INRIA LORRAINE
EQUIPE PROTHEO
615 rue du Jardin Botanique
54600 Villers-lès-Nancy

Rapport de stage

TOM et XQuery

Réalisateur: Phan Nghiem Long
Tuteur: Pierre-Etienne Moreau

Octobre 2004

Index

TOM et XQuery

1. Le modèle de donnée de XQuery

Tout les modèles de donnée de XQuery sont les séquences (Sequence) d'un ou plusieurs éléments (Item). Le résultat d'une requête XQuery est toujours une séquence.
Les éléments sont toujours singletons. Un élément corresponde à une séquence ayant la longueur de 1. Une séquence peut être vide (ne contient aucun élément), mais elle ne peut pas contenir d'autre séquence. (ex : une séquence (0,1,(2,2)) est considérée comme (0,1,2,2).
Chaque élément peut être, soit un atome (Atom), soit un nœud (Node). Un atome est un singleton qui ne contient aucune structure dedans. (XQuery défini 50 types d'atomes différents).
Les nœuds sont les structures ayant plusieurs propriétés : type, nom et genre. Il y a 7 genre de nœud dans XQuery (comme XML). Un nœud de XQuery peut être considérés comme un nœud XML.

2. Document XML dans XQuery

2.1 Ordre de document

l'order de document XML est l'ordre parmi les noeuds dans un ou plusieurs documents XML. En général, on peut considérer l'ordre des noeuds dans un document comme l'ordre retourné par le travers infixe, depth-first dans le document.

- Le noeud document est le premier noeud
- Les noeuds éléments précèdent leurs enfants, et un noeud enfant précède les noeuds "sibling" qui suivent.
- le noeud N1 est précède le noeud N2 si et seulement si le parent du noeud N1 précède celui du noeud N2.

Pour déterminer l'ordre des noeuds XML, XQuery fournie deux opérateurs booléens: << (devant) et >> (après). L'opérateur << acceptent deux noeuds n1 et n2, il retourne true si n1 précèdent n2, si non il retourne false.

2.2 Atomization

la sémantique des opérateurs dans XQuery dépendent de l'atomization. L'atomization est appliquée à une valeur quand elle est utilisée dans un contexte où il faut avoir une séquence d'atome. La résultat de l'atomization est une séquence de valeur atomique.

- Si l'élément est un atome, le résultat est cet atome
- Si l'élément est un noeud, le résultat est la valeur typé.

3. Une requête XQuery

3.1 Introduction

Tandis que la partie fondamentale d'une requête XQuery est toujours l'expression, mais en effet, XQuery est utilisé pour traiter les documents XML, qui est une séquence des records de donnée en général, donc on peut considérer FLWOR est la partie principale d'une requête XQuery.
La structure FLWOR (on l'appelle « Flower » en anglais) se compose de 5 clauses : *for*, *let*, *where*, *order by*, et *return*. Chaque FLWOR commence par un ou plusieurs *for* et/ou *let* (dans n'importe quel ordre), se suivent une clause optionnelle *where*, et une clause optionnelle *order by*, et dernièrement une clause *return,* pour retourner les données attendues.

```
For
Let
Where
Order by
Return
```

Les FLWORs peuvent "nested":

```
For
Let
Where
Order by
Return
      For
      Let
      Where
      Order by
      Return
```

3.2 Clause let

La clause *let* est utilisé pour déclarer les variables, et puis, assigner des valeurs quelconque à ces variables. Par exemple, on déclare deux variables $x et $y:

```
let $x := 1
let $number := "string"
let $y := $x
```

Les variables déclarées sont accessibles dans le reste du FLWOR, et aussi dans les "nested" FLWORs.

3.3 Clause for

La clause *For* est utilisé pour traverser séquentiellement des éléments dans une séquence. Normalement, dans une clause *for* on introduit un variable, qui contient la valeur de chaque élément dans chaque boucle. Par exemple :

```
For $i in (0, 1, 2, 3)
For $s in doc("book.xml")/book/section
```

3.4 Clause where et order by

Ces clauses sont utilisées pour vérifier les variables, et pour trier les données dans la sortie. Elles correspondent aux clauses *where* et *order by* de SQL.

3.5 Clause return

Cette clause est utilisée pour écrire les résultats à la sortie.

Exemples:

requête	*résultat*
`For $s in (0,3,2,1)` `return $s`	0 3 2 1

`Let $sequence := (0, 3, 2, 1)`
`For $i in $sequence`
`order by $i`
`return $i`
	0 1 2 3

`For $s in "string"` `return $s`	string *car l'élément "string" peut être considéré comme une séquence ("string")*
`For $s in "sequence"` `return $s`	sequence
`For $s in (0, "string", 2, 3)` `return $s`	0 string 2 3

`For $s in (0, "string", 2, 3)`
`order by $s`
`return $s`
	Error
	car on ne peut pas comparer les numéros et les chaines de caractère.

3.6 Les options de FLWOR:

Dans la structure FLWOR on peut utiliser quelques mots clé pour modifier le résultat, ou ajouter quelques choses:

3.6.1 "as"

C'est l'option de déclaration de type, pour indiquer le type de la variable dans l'itération. "as" peut être utilisé dans "let" et "for"

Ex:

```
for $i as xs:integer in (1,2,3,"string")
return $i
```

Dans cet exemple la variable $i doit accepter des valeurs integer (défini dans XML Schema), et on va obtenir une erreur puisque "string" est de type xs:string.

3.6.1 "at"

Chaque variable dans l'itération peut avoir une variable accompagnant, c'est la variable "position". Pour déclarer cet variable on utilise le mot clé "at"

ex:

```
for $number at $i in (1,2,3,"string")
return $i
```

et $i va être assignée les valeurs 1,2,3,4 séquentiellement.

4. La chemin de XQuery

Le principale but de XQuery est de traiter les documents XML. En effet, XQuery utilise la chemin de XPath qui permet aux developpeurs de naviguer dans les arbres DOMs des documents XML. XQuery suit la chemin pour retraire les informations stoquées dans les noeuds DOM. Les chemins XQuery sont utilisées partout dans les requêtes XQuery.
Une chemin XQuery contient plusieurs pas: les pas sont séparés par l'opérator de chemin slash "/"

```
$book/section/section
/--pas--/
/------ chemin------/
```

Exemple:

`$book/section`	accéder a des noeuds "section" sous le noeud qui est nommé $book
`$book/section/section`	accéder a des noeuds "section" sous des noeuds "section" sous le noeud nommé $book
`document("book.xml")/ book/section`	accéder a des noeuds "section" sous des noeuds "book" sous la racine du document XML book.xml

4.1 Départ d'une chemin

Chaque chemin commence à partir d'une place quelconque. Il y a trois places où les chemins peuvent commencer:

- le noeud courant (dans le contexte)
- la racine de l'arbre où le noeud courant se trouve
- l'ensemble de noeud: comme les variables ou les contructeurs XML.

Après chaque pas, le chemin peut aller aux autres noeuds ou changer le noeud courant.
La racine de l'arbre où le noeud courant se trouve peut être obtenu en utilisant l'opérateur de chemin "/".
ex:

```
let $s := (1,2)
for $i in <book><section>{$s}</section></book>
return $s/section
```

4.2 Pas

Un pas se compose de trois parties: l'axis, NodeTest, et les prédicats.

```
$book/[child::]sectionsection [1]       [1=2]
     /-axis-::---NodeTest--- /predicate/predicate
```

4.2.1: Axis

L'axis est utilisé pour indiquer la direction de la chemin que XQuery doit suivre. Les implémentations de XQuery doivent supporter au moins 6 axes suivants:

nom	*abbréviation*	*exemple*	*exemple*
`attribute`	`@`	`x/attribute::y`	`x/@y`

`parent`	`..`	`x/parent::y`	`x/..`
`child`		`x/child::y`	`x/y`
`self`	`..`	`x/self::y`	`x/.`
`descendant`		`x/descendent::y`	
`descendant-or-self`	`//`	`x/descendent-or-self::y`	`x//y`

Dans un pas, l'axis est suivi par le signe "::" pour distinguer l'axis et le nom XML qualifié.
En réalité, l'axis "child" est utilisé le plus souvent donc "child" est la valeur par défault de l'axis.
(L'axis descendant-or-self (//) est toujours considéré comme descendant dans quelques implémentation de XQuery, comme Galax)

4.2.2 NodeTest

Il y a trois types différents de NodeTest: nomtest, nodekindtest, et wildcardtest.

- *Nametest*: le plus utilisé. Un nomtest sélectionne les noeuds qui ont le même nom. Le nom peut être qualifié ou non.
  Ex: `$book/section`
- *Nodekindtest*: la raison d'existance de nodekindtest est qu'il y a des noeuds XML n'ayant pas de nom, comme textNode, commentNode, ou documentNode. Il y a sept nodekindtest:

`Document-node()`	Selectionne le noeud de type document
`Element()`	Les éléments
`Comment()`	Les noeuds commentaires
`Attribute()`	Les attributes
`Text()`	Les noeuds texte
`Node()`	n'import quel noeud
`Processing-instruction()`	Les noeuds processing intruction

  Ex: $book/node() // retourne n'importe quel noeud qui est le son direct de $book
- *Wildcardtest:* utilisé pour sélectionner les noeuds ayant le même nom dans un namespace, ou ayant le même nom locale, sans regardant le namespace .... Il y a deux méthodes pour le faire: utiliser les prédicats, ou utiliser wildcardtest.

XQuery supporte trois types de wildcardtest:

`*`	n'importe quel noeud
`préfix:*`	n'importe quel noeud dans un namespace "préfix"
`*:local-name`	n'importe quel noeud, sans regardant le nom du namespace

4.2.3 Précicat

La dernière partie d'un pas est un ou plusieurs prédicat. Les prédicats sont utilisés pour filtre les noeuds sélectionnés par le nodetest, en utilisant quelques branches conditionnelles. Pour chaque noeud sélectionné par nodetest, le contexte courant est changé à ce noeud, et la condition de chaque prédicat est calculée basé sur ce contexte.

