XMLSchema - XQuery - (JSON)

E.Coquery

emmanuel.coquery@univ-lyon1.fr

http://liris.cnrs.fr/~ecoquery
→ Enseignement → GDW

Espaces de nommage

- Ambiguïté sur les noms XML
  - Problème similaire aux modules/packages en programmation
- Nom qualifié = Espace de nommage + nom local
- Espace de nommage : une URI
- Nom local : plus ou moins un identifiant dans un langage de programmation
  `([A-Z]|"_"|[a-z]|...)([A-Z]|"_"|[a-z]|"-"|"."|[0-9]|...)*`
- Syntaxiquement :
  - *nomLocal*
    utilise un espace de nommage par défaut
  - *prefixe*:*nomLocal*
    l'espace de nommage est celui rattaché à *prefixe*

Espaces de nommage : déclarations

- Via des attributs spéciaux,
  - valeur de l'attribut = espace de nommage concerné
- Portée : élément contenant l'attribut spécial et tous ses descendants
- Attribut `xmlns` : définit l'espace de nommage par défaut pour les éléments
- Attribut `xmlns:`*`prefixe`* : attache un espace de nommage au préfixe *`prefixe`*

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<livres xmlns="http://www.livres-pas-chers.com">
  <livre xmlns:encyclo="http://toutsurleslivres.org"
         ISBN="123456">
    <auteur encyclo:nat="Américain">
       Stephen King
    </auteur>
    <titre>Le fléau</titre>
    <annee>2003</annee>
    <encyclo:annee>1978</encyclo:annee>
    <prix>5.3</prix>
  </livre>
</livres>
```

noir : pas d'espace de nommage

Qu'est qu'un schema ?

- Relationnel : Ensemble de contraintes que doit vérifier une instance d'une BD
  - Attributs des tuples d'une relation
  - Contraintes de type
  - Contraintes de clé
  - ...
- XML : Ensemble de contraintes structurelles que doit vérifier un document XML
  - Attributs/Enfants autorisés/requis dans un éléments
  - Type des valeurs pour les attributs et le texte
  - ...

$\Rightarrow$ DTD, XML Schema

Langages de schema pour XML

- Les plus connus :
  - **DTD** : Document Type Definition
    - Pas de gestion des espaces de nommage
  - **XML Schema**
    - Syntaxe XML qui peut prêter à confusion
  - Relax NG
- Certains schemas sont publics
  - Ex : XHTML, SVG, SOAP, MathML, OpenDocument, OpenXML, ...

Un peu de théorie : les grammaires d'arbres

- Analogue aux grammaires algébriques
  - mais reconnaissent des arbres
- grammaire $\approx$ ensemble d'arbres
- décrit la structure qui doit être respectée
- Simplification de la syntaxe vàv DTD et XML Schema
- Pouvoir d'expression :
  - $>$ DTD
  - $\cong$ XML Schema

Digression : les types primitifs

- Types des données
- Type $\approx$ ensemble de chaînes de caractères
  - Éventuellement sémantique associée
- Dans les grammaires d'arbres :
  - Types primitifs supposés connus, fixés à l'avance
- Dans les DTD :
  - Types enum + 4 types prédéfinis
- Dans XML Schema : *simple types*
  - Types prédéfinis + mécanismes de création

Types primitifs dans les DTDs

- `#PCDATA` / `CDATA` : n'importe quel texte
- `(`val_1 `|`val_2 `|...)` : type énuméré
- `ID` valeur n'apparaissant qu'une fois dans le document là où on attend un ID ($\approx$ clé).
  - caractères : `A-Z`, `a-z`, `0-9`, `-`, `_`, ...
- `IDREF` / `IDREFS` valeur(s) apparaissant ailleurs en tant qu'ID ($\approx$ clé étrangère)

`#PCDATA` $\leftrightarrow$ noeuds texte uniquement

Types simples en XML Schema

Types primitifs en XML Schema

- Types prédéfinis
  - string, boolean, integer, float, time, date
  - `http://www.w3.org/TR/xmlschema-2/`
- Définition de nouveau types :
  - par restriction d'un type existant
  - par union
  - comme listes de types

Définition par restriction

- Restriction $\leftrightarrow$ sous-ensemble du type restreint
- Exemple code barre :

