UFR 919 Informatique – Master Informatique

Spécialité STL – UE MI016 – ILP

TME2 — Syntaxe abstraite d'ILP1

Jacques Malenfant, Christian Queinnec

1 Les schémas RelaxNG

Objectif : Compter, de plusieurs manières, les constantes apparaissant dans un programme ILP.

Buts :

— Comprendre les grammaires RelaxNG
— Savoir étendre la grammaire d'ILP
— Savoir arpenter un DOM ou un AST

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`, `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
    }+
}
```

— 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.rc" { # définition element-name remplaçante
    element-name = element name { xsd:token, maxLength="80" }
}
```

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

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

Consultez la grammaire d'ILP1 pour voir un premier exemple en « grandeur nature » d'un schéma Relax NG.

1.2 Transformer des schémas RelaxNG et valider les documents

L'un des avantages majeurs des schémas RelaxNG est d'avoir une syntaxe compacte claire et lisible. Cependant, pour ne pas retomber dans les difficultés des DTD qui utilisent une syntaxe mixte XML/non-XML dans des documents qui sont supposés être en XML, RelaxNG introduit une distinction claire entre la syntaxe compacte non-XML et la syntaxe XML.

La syntaxe compacte est utilisée par les développeurs qui profitent ainsi de sa lisibilité, alors que la syntaxe XML est utilisée par les outils, comme le valideur Jing, qui profitent de sa facilité de traitement. Pour faire le pont entre les deux syntaxes, l'outil Trang pour passer de la syntaxe compacte à la syntaxe XML.

Pour utiliser Trang, il faut faire :

```
java -jar $ILPDIR/Java/jars/trang.jar grammaire.rnc grammaire.rng
```

où `ILPDIR` est une variable d'environnement donnant le chemin du workspace Eclipse de votre projet `ILP`, `grammaire.rnc` est le fichier contenant le schéma en syntaxe compacte et `grammaire.rng` est le fichier dans lequel sera écrit le schéma en syntaxe XML.

Vous pouvez aussi utiliser le greffon ILP dans le menu contextuel des grammaires. Vous pouvez également convertir vers un schéma XML (`.xsd`) pour lequel un éditeur spécifique existe en Eclipse.

1.3 Travail à réaliser

Pour chacun des traits de langage suivants, créez une nouvelle grammaire que vous nommerez `Grammars/grammar1-tme2.rnc`, ajoutant les traits suivants (ils seront traités dans ILP2 : Ω

1. affectations
2. boucles while
3. blocs locaux n-aires
4. définition et application de fonction

Pour chacun de ces traits, vous écrirez au moins un programme l'utilisant. Générez les fichiers rng de votre grammaire étendue (comparez rnc et rng au passage) puis validez vos programmes comme vous le souhaitez, soit en ligne de commande, soit en utilisant le greffon ILP.

***Pour enseignant*:** Réponse pour les affectations :

On crée la nouvelle grammaire dans le répertoire Grammars sous le nom `grammar1-td2_1.rnc` :

Pour générer le fichier rng correspondant, on pourra lancer la commande suivante dans le répertoire `Grammars` :

```
java -jar $ILPDIR/Java/jars/trang.jar grammaire1-td2_1.rnc grammaire1-td2_1.rng
```

On ajoute ensuite un exemple de programme dans le répertoire Grammars/Samples de nom u01-1-td2_1.xml :

```
<?xml version='1.0' encoding='UTF-8' ?>
<!--  -->
<programme1>
<!-- test:name description='TD2 : affectation simpliste' -->
  <blocUnaire>
    <variable nom="x"/>
    <valeur><entier valeur='0'/></valeur>
    <corps>
      <affectation nom="x">
        <valeur><entier valeur='34'/></valeur>
      </affectation>
    </corps>
  </blocUnaire>
</programme1>
```

Pour valider le programme, on pourra utiliser la commande suivante dans le répertoire Grammars :

```
java -jar $ILPDIR/Java/jars/jing.jar grammaire1-td2_1.rng Samples/u01-1-td2_1.xml
```

Réponse pour les boucles while :