N'importe quelle expression XQuery peut être utilisé dans un prédicat, et le sens de ce prédicat dépend du type de l'expression calculée. Il y a deux sens possibles, cela veut dire qu'il y a deux types différents de prédicat: prédicat numérique et prédicate booléen.

- Prédicat numérique: (quand le type de l'expression est numérique) Ce type de prédicate sélectionne les noeuds en regardant leur positions dans le contexte courant. Par exemple la chemin /book/section[1] sélectionne le premier noeud "section" de chaqu'élément "book".

Le prédicat numérique peut être appliqué à une chemin entière (avec l'aide des parenthèses):

`$book/section/section [1]`	Sélectionne le premier noeud "section" de chaqu'élément "section". Supposons que le noeuds nommé "$book" a plusieurs child-noeuds "section", cette requête peut retourner **plusieurs noeuds** comme résultat.
`$book/ (section/section)[1]`	Sélectionne tous les noeuds "section" de chaqu'élément "section" de haut niveau. Et après, sélectionne un seul noeud parmi les noeud sélectionnés avant. Supposons que le noeuds nommé "$book" a plusieurs child-noeuds "section", cette requête peut retourner **un seul noeud** comme résultat.

- Prédicat booléen: représente les autres genres de l'expression. La valeur de l'expression dans le prédicat est convertie à une valeur booléenne. Il y a une convention générale pour convertir les éléments ou les séquences aux valeurs booléennes.

  Par exemple:

```
/x[@a=1 and @b=1]:
```

(retourne les noeuds commés "x" ayant une attribute a égale à 1, et une attribute b égale à 1)

En effet, les prédicats numériques sont des cas spéciaux des prédicats booléens. Par exemple, le prédicat numérique:

```
/x[@y + 1]
```

corresponde au prédicat booléen:

```
/x[position() = @y + 1]
```

(dont la "position()" est une primitive de XQuery)

4.3 les fonctions de navigation

Les navigations ci-dessous sont associées aux pas locaux: à partir du contexte courant, on navigate vers quelques noeuds à côté. XQuery fourni aussi des fonctions qui nous permettent de naviguer vers les différentes parties du document ou même l'autre document.

`Collection()`	Naviguer vers une séquence nommée
`Doc()`	Naviguer vers la racine d'un document XML nommé
`Id()`	Naviguer vers un noeud unique avec l'ID
`Idref()`	Naviguer vers les éléments qui réferent à ce noeud
`Root()`	Naviguer vers la racine du document

5. Constructeur XML

XQuery nous permit de construire n'importe quel genre de noeud XML. Deux types de constructor sont fournis: les constructeurs directs (DirElementConstructor) (utilisent la notation XML) et les constructeurs calculés (ComputedConstructor), qui utilisent la notation basée sur les expressions. (il existe aussi d'autre constructeur: XmlComment, XmlPI, CdataSection, et CompAttrConstructor

```
Constructor := DirElementConstructor
            | ComputedConstructor
            | XmlComment
            | XmlPI
            | CdataSection
            | CompAttrConstructor
```

5.1 DirElementConstructor

Un constructeur directe crée un élément XML.
Ex:

```
<book>
  <section id="001">
    <title>Introduction</title>
  </section>
</book>
```

Les braces "{" et "}" peuvent être utilisés pour délimiter des expressions. Les expressions sont évaluées et sont remplacées par leurs valeurs. Par exemple, la requête suivant retourne le même noeud XML:

```
let $Id:="001"
let $Title:="Introduction"
return
  <book>
    <section id="{$Id}">
      <title>{$Title}</title>
    </section>
  </book>
```

(pour représenter les braces { et }, on utilise la notation {{ et }} correspondant)
Le résultat d'un constructeur est un nouveau noeud, tout attribut et tout noeud descendant sont aussi nouveaux.
L'existance des espaces dans la notation peut être controlé par une option de XQuery.

5.2 Constructeur calculé

```
ComputedConstructor := CompDocConstructor
            | CompElementConstructor
            | CompAttrConstructor
            | CompTextConstructor
            | CompXmlPI
            | CompXmlComment
            | CompNSConstructor
```

Un constructeur calculé commence par un mot clé qui indique le genre du noeud sera contruit: element, document, text, processing-instruction, comment, namespace
par exemple:

```
element book {
      element section {
            attribute id {"001"}
            element title {"Introduction"}
      }
}
```

5.3 Autres constructeurs

XQuery nous permit de construire directement les noeuds de types CDATA, comment, processing-instruction, et attribute.

6. Le support de TOM pour XML

6.1 Constructeur XML

Dans TOM, on peut utiliser "`xml(...)" pour construire un noeud XML.
Par exemple:

```
`xml(<doc id="100"><author></author></doc>)
```

Mais avec `xml, on ne peut que construire les noeuds element, on doit utiliser les constructeur algébraique.
Ex:

```
`CommentNode(documentNode, "comment here")
```

6.2 Pattern pour %match

Pour représenter un noeud XML, on peut utiliser la notation algébraique basée sur TNode et TNodeList, qui est correspondent à Node et NodeList de DOM.

```
%match (TNode node) {
nodeBook@<book>
nodeSection@<section id=value1 note="abcd">
<title>#TEXT("Introduction")<title>
<content>
#TEXT(ElementContent)
</content>
</section>
</book> -> {
      ...
}
}
```

On constate que la notation algébraique ci-dessus est très familiar pour les programmeurs XML. Cette notation peut "match" un noeud XML suivant:

```
<book>
  ...
  <section id="001" note="abcd">
    ...
    <title>Introduction</title>
    ...
    <content>
      // ... content
    </content>
    ...
  </section>
  ...
</book>
```

De plus, on peut initialiser les variables différentes, comme avec les patterns normaux dans %match. Dans cet exemple la variable nodeBook va contenir la valeur du noeud "book", la variable "value1" va contenir la valeur string "001".
Mais il existe un sérieux problème de TOM, c'est que TOM ne support pas namespace dans XML.

7. Implémenter les requêtes XQuery dans TOM et Java

7.1 TNode ou DOM

Au début on doit faire le choix entre TNode ou DOM.
On peut constater que à partir un noeud quelconque, on doit pouvoir monter vers son parent (ou même vers la racine du document), descendre aux ses enfants. Avec TNode c'est impossible d'obtenir la référence du parent d'un noeud.
De plus, si on utilise TNode, le partage maximal est utilisé. Le partage maximal est bon pour la gestion de mémoire, le temp de création des instances. Mais pour but de gérer les résultats retournés par XQuery, c'est difficile. Je donne ici un exemple:

```
<book>
  <section>
    <title>Audience</title>
  </section>
  <section>
    <title>Intro</title>
  </section>
  <section>
    <title>Audience</title>
  </section>
</book>
```

Supposons qu'on a un document "book.xml" au-dessus, attention qu'il y a deux sections ayant le title "Audience". Ces deux sections sont les mêmes.
Et si on a une requête:

```
for $s in doc("book.xml")//section
return $s
```

Normalement on doit avoir trois records dans le queryrecordset final. Mais avec le TNode, le partage maximal nous donne seulement deux instances dans la mémoire.

- *Solution:* On peur le résoudre en utilisant un vecteur pour stocker les résultats.

Mais dans l'exemple suivant, on verra la restricte de TNode:

```
<book>
  <section>
    <title>Intro</title>
    <section>
      <title>Audience</title>
    </section>
  </section>
  <section>
    <title>Audience</title>
  </section>
</book>
```

Avec la requête:

```
for $s in doc("book.xml")//section
  for $t in $s//title
return $t
```

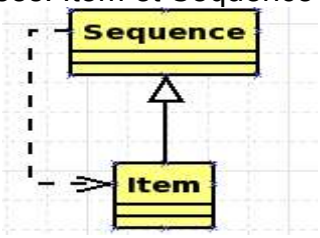
Avec le premier boucle, $s contient trois sections. Et $t contient seulement trois titles.
Si on utilise Java vector pour contenir les noeuds, $s doit contenir 3 sections, et $t doit contenir 4 titles.

- *Solution:* developper une classe hash qui accept le clé est la référence vers un objet dans la mémoire, et la valeur correspondante est le nombre de référence vers cet objet. Cela peut bien marcher, mais c'est difficile à conserver l'order des noeuds retournés comme résultat.

De plus, c'est dur de mettre à jour TNode pour qu'on puisse monter vers le parent à partir d'un noeud, car si on le faire on doit modifier la base de Tom, Pour les raisons ci-dessus, je pense que c'est mieux si l'on utilise DOM.

7.2 Structure de donnée:

Lorsque les modèles de donnée de XQuery sont l'élément et la séquence, donc on doit avoir deux classes: Item et Sequence



La classe Item hérite la classe Sequence puisque un Item est aussi une séquence de longueur 1.