```
<simpleType name="codeBarre">
    <restriction base="string">
        <pattern value="[0-9]{6}-[0-9]{6}"/>
    </restriction>
</simpleType>
```

- Possibilités de restriction :
  - Expressions rationnelles, taille (min,max)
  - Énumération de valeurs (type en extension)
  - Valeur minimale et/ou maximale
    - liée à la sémantique sous-jacente au type restreint

Définition par union/liste

- Union de types simples
  - Union des ensembles de chaînes de caractères correspond aux type simples utilisés
  - Exemple code barre ou référence : Spécifie un type qui est soit un code barre soit un entier compris entre 0 et 65535
- Liste de valeurs simples
  - Liste dont les valeurs sont définies par un type simple existant

Exemple

```
<simpleType name="codeOuRef">
  <union>
    <simpleType>
      <restriction base='positiveInteger'>
        <maxInclusive value="65535"/>
      </restriction>
    </simpleType>
    <simpleType name="codeBarre">
      <restriction base="string">
        <pattern value="[0-9]6-[0-9]6"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>

<simpleType name="codeList">
  <list itemType="codeBarre"/>
</simpleType>
```

Grammaire d'arbres régulière : définition

Une grammaire d'arbre régulière est un triplet (NT, TP, R) où NT est un ensemble de non terminaux, TP un ensemble de types primitifs et R un ensemble de règles de la forme :

- $A \rightarrow nomElt\ atts\ (regexp)$
- $A \rightarrow tp$

où :

- $A \in NT$ est un non terminal
- $nomElt$ est un nom d'élément XML
- $atts$ est un ensemble de triplets $(nomAttribut, typePrimitif, card)$
  - $typePrimitif \in TP$ et $card \in \{?, 1\}$
- $regexp$ est une expression rationnelle de non terminaux et de types primitifs
- $tp \in TP$ est un type primitif

Sémantique : principe

- Définie par un ensemble de fonctions $L(G)_A$ où :
  - $G = (NT, TP, R)$ est une grammaire
  - $A \in NT$ est un non terminal
- Définition peut être récursive si les structures définies peuvent être arbitrairement profondes
  - Ex : document décrivant des catégories pouvant être imbriquées
  - Ex : document décrivant des expressions mathématiques
- $L(G)_A$ se lit comme "l'ensemble des suites d'arbres XML reconnus par la grammaire G en partant du non terminal A".

Sémantique : règles

Règle $A \rightarrow nomElt\ atts\ (regexp)$ de G

$L(G)_A$ est l'ensemble des arbres a tels que :

- La racine de a est étiquetée par *nomElt*
- La suite $s = a_1 \ldots a_n$ des fils de la racine est telle que :
  - Il existe n non terminaux $A_1 \ldots A_n$ tels que :
    - $A_1 \ldots A_n$ est reconnu par *regexp*
    - $a_i \in L(G)_{A_i}$
- Tous les attributs de a apparaissent dans *atts*
- Pour tout triplet $(att, tp, 1) \in atts$:
  - a possède un attribut *att* dont la valeur appartient au type *tp*
- Pour tout triplet $(att, tp, ?) \in atts$:
  - **Si** a possède un attribut *att* **alors** sa valeur appartient au type *tp*

Sémantique : règles

Règle $A \rightarrow tp$ de G

$L(G)_A$ est l'ensemble des arbres

- se réduisant à un noeud texte
- dont la valeur appartient au type tp

DTD : Elements

<!ELEMENT *nom contenu*>

- Décrit les suites d'enfants possibles pour un élément.
- *contenu* peut être :
  - EMPTY : pas d'enfant
  - ANY : contenu arbitraire
  - (#PCDATA | nom_1 | nom_2 | ...) : mélange de texte et d'éléments
  - (*expr*) : expression rationnelle de nom d'éléments

$$
\begin{array}{rll}
expr & ::= & expr_1 , expr_2 \\
 & & expr* \\
 & & expr? \\
 & & expr+ \\
 & & expr_1 \mid expr_2 \\
 & & nom \\
 & & (expr)
\end{array}
$$

DTD : Attributs

<!ATTLIST *nom* dec_1 dec_2>

- Décrit les attributs possibles pour un élément
- *dec* peut être :
  - *nom type* "*valeur*"
  - *nom type* #REQUIRED
  - *nom type* #IMPLIED
- *type* peut être :
  - CDATA, ID, IDREF, IDREFS
  - (val_1|val_2|...)

DTD : Exemple

