ROYAUME DU MAROC

مكتب التكوين المهني وإنعاش الشغل

Office de la Formation Professionnelle et de la Promotion du Travail

Développer des applications XML

OFPPT

DIRECTION RECHERCHE ET INGENIERIE DE FORMATION

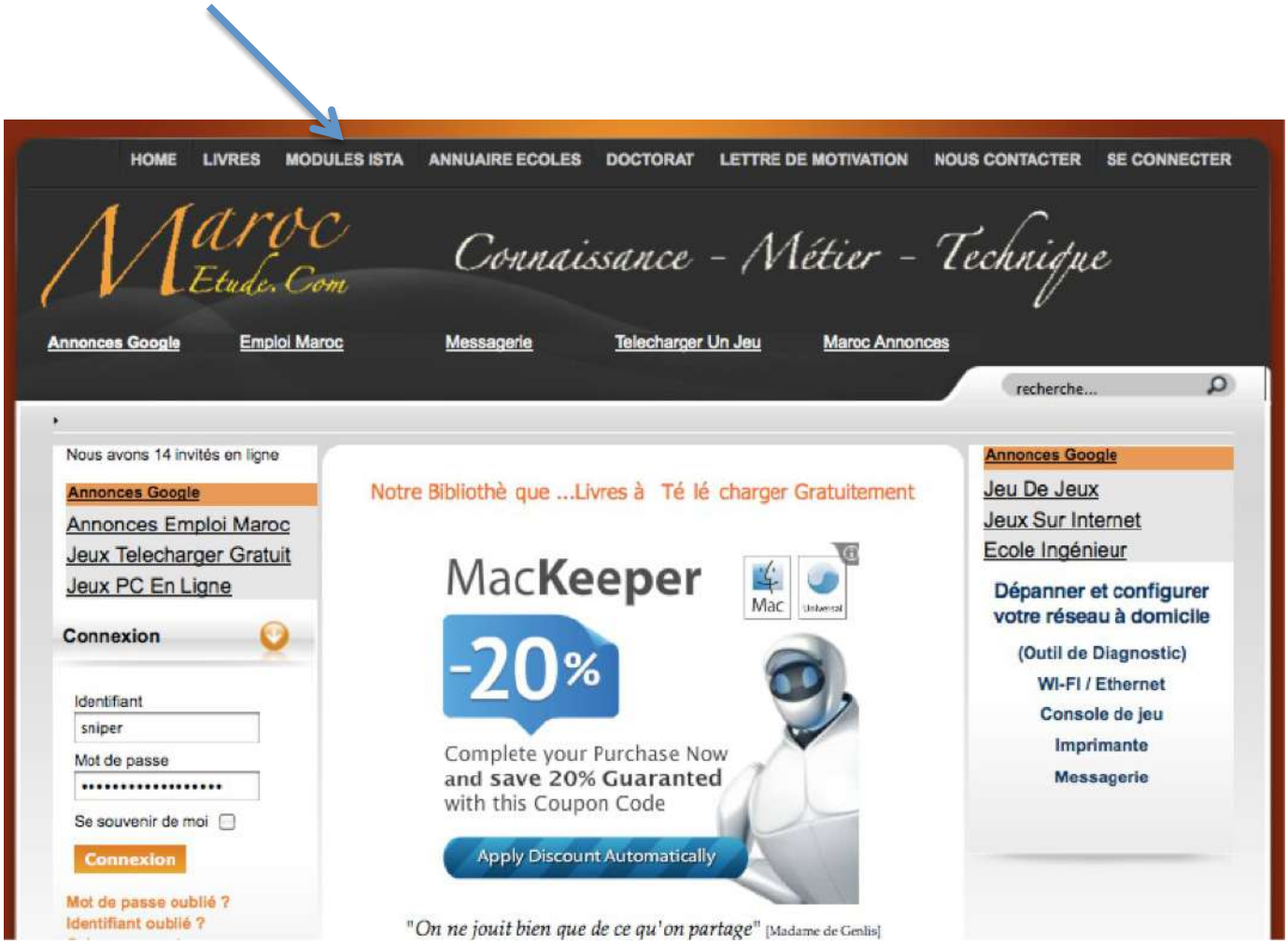
SECTEUR NTIC

PORTAIL DE LA FORMATION PROFESSIONNELLE AU MAROC

Télécharger tous les modules de toutes les filières de l'OFPPT sur le site dédié à la formation professionnelle au Maroc : **www.marocetude.com**

Pour cela visiter notre site www.marocetude.com et choisissez la rubrique :

MODULES ISTA



Sommaire

1. Introduction 4
2. Chapitre I : XML 4
2.1. Présentation de XML 4
2.2. Mise en page de XML 5
2.3. Les avantages de XML 5
3. Chapitre II : DTD 6
3.1. Introduction 6
3.2. Types de DTD 6
3.2.1. Introduction 6
3.2.2. Syntaxe 7
3.2.3. DTD externe 7
3.3. Déclarations d'éléments 8
3.3.1. Généralités 8
3.3.2. Élément texte 8
3.3.3. Élément vide 8
3.3.4. Indicateurs d'occurrence 9
3.3.5. Séquence d'éléments 9
1. Exemple d'utilisation valide 9
2. Exemples d'utilisations non valides : 10
3.3.6. Choix d'éléments 10
3.3.7. Élément quelconque 12
3.3.8. Élément à contenu mixte 12
3.4. Déclarations d'attributs 12
3.4.1. Introduction 12
3.4.2. Valeurs par défaut 13
3. Déclaration d'un attribut avec une valeur par défaut 13
4. Déclaration d'un attribut requis 13
5. Déclaration d'un attribut optionnel 13
3.4.3. Type chaîne de caractères 14
6. Déclaration d'un attribut avec une valeur par défaut 14
7. Déclaration d'un attribut requis 14
8. Déclaration d'un attribut avec une valeur fixe 14
3.4.4. Type chaîne de caractères 15
3.4.5. Type ID 15
3.4.6. Type énuméré 16
3.5. Déclarations d'entités 16
3.5.1. Introduction 16
3.5.2. Les entités paramétriques 16
3.5.3. Les entités de caractères 17
3.5.4. Les entités internes 17
3.5.5. Les entités externes 17
4. Chapitre III : XSD 19
4.1. Introduction 19
4.1.1. Limitations des DTD 19
4.1.2. Apports des schémas 19
4.2. Les premiers pas 20
4.2.1. Introduction 20
4.2.2. Structure de base 20

4.3. Déclarations d'éléments et d'attributs 21
4.3.1. Déclarations d'éléments 21
4.3.2. Déclarations d'attributs 21
4.3.2.1. Déclaration simple 21
4.3.2.2. Contraintes d'occurrences 22
4.3.2.3. Regroupements d'attributs 23
4.3.2.4. Déclaration d'élément ne contenant que du texte 23
4.3.3. Déclaration et référencement 24
4.4. Les types de données 24
4.4.1. Introduction 24
4.4.2. Types simples 24
4.4.2.3. Listes 25
4.4.2.4. Unions 26
4.4.3. Les types complexes 26
4.4.3.1. Introduction 26
4.4.3.2. Séquences d'éléments 26
4.4.3.3. Choix d'éléments 27
4.4.3.4. L'élément All 28
4.4.3.5. Indicateurs d'occurrences 29
4.4.3.6. Création de type complexe à partir de types simples 29
4.5. Espaces de noms 30
4.5.1. Introduction 30
4.5.2. Comment lier un fichier XML à un schéma ? 30
4.6. Les dérivations 31
4.6.1. Introduction 31
4.6.2. Restriction de type 31
4.6.2.1. Généralités 31
4.6.2.2. Exemples 32
4.7. Diverses autres fonctionnalités 33
4.7.1. Inclusion de schémas 33
4.7.2. Documentation 33
4.7.3. Attribut null 34
5. Chapitre IV : XSL 35
5.1. Présentation 35
5.1.1. Introduction 35
5.1.2. Structure d'un document XSL 35
5.2. Exemples de mises en forme 36
5.2.1. Exemple simple 36
5.2.1.1. Introduction 36
5.2.1.2. Exemple 36
5.2.2. Exemple avec boucle 37
5.2.2.1. Introduction 37
6. Chapitre V : XSLT 39
6.1. Les expressions de sélection 39
6.1.1. Introduction 39
6.1.2. Sélection d'éléments et d'attributs 39
6.1.2.1. Sélection d'élément, syntaxe de base 39
6.1.2.2. Sélection d'élément, appel de fonctions 40
6.1.2.3. Sélection d'élément et DOM 40
6.1.2.4. Sélection d'attributs 41
6.1.2.5. Opérateurs logiques 41
6.2. XPath 42

6.2.1. Introduction 42
6.2.2. Chemin de localisation 42
6.2.2.1. Introduction 42
6.2.2.2. Axes 43
6.2.2.3. Prédicats 44
6.2.2.4. Syntaxe non abrégée 44
6.2.2.5. Syntaxe abrégée 45
6.2.3. Fonctions de base 46
6.2.3.1. Généralités 46
6.2.3.2. Manipulation de nœuds 46
6.2.3.3. Manipulation de chaînes de caractères 46
6.2.3.4. Manipulation de booléens 47
6.2.3.5. Manipulation de nombres 47
6.2.4. Éléments XSLT 47
Généralités 47
Introduction 47
Rappel : prologue d'un document XSL 47
2. Les fondamentaux 48
Généralités 48
<xsl:stylesheet> 48
<xsl:output> 48
<xsl:template> 49
<xsl:value-of> 50
Ajout d'éléments et d'attributs 50
<xsl:element> 50
<xsl:attribute> 51
Syntaxe courte 51
Gestion des boucles 52
<xsl:for-each> 52
<xsl:sort> 52
<xsl:number> 53
Conditions de test 54
<xsl:if> 54
<xsl:choose> 54
Variables et paramètres 55
Introduction 55
Élément <xsl:variable> 55
L'élément <xsl:call-template> 56
Les éléments <xsl:param> et <xsl:with-param> 56

1. Introduction

Présenter les objets de formation.
Situer les apports de connaissances et établir le lien avec les objectifs visés par la compétence afin de susciter une motivation plus importante du stagiaire.

2. Chapitre I : XML

2.1. Présentation de XML

XML (entendez *eXtensible Markup Language* et traduisez *Langage à balises étendu*, ou *Langage à balises extensible*) est en quelque sorte un langage HTML amélioré permettant de définir de nouvelles balises. Il s'agit effectivement d'un langage permettant de mettre en forme des documents grâce à des balises (markup).

Contrairement à HTML, qui est à considérer comme un langage défini et figé (avec un nombre de balises limité), XML peut être considéré comme un métalangage permettant de définir d'autres langages, c'est-à-dire définir de nouvelles balises permettant de décrire la présentation d'un texte (Qui n'a jamais désiré une balise qui n'existait pas ?).
La force de XML réside dans sa capacité à pouvoir décrire n'importe quel domaine de données grâce à son extensibilité. Il va permettre de structurer, poser le vocabulaire et la syntaxe des données qu'il va contenir.

En réalité les balises XML décrivent le contenu plutôt que la présentation (contrairement À HTML). Ainsi, **XML permet de séparer le contenu de la présentation** .. ce qui permet par exemple d'afficher un même document sur des applications ou des périphériques différents sans pour autant nécessiter de créer autant de versions du document que l'on nécessite de représentations !

XML a été mis au point par le XML Working Group sous l'égide du World Wide Web Consortium (W3C) dès 1996. Depuis le 10 février 1998, les spécifications *XML 1.0* ont été reconnues comme recommandations par le W3C, ce qui en fait un langage reconnu. (Tous les documents liés à la norme XML sont consultables et téléchargeables sur le site web du W3C, http ://www.w3.org/XML/)

2.2. Mise en page de XML

XML est un format de description des données et non de leur représentation, comme c'est le cas avec HTML. La mise en page des données est assurée par un langage de mise en page tiers.

