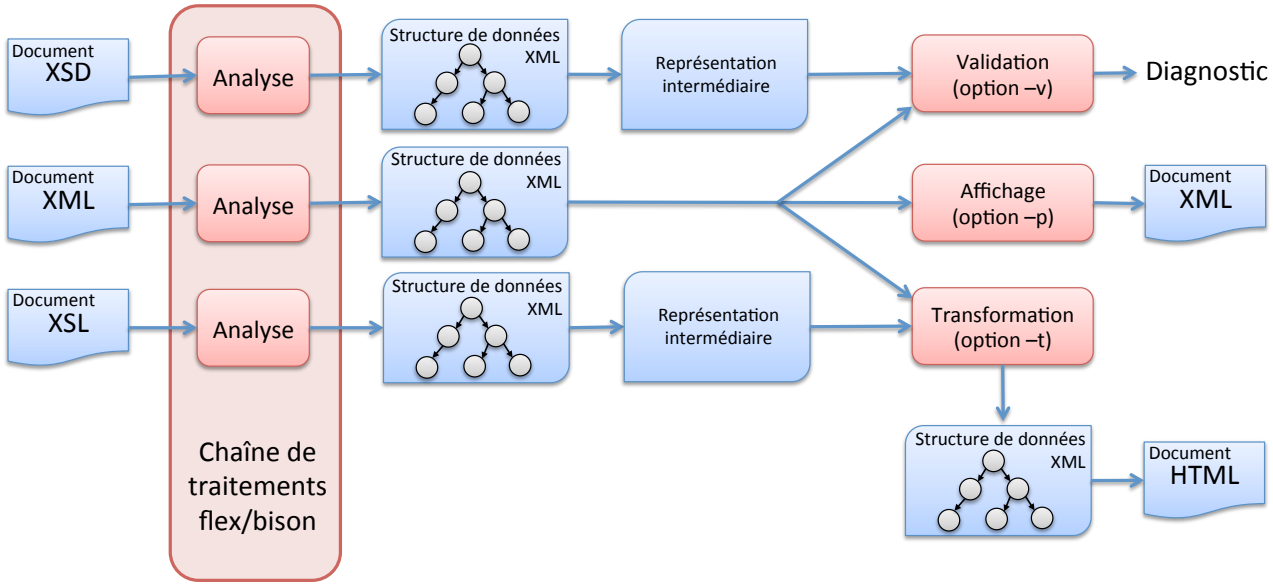
Grammaires et Langages - Projet
Développement d'un processeur XML

Les différentes recommandations du W3C qui définissent XML suivent un schéma classique en compilation : analyse syntaxique et sémantique, création d'un arbre, transformation de cet arbre et génération d'un résultat. Nous allons développer un processeur XML en utilisant les outils Flex et Bison et le langage C++ en limitant volontairement les fonctionnalités de validation et transformation. Pour tester la solution, nous utiliserons un framework de test comportant une liste pré-établie de tests fonctionnels.

1 Architecture globale de l'outil



Le logiciel se présente sous la forme d'un outil en ligne de commande et comporte trois modes de fonctionnement :

1. Parsage/affichage (option en ligne de commande -p)
2. Validation d'un document XML par rapport à un schéma XSD (option en ligne de commande -v)
3. Transformation d'un document XML par rapport à une feuille de style XSL (option en ligne de commande -t)

2 Étapes de travail

Analyse syntaxique XML. Finir le travail commencé en TD : compléter la grammaire d'un élément XML pour y ajouter les attributs, les en-têtes, les commentaires et les sections CDATA. Une ossature de fichier Bison (xml.y) vous est fournie et utilise un module de reconnaissance des jetons écrit en Flex (xml.l). Compiler en utilisant le makefile fourni.

Restrictions : Vous ne traiterez pas les objets XML de type référence, ni les DTD directement incluses dans la balise DOCTYPE. De plus, on considèrera que les Processing Instructions ne peuvent contenir que des listes d'attributs et sont donc moins générales que ce que le W3C spécifie.

Conception de la structure de données. Définir la structure d'un arbre XML. Pour cela, définir une hiérarchie de classes en C++ (compléter le travail effectué en TD).

Construction de la structure de données depuis l'analyse. Définir les actions de construction de la structure dans bison en utilisant les actions dans les règles d'analyse. A la fin de l'analyse, on doit obtenir une structure interne représentant le document XML. Réaliser l'affichage du contenu de la structure XML de façon à vérifier que l'on peut restituer à l'identique (ou au moins une représentation équivalente) les documents analysés.