```
<!ELEMENT collection (serie*)>
<!ELEMENT serie (tome+,editeur?)>
<!ATTLIST serie nom CDATA #REQUIRED>
<!ELEMENT tome (scenariste?,dessinateur?,titre)>
<!ATTLIST tome numero CDATA #REQUIRED>
<!ELEMENT scenariste (#PCDATA)>
<!ELEMENT dessinateur (#PCDATA)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT editeur EMPTY>
<!ATTLIST editeur nom CDATA #REQUIRED
                  adresse CDATA #IMPLIED>
```

DTD et grammaires

Exercice : comment, étant donné une DTD, la traduire en grammaire d'arbre ?

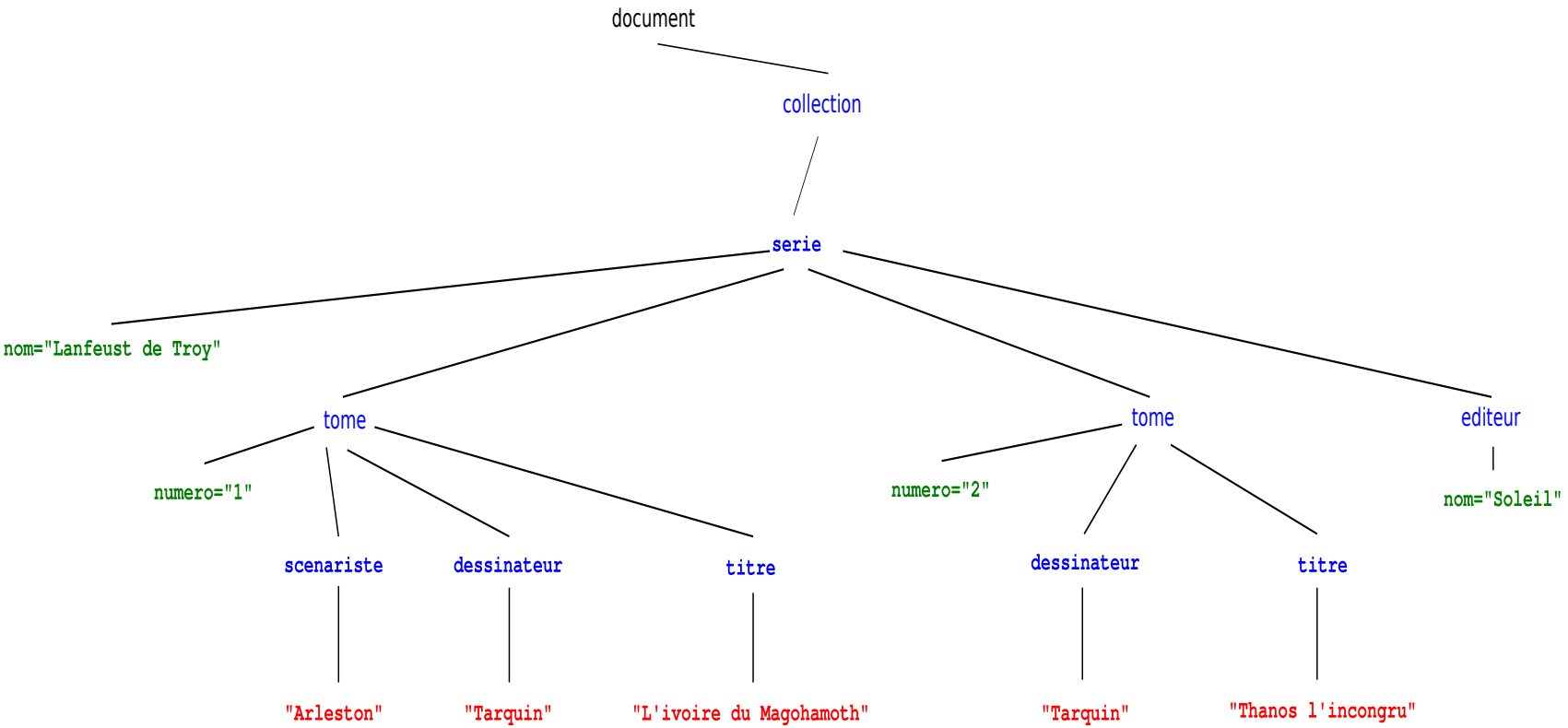
Indice : introduire un non terminal par déclaration `<!ELEMENT`

Exercice : expliquer pourquoi le passage grammaire $\rightarrow$ DTD n'est pas toujours possible

Exemple : comme grammaire d'arbre

$C \rightarrow collection\ \emptyset\ (S*)$
$S \rightarrow serie\ \{(nom, string, 1)\}\ (T+, Ed?)$
$T \rightarrow tome\ \{(numero, string, 1)\}\ (Sc?, D?, Ti)$
$Sc \rightarrow scenariste\ \emptyset\ (St)$
$D \rightarrow dessinateur\ \emptyset\ (St)$
$Ti \rightarrow titre\ \emptyset\ (St)$
$Ed \rightarrow editeur\ \{(nom, string, 1), (adresse, string, ?)\}\ ()$
$St \rightarrow string$

Exemple - arbre



XML Schema : Types complexes

- Type d'un élément :
  - `<element name="nomElt" type="nomType"/>`
  - `<element name="nomElt">`
    ...description du type ...
    `</element>`
  - Le type peut être un type simple
    $\Rightarrow$ contenu textuel/#PCDATA uniquement
  - Ou un type dit "complexe" qui défini les attributs et les éléments pouvant apparaître dans l'élément concerné

XML Schema : types complexes

Ils sont utilisés :

- Comme déclaration principale :