Il existe trois solutions pour mettre en forme un document XML :

- CSS (*Cascading StyleSheet*), la solution la plus utilisée actuellement, étant donné qu'il s'agit d'un standard qui a déjà fait ses preuves avec HTML
- XSL (*eXtensible StyleSheet Language*), un langage de feuilles de style extensible développé spécialement pour XML. Toutefois, ce nouveau langage n'est pas reconnu pour l'instant comme un standard officiel
- XSLT (*eXtensible StyleSheet Language Transformation*). Il s'agit d'une recommandation W3C du 16 novembre 1999, permettant de transformer un document XML en document HTML accompagné de feuilles de style

2.3. Les avantages de XML

Voici les principaux atouts de XML :

- La lisibilité : aucune connaissance ne doit théoriquement être nécessaire pour comprendre un contenu d'un document XML
- Autodescriptif et extensible
- Une structure arborescente : permettant de modéliser la majorité des problèmes informatiques
- Universalité et portabilité : les différents jeux de caractères sont pris en compte
- Déployable : il peut être facilement distribué par n'importe quels protocoles à même de transporter du texte, comme HTTP
- Intégrabilité : un document XML est utilisable par toute application pourvue d'un parser (c'est-à-dire un logiciel permettant d'analyser un code XML)
- Extensibilité : un document XML doit pouvoir être utilisable dans tous les domaines d'applications

Ainsi, XML est particulièrement adapté à l'échange de données et de documents.

L'intérêt de disposer d'un format commun d'échange d'information dépend du contexte professionnel dans lequel les utilisateurs interviennent. C'est pourquoi, de nombreux formats de données issus de XML apparaissent (il en existe plus d'une centaine) :

- OFX : Open Financial eXchange pour les échanges d'informations dans le monde financier
- MathML : Mathematical Markup Language permet de représenter des formules mathématique
- CML : Chemical Markup Language permet de décrire des composés chimiques

- SMIL : Synchronized Multimedia Integration Language permet de créer des présentations multimédia en synchronisant diverses sources : audio, vidéo, texte,…

3. Chapitre II : DTD

3.1. Introduction

Il peut être parfois nécessaire de préciser les balises et attributs auxquels on a droit lors de la rédaction d'un document XML, par exemple si l'on veut pouvoir partager le même type de document avec une communauté d'autres rédacteurs. Deux solutions sont possibles : les "Schémas XML" et les "Document Type Definition". Ces dernières sont les plus simples à manipuler et sont apparues en premier, alors que les Schémas n'étaient pas encore définis. Ce sont les raisons pour lesquelles nous allons nous limiter à elles pour le moment. Il ne faut pas oublier néanmoins qu'il existe une autre solution, plus complexe certes, mais aussi plus puissante. Elle permet notamment d'informer plus efficacement l'utilisateur sur les balises auxquelles il a droit, ou bien de spécifier de manière plus détaillée le formatage autorisé pour le contenu de la balise ou de l'attributToute déclaration de type de document peut être composée de déclarations d'éléments, de déclarations d'attributs, de déclarations d'entités, de déclarations de notations et de commentaires.

3.2. Types de DTD

3.2.1. Introduction

Une DTD peut être stockée dans deux endroits différents. Elle peut être incorporée au document XML (elle est alors dite *interne*), ou bien être un fichier à part (on parle alors de DTD *externe*). Cette dernière possibilité permet de la partager entre plusieurs documents XML. Il est possible de mêler DTD interne et externe. Il existe de surcroît deux types de DTD externes : privé ou public. Les DTD privées sont accessibles uniquement en local (sur la machine de développement), tandis que les publiques sont disponibles pour tout le monde, étant accessibles grâce à un URI (*Uniform Resource Identifier*). Une déclaration de type de document est de la forme :

```
<!DOCTYPE elt.racine ... "..." "...">
```

Cette déclaration se place juste après le prologue du document. L'élément racine du document XML rattaché à cette DTD est alors obligatoirement elt.racine. Par exemple...

La syntaxe DTD ne diffère pas entre une DTD interne et une externe.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE commande ... "..." "boncommande.dtd">
<commande>
<item>(...)</item>
<item>(...)</item>
<item>(...)</item>
</commande>
```

3.2.2. Syntaxe

Le contenu ne change pas suivant le type de DTD, mais les déclarations d'une DTD interne sont écrites à l'intérieur du prologue du document XML alors que celles d'une DTD externe sont stockées dans un fichier... externe

Exemple de déclarations pour une DTD interne :

```
<!DOCTYPE biblio[
<!ELEMENT biblio (livre)*>
<!ELEMENT livre (titre, auteur, nb_pages)>
  <!ATTLIST livre
    type (roman | nouvelles | poemes | théâtre) #IMPLIED
    lang CDATA "fr"
  >
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteur (#PCDATA)>
<!ELEMENT nb_pages (#PCDATA)>
]>
```

3.2.3. DTD externe

Les deux types de DTD externes sont les DTD de type *public* et les DTD de type *system*. Le mot-clef SYSTEM indique que le fichier spécifié se trouve sur l'ordinateur local et qu'il est disponible uniquement à titre privé. Par contre, le mot-clé PUBLIC indique une ressource disponible pour tous sur un serveur web distant.
Exemple de déclaration de DTD externe de type SYSTEM

```
<!DOCTYPE biblio SYSTEM "bibliographie.dtd">
```

Le fichier associé est le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT biblio (livre*)>
<!ELEMENT livre (titre, auteur, nb_pages)>
  <!ATTLIST livre
    type (roman | nouvelles | poemes | théâtre) #IMPLIED
    lang CDATA "fr"
  >
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteur (#PCDATA)>
<!ELEMENT nb_pages (#PCDATA)>
```

Exemple de déclaration de DTD externe de type PUBLIC :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Dans l'exemple précédent, la chaîne de caractères après le mot PUBLIC fait référence tout d'abord à l'identifiant de la DTD (ici **-**, qui signifie que la DTD n'a pas de numéro d'enregistrement officiel), au propriétaire de la DTT (ici le W3C), puis son nom, enfin sa langue.

3.3. Déclarations d'éléments

3.3.1. Généralités

Une déclaration d'éléments est de la forme :

```
<!ELEMENT nom type_element>
```

- **nom** est le nom de l'élément
- **type_element** est le type auquel il est associé.

Un élément peut être de type texte, vide (*EMPTY*), séquence ou choix d'élément. Dans ces deux derniers cas, on indique la liste des éléments enfants.

3.3.2. Élément texte

Cet élément est le plus répandu, puisque c'est celui qui contient... du texte. Il se déclare ainsi :

```
<!ELEMENT elt (#PCDATA)>
```

3.3.3. Élément vide

Un élément vide est, comme son nom l'indique, un élément qui n'a aucun contenu -que ce soit de type texte, ou bien un autre élément. Le mot-clef utilisé pour la déclaration de ce type d'élément est EMPTY :

```
<!ELEMENT elt EMPTY>
```

Exemple d'utilisation :

```
<elt />
```

Un élément vide peut fort bien posséder un ou plusieurs attributs. Par exemple

```
<img width="100" height="30" src="paysage.jpg" alt="Paysage de
neige" />
```

3.3.4. Indicateurs d'occurrence

Lors de la déclaration de séquence ou de choix d'éléments, à chaque élément enfant peut être attribuée une indication d'occurence (`?`, `+` ou `*`)
Exemples d'indicateur d'occurences

```
<!ELEMENT elt0 (elt1, elt2?, elt3+, elt*)>
```

1. **elt1** ne comprend aucune indication d'occurrence. Il doit donc apparaître une *seule et unique fois* dans l'élément elt0 ;
2. **elt2** a pour indication d'occurrence ?. Cela signifie que l'élément doit apparaître *au maximum* une fois (il peut ne pas apparaître du tout)
3. **elt3** a pour indication d'occurrence +. Cela signifie que l'élément doit apparaître *au moins* une fois
   - **elt4** a pour indication d'occurrence *. Cela signifie que l'élément doit apparaître autant de fois que l'auteur le désire.

3.3.5. Séquence d'éléments

Une séquence d'éléments est une liste ordonnée des éléments qui doivent apparaître en tant qu'éléments enfants de l'élément que l'on est en train de définir. Ce dernier ne pourra contenir *aucun* autre élément que ceux figurant dans la séquence. Cette liste est composée d'éléments séparés par des virgules et est placée entre parenthèses Chaque élément enfant doit de plus être déclaré par ailleurs dans la DTD (avant ou après la définition de la liste, peu importe). Dans le fichier XML, ils doivent apparaître *dans l'ordre* de la séquence

```
<!ELEMENT elt0 (elt1, elt2, elt3)>
```

1. Exemple d'utilisation valide

```
<elt0>
  <elt1>(...)</elt1>
  <elt2>(...)</elt2>
  <elt3>(...)</elt3>
</elt0>
```

2. Exemples d'utilisations non valides :

Exemple 1 :

```
<elt0>
  <elt1>(...)</elt1>
  <elt3>(...)</elt3>
</elt0>
```

Car l'élément elt2 est manquant.

Exemple 2 :

```
<elt0>
  <elt1>(...)</elt1>
  <elt3>(...)</elt3>
  <elt2>(...)</elt2>
</elt0>
```

Car l'ordre des éléments n'est pas respecté.

On rappelle qu'il est possible d'utiliser des indicateurs d'occurrence.
Par exemple

```
<!ELEMENT elt0 (elt1+, elt2*, elt3?)>
```

3.3.6. Choix d'éléments

Un choix d'élément donne... le choix dans une liste de plusieurs éléments possibles. L'utilisation précise dépend des indicateurs d'occurrence. De même que pour la séquence, les éléments enfants doivent être déclarés dans la DTD. Cette liste est composée d'éléments séparés par le caractère | (combinaison de touches AltGr+6 sur un clavier AZERTY)

```
<!ELEMENT elt0 (elt1 | elt2 | elt3)>
```

Exemple d'utilisation valide :

```
<elt0><elt2>(...)</elt2></elt0>
```

Exemple d'utilisation non valide

```
<elt0>
  <elt2>(...)</elt2>
  <elt3>(...)</elt3>
</elt0>
```

Exemple d'utilisation d'un choix d'éléments avec indicateurs d'occurence par élément

```
<!ELEMENT choix.elt (elt1* | elt2* | elt3*)>
```

Exemple d'utilisation valide :

```
<elt0>
  <elt2>(...)</elt2>
  <elt2>(...)</elt2>
</elt0>
```

Exemples d'utilisation non valide :

```
<elt0>
  <elt3>(...)</elt3>
  <elt2>(...)</elt2>
</elt0>

<elt0>
  <elt2>(...)</elt2>
  <elt3>(...)</elt3>
</elt0>
```

Exemple d'utilisation d'un choix d'éléments avec indicateur d'occurrence global :

```
<!ELEMENT elt0 (elt1 | elt2 | elt3)*>
```

Exemple d'utilisation valide :

```
<elt0>
  <elt2>(...)</elt2>
  <elt3>(...)</elt3>
  <elt1>(...)</elt1>
</elt0>
```

Dans ce dernier cas, il n'y a pas de contrainte visible sur l'ordre d'apparition des éléments

3.3.7. Élément quelconque

L'élément quelconque est l'élément-"fourre-tout" dans une DTD. Il peut contenir tout autre élément défini dans la DTD, aussi bien qu'être vide ou contenir du texte. Les éléments-enfants éventuels peuvent apparaître dans n'importe quel ordre, et en quantité non définie. Il est préférable de ne pas utiliser trop souvent ce type de déclaration, car on perd les avantages qu'offre la rédaction d'une DTD, qui sont de fixer des contraintes précises sur la structure du document XML qui lui est lié. Le mot-clef utilisé pour la déclaration de ce type d'élément est ANY.

```
<!ELEMENT elt ANY>
```

3.3.8. Élément à contenu mixte

Un élément à contenu mixte peut contenir aussi bien du texte, que des éléments-enfants. Il se présente comme une liste de choix, avec des indicateurs d'occurrence bien choisis. Le texte contenu peut se trouver à n'importe quel endroit dans l'élément, et peut être une section CDATA

Exemple de déclaration :

```
<!ELEMENT citation (#PCDATA | auteur)*>
```

Exemple d'utilisation

```
<citation>
Être ou ne pas être   <auteur>Shakespeare</auteur>
</citation>
```

3.4. Déclarations d'attributs

3.4.1. Introduction

Comme on peut trouver dans un document XML des éléments possédant des attributs, il est normal que la DTD permette de définir des contraintes sur ces derniers. On peut déclarer et attacher à un élément donné chaque attribut séparément, mais il est préférable de les assembler sous la forme d'une liste. Chaque attribut défini dans la liste possède un nom et un type.

On peut lui donner une valeur par défaut, ou bien le spécifier obligatoire. Le mot-clef de cette déclaration est ATTLIST.

