Rapport du projet XMLLiteParser

Modélisation

Mathis Deloge, Antoine Petot, Ange Picard

1 Descriptif du sujet

Un parseur / validateur XML-Lite est un programme capable de lire un fichier, d'indiquer s'il vérifie la norme XML-Lite et si oui, de l'analyser et de retenir sa structure ainsi que son contenu. Pour nous permettre de concevoir un programme réalisable, notre parseur / validateur opère sur un langage simplifié de XML, le XML-Lite conçu pour faciliter l'utilisation, les performances ainsi que les normes de conformité (XML 1.0).

1.1 Le XML-Lite

Pour être considéré comme du XML-Lite, les fichiers parsés / validés par notre programme doivent respecter certaines règles :

- Une balise possède un nom.
- Une balise doit être ouverte puis fermée.
- Une balise peut contenir du texte.
- Une balise peut contenir d'autre balises.
- L'ordre des balises filles n'a pas d'importance et tout le texte contenu dans une balise est regroupé en un seul bloc.
- Une balise fille doit être fermée avant la fermeture de la balise parent.
- Une balise peut contenir une balise du même nom.
- Un document doit commencer par l'ouverture d'une balise se fermant à la fin du document.

1.2 Exemple de fichiers XML-Lite

1.2.1 Fichier XML-Lite correct

```
<FirstTag>
    <ChildTag>
        <AnotherChildTag>
        </AnotherChildTag>
    </ChildTag>
    <tag>
    </tag>
</FirstTag>
```

1.2.2 Fichier XML-Lite Faux

```
  <FirstTag>
      <SecondTag>
        <EndTag>
          <AloneTag>
        </>
  </FirstTag>
  </SecondTag>
Un peu de texte
```

1.3 Structure du document

Le parseur / validateur doit être capable de lire n'importe quel fichier XML-Lite mais doit aussi être en mesure d'attendre une certaine structure de document grâce à l'ajout d'un fichier .dtd appelé schéma. Grâce aux fichiers schéma, le parseur / validateur connais avec plus de finesse les balises filles autorisées ou non pour chaque balises. C'est une sorte de modèle qui permettra la validation du fichier XML-Lite.

2 Journal de bord

2.1 Séance 1

Lors de la première séance, nous avons tout d'abord effectué le choix de sujet. Le parseur / validateur XML-Lite nous a intéressé étant donné le grand nombre de programmes fonctionnant avec XML pour la persistance et la souplesse de ce format de base de données, nous étions intéressé de découvrir les notions de bases du XML.

Par ailleurs, durant cette séance, nous avons trouvé des informations sur les validateurs de documents et avons pensé à implémenter un automate fini pour modéliser notre validateur. Le design objet "state pattern" semblait particulièrement adapté.

2.2 Séance 2

Lors de la deuxième séance, nous avons modélisé l'automate fini schématiquement, puis, nous l'avons implémenté. Il est utilisé pour valider le document. Nous avons également codé les différents états.

Exemple d'un état du validateur

```
1  public class NewTag implements State {
2      @Override
3      public State transition(char c) {
4          if (c == '/')
5              return new NewClosingTag();
6          else if ((c != '<') && (c != '>')) {
7              XMLLiteParser.getInstance().fillBuffer(c);
8              return new NewTagName();
9          } else
10             return new Error();
11     }
12
13     @Override
14     public boolean isFinal() {
15         return false;
16     }
17 }
```

Puis, nous avons réfléchis à la structure mathématique du pasreur, nous sommes vite arrivé à celle d'un arbre. Cette structure à l'avantage d'être facile à designer en objet. Nous avons donc implémenté deux classes :

- XMLLiteNode : Pour représenter une feuille ou un nœud de l'arbre.
- XMLLiteParser : Pour construire l'arbre.

Il a également fallu implémenter un buffer afin de stocker caractère par caractère les informations provenant des états du validateur.

