

## 2 Introduction à XML

### 2.1 Présentation

Le langage XML (eXtended Markup Language) est un langage de format de document. Il dérive de SGML (Standard Generalized Markup Language) et HTML (HyperText Markup Language). Comme ces derniers, il s'agit d'un langage formé de balises qui permet de structurer les documents.

Le langage XML s'est imposé comme le format standard pour les communications entre applications. Il est utilisé dans la plupart des projets de publication sur le WEB ainsi que dans les bases de données.

### 2.2 Historique

1986

Introduction de SGML par Charles Goldfarb.

1991

Introduction de HTML par Tim Berners-Lee pour le WEB.

1998

Publication par le W3C de la version 1.0 d'XML.

1999

Redéfinition de HTML 4.0 en XHTML à travers XML.

2004

Publication par le W3C de la version 1.1 de XML pour une mise à jour des caractères spéciaux en lien avec Unicode.

### 2.3 Intérêts

- Séparation stricte entre contenu et présentation
- Structuration forte du document
- Extensibilité
- Modèles de documents (DTDs et XML-Schémas)
- Modularité des modèles
- Validation du document par rapport au modèle
- Format texte avec gestion des caractères spéciaux
- Format libre

### 2.4 Dialectes et extensions

XLink et XPointer

Liens entre documents

XPath

XQuery

Requêtes

XSL

Transformation de documents

Schémas XML

Modèles de documents

- 2 -

## 1 XML en M2

- Cours le lundi en salle 248E ou Td en salle E (halle aux farines) de 8h30 à 13h
- Bibliographie
- Références
- Logiciels
- Acronymes
- Ce cours au format PDF
- Tds
- Année 2007/2008
- Sujet d'examen de 2007/2008
  
- Présentation du cours
- Cours n° 1 : Introduction à XML
  - Syntaxe et validation
  - DTDs
- Td n° 1 (salle E halle aux farines)
- Cours n° 2 : Espaces de noms et schémas XML
  - Espaces de noms
  - Schémas XML
- Td n° 2 (salle E halle aux farines)
- Cours n° 3 : XPath et XSLT
  - XPath
  - XSLT
- Td n° 3 (salle E halle aux farines)
- Cours n° 4 : CSS, SVG et programmation
  - CSS
  - SVG
  - Programmation
- Td n° 4 (salle E halle aux farines)
- Td n° 5 (salle E halle aux farines)
- Examen le 23 mars de 8h30 à 12h en salle 247E (halle aux farines)

---

# XML

---

Année 2008/2009

Université Paris Diderot

Olivier Carton

Version du 17 Mar 2009

- 1 -

## 3 Syntaxe

### 3.1 Premier exemple

On donne ci-dessous un premier exemple de document XML représentant une petite bibliographie.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- Time-stamp: "bibliography.xml 3 Mar 2008 16:24:04" -->
<!DOCTYPE bibliography SYSTEM "bibliography.dtd" >
<bibliography>
  <book key="Michard01" lang="fr">
    <title>XML langage et applications</title>
    <author>Alain Michard</author>
    <year>2001</year>
    <publisher>Eyrolles</publisher>
    <isbn>2-212-09206-7</isbn>
    <url>http://www.editions-eyrolles/livres/michard/</url>
  </book>
  <book key="Zeldman03" lang="en">
    <title>Designing with web standards</title>
    <author>Jeffrey Zeldman</author>
    <year>2003</year>
    <publisher>New Riders</publisher>
    <isbn>0-7357-1201-8</isbn>
  </book>
</bibliography>
```

### 3.2 Syntaxe et structure

Pour être correct, un document XML doit être *bien formé* et *valide*. La première contrainte est de nature *syntaxique*. Un document bien formé doit respecter certaines règles syntaxiques propres à XML qui sont explicitées ci-dessous. La seconde contrainte est de nature *structurelle*. Un document valide doit suivre un modèle appelé *type* décrit par une DTD (*Document Type Description*) ou un schéma XML.

Un document XML est généralement contenu dans un fichier texte dont l'extension est `.xml`. Il peut aussi être réparti en plusieurs fichiers en utilisant les entités externes. Les fichiers contenant des documents dans un dialecte XML peuvent avoir une autre extension qui précise le format. Les extensions pour les schémas XML, les feuilles de style XSL, les dessins en SVG sont par exemple `.xsd`, `.xsl` et `.svg`.

Un fichier XML contient du texte dans un format de codage d'UNICODE, par exemple UTF-8 ou Latin1. Le codage utilisé par le fichier est précisé dans l'entête du fichier.

### 3.3 Composition globale d'un document

Un document XML est composé des trois constituants suivants.

Prologue  
Il contient des déclarations facultatives.  
Corps du document  
C'est le contenu même du document.

- 4 -

## 2.5 Applications

RSS  
Flux de données  
SVG  
Dessin vectoriel  
SMIL  
Multimédia  
MathML  
Formules mathématiques

- 3 -

UCS-4 ou UTF-32

Chaque caractère est codé directement par son code UNICODE sur quatre octets. Ce codage permet donc de coder tous les caractères UNICODE.

UCS-2

Chaque caractère est codé par son code sur deux octets. Ce codage permet donc uniquement de coder les caractères du BMP.

UTF-16

Ce codage coïncide essentiellement avec UCS-2 à l'exception d'une plage de 2048 positions (0xD800 A 0xDFFF) qui permettent de coder des caractères en dehors du BMP.

UTF-8

Chaque caractère est codé sur un nombre variant de un à quatre octets. Ce codage est le codage par défaut de XML. Les caractères de l'ASCII sont codés sur un seul octet dont le bit de poids fort est 0. Les caractères en dehors de l'ASCII utilisent au moins deux octets. Le premier octet commence par autant de 1 que d'octets dans la séquence suivis par un 0. Les autres octets commencent par 10. Ce codage a l'avantage d'être relativement efficace pour les langues européennes.

ISO-8859-1

Chaque caractère est codé sur un seul octet. Ce codage coïncide avec l'ASCII pour les codes de 0 à 0x7F. Les codes de 0x80 à 0xFF sont utilisés pour les caractères spéciaux (caractères accentués, cédilles, ...) des langues d'Europe de l'ouest.

Il est possible d'insérer n'importe quel caractère UNICODE dans un document XML en utilisant une des deux syntaxes `&#(code décimal)` ou `&#x(code hexadecimal)`. Le caractère euro € peut par exemple être inséré par `&#364` ou `&#x20AC`.

### 3.4.3 Déclaration de type de document

La déclaration de type peut prendre plusieurs formes suivant que la définition du type est incluse dans le document ou externe. Elle a la forme générale suivante.

```
<!DOCTYPE ... >
```

La forme précise de cette déclaration est explicitée à la section portant sur les DTDs.

### 3.5 Corps du document

Le corps du document est constitués d'éléments qui forment son contenu. Chacun de ses éléments contient du contenu textuel, d'autres éléments ou encore un mélange des deux. Cette structure des éléments est semblable à l'arborescence des fichiers et des répertoires d'un système de fichiers.

#### 3.5.1 Éléments

Un *élément* est formé d'une balise ouvrante, d'un contenu et de la balise fermante correspondante. Le nom de l'élément est l'identificateur utilisé dans les balises ouvrantes et fermantes.

Les balises de XML sont semblables à celles de HTML. Par contre, elles ne sont pas limitées à un certain nombre de balises prédéfinies comme en HTML. Il est possible d'utiliser n'importe quel identificateur comme nom d'élément. Cet identificateur doit commencer par une lettre ou le caractère `'_'`. Il peut ensuite contenir des lettres, des chiffres et des caractères `'_'`, `'-'`, `'.'` et `':'`. Le caractère `':'` est réservé à l'utilisation des espaces de noms. La casse des lettres est prise en compte :

- 6 -

Commentaires et instructions de traitement

Ceux-ci peuvent apparaître partout dans le document.

Le document se découpe en fait en deux parties consécutives qui sont le prologue et le corps. Les commentaires et les instructions de traitement sont ensuite librement insérés avant, après et à l'intérieur du prologue et du corps. La structure globale d'un document XML est la suivante.

```
<?xml ... ?> ] Prologue
...
<root> ]
...
</root> ] Corps
```

### 3.4 Prologue

Le prologue contient deux déclarations facultatives mais fortement conseillées. La première est la déclaration XML qui donne entre autre la version de XML et le codage du fichier. La seconde est la déclaration du type du document (DTD) qui définit la structure du document. La déclaration de type est omise lorsqu'on utilise des schémas XML ou d'autres types de modèles qui remplacent les DTDs. La structure globale du prologue est la suivante.

```
<?xml ... ?> ] Déclaration XML
<!DOCTYPE root [ ] DTD
...
]> ] Prologue
```

#### 3.4.1 Déclaration XML

La déclaration XML à généralement la forme suivante. L'exemple suivant est en fait l'entête du fichier contenant cette page WEB.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
```

Il s'agit en fait d'une instruction de traitement particulière comme l'indique la syntaxe `<?xml ... ?>`. L'attribut `version` précise la version d'XML utilisée. Les valeurs possibles actuellement sont 1.0 ou 1.1. L'attribut `encoding` précise le codage des caractères utilisé dans le fichier. Les principales valeurs possibles sont US-ASCII, ISO-8859-1, UTF-8, et UTF-16. Ces noms de codage peuvent aussi être écrits en minuscule. L'attribut `standalone` précise si le fichier est autonome, c'est-à-dire s'il requiert ou non des ressources extérieures. La valeur de cet attribut peut être `yes` ou `no`.

#### 3.4.2 Codage des caractères

XML utilise la norme Unicode ISO-10646 aussi appelée UCS pour *Universal Character Set* pour coder les caractères. Unicode associe des codes 32 bits (4 octets) à tous les caractères des langues connues et bien d'autres symboles mathématiques ou autres. On appelle BMP pour *Basic Multilingual Plane* l'ensemble des caractères ayant un code sur deux octets c'est-à-dire entre 0 et 0xFFFF. L'ensemble de ces caractères suffit largement pour la très grande majorité des langues usuelles.

Pour éviter d'utiliser quatre octets pour chaque caractère, il existe différents codage des caractères.

- 5 -

Comme des espaces peuvent être présents dans la balise après le nom de l'élément, l'indentation est libre pour écrire les attributs d'une balise ouvrante. Il est ainsi possible d'écrire l'exemple générique suivant.

```
<tag attribut1="valeur1"
      attribut2="valeur2">
  ...
</tag>
```

L'ordre des attributs n'a pas d'importance. Voici encore d'autres exemples d'attributs.

```
<body background="yellow" ... </body>
<a href="#{node/@idref}">
<xsl:value-of select="key('idchapter', @idref)/title"/>
```

Un bon usage des attributs est pour les meta-données plutôt que les données elles-mêmes. Celles-ci doivent être placées de préférence dans le contenu des éléments. Dans l'exemple suivant, la date proprement dite est placée dans le contenu alors que l'attribut `format` précise son format. La norme ISO 8601 spécifie la représentation numérique de la date et de l'heure.

```
<date format="ISO-8601">2009-01-08</date>
```

### 3.5.4 Attributs particuliers

Il existe quatre attributs particuliers `xml:lang`, `xml:space`, `xml:base` et `xml:id` qui font partie de l'espace de noms XML.

L'attribut `xml:lang` est utilisé pour décrire la langue du contenu de l'élément. Sa valeur est un code de langue sur deux ou trois lettres de la norme ISO 639 (comme par exemple en, fr, es, de, it, pt, ...). Ce code peut être suivi d'un code de pays sur deux lettres de la norme ISO 3166. Cet attribut est hérité par les éléments contenus.

```
<p xml:lang="fr">Bonjour</p>
<p xml:lang="en-GB">Hello</p>
<p xml:lang="en-US">Hi</p>
```

L'attribut `xml:space` permet d'indiquer à une application le traitement des espaces. Les deux valeurs possibles de cet attribut sont `default` et `preserve`.

L'attribut `xml:base` permet un URL de base pour résoudre les URL relatifs dans le document.

L'attribut `xml:id` est de type `xsd:ID`. Il permet d'associer un identificateur à tout élément indépendamment de toute DTD ou de tout schéma.

### 3.5.5 Élément racine

Tout le corps du document doit être compris dans le contenu d'un unique élément appelé *élément racine*. Le nom de cet élément racine est donné par la déclaration de type de document. L'élément `bibliography` est l'élément racine de l'exemple donné au début du chapitre. L'exemple suivant est un document **non valide**.

- 8 -

lettres minuscules et majuscules sont distinctes. Les noms d'éléments commençant par les trois lettres `xml` en minuscule ou majuscule sont réservés. La norme XML prévoit que tout caractère UNICODE de catégorie lettre peut être utilisé comme lettre. Il est cependant conseillé de se limiter aux caractères de [A-Za-z] pour les noms d'éléments.

Il ne peut pas y avoir d'espace entre le caractère '`<`' et le nom de l'élément. Par contre, il peut y avoir des espaces, des tabulations et des retours à la ligne entre le nom de l'élément et le caractère '`>`'.

Les balises XML vont par paire balise ouvrante et balise fermante. Une *balise ouvrante* a la forme `<ident>` et la *balise fermante* correspondante a la forme `</ident>` où `ident` est le nom de l'élément.

Le *contenu* d'un élément est formé d'autres éléments et de texte. Le contenu peut éventuellement être vide. On obtient un arbre d'éléments ou les fils d'un élément sont les éléments qu'il contient

L'imbrication des balises doit être correcte. À toute occurrence d'une balise ouvrante doit correspondre une occurrence de la balise fermante de même nom. De plus, l'ordre des balises fermantes doit être l'ordre inverse des balises ouvrantes.

```
<extern-tag>
  <intern-tag></intern-tag>
</extern-tag>
```

### 3.5.2 Sections littérales

Il est souvent fastidieux d'inclure beaucoup de caractères spéciaux à l'aide des entités. Les sections CDATA permettent inclure toute une partie qui est recopiée verbatim. Une section CDATA a la forme suivante.

```
<![CDATA[ Contenu avec des caractères spéciaux <, > et & ]]>
```

Une section CDATA ne peut pas contenir la chaîne `]]` qui permet aux parseurs de détecter la fin de la section. Il est en particulier impossible d'imbriquer des sections CDATA.

### 3.5.3 Attributs

Les balises ouvrantes peuvent contenir des *attributs* associés à des valeurs. L'association de la valeur à l'attribut prend la forme `(name)="(value)"` ou la forme `(name)=(value)` où `name` et `value` sont respectivement le nom et la valeur de l'attribut. Chaque balise ouvrante peut contenir zéro ou plusieurs associations de valeurs à des attributs comme dans les exemples ci-dessous.

```
<tag attribut="valeur"></tag>
<tag attribut1="valeur1" attribut2="valeur2">...</tag>
```

Le nom de chaque attribut doit être un identifiant ne commençant par les trois lettres `xml` en minuscule ou en majuscule. La valeur d'un attribut doit être une chaîne quelconque délimitée par des apostrophes (caractère simple quote `'`) ou des guillemets (caractère double quotes `"`). Elle ne peut pas contenir les caractères spéciaux `<`, `>` et `&`. Ces caractères peuvent toutefois être introduits par des entités. Si la valeur de l'attribut est délimitée par des apostrophes `'`, elle peut contenir des guillemets `'` et inversement.

- 7 -

### 3.6.1 Exemple minimal

L'exemple suivant contient uniquement un prologue avec la déclaration XML et un élément de contenu vide. Les balises ouvrante `<tag>` et fermante `</tag>` ont été contractées en une seule balise `<tag/>`. Ce document n'a pas de déclaration de DTD.

```
<?xml version="1.0" ?>
<tag/>
```

L'exemple aurait pu être encore réduit en supprimant la déclaration XML mais celle-ci est fortement conseillée.

### 3.6.2 Exemple simple avec un commentaire mais sans DTD

La déclaration XML de l'exemple suivant s'est enrichie des attributs `encoding` et `standalone`. Le prologue contient également un commentaire. L'élément simple a un contenu textuel.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<!-- Document XML très simple avec juste un élément -->
<simple>Un exemple simplissime</simple>
```

### 3.6.3 Exemple simple avec une DTD incluse

Cet exemple contient une déclaration de DTD qui permet de valider le document. Cette DTD définit l'élément simple et déclare que son contenu doit être textuel.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<!DOCTYPE simple [
  <!ELEMENT simple (#PCDATA) >
]>
<simple>Un exemple simplissime</simple>
```

- 10 -

```
<?xml version="1.0" encoding="iso-8859-1" ?>
...
<root>
  ...
</root>
<root>
  ...
</root>
```

### 3.5.6 Commentaires

Les commentaires sont encadrés par `<!--` et `-->` comme en HTML. Ils ne peuvent pas contenir la chaîne `--` formés de deux tirets et ils ne peuvent donc pas être imbriqués. Ils sont insérés dans le prologue et en particulier dans la DTD. Ils peuvent aussi apparaître dans le contenu de n'importe quel élément et après l'élément racine. Un exemple de commentaire est donné ci-dessous.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!-- Commentaire dans le prologue avant la DTD -->
<!DOCTYPE simple [
  <!-- Commentaire dans la DTD -->
  <!ELEMENT simple (#PCDATA) >
]>
<!-- Commentaire entre le prologue et le corps -->
<simple>
  <!-- Commentaire au début du contenu de l'élément simple -->
  Un exemple simplissime
  <!-- Commentaire à la fin du contenu de l'élément simple -->
</simple>
<!-- Commentaire après le corps -->
```

### 3.5.7 Instructions de traitement

Les instructions de traitement sont destinées aux programmes qui traitent le document XML. Elles peuvent apparaître aux mêmes endroits que les commentaires. Elles sont encadrées par `<? et ?>`. Le nom de l'instruction est immédiatement après les deux caractères `<?`. Ce nom est ensuite suivi de paramètres dont la syntaxe est semblable à celle des attributs. Chaque paramètre prend la forme `(param)=(value)` où `param` est son nom et `value` sa valeur. L'instruction de traitement donnée en exemple ci-dessous déclare une feuille de style XSL que le navigateur doit appliquer au document pour l'afficher.