3.4.2. Valeurs par défaut

Chaque attribut peut être requis, optionnel ou fixe et avoir une valeur par défaut. Les exemples suivants montrent la déclaration d'un attribut appelé attr attaché à un élément nommé elt

3. Déclaration d'un attribut avec une valeur par défaut

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA "valeur">
```

Un tel attribut n'est pas obligatoire. S'il est omis dans le fichier XML lors de l'utilisation de l'élément elt, il est considéré comme valant valeur. Dans cet exemple, si on écrit <elt>(...)</elt>, cela est équivalent à écrire <elt attr="valeur">(...)>/elt>

4. Déclaration d'un attribut requis

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA #REQUIRED>
```

Un tel attribut est obligatoire. Son absence déclenche une erreur du vérificateur syntaxique sur le fichier XML.

5. Déclaration d'un attribut optionnel

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA #IMPLIED>
```

3.4.2.1. Déclaration d'un attribut avec une valeur fixe

L'utilité d'un tel attribut peut sembler bizarre à première vue, puisqu'il ne peut prendre qu'une seule valeur. Cette fonctionnalité est cependant utile lors d'une mise à jour d'une DTD, pour préserver la compatibilité avec des versions ultérieures

3.4.3. Type chaîne de caractères

Chaque attribut peut être requis, optionnel ou fixe et avoir une valeur par défaut. Les exemples suivants montrent la déclaration d'un attribut appelé attr attaché à un élément nommé elt

6. Déclaration d'un attribut avec une valeur par défaut

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA "valeur">
```

Un tel attribut n'est pas obligatoire. S'il est omis dans le fichier XML lors de l'utilisation de l'élément elt, il est considéré comme valant valeur. Dans cet exemple, si on écrit <elt>(...)</elt>, cela est équivalent à écrire <elt attr="valeur">(...)>/elt>

7. Déclaration d'un attribut requis

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA #REQUIRED>
```

Un tel attribut est obligatoire. Son absence déclenche une erreur du vérificateur syntaxique sur le fichier XML

3.4.3.1. Déclaration d'un attribut optionnel

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA #IMPLIED>
```

8. Déclaration d'un attribut avec une valeur fixe

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA #FIXED "valeur">
```

L'utilité d'un tel attribut peut sembler bizarre à première vue, puisqu'il ne peut prendre qu'une seule valeur. Cette fonctionnalité est cependant utile lors d'une mise à jour d'une DTD, pour préserver la compatibilité avec des versions ultérieures

3.4.4. Type chaîne de caractères

Il s'agit là du type d'attribut le plus courant. Une chaîne de caractères peut être composée de caractères ainsi que d'entités analysables. Le mot-clef utilisé pour la déclaration de chaîne de caractère est **CDATA**

Exemple de déclaration de **CDATA**

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr CDATA #IMPLIED>
```

Exemples d'utilisations :

```
1.    <elt attr="Chaîne de caractères"></elt>
2.    <!ENTITY car "caractères">
<elt attr="Chaîne de &car;">(...)</elt>
```

3.4.5. Type ID

Ce type sert à indiquer que l'attribut en question peut servir d'*identifiant* dans le fichier XML. Deux éléments ne pourront pas posséder le même atribut possédant la même valeur

Exemple de déclaration de type ID optionnel

```
<!ELEMENT elt (...)>
  <!ATTLIST elt attr ID #IMPLIED>
<!ELEMENT elt1 (...)>
  <!ATTLIST elt1 attr ID #IMPLIED>
<!ELEMENT elt2 (...)>
  <!ATTLIST elt2 attr ID #IMPLIED>
```

La déclaration précédente interdit par exemple

```
<elt1 attr="machin"></elt1>
<elt2 attr="truc"></elt2>
<elt1 attr="machin"></elt1>
```

ainsi que

```
<elt1 attr="machin"></elt1>
<elt2 attr="machin"></elt2>
<elt1 attr="truc"></elt1>
```

3.4.6. Type énuméré

On peut parfois désirer limiter la liste de valeurs possibles pour un attribut. On le définit alors comme étant de type énuméré. Donner une autre valeur dans le fichier XML provoque une erreur.

Exemple de déclaration d'une liste de choix d'attributs

```
<!ELEMENT img EMPTY>
  <!ATTLIST img format (BMP | GIF | JPEG) "JPEG">
```

Cet exemple déclare un attribut format d'un élément img. La valeur de cet attribut peut être BMP, GIF ou JPEG. Si aucune valeur n'est affectée à cet attribut, c'est la valeur par défaut qui le sera, ici JPEG. On notera l'absence de guillemets dans la liste des valeurs possibles. C'est une source courante d'erreur dans la rédaction d'une DTD

3.5. Déclarations d'entités

3.5.1. Introduction

Les déclarations d'entités permettent de disposer de l'équivalent de raccourcis clavier et de caractères *a priori* non accessibles dans le jeu de caractères sélectionné

3.5.2. Les entités paramétriques

Elles servent à définir des symboles qui seront utilisés ailleurs dans la DTD. Ce sont en quelque sorte des raccourcis d'écriture : partout où une entité est mentionnée, elle peut être remplacée par la chaîne de caractères qui lui est associée. Ce mécanisme s'apparente à un mécanisme de "macro"Les entités paramétriques ne peuvent pas être utilisées en-dehors d'une DTD

Exemple tiré de la spécification du langage HTML

```
<!ENTITY % heading "H1|H2|H3|H4|H5|H6">
```

L'exemple précédent a pour effet d'indiquer au système que toute occurrence de % heading; doit être remplacée par H1|H2|H3|H4|H5|H6
Ce mécanisme peut également servir à utiliser un nom relativement compréhensible à la place d'une séquence de caractères peu évocatrice.

La définition d'une entité peut également faire référence à d'autres entités ; la substitution est alors effectuée de proche en proche

3.5.3. Les entités de caractères

Elles servent à donner un nom facilement lisible à des caractères qui ne sont pas représentables dans l'alphabet utilisé, ou qui ne sont pas disponibles au clavier.

Exemples tirés de la DTD du langage HTML 4.01

```
<!ENTITY nbsp " ">
<!ENTITY eacute "&#233;">
```

Les entités de caractères définies dans une DTD peuvent être utilisées dans un document XML référençant cette DTD à l'aide de la notation &NomEntité;.

3.5.4. Les entités internes

Ce sont des symboles pouvant être définis dans une DTD et utilisés dans un document XML comme raccourcis d'écriture. La définition complète du symbole est entièrement incluse dans la DTD.

Exemple

```
<!ENTITY ADN "Acide désoxyribonucléique">
```

Dans le fichier XML, l'appel à &ADN; sera aumatiquement remplacé, lors de l'affichage ou du traitement, par la chaîne de caractères "Acide désoxyribonucléique".

3.5.5. Les entités externes

Il s'agit soit de symboles pouvant être définis dans un autre fichier, mais pouvant être utilisés dans un document XML ou la DTD elle-même.

Par exemple :

```
<!ENTITY Inclusion SYSTEM "toto.xml">
<!ENTITY % Inclusion SYSTEM "toto.inc">
```

Dans le fichier XML, le contenu du fichier toto.xml sera inséré à l'appel de l'entité &Inclusion;, et dans la DTD, le contenu du fichier toto.inc sera inséré à l'appel de l'entité &Inclusion ;
... soit de symboles pouvant être définis dans une autre DTD et utilisés dans la DTD courante :

```
<!ENTITY % HTMLSpecial PUBLIC "-//W3C//ENTITIES Special for
XHTML//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent">
```

Le contenu de cette DTD (qui peut être de type SPECIAL ou PUBLIC) est importé dans la DTD courante par l'appel de %HTMLSpecial;.

4. Chapitre III : XSD

4.1. Introduction

4.1.1. Limitations des DTD

Lors de son lancement, XML a été perçu comme une réelle chance pour les développeurs, d'avoir à disposition un langage simple d'utilisation, portable sans difficulté d'une machine -et d'une application- à une autre, et libre de droits. Dans les premiers temps, un fichier XML, si on voulait le standardiser en utilisant un vrai langage général de description, devait dépendre d'une DTD. Mais ce format de description, hérité de SGML, souffre de nombreuses déficiences.

1. Premièrement, les DTD ne sont pas au format XML. Cela signifie qu'il est nécessaire d'utiliser un outil spécial pour "*parser*" un tel fichier, différent de celui utilisé pour l'édition du fichier XML
2. Deuxièmement, les DTD ne supportent pas les "espaces de nom" (nous reviendrons sur cette notion plus loin). En pratique, cela implique qu'il n'est pas possible, dans un fichier XML défini par une DTD, d'importer des définitions de balises définies par ailleurs
3. Troisièmement, le "typage" des données est extrêmement limité.

4.1.2. Apports des schémas

Conçu pour pallier aux déficiences pré-citées des DTD, XML Schéma propose, en plus des fonctionnalités fournies par les DTD, des nouveautés :

Le typage des données est introduit, ce qui permet la gestion de booléens, d'entiers, d'intervalles de temps... Il est même possible de créer de nouveaux types à partir de types existants.
La notion d'héritage. Les éléments peuvent hériter du contenu et des attributs d'un autre élément. C'est sans aucun doute l'innovation la plus intéressante de XML Schéma.
Le support des espaces de nom
Les indicateurs d'occurrences des éléments peuvent être tout nombre non négatif (rappel : dans une DTD, on était limité à 0, 1 ou un nombre infini d'occurrences pour un élément).
Les schémas sont très facilement concevables par modules.

4.2. Les premiers pas

4.2.1. Introduction

Le but d'un schéma est de définir une classe de documents XML. Il permet de décrire les autorisations d'imbrication et l'ordre d'apparition des éléments et de leurs attributs, tout comme une DTD. Mais il permet aussi d'aller au-delà
Un premier point intéressant est qu'un fichier Schéma XML est un document XML. Cela permet à un tel document d'être manipulé de la même manière que n'importe quel autre fichier XML, et en particulier par une feuille de style XSL. Il est notamment possible d'automatiser, par exemple, la création d'une documentation à partir d'un schéma, fondé sur les commentaires et explications qui s'y trouvent. C'est d'ailleurs chose facile avec l'éditeur oXygen. Il est facile, via le menu Modules d'extension>Schema Documentation, à partir d'un exemple de schéma, de produire la documentation correspondante
Le vocabulaire de XML Schéma est composé d'environ 30 éléments et attributs. Ce vocabulaire est, de manière bizarrement récursive et "auto-référente", défini dans un Schéma. Mais il existe également une DTD

4.2.2. Structure de base

Comme tout document XML, un Schema XML commence par un prologue, et a un élément racine.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <!-- déclarations d'éléments, d'attributs et de types ici -->
</xsd:schema>
```

L'élément racine est l'élément xsd:schema. Pour le moment, oubliez l'attribut xmlns:xsd (dont le rôle est le même que celui que nous avons déjà rencontré lors du cours sur les feuilles de style), et qui fait référence à l'"espace de noms" utilisé pour l'écriture du fichier. Il faut simplement retenir que tout élément d'un schéma doit commencer par le préfixe xsd.
Nous allons voir, par la suite, comment déclarer éléments et attributs à l'aide d'un schéma.

4.3. Déclarations d'éléments et d'attributs

4.3.1. Déclarations d'éléments

Un élément, dans un schéma, se déclare avec la balise <xsd:element>. Par exemple

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
<xsd:element name="contacts" type="typeContacts"></xsd:element>
<xsd:element name="remarque" type="xsd:string"></xsd:element>
<!-- déclarations de types ici -->
</xsd:schema>
```

Le schéma précédent déclare deux éléments : un élément contacts et un élément remarque. Chaque élément est "typé" -c'est-à-dire qu'il doit respecter un certain format de données. L'élément contacts est ainsi du type typeContacts, qui est un type complexe défini par l'utilisateur. L'élément remarque quant à lui est du type xsd:string qui est un type simple prédéfini de XML Schema.
Chaque élément déclaré est associé à un type de données via l'attribut type. Les éléments pouvant contenir des élément-enfants ou posséder des attributs sont dits de type *complexe*, tandis que les éléments n'en contenant pas sont dits de type *simple*. Nous reviendrons plus loin sur cette notion de type de données.