```
<complexType name="nomType">
...
</complexType>
```

- Dans un élément ou un autre type complexe :
  - `<complexType ref="nomType"/>`
  -
```
<complexType>
...
</complexType>
```

Attributs dans un type complexe

`<attribute name="`*nomAtt*`" type="`*type*`" use="`*opt*`"/>`

- *nomAtt* est le nom de l'attribut
- *type* est un type simple qui contraint les valeurs de l'attribut
- *opt* peut être `optional`, `prohibited` ou `required`

Éléments dans un type complexe

- Déclarations similaires à des déclarations externes à un type :
  - `<element name="nomElt" type="typeElt"/>`
  - `<element name="nomElt">`
    `...`
    `</element>`
- Références à d'autres déclarations
  - `<element ref="nomElt"/>`

Combinaisons dans les types complexes

- Suite (, en regexp)
  `<sequence>`
  `...`
  `</sequence>`
- Choix (| en regexp)
  `<choice>`
  `...`
  `</choice>`

Nombre d'occurrences

- Les attributs minOccurs et maxOccurs :
  - Définissent le nombre mini/maxi d'occurrences
  - Applicable sur element, choice, sequence et complexType
  - Valeur si non spécifié : 1
  - maxOccurs peut prendre la valeur `unbounded` si on ne veut pas de nombre maxi d'occurrences
- Permettent de coder les opérateurs de regexp :
  - * : minOccurs="0" maxOccurs="unbounded"
  - + : minOccurs="1" maxOccurs="unbounded"
  - ? : minOccurs="0" maxOccurs="1"
  - n : minOccurs="n" maxOccurs="n"
  - m,n : minOccurs="m" maxOccurs="n"

XML Schema et espaces de nommages - 1

- Syntaxe XML de XML Schema
  - Permet la définition d'espaces de nommage par défaut et/ou de préfixes
- Les éléments propres à XML Schema sont attachés à l'espace de nommage :
  `http://www.w3.org/2001/XMLSchema`
- Les noms définis et utilisés dans XML Schema sont des noms qualifiés
- Les types (simples) prédéfinis dans la norme XMLSchema ont leur noms attachés à l'espace de nommage :
  `http://www.w3.org/2001/XMLSchema`

XML Schema et espaces de nommages - 2

- Attribut targetNamespace
  - Espace de nommage auquel seront attachés les noms (de types) définis dans l'élément portant cette déclaration
    - En général placé sur la racine du schéma
- Attribut elementFormDefault
  - Valeur "unqualified" (par défaut) :
    - Les noms d'éléments définis se voient attachés à un espace de nommage de par leur préfixe uniquement
  - Valeur "qualified" :
    - Les noms d'éléments définis se voient attachés par défaut au targetNamespace
- Attribut attributFormDefault
  - Fonctionnement similaire, mais pour les attributs

Exemple - préambule

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.collection.com"
    xmlns:tns="http://www.collection.com"
    elementFormDefault="qualified">
```

