Projet Grammaire et Langages

Équipe H4314 – 22 avril 2014

Membres de l'équipe
Franck MPEMBA BONI
Grégoire CATTAN
Iler VIRARAGAVANE Jean-Marie COMETS
Pierre TURPIN
Samuel CARENSAC
Van PHAN HAU

Table des matières

1 Analyse lexicale et syntaxique 2

2 Affichage 2

3 Validation sémantique d'un document 2

4 Transformation d'un document 3

5 Validation d'un document 4

1 Analyse lexicale et syntaxique

L'analyse lexicale utilise largement celui fourni pour le TP. Une modification a été fait pour supprimer le token COLON. Celui-ci est en effet directement intégré au niveau du token NOM (valide selon XML) et l'étude de l'espace de nom se faire alors uniquement dans l'analyse sémantique.

L'analyse syntaxique utilise les règles du XML et incorpore également le mot clé *error* dans certaines règles de grammaire afin d'avoir une exécution de Bison/Yacc laxiste. Les erreurs peuvent ainsi être détecté sans pour autant stopper le parsing et le traitement du flot de tokens. L'idée est alors de faire remonter toutes les erreurs syntaxique jusqu'à la règle racine afin de libérer proprement les ressources allouées et d'indiquer l'erreur à l'utilisateur.

La construction de l'arbre d'objet représentant l'XML se fait en parallèle de la lecture du flot de donnée. C'est pourquoi, il est nécessaire de remonter toutes les erreurs pour libérer les objets construits.

2 Affichage

L'affichage du document se fait de manière récursive en effectuant un parcours en profondeur de l'arbre. Chaque noeud parcouru ajoute sa sortie à un flux *std : :ostream*.

Chacun des éléments hérite d'une classe de base qui permet d'ajouter cette fonction *str()* qui permet de rendre le document sous forme d'un flux. L'aspect polymorphique permet alors à chaque élément de décrire son rendu lui-même. L'algorithme devient alors largement simplifié.

3 Validation sémantique d'un document

L'arbre XML d'un document est validé sémantiquement en effectuant pour chaque noeud les vérifications suivantes :

1. Validation du noeud.
2. Validation des noeuds contenus (enfants).

Pour cela, on considère que chaque noeud de l'arbre *peut* être validé une fois construit par la grammaire, et sait comment se valider lui et ses enfants. Au niveau de l'implémentation, cela constitue en une simple méthode virtuelle pour les noeuds de l'arbre, appelée au niveau de la racine du document.

4 Transformation d'un document

Un document est transformé à partir d'un fichier XSL.

Le document XSL est d'abord lu puis parsé en arbre XML. C'est à partir de cet arbre XML qu'on pourra effectuer la transformation du fichier XML source en rendu XML à partir d'un modèle XSL.

Il est important de noter que nous nous sommes limités au niveau de l'interprêtation des fichiers XSL :

- Nous ne gérons que certaines directives XSL : *template*, *for-each*, *value-of* et *apply-templates*, qui seront détaillés par la suite. Il faut aussi noter que bien que nous ne traitons pas la directive *stylesheet*, nous requérons sa présence dans le fichier XSL. Dans le cas d'une erreur d'interprétation de ces directives, ou de la présence d'une directive non-reconnue, nous recopions uniquement le texte correspondant.
- En ce qui concerne la directive template, elle, permet de spécifier un certain arbre XSL à appliquer pour un certain noeud source. C'est la base de la transformation XSL, et le noeud source est spécifié à partir de l'attribut "match".
- En ce qui concerne la directive for-each, elle permet de répéter un certain arbre XSL pour toutes les balises correspondant à la sélection précisée, via l'attribut "select". Celui-ci ne peut correspondre qu'à une balise XML directement présente dans le noeud d'application du for-each.
- En ce qui concerne la directive value-of, elle ressemble beaucoup à la balise for-each (toujours en spécifiant avec l'attribut "select"), mais au lieu de répéter, applique l'arbre XSL uniquement pour le premier noeud correspondant.
- En ce qui concerne la directive apply-templates, elle permet d'appliquer un certain arbre XSL au niveau du noeud courant, soit en spécifiant celui à appliquer (attribut "select"), soit sans le spécifier, auquel cas l'arbre XML source (fichier à transformé) sera parcouru récursivement en appliquant tous les templates possibles applicables.

La plupart des algorithmes utilisés sont des simples applications récursives, traitant en fonction du type d'élément racine courant (simple, multiple, texte).

Voici l'algorithme général de transformation d'un fichier XML :

```
fonction XSL::transformer_xml(ArbreXML xml)
  Si xml n'est pas nul
    transformer_element_multiple(xml)
  Sinon
    appliquer_tous_les_templates(xml)

fonction XSL::transformer_element(Element elem)
  Si elem est un element_xsl
    transformer_element_xsl(elem)
  SinonSi elem est un element_multiple
    elem.afficher_debut()
```

```
    transformer_element_multiple(elem)
    elem.afficher_fin()
  Sinon
    elem.afficher()

fonction XSL::transformer_element_multiple(ElementMultiple elem)
  Pour chaque tag t de elem
    Si t est un element
      transformer_element(t)
    Sinon
      t.afficher()

fonction XSL::transformer_element_xsl(ElementXSL elem)
  oper = operation_xsl(elem)
  Si oper n'est pas nulle
    oper.effectuer()
  Sinon
    elem.afficher()
```

5 Validation d'un document

Un document est validé grâce à un fichier XSD.

