UFR 922 Informatique
Master Informatique – Spécialité STL
Implantation des langages de programmation (ILP)

TD2 — Syntaxe abstraite d'ILP1

Jacques Malenfant, Christian Queinnec

1 Les schémas RelaxNG

Les liens :

RelaxNG `http://www.oasis-open.org/committees/relax-ng/`

Documents :

Tutoriel	`http://www.oasis-open.org/committees/relax-ng/tutorial-20011203.html`
Syntaxe compacte :	
spécification	`http://www.oasis-open.org/committees/relax-ng/compact-20021121.html`
tutoriel	`http://www.relaxng.org/compact-tutorial-20030326.html`
Livre	`http://books.xmlschemata.org/relaxng/`

Outils spécifiques :

Jing	`http://www.thaiopensource.com/relaxng/jing.html`
Trang	`http://www.thaiopensource.com/relaxng/trang.html`

1.1 Introduction à RelaxNG

Généralités :

– Langage de schéma permettant de contraindre la forme d'un document XML.
– Un document XML instance d'un schéma pourra être validé par rapport à celui-ci.
– Relax NG est un concurrent des DTD et autres XML Schémas.
– C'est une norme ISO/IEC 19757-2.

Principaux éléments du langage de schéma dit « syntaxe compacte » :

– Contenu textuel : patron `text`
– Attribut : patron `attribute id {` *<contenu>* `}`
– Élément : patron `element name {` *<contenu>* `}`
– Optionalité : patron ?

```
element author {
    element name { text },
    element born { text },
    element died { text }?
}
```

– Une ou plusieurs répétitions : patron “+”

```
element author {
    element name { text },
    element born { text },
    element died { text }?
}+
```

– 0, une ou plusieurs répétitions : patron “*”
– Patrons nommés : permet de référencer un patron, y compris récursivement

```
author-element = element author {
    element name { text },
    element born { text },
    element died { text }?
}
```

– Référence à un patron nommé : utilisation du nom

```
element livre {
    attribute isbn { text },
    author-element,
    element title { text }
}
```

– Notion de grammaire : élément de départ `start` donne la racine du document

```
grammar {                       // optionnel (implicite)
    name-element = ...
    ...
    start = element library { ... }
    ...
}
```

– Choix entre plusieurs possibilités : patron `choice` ou `|`

```
element name {
    text | (element first { text }, element middle { text },
            element last { text }) }
```

Contraintes sur le contenu :

– Type token : chaîne de caractères avec blancs normalisés
– Type string : chaîne de caractères sans normalisation
– Type liste : `list {` *<contenu>* `}`
chimère tuple/liste car la répétition doit être définie explicitement

```
attribute name { list { token token } }   // prénom nom
attribute names { list { token+ token } } // prénoms nom
```

– Valeurs fixes ou énumérations :

```
attribute available { "available" | "checked out" | "on hold" }
```

– Exclusions : préfixe `-`

```
attribute name { token - "ilp" }
```

– Types provenant des bibliothèques de types :
  – XML Schémas : `xsd:string`, `language`, `xsd:integer`, `xsd:decimal`, `xsd:int`, `xsd:dateTime`, ...
– Facettes : propriétés contraignant les valeurs de types
  – `length`, `maxLength`, `minLength` : contraintes sur la longueur des types `string`, `binary` et `list`
  – `minInclusive`, `minExclusive`, `maxInclusive`, `maxExclusive` : contraintes d’intervalle sur des

types numériques (`decimal`, `integer`, `float`, `double`, dates, durées)
- `totalDigits`, `fractionDigits` : contraintes sur la taille des types de nombres réels
- `pattern` : contrainte d'un champ texte par une expression régulière

```
element author {
    attribute id { xsd:ID { maxLength = "16" } },
    element name { xsd:token { maxLength = "255" } },
    element born { xsd:date { minInclusive = "1900-01-01",
                              maxInclusive = "2099-12-31",
                              pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}" } }
}
```

Composition de schémas (grammaires)

- Références externes : inclusion de définitions externes de grammaires

```
element library {
    element book {
        attribute id { xsd:ID },
        attribute available { xsd:boolean },
        element isbn { token },
        element title { attribute xml:lang { xsd:language }, token },
        external "author.rnc" +,     # définit l'élément author
        external "character.rnc" *   # définit l'élément character
    }+
}
```

- Référence aux éléments d'une grammaire parent : `parent`

```
start = element-author
element-author = element author {
    attribute id { xsd:ID },
    parent element-name,      # réference à la grammaire incluante
    parent element-born?
}
```