```
<?xml-stylesheet href="list.xml" type="text/xsl" title="En liste"?>
```

La déclaration XML est en fait une instruction de traitement particulière de nom `xml` avec des paramètres `version`, `encoding` et `standalone`.

### 3.6 Exemples

Voci quelques exemples minimalistes de documents XML.

- 9 -

Chaque déclaration commençant par `<!ELEMENT` déclare un nom d'élément. La déclaration `<!ELEMENT bibliography (book)+ >` indique par exemple que le contenu de l'élément `bibliography` est une suite non vide d'éléments `book`. Chaque déclaration commençant par `<!ATTLIST` déclare un attribut pour un élément. La ligne `<!ATTLIST book key NMTOKEN #REQUIRED >` déclare l'attribut `key` de l'élément `book`. Elle donne son type `NMTOKEN` et précise que cet attribut est obligatoire.

## 4.2 Déclaration de la DTD

La déclaration de la DTD du document doit être placée dans le prologue. La DTD peut être interne, externe ou mixte. Elle est *interne* si elle est directement incluse dans le document. Elle est *externe* si le document contient seulement une référence à un autre document contenant la DTD. Elle est finalement mixte si elle est constituée d'une partie interne et d'une partie externe.

Une DTD est généralement prévue pour être utilisée pour de multiples documents. Elle est alors utilisée comme DTD externe. En revanche, il est pratique d'inclure directement la DTD dans le document en phase de développement.

### 4.2.1 DTD interne

Lorsque la DTD est incluse dans le document, sa déclaration prend la forme suivante.

```
<!DOCTYPE (element) [ (declarations) ] >
```

L'identifiant `element` est le nom de l'élément qui contient tout le contenu du document. Les déclarations `declarations` constituent la définition du type du document. Voici un exemple très simple de déclaration de type.

```
<!DOCTYPE simple [
<!ELEMENT simple (#PCDATA) >
]>
```

Dans cet exemple, le nom de l'élément racine est `simple`. La DTD déclare en outre que cet élément ne peut contenir que du texte (*Parsed Characters DATA*) et pas d'autre élément.

### 4.2.2 DTD externe

Une DTD externe peut être référencée par une URL ou un FPI (*Formal Public Identifier*).

#### 4.2.2.1 Par URL

La référence à une URL est introduite par le mot clé `SYSTEM`.

```
<!DOCTYPE (element) SYSTEM "(url)" >
```

L'url peut être une URL complète ou plus simplement le nom d'un fichier comme dans les exemples suivants.

```
<!DOCTYPE bibliography SYSTEM
"http://www.liafa.jussieu.fr/~carton/Enseignement/XML/Cours/DTD/bibliography.dtd" >
<!DOCTYPE bibliography SYSTEM "bibliography.dtd" >
```

## 4 DTD

Le rôle d'une DTD (*Document Type Definition*) est de définir précisément la structure d'un document. Il s'agit d'un certain nombre de contraintes que doit respecter un document pour être *valide*. Ces contraintes spécifient quelles sont les éléments qui peuvent apparaître dans le contenu d'un élément, l'ordre éventuel de ces éléments et la présence de texte brut. Elles définissent aussi, pour chaque élément les attributs autorisés et les attributs obligatoires.

Les DTD ont l'avantage d'être relativement simples à utiliser mais elles sont parfois aussi un peu limitées. Les schémas XML permettent de décrire de façon plus précise encore la structure d'un document. Ils sont plus sophistiqués mais plus difficiles à manipuler.

## 4.1 Un premier exemple

On commence par un exemple de petit document XML contenant sa DTD.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- Document XML avec une DTD interne -->
<!DOCTYPE bibliography [
<!-- Début de la DTD -->
<!ELEMENT bibliography (book)+ >
<!ELEMENT book (title, author, year, publisher, isbn, url?) >
<!ATTLIST book key NMTOKEN #REQUIRED >
<!ATTLIST book lang (fr | en) #REQUIRED >
<!-- Éléments -->
<!-- titre -->
<!-- auteur -->
<!-- année -->
<!-- éditeur -->
<!-- isbn -->
<!-- url -->
<!-- Fin de la DTD -->
]>
<bibliography>
<book key="Michard01" lang="fr">
<title>XML langage et applications</title>
<author>Alain Michard</author>
<year>2001</year>
<publisher>Eyrolles</publisher>
<isbn>2-212-09206-7</isbn>
<url>http://www.editions-eyrolles/livres/michard/</url>
</book>
<book key="Marchal00" lang="en">
<title>XML by Example</title>
<author>Benoît Marchal</author>
<year>2000</year>
<publisher>Macmillan Computer Publishing</publisher>
<isbn>0-7897-2242-9</isbn>
</book>
</bibliography>
```

La première ligne du document XML donne la version de XML et le codage du document. Les lignes suivantes constituent la DTD. La première ligne indique en particulier que l'élément racine du document est `bibliography`.

Les nombreuses entités prédéfinies en XHTML comme `&euro;` pour le symbole € n'existent pas en XML.

### 4.3.1.2 Entité interne

Une entité est dite *interne* lorsque le fragment est inclus directement dans le document. La déclaration d'une telle entité prend la forme suivante.

```
<!ENTITY (entity) "(fragment)" >
```

Si la DTD contient par exemple la déclaration d'entité suivante.

```
<!ENTITY aka "also known as" >
<!ENTITY euro "&#20AC;" >
```

il est possible d'inclure le texte `also known as` en écrivant seulement `&aka;`.

Il est possible d'utiliser des entités dans la définition d'une autre entité pourvu que ces entités soient également définies. L'ordre de ces définitions est sans importance car les substitutions sont réalisées au moment où le document est lu par le parseur. Les définitions récursives sont bien sûr interdites.

```
<!DOCTYPE book [
<!-- Entités -->
<!-- Entité jmh "James Marshall Hendrix" -->
<!-- Entité aka "also known as" -->
]>
<book>&jmh;</book>
```

### 4.3.1.3 Entité externe

Une entité peut désigner une fraction de document contenu dans un autre fichier. Ce mécanisme permet de répartir un même document sur plusieurs fichiers comme dans l'exemple suivant.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE book [
<!-- Entités externes -->
<!-- Entité chapter1 -->
<!-- Entité chapter2 -->
]>
<book>
<chapter1>
<chapter2>
</book>
```

### 4.3.1.4 Entité paramètre

Les *entités paramètres* sont des entités qui peuvent uniquement être utilisées à l'intérieur de la DTD. Elles permettent d'écrire des DTD modulaires en déclarant des groupes d'éléments ou d'attributs utilisés plusieurs fois. La déclaration d'une entité paramètre prend la forme suivante.

```
<!ENTITY % (entity) "(fragment)" >
```

L'entité `entity` ainsi déclarée peut être utilisée en écrivant `%entity;`. Une entité paramètre peut uniquement être utilisée dans la partie externe de la DTD. L'exemple suivant définit deux entités paramètre `idatt` et `langatt` permettant de déclarer des attributs `id` et `xml:lang` facilement.

### 4.2.2.2 Par FPI

La référence à un FPI est introduite par le mot clé `PUBLIC`. Le FPI est suivi d'une URL dans le cas où le FPI ne permet pas à l'application de retrouver la DTD.

```
<!DOCTYPE (element) PUBLIC "(fpi)" "(url)" >
```

L'exemple suivant est la déclaration de cette page qui utilise une des DTDs de XHTML.

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

### 4.2.3 DTD mixte

Il est possible d'avoir une DTD externe adressée par URL ou FPI et des déclarations internes. Les déclarations internes peuvent alors redéfinir des éléments ou des attributs déjà définis dans la DTD externe. La déclaration prend alors une des deux formes suivantes.

```
<!DOCTYPE (element) SYSTEM "(url)" [ (declarations) ] >
<!DOCTYPE (element) PUBLIC "(fpi)" "(url)" [ (declarations) ] >
```

Il est possible de redéfinir dans la partie interne un élément ou une entité déjà définie dans la partie externe. La définition interne a alors priorité sur la définition externe. Ce mécanisme permet d'utiliser une DTD externe tout en adaptant certaines définitions au document.

## 4.3 Contenu de la DTD

Une DTD est essentiellement constituée de déclarations d'éléments et d'attributs. Elle peut aussi contenir des déclarations d'entités qui sont des *macros* semblables aux `#define` du langage C.

### 4.3.1 Déclaration d'entité

Une *entité* est un nom donné à un fragment de document. Ce fragment peut être inséré dans le document en utilisant simplement le nom de l'entité. Il s'agit en fait d'un mécanisme d'abréviation. Si l'entité a pour nom `ident`, le fragment est inséré par `&ident;`.

#### 4.3.1.1 Entités prédéfinies

Il existe des entités prédéfinies permettant d'inclure les caractères spéciaux `<`, `>`, `&`, `"` et `'` dans les contenus d'éléments et dans les valeurs d'attributs. Ces entités sont les suivantes.

Entité	Caractère
<code>&amp;lt;</code>	<code>&lt;</code>
<code>&amp;gt;</code>	<code>&gt;</code>
<code>&amp;amp;</code>	<code>&amp;</code>
<code>&amp;quot;</code>	<code>"</code>
<code>&amp;apos;</code>	<code>'</code>

```
<!ELEMENT elem (elem1, (elem2 | elem4), elem3) >
```

L'élément `elem` doit contenir un élément `elem1`, un élément `elem2` ou un élément `elem4` puis un élément `elem3` dans cet ordre.

```
<!ELEMENT elem (elem1, elem2, elem3)* >
```

L'élément `elem` doit contenir une suite d'éléments `elem1`, `elem2`, `elem3`, `elem1`, `elem2`, ... jusqu'à un élément `elem3`.

```
<!ELEMENT elem (elem1 | elem2 | elem3)* >
```

L'élément `elem` doit contenir une suite quelconque d'éléments `elem1`, `elem2` ou `elem3`.

```
<!ELEMENT elem (elem1 | elem2 | elem3)+ >
```

L'élément `elem` doit contenir une suite non vide d'éléments `elem1`, `elem2` ou `elem3`.

#### 4.3.2.2 Contenu textuel

La déclaration de la forme suivante indique qu'un élément peut uniquement contenir du texte. Ce texte est formé de caractères, d'entités qui seront remplacées au moment du traitement et de sections littérales CDATA.

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

Dans l'exemple suivant, l'élément `text` est de type `#PCDATA`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE texts [
  <!ELEMENT texts (text)* >
  <!ELEMENT text (#PCDATA) >
]>
<text>
  <text>Du texte simple</text>
  <text>Une <![CDATA[ Section CDATA avec < et > ]]></text>
  <text>Des entités &lt; et &gt;</text>
</texts>
```

#### 4.3.2.3 Contenu mixte

La déclaration de la forme suivante indique qu'un élément peut uniquement contenir du texte et les éléments `elem1`, ..., `elemN`. C'est la seule façon d'avoir un contenu mixte avec du texte et des éléments. Il n'y a aucun contrôle sur le nombre d'occurrences de chacun des éléments et sur leur ordre d'apparition dans le contenu de l'élément ainsi déclaré. Dans une telle déclaration `#PCDATA` doit apparaître avant les noms des éléments.

```
<!ELEMENT (ident) (#PCDATA, (elem1) | ... | (elemN))* >
```

Dans l'exemple suivant, l'élément `book` est de type mixte.

```
<?xml version="1.0" encoding="us-ascii" ?>
<!DOCTYPE book [
  <!ELEMENT book (#PCDATA | em | cite)* >
  <!ELEMENT em (#PCDATA) >
  <!ELEMENT cite (#PCDATA) >
]>
<book>
  Du <em>text</em>, une <cite>citation</cite> et encore du <em>text</em>.
</book>
```

```
<!ATTLIST (element) (attribut1) (type1) (default1)
  (attribut2) (type2) (default2)
  ...
  (attributN) (typeN) (defaultN) >
```

#### 4.3.3.1 Types des attributs

Le type d'un attribut détermine quelles sont ses valeurs possibles. Les DTDs proposent uniquement un choix fini de types pour les attributs. Le type `dpot` en effet être pris dans la liste suivante. Le type le plus général est `CDATA` qui autorise tous les valeurs correctes d'un point de vue syntaxique. Les types `NMTOKEN` et `NMTOKENS` imposent respectivement que la valeur de l'attribut soit un identificateur ou une suite d'identificateurs séparés par des espaces.

CDATA

La valeur de l'attribut est une chaîne quelconque.

Liste énumérative (`val1 | val2 | ... | valN`)

La valeur de l'attribut doit être une des valeurs énumérées.

NMTOKEN

La valeur de l'attribut est un identificateur

NMTOKENS

La valeur de l'attribut est une liste d'identificateurs séparés par des espaces.

ID

La valeur de l'attribut est un identificateur unique. Un élément peut avoir un seul attribut de ce type.

IDREF

La valeur de l'attribut est une référence à un élément identifié par la valeur de son attribut de type ID.

IDREFS

La valeur de l'attribut une liste de références séparés par des espaces.

NOTATION

La valeur de l'attribut est une notation

ENTITY

La valeur de l'attribut une entité externe non XML

ENTITIES

La valeur de l'attribut une liste d'entités externes non XML

Les types `ID`, `IDREF` et `IDREFS` s'utilisent conjointement. La valeur d'un attribut de type `ID` doit être un identificateur comme pour le type `NMTOKEN`. Cette identificateur doit en outre identifier de manière unique l'élément qui le contient. Ceci signifie qu'aucun attribut d'un autre élément du document a un attribut de type `ID` avec la même valeur. Chaque élément peut avoir au plus un attribut de type `ID`. La valeur de cet attribut joue le rôle de *clé* des bases de données. Cette clé permet de désigner de manière unique un élément dans le document. Ce mécanisme est par exemple utilisé par CSS pour attacher une règle à un élément précis d'une page XHTML.

La valeur d'un attribut de type `IDREF` doit aussi être un identificateur. Celui-ci doit être la valeur d'un attribut de type d'un (autre) élément du document. Un attribut de type `IDREF` est donc utilisé pour référencer un élément du document. Un attribut de type `IDREFS` doit contenir une liste d'identificateurs séparés par des espaces. Chacun des identificateurs doit être la valeur d'un attribut de type `ID` d'un (autre) élément.

```
<!ENTITY % idatt "id ID #REQUIRED" >
<!ENTITY % langatt "xml:lang NMTOKEN 'fr';" >
```

```
<!ATTLIST chapter %idatt; %langatt; >
<!ATTLIST section %langatt; >
```

### 4.3.2 Déclaration d'élément

La déclaration d'un élément est nécessaire pour qu'il puisse apparaître dans un document. Cette déclaration précise le nom et le type de l'élément. Le type détermine les contenus valides de l'élément. On distingue les *contenus purs* uniquement constitués d'autres éléments, les *contenus textuels* uniquement constitués de texte et les *contenus mixtes* qui mélangent éléments et texte.

De manière générale, la déclaration d'un élément prend la forme suivante où `ident` et `type` sont respectivement le nom et le type de l'élément.

```
<!ELEMENT (ident) (type) >
```

#### 4.3.2.1 Contenu pur d'éléments

Une déclaration d'élément a la forme suivante.

```
<!ELEMENT (ident) (regex) >
```

Le nom de l'élément est donné par l'identifiant `ident` et l'expression rationnelle `regex` décrit les suites d'éléments autorisées dans le contenu de l'élément. Cette expression rationnelle est construite à partir des noms d'éléments en utilisant les opérateurs `,`, `|`, `?`, `*`, `+` et les parenthèses `( )` pour former des groupes. La signification des cinq opérateurs est donnée par la table suivante.

Opérateur	Signification
,	Mise en séquence
	Choix
?	0 ou 1 occurrence
*	Itération (nombre quelconque d'occurrences)
+	Itération stricte (nombre non nul d'occurrences)

Cette définition est illustrée par les exemples suivants.

```
<!ELEMENT elem (elem1, elem2, elem3) >
```

L'élément `elem` doit contenir un élément `elem1`, un élément `elem2` puis un élément `elem3` dans cet ordre.

```
<!ELEMENT elem (elem1 | elem2 | elem3) >
```

L'élément `elem` doit contenir un seul des éléments `elem1`, `elem2` ou `elem3`.

```
<!ELEMENT elem (elem1, elem2?, elem3) >
```

L'élément `elem` doit contenir un élément `elem1`, un ou zéro élément `elem2` puis un élément `elem3` dans cet ordre.

```
<!ELEMENT elem (elem1, elem2*, elem3) >
```

L'élément `elem` doit contenir un élément `elem1`, une suite éventuellement vide d'éléments `elem2` et un élément `elem3` dans cet ordre.

#### 4.3.2.4 Contenu vide

La déclaration suivante indique que le contenu de l'élément `ident` est nécessairement vide. Cet élément peut uniquement avoir des attributs. Les éléments vides sont particulièrement utiles pour des renvois à d'autres parties du document.

```
<!ELEMENT (ident) EMPTY >
```

Lorsqu'un élément est déclaré vide, les balises ouvrante `<ident>` et fermante `</ident>` peuvent être contractées en une seule balise `<ident/>`. Cette contraction n'est pas obligatoire mais conseillée. Cette unique balise contient alors les attributs comme dans l'exemple suivant.

```
<hr style="color:red; height:15px; width:350px;" />
```

Un élément non déclaré vide peut avoir un contenu vide si sa déclaration le permet. En revanche, il doit avoir une balise ouvrante et une balise fermante sans contraction. Dans l'exemple suivant, l'élément `sec` a un contenu textuel (de type `#PCDATA`) qui peut être vide. En revanche, l'élément `ref` est déclaré vide (de type `EMPTY`). L'élément `sec` n'est pas contracté s'il est vide alors que l'élément `ref` peut être contracté.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE book [
  <!ELEMENT book (sec | ref)* >
  <!ELEMENT sec (#PCDATA) >
  <!ATTLIST sec id ID #IMPLIED >
  <!ELEMENT ref EMPTY >
  <!ATTLIST ref idref IDREF #REQUIRED >
]>
<book>
  <sec id="sec-0">Du texte dans une section</sec>
  <!-- élément déclaré vide -->
  <ref idref="sec-0" />
  <!-- élément non déclaré vide mais pouvant être vide -->
  <sec id="sec-1"></sec>
</book>
```

#### 4.3.2.5 Contenu libre

La déclaration suivante n'impose aucune contrainte sur le contenu de l'élément `ident`. Par contre, ce contenu doit bien entendu être bien formé et les éléments contenus doivent également être déclarés. Ce type de déclarations permet de déclarer des éléments dans une DTD en cours de mise au point.

```
<!ELEMENT (ident) ANY >
```

### 4.3.3 Déclaration d'attribut

Une déclaration d'élément a la forme suivante.

```
<!ATTLIST (element) (attribut) (type) (default) >
```

Elle déclare un attribut de nom `attribut` pour l'élément `element`. Il est possible de déclarer simultanément plusieurs attributs pour un même élément. Cette déclaration prend alors la forme suivante où l'indentation est bien sûr facultative.

