** `b` **est plus grand que 4.**

`position()` et `last()`

La fonction `position()` **renvoie la position du nœud au sein de l'ensemble de résultats en cours de filtrage. Last renvoie le nombre d'éléments dans cet ensemble (ou l'indice du dernier élément). Les indices commencent à 1 :**

```
<a>
  <b>AA</b>
  <b>BB</b>
  <b>CC</b>
</a>

/child::a/child::b[ position() = 2 ]          (renvoie <b>BB</b>)
/child::a/child::b[ position() = last() ]     (renvoie <b>CC</b>)
/child::a/child::b[ position() mod 2 = 1 ]    (renvoie <b>AA</b>
                                                       <b>CC</b>)
```


Plan

L'axe attribute::

Permet d'accéder aux attributs d'un élément.**Attention**, les attributs ne font pas partie des fils ni des descendants!

```
<a>
  <b id="a1" v="x" >AA</b>
  <b id="b2" v="y" >BB</b>
  <b id="b3" v="z" >CC</b>
</a>

/descendant::b[ attribute::* = "y" ]          (renvoie <b …>BB</b>)
/descendant::b[ attribute::id = "y" ]         (ne renvoie rien)
```


Les axes preceding:: et following::

L'axe `preceding::` selectionne tous les nœuds arrivant avant le nœud courant et qui ne sont pas des ancetres de ce dernier.

L'axe `following::` selectionne tous les nœuds arrivant après le nœud courant et qui ne sont pas des descendants de ce dernier.

```
<a>
  <b > <c/>
       <d> <e/> </d>
       <f ><g/></f>
  </b>
  <h/>
  <i/>
  <j> <k>
      <l/> <m/> <n/>
    </k>
  </j>
</a>

/descendant::m/preceding::* (séléctionne l, i, h, b, c, d, e, f, g)
/descendant::d/following::* (sélectionne h, i, j, k, l, m)
```


Autres opérateurs

On peut donner plusieurs prédicats à un chemin :

```
/descendant::a [ descendant::b ][ position () > 4 ][ child::c ]
```

Sélectionne l'ensemble des nœuds A_1 ayant un tag `a` et ayant un descendant `b`. Filtre A_1 pour ne garder que A_2, l'ensemble des nœuds de A_1 étant en position supérieure à 4. Filtre A_2 pour ne garder que les nœuds ayant un fils `c`.

On peut prendre l'union de plusieurs chemins :

```
/descendant::a/parent::b | /descendant::c/following-sibling::d
```


Plan

Abréviations

XPath étant très verbeux il existe une syntaxe abrégée pour les situations les plus courantes :

- **un nom de tag** `foo` **est l'abréviation de** `child::foo`**. Exemple :**
  `/a/b/c ≡ /child::a/child::b/child::c`

- **un** `//` **est l'abréviation de** `/descendant-or-self::node()/`**. Exemple :**
  `//a ≡ /descendant-or-self::node()/child::a`

Prend tous les nœuds du document (y compris le nœud fictif `#document` **et exécute** `child::a` **sur cet ensemble.**

- `..` **est un synonyme pour** `parent::node()`
- `@foo` **est un synonyme pour** `attribute::foo`

Exemple :

```
//book [ year > 2005 ]/title
```