Transformations XSL

Fabrice Rossi

12 mars 2003

Les exercices présentés dans ce document sont construits en partie grâce aux exercices de [5] et [6].

1 Instructions spécifiques au CRIO UNIX

Au début de chaque séance :

1. dans chaque terminal, utilisez tcsh (en tapant `tcsh`)
2. effectuez les réglages nécessaires au bon fonctionnement des différents programmes utilisés, en tapant `source /home/perm/ufrmd/rossi/xml/env.csh`

On lance une transformation par la commande `transform entrée style sortie`. Cette transformation est basée sur le fichier `style` et s'applique au fichier `entrée`. Le fichier `sortie` contient le résultat de la transformation. Il est facultatif : en cas d'absence, le résultat s'affiche directement à l'écran.

2 Une première approche de XSLT

2.1 Introduction

XSLT (cf [1]) est un dialecte XML qui permet de décrire des transformations abitraires à réaliser sur des documents XML. L'idée est de permettre de passer facilement d'un dialecte XML à un autre format (souvent XML, mais en fait assez libre), par exemple pour traduire un fichier XML en une page HTML. Le langage proposé est assez délicat à maîtriser pour diverses raisons :

- il est basé sur XPath (cf [2]), un langage à la syntaxe non XML qui permet de décrire des ensembles de nœuds dans un document XML. XPath est très puissant et assez compact, mais aussi assez complexe ;
- XSLT est un langage déclaratif : on donne des règles de transformation que le moteur se charge d'appliquer, sans qu'on écrive la séquence des opérations. Cette façon de procéder est déroutante pour de nombreux programmeurs ;
- XSLT est un langage fonctionnel : beaucoup de transformations s'expriment de façon récursive, ce qui n'est pas toujours facile à appréhender.

Comme DOM, XSLT est basé sur une représentation par arbre du document XML traité. Exactement comme dans DOM, la racine de l'arbre XSLT représente le document tout entier, alors que la racine XML est le fils principal de la racine XSLT.

Notons qu'il est illusoire dans ce tutoriel de vouloir dépasser un niveau élémentaire. Le lecteur intéressé par les subtilités pourra se reporter aux deux normes déjà citées ([1] et [2]), ainsi qu'à [3] qui est particulièrement complet et clair.

2.2 Un programme XSLT

Un programme XSLT est constitué d'un ensemble de règles de transformation. Chaque règle comporte deux parties :

1. un motif (*pattern*) qui précise les nœuds de l'arbre XSLT auxquels la règle peut s'appliquer ;
2. un modèle de résultat (*template*) qui indique ce que le moteur XSLT doit produire (en général un fragment de document XML) quand il applique la règle.

Le fonctionnement d'un programme XSLT est basé sur la notion de nœud courant et de liste de nœuds courants. A l'origine, la liste de nœuds courants est réduite à la racine de l'arbre XSLT. Le résultat du programme est celui de la transformation de cette racine. A un moment donné de l'exécution du programme, on doit calculer le résultat du programme sur une liste de nœuds courants. Par définition, ce résultat est obtenu en concaténant le résultat du programme sur chacun des nœuds de la liste. Enfin, le résultat du programme pour un nœud est obtenu de la façon suivante :

1. le moteur XSLT cherche dans les règles de transformation celle dont le motif s'adapte le mieux au nœud ;
2. le résultat du programme sur le nœud est alors le *template* associé à la règle.

Pour fixer les idées, commençons par un exemple basique, fonctionnant à partir du fichier XML suivant :

liste-personnes.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<liste>
<personne>
<prénom>John</prénom><nom>Doe</nom>
</personne>
<personne>
<prénom>Robert</prénom><nom>Durand</nom>
</personne>
<personne>
<prénom>Eléanore</prénom><nom>Dupuis</nom>
</personne>
</liste>
```

On veut produire une version HTML de ce document. La première étape est bien sûr de créer l'entête du document HTML. Pour ce faire, on propose le fichier XSLT suivant :

perso2html1.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- racine du document, précise le sens du préfixe xsl -->
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- format de sortie, ici html -->
  <xsl:output method="html"
              doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"
              indent="yes"/>
  <!-- patron pour la racine du document de départ -->
  <xsl:template match="/">
    <html>
      <head><title>Liste de personnes</title>
      </head>
      <body>
        <h1>Liste de personnes</h1>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

On constate en lisant les commentaires qu'une règle est déclarée grâce à l'élément `template`. Le motif est donné par le contenu de l'attribut `match`. C'est une expression au format XPath. Dans notre exemple, l'expression se réduit à `/` qui correspond à l'unique nœud racine (de l'arbre XSLT). Enfin, le modèle de résultat est le contenu de l'élément `template`, ici du code HTML. Quand on applique la transformation au fichier XML, on obtient le fichier HTML suivant :

liste1.html