```

<!ATTLIST entry ref IDREF #IMPLIED >
  <entry id="id-437" ref="id-234">
<!ATTLIST entry refs IDREFS #IMPLIED >
  <entry id="id-561" refs="id-234 id-437">
<!ATTLIST rule style CDATA #FIXED "free" >
  <rule>Valeur par défaut</rule>
  <rule style="free">
  <rule style="libre">Document non valide</rule>

```

#### 4.4 Outils de validations

- Page de validation du W3C
- Page de validation du Scholarly Technology Group de l'université de Brown

Il est bien sûr possible de déclarer plusieurs espace de noms en utilisant plusieurs attributs de la forme `xmlns:(prefix)`. Dans l'exemple suivant, on déclare également l'espace de noms de MathML et on l'associe au préfixe `math`.

```

<html:html xmlns:html="http://www.w3.org/1999/xhtml"
  xmlns:math="http://www.w3.org/1998/Math/MathML" >
<html:head>
<html:title>
  Espaces de noms
</html:title>
</html:head>
<html:body>
  ...
  <math:math>
  <math:apply>
  <math:eq/>
  ...
  </math:apply>
</math:math>
  ...
</html:body>
</html:html>

```

C'est l'URI associé au préfixe qui détermine l'espace de noms. Le préfixe est juste une abréviation pour l'URI. Deux préfixes associés au même URI déterminent le même espace de noms. Dans l'exemple suivant, les deux éléments `firstname` et `lastname` font partie du même espace de noms.

```

<name xmlns:foo="http://www.somewhere.org/uri"
  xmlns:bar="http://www.somewhere.org/uri" >
  <!-- Les éléments firstname et lastname appartiennent au même espace de noms. -->
  <foo:firstname>Gaston<foo:firstname>
  <bar:lastname>Lagaffe<bar:lastname>
</name>

```

#### 5.3 Portée d'une déclaration

La portée d'une déclaration d'un espace de noms est l'élément dans laquelle elle est faite. L'exemple précédent aurait pu aussi être écrit de la manière suivante.

```

<html:html xmlns:html="http://www.w3.org/1999/xhtml" >
<html:head>
<html:title>
  Espaces de noms
</html:title>
</html:head>
<html:body>
  ...
  <math:math xmlns:math="http://www.w3.org/1998/Math/MathML" >
  <math:apply>
  <math:eq/>
  ...
  </math:apply>
</math:math>
  ...
</html:body>
</html:html>

```

## 5 Espaces de noms

Les espaces de noms permettent d'utiliser simultanément des éléments de même nom mais définis dans des modèles différents.

### 5.1 Identification d'un espace de noms

Un espace de noms est identifié par une URL appelée *URL de l'espace de noms*. Il est sans importance que l'URL pointe réellement sur un document. Cette URL garantit seulement que l'espace de noms est identifié de manière unique. Dans la pratique, l'URL permet aussi souvent d'accéder à un document qui décrit l'espace de noms. Voici les URLs de quelques espaces de noms.

```

XML
  http://www.w3.org/XML/1998/namespace
MathML
  http://www.w3.org/1998/Math/MathML
XHTML
  http://www.w3.org/1999/xhtml
SVG
  http://www.w3.org/2000/svg
Schémas
  http://www.w3.org/2001/XMLSchema

```

### 5.2 Déclaration d'un espace de noms

Un espace de noms déclaré par un attribut de forme `xmlns:(prefix)` dont la valeur est un URI ou URL qui identifie l'espace de nom. Le préfixe `prefix` peut être quelconque mais il ne doit pas commencer par les trois lettres `xml`. Il est ensuite utilisé pour *qualifier* les noms d'éléments.

Un *nom qualifié* d'élément a la forme `(prefix):(local)` où `prefix` est un préfixe associé à un espace de noms et `local` est le *nom local* de l'élément.

Dans l'exemple suivant, on associe le préfixe `html` à l'espace de noms de XHTML identifié par l'URL `http://www.w3.org/1999/xhtml`. Ensuite, tous les éléments de cet espace sont préfixé par `html:`.

```

<html:html xmlns:html="http://www.w3.org/1999/xhtml" >
<html:head>
  <html:title>
  Espaces de noms
  </html:title>
</html:head>
<html:body>
  ...
</html:body>
</html:html>

```

Le choix du préfixe est complètement arbitraire. Dans l'exemple précédent, on aurait pu utiliser `foo` ou `bar` à la place du préfixe `html`. Il faut par contre être cohérent entre la déclaration du préfixe et son utilisation. Même si les préfixes peuvent être librement choisis, il est d'usage d'utiliser certains préfixes pour certains espaces de noms. Ainsi, on prend souvent `html` pour XHTML, `xs:d` ou `xs:s` pour les schémas XML et `xs:l` pour les feuilles de style XSL.

```

<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<!DOCTYPE book [
  <!ELEMENT book (section)* >
  <!ELEMENT section (#PCDATA | ref | refs)* >
  <!ATTLIST section id ID #IMPLIED>
  <!ELEMENT ref EMPTY >
  <!ATTLIST ref idref IDREF #REQUIRED >
  <!ELEMENT refs EMPTY >
  <!ATTLIST refs idrefs IDREFS #REQUIRED >
]>
<book>
  <section id="sec0">Du texte et une référence <ref idrefs="sec1"/></section>
  <section id="sec1">Des références <refs idrefs="sec0 sec2"/></section>
  <section id="sec2">Section sans référence</section>
  <section id="sec3">Une auto-référence <refs idrefs="sec3"/></section>
</book>

```

#### 4.3.3.2 Valeur par défaut

La valeur par défaut peut prendre une des valeurs suivantes.

Une valeur par défaut "val"

Si l'attribut est absent pour un élément du document, sa valeur est implicitement la valeur `val`.

#IMPLIED

L'attribut est *optionnel*. Il n'y a pas de valeur par défaut. Si l'attribut est absent, il n'a pas de valeur.

#REQUIRED

L'attribut est *obligatoire*. Il n'y a pas de valeur par défaut.

#FIXED "val"

La valeur de l'attribut est fixée à la valeur `val`. Si l'attribut est absent, sa valeur est implicitement `val`. Si l'attribut est présent, sa valeur doit être `val`. Sinon, le document n'est pas valide.

#### 4.3.3.3 Exemples

Voici quelques exemples de déclarations d'attributs avec pour chaque déclaration des valeurs possibles pour l'attribut.

```

<!ATTLIST tag style CDATA "Chaine" >
  <tag>Valeur par défaut</tag>
  <tag style="color:red; height:15px; width:350px;">
  <tag style="Entités &lt;&amp;&gt;">
  <tag style="Caractère accentués">
<!ATTLIST book lang (fr | en) "fr" >
  <book>Valeur par défaut</book>
  <book lang="fr">
  <book lang="en">
<!ATTLIST book lang NMTOKEN #IMPLIED >
  <book>Pas de valeur</book>
  <book id="en">
  <book id="ascii">
<!ATTLIST entry id ID #REQUIRED >
  <entry>Document non valide</entry>
  <entry id="id-234">

```

```

...
<name xmlns="" >
<!-- L'espace de noms par défaut n'est plus spécifié -->
<!-- Les éléments name, firstname et lastname n'appartiennent à aucun espace de noms. -->
<firstname>Gaston<firstname>
<lastname>Lagaffe<lastname>
</name>
...
</body>
</html>

```

## 5.5 Attributs

Les attributs peuvent également avoir des noms qualifiés. Ils font alors partie de l'espace de noms donné par le préfixe comme dans l'exemple suivant. L'attribut `schemaLocation` fait partie de l'espace de noms des instances de schémas identifié par l'URL `http://www.w3.org/2001/XMLSchema-instance`.

```

<?xml version="1.0" encoding="iso-8859-1" ?>
<bibliography
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="bibliography.xsd">
...

```

En revanche, les attributs dont le nom n'est pas qualifié ne font jamais partie de l'espace de noms par défaut que celui-ci soit spécifié ou non.

## 5.6 Espace de noms xml

Le préfixe `xml` est implicitement lié à l'espace de noms `http://www.w3.org/XML/1998/namespace` et il n'a pas besoin d'être déclaré. Les attributs `xml:lang`, `xml:space`, `xml:base` et `xml:id` font partie de cet espace de noms.

```

<xsd:attribute name="key" type="xsd:NMTOKEN" use="required"/>
<xsd:attribute name="lang" type="xsd:NMTOKEN" use="required"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

Ce schéma déclare l'élément `bibliography` du type `Bibliography` qui est ensuite introduit par `<xsd:complexType>`. Ce type est alors défini comme une suite d'autres éléments introduite par le constructeur `<xsd:sequence>`. Les deux attributs `key` et `lang` de l'élément `book` sont introduits par les déclarations `<xsd:attribute ... />`.

## 6.3 Structure globale d'un schéma

Un schéma XML se compose essentiellement de déclarations d'éléments et de types. Chaque élément est déclaré avec un type qui peut être soit un des types prédéfinis, soit un nouveau type construit explicitement ou obtenu par modification d'un type prédéfini.

L'espace de noms des schémas XML est `http://www.w3.org/2001/XMLSchema`. Il est généralement associé, comme dans l'exemple précédent au préfixe `xsd` ou à `xs`. Tout le schéma est inclus dans l'élément `schema`. Un schéma contient principalement des déclarations d'éléments, d'attributs et de types. Les déclarations d'éléments et d'attributs utilisent un type prédéfini ou un type défini dans le schéma. Un schéma peut aussi contenir des imports d'autres schémas, des définitions de groupes d'éléments et d'attributs et des contraintes de cohérences. La structure globale d'un schéma est donc la suivante.

```

<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Déclarations d'éléments, d'attributs et de types -->
  ...
</xsd:schema>

```

Les déclarations d'éléments et de types peuvent être globales ou locales. Une déclaration est globale lorsque elle est fille directe de l'élément `schema`. L'élément ou le type définis peuvent alors être nommés pour être utilisés par référence dans d'autres déclarations.

Les éléments sont déclarés par l'élément `xsd:element`, les attributs par l'élément `xsd:attribute` et les types par les éléments `xsd:simpleType` ou `xsd:complexType`.

### 6.3.1 Attributs de l'élément schema

L'élément racine `schema` peut avoir les attributs suivants.

`targetNamespace`

La valeur de cet attribut est l'URI qui identifie l'espace de noms cible, c'est-à-dire l'espace de noms des éléments et types définis par le schéma. Si cet attribut est absent, les éléments et types définis n'ont pas d'espace de noms.

`elementFormDefault` et `attributeFormDefault`

Ces deux attributs donnent la valeur par défaut de l'attribut `form` pour respectivement les éléments et les attribut. Les valeurs possible sont `qualified` et `unqualified`. La valeur par défaut est `unqualified`.

## 5.4 Espace de noms par défaut

Les éléments dont le nom n'est pas qualifié font partie de l'espace de noms par défaut. Celui-ci peut être spécifié par un attribut de nom `xmlns` dont la valeur est l'URL de l'espace de noms. L'exemple précédent aurait pu être simplifié de la façon suivante.

```

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>
    Espaces de noms
  </title>
</head>
<body>
  ...
  <math xmlns:math="http://www.w3.org/1998/Math/MathML" >
    <math:apply>
      <math:eq/>
    </math:apply>
  </math>
  ...
</body>
</html>

```

Comme la déclaration de l'espace de noms est locale à l'élément, l'exemple précédent aurait pu être écrit de façon encore plus simplifiée en changeant localement (dans l'élément `math`) l'espace de noms par défaut.

```

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>
    Espaces de noms
  </title>
</head>
<body>
  ...
  <math xmlns="http://www.w3.org/1998/Math/MathML" >
    <apply>
      <eq/>
    </apply>
  </math>
  ...
</body>
</html>

```

Tant que l'espace de noms par défaut n'a pas été spécifié, les éléments dont le nom n'est pas qualifié ne font partie d'aucun espace de noms. Leur propriété *espace de noms* n'a pas de valeur. Il est possible de revenir à l'espace de noms par défaut non spécifié en affectant la chaîne vide à l'attribut `xmlns` comme dans l'exemple suivant.

```

<html xmlns="http://www.w3.org/1999/xhtml" >
<!-- L'espace de noms par défaut est spécifié -->
<!-- Les éléments html, head, title, body, ... appartiennent à l'espace de noms par défaut. -->
<head>
  <title>
    Espaces de noms
  </title>
</head>
<body>

```

## 6 Schémas XML

### 6.1 Introduction

Les *schémas XML* permettent comme les DTD de définir des modèles de documents. Il est ensuite possible de vérifier qu'un document donné respecte un schéma. Les schémas ont été introduits pour combler les lacunes des DTD.

#### 6.1.1 Inconvénients des DTD

- Syntaxe non XML
- Manque de concision dans les descriptions des contenus
- Modularité très limitée
- Pas de gestion des espaces de noms

#### 6.1.2 Apports des schémas XML

- Syntaxe XML
- Nombreux types de données prédéfinis (nombres, dates, ...)
- Possibilité de définir de nouveaux types
- Approche objet pour une hiérarchie de types
- Modularité accrue
- Prise en compte des espaces de noms

### 6.2 Un premier exemple

Voici un exemple de schéma XML définissant le type de document de la bibliographie. Ce schéma est volontairement rudimentaire pour un premier exemple. Il n'est pas très précis sur les contenus de certains éléments. Un schéma plus complet peut être donné pour la bibliographie. Le cours est essentiellement basé sur des exemples.

```

<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="fr">
      Schéma XML pour bibliographie.xml
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="bibliography" type="Bibliography"/>
  <xsd:complexType name="Bibliography">
    <xsd:sequence>
      <xsd:element name="book" minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="author" type="xsd:string"/>
            <xsd:element name="year" type="xsd:string"/>
            <xsd:element name="publisher" type="xsd:string"/>
            <xsd:element name="isbn" type="xsd:string"/>
            <xsd:element name="url" type="xsd:string" minOccurs="0"/>
          </xsd:sequence>

```

```
<xsd:element name="(elem)" type="(type)"/>
```

où `elem` et `type` sont respectivement le nom et le type de l'élément. Ce type peut être un des types prédéfinis comme `xsd:string` ou `xsd:integer` ou encore un type défini dans le schéma. L'exemple suivant déclare l'élément `title` de type `xsd:string`. Le nom du type doit être un nom qualifié comme ici par le préfixe `xsd:` associé à l'espace de noms des schémas. Cette règle s'applique aussi lorsque le type est défini dans le schéma.

```
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="title" type="ns:Title"/>
```

## 6.5.2 Valeur par défaut et valeur fixe

Lorsque le type est simple, il est possible de donner une valeur par défaut ou une valeur fixe à l'élément comme dans les deux exemples suivants. Il faut pour donner des valeurs aux attributs `default` ou `fixed` de l'élément `xsd:element`.

```
<xsd:element name="title" type="xsd:string" default="Un titre par défaut"/>
<xsd:element name="title" type="xsd:string" fixed="Un titre fixe"/>
```

## 6.5.3 Type anonyme

Lors de la déclaration d'un élément, il est possible de décrire explicitement le type. La déclaration du type est alors le contenu de l'élément `xsd:element`. Le type est alors local et sa déclaration prend alors une des deux formes suivantes où `elem` est bien sûr le nom de l'élément déclaré.

```
<xsd:element name="(elem)"
  <xsd:simpleType>
  ...
</xsd:simpleType>
</xsd:element>
```

```
<xsd:element name="(elem)"
  <xsd:complexType>
  ...
</xsd:complexType>
</xsd:element>
```

## 6.5.4 Référence à un élément global

Dans la définition d'un élément ou d'un type, il est possible d'utiliser un élément défini globalement, c'est-à-dire comme fils de l'élément `schema`. Une telle référence s'écrit de la façon suivante.

```
<!-- Définition globale de l'élément title -->
<xsd:element name="title" type="Title"/>
...
<!-- Définition d'un type -->
<xsd:complexType ... >
  ...
  <!-- Utilisation de l'élément title -->
  <xsd:element ref="title"/>
  ...
</xsd:complexType>
```

- 28 -

byte

Entier signé sur 8 bits

unsignedByte

Entier non signé sur 8 bits

integer

Entier

positiveInteger

Entier strictement positif

negativeInteger

Entier strictement négatif

nonPositiveInteger

Entier négatif ou nul

nonNegativeInteger

Entier positif ou nul

int

Entier non signé sur 32 bits

unsignedInt

Entier non signé sur 32 bits

long

Entier signé sur 64 bits

unsignedLong

Entier non signé sur 64 bits

short

Entier signé sur 16 bits

unsignedShort

Entier non signé sur 16 bits

decimal

Nombre décimal

time

Heure au format hh:mm:ss, par exemple 14:07:23. Tous les champs sont obligatoires.

date

Date au format YYYY-MM-DD, par exemple 2008-01-16. Tous les champs sont obligatoires.

dateTime

Date et heure au format YYYY-MM-DDThh:mm:ss, par exemple 2008-01-16T14:07:23.

Tous les champs sont obligatoires.

duration

Durée au format PnYnMnDnTnHnMnS

## 6.7 Déclarations de types

Parmi les types, les schémas XML distinguent les *types simples* introduits par le constructeur `xsd:simpleType` et les *types complexes* introduits par le constructeur `xsd:complexType`. Les types simples décrivent des contenus purement textuels. Ils peuvent être utilisés pour les éléments comme pour les attributs. Ils sont généralement obtenus par dérivation des types prédéfinis. Au contraire les types complexes décrivent des contenus formés uniquement d'éléments ou des contenus mixtes. Ils peuvent uniquement être utilisés pour déclarer des éléments. Seuls les types complexes peuvent définir des attributs.

