Le langage XPath

Pierre Genevès
CNRS

ENSIMAG/TELECOM 3A, Dec. 2008

Pourquoi XPath ?

La recherche, sélection et extraction d'information dans les documents XML est au coeur de n'importe quel traitement de données XML.

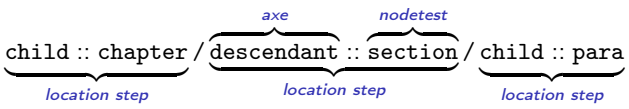
→ XPath est le langage standard du W3C pour exprimer le parcours ou la navigation dans les arbres XML.

Introduction à XPath

- Une syntaxe et une sémantique communes à de nombreux langages web
- Une recommandation du W3C (www.w3.org/TR/xpath)
- Syntaxe compacte, non-XML, pour utilisation dans les attributs XML
- Un langage d'expression de chemins
- XPath opère sur la structure logique (arborescente) des documents XML, pas sur leur syntaxe

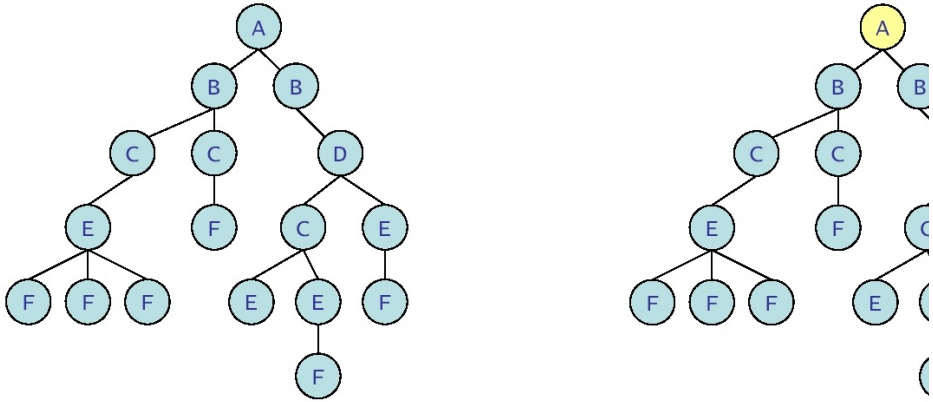
Expressions XPath

- XPath fournit un mécanisme puissant de navigation dans les arbres XML: le *location path*
- Un *location path* est une succession d'étapes de navigation (*location steps*) séparées par des '/':



Évaluation d'un *location path*

- A partir d'un nœud courant, un *location path* retourne un *node-set*
- Chaque nœud de ce *node-set* devient à son tour le nœud courant pour l'évaluation du *step* suivant



Contexte d'évaluation

- Toute expression XPath est évaluée par rapport à un *contexte* comprenant :
  - le nœud courant (*context node*)
  - deux entiers > 0 résultants de l'évaluation du *step* précédent:
    - la *taille du contexte*: le nombre de nœuds du *node-set*
    - la *position*: l'indice du nœud courant dans le *node-set*
  - un ensemble de variables valuées (liées par le langage hôte)
- La navigation "propage" le *contexte*: l'évaluation de chaque *step* produit un nouvel *état de contexte*
- Remarque: un *location path* débutant par '/' indique que le *contexte* initial est restreint à la racine du document, un tel *location path* est dit *absolu*