Exemple - élément collection

```
<element name="collection"
         type="tns:collectionC"/>

<complexType name="collectionC">
   <sequence>
      <element name="serie"
                    type="tns:serieC"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
   </sequence>
</complexType>
```

Exemple - contenu d'un tome

```
<complexType name="tomeC">
   <sequence>
      <element name="scenariste" type="string"
               minOccurs="0"/>
      <element name="dessinateur" type="string"
               minOccurs="0"/>
      <element name="titre" type="string"/>
   </sequence>
   <attribute name="numero" type="string" use="required"/>
</complexType>
```

Exemple - contenu d'une serie

```
<complexType name="serieC">
   <sequence>
     <element name="tome" type="tns:tomeC" minOccurs="1"
              maxOccurs="unbounded"/>
     <element name="editeur">
       <complexType>
         <attribute name="nom" type="string"
                    use="required"/>
         <attribute name="adresse" type="string"
                    use="optional"/>
       </complexType>
     </element>
   </sequence>
   <attribute name="nom" type="string" use="required"/>
</complexType>

</schema>
```

XML Schema et grammaires

Exercice : en considérant le sous ensemble du langage XML Schema présenté dans le cours, expliquer comment le traduire en grammaire d'arbre régulière.

Exercice : expliquer comment passer une grammaire d'arbres régulière à un document XML Schema.

Extensions de types complexes

- Principe : reprendre une définition existante et y ajouter des éléments/attributs.
- Exemple : ajouter un élément `annee-parution` au contenu `tomeC`
- Principe similaire à l'héritage entre classes en programmation orientée objet.

Restriction de types complexes

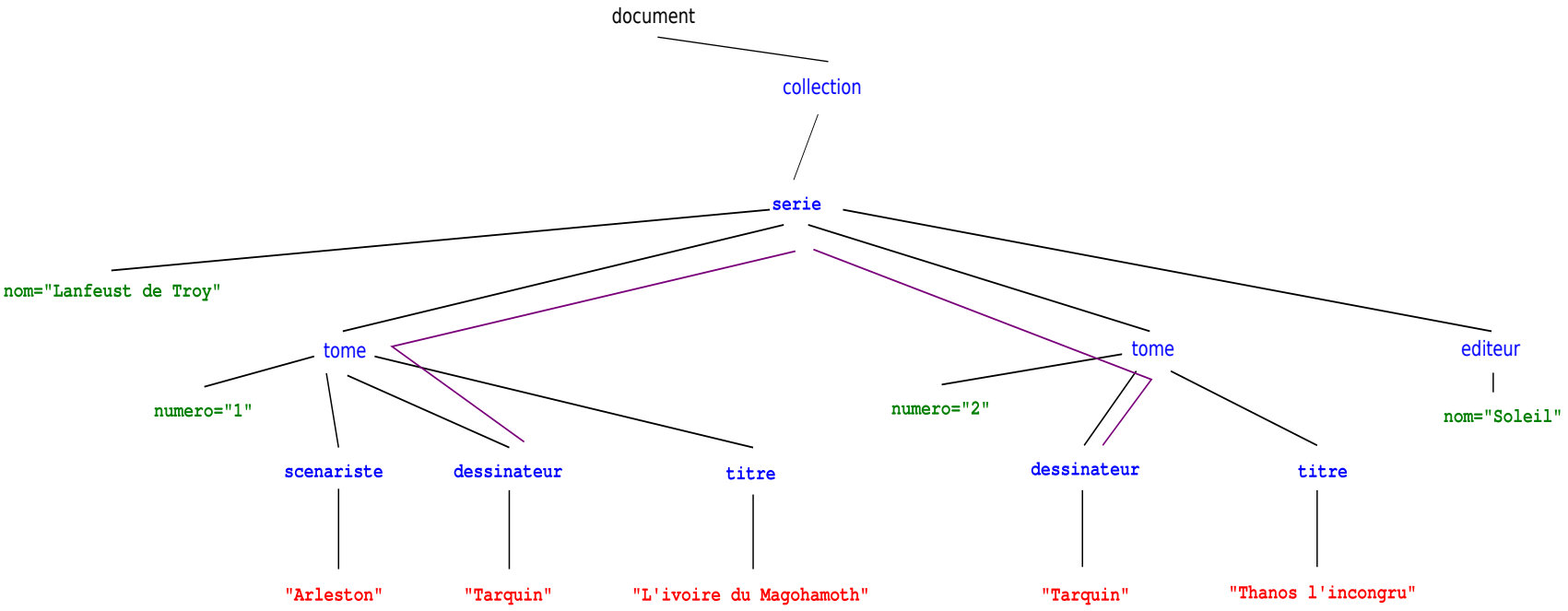
Principe : définir un type comme reconnaissant un sous-ensemble des arbres reconnus par un type précédemment défini