On reprend le même procédé que pour l'affectation. On crée la nouvelle grammaire dans le répertoire Grammars avec pour nom grammar1-td2_2.rnc :

```
# -*- coding: utf-8 -*-
# **************************************************************************
# ILP - Implantation d'un langage de programmation.
# Copyright (C) 2004-2007 <Jacques.Malenfant@lip6.fr>
# $Id$
# GPL version>=2
# **************************************************************************

# Seconde version intermédiaire du langage étudié: ILP1-TD2-2 On étend la
# grammaire précédente pour accepter des programmes enrichis et l'on
# modifie la notion d'instruction pour accepter la présence de la boucle en
# plus des affectations de ILP1-TD2-1.

include "grammar1-td2_1.rnc"

instruction |= iteration

# La boucle tant-que n'a de sens que parce que l'on dispose maintenant
# de l'affectation, d'où l'utilisation de grammar1-td2_1.rnc comme base.

iteration = element iteration {
   element condition { expression },
   element corps     { instruction + }
}

# fin de grammar1-td2_2.rnc
```

Un exemple de programme :

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<!--  -->
<programme1>
  <!-- test:name description='TD2 : iteration' -->
  <!-- solution de l'exercice du TD2 : programme utilisant l'iteration. -->
  <blocUnaire>
    <variable nom="x"/><valeur><entier valeur="0"/></valeur>
    <corps>
      <iteration>
        <condition>
          <operationBinaire operateur="&lt;">
            <operandeGauche>
              <variable nom="x"/>
            </operandeGauche>
            <operandeDroit>
              <entier valeur="10"/>
            </operandeDroit>
          </operationBinaire>
        </condition>
        <corps>
          <affectation nom="x">
            <valeur>
              <operationBinaire operateur="+">
                <operandeGauche><variable nom="x"/></operandeGauche>
                <operandeDroit><entier valeur="1"/></operandeDroit>
              </operationBinaire>
            </valeur>
          </affectation>
          <invocationPrimitive fonction="print">
            <variable nom="x"/>
          </invocationPrimitive>
          <invocationPrimitive fonction="newline"/>
        </corps>
      </iteration>
    </corps>
  </blocUnaire>
</programme1>
```

Réponse pour les blocs locaux n-aires :

La nouvelle grammaire :

```
# -*- coding: utf-8 -*-
# ************************************************************************
# ILP - Implantation d'un langage de programmation.
# Copyright (C) 2004-2007 <Jacques.Malenfant@lip6.fr>
# $Id$
# GPL version>=2
# ************************************************************************

# Deuxième version du langage étudié: ILP2 pour « Insipide Langage
# Prévu » Il sera complété dans les cours qui suivent.

# On étend la grammaire précédente pour accepter des programmes
# enrichis qui acceptent la présence des blocs locaux.

include "grammar1-td2_2.rnc"
```

```
instruction |= blocLocal

# Un bloc local qui introduit un nombre quelconque (éventuellement nul)
# de variables locales associées à une valeur initiale (calculée avec
# une expression).

blocLocal = element blocLocal {
   element liaisons {
      element liaison {
         variable, element valeur { expression }
      } *
   },
   element corps { instruction + }
}

# fin de grammar1-tme2_1.rnc
```

Un exemple de programme :

```
<?xml version="1.0" encoding="UTF-8"?>
<programme1>
  <!-- solution de l'exercice du TP2 : programme utilisant les blocs
       locaux n-aires. -->
  <!-- $Id$ -->
  <blocLocal>
    <liaisons>
      <liaison>
        <variable nom="x"/><valeur><entier valeur="0"/></valeur>
      </liaison>
      <liaison>
        <variable nom="y"/><valeur><entier valeur="10"/></valeur>
      </liaison>
    </liaisons>
    <corps>
      <iteration>
        <condition>
          <operationBinaire operateur="&lt;">
            <operandeGauche>
              <variable nom="x"/>
        </operandeGauche>
            <operandeDroit>
              <variable nom="y"/>
        </operandeDroit>
      </operationBinaire>
    </condition>
        <corps>
          <affectation nom="x">
            <valeur>
              <operationBinaire operateur="+">
                <operandeGauche><variable nom="x"/></operandeGauche>
                <operandeDroit><entier valeur="1"/></operandeDroit>
          </operationBinaire>
        </valeur>
      </affectation>
    </corps>
      </iteration>
    </corps>
  </blocLocal>
```

```
</programme1>
```

Réponse pour les fonctions :

La nouvelle grammaire :

```
# -*- coding: utf-8 -*-
# ********************************************************************
# ILP - Implantation d'un langage de programmation.
# Copyright (C) 2004-2007 <Jacques.Malenfant@lip6.fr>
# $Id$
# GPL version>=2
# ********************************************************************

# Deuxième version du langage étudié: ILP2 pour « Insipide Langage
# Prévu » Il sera complété dans les cours qui suivent.