7.3 la requête

Comme la structure FLWOR est la partie principale d'une requête XQuery, donc je doit structurer le programme TOM/Java correspond à la structure FLWOR.
La structure FLWOR est décomposé en quatre parties, la partie FL (for let), la partie W(here), la partie O(rder) et la partie R(eturn). Le partie FL collecte les recordes de donnée, la partie W élimine les records qui ne conviennent pas, et puis, la partie Order trie ces recordes si nécessaire, et la partie Return retourne ces recordes vers la sortie.

```
class XQuery1 {
  QueryRecordSet _FLW01() {}
  QueryRecordSet _Where01(QueryRecordSet input) {}
  QueryRecordSet _Order01(QueryRecordSet input) {}
  void _Return01(QueryRecordSet result) {}

  void _FLWOR01()
  {
      QueryRecordSet result;
      result = _FL01(result);
      result = _Where01(result);
      result = _Order01(result);
      _Return01(result);
  }
}
```

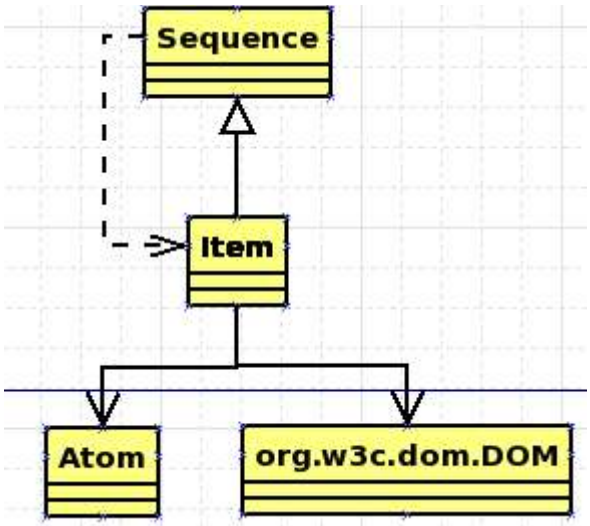
Les fonctions FLWOR, FL, Where, Order, Return sont numérotées, par exemple _FL01, ...
Dans la classe si-dessus, on peut voir les quatre fonctions _FL01, _Where01, _Order01, et _Return01 qui correspondent aux trois parties FL, Where, Order, et Return.
QueryRecordSet est sub-classe de la classe Sequence, elle contient les records de résultat. Ces records sont de type QueryRecord qui est un ensemble des variables utilisées dans ce FLWOR.
Parex: Si on déclare deux variable $i et $j dans un FLWOR, chaque valeur de $i et la valeur correspondente de $j sont stockées dans une instance de QueryRecord. On doit le faire pour pouvoir trier le résultat.

- *Règle: pour chaque variable déclarée en utilisant les mots clés for et let, la classe correspondant doit déclarer une variable de type Item.*

Ex:

```
      for $i in (1,2,3,4)
==>
      class XQuery1 {
        Item _i01 = null;
      }
```

La classe Item est une classe "wrapper", elle peut contenir un noeud XML, ou un atome.



- *Règle: pour chaque document XML, on a une variable correspondante.*

```
      Let $s := document("a.xml")
==>
      class XQuery1 {
        Document _xmlDocument01 = null;
      }
```

7.4 la fonction _FLxx

On peut considérer un boucle:

```
for $i in (une séquence ici)
return $i
```

comme:

```
let $iSequence := (une séquence ici)
for $i in $iSequence
return $i
```

Alors la fonction _FLW01 devient:

```
QueryRecordSet _FLW01(QueryRecordSet result)
{
  Sequence iSequence = _FLW01_CollectData01();
  // utilise itération pour traversers iSequence
  // ajouter $i, et les autres variables dans un record
  // ajouter ce record dans le queryrecordset
  // return
}
```

Le problème est la fonction _FLW01_CollectData01(), elle correspond à une chemin de XQuery. A partir des variables de membre de la classe, cette fonction suit la chemin pour extraire les noeuds souhaités.
On peut suivre les règles ci-dessous pour pourvoir écrire une fonction correspond à une chemin de XQuery.

```
pathexpr            := expr steps
steps               := (normalstep |  slashslashstep | parenthesestep)
*
normalstep          := axis nodetest (predicate)*
slashslashstep      := axis nodetest (predicate)*
parenthesestep      := steps (predicate)*
nodetest            := nametest | kindtest
axis                := self | parent | child | attribute | descendant-
or-self
predicate           := booleanpredicate | numericpredicate
booleanpredicate    := expr
numericpredicate    := expr
```

Une chemin contient une expression au début, et plusieurs pas suivant (steps). Un pas peut être un pas normal (avec "/"), ou un pas slashslash (avec "//"), ou un pas parenthese (avec "(" et ")"). Un pas parenthese peut contenir plusieurs pas.
Ex:

```
$book/section/section
```

peut être décomposé en:

```
(expr  normalstep  normalstep)
```

et:

```
$book/section//section
```

peut être décomposé en:

```
(expr  normalstep  slashslashstep)
```

et:

```
$book/(section/section)[1]
```

peut être décomposé en:

```
(expr       parenthesestep)
```

```
      => (expr (normalstep normalstep) predicate)
```

Alors on peut imaginer la fonction _FLWxx_CollectData01 comme suit: (la fonction evaluateInitialValue est utilisé pour calculer la valeur de l'expression au début du pathexpr.

```
Sequence _FLWxx_CollectDataxx ()
{
  Sequence result;
  result = _FLWxx_getInitialValue();
  ...
  result = _FLWxx_SlashSlashStepxx(result);
  ...
  result = _FLWxx_NormalStepxx(result);
  ...
  return result;
}
```

7.4.1 normalstep

Un pas normal (normalstep) est implémenté comme une fonction dans la classe Tom/Java: _FLWxx_NormalStepxx() , ces fonctions sont numérotées comme les fonctions FLWOR. Cette foncstion accepte une séquence des éléments, et retourne aussi une autre séquence d'élément comme le résultat.

Sequence _FLWxx_NormalStepxx(Sequence input) {}

Comme les supporteurs de Tom, je souhaite d'implementer cette fonctions comme une operateur %match:

```
Sequence result = new Sequence()
// itération de la séquence input
for (int i=0; i<input.size(); i++) {
  Node node = input.getItem(i);
  %match (TNode node) {
    <matching partern> -> {
      //dosoming here
     }
  }
  return result;
}
```

Comme dans la spécification de XQuery, un pas se compose de trois partie: l'axis, le nodetest, et une liste des prédicats.

7.4.1.1 axis

Pour l'axis child et self, c'est facile. On peut considérer un %match comme suit:

// pour child

```
      %match (TNode node) {
        <_><childname></childname></_>
      }
```

// pour self

```
      %match (TNode node) {
        <selfname></selfname>
      }
```

Pour l'attribut, il existe aussi des notations en Tom

```
      %match (TNode node) {
        <_ attrname="attrvalue"></_>
      }
```

Pour le parent, il faut obtenir la référence du noeud parent avant de faire un % match.

7.4.1.2 nodetest

Pour le nodekindtest, il n'y a pas de problème: on peut facilement obtenir la genre de chaque noeud Node.

Pour le nodenametest, en combinaison avec l'axes parent, child, et self, c'est facile d'implémenter en Tom.

Voilà la notation de Tom pour les attributs Xml:

```
XMLAttribute ::=  _*
            |VariableName *
            |AttributeName=[AnnotedName @] (Identifier|String)
            |[AnnotedName@]_=[AnnotedName @](Identifier|String )
```

La notation VariableName* maintenant nous donne une Java arraylist qui peut être bien utilisée dans le constructeur `xml:

```
      `xml(<A VariableName/>)
```

Avec la notation annotedName1@_=annotedName2, il doit bien itérer les attributs du noeud examiné: annotedName1 doit être de type Node, qui contient la référence à un des attribut, et la variable annotedName2 doit avoir la valeur string de cet attribut, mais maintenant ca ne marche pas**.** On ne peut pas utiliser annotedName1, mais annotedName2 est possible.

Pour les trois types de wildcard, maintenant c'est impossible de l'implémenter en utilisant %match.

7.4.1.3 les predicats

Les prédicats numériques peuvent être aidés par un compteur "counter".

On doit implémenter une fonction pour chaque prédicat, car chaque prédicat est une expression:

```
Sequence _FLWxx_NormalStepxx_Predicatxx(Node node, int position){}
```

La classe Sequence doit avoir deux fonctions getBooleanValue et getDecimalValue pour obtenir les valeurs correspondant. Elle doit aussi avoir deux fonctions isBooleanValue et isDecimalValue pour déterminer le type du résultat retourné.

On a aussi une fonction _FLWxx_NormalStepxx_PredicatList qui accepte un noeud Node, une paramètre de type int (la position du noeud dans la liste des noeuds acceptés par le nodetest) et retourne une valeur booléenne pour indiquer si le node courant soit acceptable ou non.

```
boolean Sequence _FLWxx_NormalStepxx_Predicatxx(Node node, int
position){}
```

Et la fonction _FLWxx_NormalStepxx devient:

```
Sequence result = new Sequence()
// itération de la séquence input
for (int i=0; i<input.size(); i++) {
  Node node = input.getItem(i);
  int counter = 0;
  %match (TNode node) {
    pattern -> {
      counter ++;
      boolean acceptable= _FLW01_NormalStep01_Predicatelist(node,
```

```
counter);
      if (acceptable) { // OK, matched
        result.add(node);
      }
    }
  }
  return result;
}
```

Les fonctions _FLWxx_NormalStepxx_Predicatexx utilisées dans la fonction _FLWxx_NormalStepxx_PredicateList peuvent contenir plusieurs structures FLWOR donc il peuvent être implémenter récursivement.

```
Boolean _FLWxx_NormalStepxx_PredicateList (Node node, int position)
{
  ...
  if (!_FLWxx_NormalStepxx_Predicatexx(node, position)) {
    return false;
  }
  ...
  return true;
}
```

7.4.2 slashslashstep

On implémente une fonction _FLWxx_SlashSlashStepxx:

```
Sequence _FLWxx_SlashSlashStepxx(Sequence input) {}
```

C'est plus difficile que dans le cas de normalstep. On doit bien implémenter une fonction collect qui traverse les noeuds input (avec l'aide du GenericTraversal) pour collecter les résultats. Biensur qu'on doit mettre à jour GenericTraversal pour pouvoir traiter DOM.

```
Sequence _FLWxx_SlashSlashStepxx(Sequence input) {
  Sequence result = new Sequence()
  // itération de la séquence input
  for (int i=0; i<input.size(); i++) {
    Node node = input.getItem(i);
    Sequence s = _FLWxx_SlashSlashStepxx_Collect(Node);
    result.add(s);
    return result;
  }
}
```