`blockDefault` et `finalDefault`

Ces deux attributs donnent la valeur par défaut des attributs `block` et `final`. Les valeurs possibles pour `blockDefault` sont `#all` ou une liste de valeurs parmi les valeurs `extension`, `restriction` et `substitution`. Les valeurs possibles pour `finalDefault` sont `#all` ou une liste de valeurs parmi les valeurs `extension`, `restriction`, `list` et `union`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton"
  xmlns="http://www.liafa.jussieu.fr/~carton"
  elementFormDefault="qualified"
  ...
```

## 6.3.2 Référence explicite à un schéma

Il est possible dans un document de donner explicitement le schéma devant servir à le valider. On utilise un des attributs `schemaLocation` ou `noNamespaceSchemaLocation` dans l'élément racine du document à valider. Ces deux attributs se trouvent dans l'espace de noms des instances de schémas identifié par l'URL `http://www.w3.org/2001/XMLSchema-instance`. L'attribut `schemaLocation` est utilisé lors d'utilisation d'espaces de noms. Il permet en effet de spécifier l'URI qui identifie l'espace de noms des éléments et l'adresse du schéma. La valeur de cet attribut doit être une liste d'URI qui vont par paire. La première de chaque paire est l'URI qui identifie l'espace de noms et la seconde est l'adresse du schéma. L'attribut `noNamespaceSchemaLocation` permet de donner l'adresse du schéma sans espace de noms.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<bibliography
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liafa.jussieu.fr/~carton/ bibliography.xsd">
  ...
```

## 6.4 Commentaires

Les commentaires sont introduits de la façon suivante. Contrairement aux commentaires XML, ils font partie à part entière du schéma XML.

```
<xsd:annotation>
  <xsd:documentation xml:lang="fr">
    Commentaire en français
  </xsd:documentation>
  <xsd:appInfo>
    Information destinée aux applications
  </xsd:appInfo>
</xsd:annotation>
```

## 6.5 Déclarations d'éléments

### 6.5.1 Type nommé

La déclaration la plus simple d'un élément prend la forme suivante.

- 27 -

Les deux attributs `name` et `ref` ne peuvent pas être présents simultanément dans l'élément `xsd:element`. Par contre, l'un des deux doit toujours être présent soit pour donner le nom de l'élément défini soit pour référencer un élément déjà défini.

### 6.5.5 Éléments locaux

Deux éléments définis non globalement dans un schéma peuvent avoir le même nom tout en ayant des types différents. Le schéma suivant définit deux éléments `local` de types `xsd:string` et `xsd:integer`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  ...
  <xsd:element name="strings">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="local" type="xsd:string" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="integers">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="local" type="xsd:integer" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Un document valide pour le schéma suivant est le suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<lists>
  <strings>
    <local>Une chaîne</local>
    <local>A string</local>
  </strings>
  <integers>
    <local>1</local>
    <local>1</local>
  </integers>
</lists>
```

## 6.6 Types prédéfinis

La liste des types prédéfinis est la suivante.

```
string
  Chaîne de caractères
boolean
  true, false, 1 ou 0
float
  Flottant 32 bits conforme à la norme IEEE 754
double
  Flottant 64 bits conforme à la norme IEEE 754
```

- 29 -

Si le type est obtenu par extension ou restriction d'un autre type, l'élément `xsd:complexType` doit un contenir un élément `xsd:simpleContent` ou `complexContent` qui précise si le contenu est purement textuel ou non. La déclaration d'un type complexe prend alors une des deux formes suivantes.

```
<!-- Type dérivé à contenu textuel -->
<xsd:complexType ...>
  <xsd:simpleContent>
    <!-- Extension ou restriction -->
    ...
  </xsd:simpleContent>
</xsd:complexType>

<!-- Type dérivé à contenu pur ou mixte -->
<xsd:complexType ...>
  <xsd:complexContent>
    <!-- Extension ou restriction -->
    ...
  </xsd:complexContent>
</xsd:complexType>
```

### 6.7.3 Contenu mixte

L'attribut `mixed` de l'élément `xsd:complexType` permet de construire un type avec du contenu mixte.

```
<xsd:element name="book">
  <xsd:complexType mixed="true">
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="em"/>
      <xsd:element ref="cite"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

L'exemple précédent est équivalent au fragment suivant de DTD.

```
<!ELEMENT book (#PCDATA | em | cite)* >
```

## 6.8 Constructions de types

Les constructeurs de types permettent de définir des nouveaux types en combinant des types déjà définis. Ils sont en fait assez semblables aux différents opérateurs des DTDs.

### 6.8.1 Élément vide

Si un type complexe déclare uniquement des attributs, le contenu de l'élément est vide. Par exemple, le type suivant déclare un type `Link`. Tout élément de ce type doit avoir un contenu vide et un attribut `ref` de type `xsd:IDREF`.

```
<xsd:element name="link" type="Link"/>
<xsd:complexType name="Link">
  <xsd:attribute name="ref" type="xsd:IDREF" use="required"/>
</xsd:complexType>
```

- 32 -

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="year" type="xsd:string"/>
      <xsd:element name="publisher" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>
```

Les attributs `minOccurs` et `maxOccurs` (voir ci-dessous) des éléments apparaissant sous l'opérateur `xsd:all` ne peuvent pas avoir des valeurs quelconques. La valeur de l'attribut `minOccurs` doit être 0 ou 1 et la valeur de l'attribut `maxOccurs` doit être 1 qui est la valeur par défaut.

### 6.8.5 Opérateur d'union

L'opérateur `xsd:union` définit un nouveau type simple dont les valeurs sont celles des types listés dans l'attribut `memberTypes`.

Voici à titre d'exemple, le type de l'attribut `maxOccurs` tel qu'il pourrait être défini dans un schéma pour les schémas.

```
<xsd:attribute name="maxOccurs" type="IntegerOrUnbounded"/>
<xsd:simpleType name="IntegerOrUnbounded">
  <xsd:union memberTypes="Unbounded xsd:nonNegativeInteger"/>
</xsd:simpleType>
<xsd:simpleType name="Unbounded">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="unbounded"/>
  </xsd:restriction>
</xsd:simpleType>
```

Les types paramètres de l'opérateur d'union peuvent aussi être anonymes. Ils sont alors explicités directement dans le contenu de l'élément `xsd:union` comme dans l'exemple suivant qui conduit à une définition équivalente à celle de l'exemple précédent.

```
<xsd:simpleType name="IntegerOrUnbounded">
  <xsd:union memberTypes="xsd:nonNegativeInteger">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="unbounded"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

### 6.8.6 Opérateur de liste

L'opérateur `xsd:list` définit un nouveau type simple dont les valeurs sont les listes de valeurs du type donné par l'attribut `itemType`. Les valeurs sont séparées par des espaces. L'exemple suivant définit des types pour les listes d'entiers et pour les listes de 5 entiers.

- 34 -

Les schémas permettent de définir une hiérarchie de types qui sont obtenus par *restriction* ou *extension* de types déjà définis. L'extension de type est similaire à l'héritage des langages de programmation orienté objet comme Java ou C++. Elle permet de définir un nouveau type en ajoutant des éléments et/ou des attributs à un type. La restriction permet au contraire d'imposer des contraintes supplémentaires au contenu et aux attributs.

Tous les types prédéfinis ou définis dans un schéma sont dérivés du type `xsd:anyType`. Ce type est aussi le type par défaut lorsqu'une déclaration d'élément ne spécifie pas le type comme la déclaration suivante.

```
<!-- Le type de l'élément object est xsd:anyType -->
<xsd:element name="object"/>
```

### 6.7.1 Types simples

Les types simples définissent uniquement des contenus textuels. Ils peuvent être utilisé pour les éléments ou les attributs. Il sont introduits par l'élément `xsd:simpleType`. Un type simple est souvent obtenu par restriction d'un autre type défini. Il peut aussi être construit par union d'autres types simples ou par l'opérateur de listes. La déclaration d'un type simple a la forme suivante.

```
<xsd:simpleType ...>
  ...
</xsd:simpleType>
```

L'élément `xsd:simpleType` peut avoir un attribut `name` si la déclaration est globale. La déclaration du type se fait ensuite dans le contenu de l'élément `xsd:simpleType` comme dans l'exemple suivant.

```
<xsd:simpleType name="Byte">
  <xsd:restriction base="xsd:nonNegativeInteger">
    <xsd:maxInclusive value="255"/>
  </xsd:restriction>
</xsd:simpleType>
```

### 6.7.2 Types complexes

Les types complexes définissent des contenus purs (constitués uniquement d'éléments), des contenus textuels ou des contenus mixtes. Tous ces contenus peuvent comprendre des attributs. Les types complexes peuvent seulement être utilisé pour les éléments. Ils sont introduits par l'élément `xsd:complexType`. Un type complexe peut être construit explicitement ou être dérivé d'un autre type par extension ou restriction.

La construction explicite d'un type se fait en utilisant les opérateurs de séquence `xsd:sequence`, de choix `xsd:choice` ou d'ensemble `xsd:all`. La construction du type se fait directement dans le contenu de l'élément `xsd:complexType` et prend donc la forme suivante.

```
<!-- Type explicite -->
<xsd:complexType ...>
  <!-- Construction du type avec xsd:sequence, xsd:choice ou xsd:all -->
  ...
</xsd:complexType>
```

- 31 -

Un fragment de document valide peut être le suivant.

```
<link ref="id-42"/>
```

### 6.8.2 Opérateur de séquence

L'opérateur `xsd:sequence` définit un nouveau type formé d'une suite des éléments énumérés. C'est l'équivalent de l'opérateur `,` des DTDs. Les éléments énumérés peuvent être soit des éléments explicites, soit des éléments référencés avec l'attribut `ref` soit des types construits récursivement avec les autres opérateurs. Dans l'exemple suivant, un élément `book` doit contenir les éléments `title`, `author`, `year` et `publisher` dans cet ordre.

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="year" type="xsd:string"/>
      <xsd:element name="publisher" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Cette déclaration est équivalente à la déclaration suivante dans une DTD.

```
<!ELEMENT book (title, author, year, publisher)>
```

### 6.8.3 Opérateur de choix

L'opérateur `xsd:choice` définit un nouveau type formé d'un des éléments énumérés. C'est l'équivalent de l'opérateur `|` des DTDs. Dans l'exemple suivant, le contenu de l'élément `publication` doit être un des éléments `book`, `article` ou `report`. Ces trois éléments sont référencés et doivent donc être définis globalement dans le schéma.

```
<xsd:element name="publication">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="book"/>
      <xsd:element ref="article"/>
      <xsd:element ref="report"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Cette déclaration est équivalente à la déclaration suivante dans une DTD.

```
<!ELEMENT publication (book | article | report)>
```

### 6.8.4 Opérateur d'ensemble

L'opérateur `xsd:all` n'a pas d'équivalent dans les DTDs. Il définit un nouveau type dont chacun des éléments doit apparaître une fois dans un ordre quelconque. Dans la déclaration ci-dessous, les éléments contenus dans l'élément `book` peuvent apparaître dans n'importe quel ordre.

- 33 -

```
<!ELEMENT elem (elem1 | elem2 | elem3)* >

<xsd:element name="elem">
  <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="elem1"/>
      <xsd:element ref="elem2"/>
      <xsd:element ref="elem3"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<!ELEMENT elem (elem1, elem2, elem3)+ >

<xsd:element name="elem">
  <xsd:complexType>
    <xsd:sequence minOccurs="1" maxOccurs="unbounded">
      <xsd:element ref="elem1"/>
      <xsd:element ref="elem2"/>
      <xsd:element ref="elem3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

### 6.8.8 Joker

L'opérateur `xsd:any` permet d'introduire dans un document n'importe quel élément même non déclaré dans le schéma.

## 6.9 Extension de types

Il est possible de définir un nouveau type en étendant un type déjà défini en lui ajoutant des champs ou des attributs. Il faut utiliser l'opérateur `xsd:extension` dont l'attribut `base` précise le type qui étendu. Les éléments sont ajoutés en séquence aux éléments déclaré dans le type de base. L'exemple suivant étend le type `Book` en lui ajoutant un élément `edition` et un attribut `type`.

```
<xsd:complexType name="BookExt">
  <xsd:complexContent>
    <xsd:extension base="Book">
      <xsd:sequence>
        <xsd:element name="edition" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="type" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

### 6.9.1 Utilisation de l'attribut `xsi:type`

Un type peut remplacer dans une instance de document un type dont il dérivent directement ou non. Soit par exemple un élément `elem` déclaré d'un type `BaseType`. Si un type `ExtendedType` a été défini par extension ou restriction du type `BaseType`, il est possible, dans une instance de document, de mettre un élément `elem` avec un contenu de type `ExtendedType`. Pour que le document reste valide, l'élément `elem` doit avoir un attribut `xsi:type` qui précise le type de son contenu. Cet attribut est dans l'espace de nom des instances de schémas. Dans l'exemple suivant, un type `Name` est d'abord déclaré puis un type `FullName` étend ce type en ajoutant un élément `title` et un attribut

Un fragment de document valide peut être le suivant.

```
<price currency="euro">3.14</price>
```

## 6.10 Restriction de types

Il est possible de définir un nouveau type en restreignant les valeurs possibles parmi celles d'un type prédéfini ou déjà défini. Dans l'exemple suivant, le type donné à l'élément `year` est un entier entre 1970 et 2050 inclus. Le type utilisé dans cet exemple est un type anonyme.

```
<xsd:element name="year">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="1970"/>
      <xsd:maxInclusive value="2050"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

L'attribut `xsi:type` peut aussi changer le type d'un élément en un autre type obtenu par restriction du type original. Dans l'exemple suivant, un type `Byte` est déclaré par restriction du type prédéfini `xsd:nonNegativeInteger`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  xmlns="http://www.liafa.jussieu.fr/~carton/">
  <xsd:element name="value" type="xsd:integer"/>
  ...
  <xsd:simpleType name="Byte">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:maxInclusive value="255"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Le document suivant est valide pour ce schéma. Il est possible de changer le type de l'élément `value` en `xsd:nonNegativeInteger` car ce type prédéfini dérive du type prédéfini `xsd:integer`. Cet exemple illustre aussi l'utilisation indispensable des espaces de noms.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<ns:value xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ns:value>-1</ns:value>
  <ns:value xsi:type="xsd:nonNegativeInteger">256</ns:value>
  <ns:value xsi:type="ns:Byte">255</ns:value>
</ns:values>
```

### 6.10.1 Restriction par énumération

Il est possible de définir un nouveau type en donnant explicitement une liste des valeurs possibles d'un type prédéfini ou déjà défini. Dans l'exemple suivant, le type donné à l'élément `language` comprend uniquement les trois chaînes de caractères `de`, `en` et `fr`. Le type utilisé est un type nommé `Language`.

```
id.

<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  xmlns="http://www.liafa.jussieu.fr/~carton/">
  <xsd:element name="name" type="Name"/>
  ...
  <xsd:complexType name="Name">
    <xsd:sequence>
      <xsd:element name="firstname" type="xsd:string"/>
      <xsd:element name="lastname" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="FullName">
    <xsd:complexContent>
      <xsd:extension base="Name">
        <xsd:sequence>
          <xsd:element name="title" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Le document suivant est valide pour ce schéma.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<ns:names xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Éléments name avec le type ns:Name -->
  <ns:name>
    <firstname>Bessiewallis</firstname>
    <lastname>Warfield</lastname>
  </ns:name>
  <!-- Éléments name avec le type ns:FullName -->
  <ns:name id="id52" xsi:type="ns:FullName">
    <firstname>Elizabeth II Alexandra Mary</firstname>
    <lastname> Windsor</lastname>
    <title>Queen of England</title>
  </ns:name>
</ns:names>
```

### 6.9.2 Éléments à contenu textuel

Pour obtenir un élément dont le contenu est uniquement du texte, il faut étendre un type simple en ajoutant éventuellement des attributs.

```
<xsd:element name="price">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:simpleType name="IntList">
  <xsd:list itemType="xsd:integer"/>
</xsd:simpleType>
<xsd:simpleType name="IntList5">
  <xsd:restriction base="IntList">
    <xsd:length value="5"/>
  </xsd:restriction>
</xsd:simpleType>
```

### 6.8.7 Répétitions

Les attributs `minOccurs` et `maxOccurs` permettent de préciser le nombre minimal ou maximal d'occurrences d'un élément ou d'un groupe. Ils sont l'équivalent des opérateurs `?`, `*` et `+` des DTDs. Ils peuvent apparaître comme attribut des éléments `xsd:element`, `xsd:sequence`, `xsd:choice` et `xsd:all`. L'attribut `minOccurs` prend un entier comme valeur. L'attribut `minOccurs` prend un entier ou la chaîne `unbounded` comme valeur pour indiquer qu'il n'y a pas de nombre maximal. La valeur par défaut de ces deux attributs est la valeur 1.

L'utilisation des attributs `minOccurs` et `maxOccurs` est illustrée par l'équivalent en schéma de quelques fragments de DTD

```
<!ELEMENT elem (elem1, elem2?, elem3*) >

<xsd:element name="elem">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="elem1"/>
      <xsd:element ref="elem2" minOccurs="0"/>
      <xsd:element ref="elem3" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!ELEMENT elem (elem1, (elem2 | elem4), elem3) >

<xsd:element name="elem">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="elem1"/>
      <xsd:choice>
        <xsd:element ref="elem2"/>
        <xsd:element ref="elem3"/>
      </xsd:choice>
      <xsd:element ref="elem4"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!ELEMENT elem (elem1, elem2, elem3)* >

<xsd:element name="elem">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="elem1"/>
      <xsd:element ref="elem2"/>
      <xsd:element ref="elem3"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

\I Caractère ne commençant pas un nom  
 {n} Répétition n fois  
 {m,n} Répétition entre m et n fois  
 [x-y] Un caractère entre x et y

Il faut remarquer que les restrictions par énumération ou par motif se combinent avec un *ou logique*. Le contenu doit être une des valeurs énumérées ou il doit être décrit par un des motifs. Au contraire, les autres restrictions comme `minInclusive` et `maxInclusive` se combinent avec un *et logique*. Le contenu doit vérifier toutes les contraintes pour être valide.

### 6.10.3 Liste des facettes

enumeration  
 length  
 maxLength  
 minLength  
 maxExclusive  
 maxInclusive  
 minExclusive  
 minInclusive  
 pattern  
 fractionDigits  
 totalDigits  
 whiteSpace

### 6.10.4 Restriction de types complexes

Il est possible de déclarer un type par restriction d'un type complexe. Il est ainsi possible d'imposer des contraintes supplémentaires sur le contenu et les attributs.

Dans l'exemple suivant, le type `Shortname` est obtenu par restriction du type `Name`. La valeur de l'attribut `maxOccurs` passe de `unbounded` à `1`. L'attribut `id` devient obligatoire.

```
<xsd:complexType name="Name">
  <xsd:sequence>
    <xsd:element name="firstname" type="xsd:string" maxOccurs="unbounded"/>
    <xsd:element name="lastname" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
<xsd:complexType name="Shortname">
  <xsd:complexContent>
    <xsd:restriction base="Name">
      <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string" maxOccurs="1"/>
        <xsd:element name="lastname" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="required"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

```
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Le document suivant est valide pour ce schéma. L'élément `integers` contient des éléments `positive` et `negative` qui sont substitués à des élément `integer`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<integers>
  <integer>0</integer>
  <positive>1</positive>
  <negative>-1</negative>
</integers>
```

### 6.11.1 Chef de groupe abstrait

Un élément peut être déclaré *abstrait* en donnant la valeur `true` à l'attribut `abstract` de l'élément `xsd:element`. Ce mécanisme est assez semblable à la notion de classe abstraite des langages de programmation orienté objet comme Java ou C++. Un élément déclaré abstrait peut être utilisé dans la déclaration d'un autre élément. Par contre, il ne peut pas être présent dans un document instance. Le mécanisme a uniquement de l'intérêt lorsque l'élément abstrait est chef d'un groupe de substitution.

Si l'élément `integer` de l'exemple précédent est déclaré abstrait, celui-ci doit obligatoirement être substitué par un des éléments `positive` ou `negative`.