Zoom sur les *location steps*

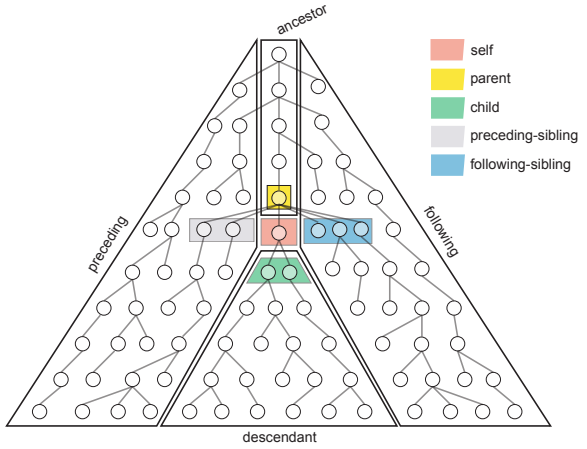
- A chaque étape de navigation les nœuds peuvent être filtrés à l'aide de *qualifiers*
- Syntaxe générale d'un *location step* :

$$axis::nodetest[qualifier][qualifier]$$

- Un *location step* a trois parties :
  1. un *axe* : relation structurale entre le nœud courant et les nœuds retournés par le *step*
  2. un *nodetest* : type et nom des nœuds retournés par le *step*
  3. des *qualifiers* optionnels qui raffinent la sélection opérée par le *step*
- Les qualifiers sont appliqués l'un après l'autre, après la sélection par l'axe et le nodetest
- Chaque qualifier retourne un *node-set* filtré par le qualifier suivant
- Exemple :
  - `child::section[child::para]`

Axes

- Indique dans quelle partie de l'arbre (par rapport au nœud courant) il faut chercher les nœuds à retourner
- XPath définit 13 *axes* permettant de naviguer:



- 5 *axes* forment une partition de l'arbre

Axes

- Chaque *axe* a une direction: avant ou arrière (par rapport à l'*ordre du document*)
- Autres axes:
  - `ancestor-or-self`, `descendant-or-self`
  - l'axe `attribute` permet d'accéder aux attributs de l'élément courant
  - l'axe `namespace` permet d'accéder aux noeuds namespace de l'élément courant

Nodetest

- Le *nodetest* d'un *location step* indique quels nœuds il faut choisir sur l'*axe* considéré
- A *nodetest* filters nodes based on *kind* and *name*

Kind Test	**Semantics**
`node()`	let any node pass
`text()`	preserve text nodes only
`comment()`	preserve comment nodes only
`processing-instruction()`	preserve processing instructions

Name test

- Un *nodetest* peut aussi être un *name test*, préservant seulement les nœuds dont le nom correspond

Name Test	**Semantics**
name	preserve **element** nodes with tag *name* only
	(for attribute axis: preserve attributes)
*	preserve **element** nodes with arbitrary tag names
	(for attribute axis: preserve attributes)

- Remarques:
  - $path/axis$`::*` $\subseteq$ $path/axis$`::node()`
  - $path$`/attribute::node()` $\not\subseteq$ $path$`/child::node()`

Qualifier

- Un *qualifier* filtre un *node-set* en fonction de l'*axe* pour produire un nouveau *node-set*
- Un *qualifier* est une expression booléenne évaluée en fonction du *contexte* :
  - nœud courant
  - *taille* : nombre de nœuds dans le *node-set* filtré
  - *position* : rang du nœud dans le *node-set*, dans l'ordre du document (ou dans l'ordre inverse pour les axes *arrières*)
- Chaque nœud du *node-set* n'est conservé que si l'évaluation du *qualifier* pour ce nœud retourne *true*
- Exemples :
  - `following-sibling::para[position()=last()]`
  - `child::para[position() mod 2 = 1]`

Comparaison de valeurs

- Les *qualifiers* peuvent comporter des comparaisons:

  $path[path_1 \textbf{ eq } path_2]$ où $\textbf{eq} \in \{=, !=, <, >, <=, >=\}$

- Comparaison quantifiée existentiellement:

$$node\text{-}set_1 \textbf{ eq } node\text{-}set_2$$
$$\text{ssi}$$
$$\exists n_1 \in node\text{-}set_1, \exists n_2 \in node\text{-}set_2 \mid \texttt{string-value}(n_1) \textbf{ eq } \texttt{string-value}(n_2)$$