Et la fonction collec est comme suit.

```
Sequence _FLWxx_SlashSlashStepxx_Collect(Sequence input) {
{
  NodeTraversal traversal = new GenericTraversal();
  final Sequence result=new Sequence();
  Collect2 collect = new Collect2() {  // use one argument
    int counter=0;
    public boolean apply(Object t,Object arg1) {
      if(t instanceof Node) {
        Node anode = (Node)t;
        if (t==arg1) { // t is the root, pass
          return true;
        }
        %match (anode) {
```

```
          matchpattern -> {
          counter++;
          if (_FLWxx_SlashSlashStepxx_PredicateList(anode, counter))
          {
            result.add(anode);
            return true; // continue
          }
          _ -> {  // no match, rien faire
            return true;
          }
        } // match
      }
      else {
        return true;  // continue
      }
    } // end apply
  }; // end new

  traversal.genericCollect(node, collect, node);  // node is the root
  return result;
}
```

Cette fonction marche comme un exemple normal du GenericTraversal. Faite attention que le paramètre arg1 de la fonction apply() est la racine (le noeud doit être traversée). On utilise ce paramètre pour éliminer la racine comme un élément dans la séquence des résultats.
Un autre problème, c'est que dans la séquence result, on peut avoir la duplication des même noeuds, donc il faut éliminer les résultats existent deux fois avant de le retourner.

```
      SequenceTool st = new SequenceTool();
      st.removeDuplicated(result);
```

7.4.3 parenthesestep

Ce pas est presque comme pathexpr: il a un séquence d'entrée comme paramètre, il contient aussi des autres pas: normalstep, slashslashstep, et autres parenthesestep, et aussi une liste des prédicats.

```
Sequence _FLWxx_ParentheseStepxx (Sequence input)
{
  Sequence s = new Sequence(input); // s = input
  ...
  s = _FLWxx_ParentheseStepxx_SlashSlashStep01(s);
  ...
  s = _FLWxx_ParentheseStepxx_NormalStep01(s);
  ...
  // check predicate
  // itération de la séquence s
  Sequence result = new Sequence();
  int counter =0;
  for (int i=0; i<input.size(); i++) {
    Node node = input.getItem(i);
    counter++;
    if (_FLWxx_ParentheseStepxx_PredicateList(node, counter)) {
      result.add(anode);
```

```
    }
  }
  SequenceTool st = new SequenceTool();
  st.removeDuplicated(result);
  return result;
}
```

7.4.4 idée

Ce sont les idées général pour contruire des fonctions qui marchent comme une chemin XQuery. En effet, on peut avoir une classe abstraite qui charge la fonctionalité de la fonction _FLWxx_CollectDataxx():

```
classe CollectData()
{
  public Sequence collect();
}
```

et pour la fonction _FLWxx_CollectDataxx() on a une classe correspondante (comme une inner classe dans la fonction _FLWxx():

```
void _FLWxx() {
classe _FLWxx_CollectData extends CollectData ()
{
  public Sequence collect();
}
  _FLWxx_CollectData c = new _FLWxx_CollectData();
  Sequence iSequence = c.collect();
  ...
}
```

Et pour les fonctions normalstep, slashslashstep, parenthesestep, on peut faire la même chose, car les fonctions slashslashstep (ou normalstep, parenthesestep) sont toujours les mêmes. Ca fait le source code généré devient facile à lire.

```
Class Step ()
{
  public Sequence run(Sequence input);
}
classe NormalStep extends Step()
{
  public Sequence run(Sequence input);
}
classe SlashSlashStep extends Step()
{
  public Sequence run(Sequence input);
}
classe ParentheseStep extends Step()
{
  public Sequence run(Sequence input);
}
```

7.5 la fonction _Wherexx

La partie Where accepte une instance de la classe QueryRecordSet, itérer les records dedans, tester si les records sont acceptable, avec l'aide d'une expression XQuery.

On a une classe qui fournie une fonction de vérification:

```
class WherePredicat {
```

```
  public boolean doTest(QueryRecord record);
}
```

et comme dans la partie précédente, une fonctions _Wherexx sera comme suit:

```
QueryRecordSet _Wherexx(QueryRecordSet input)
{
  class _Wherexx_Predicat extends WherePredicat {
    public boolean doTest(QueryRecord record){...}
  }

  QueryRecordSet result = new QueryRecordSet();
  _Wherexx_Predicat tester = new _Wherexx_Predicat();
  for (int i=0; i<input.size(); i++) {
    QueryRecord record = input.getItem(i);
    if (tester.doTest(record)) {
      result.add(record);
    }
  }

  return result;
}
```

La classe _Wherexx_Predicat doivent être générée. Puisque le contenue du clause *where* est toujours une expression XQuery, donc il peut contient les expressions de chemins récursives.

7.6 la fonction _Orderxx

Le but de cette fonction est de trier la variable QueryRecordSet contenant les records des variables. Pour le faire, on doit avoir une fonction qui tient à comparer les records. On peut utiliser l'interface Comparator de Java:

```
interface Comparator {
  public int compare(Object o1, Object o2);
}
```

Alors, comme la fonction _WhereXX, la fonction _OrderXX devient facile à comprendre.

```
QueryRecordSet _Orderxx(QueryRecordSet input)
{
  class _Comparator implement Comparator {
    public int compare(Object o1, Object o2){...}
  }

  QueryRecordSet result = new QueryRecordSet();
  _Comparator comparator= new _Comparator();

  .. // sort here

  return result;
}
```

Nous pouvons utiliser les outils fournie par Java pour tries une queryrecordset. Par exemple, on convert la variable input à une array Java, et puis, appeler la fonction quicksort fournie dans la librairie Java, avec l'aide de la classe _Orderxx_Comparator.

7.7 la fonction _Returnxx

cette fonction accepte aussi la recordset et écrire les données dans ces records à la sortie standard. La raison d'écriture à la sortie standard est que c'est facile à redirect ce flux de donnée vers un fichier, ou vers le client.
Une classe peut être utilisée pour que les fonctions _ReturnXX deviennent faciles à comprendre.

```
Class RecordPrinter {
  public void print(QueryRecord record) {}
}
```

Maintenant la fonction _ReturnXX est:

```
QueryRecordSet _Returnxx(QueryRecordSet input)
{
  class _RecordPrinter extends RecordPrinter {
    public print(QueryRecord record){...}
  }

  _RecordPrinter printer = new _RecordPrinter();

  for (int i=0; i<input.size(); i++) {
    QueryRecord record = input.getItem(i);
    printer.print(record);
  }
}
```

8. Qu'est ce que TOM manque

8.1 Attribute matching

Je pense que c'est pas vraiment une chose manquée, mais c'est une erreur dans l'implémentation de TOM pour supporter XML.
Voilà la notation de Tom pour les attributs Xml:

```
XMLAttribute ::=  _*
      |VariableName *
      |AttributeName=[AnnotedName @] (Identifier|String)
      |[AnnotedName@]_=[AnnotedName @](Identifier|String )
```

La notation VariableName* maintenant nous donne une Java ArrayList qui peut être bien utilisée dans le constructeur `xml:

```
`xml(<A VariableName/>)
```

Avec la notation annotedName1@_=annotedName2, il doit bien itérer les attributs du noeud examiné: annotedName1 doit être de type Node, qui contient la référence à un des attribut, et la variable annotedName2 doit avoir la valeur string de cet attribut, mais maintenant ca ne marche pas**.** On ne peut pas utiliser annotedName1, mais la variable annotedName2 est possible.

8.2 Support de NameSpace

8.2.1 Que c'est le namespace

XML permet developpers à créer leurs propres langages de markup pour leurs propres projets. Ces langages peuvent être partagés parmi les individuels qui travaillent dans les projets similaires. XSL est un exemple très clair: XSL est une application XML pour transformer les documents XML. La sortie du langage XSL doit être bien formée, est de plus elle peut aussi contenir XSL. Alors on doit distinguer entre les éléments XML de XSL et les éléments XML sortis, même dans le case où ils ont les mêmes noms.
Le namespace est la solution. Il nous permet de mettre les éléments et les attributs dans des différents namespaces.

8.2.2 Syntaxe de namespace

Le namespace est définit en utilisant un attribut xmlns:prefix des éléments qui sont appliqués. La valeur de cet attribut est l'URI du namespace. Par exemle, le langage XSL est définit avec l'URI http://www.w3.org/XLK/Transform/1.0.
Par exemple, on declare le namespace "tom" dans l'élément "match":

```
<tom:match xmlns:tom="http://tom.loria.fr/TOM">
<tom:match>
```

A partir de maintenant, on peut utiliser le namespace "tom" dans les éléments et les attributs dans l'élément tom:match ci-dessus.
L'URI utilisé est formel: son but est grouper et désambiguïser les éléments et les attributs dans un document. On n'est pas sure que le document qui se situe à l'URI décris le syntaxe utilisé.
Le nom du namespace est un nom XML normal (ne doit pas contenir les ":").

Et il y a deux autres noms ne sont pas permits: xml et xmlns.
En utilisant le namespace, on va avoir un problème avec la validation. Si le DTD est écrit sans les préfix de namespace, alors il doit être réécrit avec l'utilisation de namespace. Ca veut dire que deux documents XML: avec et sans namespace, ne peuvent pas utiliser ce DTD.