4.3.2. Déclarations d'attributs

4.3.2.1. Déclaration simple

A la différence des éléments, un attribut ne peut être que de type simple. Cela signifie que les attributs, comme avec les DTD, ne peuvent contenir d'autres éléments ou attributs. De plus, les déclarations d'attributs doivent être placées *après* les définitions des types complexes, autrement dit, après les éléments <xsd:sequence>, <xsd:choice> et <xsd:all>. Pour mémoire, rappelons que dans une DTD, l'ordre des déclarations n'a pas d'importance.
L'exemple suivant montre la déclaration d'un attribut maj de type xsd:date (un autre type simple) qui indique la date de dernière mise à jour de la liste des contacts.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
 <xsd:element name="contacts" type="typeContacts"></xsd:element>
 <xsd:element name="remarque" type="xsd:string"></xsd:element>
 <!-- déclarations de types ici -->
 <xsd:complexType name="typeContacts">
  <!-- déclarations du modèle de contenu ici -->
  <xsd:attribute name="maj" type="xsd:date" />
 </xsd:complexType>
</xsd:schema>
```

4.3.2.2. Contraintes d'occurrences

Tout comme dans une DTD, un attribut peut avoir un indicateur d'occurences.
L'élément attribute d'un Schema XML peut avoir trois attributs optionnels : use, default et fixed. Des combinaisons de ces trois attributs permettent de paramétrer ce qui est acceptable ou non dans le fichier XML final (attribut obligatoire, optionnel, possédant une valeur par défaut...). Par exemple, la ligne suivante permet de rendre l'attribut maj optionnel, avec une valeur par défaut au 11 octobre 2003 s'il n'apparaît pas (le format de date est standardisé : cette date s'écrit donc à l'anglo-saxonne 2003/10/11 ; cela permet en outre de plus facilement classer les dates)

```
<xsd:attribute name="maj" type="xsd:date" use="optional"
default="2003-10-11" />
```

Quand l'attribut fixed est renseigné, la seule valeur que peut prendre l'attribut déclaré est celle de l'attribut fixed. Cet attribut permet de "réserver" des noms d'attributs pour une utilisation future, dans le cadre d'une mise à jour du schéma.
Le tableau suivant présente une comparaison entre le format DTD et le XML Schema

DTD	Attribut `use`	Attribut `default`	Commentaire
#REQUIRED	Required	-	
"blabla" #REQUIRED	Required	blabla	
#IMPLIED	optional	-	
"blabla" #IMPLIED	optional	blabla	
-	Prohibited	-	Cet attribut ne doit pas apparaître

Table 1. Contraintes d'occurrences fixables par les attributs use et default

Il est à noter que la valeur de l'attribut default doit être conforme au type déclaré.

Par exemple :

```
<xsd:attribute name="maj" type="xsd:date" use="optional" default="-43" />
```

Produirait une erreur à la validation du schéma

Un autre type de déclaration d'attributs dans les DTD, la liste de choix, est possible grâce à une restriction de type

4.3.2.3. Regroupements d'attributs

XML Schema propose une fonctionnalité supplémentaire, permettant de déclarer des groupes d'attributs (groupes auxquels il est possible de faire appel lors d'une déclaration d'éléments). Cela permet d'éviter de répéter des informations de déclarations.

4.3.2.4. Déclaration d'élément ne contenant que du texte

Un tel élément est de type complexe, car il contient au moins un attribut. Afin de spécifier qu'il peut contenir également du texte, on utilise l'attribut mixed de l'élément <xsd;complexType>. Par défaut, mixed="false"; il faut dans ce cas forcer mixed="true".

Par exemple :

```
<xsd:element name="elt">
  <xsd:complexType mixed="true">
    <xsd:attribute name="attr" type="xsd:string" use="optional" />
  </xsd:complexType>
</xsd:element>
```

4.3.3. Déclaration et référencement

La procédure précédente de déclaration d'éléments peut amener à une structure de type "poupée russe" des déclarations. Il est beaucoup plus avantageux, pour des raisons de clarté, d'ordonner ces déclarations, ainsi qu'on peut le voir sur cet exemple (on ne fera pas attention, pour le moment, aux "définitions de type")
Il est recommandé de commencer par déclarer les éléments et attributs de type simple, puis ceux de type complexe. On peut en effet faire "référence", dans une déclaration de type complexe, à un élément de type simple préalablement défini. Par exemple

```
<xsd:element name="pages"
type="xsd:positiveInteger"></xsd:element>
<xsd:element name="auteur" type="xsd:string"></xsd:element>
<xsd:element name="livre">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="auteur" />
   <xsd:element ref="pages" />
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

4.4. Les types de données

4.4.1. Introduction

Ainsi que nous l'avons déjà brièvement signalé, XML Schema permet de spécifier des types de données bien plus finement que le langage DTD. Il distingue notamment types simples et types complexes

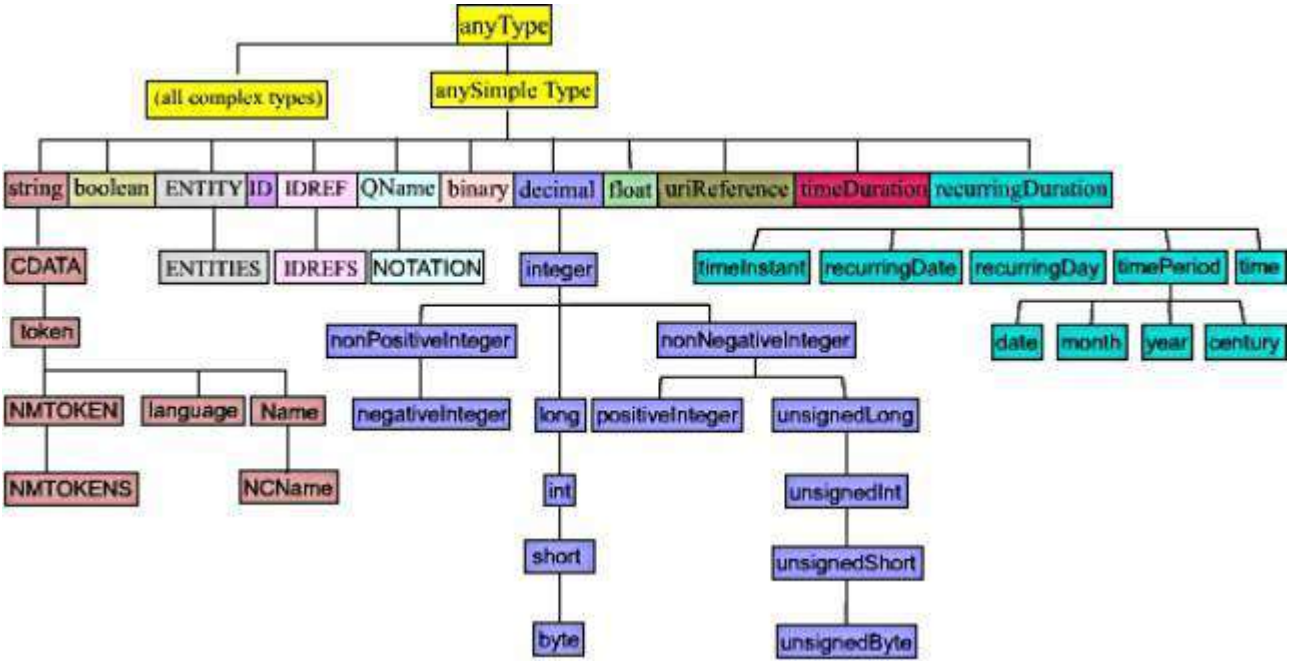
4.4.2. Types simples

4.4.2.1. Généralités

Les types de données simples ne peuvent comporter ni attributs, ni éléments enfants. Il en existe de nombreux, prédéfinis, mais il est également possible d'en "dériver" de nouveaux (nous reviendrons plus loin sur cette fonctionnalité). Enfin, il est possible de déclarer des "listes" de types

4.4.2.2. Bibliothèque de types intègres

Nombreux sont les types prédéfinis dans la bibliothèque de types intégrés de XML Schema. la figure suivante en donne la hiérarchie, et leur liste détaillée figure sur le site du W3C



Les types de données les plus simples (les chaînes de caractères) que permettaient les DTDs sont conservés, mais d'autres ont fait leur apparition. On pourra envisager, par exemple, dans un schéma décrivant un bon de commande, la déclaration d'un attribut quantite.

```
<xsd:attribute name="quantite" type="xsd:positiveInteger"
use="default" value="1" />
```

Qui force la valeur de l'attribut à un être un entier positif. Un bon de commande XML suivant ce schéma, ayant une commande spécifiant quantite="-3" sera alors automatiquement refusé par le système. Un tel document peut dès lors être enregistré directement dans une base de données, sans contrôle supplémentaire sur ce point

4.4.2.3. Listes

Les types listes sont des suites de types simples (ou *atomiques*). XML Schema possède trois types de listes intégrés : NMTOKENS, ENTITIES et IDREFS. Il est également possible de créer une liste personnalisée, par "dérivation" de types existants.
Par exemple :

```
<xsd:simpleType name="numéroDeTéléphone">
  <xsd:list itemType="xsd:unsignedByte" />
</xsd:simpleType>
```

Un élément conforme à cette déclaration serait <téléphone>01 44 27 60 11</téléphone>.

Il est également possible d'indiquer des contraintes plus fortes sur les types simples ; ces contraintes s'appellent des "facettes". Elles permettent par exemple de limiter la longueur de notre numéro de téléphone à 10 nombres.

4.4.2.4. Unions

Les listes et les types simples intégrés ne permettent pas de choisir le type de contenu d'un élément. On peut désirer, par exemple, qu'un type autorise soit un nombre, soit une chaîne de caractères particuliers. Il est possible de le faire à l'aide d'une déclaration d'union. Par exemple, sous réserve que le type simple numéroDeTéléphone ait été préalablement défini (voir précédemment), on peut déclarer

```
<xsd:simpleType name="numéroDeTéléphoneMnémoTechnique">
  <xsd:union memberTypes="xsd:string numéroDeTéléphone" />
</xsd:simpleType>
```

Les éléments suivants sont alors des "instances" valides de cette déclaration :

```
<téléphone>18</téléphone>
<téléphone>Pompiers</téléphone>
```

4.4.3. Les types complexes

4.4.3.1. Introduction

Un élément de type simple ne peut contenir de sous-élément. Il est nécessaire pour cela de le déclarer de type "complexe". On peut alors déclarer, des séquences d'éléments, des types de choix ou des contraintes d'occurrences

4.4.3.2. Séquences d'éléments

Nous savons déjà comment, dans une DTD, nous pouvons déclarer un élément comme pouvant contenir une suite de sous-éléments, dans un ordre déterminé. Il est bien sûr possible de faire de même avec un schéma.

On utilise pour ce faire l'élément xsd:sequence, qui reproduit l'opérateur, du langage DTD. Ainsi...

```
<xsd:complexType>
 <xsd:sequence>
  <xsd:element name="nom" type="xsd:string" />
  <xsd:element name="prénom" type="xsd:string" />
  <xsd:element name="dateDeNaissance" type="xsd:date" />
  <xsd:element name="adresse" type="xsd:string" />
  <xsd:element name="adresseElectronique" type="xsd:string" />
  <xsd:element name="téléphone" type="numéroDeTéléphone" />
 </xsd:sequence>
</xsd:complexType>
```

Est équivalent à une déclaration d'élément, dans une DTD, où apparaîtrait (nom, prénom, dateDeNaissance, adresse, adresseElectronique, téléphone)

4.4.3.3. Choix d'éléments

On peut vouloir modifier la déclaration de type précédente en stipulant qu'on doive indiquer soit l'adresse d'une personne, soit son adresse électronique. Pour cela, il suffit d'utiliser un élément xsd:choice

```
<xsd:complexType name="typePersonne">
 <sequence>
  <xsd:element name="nom" type="xsd:string" />
  <xsd:element name="prénom" type="xsd:string" />
  <xsd:element name="dateDeNaissance" type="xsd:date" />
  <xsd:choice>
   <xsd:element name="adresse" type="xsd:string" />
   <xsd:element name="adresseElectronique" type="xsd:string" />
  </xsd:choice>
 </sequence>
 <xsd:element name="téléphone" type="numéroDeTéléphone" />
</xsd:complexType>
```

Ce connecteur a donc les mêmes effets que l'opérateur | dans une DTD.

4.4.3.4. L'élément All

Cet élément est une nouveauté par rapport aux DTD. Il indique que les éléments enfants doivent apparaître une fois (ou pas du tout), et dans n'importe quel ordre. Cet élément xsd:all doit être un enfant direct de l'élément xsd:complexType.