- `string-value`(n) : concaténation des nœuds texte descendants dans l'*ordre du document*
- Exemple: `descendant::chapter[child::section="Conclusion"]`

  tous les noeuds "`chapter`" dont au moins un enfant "`section`" a la *string-value* "Conclusion".
- Les comparaisons se font après avoir potentiellement converti les objets à comparer en un même type (ex: `a[b>7]`)

Expressions XPath générales

- La construction syntaxique de base (*expression*) de XPath peut être un *location path*, ou une union de *location paths* séparés par '|'
- Les *qualifiers* peuvent faire intervenir des expressions booléennes : $path$`[(`$path$ **eq** $path$`) or (`$qualifier$ `and not(`$qualifier$`))]`
- Une expression XPath peut faire intervenir des *variables* (notation: $x)
  - les variables sont liées par le langage hôte (ce sont des constantes ☺)
  - elles font partie du contexte d'évaluation

Observation sur les comparaison de valeurs

- Supposons la variable $x liée à un *node-set*
- Que pensez-vous des expressions XPath e_1 et e_2 suivantes ?

$\underbrace{\texttt{\$x="foo"}}_{e_1}$ $\underbrace{\texttt{not(\$x!="foo")}}_{e_2}$

- e_1 ne signifie pas la même chose que e_2 :

→ e_1 est vraie ssi il existe un nœud de $x qui a la *string-value* foo;

→ e_2 est vraie ssi tous les nœuds de $x ont la *string-value* foo.

- Grâce à la négation et la comparaison définie par quantification existentielle, on peut exprimer la quantification universelle...
  - les noeuds "`chapter`" dont tous les enfants "`section`" sont vides[1] ?

  → `descendant::chapter[not(child::section!="")]`

[1] ont une *string-value* vide

Fonctions de base

- Les *node-sets* ne sont pas les seuls types d'expressions XPath: il y a aussi des *expressions booléennes*, *numériques* et *chaînes de caractères*
- Toute implémentation de XPath doit comprendre au moins les fonctions de base (*Core Function Library*: référence donnée en annexe)
- Exemples :
  - `last()` : un nombre, la *taille* du contexte
  - `position()` : un nombre, la *position* dans le contexte
  - `count(`*node-set*`)` : nombre de nœuds dans le *node-set*
  - `concat(`*string*`, `*string*`, `*string**`)` : concatène plusieurs chaînes
  - `contains(`*str1*`, `*str2*`)` : booléen, vrai si *str1* contient *str2*
  - ...
- Toute expression XPath peut être utilisée dans un *qualifier*, exemple:

```
descendant::recipe[count(descendant::ingredients)<5 and
contains(child::title, "cake")]
```

Syntaxe abrégée

- `child::` est l'axe par défaut, il peut être omis
- `@` est l'abréviation de `attribute::`
- `//` est l'abréviation de `/descendant-or-self::node()/`
- `.` est l'abréviation de `self::node()`
- `..` est l'abréviation de `parent::node()`
- `[4]` est l'abréviation de `[position()=4]`

Exemple	Forme expansée
`book/section`	`child::book/child::section`
`p[@id="bla"]`	`child::p[attribute::id="bla"]`
`.//p`	`self::node()/descendant-or-self::node()/child::p`
`../title`	`parent::node()/child::title`
`p[3]`	`child::p[position()=3]`

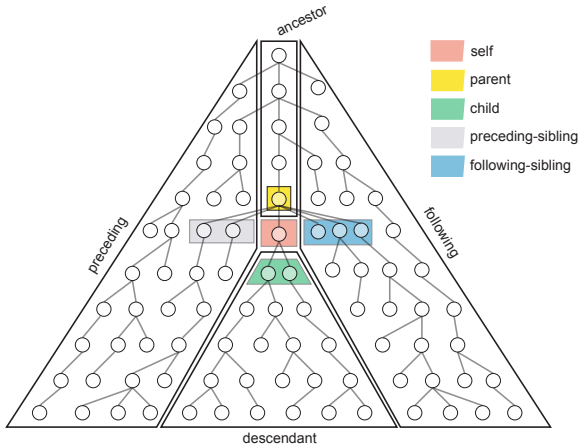
Question...