8.2.3 Multiple namespaces:

On peut déclarer plusieurs namespaces dans un élément:

```
<tom:application xmlns:tom="http://tom.loria.fr/TOM"
      xmlns:vas="http://tom.loria.fr/TOM/vas">
      <tom:match>
      </tom:match>
      <vas:constructor>
      </vas:constructor>
</tom:application>
```

Et on peut aussi redéfinir le namespace dans un élément fils:

```
<ast:contructor xmlns:ast="http://tom.loria.fr/TOM/vas">
      <ast:contructor xmlns:adt="http://tom.loria.fr/TOM/adt">
      </ast:constructor>
</ast:constructor>
```

8.2.4 Attributs

Normalement un attribut est associé à l'élément où il se situe, il ne faut pas ajouter le namespace dans les attributs, mais en réalité, on peut utiliser les autres namespaces dans les déclaration des attributs. Mais c'est pas pratique du tout.

```
<loria:team loria:name="protheo" inria:name="protheo"/>
```

8.2.5 Namespace default

Dans un longue document XML, c'est impratique si on ajoute le namespace dans tout les éléments. On peut attacher un namespace par default à un élément et ses éléments fils en utilisant un attribut xmlns sans préfix:

```
<css xmlns="http://www.w3.org/CSS">
      <template ...>
      </template>
</css>
```

8.2.6 Namespace et TOM

TOM ne supporte pas le compiler. Mais malheureusement le namespace est crucial dans XML.
Pour supporter le namespace dans TOM, c'est pas très difficile: il y a deux pas qu'on doit compléter:

- Avoir la possibilité de parser les noms XML qualifiés

Actuellement les noms XML utilisés dans TOM ne sont pas qualifiés:

```
%match (TNode node) {
  <match></match> -> {...}
}
```

Le parseur doit parser une instruction match avec les noms qualifiés:

```
%match (TNode node) {
  <tom:match></tom:match> -> {...}
}
```

Les notations XML deviennent:

explicite:

```
<prefixe1:A(_*)> (_*, <prefixe2:B(_*)>(_*)</prefixe2:B>, _*)
</prefixe1:A>
```

explicite réduite:

```
<prefixe1:A[]> <prefixe2:B[]> [] </prefixe1:B> </prefixe1:A>
```

standard:

```
<prefixe1:A><prefixe2:B></prefixe2:B></prefixe1:A>
```

Et on peut ajouter aussi les wildcards: (on utilise la charactère "_" pour représenter le wildcard normal "*" , puisque dans TOM actuel, par convention, l'étoile "*" signifie une liste des éléments)

explicite:

```
<_:A(_*)> (_*, <_:B(_*)>(_*)</_:B>, _*)   </_:A>
```

explicite réduite:

```
<_:A[]> <_:B[]> [] </_:B> </_:A>
```

standard:

```
<_:A><_:B></_:B></_:A>
```

Ca signifie que l'élément A peut être dans n'importe quel namespace, et le même avec l'élément B.

On peut constater que l'utilisation de "_" au lieu de "*" ne nous donne aucun conflit avec la convention de nommage de XML de W3C, car un nom XML formel ne peut pas avoir une seule charactère "_".

Pour les attributs, on peut aussi utiliser ces notations.

Alors la syntaxe formelle devient:

```
XMLTerm ::=
      [AnnotedName `@']`<' XMLNameList XMLAttributeList `/>'
      | [AnnotedName `@']`<' XMLNameList XMLAttributeList `>'
XMLChilds `</'
XMLNameList `>'
      | `#TEXT' `(' Identifier | String `)'
      | `#COMMENT' `(' Identifier | String `)'
      | `#PROCESSING-INSTRUCTION' `(' (Identifier | String) `,'
(Identifier | String`)'

XMLNameList ::=
      `_`
      | XMLNamespace  `_`
      | XMLName
      | XMLNamespace  XMLName
      | `(' XMLName ( `|' XMLName )* `)'
      | `(' XMLNamespace XMLName ( `|' XMLNamespace XMLName )* `)'

XMLNamespace ::=
      XMLName `:`
      | `_` `:`
```

```
XMLAttributeList ::=
      `[' [ XMLAttribute (`,' XMLAttribute)* ] `]'
      | `(' [ XMLAttribute (`,' XMLAttribute)* ] `)'
      | ( XMLAttribute )*

XMLAttribute ::=
      `_*'
      | VariableName `*'
      | XMLNamespace AttributeName `=' [AnnotedName `@'] ( Identifier
| String )
      | [AnnotedName `@'] XMLNamespace `_' `=' [AnnotedName `@']
( Identifier | String )

XMLChilds ::=
      ( Term )*
      | `[' Term ( `,' Term )* `]'
```

- Comment distinguer entre les noms qui viennent de différents namespaces:

La spécification DOM fournie quelques fonctions pour obtenir le nom de namespace et aussi l'URI d'un élément ou un attribut XML.

```
Node.getPrefix();
```

obtenir le nom de namespace d'un noeud XML.

```
Node.getNamespaceURI();
```

obtenir l'URI correspondant du namespace d'un noeud. Si cet élément ne contient pas l'information concernant le namespace, l'URI obtenu est l'URI du namespace de l'élément au plus haut niveau.

8.3 Chemin de XQuery

Si on jete un coup d'oeil sur XQuery, on espère qu'on peut écrire la chemin de XQuery en une pattern dans %match. C'est idéal. Mais malheureusement c'est un peu loin. Les quatre choses manquées dans l'expression %match sont:

- la possibitilé de traverser vers le parent d'un noeud
- le prédicat numérique
- le prédicat booléen
- l'opérateur slashslash //

8.3.1 La possibitilé de traverser vers le parent d'un noeud

La base de TOM est la librairie ATerm qui ne donne aucune possibilité de monter vers le noeud parent à partir d'un terme. Pour que cet fonctionalité soit possible, on doit changer la librairie ATerm, et aussi ajouter les nouvelles signatures dans les déclarations de TOM.

Dans TOM actuel, un noeud quelconques est considérer comme un terme: une application de ce noeud à ses sous-termes:

```
f(x, y)
(ce noeud est une application de f à deux sous-termes x et y )
```

Si maintenant f est un des sous-termes de g, alors le terme g est représenté

par:

```
g(  ... , f (x , y), ...)
```

La question est: comment modifier TOM pour que à partir du terme f, on peut obtenir g, et comment le représenter mathématiquement.
La solution: pour le terme g(..., f(x,y), ...), on peut considérer que le terme f est l'application de f à x, y, et g^{-1}:

f(g^{-1} ,x ,y)

g^{-1} est toujour le premier argument du terme f.
Regarder une définition d'un term dans TOM (PeanoSimple3.java):

```
%typeterm term {
  implement           { ATermAppl }
  get_fun_sym(t)      { t.getAFun() }
  cmp_fun_sym(t1,t2)  { t1 == t2 }
  get_subterm(t, n)   { t.getArgument(n) }
  check_stamp(t)      { ...  return; }
  set_stamp(t)        { .... }
  get_implementation(t) { t }
}
```

On peut conserver les signatures de %typeterm, et ajouter une signature pour changer et obtenir le parent:

```
set_parent_sym(t, parent)
get_parent_sym(t)
```

Dans la contruction de l'arbre algébraique, un terme doit changer le sous-terme g^{-1} de ses sous-termes à lui-même, avec la signature set_parent_sym. Alors la définition

```
%typeterm term {
  implement           { ATermAppl }
  get_fun_sym(t)      { t.getAFun() }
  cmp_fun_sym(t1,t2)  { t1 == t2 }
  get_subterm(t, n)   { t.getArgument(n) }
  check_stamp(t)      { ...  return; }
  set_stamp(t)        { .... }
  get_implementation(t) { t }

  set_parent_sym(t, parent) {t.setParent(parent);}
  get_parent_sym(t)         {t.getParent(); }
}
```

(Dans les lignes de code ci-dessus, on suppose que la librairie ATerm fournie les deux fonctions setParent et getParent pour changer, obtenir la variable de membre interne m_parent de la classe ATerm.)

Malgré que g^{-1} est une sous-terme de f, mais la signature get_subterm(t, n) doit bien retourner x si n est égal à 0, et doit retourner y si n est égal à 1.
De plus, on doit changer la définition des opérateurs. Ici est la définition de l'opérateur zero qui n'a aucun réel sous-terme dans l'exemple PeanoSimple3.java:

```
  %op term zero {
    fsym { factory.makeAFun("zero",0,false) }
    make { factory.makeAppl(factory.makeAFun("zero",0,false)) }
  }
```

Avec g^{-1}, le nombre de tous les sous-termes augmente par 1. Et l'opérateur zero doit être redénifi par les lignes suivantes:

```
  %op term zero(_parent) {
    fsym { factory.makeAFun("zero",1,false) }
    make(parent)
      { factory.makeAppl(factory.makeAFun("zero",1,false), parent) }
  }
```

"_parent" est un terme prédéfinit, qui désigne le parent de tous les termes.

Bien sur que pour utiliser cet fonctionnalité, on doit développer une librairie qui fournie la possibilité d'obtenir la référence vers le parent d'un noeud.

8.3.2 L'opérateur SlashSlash //

L'opérateur slashslash est très fort, il nous donne la possiblité de extraire les noeuds à n'importe quel "nested" niveau.
Le principale de TOM est: un terme peut être considéré comme une application du noeud qui se situe au racine aux ses enfants directs. C'est difficile à contruire un modèle mathématique pour décrire la sémantique du l'opérateur slashslash //. Je pense que l'idée principale de TOM est de traiter les arbres algébraique, où on ne compte que seulement les enfants directs d'un noeud. La modification sera effectuée pour supporter l'opérateur // peut tout changer.

8.3.3 Le prédicat numérique

Les prédicats numériques sont beaucoup utilisés dans XQuery, on peut les voir nombreux dans les uses case de XQuery, surtout les requêtes basées sur la séquence.
Par exemple: (use case SGML, query 4, on utilise les prédicats numérique partout)

Locate the second paragraph in the third section in the second chapter (the second "para" element occurring in the third "section" element occurring in the second "chapter" element occurring in the "report").

(Chercher la deuxième paragraphe dans la troisième session dans le deuxième chapitre dans le document "sgml.xml")

Solution in XQuery:

```
<result>
  {
    (((doc("sgml.xml")//chapter)[2]//section)[3]//para)[2]
  }
</result>
```

C'est une démonstration que les prédicats numériques sont utilisés partout.

Comme j'ai dit avant, il y deux types de prédicat numérique: sans parentheses () et avec parentheses. si on utilise pas de parentheses, ce prédicat numérique est appliqué au pas courant.