Par exemple

```
<xsd:complexType>
 <xsd:all>
  <xsd:element name="nom" type="xsd:string" />
  <xsd:element name="prénom" type="xsd:string" />
  <xsd:element name="dateDeNaissance" type="xsd:date" />
  <xsd:element name="adresse" type="xsd:string" />
  <xsd:element name="adresseElectronique" type="xsd:string" />
  <xsd:element name="téléphone" type="numéroDeTéléphone" />
 </xsd:all>
</xsd:complexType>
```

Indique que chacun de ces éléments peut apparaître une fois ou pas du tout (équivalent de l'opérateur ? dans une DTD), et que l'ordre des éléments n'a pas d'importance (cela n'a pas d'équivalent dans une DTD).
Les fils de l'élément xsd:all doivent impérativement apparaître au plus une fois, ce qui signifie que deux attributs, que nous allons voir maintenant, doivent être renseignés. Ces attributs sont minoccurs et maxoccurs.

4.4.3.5. Indicateurs d'occurrences

Dans une DTD, un indicateur d'occurrence ne peut prendre que les valeurs 0, 1 ou l'infini. On peut forcer un élément sselt à être présent 378 fois, mais il faut pour cela écrire (sselt, sselt..., sselt, sselt) 378 fois. XML Schema permet de déclarer directement une telle occurence, car tout *nombre entier non négatif* peut être utilisé. Pour déclarer qu'un élément peut être présent un nombre illimité de fois, on utilise la valeur unbounded. Les attributs utiles sont minOccurs et maxOccurs, qui indiquent respectivement les nombres minimal et maximal de fois où un élément peut apparaître. Le tableau suivant récapitule les possibilités :

Dans une DTD	**Valeur de `minOccurs`**	**Valeur de `maxOccurs`**
`*`	0	`unbounded`
`+`	1 (pas nécessaire, valeur par défaut)	`unbounded`
`?`	0	1 (pas nécessaire, valeur par défaut)
rien	1 (pas nécessaire, valeur par défaut)	1 (pas nécessaire, valeur par défaut)

Table 2. Liste des indicateurs d'occurrence

4.4.3.6. Création de type complexe à partir de types simples

Il est possible également de créer un type complexe à partir d'un type simple.
On peut avoir besoin de définir un élément contenant une valeur simple, et possédant un attribut, comme <poids unite="kg">67</poids>, par exemple. Un tel élément ne peut pas être déclaré de type simple, car il contient un attribut. Il faut *dériver* un type complexe à partir dui type simple positiveInteger :

```
<xsd:complexType name="typePoids">
  <xsd:simpleContent>
    <xsd:extension base="xsd:positiveInteger">
      <xsd:attribute name="unite" type="xsd:string" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

4.5. Espaces de noms

4.5.1. Introduction

La notion d'espace de nom est complexe ; elle permet à un document XML quelconque d'utiliser les balises définies dans un schéma donné... quelconque. Nous avons déjà utilisé cette notion :

Dans les feuilles de style XSL, où nous utilisions des éléments *préfixés* par xsl, après avoir écrit
<xsl:stylesheet version="1.0"
xmlns:xsl=http://www.w3.org/1999/XSL/Transform">

dans un schéma, où nous utilisions des éléments préfixés par xsd, après avoir écrit <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

Cela signifie que l'*espace de nom* auquel ces balises font référence, là où elles sont définies, est un schéma particulier, que l'on peut consulter.

4.5.2. Comment lier un fichier XML à un schéma ?

Nous n'allons pas ici entrer dans les détails de la notion d'espace de nom, mais simplement apprendre à valider un document XML d'après un Schema XML. On utilise pour ce faire le préfixe xmlns.

Nous avons déjà vu le cas, équivalent à une DTD de type PUBLIC, où le schéma est... public. Un schéma est en effet un document XML, et on trouve dans son élément racine l'attribut xmlns:xsd="http://www.w3.org/2001/XMLSchema". Cela signifie que dans le document, tous les éléments commençant par xsd sont référencés à cette URL. Donc si on a déposé un schéma à l'adresse http://www.monsite.org/collection_schemas/biblio, on peut l'appeler par <xsd:biblio xmlns="http://www.monsite.org/collection_schemas/biblio">.

Dans le cas d'une référence locale, correspondant à une DTD de type SYSTEM, on fait référence au schéma dans le document XML en utilisant l'attribut noNamespaceSchemaLocation, par

<biblio xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="lien_relatif_vers_le_schema">. Par exemple

<biblio xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="biblio10.xsd">.

4.6. Les dérivations

4.6.1. Introduction

Les types simples et complexes permettent déjà de faire plus de choses que les déclarations dans le langage DTD. Il est possible de raffiner leur déclaration de telle manière qu'ils soient une "restriction" ou une extension d'un type déjà existant, en vue de préciser un peu plus leur forme. Nous allons nous limiter dans ce cours d'initiation à la restriction des types simples.

4.6.2. Restriction de type

4.6.2.1. Généralités

On peut appliquer une "dérivation" aussi bien à un type simple, qu'à un type complexe. La dérivation par restriction permet de créer de nouveaux types simples à partir des types simples prédéfinis par le format XML Schema. On utilise pour ce faire des "facettes", qui sont des contraintes supplémentaires appliquées à un type simple particulier.

Une "facette" permet de placer une contrainte sur l'ensemble des valeurs que peut prendre un type de base. Par exemple, on peut souhaiter créer un type simple, appelé MonEntier, limité aux valeurs comprises entre 0 et 99 inclus. On dérive ce type à partir du type simple prédéfini nonNegativeInteger, en utilisant la facette maxExclusive.

```
<xsd:simpleType name="monEntier">
  <xsd:restriction base="nonNegativeInteger">
    <xsd:maxExclusive value="100" />
  </xsd:restriction>
</xsd:simpleType>
```

Il existe un nombre important de facettes qui permettent de :

fixer, restreindre ou augmenter la longueur minimale ou maximale d'un type simple
énumérer toutes les valeurs possibles d'un type
prendre en compte des expressions régulières
fixer la valeur minimale ou maximale d'un type (voir l'exemple ci-dessus)
fixer la précision du type...

4.6.2.2. Exemples

On peut utiliser cette fonctionnalité pour reproduire ce qui, dans les DTD, permettait de limiter les valeurs de certains attributs. Ainsi...

```
<xsd:attribute name="jour" type="typeJourSemaine"
use="required" />
<xsd:simpleType name="jourSemaine">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="lundi" />
  <xsd:enumeration value="mardi" />
  <xsd:enumeration value="mercredi" />
  <xsd:enumeration value="jeudi" />
  <xsd:enumeration value="vendredi" />
  <xsd:enumeration value="samedi" />
  <xsd:enumeration value="dimanche" />
 </xsd:restriction>
</xsd:simpleType>
```

Pour limiter la longueur d'une chaîne :

```
<xsd:simpleType name="typeMotLangueFrancaise">
 <xsd:restriction base="xsd:string">
<xsd:length value="21" />
 </xsd:restriction>
</xsd:simpleType>
```

Plus complexe, on peut utiliser des *expressions régulières*, qui permettent de spécifier quels sont les caractères autorisés, à l'aide de l'élément <xsd:pattern>. Par exemple...

```
<xsd:simpleType name="typeAdresseElectronique">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="(.)+@(.)+" />
 </xsd:restriction>
</xsd:simpleType>
```

Dans cet exemple, (.)+ signifie que l'on peut mettre n'importe quel caractère au moins une fois, et qu'entre les deux chaînes doit impérativement apparaître le caractère @.

Un numéro ISBN est un référent international pour une publication. Il s'agit d'un numéro à 10 chiffres. On peut le déclarer ainsi :

```
<xsd:simpleType name="typeISBN">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="[0-9]{10}" />
 </xsd:restriction>
</xsd:simpleType>
```

Bien sûr, toutes les facettes ne sont pas acceptées par tous les types. Il serait fastidieux ici d'en donner la liste ; elle est accessible sur le site du W3C à l'adresse http://www.w3.org/TR/xmlschema-0/#SimpleTypeFacets.

4.7. Diverses autres fonctionnalités

4.7.1. Inclusion de schémas

Un Schema XML a rapidement tendance à devenir... verbeux. Autrement dit, il devient rapidement suffisamment long pour que sa complexité apparente rebute l'oeil humain. Il est alors plus raisonnable de le scinder en plusieurs morceaux, chacun chargé de définir précisément tel ou tel sous-domaine du schéma. Il est alors possible d'*inclure* plusieurs schémas dans un seul, à l'aide de l'élément xsd:include. Cela offre de plus l'énorme avantage de modulariser un schéma, et donc de pouvoir sans difficulté importer certaines parties à partir de schémas déjà existants. Nul besoin de réinventer la roue, ou de procéder à de masifs "copier/coller", en ayant besoin à chaque fois que l'on fait une mise à jour, de tout reprendre.

Supposons par exemple que nous ayons défini le schéma d'une bibliographie dans le cadre plus général de l'écriture d'un mémoire. Ce Schema XML est stocké à l'URL http://www.monsite.org/schemas/biblio.xsd. Nous avons besoin d'une bibliographie pour une autre aplication -par exemple, lister un ensemble de ressources pour un site Web de e-formation. Nous pouvons inclure le schéma précédemment écrit à l'aide de la commande...

```
<xsd:include
schemaLocation="http://www.monsite.org/schemas/biblio.xsd" />
```

Il s'agit d'une sorte de "copier-coller" du contenu de la bibliographie dans le schéma en cours d'écriture. La seule condition est que le targetNameSpace soit le même dans le Schema XML inclus et dans le Schema XML importateur.

4.7.2. Documentation

XML Schema permet, outre l'utilisation des commentaires comme tout format XML, l'adjonction de documentation aux éléments.

La documentation à l'intention des lecteurs humains peut être définie dans des éléments xsd:documentation, tandis que les informations à l'intention de programmes doivent être incluses dans des éléments xsd:appinfo. Ces deux éléments doivent être placés dans un élément xsd:annotation. Ils disposent d'attributs optionnels : xml:lang et source, qui est une référence à une URI pouvant être utilisée pour identifier l'objectif du commentaire ou de l'information.

Les élémenst xsd:annotation peuvent être ajoutés au début de la plupart des constructions. Voir par exemple le schéma biblio10.xsd déjà donné.

4.7.3. Attribut null

Il est toujours préférable de pouvoir indiquer explicitement qu'un élément est non renseigné plutôt que d'omettre cet élément. La valeur null des bases de données relationnelles est utilisée dans ce but. XML Schema intègre un mécanisme similaire permettant d'indiquer qu'un élément peut être non renseigné.

Nous déclarons maintenant l'élément courriel comme pouvant être null à l'aide de l'attribut nullable du vocabulaire de XML Schema :

```
<personne>
  <nom>Jean Dupont</nom>
  <courriel xsi:null></courriel>
</personne>
```

Cet attribut doit toujours être préfixé par xsi. Quant à l'élément portant cet attribut, il peut contenir d'autres attributs, mais pas de sous-élément.

5. Chapitre IV : XSL

5.1. Présentation

5.1.1. Introduction

XSL signifie *eXtensive Stylesheet Langage*, ou langage extensible de feuille de style. XSLT signifie *eXtensible Stylesheet Langage Transformation*.

Comme son nom l'indique, le XSL ne sert pas uniquement de langage de feuille de style. Il est aussi un très puissant manipulateur d'éléments. Il permet de transformer un document XML source en un autre, permettant ainsi, à l'extrême, d'en bouleverser la structure.

Le fonctionnement du XSL est fondé sur les manipulations de modèles (*templates*). Les éléments du document XML d'origine sont remplacés (ou légèrement modifiés) par ces modèles. Un modèle contient ainsi le texte (éléments, attributs, texte...) de remplacement d'un élément donné.

Tout élément pouvant être remplacé dans le fichier de sortie par tout type de contenu texte, XSL est un outil privilélié de production de fichiers HTML à partir de sources XML. PHPfait ainsi appelà des bibliothèques de procédures de type XSL quand il doit gérer l'interfaçage avec des bases de données XML.

Un fichier XSL étant un fichier XML, il doit respecter les normes de syntaxe de ce format.

5.1.2. Structure d'un document XSL

La structure de base d'un document XSL commence par un *prologue*, puis un élément <xsl:stylesheet pouvant contenir quelques attributs, notamment une déclaration d'espace de noms ainsi que le numéro de version. L'exemple suivant présente l'appel à cet élément tel que nous le pratiquerons dans ce cours :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> (...)
</xsl:stylesheet>
```

L'élément <xsl:stylesheet> est l'élément racine du document XSL. C'est lui qui contient tous les modèles, y compris celui qui est associé à la racine du document XML, modèle que l'on note <xsl:template match="/">. L'attribut match="/" indique que ce modèle s'applique à la racine du document XML.