- par exemple en rendant obligatoire un élément qui était auparavant optionnel.
- Si A est défini à partir de B via une restriction, il n'est pas forcément possible de définir $C \equiv A$ par extension sur B
  - L'extension et la restriction ne sont pas exactement opposées

XPath

- Objectif : sélection de morceaux de documents XML
- Utilisé dans d'autre langages
  - XQuery, XSLT, XPointer, WS-BPEL
  - Utilisable via des bibliothèques Java, Python, C, ...
- Principe : spécification de chemins dans l'arbre menant aux morceaux intéressants
  - expression XPath + noeud de départ
    $\rightarrow$ ensemble de chemins dans l'arbre XML
    $\rightarrow$ ensemble de noeuds sélectionnés

Exemple



"Aller sur un élément `tome`, puis sur un élément `dessinateur`"
Evaluer à partir de l'élément `série`

Expressions de chemin

- Suite d'étapes séparée par "/"
- un "/" en début d'expression : départ forcé depuis la racine (document)
- Une étape est de la forme $axe::test[predicat]$
  - Le prédicat est optionnel
- Pour chaque étape, pour chaque noeud n d'ensemble N_d de noeuds de départs :
  - Calculer N_n^a obtenu en suivant l'axe à partir de n
  - Calculer N_n^t en filtrant N_n^a via le test
  - Calculer N_n^p en filtrant N_n^t via le predicat
- Résultat de l'évaluation de l'étape : $\bigcup_{n \in N_d} N_n^p$

Expressions booléennes

- Expressions classiques :
  - `and`, `or`, `not(...)`
  - fonctions renvoyant un booléen
- Nombre n : seul le n-ième élément de N_n^t est conservé
- Expression de chemin :
  - évaluation à partir du noeud à tester ;
  - vrai si résultat non vide
- Si une expression de chemin apparaît comme argument d'une fonction/d'un opérateur non booléen :
  - Evaluer l'expression à partir du noeud à tester ;
  - la (sous) expression booléenne est vraie si une des valeurs obtenues rend l'expression booléenne vraie

Abréviations

$$\texttt{child::}\mathit{test} \quad \leftrightarrow \quad \mathit{test}$$

$$\texttt{attribute::}\mathit{test} \quad \leftrightarrow \quad \texttt{@}\mathit{test}$$

$$\texttt{xxx/descendant-or-self::node()/yyy} \quad \leftrightarrow \quad \texttt{xxx//yyy}$$

$$\texttt{parent::node()/xxx} \quad \leftrightarrow \quad \texttt{../xxx}$$

$$\texttt{axe::test[(}\mathtt{pr}_1\texttt{) and (}\mathtt{pr}_2\texttt{)]} \quad \leftrightarrow \quad \texttt{axe::test[}\mathtt{pr}_1\texttt{][}\mathtt{pr}_2\texttt{]}$$

Exemples

Revoir la DTD “collection”.
Donner une expression XPath pour obtenir :