Le document XSD est d'abord lu, parsé en arbre XML, puis transformé en graphe nommée d'expressions régulières. Ce graphe est modélisé par une *std : :map* de *std : :string* vers *Node*. L'index représente le nom du tag XML et la valeur, les conditions de validation de ce tag. Ce dernier est développer par une structure C++ avec 3 *std :string* :

- **reg_tag** : Expression régulière sur les tags intérieurs
- **reg_attr** : Expression régulière sur les attributs (non utilisé finalement)
- **reg_content** : Expression régulière sur le contenu

La construction du graphe se fait en parcourant uniquement le document XSD. Une autre *std : :map* est fourni pour contenir les types déclarés pendant la construction pour gérer les tags *complexType*. Afin de gérer les déclarations des types après leur utilisation, la construction est faite deux fois.

Voici l'algorithme de construction du graphe :

```
fonction construct_schema(SchemaTag S, Graph G, Type T)
    reg_tag = ""
    Pour chaque tag t de S ayant pour nom "element"
        Si t a un attribut "name"
            reg_tag += "<" + t.attributes["name"] + ">|"
            Si t est un tag inline
                construct_empty_tag(t, G, T)
            Sinon
                construct_composite_tag(t, G, T)
```

```
        Sinon si t a un attribut "ref"
            min = t.attributes["minOccurs"] ou "1"
            max = t.attributes["maxOccurs"] ou "1"
            Si max = "unbounded"
                max = ""
            reg_tag += "(<" + t.attributes["ref"] + ">){" + min + "," + max +
                    "}|"
    Pour chaque tag t de S ayant pour nom "complexType"
        construct_complex_type(t, G, T)
    Si reg_tag not empty
        reg_tag.pop_last() # On enlève le dernier caractère |
    G[""] = Node(reg_tag=reg_tag) # Représente la racine du graphe

fonction construct_empty_tag(EmptyTag E, Graph G, Type T)
    node = Node()
    Si E.attributes["type"] = "string"
        node.reg_content = REGEX_STRING
    Sinon si E.attributes["type"] = "date"
        node.reg_content = REGEX_DATE
    Sinon si E.attributes["type"] dans T
        node = T[E.attributes["type"]]
    G[T.name] = node

fonction construct_composite_tag(CompositeTag C, Graph G, Type T)
    Si premier element P de C est nommé "complexType"
        G[C.name] = construct_complex_type(P, G, T)

fonction construct_complex_type(ComplexTypeTag C, Graph G, Type T)
    Si premier element P de C est nommé "sequence"
        node = construct_sequence(P, G, T)
    Si premier element P de C est nommé "choice"
        node = construct_choice(P, G, T)
    Si C a l'attribut "name"
        T[C.attributes["name"]] = node
    retourner node

fonction construct_sequence(SequenceTag S, Graph G, Type T)
    reg_tag = ""
    Pour chaque tag t de S ayant pour nom "element"
        Si t a un attribut "name"
            reg_tag += "<" + t.attributes["name"] + ">"
            Si t est un tag inline
                construct_empty_tag(t, G, T)
            Sinon
                construct_composite_tag(t, G, T)
        Sinon si t a un attribut "ref"
            min = t.attributes["minOccurs"] ou "1"
            max = t.attributes["maxOccurs"] ou "1"
            Si max = "unbounded"
                max = ""
```

```
            reg_tag += "(<" + t.attributes["ref"] + ">){" + min + "," + max +
                    "}"
    retourner Node(reg_tag=reg_tag)

fonction construct_choice(ChoiceTag S, Graph G, Type T)
    reg_tag = ""
    Pour chaque tag t de S ayant pour nom "element"
        Si t a un attribut "name"
            reg_tag += "<" + t.attributes["name"] + ">|"
            Si t est un tag inline
                construct_empty_tag(t, G, T)
            Sinon
                construct_composite_tag(t, G, T)
        Sinon si t a un attribut "ref"
            min = t.attributes["minOccurs"] ou "1"
            max = t.attributes["maxOccurs"] ou "1"
            Si max = "unbounded"
                max = ""
            reg_tag += "(<" + t.attributes["ref"] + ">){" + min + "," + max +
                    "}|"
    Si reg_tag not empty
        reg_tag.pop_last() # On enlève le dernier caractère |
    retourner Node(reg_tag=reg_tag)
```

Une fois le graphe de règles créé, le document XML est validé par un parcours en profondeur :

```
  fonction validate(Element C, Graph G)
      Si C.name n'est pas dans G
          retourner Faux

      regle = G[C.name]
      tags = ""
      Pour chaque Element E nommée nom dans C
          tags += "<" + nom + ">"
          Si validate(E, G) est Faux
              retourner Faux
      Si regle.reg_tag existe et G[C.name].reg_tag.not_match(tags)
          retourner Faux

      Si regle.reg_content existe
          regex = "<" + C.name + ">" \
            + regex_blank + "(" + regle.reg_content + ")" + regex_blank \
            + "</" + C.name + ">"
          Si regex.not_match(C.toString())
            retourner Faux
      Retourner Vrai
```

On appel cette fonction avec l'élément racine du document. Elle retourne vrai ou faux selon la validité du document XML par rapport aux document XSD.