5.2. Exemples de mises en forme

5.2.1. Exemple simple

5.2.1.1. Introduction

Il est possible de traiter de manière simple un fichier XML contenant une information relativement linéaire. Ainsi, l'exemple déjà présenté d'une composition de bouteille d'eau, dans le cas d'une seule bouteille, se prête facilement à une simple mise en forme HTML.

5.2.1.2. Exemple

Exemple d'un document XML lié à une feuille de style XSL simple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="bouteille1.xsl"?>
<bouteille>
  <marque>Cristaline</marque>
  <composition>calcium 71mg/l, magnésium 5,5mg/l, chlorure 20mg/l,
nitrate 1mg/l, traces de fer.</composition>
  <source>
    <ville>St-Cyr la Source</ville>
    <departement>Loiret</departement>
  </source>
  <code_barre>3274080005003</code_barre>
  <contenance>150cl</contenance>
  <ph>7,45</ph>
</bouteille>
```

Et voici la feuille de style XSL associée :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <head>
      <title>Exemple de sortie HTML</title>
      <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1" />
    </head>
    <body>
      <h1>Bouteille de marque <xsl:value-of
select="bouteille/marque" /></h1>
      <h2>Composition:</h2>
      <p><xsl:value-of select="bouteille/composition" /></p>
      <h2>Lieu d'origine:</h2>
      <p>Ville de <b><xsl:value-of
select="bouteille/source/ville" /></b>, dans le
département <b><xsl:value-of
select="bouteille/source/departement" /></b></p>
```

```
      <h2>Autres informations</h2>
      <ul>
        <li>Contenance: <xsl:value-of
select="bouteille/contenance" /></li>
        <li>pH: <xsl:value-of select="bouteille/ph" /></li>
      </ul>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

5.2.2. Exemple avec boucle

5.2.2.1. Introduction

Il arrive que la structure du document XML ne soit pas linéaire, mais fondée sur, par exemple, des boucles, ou bien comporte un nombre indéterminé de fois un même élément ; c'est d'ailleurs le plus souvent le cas.

On peut ainsi reprendre l'exemple de la bouteille d'eau, qui se présente sous la forme du fichier XML suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="bouteille1.xsl"?>
<bouteille>
    <marque>Cristaline</marque>
    <composition>
      <ion_positif>calcium 71mg/l</ion_positif>
      <ion_negatif>nitrate 1mg/l</ion_negatif>
      <ion_positif>magnésium 5,5mg/l</ion_positif>
      <ion_negatif>chlorure 20mg/l</ion_negatif>
      <autres_materiaux>fer</autres_materiaux>
    </composition>
    <source>
      <ville>St-Cyr la Source</ville>
      <departement>Loiret</departement>
    </source>
  <code_barre>3274080005003</code_barre>
  <contenance>150cl</contenance>
  <ph>7,45</ph>
</bouteille>
```

Cette fois-ci, il faut tenir compte d'un nombre *a priori* indéterminé d'éléments ion_positif, par exemple. Il suffit pour cela d'introduire dans la feuille de style un élément <xsl:for-each select="ce_qu_on_cherche_a_afficher"/>, qui permet de faire une boucle sur l'élément cherché :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <head>
      <title>Exemple de sortie HTML</title>
      <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1" />
    </head>
    <body>
      <h1>Bouteille de marque <xsl:value-of
select="bouteille/marque" /></h1>
      <h2>Composition:</h2>
      <h3>Ions positifs</h3>
      <ul>
        <xsl:for-each select="bouteille/composition/ion_positif">
          <li><xsl:value-of select="." /></li>
        </xsl:for-each>
      </ul>
      <h3>Ions négatifs</h3>
      <ul>
        <xsl:for-each select="bouteille/composition/ion_negatif">
          <li><xsl:value-of select="." /></li>
        </xsl:for-each>
      </ul>
      <h3>Autres matériaux</h3>
      <ul>
        <xsl:for-each select="//autres_materiaux">
          <li><xsl:value-of select="." /></li>
        </xsl:for-each>
      </ul>
<h2>Lieu d'origine</h2>
      <p>Ville de <b><xsl:value-of
select="bouteille/source/ville" /></b>, dans le département
<b><xsl:value-of select="bouteille/source/departement" /></b></p>
      <h2>Autres informations</h2>
      <ul>
        <li>Contenance: <xsl:value-of
select="bouteille/contenance" /></li>
        <li>pH: <xsl:value-of select="bouteille/ph" /></li>
      </ul>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

1. la présence du caractère / à la fin de l'élément "Retour-Ligne"
. Dans le fichier HTML produit, il n'apparaît plus.
2. le réarrangement des entrées : les ions positifs sont de toutes manières affichés *avant* les ions négatifs.
3. la sélection des "autres matériaux" avec select="//autres_materiaux". Cela est une des manières de faire référence à un élément dans l'arbre des éléments. Nous verrons la liste de ces sélecteurs plus tard.

6. Chapitre V : XSLT

6.1. Les expressions de sélection

6.1.1. Introduction

Connues en anglais sous le nom de *XSL patterns*, les expressions de sélection sont des chaînes de caractères qui permettent de sélectionner des noeuds dans un document source. Il est également possible d'utiliser une syntaxe spéciale, appelée XPath, qui permet, en se fondant sur la structure arborescente du document XML (le *Document Object Model* -DOM), de faire référence à des éléments et/ou des attributs.

6.1.2. Sélection d'éléments et d'attributs

6.1.2.1. Sélection d'élément, syntaxe de base

L'exemple le plus simple d'expression de sélection est le *nom* d'un type d'élément. Cette expression sélectionne tous les éléments du type précisé descendant ou ancêtre d'un noeud donné. Exemple :

```
<xsl:value-of select="nom_element" />
```

L'opérateur / permet de définir le chemin d'accès aux éléments à sélectionner, et donc leur parenté. Par exemple, section/paragraphe sélectionne les éléments section du noeud courant et pour chaque élément section, sélectionne les éléments paragraphe qu'il contient. En d'autres termes, cette expression sélectionne les petits-fils paragraphe du noeud courant qui ont pour père un noeud section.

Un nom d'élément peut être remplacé par * dans une expression. Par exemple, */paragraphe sélectionne tous les petits-fils paragraphe quel que soit leur père.

L'utilisation de // permet d'appliquer la recherche aux descendants et non pas seulement aux fils directs. Par exemple, section//paragraphe sélectionne tous les éléments paragraphe descendant d'un élément section fils direct du noeud courant.

Le caractère **.** sélectionne le noeud courant. Par exemple, .//paragraphe sélectionne tous les descendants paragraphe du noeud courant.

La chaîne **..** sélectionne le père du noeud courant. Par exemple, ../paragraphe sélectionne tous les noeuds paragraphe frères du noeud courant.

6.1.2.2. Sélection d'élément, appel de fonctions

L'expression comment() sélectionne tous les noeuds commentaires fils du noeud courant.

L'expression text() sélectionne tous les noeuds fils du noeud courant, ne contenant que du texte.

L'expression node() sélectionne tous les noeuds fils du noeud courant.

L'expression id("UnIdentifiant") sélectionne l'élément, normalement unique, qui a un attribut attr de type ID (*cf.* le paragraphe "Déclaration d'attribut de type ID") valant "UnIdentifiant".

6.1.2.3. Sélection d'élément et DOM

Il est également possible de "naviguer" dans les branches de l'arborescence du document XML, en utilisant les ressources du DOM. Différents types de syntaxes sont possibles, fondées sur une expression de la forme Element[Expression]. Par exemple :

elt[i] où i est un nombre entier désigne le i-ème descendant direct d'un même parent ayant le nom indiqué. Par exemple, paragraphe[3] désigne le *3*ème enfant de l'élément courant, portant le nom paragraphe. *Attention*, la numérotation commence à 1 et non à 0.

elt[position()>i] où i est un nombre entier sélectionne tous les éléments précédés d'au moins i éléments de même nom comme descendants du même parent. Par exemple, paragraphe[position()>5] sélectionne tous les éléments paragraphe dont le numéro d'ordre est strictement supérieur à 5.

elt[position() mod 2 =1] sélectionne tout élément qui est un descendant impair.

elt[souselt] sélectionne tout élément elt qui a au moins un descendant souselt (à ne pas confondre avec elt/souselt, qui sélectionne tout élément souselt ayant pour parent elt...).

elt[first-of-any()] sélectionne le premier élément elt fils de l'élément courant.

elt[last-of-any()] sélectionne le dernier élément elt fils de l'élément courant.

elt[first-of-type()] sélectionne l'élément elt fils de l'élément courant, s'il est premier de son type. Par exemple, si l'élément section peut contenir des noeuds de type texte paragraphe et sous_section dans le désordre, l'expression paragraphe[first-of-type()] ne sélectionnera le premier élément paragraphe que s'il n'y a aucun élément sous_section avant lui.

elt[last-of-type()] sélectionne de même l'élément elt fils de l'élément courant, s'il est le dernier de son type.

Par exemple, l'expression section/paragraphe[last-of-type() and first-of-type()] sélectionne les éléments paragraphe fils uniques dont le père est un élément section ; l'expression section/paragraphe[last-of-any() and first-of-any()] sélectionne les éléments paragraphe dont le père est un élément section qui ne contient qu'un seul élément paragraphe.

La fonction ancestor() permet la sélection d'un ancêtre du noeud courant. Elle reçoit en argument une expression de sélection et recherche le premier ancêtre du nom correspondant à la sélection. Par exemple, ancestor(chapitre)/titre sélectionne l'élément titre du chapitre contenant l'élément courant.

6.1.2.4. Sélection d'attributs

Les attributs d'un élément sont sélectionnés en faisant précéder leur nom par le caractère @. Les règles relatives à la sélection des éléments s'appliquent également aux attributs :

section[@titre] sélectionne les éléments section qui ont un attribut titre.
section[@titre="Introduction"] sélectionne les éléments section dont l'attribut titre a pour valeur Introduction.

Si l'on veut *afficher* le contenu de l'attribut, on le fait précéder du caractère /. Par exemple, <xsl:value-of select=paragraphe/@titre permet l'affichage du titre de l'élément paragraphe fils de l'élément courant (si rien n'est précisé, par défaut il s'agit du premier élément paragraphe fils).

6.1.2.5. Opérateurs logiques

Les opérateurs logiques not(), and et or peuvent être utilisés, comme par exemple section[not(@titre)] sélectionne les éléments section qui n'ont pas d'attribut titre. Attention : lorsque, dans la DTD par exemple, l'attribut est défini comme ayant une valeur par défaut, même s'il n'est pas explicité dans le document XML, il est considéré comme existant.

6.2. XPath

6.2.1. Introduction

Comme son nom l'indique, XPath est une spécification fondée sur l'utilisation de chemin d'accès permettant de se déplacer au sein du document XML. Dans ce but, un certain nombre de fonctions ont été définies. Elles permettent de traiter les chaînes de caractères, les booléens et les nombres.

Le XPath établit un arbre de noeuds correspondant au document XML. Les types de noeuds peuvent être différents : nœud d'élément, nœud d'attribut et nœud de texte. En vue d'une utilisation plus aisée, le XPath comprend un mécanisme qui associe à tous ces types une chaîne de caractères.

La syntaxe de base du XPath est fondée sur l'utilisation d'expressions. Une expression peut s'appliquer à quatre types d'objets :

un ensemble non ordonné de noeuds ;
une valeur booléenne (vrai ou faux) ;
un nombre en virgule flottante ;
une chaîne de caractères.

Chaque évaluation d'expression dépend du contexte courant. Une des expressions les plus importantes dans le standard XPath est le *chemin de localisation*. Cette expression sélectionne un ensemble de nœuds à partir d'un nœud contextuel.

6.2.2. Chemin de localisation

6.2.2.1. Introduction

Un chemin de localisation peut être de type absolu ou relatif.

Dans le cas où il est de type absolu, il commence toujours par le signe / indiquant la racine du document XML ;

Dans le cas où il est de type relatif, le nœud de départ est le nœud contextuel courant.

La syntaxe de composition d'un chemin de localisation peut être de type abrégé ou non abrégé. Toute syntaxe non abrégée ne trouve pas forcément d'équivalence en syntaxe abrégée.

Un chemin de localisation est composé de trois parties :

un axe, définissant le sens de la relation entre le nœud courant et le jeu de nœuds à localiser;

un nœud spécifiant le type de nœud à localiser;

0 à n prédicats permettant d'affiner la recherche sur le jeu de nœuds à récupérer.

Par exemple, dans le chemin child::section[position()=1], child est le nom de l'axe, section le type de nœud à localiser (élément ou attribut) et [position()=1] est un prédicat. Les doubles :: sont obligatoires.