```

<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

      <title>Liste de personnes</title>
   </head>
   <body>
      <h1>Liste de personnes</h1>
   </body>
</html>
```

2.3 Les appels récursifs

L'exemple proposé dans la section précédente est très limité car il ne traite que la racine. Pour aller plus loin, il faut pouvoir choisir la liste des nœuds courants, ce qui correspond en XSLT à un appel récursif : l'instruction `apply-templates` utilisée dans un modèle de résultat est remplacée par le résultat du programme de transformation appliqué à la liste des enfants du nœud courant. Voici un exemple d'utilisation de cette instruction :

perso2html2.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"
              doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"
              indent="yes"/>
  <xsl:template match="/">
    <html>
      <head><title>Liste de personnes</title>
      </head>
      <body>
        <h1>Liste de personnes</h1>
        <ul>
        <!-- appel récursif -->
        <xsl:apply-templates/>
        </ul>
      </body>
    </html>
  </xsl:template>
  <!-- patron pour un élément personne -->
  <xsl:template match="personne">
    <li><xsl:value-of select="prénom"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="nom"/></li>
  </xsl:template>
</xsl:stylesheet>
```

Le programme proposé produit le résultat suivant :

liste2.html

```

<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

      <title>Liste de personnes</title>
```

```
9     </head>
10    <body>
11       <h1>Liste de personnes</h1>
12       <ul>
13
14          <li>John Doe</li>
15
16          <li>Robert Durand</li>
17
18          <li>El&eacute;anore Dupuis</li>
19
20       </ul>
21    </body>
22 </html>
```

Pour comprendre ce qui se passe, il nous faut d'abord expliciter quelques instructions XSLT et XPath utilisées :

- `value-of` est une instruction XSLT qui est remplacée par la traduction en texte de son attribut `select` qui doit être une expression XPath ;
- l'expression XPath `prénom` doit être interprétée comme ceci : elle fabrique l'ensemble des nœuds qui sont des éléments `prénom` et qui sont fils du nœud **contexte**. C'est une abbréviation de `./prénom`, car `.` désigne le nœud contexte alors que `/` est le symbole XPath qui signifie fils. Le nœud contexte est déterminé par les instructions qui englobent l'expression XPath. Dans notre exemple, c'est en fait le nœud courant, mais ce n'est pas toujours le cas ;
- l'expression XPath `personne` correspond à l'ensemble des fils du nœud contexte qui sont des éléments `personne`. Nous allons voir comment la règle correspond à ce motif est choisie par le moteur XSLT ;
- enfin, l'instruction `text` permet d'insérer du texte dans le résultat d'une règle, ce qui est très utile pour insérer des espaces blancs.

Analysons le fonctionnement du programme :

1. on démarre le programme avec comme nœud courant la racine XSLT :
   (a) le moteur choisi la règle adaptée à la racine : c'est ici la première
   (b) le résultat est obtenu grâce au modèle de résultat
2. comme le modèle de résultat contient un `apply-templates`, on relance le programme avec comme liste de nœuds courants les fils de la racine XSLT, c'est-à-dire dans notre exemple le nœud `liste` :
   (a) le moteur cherche une règle adaptée à `liste` et n'en trouve pas. Il utilise alors la règle par défaut suivante :

   ```
   <xsl:template match="/|*">
     <xsl:apply-templates/>
   </xsl:template>
   ```

   L'expression XPath utilisée (`/|*`) assure qu'elle peut être appliquée à tout élément d'un document XML, ainsi qu'au document lui-même (la racine).
   (b) le résultat est obtenu grâce au modèle de résultat
3. pour obtenir le résultat, on relance le programme avec comme liste de nœuds courants les fils de `liste`, c'est-à-dire les `personne`s :
   (a) le moteur choisi la règle adaptée : c'est la deuxième règle. En effet, pour chaque nœud de la liste (qui est alors le nœud courant), le moteur évalue l'expression XPath de chaque règle avec comme nœud contexte tour à tour tous les ancêtres du nœud courant. Pour un nœud `personne`, on considère le père, le nœud `liste`. Avec un tel nœud contexte, l'expression `personne` donne comme résultat la liste des éléments `personne` du document XML. Comme le nœud courant appartient à cette liste, on considère que le motif est validé et on choisit donc la règle.
   (b) le résultat est obtenu grâce au modèle de résultat : ici, on arrête la récursivité car il n'y pas de `apply-templates`. L'utilisation de `value-of` personne d'obtenir le contenu des éléments sélectionnés par les expressions XPath.