- *Prédicat numérique sans parentheses:* Maintenant, pour la terme <B></B> est la sous-terme de <A></A>, on peut utiliser ces trois notations:

```
explicite:              <A(_*)> (_*, <B(_*)>(_*)</B>, _*) </A>
explicite réduite:      <A[]> <B[]> [] </B> </A>
standard:               <A><B></B></A>
```

On doit ajouter quelques choses pour indiquer la position des sous-termes, et aussi les attributs. Par exemple:
Avec la chemin dans XQuery:

```
A/B[1]
```

la syntaxe explicite dans Tom:

```
<A(_*)> (_*, <B(_*)>[1](_*)</B>, _*) </A>
```

ou:

```
<A[]> <B[]>[1][]</B> </A>
```

ou:

```
<A> <B>[1]</B> </A>
```

- *Prédicat numérique avec parentheses:*

En regardant sur quelques exemples XML de TOM, on constate que TOM est assez fort: il peut comparer un arbre XML donné avec un vrai arbre algébraique ayant plusieurs branches et plusieurs feuilles:

```
%match (TNode node) {
      <book>
            <section>
                  <title>
                        section 1
                  </title>
            </section>
            <section>
                  <title>
                        section 2
                  </title>
            </section>
      </book>
}
```

Pour assurer cette capabilité, TOM doit utiliser la syntaxe comme celle de XML: chaque clause ouvert doit avoir un clause fermé correspondant.

```
<book></book>
```

Avec la syntaxe utilisée, c'est impossible d'ajouter les prédicats numériques avec parenthèses dans un statement %match.
De plus, TOM utilise les différentes notations pour représenter les éléments et les attributs, tandis que XQuery utilise la même notation: XQuery considère que une attributs est aussi un noeud mais ne pas avoir d'enfant.
Par exemple, le terme:

```
<A>
```

```
  <B id="100">
  </B>
</A>
```

est représenté dans Tom sous forme (pour obtenir la valeur "100")

```
<A><B id=value></B></B>
```

mais dans XQuery:

```
A/B/@id
```

Donc c'est facile pour XQuery à ajouter les prédicats numériques n'importe où:

```
(A[1]/B[1])[1]/@id[1]
```

Alors je propose d'ajouter une autre convention de syntaxe dans TOM: On constate que dans si on utilise un pattern XML de TOM pour représenter une chemin dans XQuery, les clauses fermées ne sont pas utils. (les parties **BOLD** suivant)

```
<A(_*)> (_*, <B(_*)>(_*)</B>, _*) </A>
<A[]> <B[]> [] </B> </A>
<A><B></B></A>
```

On peut les éliminer (bien sur que la syntaxe actuelle doit être bien conservée, car elle est très utile pour les autres applications) donc le pattern pour le noeud XML

```
<A><B></B></A>
```

devient:

```
<A(_*)> (_*, <B(_*)>(_*), _*)
<A[]> <B[]> []
<A><B>
```

Cela presque ressemble à la notation de XQuery, et donc on peut ajouter les prédicats numérique:

XQuery	***TOM***
A/B[1]	<A><B>[1]
A[1]/B[2]	<A>[1]<B>[2]
(A[1]/B[2])[3]	(<A>[1]<B>[2])[3]
(A[1]/B[2])[3]/C	(<A>[1]<B>[2])[3]<C>

Pour les éléments, il aura l'air assez bon, mais pour les attributs, il existe encore un problème. Une chemin de XQuery peut être:

```
$book/(A/B[1]/@*[2])[3]
```

avec la notation des attributs "nested" dans un élément comme celle de TOM maintenant, c'est difficile à écrire cette chemin.

Alors on peut changer la position des attributs dans les patterns TOM: par exemple

```
<A><B id=value>
```

devient:

```
<A><B><#id=value>
```

Avec cette nouvelle notation, la chemin $book/(A/B/@*[1])[2] dans XQuery devient plus facile:

```
%match (TNode book) {
  <_>(<A><B[1]><#id[2]>)[3]
}
```

Alors on peut donner la syntaxe formelle de cette nouvelle convention:

```
XMLTerm ::=
      XMLClauseList

XMLClauseList ::=
      | [AnnotedName `@'] XMLClause ([AnnotedName `@'] XMLClause)*
      `(` [AnnotedName `@']XMLClause ([AnnotedName `@'] XMLClause)* `)
` NumericPredicateList

XMLClause ::=
      `<' XMLNameList | `#` XMLAttributeList `>'  NumericPredicateList

XMLNameList ::=
      (XMLNamespace)?  `_`
      | (XMLNamespace)? XMLName ( `|' (XMLNamespace)? XMLName )*

XMLNamespace ::=
      XMLName `:`
      | `_` `:`

XMLAttributeList ::=
      XMLAttribute ( `|' XMLAttribute )*

XMLAttribute ::=
       `_*'
      | VariableName `*'
      | XMLNamespace AttributeName `=' [AnnotedName `@'] ( Identifier
| String )
      | [AnnotedName `@'] `_'  `=' [AnnotedName `@'] ( Identifier |
String )

NumericPredicateList ::=
      (NumericPredicat)*

NumericPredicate ::=
      `[' Number `]'
      | `[' Range `]'

Number ::=
      [0 - 9]+

Range ::=
      [0 - 9]+ `-` [0 - 9]+
```

En effet, il y a deux type de prédicats numériques: Range et Number. Avec le type "Range", la chemin sélectionne une plage de noeud dont la position est spécifiée par le partern "number-number"

- La notation pour indiquer l'axis dans une chemin

Il nous rester une chose: comment indiquer que la navigation dans l'arbre XML oriente vers le parent du noeud courant ou vers une autre direction.
J'ajoute l'axis de navigation dans la production XMLNameList:

```
XMLClause ::=
```

```
      `<' Axis XMLNameList | `#` XMLAttributeList `>'
NumericPredicateList

Axis ::=
      `parent` `::`
      | `..` `::`
      | `self` `::`
      | `.` `::`
      | `child` `::`
```

Parmi les trois axes, "child::" est par défault. ".." est l'abréviation de "parent", et "." est l'abréviation de "self".

- Problèmes de parsing

Le parseur de TOM peut confuser entre l'ancienne syntaxe et la nouvelle syntaxe: il nous faut ajouter quelques choses au début pour distinguer entre les deux. Par exemple, j'ajoute une signe XML empty "<>" pour indiquer la nouvelle syntaxe:

```
      XMLTerm ::=
            "Ancienne Syntaxe"
            | `<>` "Nouvelle Syntaxe"
```

- Les prédicats et le support de namespace

Une fois que TOM supporte le namespace, l'ancienne syntaxe de TOM peut être gardé, mais on peut encore supporter le prédicat numérique sans parenthèses. En général, les prédicats numériques avec parenthèses sont très rarement utilisés.
Par exemple, on définit un namespace tomxquery, et définit quatre éléments et attributs comme suit:

```
<tomxquery:predicatenumeric value="{Number}">
<tomxquery:rangenumeric low-value="{Number}" high-value="{Number}">
```

et les attributs:

```
tomxquery:predicatenumeric="{Number}">
tomxquery:lowrangenumeric="{Number}" highrangenumeric="{Number}">
```

Alors la syntaxe formelle devient:

```
XMLTerm ::=
      [AnnotedName `@']`<' XMLNameList XMLAttributeList `/>'
      | [AnnotedName `@']`<' XMLNameList XMLAttributeList `>'
[ElementPredicate] XMLChilds `</'XMLNameList `>'
      | `#TEXT' `(' Identifier | String `)'
      | `#COMMENT' `(' Identifier | String `)'
      | `#PROCESSING-INSTRUCTION' `(' (Identifier | String) `,'
(Identifier | String`)'

ElementPredicat ::=
      `<` `tomxquery:predicatenumeric` `value` `=` NumericLiteral `>`
      | `<` `tomxquery:predicatenumeric` `low-value` `=`
NumericLiteral `high-value` `=` NumericLiteral `>`


XMLNameList ::=
      `_`
```

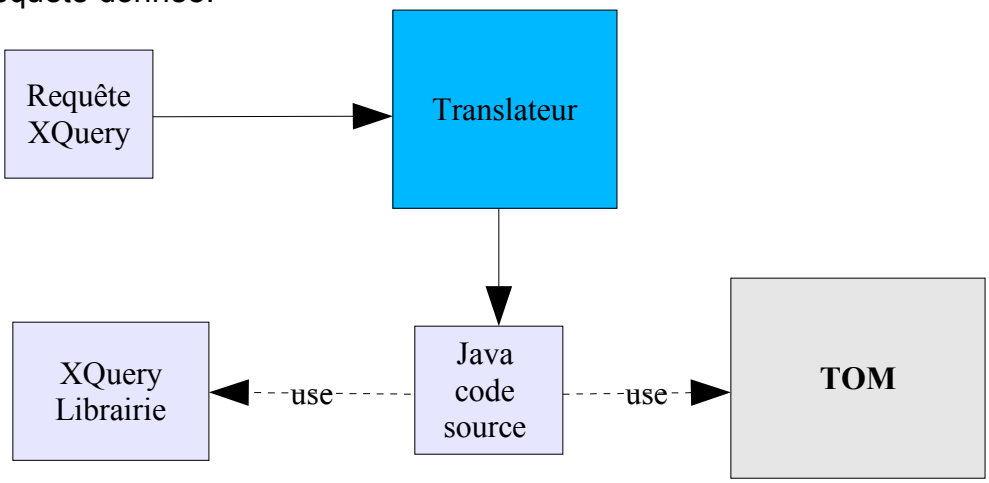
```
      | XMLNamespace  `_`
      | XMLName
      | XMLNamespace  XMLName
      | `(' XMLName ( `|' XMLName )* `)'
      | `(' XMLNamespace XMLName ( `|' XMLNamespace XMLName )* `)'

XMLNamespace ::=
      XMLName `:`
      | `_` `:`

XMLAttributeList ::=
      `[' [ XMLAttribute (`,' XMLAttribute)* ] `]'
      | `(' [ XMLAttribute (`,' XMLAttribute)* ] `)'
      | ( XMLAttribute )*

XMLAttribute ::=
      `_*'  [AttributePredicate]
      | VariableName `*'
      | XMLNamespace AttributeName `=' [AnnotedName `@'] ( Identifier
| String )
      | [AnnotedName `@'] XMLNamespace `_' `=' [AnnotedName `@']
( Identifier | String ) [AttributePredicate]


AttributePredicate ::=
       `tomxquery:predicatenumeric` `=` NumericLiteral `>`
      | `tomxquery:lowrangenumeric` `=` NumericLiteral
`tomxquery:lowrangenumeric ` `=` NumericLiteral


XMLChilds ::=
      ( Term )*
      | `[' Term ( `,' Term )* `]'
```

8.3.3 Le prédicat booléen

C'est vraiment difficile pour intégrer les prédicat booléen dans un pattern XML. J'ai aucune idée.