```
...
<xsd:element name="integer" type="xsd:integer" abstract="true"/>
...
```

Le document précédent n'est alors plus valide pour le nouveau schéma car l'élément `integer` ne peut plus apparaître dans le contenu de l'élément `integers`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<integers>
  <!-- Élément integer impossible -->
  <positive>1</positive>
  <negative>-1</negative>
</integers>
```

## 6.12 Contrôle des dérivations et substitutions

### 6.12.1 Éléments et types abstraits

Il a été vu qu'un élément peut être déclaré abstrait. L'élément ne peut alors pas apparaître dans un document instance. Il est aussi possible de déclarer un type abstrait en donnant la valeur `true` à l'attribut `abstract` des éléments `xsd:simpleType` et `xsd:complexType`. Cette déclaration n'empêche pas de déclarer un élément du type en question. L'élément ainsi déclaré peut apparaître dans un document instance mais seulement avec un type dérivé.

Dans l'exemple suivant, on définit un type abstrait `Abstract` sans contrainte. Ce type est alors équivalent au type `xsd:anyType`. On dérive ensuite deux types par extension `Derived1` et `Derived2`.

```
<xsd:element name="language" type="Language"/>
<xsd:simpleType name="Language">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="de"/>
    <xsd:enumeration value="en"/>
    <xsd:enumeration value="fr"/>
  </xsd:restriction>
</xsd:simpleType>
```

### 6.10.2 Restriction par motif

Il est possible de définir un nouveau type en donnant un motif, c'est-à-dire une expression rationnelle qui décrit les valeurs possibles d'un type pré-défini ou déjà défini. Dans l'exemple suivant, le type `ISBN` décrit explicitement tous les formes possibles des numéros `ISBN`.

```
<xsd:simpleType name="ISBN">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d-\d(2)-\d(6)-[\dX]"/>
    <xsd:pattern value="\d-\d(3)-\d(5)-[\dX]"/>
    <xsd:pattern value="\d-\d(4)-\d(4)-[\dX]"/>
    <xsd:pattern value="\d-\d(5)-\d(3)-[\dX]"/>
  </xsd:restriction>
</xsd:simpleType>
```

Le type suivant `Identifieur` définit un type pour les identificateurs qui commencent par une lettre minuscule ou majuscule ou le caractère `'_'` puis se prolongent par une suite de caractères alphanumériques ou le caractère `'_'`.

```
<xsd:simpleType name="Identifieur">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Za-z_][0-9A-Za-z_]*"/>
  </xsd:restriction>
</xsd:simpleType>
```

La syntaxe des expressions rationnelles qui peuvent être employées avec la restriction `pattern` sont les suivantes.

- .
- \S Tous les caractères sauf les retours à la ligne (`[\n\r]`)
- \s Espaces (espace, tabulation ou retour de ligne [`#x20\t\n\r`])
- \S Caractère autres que les espaces
- \d Chiffre
- \D Caractère autres que les chiffres
- \w Caractère alphanumérique et le tiret `'-'`
- \W Caractère autres que les Caractère alphanumérique et le tiret
- \i Caractère commençant un nom (lettre, `'_'` ou `':'`)

```
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:schema>
```

Il est aussi possible de restreindre un type complexe en remplaçant le type d'un élément par un type dérivé. Dans l'exemple suivant, le type de l'élément `integer` est `xsd:integer` dans le type `Base`. Ce type est remplacé par le type `xsd:nonNegativeInteger` dans le type `Restriction`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <xsd:complexType name="Base">
    <xsd:sequence>
      <xsd:element name="integer" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Restriction">
    <xsd:complexContent>
      <xsd:restriction base="Base">
        <xsd:sequence>
          <xsd:element name="integer" type="xsd:nonNegativeInteger"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

## 6.11 Groupes de substitution

Il est possible de spécifier qu'un élément peut être remplacé par un autre élément dans les documents instances. Ce mécanisme est différent de l'utilisation de l'attribut `xsi:type` puisque c'est l'élément même qui est remplacé et pas seulement le type. Le type de l'élément substitué doit avoir un type dérivé du type de l'élément original.

Ce mécanisme est mis en œuvre en créant une *groupe de substitution*. Un groupe est formé d'un élément *chef de groupe* (*group head* en anglais) et d'autres éléments qui se rattachent au chef de groupe. Le chef de groupe peut être remplacé dans un document instance par n'importe quel autre élément du groupe. Le chef de groupe n'est pas identifié directement. En revanche, tous les autres éléments déclarent leur rattachement au groupe avec l'attribut `substitutionGroup` dont la valeur est le nom du chef de groupe. Dans l'exemple suivant, le chef de groupe est l'élément `integer`. Les éléments `positive` et `negative` peuvent être substitués à l'élément `integer`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <!-- Chef de groupe -->
    <xsd:element name="integer" type="xsd:integer"/>
    <!-- Autres éléments du groupe -->
    <xsd:element name="positive" type="xsd:positiveInteger" substitutionGroup="integer"/>
    <xsd:element name="negative" type="xsd:negativeInteger" substitutionGroup="integer"/>
    <xsd:element name="integers">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="integer" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
```

schema.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  . . .
  <!-- L'attribut final="#all" empêche les extensions et restrictions -->
  <xsd:complexType name="Base" final="#all">
    <xsd:sequence>
      <xsd:element name="integer" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Extension">
    <xsd:complexContent>
      <xsd:extension base="Base">
        <xsd:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Restriction">
    <xsd:complexContent>
      <xsd:restriction base="Base">
        <xsd:sequence>
          <xsd:element name="integer" type="xsd:nonNegativeInteger"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Le document suivant n'est pas valide pour ce schéma mais il le devient dès qu'on supprime l'attribut `final="#all"` de l'élément `complexType`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<values xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Document non valide à cause de final="#all" dans le schéma -->
  <value>
    <integer>-1</integer>
  </value>
  <value xsi:type="Extension" att="Un attribut">
    <integer>-1</integer>
  </value>
  <value xsi:type="Restriction">
    <integer>1</integer>
  </value>
</values>
```

### 6.12.3 Substitutions de types

L'attribut `block` de l'élément `xsd:complexType` permet d'empêcher qu'un élément du type défini puisse prendre un autre type dérivé dans un document instance. La valeur de cet attribut est soit la chaîne `#all` soit une liste de valeurs parmi les valeurs `extension`, `restriction` et `substitution`. Les valeurs énumérées ou toutes pour `#all` bloquent les différents types qui peuvent remplacer le type pour un élément. La valeur par défaut de cet attribut est donnée par la valeur de l'attribut `blockDefault` de l'élément `schema`.

La différence entre les attributs `final` et `block` est que `final` concerne la définition de type dérivés alors que `block` concerne l'utilisation des types dérivés dans les documents instance.

- 44 -

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  xmlns="http://www.liafa.jussieu.fr/~carton/"
  xmlns:ht="http://www.liafa.jussieu.fr/~carton/"
  <xsd:element name="value" type="Abstract"/>
  <xsd:element name="values">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="value" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="Abstract" abstract="true"/>
  <xsd:complexType name="Derived1">
    <xsd:complexContent>
      <xsd:extension base="Abstract">
        <xsd:sequence>
          <xsd:element name="string" type="xsd:string"/>
        </xsd:sequence>
      </xsd:extension>
      <xsd:attribute name="att" type="xsd:string"/>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Derived2">
    <xsd:complexContent>
      <xsd:extension base="Abstract">
        <xsd:sequence>
          <xsd:element name="string" type="xsd:string"/>
          <xsd:element name="integer" type="xsd:integer"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Le document suivant est valide pour ce schéma. L'élément apparaît deux fois dans le document mais avec les types `Derived1` et `Derived2`. Ces types sont déclarés à l'aide de l'attribut `xsi:type`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<ns:values xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Élément value de type Abstract impossible -->
  <ns:value xsi:type="ns:Derived1" att="avec un attribut">
    <string>Une chaîne</string>
  </ns:value>
  <ns:value xsi:type="ns:Derived2">
    <string>Un entier</string>
    <integer>1</integer>
  </ns:value>
</ns:values>
```

### 6.12.2 Types dérivés

L'attribut `final` de l'élément `complexType` permet d'empêcher que le type défini puisse servir de type de base à des constructions ou à des dérivations de type. La valeur de cet attribut est soit la chaîne `#all` soit une liste de valeurs parmi les valeurs `extension`, `restriction`, `list` et `union`. Les valeurs énumérées ou toutes pour `#all` bloquent les différentes façons de définir des nouveaux types. La valeur par défaut de cet attribut est donnée par la valeur de l'attribut `finalDefault` de l'élément

- 43 -

## 6.14.1 Groupe d'éléments

L'opérateur `xsd:group` permet de nommer un groupe d'éléments. Dans l'exemple suivant, le groupe `fullname` est constitué d'un élément `first` et d'un élément `last` dans cet ordre.

```
<xsd:group name="Fullname">
  <xsd:sequence>
    <xsd:element name="first" type="xsd:string"/>
    <xsd:element name="last" type="xsd:string"/>
  </xsd:sequence>
</xsd:group>
```

Une telle déclaration peut être employée de la façon suivante dans la déclaration d'éléments ou de type.

```
<xsd:complexType name="Person">
  <xsd:sequence>
    <xsd:group ref="Fullname"/>
    <xsd:element name="country" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

## 6.14.2 Groupe d'attributs

De façon similaire, l'opérateur `attributeGroup` permet de regrouper la définition de plusieurs attributs dans le but d'une réutilisation. Dans l'exemple suivant, le groupe d'attributs `langtype` regroupe des attributs `lang` et `type`.

```
<xsd:attributeGroup name="Langtype">
  <xsd:attribute name="lang" type="xsd:string"/>
  <xsd:attribute name="type" type="xsd:string"/>
</xsd:attributeGroup>
```

Une telle déclaration peut être employée de la façon suivante dans la définition d'un type complexe nommé ou anonyme.

```
<xsd:complexType name="SomeType">
  <xsd:attributeGroup ref="Langtype"/>
</xsd:complexType>
```

Un groupe d'attributs peut bien sûr être utilisé dans la définition d'un autre groupe d'attributs.

```
<xsd:attributeGroup name="langtypeclass">
  <xsd:attributeGroup ref="Langtype"/>
  <xsd:attribute name="class" type="xsd:string"/>
</xsd:attributeGroup>
```

## 6.15 Contraintes de cohérence

Les schémas permettent de spécifier des contraintes globales de cohérence. Ces contraintes doivent être vérifiées par un document pour que celui-ci soit valide. Ces contraintes sont de deux types. Elles peuvent porter sur l'*unicité* ou sur l'*existence* de certains éléments.

- 46 -

## 6.13 Déclarations d'attributs

La déclaration d'un attribut prend la forme suivante.

```
<xsd:attribute name="(attr)" type="(type)"/>
```

où `attr` et `type` sont respectivement le nom et le type de l'attribut. L'exemple suivant déclare un attribut `lang` de type `xsd:NMTOKEN`.

```
<xsd:attribute name="lang" type="xsd:NMTOKEN"/>
```

Comme pour un élément, le type d'un attribut peut être anonyme. Il est alors défini dans le contenu de l'élément `xsd:attribute`. Cette possibilité est illustrée dans l'exemple suivant. La valeur de l'attribut `lang` déclaré ci-dessous peut être la chaîne `en` ou la chaîne `fr`.

```
<xsd:attribute name="lang">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="en"/>
      <xsd:enumeration value="fr"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

### 6.13.1 Attribut optionnel ou obligatoire

Par défaut, un attribut est optionnel. Il peut être rendu obligatoire en donnant la valeur `required` à l'attribut `use` de l'élément `xsd:attribute`. L'attribut `use` peut aussi prendre la valeur `optional` pour indiquer que l'attribut est optionnel. C'est la valeur par défaut de cet attribut.

```
<xsd:attribute name="lang" type="xsd:NMTOKEN" use="required"/>
```

### 6.13.2 Valeur par défaut et valeur fixe

Comme pour les éléments, il est possible de donner une valeur par défaut ou une valeur fixe à l'attribut comme dans les exemples suivants.

```
<xsd:attribute name="lang" type="xsd:NMTOKEN" default="fr"/>
```

```
<xsd:attribute name="lang" type="xsd:NMTOKEN" fixed="fr"/>
```

## 6.14 Groupes d'éléments et d'attributs

Il est possible de nommer des groupes d'éléments et des groupes d'attributs afin de pouvoir les réutiliser. Ce mécanisme aide à structurer un schéma complexe et vise à obtenir une meilleure modularité dans l'écriture de schémas. Les groupes d'éléments et d'attributs sont respectivement définis par les éléments `xsd:group` et `xsd:attributeGroup`.

- 45 -

Le schéma correspondant impose trois contraintes suivantes sur le fichier XML.

1. Deux commandes `orders` n'ont pas la même date *et* la même heure.
2. Deux produits du catalogue n'ont pas le même numéro de série.
3. Tous les produits référencés dans les commandes sont présents dans le catalogue.

Le début de ce schéma XML est le suivant.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="list">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="orders" type="Orders"/>
        <xsd:element name="catalog" type="Catalog"/>
      </xsd:sequence>
      <xsd:attribute name="period" type="xsd:duration"/>
    </xsd:complexType>
    <!-- Unicité du couple (date,heure) -->
    <xsd:unique name="dummy">
      <xsd:selector xpath="orders/order"/>
      <xsd:field xpath="@date"/>
      <xsd:field xpath="@time"/>
    </xsd:unique>
    <!-- Unicité du numéro de série -->
    <xsd:key name="serial">
      <xsd:selector xpath="catalog/product"/>
      <xsd:field xpath="@serial"/>
    </xsd:key>
    <!-- Existence dans le catalogue de tout produit commandé -->
    <xsd:keyref name="unused" refer="serial">
      <xsd:selector xpath="orders/order/product"/>
      <xsd:field xpath="@serial"/>
    </xsd:keyref>
  </xsd:element>
  <!-- Suite du schéma -->
  ...
</xsd:schema>
```

## 6.16 Schémas et espaces de noms

Un des avantages des schémas par rapport aux DTDs est la prise en charge des espaces de noms. L'attribut `targetNamespace` de l'élément `schema` permet de préciser l'espace de noms des éléments et des types définis par le schéma.

### 6.16.1 Utilisation sans espace de noms

Pour une utilisation plus simple, il est possible d'ignorer les espaces de noms. Il est alors possible de valider des documents dont tous les éléments n'ont pas d'espace de noms. Il suffit pour cela les noms des éléments du document ne soit pas qualifiés (sans le caractère ':') et que l'espace de noms par défaut ne soit pas spécifié (cf. espaces de noms).

Si l'attribut `targetNamespace` de l'élément `schema` est absent, tous les éléments et types définis dans le schéma sont sans espace de noms. Il faut cependant déclarer l'espace de noms des schémas pour qualifier les éléments des schémas (`xsd:element`, `xsd:complexType`, ...).

- 48 -

Un document valide pour ce schéma est le suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<ns:name xmlns:ns="http://www.liafa.jussieu.fr/~carton/">
  <ns:firstname>Gaston</ns:firstname>
  <ns:lastname>Lagaffe</ns:lastname>
</ns:name>
```

Le comportement pour les attributs est identique mais il est gouverné par l'attribut `attributeFormDefault` de l'élément `schema`. La valeur par défaut de cet attribut est aussi unqualified.

Les éléments et attributs définis globalement sont toujours dans l'espace de noms cible. Pour les éléments et attributs locaux, il est possible de changer le comportement dictés par `elementFormDefault` et `attributeFormDefault` en utilisant l'attribut `form` des éléments `element` et `attribute`. Cet attribut peut prendre les valeurs `qualified` ou `unqualified`. Le schéma suivant spécifie que l'élément `firstname` doit être qualifié.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  elementFormDefault="unqualified">
  <xsd:element name="name">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string" form="qualified"/>
        <xsd:element name="lastname" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Un document valide pour ce schéma est le suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<ns:name xmlns:ns="http://www.liafa.jussieu.fr/~carton/">
  <ns:firstname>Gaston</ns:firstname>
  <lastname>Lagaffe</lastname>
</ns:name>
```

### 6.16.3 Références

Lorsqu'un élément, un attribut ou un type défini globalement est référencé par un attribut `ref` ou `type`, la valeur de cet attribut doit contenir le nom qualifié. Ceci oblige à associer un préfixe à l'espace de noms cible et à l'utiliser pour qualifier l'élément ou le type référencé comme dans le schéma suivant. Il faut remarquer que les éléments, attributs et type définis doivent être nommé avec un nom non qualifié.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  elementFormDefault="qualified">
  <!-- Référence au type Name par son nom qualifié -->
  <xsd:element name="name" type="ns:Name" />
  <xsd:complexType name="Name">
```

- 50 -

### 6.15.1 Unicité

L'*unicité* signifie que le contenu ou des valeurs des attributs de certains éléments doivent être unique dans une partie déterminée du document. Cette notion généralise les attributs de type ID des DTDs. Ces contraintes sont introduites par les éléments `xsd:key` et `xsd:unique`.

Dans l'exemple, la contrainte est décrite au niveau de l'élément `bibliography` pour exprimer que l'attribut `key` de `book` doit être unique dans le contenu de l'élément `bibliography`.