On remarque que le fichier résultat contient de nombreux espaces blancs et sauts de lignes. Pour comprendre ce phénomène, il faut noter que nous avons simplifié la description du traitement en oubliant le fait que les espaces blancs apparaissent sous forme de nœuds dans l'arbre XSLT. De ce fait, quand on traite les nœuds récursivement, on doit tenir compte des textes blancs. Or, il existe une deuxième règle par défaut en XSLT :

```
<xsl:template match="text()|attribute::*">
  <xsl:value-of select="."/>
</xsl:template>
```

L'expression XPath `text()` sélectionne tout les nœuds de texte fils du nœud contexte, alors que l'expression `attribute::*` correspond aux attributs du nœud contexte (la barre verticale est une union d'ensembles). Enfin, l'expression `.` correspond au nœud contexte lui-même. Pour éviter d'avoir des blancs intempestifs, plusieurs solutions sont envisageables :

- on peut remplacer la règle par défaut par une règle vide :
  `<xsl:template match="text()|attribute::*">`
  `</xsl:template>`
  Comme cette règle est vide, on peut la remplacer par :
  `<xsl:template match="text()|attribute::*"/>`
- on peut choisit explicitement la nouvelle liste de nœuds courants, en utilisant l'attribut `select` de `apply-templates`. Par exemple, on remplace la ligne 15 du programme XSLT par :
  `<xsl:apply-templates select="//personne"/>`
  L'expression `//personne` sélectionne tout les descendants (au sens large) du nœud contexte qui sont des éléments `personne` (on rappelle que `/` signifie fils direct). On saute ainsi la deuxième étape de la récursivité et on évite les nœuds de texte correspondant à des espaces blancs.
- on peut aussi utiliser l'instruction `strip-space` en insérant par exemple entre les lignes 6 et 7 :
  `<xsl:strip-space elements="*"/>`
  Cette instruction a pour effet d'enlever dans l'arbre XSLT tous les nœuds de texte correspondant à des espaces blancs. L'attribut `element` permet de sélectionner (par une expression XPath) les éléments dans lesquelles cette suppression doit avoir lieu (ici tous).

Dans tous les cas, on obtient le fichier HTML suivant :

liste2-nowhite.html

```
1
2  <!DOCTYPE html
3    PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
4  <html>
5     <head>
6        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
7
8        <title>Liste de personnes</title>
9     </head>
10    <body>
11       <h1>Liste de personnes</h1>
12       <ul>
13          <li>John Doe</li>
14          <li>Robert Durand</li>
15          <li>El&eacute;anore Dupuis</li>
16       </ul>
17    </body>
18 </html>
```

2.4 Exercices

Exercice 2.1

Ecrire un programme XSLT qui transforme un document au format de la DTD carnet d'adresse de l'exercice 2.1 de [5] en une page HTML.

Exercice 2.2

On reprend la DTD play étudiée dans l'exercice 2.3 de [6] :

1. Ecrire un programme XSLT qui produit le plan de la pièce sous forme d'un fichier HTML (en tenant compte des cas particuliers comme les prologues, etc.).
2. Ecrire un programme XSLT qui produit la liste des personnages de la pièce en respectant le regroupement opéré par l'élément `PGROUP` et sous la forme d'un fichier HTML.
3. Ecrire un programme XSLT qui transforme une pièce au format play en un fichier HTML (on conservera donc le maximum d'informations, avec une présentation correcte).