Validation d'un document. Un schéma XSD est un fichier XML qui décrit la structure d'un document. Afin de vérifier la validité d'un document par rapport à son schéma, nous utiliserons des expressions régulières associées à chaque élément XML. Par exemple, la portion de déclaration suivante :

```
<xsd:element name="choice">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="a" type="xsd:string" />
      <xsd:element name="b" type="xsd:string" />
      <xsd:element name="c" type="xsd:string" />
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

indique que l'élément `element` comporte au choix un élément `a`, `b`, ou `c`. Imaginons que l'on représente ce choix par l'opérateur `|` d'une expression régulière qui pourrait ressembler à cela :

```
^((<a>)|(<b>)|(<c>))$
```

La validation de l'élément `element` consistera à tester si cette expression régulière [1] est bien en correspondance avec la réalité. Pour cela, on va générer une chaine de caractères qui représente les fils de l'élément. Exemple avec la portion de code XML suivante :

```
<choice>
  <a>a</a>
</choice>
```

qui sera représentée par `"<a>"` et qui correspond bien à l'expression régulière. La validation du document va donc consister à construire un catalogue d'expressions régulières associées aux éléments grâce à parcours de l'arbre XSD, et dans un deuxième temps la vérification sur chacun des noeuds de l'arbre XML de vérifier que la liste des fils correspond bien à l'expression stockée.

Restrictions : Afin de pouvoir faire le projet dans le temps imparti, les restrictions suivantes seront appliquées :

- Traitement des types de base `string` et `date`
- Traitement des constructions `choice` et `sequence`
- La construction `mixed` est optionnelle
- La validation des attributs est optionnelle

1. vous utiliserez la bibliothèque d'expressions régulière POSIX ou boost pour votre implémentation, car l'implémentation g++/C++11 est incomplète

Transformation d'arbre. Dans un premier temps, il faudra bien comprendre le fonctionnement des templates lors de la transformation XSL. Utiliser un exemple simple pour cela et dérouler l'algorithme. Formaliser ensuite cet algorithme sur papier. Vous aurez besoin comme pour la validation d'une représentation intermédiaire qui donnera un catalogue de tous les templates.

La transformation se fera ensuite en deux temps :
— Construction du catalogue de templates grâce au parcours du document XSL ;
— Application des templates au document XML en entrée.

L'algorithme de transformation peut se résumer aux étapes suivantes :

1. Partir de la racine XML
2. Chercher un template applicable
3. Traiter le template ce qui peut entrainer le traitement d'autres noeuds (notamment avec les `apply-templates`) pour lesquels on repartira à l'étape 2
4. S'il n'y a pas de template applicable, recopier (uniquement le texte)

Restrictions : Vous traiterez au minimum dans les documents XSL les cas suivants :
— L'attribut `match` de l'élément `template` est un nom d'élément ou `/`
— Les `apply-templates` avec un attribut `select` ou non
— Les `for-each` avec un attribut `select`

3 Intégration

Comme indiqué précédemment, l'outil final est un outil en ligne de commande qui possède trois modes de fonctionnement dont vous trouverez ici les spécifications.

Parsage/affichage. La syntaxe d'utilisation est la suivante :

```
xmltool -p fichier.xml
```

Le comportement sera d'analyser le fichier donné en argument puis le réaffichera sur la sortie standard. Les différents cas de fonctionnement de ce mode d'utilisation sont détaillés dans le plan de tests fourni.

Validation. La syntaxe d'utilisation est la suivante :

```
xmltool -v fichier.xml fichier.xsd
```

Le comportement sera d'analyser les fichiers donnés en argument, et de tenter de valider `fichier.xml` par rapport au schéma `fichier.xsd`. Les différents cas de fonctionnement de ce mode d'utilisation sont détaillés dans le plan de tests fourni.

Transformation. La syntaxe d'utilisation est la suivante :

```
xmltool -t fichier.xml fichier.xsl
```

Le comportement sera d'analyser les fichiers donnés en argument, et de tenter de transformer le fichier `fichier.xml` avec la feuille de style `fichier.xsl`. Le résultat sera ensuite affiché sur la sortie standard. Les différents cas de fonctionnement de ce mode d'utilisation sont détaillés dans le plan de tests fourni.

Plan de tests. Un plan de tests est fourni. Ce plan de tests est minimal et permet de vérifier que votre outil est bien conforme aux spécifications de base qui sont données (on peut même dire que le plan de test fait partie des spécifications). Vous avez la possibilité de le compléter notamment pour la partie validation et transformation.

4 Livrables

1. Un document de conception qui contiendra
   — la description des structures de données sous forme d'un diagramme de classes
   — les algorithmes de validation et de transformation
2. Le dossier de réalisation (application testable sur les machines LINUX utilisées en projet) sera déposé dans le répertoire /public/RenduProcesseurXML/Groupe (sur iftpserv2) contenant
   — tous les sources de votre application
   — le makefile avec une cible par défaut qui crée les exécutables et une cible « test » qui lance les jeux de tests (une première ébauche de ce fichier est fournie)
   — vos jeux de tests dans le formalisme du framework (et les fichiers nécessaires)
3. Une présentation orale montrant l'état de l'avancement du projet et l'exécution commentée des tests dans divers cas.

5 Annexes

Pour démarrer

Commencer par copier l'archive suivante dans votre répertoire personnel :

```
/public/tp/ProcesseurXML.tgz
```

la décompresser et lancer la compilation grâce à `make`. Le programme compile et peut s'exécuter mais n'est pas du tout conforme aux spécifications. Il lit l'entrée standard par défaut.

Fichiers fournis

`commun.h xml.l xml.y main.cpp Tests* files* Makefile readme.txt faq.txt`

Le répertoire `Tests` contient le framework de tests avec un plan de tests fonctionnels. `files` est un répertoire contenant les fichiers nécessaires au plan de tests. Le fichier `readme.txt` vous aidera à démarrer ainsi que le fichier `faq.txt`.

Documentations

Documentation Bison

`http://www.gnu.org/software/bison/manual/`

Spécification XML

`http://www.w3.org/TR/xml/`