```
<xsd:element name="bibliography" type="Bibliography">
  <xsd:key name="dummy">
    <xsd:selector xpath="book"/>
    <xsd:field xpath="@key"/>
  </xsd:key>
</xsd:element>
```

Une contrainte décrite avec `xsd:key` implique que les champs impliqués soient nécessairement présents et non annulables. Une contrainte décrite avec `xsd:unique` est au contraire seulement vérifiée pour les éléments dont les champs impliqués sont présents.

### 6.15.2 Référence

L'*existence* signifie qu'il doit exister dans une partie déterminée du document un élément ayant une certaine valeur pour son contenu ou des attributs. Cette notion généralise les attributs de type IDREF des DTDs. Ces contraintes sont introduites par l'élément `xsd:keyref`.

### 6.15.3 Exemple complet

Voici un exemple de fichier XML représentant une liste de commandes. Chaque commande concerne un certain nombre d'articles qui sont référencés dans le catalogue donné à la fin.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<list period="P2D">
  <orders>
    <order date="2008-01-08" time="17:32:28">
      <product serial="101-XX" number="12"/>
      <product serial="102-XY" number="23"/>
      <product serial="101-2A" number="10"/>
    </order>
    <order date="2008-01-09" time="17:32:28">
      <product serial="101-XX" number="32"/>
    </order>
    <order date="2008-01-09" time="17:32:29">
      <product serial="101-XX" number="32"/>
    </order>
  </orders>
  <catalog>
    <product serial="101-XX">Product n° 1</product>
    <product serial="101-2A">Product n° 2</product>
    <product serial="102-XY">Product n° 3</product>
    <product serial="102-XA">Product n° 4</product>
  </catalog>
</list>
```

- 47 -

### 6.16.2 Noms et attributs dans l'espace de noms cible

Pour spécifier un espace de noms cible dans lequel sont définis les éléments, l'attribut `targetNamespace` de l'élément `schema` doit contenir l'URI associé à cet espace de noms. Les éléments qui sont effectivement définis dans l'espace de noms dépend de la valeur de l'attribut `elementFormDefault` de l'élément `xsd:schema`.

Si la valeur de l'attribut `elementFormDefault` est `unqualified` qui est sa valeur par défaut, seuls les éléments définis globalement, c'est-à-dire quand l'élément `element` est directement fils de l'élément `schema` sont dans l'espace de noms cible. Les autres sont sans espace de noms. Dans le schéma suivant, l'élément `name` est dans l'espace de noms `http://www.liafa.jussieu.fr/~carton/` alors que les éléments `firstname` et `lastname` sont sans espace de noms.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- unqualified est la valeur par défaut de elementFormDefault -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  elementFormDefault="unqualified">
  <xsd:element name="name">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string"/>
        <xsd:element name="lastname" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Un document valide pour ce schéma est le suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<ns:name xmlns:ns="http://www.liafa.jussieu.fr/~carton/">
  <firstname>Gaston</firstname>
  <lastname>Lagaffe</lastname>
</ns:name>
```

Si la valeur de l'attribut `elementFormDefault` est `qualified`, tous les éléments sont dans l'espace de noms cible. Dans le schéma suivant, les trois éléments `name`, `firstname` et `lastname` sont dans l'espace de noms `http://www.liafa.jussieu.fr/~carton/`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  elementFormDefault="qualified">
  <xsd:element name="name">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string"/>
        <xsd:element name="lastname" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

- 49 -

Un document valide pour ce schéma est le suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<ns:name xml:lang="fr"
  xmlns:ns="http://www.liafa.jussieu.fr/~carton/"
  >
  Élément avec attribut xml:lang
</ns:name>
```

- 52 -

```
<xsd:sequence>
  <xsd:element name="firstname" type="xsd:string"/>
  <xsd:element name="lastname" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Une autre solution consiste assez commode consiste à rendre l'espace de noms par défaut égal à l'espace de noms cible comme dans l'exemple suivant. Ceci impose bien sûr de pas utiliser l'espace de noms par défaut pour les éléments des schémas comme il pourrait être tentant de le faire.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  xmlns="http://www.liafa.jussieu.fr/~carton/"
  elementFormDefault="qualified">
  <!-- Référence au type Name par son nom qualifié -->
  <xsd:element name="name" type="Name" />
  <xsd:complexType name="Name">
    <xsd:sequence>
      <xsd:element name="firstname" type="xsd:string"/>
      <xsd:element name="lastname" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

## 6.17 Imports d'autres schémas

Dans un soucis de modularité, il est possible d'importer d'autres schémas dans un schéma à l'aide des attributs `xmlns:include` et `xmlns:import`. L'élément `xmlns:include` est employé lorsque l'espace de noms cible est identique pour le schéma importé. L'élément `xmlns:import` est employé lorsque l'espace de noms cible du schéma importé est différent de celui qui réalise l'import.

Le schéma à l'adresse <http://www.w3.org/2001/xml.xsd> contient une définition des attributs `xml:lang`, `xml:space`, `xml:id` de l'espace de noms xml associé à l'URL <http://www.w3.org/XML/1998/namespace>. Le schéma suivant importe ce schéma et utilise le groupe d'attributs `xml:specialAttrs` pour ajouter des attributs à l'élément `name`.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.liafa.jussieu.fr/~carton/"
  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="http://www.w3.org/2001/xml.xsd"/>
  <xsd:element name="name">
    <xsd:complexType>
      <xsd:simpleContent>
        <!-- Le contenu est purement textuel -->
        <xsd:extension base="xsd:string"/>
        <!-- L'élément name a les attributs xml:lang, xml:space ... -->
        <xsd:attributeGroup ref="xml:specialAttrs"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

- 51 -

La forme générale d'un pas d'un chemin XPath est la suivante.

```
(axe):(type){(prédicate1)} ... {(prédicatem)}
```

La relation appelée *axe* dans la terminologie XML détermine la relation qui doit exister entre les nœuds du nouvel ensemble défini et ceux du contexte. Le *type* fixe ensuite la catégorie des nœuds sélectionnés. Les *m* prédicats `prédicate1`, ..., `prédicatem` donnent des contraintes que doivent satisfaire les nœuds pour être sélectionnés.

## 7.3 Axes

Chacun des axes donne une relation qui relie le nœud du nouvel ensemble défini et ceux du contexte à partir duquel est évalué le pas de l'expression. Les axes qu'il est possible d'utiliser dans les expressions XPath sont les suivants.

```
self
  Le nœud lui-même (égalité)
child
  Fils direct
parent
  Parent
attribute
  Attribut du nœud
descendant
  Descendant stricte
descendant-or-self
  Descendant ou le nœud lui-même
ancestor
  Ancêtre strict
ancestor-or-self
  Ancêtre ou le nœud lui-même
preceding-sibling
  Frère gauche (fils du même parent)
following-sibling
  Frère droit (fils du même parent)
preceding
  À gauche
following
  À droite
namespace
  Espace de nom du nœud
```

Les axes `self`, `parent`, `child`, `preceding-sibling` et `following-sibling` permettent de sélectionner le nœud lui-même et les nœuds proches comme le montre la figure ci-dessous.

## 7 XPath

XPath est un langage d'expressions permettant de décrire des ensembles de nœuds d'un document XML. Ces expressions ressemblent aux chemins Unix pour nommer des fichiers dans une arborescence.

Il existe deux syntaxes pour ces expressions : une syntaxe explicite et une syntaxe abrégée permettant d'écrire de façon concise les constructions les plus courantes.

### 7.1 Arbre d'un document XML

Nœuds possibles

- document node (root node)
- element node
- attribute node
- comment node
- processing instruction node
- text node
- namespace node

### 7.2 Forme des expressions

Les expressions XPath ont la forme suivante.

```
(path1) | (path2) | ... | (pathm)
```

où chaque chemin `pathi` décrit un ensemble de nœud. L'ensemble des nœuds décrit est l'union des ensembles décrits par chacun des chemins `pathi`.

Comme pour les noms de fichier Unix, il existe des chemins *absolus* et des chemins *relatifs*. Les chemins absolus et relatifs ont les formes respectives suivantes.

```
/(step1)/(step2)/(step3)/.../(stepn)
(step1)/(step2)/(step3)/.../(stepn)
```

où chaque partie `stepi` est appelée un *pas* du chemin. Chaque pas décrit un ensemble de nœuds d'un document XML. Cet ensemble de nœuds est évalué de la gauche vers la droite. Un chemin relatif est nécessairement évalué à partir d'un contexte égal à un ensemble de nœuds de l'arbre. Au contraire, les chemins absolus sont évalués en partant de la racine de l'arbre.

L'ensemble de nœuds décrit par un chemin absolu `/(step1)/.../(stepn)` est égal à l'ensemble de nœuds décrit par le chemin relatif `(step1)/.../(stepn)` (obtenu en supprimant le caractère `/'` initial) évalué en partant de l'ensemble de nœuds réduit à la racine de l'arbre.

L'ensemble de nœuds décrit par un chemin relatif est évalué de la façon suivante. On part du contexte initial. Ensuite, chaque pas permet de passer d'un contexte à un nouvel ensemble de nœuds qui sert de contexte au pas suivant. L'ensemble donné pas le dernier pas du chemin est alors l'ensemble décrit par le chemin tout entier.

- 53 -

## 7.4 Types

Une fois donné un axe, le type permet de restreindre l'ensemble des nœuds sélectionnés à un des nœuds d'une certaine forme. Les types possibles sont les suivants.

```
*
  tous les éléments ou tous les attributs (suivant l'axe)
(name)
  les nœuds de nom name (éléments ou attributs suivant l'axe choisi)
node()
  tous les nœuds
text()
  tous les nœuds textuels
comment()
  tous les commentaires
processing-instruction()
  toutes les instructions de traitement
```

## 7.5 Prédicats

Un prédicat mis entre crochets permet encore de restreindre l'ensemble des nœuds sélectionnés. Les prédicats peuvent combiner avec les opérateurs `or` et `and` plusieurs conditions qui peuvent faire intervenir des fonctions prédéfinies.

Les fonctions prédéfinies dans XPath sont les suivantes.

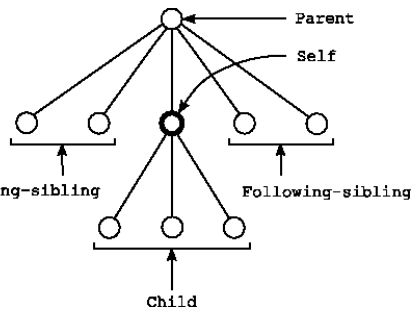
- Les relations `=`, `!=`, `<`, `>`, `<=` et `>=` pour les comparaisons.
- Les opérateurs `+`, `-`, `*`, `div` et `mod` pour les opérations arithmétiques.
- Les opérateurs `and` et `or` ainsi que les fonctions booléennes `true()`, `false()` et `not(...)`.
- Les fonctions `name()`, `local-name()`, `count()`, `position()`, `last()`, `id()`.
- Les fonctions `sum(...)`, `floor(...)`, `ceiling(...)`, `round(...)`.
- Les fonctions `boolean(...)`, `number(...)`, `string(...)` pour les conversions de type.
- Les fonctions `starts-with(...)`, `contains(...)`, `string-length(...)`, `substring(...)`, `substring-before(...)`, `substring-after(...)`, `concat(...)`, `normalize-space(...)` pour la manipulation des chaînes de caractères.

## 7.6 Syntaxe abrégée

Les constructions les plus fréquentes des expressions XPath peuvent être abrégées de façon à avoir des expressions plus concises. Les abréviations possibles sont les suivantes.

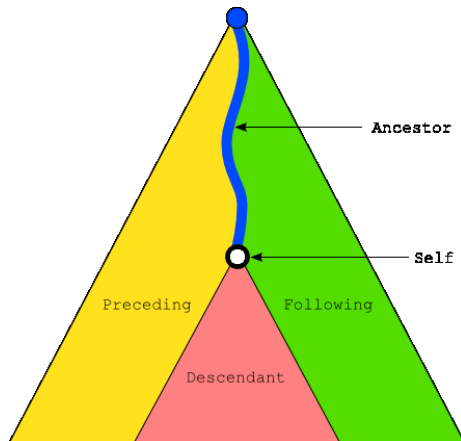
```
l'axe child:: peut être omis.
..
est l'abréviation de parent::.
.
est l'abréviation de self::node().
@
est l'abréviation de attribute::.
```

- 56 -



Nœuds proches

Les cinq axes `self`, `ancestor`, `preceding`, `descendant` et `following` partitionnent l'ensemble de tous les éléments du document en cinq parties comme le montre la figure ci-dessous.



Partitionnement des éléments selon les cinq axes

- 55 -

```
1 ELEMENT book
  ATTRIBUTE key
  TEXT
  content=Marchal00
  ATTRIBUTE lang
  TEXT
  content=en
2 ELEMENT book
  ATTRIBUTE key
  TEXT
  content=Zeldman03
  ATTRIBUTE lang
  TEXT
  content=en
/ > cd bibliography/book[3]
book > xpath @lang
Object is a Node Set :
Set contains 1 nodes:
1 ATTRIBUTE lang
  TEXT
  content=fr
book > cd ..
bibliography >
```

## 7.9 XPath 2.0

### 7.9.1 Modèle de données

La donnée universelle de XPath 2.0 est la *liste*. Une liste XPath peut contenir des nœuds de l'arbre XML ou des valeurs atomiques. Les valeurs atomiques comprennent les chaînes de caractères, les entiers, les flottants et tous les types de base des schémas XML. Chaque liste possède une longueur. Une seule valeur est assimilée à une liste de longueur 1. Les éléments d'une liste XPath sont bien sûr ordonnés.

Les types pour les valeurs atomiques sont les suivants.

```
xsd:string
  Chaîne de caractères: 'string' ou "string"
xsd:boolean
  Booléen: true() et false()
xsd:decimal
  Nombre décimal: 3.14
xsd:float et xsd:double
  Nombre flottant en simple ou double précision
xsd:integer
  Entier: 42
xsd:duration, xsd:yearMonthDuration et xsd:dayTimeDuration
  Durée: P6Y4M2DT11H22M44S
xsd:dateTime, xsd:date et xsd:date
  Date et heure: 2009-03-02T11:44:22
xsd:anyURI
  URL: http://www.liafa.jussieu.fr/~carton/
```

- 58 -

```
//
est l'abréviation de descendant-or-self::node().
[n]
est l'abréviation de [position()=n] où n est un entier.
```

## 7.7 Exemples

Voici quelques exemples d'expressions XPath.

```
child::text ou text
  élément text fils d'un nœud du contexte.
/child::text ou /text
  élément text sous la racine c'est-à-dire contenant tout le document
attribute::text ou @text
  attribut text d'un nœud du contexte.
child::chapter/child::section ou chapter/section
  élément section fils d'un nœud chapter lui-même fils d'un nœud du contexte.
parent::node()/child::chapter ou ../chapter
  élément chapter frère d'un nœud du contexte.
self::node()/descendant-or-self::section ou ../section
  élément section descendant d'un nœud du contexte.
child::section[position()=2] ou section[2]
  deuxième élément section fils du nœud courant.
descendant::p/following-sibling::em[position()=1] ou
../p/following-sibling::em[1]
  premier frère em d'un élément p descendant du nœud courant.
child::section[child::title] ou section[title]
  élément section fils du nœud courant et ayant un fils title.
child::section[attribute::title] ou section[@title]
  élément section fils du nœud courant et ayant un attribut title.
child::section[attribute::title='Un titre'] ou section[@title='Un
titre']
  élément section fils du nœud courant et ayant un attribut title égal à la chaîne Un titre.
chapter[count(child::section) > 1]
  élément chapter ayant plus d'un élément section comme fils.
@*[name() != 'id']
  attributs autres que l'attribut id.
section[2][@type='dual']
  deuxième élément section du nœud courant si son attribut type vaut dual.
```

## 7.8 Utilisation de xmllint

L'application `xmllint` possède un interpréteur de commandes qui est lancé avec l'option `--shell`. Il est alors possible de naviguer dans le document XML et d'obtenir la valeur d'une expression XPath comme dans l'exemple ci-dessous. La commande `help` de l'interpréteur affiche toutes les commandes disponibles.

```
bash $ xmllint --shell bibliography.xml
/ > xpath bibliography/book[@lang='en']
Object is a Node Set :
Set contains 2 nodes:
```

- 57 -

### 7.9.3 Exemples

Cette page contient des exemples d'expressions XPath avec le résultat de leur évaluation.

```
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

En appliquant ce programme à n'importe quel fichier XML, on obtient le résultat suivant qui est un document XHTML valide.

```
<?xml version="1.0" encoding="utf-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello world</h1>
  </body>
</html>
```

### 8.3 Modèle de traitement

L'arbre source du document XML est transformé en un arbre résultat. Cet arbre résultat est obtenu en appliquant récursivement des règles de transformation sur les nœuds à partir de la racine de l'arbre. Pour chaque nœud, la règle de plus grande priorité est appliquée.

Un programme (ou feuille de style) XSLT est en fait constitué de règles de transformation. Chaque règle déclare sur quels nœuds elle est susceptible d'être appliquée.

### 8.4 Entête

La programme souvent appelé *feuille de style* est entièrement inclus dans l'élément `xsl:stylesheet` ou de façon équivalente `xsl:transform`. L'élément `xsl:output` permet de contrôler le format du document résultat. Son attribut `method` qui peut prendre les valeurs `xml`, `xhtml`, `html` et `text` indique le type de document résultat produit. Ses attributs `encoding`, `doctype-public`, `doctype-system` précisent respectivement l'encodage du document, le FPI et l'URL de la DTD. Un exemple typique d'utilisation est le suivant.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml"
  version="2.0">
<xsl:output method="xml"
  encoding="iso-8859-1"
  doctype-public="-//W3C//DTD XHTML 1.1//EN"
  doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
  indent="yes"/>
```

### 8.5 Définition et application de règles

`xsl:template`

Déclaration d'une règle. L'attribut `match` définit le contexte sur lequel la règle s'applique. Le contenu de l'élément est le fragment d'arbre à insérer dans l'arbre résultat.

`xsl:apply-templates`

Application des règles sur les nœuds désignés par le l'attribut `select`.

`xsd:anyType`, `xsd:anySimpleType` et `xsd:anyAtomicType`  
Types racine de la hiérarchie  
`xsd:untyped` et `xsd:untypedAtomic`  
Nœud et valeur atomique non typé

### 7.9.2 Opérateurs

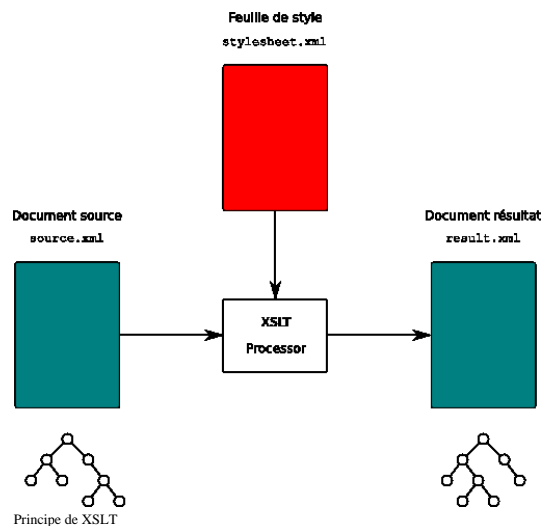
Le tableau suivant récapitule les principaux opérateurs de XPath.

Opérateur	Action	Syntaxe	Exemples
,	Concaténation de listes	E1,E2	1,'Two',3.14,true()
for	Itération	for \$i in E1 return E2	for \$i in 1 to 5 return \$i * \$i
some	Quantification existentielle	some \$i in E1 satisfies E2	
every	Quantification universelle	every \$i in E1 satisfies E2	
if	Test	if (E1) then E2 else E3	if (\$x > 0) then \$x else 0
/	Enchaînement	E1/E2	
[ ]	Prédicat	E1[E2]	chapter[count(section) > 1]
and or not	Opérations logiques	E1 or E2	
to	Intervalle	E1 to E2	1 to 5
eq ne lt le gt ge	Comparaisons de valeurs atomiques	E1 eq E2	\$x lt \$y
= != < <= > >=	Comparaisons générales	E1 = E2	\$x < \$y
<< is >>	Comparaisons de nœuds	E1 is E2	
+ * - div idiv	Opérations arithmétiques	E1 + E2	\$price * \$qty
union intersection except	Opérations sur les listes de nœuds	E1   E2	/ *
instance of cast as castable as treat as	Changements de type	E1 instance of type	\$x instance of xsd:string

## 8 Programmation XSL

Le cours est essentiellement basé sur des exemples. Les différentes constructions du langage XSLT sont illustrées par des fragments de programmes extraits des exemples.

### 8.1 Principe



### 8.2 Premier programme : Hello, World!

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml"
  version="2.0">
<xsl:template match="/*">
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello world</h1>
```

2002  
Eyrrolles  
2-212-11082-0

Designing with web standards  
Jeffrey Zeldman  
2003  
New Riders  
0-7357-1201-8

Ce résultat s'explique par la présence de règles par défaut qui sont les suivantes.