- Fusion de grammaires : `include`, inclusions d'éléments, pas de grammaires, c'est-à-dire que le fichier inclus ne contient pas de patron nommé `start`

```
# contenu de common.rnc
element-name = element name { token }
element-born = element born { xsd:date }
attribute-id = attribute id { xsd:ID }
content-person = attribute-id, element-name, element-born?


# grammaire
include "common.rnc"
start = element library {
    element book {
        attribute-id,       # référence au schéma inclus
        attribute available { xsd:boolean },
        element   isbn      { token },
        element   title     { ... },
        element   author    { content-person, element-died? }+,
        element   character { content-person, element-qualification }*
    }+
}
element-died = element died { xsd:date }
element-qualification = element qualification { token }
```

– Fusion avec remplacement de définition
  – une redéfinition du nom remplace la définition importée

```
include "library.rnc" { # définition element-name remplaçante
    element-name = element name { xsd:token, maxLength="80" }
}
```

  – combinaison par choix : ajoute des choix à une définition importée

```
include "library.rnc"
start |= element-book       # s'ajoute à element-library comme point
                            # d'entrée de la grammaire
```

2 Exercices de modification de la grammaire 1

Pour chacun des traits de langage suivants, introduisez successivement le trait dans la grammaire de niveau 1 puis donnez un programme (en syntaxe XML) qui respecte la nouvelle grammaire et qui utilise le trait ajouté :