3 Mécanismes avancées

3.1 Références croisées

La technique classique utilisée pour éviter la redondance dans un fichier XML consiste à ajouter des attributs de type ID/IDREF(S), comme dans l'exemple suivant :

DisquesML.dtd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT disques (groupe*, disque*)  >
<!ELEMENT groupe (nom,membre+) >
<!ATTLIST groupe nom ID #REQUIRED>
<!ELEMENT nom (#PCDATA) >
<!ELEMENT membre (#PCDATA) >
<!ELEMENT disque (interprète, titre)>
<!ELEMENT interprète EMPTY >
<!ATTLIST interprète nom IDREF #REQUIRED>
<!ELEMENT titre (#PCDATA) >
```

On peut donc se demander comment profiter des références croisées dans un programme XSLT.

3.1.1 Référence directe

Si on souhaite retrouver l'élément associé à un ID donné, la solution est très simple. En effet, XPath propose une fonction `id` qui à une chaîne de caractères associe l'élément identifié par cette chaîne. Voici un exemple d'application où on fabrique une liste de disques en rappelant le nom du groupe interprète du disque :

disques1.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="ISO-8859-1"/>
  <xsl:strip-space elements="*"/>
  <xsl:template match="/">
    <liste>
        <xsl:apply-templates select="//disque"/>
    </liste>
  </xsl:template>
  <xsl:template match="disque">
    <disque><xsl:value-of select="titre"/><xsl:text> interprété par </xsl:text>
    <xsl:value-of select="id(interprète/@nom)/nom"/></disque>
  </xsl:template>
</xsl:stylesheet>
```

Appliqué à

disques.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE disques SYSTEM "DisquesML.dtd">
<disques>
<groupe nom="muse">
<nom>MUSE</nom>
<membre>Matthew Bellamy</membre>
<membre>Dominic Howard</membre>
<membre>Chris Wolstenholme</membre>
</groupe>
<groupe nom="feeder">
<nom>FEEDER</nom>
<membre>Grant Nicholas</membre>
<membre>Taka Hirose</membre>
<membre>Jon Henry Lee</membre>
</groupe>
<disque>
<interprète nom="muse"/>
<titre>Showbiz</titre>
</disque>
<disque>
<interprète nom="feeder"/>
<titre>Echo Park</titre>
</disque>
<disque>
<interprète nom="muse"/>
<titre>Origin of symmetry</titre>
</disque>
<disque>
<interprète nom="feeder"/>
<titre>Comfort In Sound</titre>
</disque>
</disques>
```

le programme produit le résultat suivant :

disques-out1.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<liste>
   <disque>Showbiz interprété par MUSE</disque>
   <disque>Echo Park interprété par FEEDER</disque>
   <disque>Origin of symmetry interprété par MUSE</disque>
   <disque>Comfort In Sound interprété par FEEDER</disque>
</liste>
```

Toute la subtilité du programme réside dans l'expression XPath `id(interprète/@nom)/nom`. Tout d'abord, l'utilisation de `@` indique qu'on s'intéresse à un attribut et donc `interprète/@nom` signifie les attributs de nom `nom` des éléments `interprète` du nœud contexte. La chaîne de caractères ainsi obtenue est passée à la fonction `id` qui renvoie l'élément designé par cette chaîne. La fin de l'expression sélectionne l'élément `nom` fils du résultat de l'appel à `id`.

Exercice 3.1

Compléter l'exemple proposé afin de produire une sortie en HTML dans laquelle la composition complète du groupe est rappelée pour chaque disque. On aura intérêt à écrire une règle de transformation spécifique à l'affichage de la composition d'un groupe.

3.1.2 Références arrières

La situation se complique quand on souhaite rechercher les références dans l'autre sens, c'est-à-dire trouver tous les IDREF(S) qui désignent un ID. Dans notre exemple, il s'agit de faire la liste des groupes et pour chaque groupe la liste des disques. Voici une solution qui introduit de nouvelles constructions intéressantes :