# On étend la grammaire précédente pour accepter des programmes
# enrichis (définis par la balise programme1-4) et l'on modifie la
# notion d'instruction pour accepter la présence de fonctions.

include "grammar1-tme2_1.rnc"
start |= programme1-df
expression |= invocation

# Un programme est une suite de définitions de fonctions suivie
# d'instructions sauf que les instructions comportent également des
# blocs locaux (étendant la notion de bloc local unaire précédente),
# et la boucle tant-que.

programme1-df = element programme1-df {
   definitionFonction *,
   instruction
}

# Définition d'une fonction avec son nom, ses variables (éventuellement
# aucune) et un corps qui est une séquence d'instructions.

definitionFonction = element definitionFonction {
   attribute nom     { xsd:Name },
   element variables { variable * },
   element corps     { instruction + }
}

# L'invocation d'une fonction définie.

invocation = element invocation {
   element fonction { expression },
   element arguments { expression * }
}

# fin de grammar1-tme2_2.rnc
```

Un exemple de programme :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<programme1-df>
  <!-- solution de l'exercice du TP2 : programme utilisant les fonctions. -->
  <!-- $Id$ -->
  <definitionFonction nom="f">
    <variables>
      <variable nom="a"/><variable nom="b"/>
    </variables>
    <corps>
      <operationBinaire operateur="+">
        <operandeGauche>
          <variable nom="a"/>
    </operandeGauche>
        <operandeDroit>
          <variable nom="b"/>
    </operandeDroit>
      </operationBinaire>
    </corps>
  </definitionFonction>
  <invocation>
    <fonction>
      <variable nom="f"/>
    </fonction>
    <arguments>
      <entier valeur="10"/>
      <entier valeur="20"/>
    </arguments>
  </invocation>
</programme1-df>
```

2 Manipulation des programmes ILP1 en XML (DOM) et AST

Objectif : comprendre la manipulation de programmes ILP1 sous la forme de documents XML en Java lorsqu'ils sont représentés par des objets selon le standard DOM et sous la forme d'objets de l'arbre de syntaxe abstraite (AST).

Buts :
- Comprendre l'architecture d'un arbre DOM
- Apprendre à le manipuler en Java au sein d'ILP
- Comprendre l'architecture d'un arbre AST
- Apprendre à parcourir un AST