Que pensez-vous des expressions XPath e_1 et e_2 suivantes ?

$\underbrace{\texttt{self::title}}_{e_1}$ $\underbrace{\texttt{parent::node()/child::title}}_{e_2}$

Question...

Peut-on réécrire l'expression XPath `following::p` sans utiliser l'axe `following` ?



XPath est au coeur des technologies XML

- XPath est utilisé dans:
  - XSLT : sélection des parties de document à transformer
  - XPointer : identification de fragments de document XML
  - XLink : ancre des liens hypertexte
  - XQuery : sous-ensemble du langage d'interrogation
  - XML Schema : expression du domaine sur lequel l'unicité doit être assurée
  - XForms: dépendances entre données (bindings)
  - ...
- Souvent, c'est même **le composant essentiel.**

Annexes

XPath *Core Function Library*

Fonctions sur les *node-sets*

- `last()` : un nombre, la *taille* du contexte
- `position()` : un nombre, la *position* du contexte
- `count(`*`node-set`*`)` : nombre de nœuds dans le *node-set*
- `id(`*`object`*`)` : *node-set* contenant les éléments du document qui ont pour ID la valeur de *object*
- `local-name(`*`node-set`*`)` : la chaîne du nom local du premier nœud de *node-set*
- `namespace-uri(`*`node-set`*`)` : la chaîne de l'URI de l'espace de nom du premier nœud de *node-set*
- `name(`*`node-set`*`)` : la chaîne du nom complet du premier nœud de *node-set*

Fonctions sur les chaînes

- `string(`*`object`*`)` : convertit *object* en une chaîne de caractères
- `concat(`*`string, string, string*`*`)` : concatène plusieurs chaînes
- `start-with(`*`string1, string2`*`)` : booléen, vrai si *string1* commence par *string2*
- `contains(`*`str1, str2`*`)` : booléen, vrai si *str1* contient *str2*
- `substring-before(`*`string1, string2`*`)` : la sous-chaîne de *string1* qui précède la première occurrence de *string2*
- `substring-after(`*`string1, string2`*`)` : la sous-chaîne de *string1* qui suit la première occurrence de *string2*
- `substring(`*`string, number1, number2`*`)` : la sous-chaîne de *string* qui commence à la position *number1* et dont la longueur est *number2*
- `string-length(`*`string`*`)` : nombre de caractères dans *string*
- `normalize-space(`*`string`*`)` : retire les espaces initiaux, finals et les espaces doublés
- `translate(`*`s1, s2, s3`*`)` : remplace dans *s1* chaque caractère de *s2* par le caractère de même rang de *s3*
  exemple : `translate("bar","abc","ABC")` retourne `BAr`

Fonctions booléennes

- `boolean(`*`object`*`)` : convertit *object* en booléen, retourne true si nombre non nul, *node-set* non vide, chaîne de longueur non nulle
- `not(`*`boolean`*`)` : négation de *boolean*
- `true()`
- `false()`
- `lang(`*`string`*`)` : la langue (attribut `xml:lang`) du nœud courant est la même ou un sous-langage de *string*

Fonctions numériques

- `number(`*`object`*`)` : convertit *`object`* en un nombre
- `sum(`*`node-set`*`)` : somme du résultat de la conversion en nombre de chaque nœud de *`node-set`*
- `floor(`*`number`*`)` : plus grand entier inférieur ou égal à *`number`*
- `ceiling(`*`number`*`)` : plus petit entier supérieur ou égal à *`number`*
- `round(`*`number`*`)` : entier le plus proche de *`number`*

Précédence des opérateurs

1. `<=`, `<`, `>=`, `>`
2. `=`, `!=`
3. `and`
4. `or`