disques2.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="ISO-8859-1"/>
  <xsl:strip-space elements="*"/>
  <xsl:template match="/">
    <liste>
        <xsl:apply-templates select="//groupe"/>
    </liste>
  </xsl:template>
  <xsl:template match="groupe">
    <groupe>
      <xsl:copy-of select="./*"/>
      <disques>
        <xsl:variable name="id" select="@nom"/>
        <xsl:for-each select="//disque[interprète/@nom=$id]">
          <xsl:copy-of select="."/>
        </xsl:for-each>
      </disques>
    </groupe>
  </xsl:template>
</xsl:stylesheet>
```

Etudions les nouvelles instructions :

- l'instruction `copy-of` effectue comme son nom l'indique une copie conforme des nœuds sélectionnés par son attribut `select`. Dans sa première utilisation, l'expression XPath `./*` sélectionne tous les descendants du nœud contexte (qui est le nœud courant), c'est-à-dire la description du groupe (cf le fichier de résultat) ;
- l'instruction `variable` déclare une variable. Le concept de variable est très délicat en XSLT et est à interprété comme une variable déclarée `final` en `Java`. L'idée est qu'une variable locale (c'est-à-dire déclarée dans une règle comme ici) est en fait constante dans chaque application de la règle. En d'autres termes, à chaque fois qu'on applique la règle, la variable prend une valeur qu'il est impossible de changer pendant la construction du résultat de la règle. Par contre, dans une autre invocation de la règle, la valeur de la variable pourra être différente. Ici c'est le cas puisque la valeur de `id` est le contenu de l'attribut `nom` du nœud contexte ;
- l'instruction `for-each` ressemble à une boucle. En fait, son résultat est la concaténation des résultats obtenus en interprétant son contenu pour chacun des nœuds contenus dans la liste produite par l'expression XPath de son attribut `select`. On aurait pu utiliser un appel récursif ;
- l'expression XPath `//disque[interprète/@nom=$id]` est très intéressante :
  - la sous-expression entre les crochets est un prédicat. Le résultat de l'expression globale est en fait l'ensemble des éléments `disque` contenus dans le document XML (à cause de `//disque`) qui vérifient le prédicat ;
  - le prédicat lui-même consiste en la comparaison de l'attribut `nom` du fils `interprète` de l'élément `disque` considéré et du contenu de la variable `id` (obtenu par `$id`). L'utilisation de la variable est rendu obligatoire par la notion de nœud contexte. En effet, quand on évalue `//disque`, le nœud contexte est le groupe en cours d'étude et donc, tout va bien. Par contre, pour sélectionner les `disque`s retenus, on évalue le prédicat pour chaque élément `disque` **avec comme nœud contexte** précisément le `disque` en cours d'étude. Si on ne conserve pas la valeur recherchée dans une variable, il faut trouver un autre moyen d'obtenir l'ancien nœud contexte, à savoir le nœud courant de la règle de transformation. Ecrire `//disque[interprète/@nom=./@nom]` ne donne par un résultat satisfaisant (aucun `disque` n'est sélectionné car aucun ne porte d'attribut `nom` !). La seule solution est d'utiliser la

fonction XPath `current`, en écrivant `//disque[interprète/@nom=current()/@nom]`. Cette solution fonctionne parfaitement mais est moins générale que l'utilisation d'une variable car elle remplace seulement la mémorisation du noeud courant.

On obtient le résultat suivant :

disques-out2.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<liste>
   <groupe>
      <nom>MUSE</nom>
      <membre>Matthew Bellamy</membre>
      <membre>Dominic Howard</membre>
      <membre>Chris Wolstenholme</membre>
      <disques>
         <disque>
            <interprète nom="muse"/>
            <titre>Showbiz</titre>
         </disque>
         <disque>
            <interprète nom="muse"/>
            <titre>Origin of symmetry</titre>
         </disque>
      </disques>
   </groupe>
   <groupe>
      <nom>FEEDER</nom>
      <membre>Grant Nicholas</membre>
      <membre>Taka Hirose</membre>
      <membre>Jon Henry Lee</membre>
      <disques>
         <disque>
            <interprète nom="feeder"/>
            <titre>Echo Park</titre>
         </disque>
         <disque>
            <interprète nom="feeder"/>
            <titre>Comfort In Sound</titre>
         </disque>
      </disques>
   </groupe>
</liste>
```

Exercice 3.2

Modifier l'exemple proposé afin de produire une sortie en HTML.

Exercice 3.3

On reprend l'exercice 3.1 de [5], en particulier la première version de la DTD basée sur les ID/IDREFS. Ecrire un programme XSLT qui transforme un fichier de structure d'entreprise en une page HTML donnant pour chaque employé la liste de ses subordonnés (identifiés par leur nom et prénom) ainsi que son supérieur. On pourra utiliser l'instruction `if` dont le contenu n'est pris en compte que si l'expression XPath contenue dans son attribut `test` a pour valeur `true`. C'est en particulier le cas quand une expression produit une liste de nœuds non vide.