La syntaxe d'une localisation s'analyse de gauche à droite. Dans notre cas, on cherche dans le nœud courant, un nœud section qui est le premier nœud de son type.

6.2.2.2. Axes

child : contient les enfants directs du nœud contextuel.

descendant : contient les descendants du nœud contextuel. Un descendant peut être un enfant, un petit-enfant...

parent : contient le parent du nœud contextuel, s'il y en a un.

ancestor : contient les ancêtres du nœud contextuel. Cela comprend son père, le père de son père... Cet axe contient toujours le nœud racine, excepté dans le cas où le nœud contextuel serait lui-même le nœud racine.

following-sibling : contient tous les nœuds cibles du nœud contextuel. Dans le cas où ce nœud est un attribut ou un espace de noms, la cible suivante est vide.

preceding-sibling : contient tous les prédécesseurs du nœud contextuel ; si le nœud contextuel est un attribut ou un espace de noms, la cible précédente est vide.

following : contient tous les nœuds du même nom que le nœud contextuel situés après le nœud contextuel dans l'ordre du document, à l'exclusion de tout descendant, des attributs et des espaces de noms.

preceding : contient tous les nœuds du même nom que le nœud contextuel situés avant lui dans l'ordre du document, à l'exclusion de tout descendant, des attributs et des espaces de noms.

attribute : contient les attributs du nœud contextuel ; l'axe est vide quand le nœud n'est pas un élément.

namespace : contient tous les nœuds des espaces de noms du nœud contextuel ; l'axe est vide quand le nœud contextuel n'est pas un élément.

self : contient seulement le nœud contextuel.

descendant-or-self : contient le nœud contextuel et ses descendants.

ancestor-or-self : contient le nœud contextuel et ses ancêtres. Cet axe contiendra toujours le nœud racine.

6.2.2.3. Prédicats

Le contenu d'un prédicat est une prédiction. Chaque expression est évaluée et le résultat est un booléen.

Par exemple, section[3] est équivalent à section[position()=3].

Ces deux expressions sont équivalentes : chacune d'entre elles produit un booléen. Dans le premier cas, il n'y a pas de test, on sélectionne simplement le troisième élément, l'expression est obligatoirement vraie. Dans le second cas, un test est effectué par rapport à la position de l'élément section ; lorsque la position sera égale à 3, l'expression sera vraie.

6.2.2.4. Syntaxe non abrégée

Cette syntaxe va être présentée par plusieurs exemples... Les parties qui dépendent du fichier XML analysé sont écrites de cette manière.

child::para : sélectionne l'élément para enfant du nœud contextuel.

child::* : sélectionne tous les éléments enfants du nœud contextuel.

child::text() : sélectionne tous les nœuds de type texte du nœud contextuel.

child::node() : sélectionne tous les enfants du nœud contextuel, quel que soit leur type.

attribute::name : sélectionne tous les attributs name du nœud contextuel.

attribute::* : sélectionne tous les attributs du nœud contextuel.

descendant::para : sélectionne tous les descendants para du nœud contextuel.

ancestor::div : sélectionne tous les ancêtres div du nœud contextuel.

ancestor-or-self::div : sélectionne tous les ancêtres div du nœud contextuel et le nœud contextuel lui-même si c'est un div.

descendant-or-self::para : sélectionne tous les descendants para du nœud contextuel et le nœud contextuel lui-même si c'est un para.

self::para : sélectionne le nœud contextuel si c'est un élément para, et rien dans le cas contraire.

child::chapitre/descendant::para : sélectionne les descendants para de l'élément chapitre enfant du nœud contextuel.

child::*/child::para : sélectionne tous les petits-enfants para du nœud contextuel.

/child:: : sélectionne l'élément racine du document.

/descendant::para : sélectionne tous les éléments para descendants du document contenant le nœud contextuel.

/descendant::olist/child::item : sélectionne tous les éléments item qui ont un parent olist et qui sont dans le même document que le nœud contextuel.

child::para[position()=1] : sélectionne le premier enfant para du nœud contextuel.

child::para[position()=last()] : sélectionne le dernier enfant para du nœud contextuel.

child::para[position()=last()-1] : sélectionne l'avant-dernier enfant para du nœud contextuel.

child::para[position()1] : sélectionne tous les enfants para du nœud contextuel autres que le premier.

following-sibling::para[position()=1] : sélectionne le prochain chapitre cible du nœud contextuel.

On pourrait continuer encore longtemps cette liste d'exemples. Cette syntaxe non-abrégée permet beaucoup de raffinement. child::*[self::chapitre or self::sstitre][position()=last()] permet ainsi de sélectionner le dernier enfant chapitre ou sstitre du nœud contextuel.

6.2.2.5. Syntaxe abrégée

Cette syntaxe recoupe en fait la "syntaxe de base" vue plus haut. Elle permet d'obtenir des expressions du type para[@type="avertissement"][5], qui sélectionne le cinquième enfant de l'élément para, parmi ceux qui ont un attribut type ayant la valeur avertissement.

6.2.3. Fonctions de base

6.2.3.1. Généralités

De nombreuses fonctions peuvent être utilisées. Ces fonctions concernent quatre catégories d'objets : nœuds, chaînes de caractères, booléens, nombres. Chaque fonction peut avoir zéro ou plusieurs arguments. Dans les descriptions suivantes, lorsqu'un élément est suivi du caractère ?,cela signifie qu'il est optionnel. Cette liste est loin d'être exhaustive. Le chapitre suivant présente les fonctions XPath de manière plus complète.

6.2.3.2. Manipulation de nœuds

Fonctions retournant un nombre :

last() : retourne un nombre égal à l'index du dernier nœud dans le contexte courant.

position() : retourne un nombre égal à la position du nœud dans le contexte courant.

Fonction retournant un jeu de nœuds : id(objet), permet de sélectionner les éléments par leur identifiant.

6.2.3.3. Manipulation de chaînes de caractères

Beaucoup de fonctions existent. Citons pour mémoire notamment :

string(noeud?) : cette fonction convertit un objet en chaîne de caractères selon les règles suivantes :

un ensemble de nœuds est converti en chaîne de caractères en retournant la valeur textuelle du premier nœud de l'ensemble dans l'ordre du document. Si l'ensemble des nœuds est vide, une chaîne vide est retournée.

un nombre est converti en chaîne suivant des règles dépendant de sa nature (`NaN`, nombre entier, non-entier, zéro...).

la valeur booléenne `false` est convertie en chaîne de caractères `"false"`, de même pour la valeur booléenne `true`.

Cette fonction n'a pas pour objet de convertir des nombres en chaînes de caractères pour les présenter aux utilisateurs ; il existe des fonctions de transformations `XSLT` pour ce faire (`format-number` et `xsl:number`)

concat(chaine1, chaine2, chaine*) : retourne une chaîne résultant de la compilation des arguments

string-length(chaine?) : cette fonction retourne le nombre de caractères de la chaîne. Dans le cas où l'argument est omis, la valeur retournée est égale à la longueur de la valeur textuelle du nœud courant

6.2.3.4. Manipulation de booléens

Outre la fonction logique not(), ainsi que les fonctions true() et false(), une fonction utile est lang(chaine). Elle teste l'argument chaine par rapport à l'attribut xml:lang du nœud contextuel ou de son plus proche ancêtre dans le cas où le nœud contextuel ne contient pas d'attribut de ce type. La fonction retourne true si l'argument est bien la langue utilisée ou si la langue utilisée est un sous-langage de l'argument (par exemple, en//us). Sinon elle retourne false.

6.2.3.5. Manipulation de nombres

Voici quelques fonctions de manipulations de nombres :

`floor(nombre)` : retourne le plus grand entier inférieur à l'argument passé à la fonction.
`ceiling(nombre)` : retourne le plus petit entier supérieur à l'argument passé à la fonction.
`round(nombre)` : retourne l'entier le plus proche de l'argument passé à la fonction.

6.2.4. Éléments XSLT

Généralités

Introduction

Dans cette partie, nous allons détailler quelques éléments XSLT et présenter des exemples d'utilisation. Ne sera introduite ici qu'environ la moitié des éléments de formatage XSLT ; libre à vous de vous renseigner sur les éléments manquants.

Le nom de domaine utilisé pour les exemples est celui de la spécification 1.0 : http://www.w3.org/TR/xslt. C'est celui qui est le plus abouti (il est à l'état de *Recommendation*, alors que la spécification XSLT 2.0 est encore à l'état de *Working Draft*).

Rappel : prologue d'un document XSL

Un fichier XSL doit commencer par les lignes indiquant le numéro de version XML et l'encodage de caractères utilisé :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

2. Les fondamentaux

Généralités

Sont présentés ici les éléments de base du document XSL, ceux que l'on retrouve dans l'immense majorité des cas, voire la totalité pour certains.

<xsl:stylesheet>

Cet élément *doit* être l'élément racine du document XSL, et en tant que tel doit figurer juste après le prologue (et les commentaires éventuels qui suivraient celui-ci). Il contient tous les autres éléments de mise en forme. Dans le cas général, l'utilisation de cet élément est de la forme :

```
<xsl:stylesheet id="id" version="nombre" xmlns:pre="URI"> (...)
</xsl:stylesheet>
```

id est l'identifiant unique de la feuille de style. version est le numéro de version de la feuille de style XSLT. A l'heure actuelle, la version peut être 1.0 ou 1.1. xmlns:pre correspond à la définition de l'espace de noms. pre indique le préfixe qui sera utilisé dans la feuille de style pour faire référence à l'URI de l'espace nominal. Exemples :

```
<xsl:stylesheet version="1.0" xmlns:xsl="uri:xsl"> (...)
</xsl:stylesheet>
```

... permet d'avoir accès uniquement à des fonctions de base.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> (...)
</xsl:stylesheet>
```

... permet d'avoir accès à des fonctions évoluées d'XSLT. C'est cette déclaration que nous allons utiliser dans la suite.

<xsl:output>

Cet élément vide, à placer comme premier enfant de <xsl:stylesheet>, permet de spécifier des options concernant l'arbre de sortie. L'utilisation de cet élément est de la forme :

```
<xsl:output method="xml | html | text" version="nmtoken"
encoding="chaine" omit-xml-declaration="yes | no" standalone="yes |
no" doctype-public="chaine" doctype-system="chaine" cdata-section-
elements="elt" indent="yes | no" media-type="chaine" />
```

`method` identifie la méthode de transformation. Dans le cas où elle est égale à `text`, aucune mise en forme n'est effectuée.
`version` identifie la version de la méthode de sortie (xml 1.0, html 4.01...).
`encoding` indique la version du jeu de caractères à utiliser pour la sortie.
`omit-xml-declaration` indique au processeur XSLT s'il doit ajouter ou non une déclaration XML.
`standalone` indique au processeur XSLT s'il doit créer un arbre de sortie avec ou sans déclaration de type de document.
`doctype-public` indique l'identifiant public utilisé par la `DTD` associée à la transformation.
`doctype-system` indique l'identifiant system utilisé par la `DTD` associée à la transformation.
`cdata-section-elements` indique les éléments dont le contenu doit être traité lors de la transformation *via* une section `CDATA`.
`indent` présente la transformation sous forme d'arbre dans le cas où la valeur de cet attribut est égale à `yes`.
`media-type` indique le type MIME des données résultantes de la transformation.

Par exemple :

```
<xsl:output method="html" version="html4.01" encoding="ISO-8859-1"
doctype-public="-//W3C//DTD HTML 4.01//EN" doctype-
system="http://www.w3.org/TR/html4/strict.dtd" />
```

... permet d'indiquer que le fichier de sortie sera au format HTML 4.01, conforme à la `DTD` publique de l'`HTML` du W3C, et que le jeu de caractères utilisé est l'ISO-8859-1. Le résultat en sortie est un fichier `HTML` commençant par

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">
```

... et possédant dans son `<head>`

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1" />
```

L'utilisation d'un tel élément permet donc de se dispenser de la spécification explicite du codage de caractères via un élément `<meta>` lors de l'écriture de la feuille de style `XSL`.

<xsl:template>

Cet élément définit un modèle à appliquer à un nœud et à un contexte spécifiques. L'utilisation de cet élément est de la forme :