- le premier tome de la collection dans chaque série
  `//serie/tome[1]`

  i.e. `/descendant-or-self::node()/`
  `child::serie/child::tome[pos()=1]`

- l'ensemble des titres d'album (sans la balise titre)
  `//tome/titre/text()`

Exemples - suite

- les séries dont on connaît l'éditeur
  `//serie[editeur]`

- les séries dont on possède le tome numéro 1
  `//serie[tome/@numero = 1]`

- le titre des albums dont le numéro est plus grand ou égal à 3
  `//tome[@numero >= 3]/titre`

Expressions avancées : parenthèses

Parenthèses

- La partie *axe*::*test* peut être remplacée par une expression entre parenthèses
  - on peut appliquer un prédicat sur le résultat
  - important pour les prédicats type *n*-ième
- Exemple : le troisième tome de la collection : `/collection/(serie/tome)[3]`

Expressions avancées : fonctions

Fonctions prenant et ou renvoyant des ensembles de noeuds

- L'appel à la fonction est :
  - utilisé dans un prédicat
  - le point de départ d'un expression de chemin
    - remplace la première étape
    - voir l'expression complète
- Exemple : Les séries également présentes dans collection2.xml :
  `//serie[@nom=document('collection2.xml')//serie/@nom]`

XQuery

- Langage de requête pour les documents XML
  - Utilisé en particulier dans les BD XML
- Fabrique des (morceaux de) documents XML à partir de documents XML
- Une expression XPath est une expression XQuery
- Permet de construire des morceaux de document : syntaxe XML + expressions XQuery entre accolades

Exemple

```
<personnes>
  <scenaristes>
    {//scenariste}
  </scenaristes>
  <dessinateurs>
    {//dessinateur}
  </dessinateurs>
</personnes>
```

FLWOR

for v1 in e_1, v2 in e_2, ...
let w1 := e'_1, w2 := e'_2, ...
where *condition*
order by eo_1, eo_2, ...return *expr*

- vi, wi : variables
- e_i, e'_i : expressions XPath
  - une variable peut remplacer la première étape d'un chemin
- eo_i : expression XPath (avec variables), suivie de `ascending` (par défaut) ou de `descending`
- *expr* : expression XQuery (contenant en général des constructions XML)

FLWOR : Evaluation

- Evaluer les combinaisons de valeurs possibles pour les $\$v_i$
  - On obtient un ensemble de tuples de valeurs
- Pour chaque tuple :
  - Evaluer les $\$w_j$
    - Si plusieurs valeurs pour une variable : elles sont concaténées
    - Les valeurs sont associées au tuple
- Filtrer les tuples avec la condition
- Pour chaque tuple, pris dans l'ordre de la clause order by, évaluer expr
  - Le résultat du FLWOR est la concaténation des résultats ainsi obtenus

Exemple

```
for $to in //tome
let $ti := $to/titre
where $to/@numero >= 3
order by $ti descending
return
<album>
  {$to/@numero}
  {$ti}
  <serie>{$to/../@nom}</serie>
</album>
```

Déclarations

- Précède l'expression (i.e. mettre au début du programme)
- `declare namespace` *`nomprefixe`*`="`*`uri_espace_nommage`*`";`
- `declare default element namespace "`*`uri_espace_nommage`*`";`
- `declare function` *`nomQualifie`* `(`*`$arg1`* `as` *`type1`*`, `*`$arg2`* `as` *`type2`*`, ...) as` *`type_retour`*
  `{` *`corps de la fonction`* `};`
- `declare default function namespace "`*`uri_espace_nommage`*`";`

JavaScript Object Notation

Format de données

- textuelles
- modèle de données en arbre
  - noeuds « objets » → dictionnaire clé/valeur
  - noeuds listes

Exemple JSON : collection

```
[
    { "nom": "Lanfeust de Troy",
      "tomes": [
          { "numero": "1",
            "scenariste": "Arleston",
            "dessinateur": "Tarquin",
            "titre": "L'ivoire du Magohamoth" },
          { "numero": "2",
            "dessinateur": "Tarquin",
            "titre": "Thanos l'incongru" }
      ],
      "editeur": { "nom": "Soleil" }
    }
]
```

Schemas & interrogation

Pas de norme bien établie

Schéma :

- Problématique similaire à XML Schema
- Regarder par exemple : `http://json-schema.org/`

Interrogation : similaire à XQuery

- `http://jsoniq.org/`
- `https://github.com/mmckegg/json-query`