1. affectations
2. boucles while

3 Manipulation de documents XML

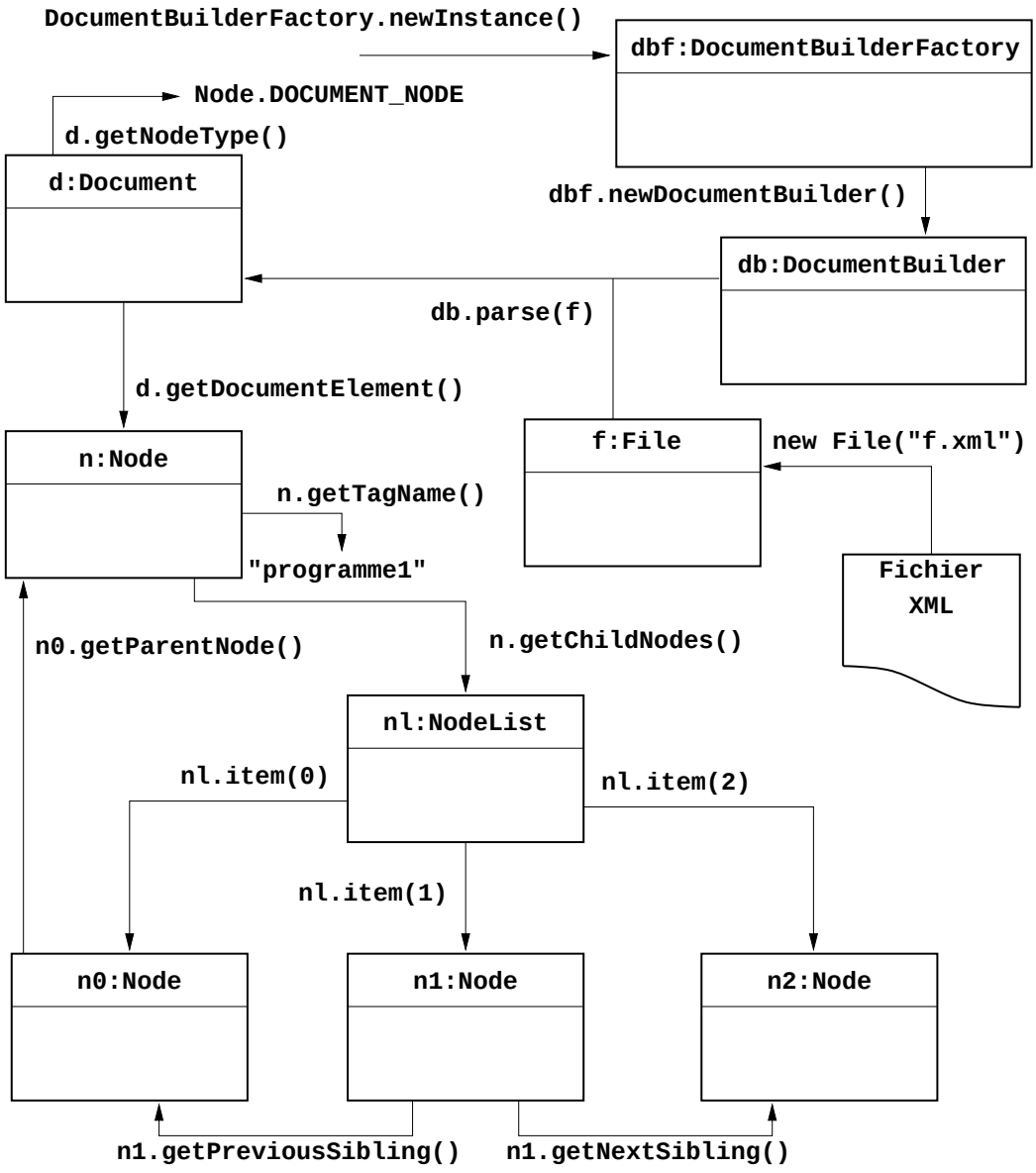
3.1 DOM

Hiérarchie des interfaces DOM (`org.w3c.dom`)

Rappel : DOM est un modèle d'arbre généralisé, c'est-à-dire un arbre dont les nœuds (non-feuilles) peuvent avoir un nombre quelconque de nœuds fils.

```
Node
    Document               // Possède un unique noeud de contenu
    DocumentFragment       // Document incomplet, permet la création dynamique
    DocumentType           // Valeur de l'attribut doctype du document
    ProcessingInstruction  // Représente une instruction de traitement
    CharacterData          // Valeurs terminales, i.e. feuilles
        Comment            // commentaires (à l'ajout)
        Text
            CDataSection   // chaîne non-interprétée
    Element                // Noeud représentant <...> ... </...>
    Attr                   // Noeud représentant <...>="..."
    Entity                 // définitions
    EntityReference
    Notation
NodeList                   // Liste de noeuds
```

Flôt de manipulation de l'arbre DOM



Nota : La liste de nœuds récupérée par `n.getChildNodes()` peut contenir des nœuds éléments, attributs, textes, ...

Un programme manipulant un arbre DOM se construit généralement autour de quelques méthodes, chacune capable de traiter un type de nœuds. Les principales sont donc :

– la méthode de traitement d'un document (nœud de type `Document`),
– la méthode de traitement des éléments (nœuds de type `Element`),
– la méthode de traitement des attributs (nœuds de type `Attr`), et
– la methode de traitement des listes de nœuds (objets DOM de type `NodeList`).

Programmes ILP1 en DOM

Un programme ILP1 est représenté par un arbre DOM :
– dont la racine est un nœud de type `Document`,
– contenant, comme il se doit, un unique nœud de type `Element` qui est le contenu du document.
– Ce nœud `Element` unique a pour nom (étiquette) `programme1`.
– Les nœuds éléments du document portent les étiquettes des éléments définis dans le schéma Relax NG `grammar1.rn{c|g}` qui régit également leur agencement les uns par rapport aux autres.
– Ces contraintes imposées par le schéma sur le document sont vérifiées par la validation du document par rapport à son schéma.
– Cette validation joue le rôle de vérification de l'exactitude syntaxique du programme dans ILP, c'est-à-dire les vérifications généralement faites dans la partie frontale des compilateurs.

3.2 Exercice avec le DOM

À partir du paquetage `fr.upmc.ilp.ilp1.eval`, créez une nouvelle variante d'implantation d'ILP1 telle que lors de l'analyse d'un programme, cette variante compte le nombre de constantes qui apparaissent dans le programme et l'imprime à l'issue des tests réalisés. Il faut créer de nouveaux « *paquetages* » `fr.upmc.ilp.ilp1tme2` et `fr.upmc.ilp.ilp1tme2.eval` de manière à redéfinir les classes nécessaires, comme `fr.upmc.ilp.ilp1.Process`, `fr.upmc.ilp.ilp1.ProcessTest` et `fr.upmc.ilp.ilp1.eval.EASTParser`.

Répétez l'exercice précédent en utilisant cette fois l'arbre de syntaxe abstraite. Redéfinissez les classes d'AST du package `fr.upmc.ilp.ilp1.eval` en ajoutant une méthode `compteConstantes` de manière à compter le nombre de constantes dans un programme ILP1. Utilisez ensuite cette méthode pour faire imprimer le nombre de constantes dans le programme.

Exécutez les tests d'ILP1 avec vos modifications et vérifiez que la décompte fait sur l'arbre DOM correspond bien à celui fait sur l'arbre de syntaxe abstraite.