2.3 Séance 3

Après avoir implémenté le parseur en structure d'arbre lors de la séance 2, nous avons pu interfacer une IHM basée sur les noeuds et feuilles de l'arbre nous permettant de faire une représentation claire et précise du fonctionnement du parseur / validateur qui nous serivra principalement lors de la présentation de projet.

L'implémentation de la classe JTree nous permet alors de représenter visuellement la structure du fichier XML-Lite analysé formaté en structure d'arbre.

2.4 Séance 4

Lors de la séance 4, nous avons dû faire face à une erreur n'arrivant uniquement lors de la lecture de gros fichiers. En effet, la validation de ces fichiers posait problème pour les dernières balises, le Parser rajoutait des balises filles au root node, à la fin, alors qu'elles n'existaient pas...

Nous avons pensé que ce problème provenait sans doute de notre classe qui lit les fichiers. Pour vérifier son fonctionnement, nous avons donc essayé de lire le fichier et de réécrire son contenu directement dans une copie. Et il est apparu qu'en effet, notre lecteur rajoutait des mauvais caractères en fin de fichier. Ceci a cause d'un buffer mal géré. Nous avons donc réécrit la classe de lecture des fichiers caractère par caractère avec des objets mieux adaptés.

3 Choix du modèle mathématique

3.1 Le modèle mathématique

Pour nous permettre de parcourir rapidement un fichier XML-Lite, nous avons opté pour le développement d'un automate fini.

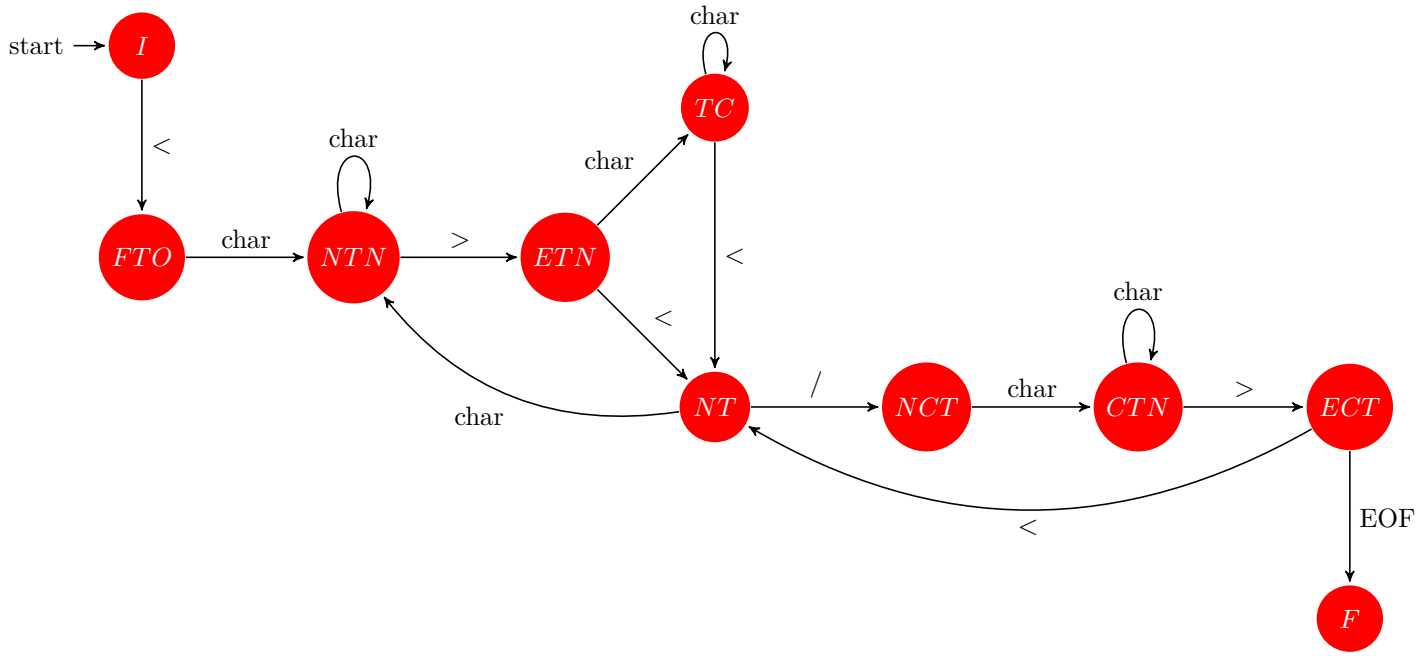
Tout d'abord puisque grâce à la simplicité et la rigidité du langage XML-Lite, il y a très peu d'états différents lors de la lecture d'un fichier. De plus, les transitions entre états se font uniquement grâce à la différenciation des caractères '<', '>', '/' et le reste.