```
<xsl:template name="nommodele" match="expression" mode="modemodele">
</xsl:template>
```

`name` correspond au nom associé au modèle.

`match` indique quel jeu de nœuds sera affecté par le modèle. Cette expression peut comprendre un test d'existence d'attribut, le caractère `|` indiquant que le modèle s'applique à un élément ou à un autre, ainsi que tout élément permettant de définir un jeu d'attributs.

`mode` permet à un élément d'avoir plusieurs modèles, chacun générant une sortie différente.

<xsl:value-of>

Cet élément permet d'insérer la valeur d'un nœud dans la transformation. Ce nœud est évalué en fonction d'une expression. Cette expression peut correspondre à un élément, à un attribut ou à tout autre nœud contenant une valeur. L'utilisation de cet élément est de la forme :

```
<xsl:value-of select="expression" disable-output-escaping="yes |
no" />
```

La valeur de `select` est évaluée et c'est cette évaluation qui sera insérée dans la transformation.

`disable-output-escaping` agit sur la transformation des caractères. Dans le cas où sa valeur est `yes`, la chaîne `<` est insérée dans la transformation en tant que signe `<`, ce qui peut entraîner des erreurs. Dans le cas contraire, la chaîne `<` est insérée telle quelle dans la transformation.

Ajout d'éléments et d'attributs

<xsl:element>

Cet élément insère un nouvel élément dans la transformation. Le nom de l'élément est donné par l'attribut `name`. L'utilisation de cet élément est de la forme :

```
<xsl:element name="nomelement" use-attribute-sets="jeuattr">
</xsl:element>
```

`name` correspond au nom de l'élément à créer. `use-attribute-sets` correspond au jeu d'attributs à associer à l'élément créé. Par exemple :

```
<xsl:element name="p"><xsl:value-of select="texte" /></xsl:element>
```

... permet de créer dans le fichier `HTML` un élément de paragraphe renfermant le contenu de l'élément `texte` du document `XML`.

<xsl:attribute>

Cet élément définit un attribut et l'ajoute à l'élément résultat de la transformation. L'utilisation de cet élément est de la forme :

```
<xsl:attribute name="nom">valeur</xsl:attribute>
```

name correspond au nom de l'attribut à ajouter dans le contexte courant. valeur correspond à la valeur à lui donner. Par exemple :

```
<image><xsl:attribute name="src">test.gif</xsl:attribute></image>
```

... permet d'ajouter à l'élément image l'attribut src et de lui affecter la valeur test.gif, ce qui a pour effet de produire en sortie l'élément suivant :

```
<image src="test.gif"></image>
```

On aurait pu, de manière tout à fait équivalente, écrire

```
<xsl:element name="image"><xsl:attribute
name="src">test.gif</xsl:attribute></xsl:element>
```

... ce qui a pour effet de produire en sortie

```
<image src="test.gif"></image>
```

Syntaxe courte

Il est possible de créer des éléments de manière plus compacte, à l'aide d'une syntaxe particulière. Supposons par exemple que nous ayons dans un fichier XML l'élément <image source="test.gif" texte_alternatif="Image de test"/>, et que nous souhaitions obtenir dans le fichier de sortie l'élément . Il existe deux manières :

La syntaxe longue, avec...

```
<xsl:element name="img"><xsl:attribute name="src"><xsl:value-of
select="@source" /></xsl:attribute><xsl:attribute
name="alt"><xsl:value-of
select="@texte_alternatif" /></xsl:attribute></xsl:element>
```

La syntaxe courte, utilisant des accolades...

```
<img src="{@source}" alt="{@texte_alternatif}" />
```

La seconde syntaxe est plus compacte ; mais elle présente deux inconvénients :

Dès lors que des expressions XPath sont un peu longues, cette syntaxe complique la relecture de la feuille de style ;

Une feuille XSL est avant tout un fichier XML. En tant que tel, on peut souhaiter sa transformation ou son traitement automatisé. La syntaxe longue s'y prête plus facilement.

Gestion des boucles

<xsl:for-each>

Cet élément de bouclage, que l'on a déjà rencontré, crée une boucle dans laquelle sont appliquées des transformations. Son utilisation est de la forme :

```
<xsl:for-each select="jeunoeud"></xsl:for-each>
```

select correspond au jeu de nœuds devant être parcouru par la boucle. Exemple d'utilisation :

```
<ul>
  <xsl:for-each select="item">
    <li><xsl:value-of select="." /></li>
  </xsl:for-each>
</ul>
```

<xsl:sort>

Cet élément permet d'effectuer un tri sur un jeu de nœuds. Il doit être placé soit dans un élément <xsl:for-each> soit dans un élément <xsl:apply-templates>. C'est un élément vide qui peut être appelé plusieurs fois pour effectuer un tri multicritères. Chaque appel à cet élément provoque un tri sur un champ spécifique, dans un ordre prédéfini. L'utilisation de cet élément est de la forme :

```
<xsl:sort select="noeud" data-type="text | number | elt"
order="ascending | descending" lang="nmtoken" case-order="upper-
first | lower-first" />
```

select permet de spécifier un nœud comme clé de tri.

data-type correspond au type des données à trier. Dans le cas où le type est number, les données sont converties puis triés.

order correspond à l'ordre de tri. Cet attribut vaut ascending ou descending.

lang spécifie quel jeu de caractères utiliser pour le tri ; par défaut, il est déterminé en fonction des paramètres système.

case-order indique si le tri a lieu sur les majuscules ou minuscules en premier.

Par exemple :

```
<ul>
  <xsl:for-each select="livre">
    <xsl:sort select="auteur" order="descending" />
    <li><b><xsl:value-of select="auteur" /></b><br /><xsl:value-of
select="titre" /></li>
  </xsl:for-each>
</ul>
```

Dans cet exemple, la liste des livres est classée dans l'ordre alphabétique décroissant des noms d'auteur.

<xsl:number>

Cet élément permet d"insérer un nombre formaté pouvant servir de compteur. L'utilisation de cet élément est de la forme :

```
<xsl:number level="single | multiple | any" count="noeud"
from="noeud" value="expression" format="chaine" lang="nmtoken"
grouping-separator="car" grouping-size="nombre" />
```

level indique quels niveaux doivent être sélectionnés pour le comptage.

count indique quels nœuds doivent être comptés dans les niveaux sélectionnés ; dans le cas où cet attribut n'est pas défini, les nœuds comptés sont ceux ayant le même type que celui du nœud courant.

from identifie le nœud à partir duquel le comptage commence.

value indique l'expression correspondant à la valeur du compteur ; si cet attribut n'est pas défini, le nombre inséré correspond à la position du nœud (position()).

format spécifie le format de l'affichage du nombre ; cela peut être un chiffre, un caractère (a-z, A-Z) et comprendre un caractère de séparation tel que le point (.), le trait d'union (-) ou autre. Les formats possibles sont "1", "01", "a", "A", "i", "I".

lang spécifie le jeu de caractères à utiliser ; par défaut, il est déterminé en fonction des paramètres du système.

grouping-separator identifie le caractère permettant de définir la séparation entre les centaines et les milliers.

grouping-size spécifie le nombre de caractères formant un groupe de chiffres dans un nombre long ; le plus souvent la valeur de cet attribut est 3. Ce dernier attribut fonctionne avec...

...grouping-separator ; si l'un des deux manque, ils sont ignorés.

Exemple d'utilisation :

```
<ul>
<xsl:for-each select="livre">
  <xsl:sort select="auteur" />
  <xsl:number level="any" from="/" format="1." />
    <li><b><xsl:value-of select="auteur" /></b><br /><xsl:value-of
select="titre" /></li>
</xsl:for-each>
</ul>
```

Conditions de test

<xsl:if>

Cet élément permet la fragmentation du modèle dans certaines conditions. Il est possible de tester la présence d'un attribut, d'un élément, de savoir si un élément est bien le fils d'un autre, de tester les valeurs des éléments et attributs. L'utilisation de cet élément est de la forme :

```
<xsl:if test="condition">action</xsl:if>
```

test prend la valeur 1 ou 0 suivant le résultat de la condition (vrai ou faux). action correspond à l'action devant être effectuée (texte à afficher, second test, gestion de chaîne...). Exemple d'utilisation:

```
<ul>
  <xsl:for-each select="livre">
    <li>
      <b><xsl:value-of select="auteur" /><br /></b>
      <xsl:value-of select="titre" />.<xsl:if
test="@langue='français'">Ce livre est en français.</xsl:if>
    </li>
  </xsl:for-each>
</ul>
```

Dans le code précédent, si l'attribut langue de l'élément livre vaut francais, le processeur ajoutera au fichier de sortie la phrase "Ce livre est en français". Il ne se passe rien si ce n'est pas le cas.

<xsl:choose>

Cet élément permet de définir une liste de choix et d'affecter à chaque choix une transformation différente. Chaque choix est défini par un élément <xsl:when> et un traitement par défaut peut être spécifié grâce à l'élément <xsl:otherwise>. Exemple d'utilisation :

```
<ul>
  <xsl:for-each select="livre">
    <li>
```

```
      <b><xsl:value-of select="auteur" /><br /></b>
      <xsl:value-of select="titre" />
      <xsl:choose>
        <xsl:when test="@langue='francais'">Ce livre est en
français.</xsl:when>
        <xsl:when test="@langue='anglais'">Ce livre est en
anglais.</xsl:when>
        <xsl:otherwise>Ce livre est dans une langue non
répertoriée.</xsl:otherwise>
      </xsl:choose>
    </li>
  </xsl:for-each>
</ul>
```

Variables et paramètres

Introduction

Il est possible en XSLT de définir des variables et des paramètres permettant de faire des calculs. Il est nécessaire pour cela de faire appel à deux éléments XSL : **xsl:variable** et **xsl:param**.

Élément <xsl:variable>

L'élément **<xsl:variable>** sert à créer les variables dans XSLT. Il possède les attributs suivants :

- **name** : cet attribut est obligatoire. Il spécifie le... nom de la variable.
- **select** : expression XPath qui spécifie la valeur de la variable.

Par exemple :

```
<xsl:variable name="nombre_livres" select="255" />
<xsl:variable name="auteur" select="'Victor Hugo'" />
<xsl:variable name="nombre_pages" select="livre/tome/@page" />
```

On notera la présence des guillemets imbriqués quand il s'agit d'affecter une chaîne de caractères à une variable.

La portée d'une variable est limitée aux éléments-frères et à leurs descendants. Par conséquent, si une variable est déclarée dans une boucle **xsl:for-each** ou un élément **xsl:choose** ou **xsl:if**, on ne peut s'en servir en-dehors de cet élément.

Une variable est appelée en étant précédée du caractère **$** : **<xsl:value-of select="$nombre_pages" />**.

On peut utiliser une variable pour éviter la frappe répétitive d'une chaîne de caractères, et/ou faciliter la mise à jour de la feuille de style.

L'élément <xsl:call-template>

L'élément `<xsl:template>` peut être appelé indépendamment d'une sélection d'un nœud. Pour cela, il faut renseigner l'attribut `name`, et l'appeler à l'aide de l'élément `<xsl:call-template>`. Par exemple

```
<xsl:template name="separateur">
  <hr />
  <img src="ligne.gif" alt="séparation" width="150" height="2" />
  <hr />
</xsl:template>
```

Il suffit alors de l'appeler avec `<xsl:call-template name="separateur"/>`.

Les éléments <xsl:param> et <xsl:with-param>

Les paramètres créés avec ces deux éléments sont habituellement utilisés dans les modèles nommés, que nous venons de voir. Ils permettent de passer des valeurs aux modèles. Un paramètre est créé avec l'élément `<xsl:param>`, et passé à un modèle avec l'élément `<xsl:with-param>`. Les deux ont deux attributs :

`name`, obligatoire, qui donne un nom au paramètre ;
`select`, une expression `XPath` facultative permettant de donner une valeur par défaut au paramètre.

Par exemple, on peut imaginer un `template` permettant d'évaluer le résultat d'une expression polynômiale :

```
<xsl:template name="polynome">
  <xsl:param name="variable_x" />
  <xsl:value-of select="2*$variable_x*$variable_x+(-
5)*$variable_x+2" />
</xsl:template>
```

Il suffit alors de l'appeler en lui passant diverses valeurs pour le paramètre `variable_x` pour qu'il évalue cette expression. Ce comportement se rapproche de celui d'une fonction dans un langage de programmation. Par exemple...

```
<xsl:call-template name="polynome">
  <xsl:with-param name="variable_x" select="3.4" />
</xsl:call-template>
```

... permet d'afficher le résultat de 2*3.4^2-5*3.4+2. On remarquera que :

1. la soustraction d'un nombre se fait par addition du nombre (négatif) opposé ;
2. la division se fait non par le caractère `/`, mais par l'opérateur `div`.