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

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

<xsl:template match="processing-instruction()|comment()" *
  </xsl:template>
```

La règle par défaut des attributs est d'afficher leur valeur. La valeur des attributs n'apparaissent pas dans le résultat ci-dessus car cette règle par défaut pour les attributs n'est pas invoquée. En effet, la valeur par défaut de l'attribut `select` de l'élément `apply-templates` est `node()` qui ne sélectionne pas les attributs.

## 8.7 Construction du résultat

`xsl:text`  
Insertion de texte brut.

`xsl:value-of`  
Insertion sous la forme chaîne de caractères des nœuds désignés par l'attribut `select`.

`xsl:copy`  
Copie à un niveau (appelée aussi copie superficielle)

`xsl:copy-of`  
Copie récursive (appelée aussi copie profonde)

`xsl:element`  
Construction d'un élément dont le nom est donné par l'attribut `name`.

`xsl:attribute`  
Ajout d'un attribut dont le nom est donné par l'attribut `name`.

`xsl:number`  
Numérotation d'éléments.

### 8.7.1 Insertion de texte

L'élément `xsl:text` utilise son contenu pour créer un nœud texte dans le document final. Les caractères spéciaux '<', '>' et '&' sont automatiquement remplacés par les entités prédéfinies correspondantes si la valeur de l'attribut `method` de l'élément `output` n'est pas `text`.

```
<xsl:text>Caractères '&lt;', '&gt;' et '&amp;' et caractères accentués</xsl:text>
```

- 64 -

```
<xsl:element name="concat('new-', $var)" *
  </element>
```

### 8.7.5 Ajout d'attribut

L'élément `xsl:attribute` permet d'ajouter un attribut à un élément. Le rôle de cette élément est similaire aux expressions XPath dans les attributs mais il offre plus de possibilités. Le nom de l'attribut est déterminé par l'attribut `name` qui peut contenir une expression XPath. Le nom de l'attribut peut donc est calculé dynamiquement. La valeur est donnée soit par l'attribut `select` de `xsl:attribute` soit par le contenu de cet élément.

Le fragment de feuille style suivant construit un élément `tr` avec en plus un attribut `bgcolor` si la position de l'élément traité est paire (`test position() mod 2 = 0`).

```
<tr>
  <xsl:if test="position() mod 2 = 0">
    <xsl:attribute name="bgcolor"#ffffff/xsl:attribute>
  </xsl:if>
  ...
</tr>
```

### 8.7.6 Insertion d'un numéro : `xsl:number`

Le rôle de l'élément `xsl:number` est double. Sa première fonction est de créer un entier ou une liste d'entiers pour numéroté un élément. La seconde fonction est de formater cet entier ou cette liste. La seconde fonction est plutôt adaptée à la numérotation. Pour formater un nombre de façon précise, il est préférable d'utiliser la fonction `format-number` de XPath.

#### 8.7.6.1 Formats

La fonction de formatage est relativement simple. L'attribut `format` de `xsl:number` contient une chaîne formée d'un *préfixe*, d'un indicateur de format et d'un *suffixe*. Le préfixe et le suffixe doivent être formés de caractères non alphanumériques. Ils sont copiés sans changement. L'indicateur de format est remplacé par l'entier. Le tableau suivant récapitule les différents formats possibles. Le format par défaut est 1.

Format	Résultat
1	1, 2, 3, ..., 9, 10, 11, ...
01	01, 02, 03, ..., 09, 10, 11, ...
a	a, b, c, ..., z, aa, ab, ...
A	A, B, C, ..., Z, AA, AB, ...
i	i, ii, iii, iv, v, vi, vii, viii, ix, x, xi, ...
I	I, II, III, IV, V, VI, VII, VIII, IX, X, XI, ...

Il existe aussi des formats `w`, `W` et `Ww` permettant d'écrire le nombre en toutes lettres. L'attribut `lang` qui prend les mêmes valeurs que l'attribut `xml:lang` spécifie la langue dans laquelle sont écrits les nombres. Il semblerait que l'attribut `lang` ne soit pas pris en compte.

- 66 -

`xsl:call-template`  
Application de la règle nommée par l'attribut `name`.

Le fragment de programme suivant définit une règle grâce à `xsl:template`. La valeur de l'attribut `match` vaut `/*` et indique donc que la règle s'applique uniquement à la racine de l'arbre. La racine de l'arbre résultat est alors le fragment XHTML contenu dans `xsl:template`. Comme ce fragment ne contient pas d'autres directives XSLT, le traitement de l'arbre source s'arrête et le document résultat est réduit à ce fragment.

```
<xsl:template match="/*" *
  <html>
    <head>
      <title>Hello, World!</title>
    </head>
    <body>
      <h1>Hello, world!</h1>
    </body>
  </html>
</xsl:template>
```

La règle ci-dessous s'applique à tous les éléments de nom `author`, `year` ou `publisher`. Cet élément du document source est remplacé dans le document résultat par le traitement récursif de ses éléments fils suivi d'une virgule. Le traitement des fils est provoqué par `xsl:apply-templates` dont la valeur par défaut de l'attribut `match` est `child::*`. La virgule est insérée par l'élément `xsl:text`.

```
<xsl:template match="author|year|publisher" *
  <xsl:apply-templates/><xsl:text>,</xsl:text>
</xsl:template>
```

## 8.6 Règles par défaut

Programme minimal

```
<?xml version="1.0" encoding="us-ascii"?>
<!-- Feuille de style minimale -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0"/>
```

En appliquant ce programme minimal au fichier `bibliography.xml` on obtient le résultat suivant.

```
<?xml version="1.0" encoding="UTF-8"?>
  XML langage et applications
  Alain Michard
  2001
  Eyrrolles
  2-212-09206-7
  http://www.editions-eyrolles/livres/michard/

  XML by Example
  Benoît Marchal
  2000
  Macmillan Computer Publishing
  0-7897-2242-9

  XSLT fondamental
  Philippe Drix
```

- 63 -

### 8.7.2 Expression XPath en attribut

Il est possible d'insérer directement dans un attribut la valeur d'une expression XPath. L'expression doit être délimitée dans l'attribut par des accolades '{' et '}'. À l'exécution, l'expression est évaluée et le résultat remplace dans l'attribut l'expression et les accolades qui l'entourent. Un même attribut peut contenir un mélange de texte et de plusieurs expressions XPath comme dans l'exemple `<p ...>` ci-dessous.

```
<body background-color="{ $color }">
  ...
  <p style="{ $property } : { $value }">
```

Cette syntaxe est beaucoup plus concise que l'utilisation classique de l'élément `xsl:attribute` pour ajouter un attribut (cf. ajout d'attribut).

```
<body>
  <xsl:attribute name="background-color" select="$color"/>
```

### 8.7.3 Insertion d'une valeur : `xsl:value-of`

L'élément `xsl:value-of` crée un nœud texte dont le contenu est calculé. L'attribut `select` doit contenir une expression XPath et le texte est alors la valeur textuelle des éléments sélectionnés par l'attribut.

Le fragment XSL suivant insère dans le document résultat un attribut `id` dont la valeur est justement la valeur de l'attribut `id` du nœud courant.

```
<xsl:attribute name="id">
  <xsl:value-of select="@id"/>
</xsl:attribute>
```

Quelques exemples d'utilisation de `xsl:value-of`.

```
<xsl:value-of select="."/>
<xsl:value-of select="generate-id()"/>
<xsl:value-of select="key('idchapter', @idref)/title"/>
<xsl:value-of select="ancestor-or-self::p[@xml:lang][1]@xml:lang"/>
```

### 8.7.4 Ajout d'élément

Tout élément contenu dans un élément `template` et n'appartenant pas à l'espace de nom `xsl` des feuilles de style est copié à l'identique dans le document résultat. Ceci permet d'ajouter facilement des éléments dont les noms sont fixes.

L'élément `xsl:element` permet également d'ajouter un élément dans le document résultat. Le nom de l'élément peut être calculé dynamiquement. Le nom est en effet déterminé par l'attribut `name` de `xsl:element`. Cet attribut peut contenir une expression XPath dont l'évaluation fournit le nom de l'élément.

Dans l'exemple suivant, le nom de l'élément est obtenu en concaténant la chaîne `'new'` avec la valeur de la variable `var`.

- 65 -



```

<xsl:template match="/">
  <xsl:variable name="list" as="xsd:integer">
    <xsl:perform-sort>
      <xsl:sort data-type="number" order="ascending"/>
      <xsl:sequence select="(3, 1, 5, 0)"/>
    </xsl:perform-sort>
  </xsl:variable>
  <!-- Produit 0,1,3,5 -->
  <xsl:value-of select="$list" separator=" "/>
</xsl:template>
</xsl:stylesheet>

```

## 8.10 Variables et paramètres

Le langage XSLT permet l'utilisation de variables pouvant stocker des valeurs. Les valeurs possibles comprennent une valeur atomique, un nœud ou une suite de ces valeurs, c'est-à-dire toutes les valeurs des expressions XPath. Les variables peuvent être utilisées dans les expressions XPath.

Le langage XSLT distingue les variables des paramètres. Les variables servent à stocker des valeurs intermédiaires alors que les paramètres servent à transmettre des valeurs aux règles. Les variables sont introduites par l'élément `xsl:variable` et les paramètres par l'élément `xsl:param`.

La valeur de la variable est fixée au moment de sa déclaration par l'élément `xsl:variable` et ne peut plus changer ensuite. Les variables ne sont donc pas vraiment *variable*. Il s'agit d'objets *non mutables* dans la terminologie des langages de programmation. La portée de la variable est l'élément XSLT qui la contient. Les variables dont la déclaration est fille de l'élément `xsl:stylesheet` sont donc globales.

L'attribut `name` détermine le nom de la variable. La valeur est donnée soit par une expression XPath dans l'attribut `select` soit directement dans le contenu de l'élément `xsl:variable`. Un attribut optionnel `as` peut spécifier le type de la valeur.

```

<xsl:variable name="squares" as="xsd:integer">
  <xsl:for-each select="1 to 5">
    <xsl:sequence select=" * ."/>
  </xsl:for-each>
</xsl:variable>
<xsl:variable name="cubes" as="xsd:integer">
  <xsl:for-each select="for $i in 1 to 5 return $i * $i * $i"/>
</xsl:variable>

```

Une variable peut apparaître dans une expression XPath en étant précédée du caractère '\$' comme dans l'exemple suivant.

```

<xsl:value-of select="$squares"/>

```

La valeur par défaut d'un paramètre est fixée au moment de sa déclaration par l'élément `xsl:param`. Parmi les paramètres, le langage XSLT distingue les paramètres globaux et les paramètres (locaux) des règles. Les paramètres globaux sont déclarés par un élément `xsl:param` fils de l'élément `xsl:stylesheet`. Leur valeur est fixée au moment de l'appel au processeur XSLT et ils peuvent être utilisés dans toute la feuille de style. La syntaxe pour fixer la valeur d'un paramètre global dépend du processeur XSLT.

La déclaration d'un paramètre d'une règle est réalisée par un élément `xsl:param` fils de `xsl:template`. Les déclarations de paramètres doivent être les premiers fils. Le passage d'une valeur en paramètre est réalisée par un élément `xsl:with-param` fils de

```

</xsl:when>
<xsl:when test="$n = 1">
  <xsl:number format="1" value="1"/>
</xsl:when>
<xsl:otherwise>
  <xsl:variable name="x">
    <xsl:call-template name="fibonacci">
      <!-- Espace nécessaire avant le '-' -->
      <xsl:with-param name="n" select="$n - 1"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="y">
    <xsl:call-template name="fibonacci">
      <!-- Espace nécessaire avant le '-' -->
      <xsl:with-param name="n" select="$n - 2"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:number format="1" value="$x+$y"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>

```

### 8.10.2 Paramètres tunnel

Il est parfois fastidieux de transmettre systématiquement des paramètres aux règles appliquées. Les *paramètres tunnel* sont transmis automatiquement. En revanche, ils ne peuvent être utilisés que dans les règles qui les déclarent.

Dans l'exemple suivant, la règle pour la racine applique une règle au nœud `text` avec les paramètres `tunnel` et `nelunt`. La règle appliquée reçoit ces deux paramètres et applique à nouveau des règles à ses fils textuels. Le paramètre `tunnel` est transmis implicitement à ces nouvelles règles car il a été déclaré avec l'attribut `tunnel` valant `yes`. La règle appliquée aux nœuds textuels déclare les paramètres `tunnel` et `nelunt`. La valeur de `tunnel` est effectivement la valeur donnée au départ à ce paramètre. Au contraire, la valeur du paramètre `nelunt` est celle par défaut car il n'a pas été transmis.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:output method="text" encoding="iso-8859-1"/>
  <xsl:template match="/">
    <xsl:apply-templates select="text">
      <xsl:with-param name="tunnel" select="'tunnel' tunnel="yes"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="text">
    <xsl:apply-templates select="text()"/>
  </xsl:template>
  <xsl:template match="text()">
    <!-- L'attribut tunnel="yes" est nécessaire -->
    <xsl:param name="tunnel" select="'default' tunnel="yes"/>
    <xsl:param name="nelunt" select="'default'"/>
    <!-- Produit la valeur 'tunnel' fournie au départ -->
    <xsl:value-of select="$tunnel"/>
  </xsl:template>

```

```

<xsl:text>Section </xsl:text>
<xsl:number level="single" count="section"/>
</xsl:otherwise>
</xsl:choose>

```

### 8.8.3 Itération xsl:for-each

```

<xsl:for-each select="bibliography/book">
  <xsl:sort select="author" order="descending"/>
  <tr>
    <xsl:if test="position() mod 2 = 0">
      <xsl:attribute name="bgcolor" #ffffff</xsl:attribute>
    </xsl:if>
    <td><xsl:number value="position()" format="1"/></td>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="author"/></td>
    <td><xsl:value-of select="publisher"/></td>
    <td><xsl:value-of select="year"/></td>
  </tr>
</xsl:for-each>

```

## 8.9 Tris

L'élément `xsl:sort` permet de trier des éléments avant de les traiter. L'élément `xsl:sort` doit être le premier fils des éléments `apply-templates`, `call-template`, `for-each` ou `for-each-group`. Le tri s'applique à tous les éléments sélectionnés (par l'attribut `select`) de ces différents éléments.

Le fragment de feuille de style suivant permet par exemple de trier les éléments `book` par auteur par ordre croissant.

```

<xsl:apply-templates select="bibliography/book">
  <xsl:sort select="author" order="ascending"/>
</xsl:apply-templates>

```

L'attribut `select` de `xsl:sort` détermine la clé du tri. L'attribut `data-type` qui peut prendre les valeurs `number` ou `text` spécifie comment les clés doivent être interprétées. Il est possible d'avoir plusieurs clés de tri en mettant plusieurs éléments `xsl:sort` comme dans l'exemple suivant. Les éléments `book` sont d'abord triés par auteur puis par année.

```

<xsl:apply-templates select="bibliography/book">
  <xsl:sort select="author" order="ascending"/>
  <xsl:sort select="year" order="descending"/>
</xsl:apply-templates>

```

### 8.9.1 Utilisation de xsl:perform-sort

L'élément `xsl:perform-sort` permet d'appliquer un tri à une suite quelconque d'éléments, en particulier avant de l'affecter à une variable. Ses fils doivent être un ou des éléments `xsl:sort` puis des éléments qui construisent la suite.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  version="2.0">
  <xsl:output method="text" encoding="iso-8859-1"/>

```

`xsl:templates` ou `xsl:call-template`. Comme pour `xsl:variable`, l'attribut `name` détermine le nom de la variable. La valeur est donnée soit par une expression XPath dans l'attribut `select` soit directement dans le contenu de l'élément `xsl:variable`. Un attribut optionnel `as` peut spécifier le type de la valeur.

Dans l'exemple suivant, la première règle (pour la racine /) applique la règle pour le fils `text` avec le paramètre `color` égal à `blue`. La valeur par défaut de ce paramètre est `black`.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="/">
    <xsl:apply-templates select="text">
      <!-- Valeur du paramètre pour l'appel -->
      <xsl:with-param name="color" select="'blue'"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="text">
    <!-- Déclaration du paramètre avec 'black' comme valeur par défaut -->
    <xsl:param name="color" select="'black'"/>
    <p style="color:{color};"><xsl:value-of select="."/;></p>
  </xsl:template>
</xsl:stylesheet>

```

### 8.10.1 Récursivité

La feuille de style suivant réalise le calcul récursif d'un nombre  $F_n$  de la suite de Fibonacci définie par récurrence par  $F_0 = 1, F_1 = 1$  et  $F_{n+2} = F_{n+1} + F_n$  pour  $n \geq 0$ . L'entier  $n$  est fourni par le fichier XML en entrée.

```

<?xml version="1.0" encoding="us-ascii"?>
<!-- Calcul des nombres de Fibonacci -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:output method="html" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Nombres de Fibonacci</title>
      </head>
      <body>
        <h1>
          <xsl:variable name="n" select="n/text()"/>
          <xsl:text>F</xsl:text>
          <sub><xsl:number format="1" value="$n"/></sub>
          <xsl:text> = </xsl:text>
          <xsl:call-template name="fibonacci">
            <xsl:with-param name="n" select="$n"/>
          </xsl:call-template>
        </h1>
      </body>
    </html>
  </xsl:template>

  <xsl:template name="fibonacci">
    <xsl:param name="n" select="1"/>
    <xsl:choose>
      <xsl:when test="$n = 0">
        <xsl:number format="1" value="1"/>

```

```

<!-- Application avec le mode #default -->
<xsl:apply-templates select="..." />
...
<!-- Application avec le mode test -->
<xsl:apply-templates select="..." mode="test" />
...
<!-- Application avec le mode déjà en cours -->
<xsl:apply-templates select="..." mode="#current" />
...