Cette façon de parcourir un fichier XML-Lite caractère par caractère s'est avérée très rapide (exécution en 13ms pour un fichier XML-Lite de près de 700Mo).

Enfin, ce modèle mathématique avec une implémantation et un débuggage relativement simple permet également une amélioration facile du programme grâce à la différenciation de tous les états dans différentes classes.

3.2 Représentation de l'automate fini

3.2.1 Schéma



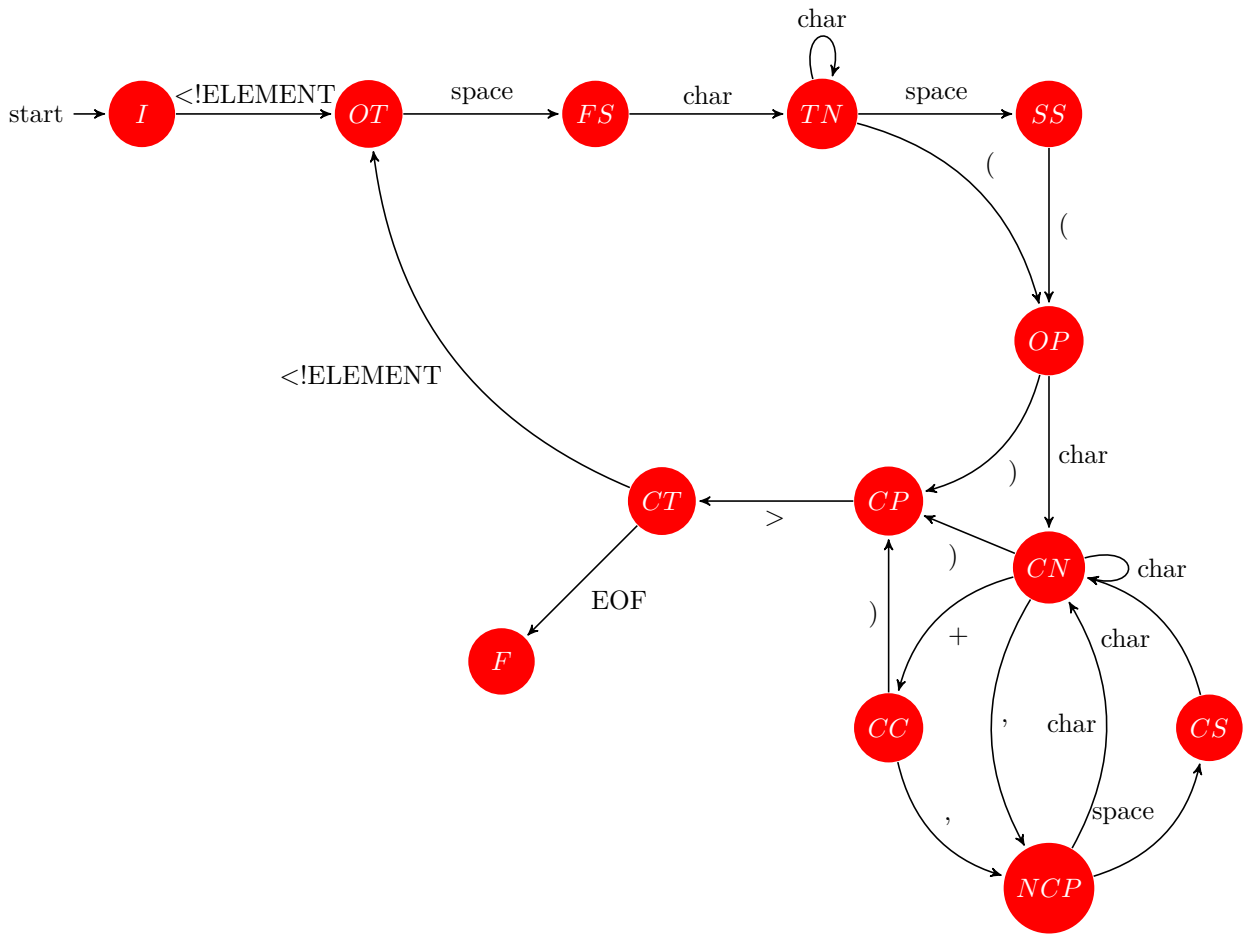
3.2.2 Description des états

I Initial
FTO First Tag Opening
NTN New Tag Name
ETN End Tag Name
TC Text Content
NT New Tag
NCT New Closing Tag
CTN Colsing Tag Name
ECT End Closing Tag
F Final

3.3 Automate de validation du fichier DTD

Afin d'utiliser un fichier DTD, nous devons vérifier ce fichier DTD. Nous avons ainsi implémenter un automate fini pour vérifier la structure du schéma.

3.3.1 Schéma



3.3.2 Description des états

I Initial

OT Opening Tag

FS First Space

TN Tag Name

SS Second Space

OP Opening Parenthesis

CN Child Name

NCP Next Child Point

CS Child Space

CP Closing Parenthesis

CT Closing Tag

F Final

4 Prolongements possibles

4.1 Permettre le débuggage du fichier XML.

Le modèle d'automate permet une vérification à la volée dans le parseur.

4.2 Permettre au validateur de s'accorder à un schéma prédéfini.

Validation de la structure en fonction de contraintes sur l'arbre.

4.3 Modifier le validateur afin qu'il prenne en compte un schéma.

4.4 Ajouter un interpréteur suivant le schéma d'un générateur de QCM.

Création des contraintes qui constituent le schéma en fonction d'un document DTD. Implémentation d'un deuxième automate fini.