3.1.3 L'instruction `key`

L'utilisation de l'instruction `key` de XSLT permet d'aller encore plus loin, en particulier en autorisant l'équivalent de plusieurs espaces de noms pour les ID/IDREFS, en permettant de se passer de DTD, de placer les références dans un élément plutôt qu'un attribut, etc. Voici un exemple d'utilisation :

disques3.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xsl:stylesheet version="1.0"
3            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4    <xsl:output method="xml" indent="yes" encoding="ISO-8859-1"/>
5    <xsl:strip-space elements="*"/>
6    <xsl:key name="groupe2disque" match="disque" use="interprète/@nom"/>
7    <xsl:template match="/">
8      <liste>
9          <xsl:apply-templates select="//groupe"/>
10     </liste>
11   </xsl:template>
12   <xsl:template match="groupe">
13     <groupe>
14       <xsl:copy-of select="./*"/>
15       <disques>
16         <xsl:for-each select="key('groupe2disque',@nom)">
17           <xsl:copy-of select="."/>
18         </xsl:for-each>
19       </disques>
20     </groupe>
21   </xsl:template>
22 </xsl:stylesheet>
```

On remarque deux différences avec la version précédente du programme :

1. l'instruction `key` est utilisée ligne 6 : elle demande au programme de construire une table d'association nommée `groupe2disque` entre les éléments `disque`s et les chaînes de caractères précisée par l'expression XPath `interprète/@nom` interprétée avec comme noeud contexte le `disque` étudié. La grosse différence avec un ID XML est qu'une chaîne peut désigner **plusieurs** éléments ;
2. l'expression XPath de la ligne 16 utilise la fonction `key`. Cette fonction va chercher dans la table `groupe2disque` les noeuds associés à la chaîne `@nom`.

Cette nouvelle version du programme donne exactement les mêmes résultats que l'ancienne.

Exercice 3.4

Modifier la DTD proposée au début de la section pour supprimer les ID/IDREF et autoriser un contenu de type texte pour l'élément `interprète`. Ecrire un programme XSLT qui donne la liste des groupes avec pour chaque groupe la liste des disques en utilisant comme mécanisme de référence le contenu des éléments `interpètre` (côté disque) et celui des éléments `nom` (côté groupe).

3.2 Attributs

Jusque à présent, nous avons prudemment évité des situations dans lesquelles nous aurions eu à produire des attributs dans le fichier résultat. Supposons qu'on souhaite par exemple passer de la liste de personnes proposée à la section 2.2 à un format XML dans lequel le nom et le prénom du contact sont contenus dans des attributs. On propose le programme XSLT suivant :

perso2attrib.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xsl:stylesheet version="1.0"
3            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4    <xsl:output method="xml" indent="yes" encoding="ISO-8859-1"/>
5    <xsl:strip-space elements="*"/>
6    <xsl:template match="/">
7      <liste>
8        <xsl:apply-templates select="//personne"/>
9      </liste>
10   </xsl:template>
```

```
11    <xsl:template match="personne">
12      <personne prénom="{prénom}" nom="{nom}"/>
13    </xsl:template>
14  </xsl:stylesheet>
```

En appliquant ce programme, on obtient :

liste3.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <liste>
3     <personne prénom="John" nom="Doe"/>
4     <personne prénom="Robert" nom="Durand"/>
5     <personne prénom="Eléanore" nom="Dupuis"/>
6  </liste>
```

Toute l'astuce vient de l'utilisation des accolades. Quand le modèle de résultat d'une règle de transformation contient un élément dont un attribut contient des accolades, le texte délimité par ces accolades est interprété comme une expression XPath dont le résultat est traduit en texte.

Exercice 3.5

On reprend la DTD XBel simplifiée proposée dans l'exercice 3.2 de [4]. Ecrire un programme XSLT qui traduit un fichier XBEL en une page HTML contenant la liste des signets. Chaque signet sera lié à l'URL correspondante. De plus, on remplacera chaque `alias` par sa cible.

Références

[1] James Clark, editor. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation. W3C, 16 November 1999. `http://www.w3.org/TR/xslt`.

[2] James Clark and Steve DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C Recommendation. W3C, 16 November 1999. `http://www.w3.org/TR/xpath`.

[3] Philippe Drix. *XSLT fondamental*. Eyrolles, 2002.

[4] Fabrice Rossi. Document object model. Recueil d'exercices, Université Paris-IX Dauphine, 2002. `http://apiacoa.org/teaching/xml/exercices/dom.pdf`.

[5] Fabrice Rossi. Dtd et schémas. Recueil d'exercices, Université Paris-IX Dauphine, 2002. `http://apiacoa.org/teaching/xml/exercices/dtd-et-schemas.pdf`.

[6] Fabrice Rossi. Simple api for xml. Recueil d'exercices, Université Paris-IX Dauphine, 2002. `http://apiacoa.org/teaching/xml/exercices/sax.pdf`.