2.1 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

Node
  Document               // Représente un document et mène à l'élément racine
  DocumentFragment       // Séquence de noeuds utile pour la confection d'un document
  DocumentType           // Valeur de l'attribut doctype du document
  ProcessingInstruction  // Représente une instruction de traitement
  CharacterData          // Valeurs terminales, i.e. feuilles
    Comment              // commentaires
    Text
      CDataSection       // chaîne non-interprétée
  Element                // Noeud représentant un sous-arbre
  Attr                   // Noeud représentant un attribut et sa valeur
  Entity                 // définitions
  EntityReference
  Notation
NodeList                   // Liste de noeuds
```

Flot de manipulation de l'arbre DOM

Voir Figure 1.

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

Un programme manipulant un arbre DOM se construit généralement autour de quelques méthodes, chacune capable de traiter un type particulier de nœud. 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 tout d'abord lu comme 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.

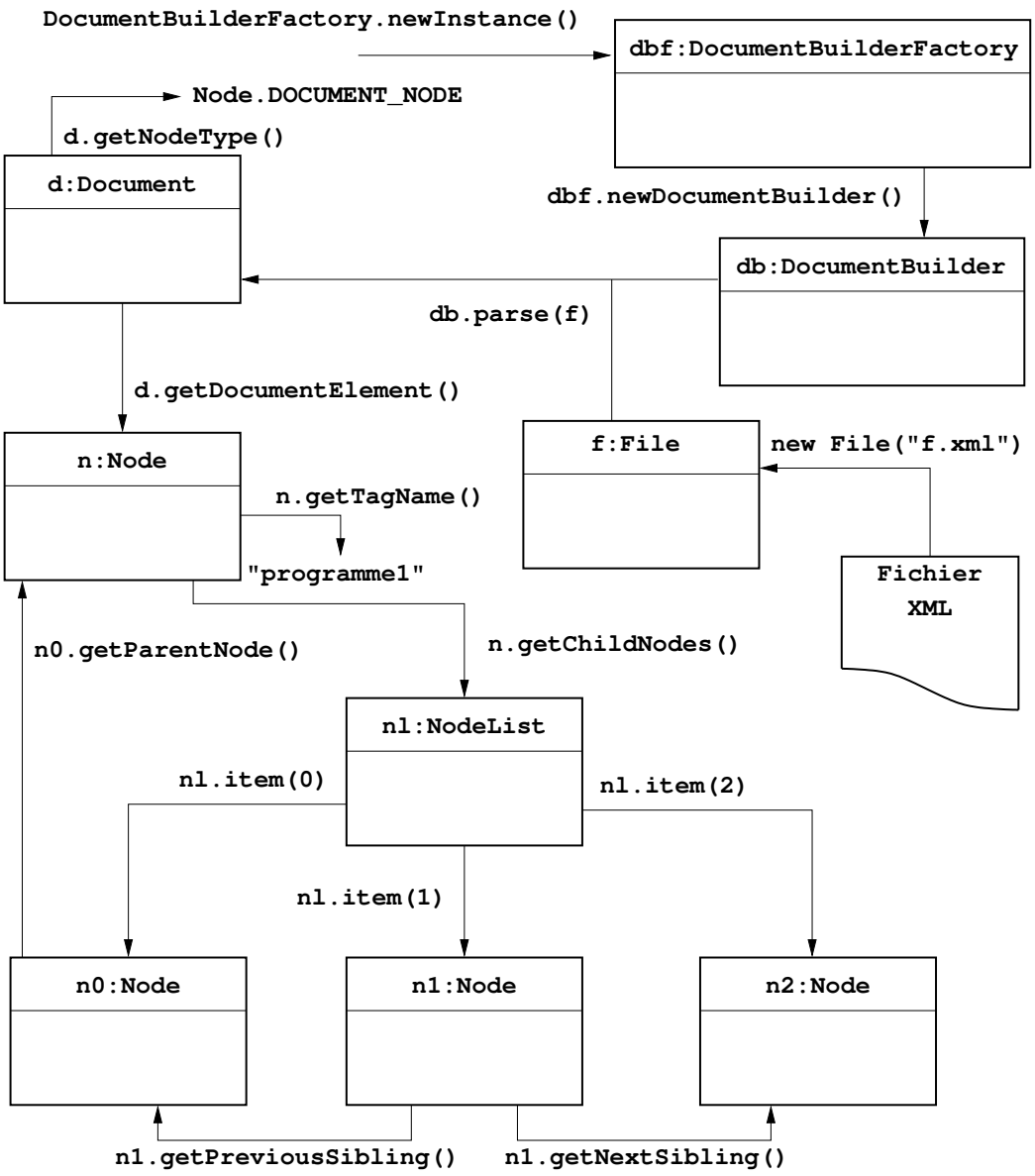

FIGURE 1 – Flot de manipulation de l'arbre DOM

— 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.

2.2 Travail à réaliser

Nous souhaitons compter le nombre de constantes apparaissant dans un programme ILP. Nous envisagerons deux méthodes pour ce faire afin de les comparer : l'une comptera les constantes dans le DOM, l'autre les comptera dans l'AST. Ainsi, vous aurez à écrire une classe `Process` (s'inspirer (hériter ?) de `fr.upmc.ilp.ilp1.Process` peut être utile) implantant l'interface `ICountingConstantsProcess` que voici :

ICountingConstantsProcess.java

```
package fr.upmc.ilp.ilp1tme2;

public interface ICountingConstantsProcess {
    // Compte les constantes a partir du DOM:
    public int getNbConstantesDOM() ;

    // Compte les constantes a partir de l'AST:
    public int getNbConstantesAST() ;
}
```

Pour le comptage de constantes dans l'AST, vous pouvez vous inspirer des classes du paquetage `fr.upmc.ilp.ilp1.eval` sauf qu'au lieu d'évaluer, il s'agit de compter.

N'oubliez pas d'écrire une classe `ProcessTest` permettant d'automatiser vos tests. Pour vous simplifier ces tests, le nombre de constantes peut être précalculé dans les fichiers `u*-1.count` par le petit script bash que voici :

```
for f in u*-1.xml ; do
  grep -c '><(entier |flottant |chaine|booleen )' < $f > ${f%.xml}.count
  echo $f `cat ${f%.xml}.count`
done
```