5 Conclusion

6 Comment ajouter du code ?

6.1 Comme ça

```
1  package XMLLiteParser;
2
3  import XMLLiteParser.Core.Node;
4  import XMLLiteParser.Core.Parser;
5  import XMLLiteParser.Core.TransitionSystem;
6  import XMLLiteParser.Gui.TreeView;
7  import XMLLiteParser.QCM.QCM;
8  import XMLLiteParser.QCM.QCMInterpreter;
9
10 import javax.swing.tree.DefaultTreeModel;
11 import java.util.Date;
12
13 /**
14  * Created by MrMan on 12/09/2016.
15  */
16 public class main {
17     public static void main(String[] args) {
18         long startTime = System.currentTimeMillis();
19
20         TransitionSystem ts = new TransitionSystem();
21         Node rootNode = null;
22
23         try {
24             ts.parseFile("XMLDocs\\Ok.xmll");
25             rootNode = Parser.getInstance().getRootNode();
26
27             QCMInterpreter qcmInterpreter = new QCMInterpreter();
28             QCM qcm = qcmInterpreter.interpreteTree(rootNode);
29             System.out.println("WOW");
30         } catch (Throwable e) {
31             e.printStackTrace();
32         }
33
34         System.out.println("Document␣validated␣in␣" + (new Date().getTime() - startTime) + "␣ms
               ");
35
36
37         DefaultTreeModel tm = new DefaultTreeModel(rootNode);
38         TreeView tv = new TreeView(tm);
39
40         tv.setVisible(true);
41         }
42     }
```

6.2 Ou comme ça

yolo

```
1 class HelloWorldApp {
2     public static void main(String[] args) {
3         System.out.println("Hello␣World!"); // Display the string.
4         for (int i = 0; i < 100; ++i) {
5             System.out.println(i);
6         }
7     }
8 }
```