9. Le translateur de XQuery

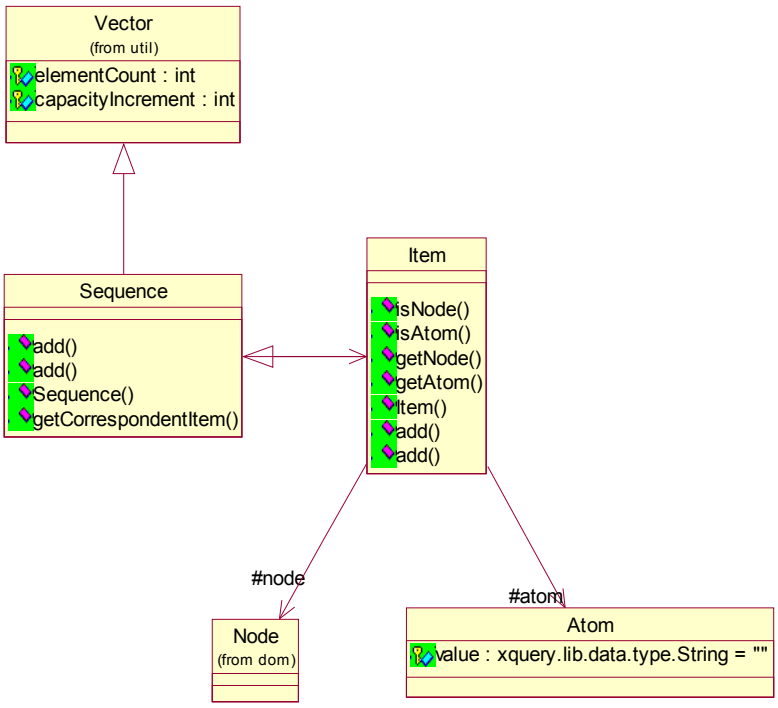
On parle d'ici d'une translateur: à partir d'une requête XQuery, cette translateur doit la transformer en un programme Java correspondant , qui utilise les facilities fournies par TOM. Ce programme Java sera compilé en utilisant un compiler Java. L'application obtenue fera la même chose que la requête donnée.



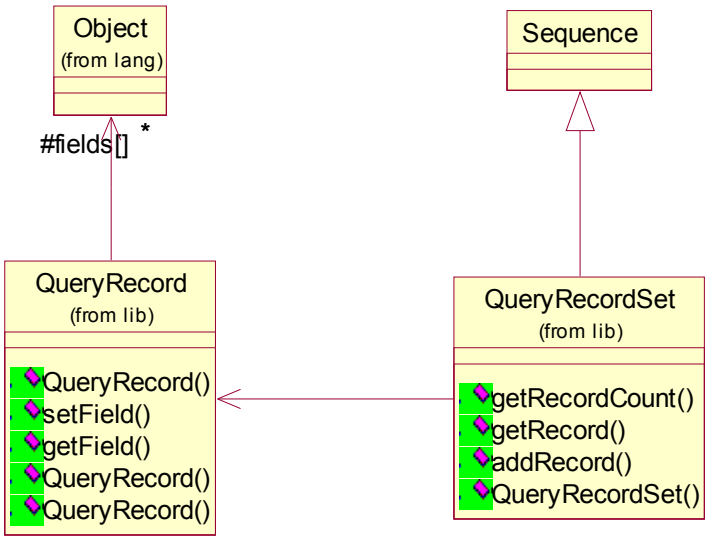
Le programme Java doit utilise aussi une autre librairie Java, pour pouvoir exécuter. Le but de cette librairie est de fournie la structure de donnée de XQuery, basée sur la norme DOM de W3C, et aussi les fonctions de base pour traiter ces données. Ces fonctions doivent correspondre à des fonctions fournies par la spécification de XQuery.

9.1 La librairie XQuery

Comme on a abordé avant, cette librairie définit les structures de donnée, et les fonctions basée sur la spécification de XQuery pour réduire la taille des fichiers Java générés par le translateur. Les deux structures de données de base de XQuery est la séquence (Sequence) et l'élément (Item).
Comme dans la partie 7.3 dans ce document, une variable de type Sequence est une séquence des éléments Item. Mais un élément est aussi une séquence d'un élément. Un élément peut être, soit un atôme: les valeurs de type élémentaire, soit un noeud: un noeud XML.

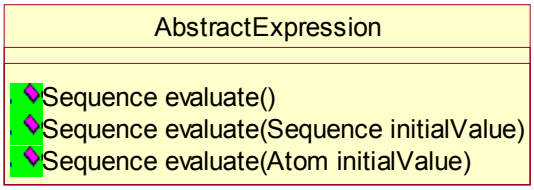


La classe QueryRecordSet dans la partie 7 est implémentées comme une séquence, elle contient plusieurs instances de la classe QueryRecord. La classe QueryRecord contient n'importe quel nombre de champ, ces champs sont des références vers les variables dans la portée (scope) courante.

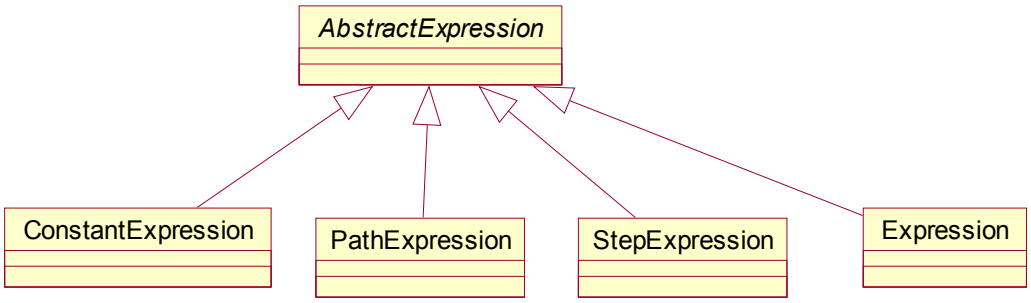


A partir des classes Sequence et XQueryRecordSet, on peut construire la classe SequenceTool qui fournie la fonction de trie et les fonctions de XQuery sur les séquences de donnée. L'idée principale de la fonction "sort" a été

abordée dans la partie 7 de ce document.
On concentre sur l'implémentation des classes pour représenter la chemin de XQuery. La base de cet implémentation est la classe abtraire AbstractExpression qui peut contienir un ou plusieurs objets dedans. Elle la une fonction de membre evaluate pour evaluer la valeur des instances de cette classe. La valeur retournée est de type Sequence, car les résultats des requêtes XQuery sont toujours les séquences. La classe AbstractExpression est vraiment un opérateurs de n-aires, (n est le nombre de objets contenus dans la classe). Il y a trois version de la fonction evaluate: les deux dernières fonctions est spécialisées pour le calcule le résultat d'une chemin XQuery.



Pour représenter une chemin de XQuery, on a quatre classes:ConstantExpression, PathExpression, Expression et StepExpression.



La ConstantExpression représente l'opérateur constant, elle ne change pas la valeur de l'objet contenu dedans:

ConstantExpression({valeur}).evaluate={valeur}

Une chemin tout d'abord est une PathExpression, si cette expression contient les parenthèses au début et à la fin, elle devient une Expression.
Par exemple:

$varname/section

est une PathExpression, mais si on ajoute les parenthèses, elle devient une Expression

($varname/section)

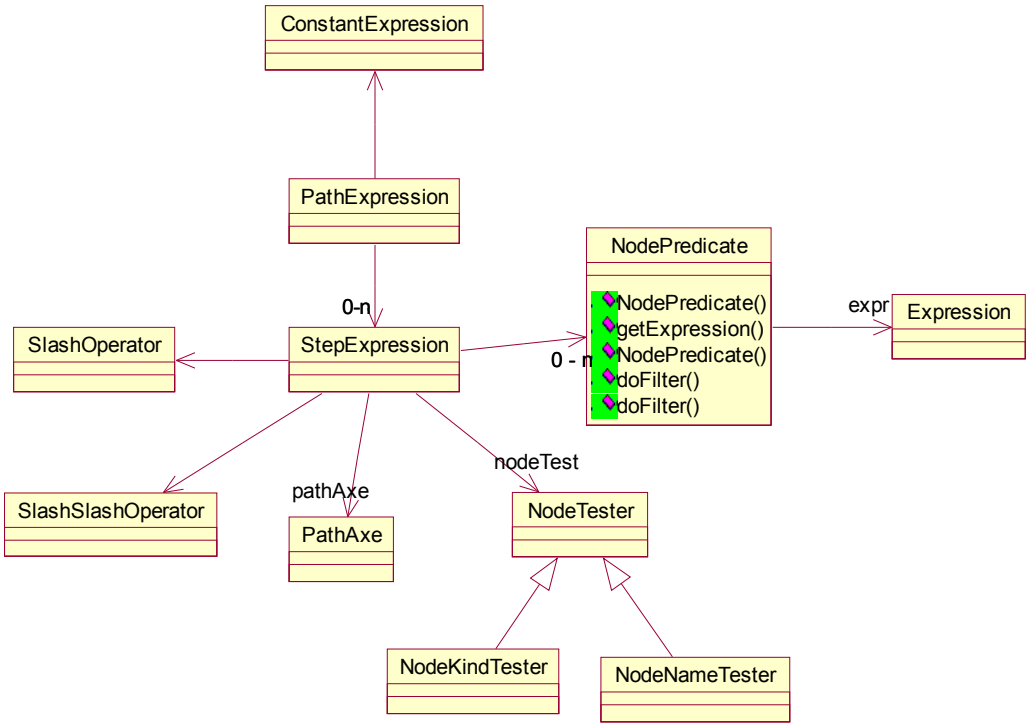
Une PathExpression contient toujour une valeur initial, et plusieurs StepExpression dedans:
Par exemple, la chemin:

$varname/book/section/

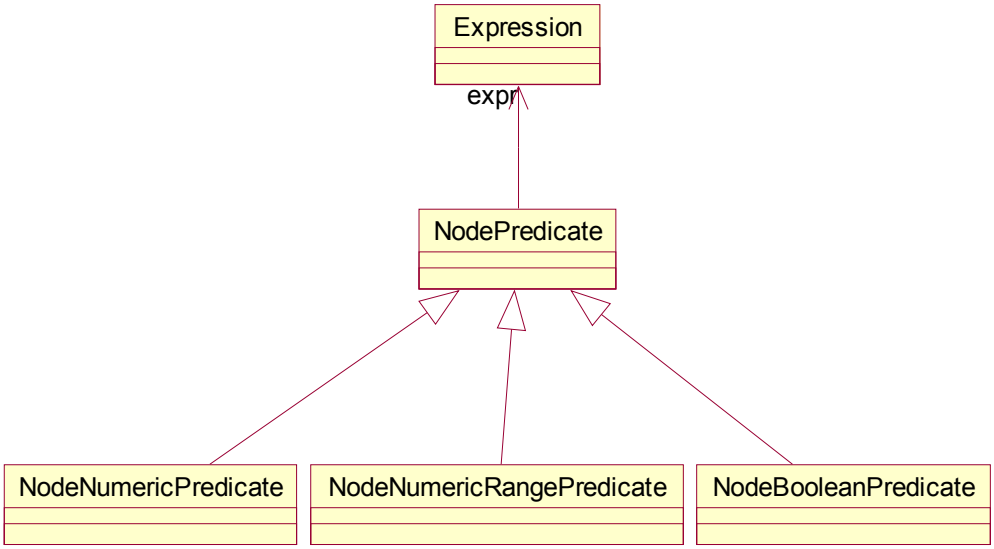
est considérée comme:
PathExpression(

ConstantExpression(varname),

```
StepExpression("book"),
StepExpression("Section"));
```



Une StepExpression peut être un pas de chemin avec SlashSlashOperator (//) ou SlashOperator (/). Elle contient aussi une PathAxis, une NodeTester, et une liste des NodePredicate (comme dans la spécification de XQuery). Une NodePredicate est en effet une Expression. Par défault, une NodePredicate est toujour considéré comme un prédicat booléen, si sa valeur n'est pas de type integer, et on ajoute aussi deux classes NodeNumericRangePredicate et NodeNumericPredicate.
Une StepExpression peut aussi contient des Expression aulieu des NodeTester et PathAxe. C'est le cas on utilise les parenthèses dans la chemin XQuery. On va la voir dans les exemples au-dessous.



Alors, avec ces classes, on peut modéliser les chemins XQuery facilement. Je donne ici quelques exemples possibles d'une chemin XQuery:

- $var:

```
PathExpression(ConstantExpression(var));
```

- ($var)

```
PathExpression(
      Expression(
            PathExpression(
                  ConstantExpression(var))));
```

- $var[1]

```
PathExpression(
      StepExpression(
            ConstantExpression(var),
            NodeNumericPredicate(1)));
```

- ($var)[1]

```
PathExpression(
      StepExpression(
            Expression(
                  PathExpression(
                        ConstantExpression(var)))
      NodeNumericPredicate(1)));
```

- ($var[2])[1]

```
PathExpression(
```

```
      StepExpression(
            Expression(
                  PathExpression(
                        StepExpression(
                              ConstantExpression(var),
                              NodeNumericPredicate(2))))),
      NodeNumericPredicate(1)));
```

- `$var/step1:`

```
PathExpression(
      ConstantExpression(var),
      StepExpression(
            SlashOperator(),
            PathAxe(CHILD),
            NameNodeTest("step1")));
```

- `$var/step1[1]:`

```
PathExpression(
      ConstantExpression(var),
      StepExpression(
            SlashOperator(),
            PathAxe(CHILD),
            NameNodeTest("step1"),
            NodeNumericPredicate(1)));
```

- `($var/step1)[1]:`

```
PathExpression(
      StepExpression(
            Expression(
                  PathExpression(
                        ConstantExpression(var),
                        StepExpression(
                              SlashOperator(),
                              PathAxe(CHILD),
                              NameNodeTest("step1")))),
            NodeNumericPredicate(1)));
```

- `$var/step1//step2`

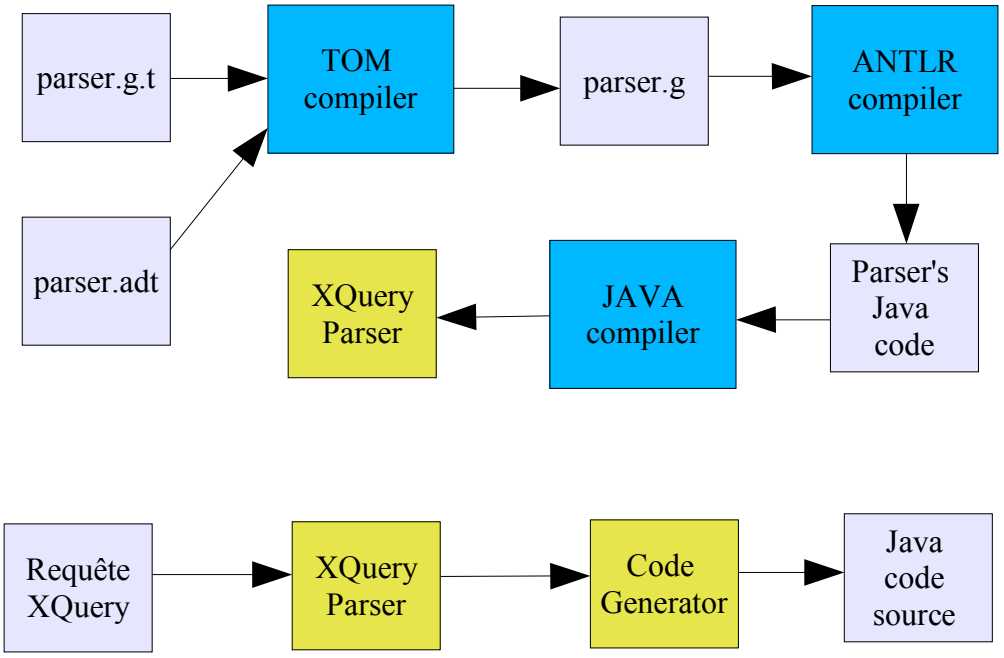
```
PathExpression(
      ConstantExpression(var),
            StepExpression(
                  SlashOperator(),
                  PathAxe(CHILD),
```

```
            NameNodeTest("step1"),
      StepExpression(
            SlashSlashOperator(),
            PathAxe(CHILD),
            NameNodeTest("step2"))));
```

Les classes ci-dessus marchent bien maintenant. Il nous reste de construire les classes pour les différents types d'expression dans XQuery, par exemple les expressions additives, multiplicatives, d'union ... Et bien sur les fonctions de base de XQuery. (XQuery fournie environs 100 fonctions différentes)

9.2 Le translateur

Le parseur est écrit en utilisant le parseur générateur ANTLR. Au début, j'écris un fichier parser.g. Ce fichier peut être parsé par ANTLR. Et après, on écrit aussi un fichier parser.adt pour définir les noeuds de l'arbre AST de XQuery. Basée sur ces deux fichiers, on peut construire un fichier parser.g.t qui contient les expressions backquote de TOM, pour bien contruire l'arbre syntax tree d'une requête XQuery.



Maintenant le fichier parser.g est complet. Il est basé sur la spécification XQuery version 1.0, la date de cette spécification est 12 novembre 2003. (http://www.w3.org/TR/2003/WD-xquery-20031112/). Il nous faut construire le fichier parser.adt, parser.g.t, Et aussi le code générateur.

10. Annexe

La structure du répertoire xquery:

Répertoire	**Description**
examples	Le répertoire des exemples de TOM
xquery	XQuery répertoire
doc	contient les documents
uc1 uc2 uc3 uc4	Contient les requêtes XQuery qui sont implémentées en utilisant seulement TOM, sans librairie supplémentaire. (les vrais exemples)
xqueryimpl	Les nouvelles implémentations
xquery	
lib	Contient la librairie abordée dans la partie 9 de ce document
util	Une ancienne implémentation de la librairie dans la partie 9
uc1 uc2 uc3 uc4	Les exemples qui utilisent les deux implémentations.
data largedata verylargedata hugedata	Le répertoire data contient les documents standards de W3C. Les trois répertoires restant contiennent les grandes sommes de données générées par TOXGENE (http://www.cs.toronto.edu/tox/toxgene/))
parser	Contient les .g, .g.t et .adt du parseur. (En utiliser avec ANTLR (http://www.antlr.org/)) Il contient aussi le fichier .jjt (en utiliser avec JavaCC (https://javacc.dev.java.net/))