```

Dans l'exemple suivant, le nœud `text` est traité d'abord dans le mode par défaut `#default` puis dans le mode `test`. Dans chacun de ces traitements, ses fils textuels sont traités d'abord dans le mode par défaut puis dans son propre mode de traitement. Les fils textuels sont donc au final traités trois fois dans le mode par défaut et une fois dans le mode `test`.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:output method="text" encoding="iso-8859-1"/>
  <xsl:template match="*" />
    <xsl:apply-templates select="text" />
    <xsl:apply-templates select="text" mode="test" />
  </xsl:template>
  <xsl:template match="text" mode="#default test">
    <xsl:apply-templates select="text()" />
    <xsl:apply-templates select="text()" mode="#current" />
  </xsl:template>
  <xsl:template match="text()">
    <xsl:text>Mode #default: </xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>
  <xsl:template match="text()" mode="test">
    <xsl:text>Mode test: </xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:template>
</xsl:stylesheet>

```

L'exemple suivant illustre une utilisation classique des modes. Le document est traité une première fois en mode `toc` pour en extraire une table des matières et une seconde fois pour créer le corps du document proprement dit.

```

<body>
  <!-- Fabrication de la table des matières -->
  <xsl:apply-templates mode="toc" />
  <!-- Fabrication du corps du document -->
  <xsl:apply-templates />
</body>
...
<!-- Règles pour la table des matières -->
<xsl:template match="book" mode="toc">
  <h1>Table des matières</h1>
  <ul><xsl:apply-templates mode="toc" /></ul>
</xsl:template>

```

```

</body>
</html>
</xsl:template>
<xsl:template match="city">
  <li><xsl:value-of select="."/></li>
</xsl:template>
</xsl:stylesheet>

```

L'ajout d'un élément `xsl:sort` comme fils de l'élément `<xsl:apply-templates select="document(@href)/cities/city"/>` permet de trier les éléments à l'intérieur d'un des documents référencés. Pour trier les éléments globalement, il faut supprimer deux itérations imbriquées et la remplacer par une seule itération comme dans l'exemple suivant.

```

<xsl:apply-templates select="document(files/file/@href)/cities/city">
  <xsl:sort select="."/>
</xsl:apply-templates>

```

Dans l'expression XPath `document(files/file/@href)/cities/city`, on utilise le fait que la fonction `document()` prend en paramètre une liste de noms de fichiers et qu'elle retourne l'ensemble des contenus des fichiers.

### 8.10.3 Récapitulatif

```

xsl:variable
  Création d'une variable.
xsl:param
  Création d'un paramètre.
xsl:with-param
  Instantiation d'un paramètre dans un appel réalisé par xsl:apply-templates ou call-template.

```

### 8.11 Modes

Il est fréquent qu'une feuille de style traite plusieurs fois les mêmes nœuds du document d'entrée pour en extraire divers fragments. Ces différents traitements peuvent être distingués par des *modes*. Chaque règle de la feuille de style déclare pour quel mode elle s'applique avec l'attribut `mode` de l'élément `xsl:template`. En parallèle, chaque application de règles avec `xsl:apply-templates` spécifie un mode avec un attribut `mode`.

Chaque mode est identifié par un identificateur. Il existe en outre les valeurs particulières `#default`, `#all` et `#current` qui peuvent apparaître dans les valeurs des attributs `mode`.

La valeur de l'attribut `mode` de l'élément `xsl:template` est soit la valeur `#all` soit une liste de modes, y compris `#default`, séparés par des espaces. La valeur `#current` n'a pas de sens dans ce contexte et ne peut pas apparaître. La valeur par défaut est bien sûr `#default`.

```

<!-- Règle applicable avec le mode #default -->
<xsl:template match="*" />
...
<!-- Règle applicable avec le mode test -->
<xsl:template match="*" mode="test" />
...
<!-- Règle applicable avec les modes #default foo et bar -->
<xsl:template match="*" mode="#default foo bar" />
...
<!-- Règle applicable avec tous les modes -->
<xsl:template match="*" mode="#all" />
...

```

La valeur de l'attribut `mode` de l'élément `xsl:apply-templates` est soit `#default` soit `#current` soit le nom d'un seul mode. La valeur `#all` n'a pas de sens dans ce contexte et ne peut pas apparaître. La valeur `#current` permet d'appliquer des règles avec le même mode que celui de la règle en cours. La valeur par défaut est encore `#default`.

### 8.12 Indexation

Afin de pouvoir accéder efficacement à des nœuds d'un document XML, il est possible de créer des index. L'élément `xsl:key` crée un index. L'attribut `name` fixe le nom de l'index pour son utilisation. L'expression XPath de l'attribut `match` détermine les nœuds qui sont indexés alors que l'expression XPath de l'attribut `use` spécifie la clé d'indexation. L'élément `xsl:key` doit être un fils de l'élément `xsl:stylesheet`.

```

<xsl:key name="idchapter" match="chapter" use="@id"/>

```

La fonction `key` de XPath permet de retrouver un nœud en utilisant un index créé par `xsl:key`. Le premier paramètre est le nom de l'index et le second est la valeur de la clé.

```

<xsl:value-of select="key('idchapter', $node/@idref)/title"/>

```

### 8.13 Documents multiples

La fonction `document()` permet de lire et de manipuler un document XML contenu dans un autre fichier. Le nom du fichier contenant le document est fourni en paramètre à la fonction. Le résultat peut être stocké dans une variable ou être utilisé directement.

Le document suivant référence deux autres documents `europa.xml` et `states.xml`.

```

<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<files>
  <file href="europa.xml"/>
  <file href="states.xml"/>
</files>

```

Le document `europa.xml` est le suivant. Le document `states.xml` est similaire avec des villes américaines.

```

<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<!-- Time-stamp: "europa.xml 24 Feb 2009 15:03:19" -->
<cities>
  <city>Berlin</city>
  <city>Paris</city>
</cities>

```

La feuille de style XSL suivante permet de collecter les différentes villes des documents référencés par le premier document pour en faire une liste unique.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:template match="*" />
  <html>
    <head>
      <title>Liste de villes</title>
    </head>
    <body>
      <h1>Liste de villes</h1>
      <ul>
        <xsl:for-each select="files/file">
          <xsl:apply-templates select="document(@href)/cities/city"/>
        </xsl:for-each>
      </ul>
    </body>
  </html>

```

# 10 Feuilles de style CSS

## 10.1 Principe

Le principe des feuilles de style CSS est de séparer le *contenu* de la *forme*. Elles sont beaucoup utilisées avec HTML et XHTML mais elles peuvent aussi l'être avec XML (cf. exemple avec la bibliographie).

Les rôles des XSLT et CSS sont différents et même complémentaires. Le rôle de XSLT est de transformer le document source en un autre document résultat, XHTML par exemple. Il s'agit donc d'agir sur le contenu et en particulier sur la structure de ce contenu. Au contraire, CSS ne permet pas (ou très peu) de changer le contenu. Il peut uniquement intervenir sur la présentation. Une bonne solution est d'utiliser XSLT et CSS de pair. XSLT produit un document XHTML dont la présentation est contrôlée par une feuille de style CSS.

## 10.2 Règles

Une feuille de style est formée de règles qui ont la forme suivante. Les espaces et les retours à la ligne jouent uniquement un rôle de séparateurs. L'indentation de la feuille de style est donc libre.

```
(selector) {
  (property1): (value1);
  (property2): (value2);
  ...
  (propertyN): (valueN);
}
```

Le sélecteur `selector` détermine quels sont les éléments auxquels s'applique la règle. Les propriétés `property1`, `property2`, ..., `propertyN` de tous ces éléments prendront les valeurs respectives `value1`, `value2`, ..., `valueN`. Chaque valeur est séparée du nom de la propriété par le caractère `:`.

Des commentaires peuvent être mis dans les feuilles de styles en dehors ou dans les règles en utilisant une syntaxe identique à celle du langage C. Ils commencent par les deux caractères `/*` et se terminent par les deux caractères `*/`. Le caractère `'` sépare les couples propriété/valeur de la règle. Il n'est donc pas indispensable après le dernier couple de la règle.

L'exemple ci-dessous est la feuille de style utilisée par cette page-ci.

```
/* Fond blanc */
body {
  background-color: white;
}

/* Equations et figures centrées */
p.equation, p.figure {
  text-align: center;
}

/* Zone de code : fond jaune clair, bordure noire et marges */
pre {
  background-color: #ffffcc;
}
```

- 80 -

L'élément unique dont l'attribut `id` a la valeur `ident` peuvent être sélectionnés par le sélecteur `#ident` où la valeur de l'attribut est précédée d'un dièse `#`.

Le sélecteur `'*'` sélectionne tous les éléments. Dans l'exemple suivant, tous les éléments (c'est-à-dire le texte) seront de couleur bleue à l'exception des éléments `p` qui seront de couleur grise.

```
* { color: blue }
p { color: gray }
```

### 10.2.2.2 Pseudo-éléments et pseudo-classes

Certaines parties d'un document qui ne correspondent pas à un élément peuvent être sélectionnées par des *pseudo-éléments*. La première ligne et le premier caractère du contenu d'un élément `name` peuvent être désignés par `name:first-line` et `name:first-letter`.

```
p:first-line {
  text-indent: 15pt;
}
```

Le pseudo-élément `:first-child` permet en outre de sélectionner le premier fils d'un élément. Dans l'exemple suivant, la règle s'applique uniquement à la première entrée d'une liste.

```
li:first-child {
  color: blue;
}
```

Les pseudo-éléments `:before` et `:after` et la propriété `content` permettent d'ajouter du contenu avant et après un élément.

```
li:before {
  content: "[" counter(c) "];";
  counter-increment: c;
}
```

Les pseudo-classes `:link`, `:visited`, `:hover` et `:active` s'appliquent à l'élément `a` et permettent de sélectionner les liens, les liens déjà traversés, les liens sous le curseur et les liens activés.

```
a:link { color: blue; }
a:visited { color: magenta; }
a:hover { color: red; }
a:active { color: red; }
```

La pseudo-classe `:focus` permet de sélectionner l'entrée d'un formulaire qui a le focus.

### 10.2.2.3 Sélection sur les attributs

Un sélecteur peut aussi prendre en compte la présence d'attributs et leurs valeurs. Un ou plusieurs prédicats portant sur les attributs sont ajoutés à un sélecteur élémentaire. Chacun des prédicats est ajouté après le sélecteur entre crochet `'[ ' et ' ]'`. Les différents prédicats possibles sont les suivants.

`{(att)}`

L'élément est sélectionné s'il a un attribut `att` quelque soit sa valeur.

`{(att)=(value)}`

L'élément est sélectionné s'il a un attribut `att` dont la valeur est exactement la chaîne `value`.

- 82 -

# 9 XSL-FO

XSL-FO est un *dialecte* de XML permettant de décrire le rendu de documents. Un document XSL-FO contient le contenu même du document ainsi que toutes les indications de rendu. Il s'apparente donc à un mélange de HTML et CSS avec une syntaxe XML mais il est plus destiné à l'impression qu'au rendu sur écran. Le langage XSL-FO est très verbeux et donc peu adapté à l'écriture directe de documents. Il est plutôt conçu pour des documents produits par des feuilles de style XSL.

## 9.1 Premier exemple

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Hello, World! en XSL-FO -->
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <!-- Modèle de pages -->
  <fo:layout-master-set>
    <fo:simple-page-master master-name="A4"
      page-width="210mm" page-height="297mm"
      margin="1cm">
      <!-- Région principale -->
      <fo:region-body margin="2cm"/>
      <!-- Tête de page aka header -->
      <fo:region-before extent="1cm"/>
      <!-- Pied de page aka footer -->
      <fo:region-after extent="1cm"/>
    </fo:simple-page-master>
  </fo:layout-master-set>
  <!-- Contenus -->
  <fo:page-sequence master-reference="A4">
    <!-- Contenu de la tête de page -->
    <fo:static-content flow-name="xsl-region-before">
      <fo:block text-align="center">XSL-FO Hello, World! example</fo:block>
    </fo:static-content>
    <!-- Contenu du pied de page : numéro de la page -->
    <fo:static-content flow-name="xsl-region-after">
      <fo:block text-align="center"><fo:page-number/> </fo:block>
    </fo:static-content>
    <!-- Contenu de la partie centrale -->
    <fo:flow flow-name="xsl-region-body">
      <fo:block text-align="center"
        font="32pt Times"
        border="black solid thick">Hello, world!</fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Le document précédent peut être traité par un programme comme `Fop` pour produire un document PDF ou un document PNG.

## 9.2 Structure globale

Un document XSL-FO est constitué de deux parties principales. La première partie contenue dans l'élément `fo:layout-master-set` contient des modèles de pages. Ces modèles décrivent la mise en page du contenu. La seconde partie contenue dans l'élément `fo:page-sequence` donne le contenu structuré en blocs.

- 79 -

```
border: 1px solid black;
margin: 10px;
padding: 5px;
}
```

## 10.2.1 Média

Les règles d'une feuille de style peuvent dépendre du média utilisé pour rendre le document. Par *média*, on entend le support physique servant à matérialiser le document. Il peut s'agir d'un écran d'ordinateur, d'un projecteur, de papier. La syntaxe est la suivante.

```
@media (medium) {
  /* Règles pour le média */
  ...
}
```

Les principales valeurs possible pour `medium` sont `screen` pour un écran d'ordinateur, `print` pour du papier et `projection` pour un projecteur.

## 10.2.2 Sélecteurs

Un sélecteur prend la forme générale suivante. Il est constitué par une suite de sélecteurs séparés par des virgules. Il sélectionne alors tous les éléments sélectionnés par chacun des sélecteurs individuels.

```
(selector1), (selector1), ... (selectorN)
```

### 10.2.2.1 Sélecteurs élémentaires

La forme la plus simple d'un sélecteur est le nom `name` d'un élément comme `h1`, `p` ou encore `pre`. Tous les éléments de nom `name` sont alors sélectionnés. Dans l'exemple suivant, le fond de l'élément `body`, c'est-à-dire de tout le document, est blanc.

```
body {
  background-color: white;
}
```

Tous les éléments dont l'attribut `class` contient la chaîne `classname` peuvent être sélectionnés par le sélecteur `.classname` où la valeur de l'attribut est précédée d'un point `'.'`.

L'attribut `class` d'un élément peut contenir plusieurs chaînes séparées par des espaces comme dans l'exemple suivant. Un sélecteur de forme `.classname` sélectionne un élément si la chaîne `classname` est une des chaînes de la valeur de l'attribut `class`.

```
<p class="numbered equation"> ... </p>
```

Cette forme de sélecteur peut être combinée avec le nom `name` d'un élément pour former un sélecteur `name.classname` qui sélectionne tous les éléments de nom `name` dont l'attribut `class` contient la chaîne `classname`. Dans l'exemple suivant, tous les éléments `p` de classe `equation` ou `figure` auront leur texte centré.

```
p.equation, p.figure {
  text-align: center;
}
```

- 81 -



# 11 SVG (Scalar Vector Graphics)

SVG est un dialecte de XML pour le dessin vectoriel. Ce format permet de définir les éléments graphiques de manière standard.

## 10.7 Principales propriétés

### 10.7.1 Polices et texte

Propriété	Valeur
color	couleur
font	combinaison des propriétés font-*
font-family	nom de police, serif, sans-serif, cursive, fantasy ou monospace
font-style	normal, italic, oblique
font-variant	normal, small-caps
font-weight	normal, bold, bolder ou lighter
font-size	dimension
text-decoration	none, underline, overline, line-through ou blink
text-transform	none, capitalize, uppercase ou lowercase
word-spacing	normal ou dimension
letter-spacing	normal ou dimension
vertical-align	baseline, sub, super, top, text-top, middle, bottom, text-bottom ou pourcentage
text-align	left, right, center ou justify
text-indent	dimension ou pourcentage
line-height	normal, facteur, dimension ou pourcentage
white-space	normal, pre, nowrap, pre-wrap ou pre-line
content	chaîne de caractères

### 10.7.2 Fond

Propriété	Valeur
background	combinaison des propriétés background-*
background-attachment	scroll ou fixed
background-color	couleur
background-image	image
background-position	pourcentage, dimension ou (top, center ou bottom) et (left, right ou center)
background-repeat	no-repeat, repeat-x, repeat-y ou repeat

```
/* Virgule avant l'URL si elle est présente */
url:before {
  content: ", ";
}
```

## 10.6 Attachement de règles de style

Les règles de style concernant un document peuvent être placées à différents endroits. Elles peuvent d'abord être mises dans un fichier externe dont l'extension est généralement .css. Le document fait alors référence à cette feuille de style. Les règles de style peuvent aussi être incluses directement dans le document. Pour un document XHTML, elles peuvent se placer dans un élément style de l'entête. Elles peuvent aussi être ajoutées dans un attribut style de n'importe quel élément.

### 10.6.1 Référence à un document externe

La façon de référencer une feuille de style externe dépend du format du document. Pour un document XML, il faut utiliser une instruction de traitement xml-stylesheet. Pour un document XHTML, il faut utiliser un élément link dans l'entête.

#### 10.6.1.1 Dans un document XML

```
<?xml-stylesheet type="text/css" href="bibliography.css" ?>
```

#### 10.6.1.2 Dans un document XHTML

```
<head>
  <title>Titre de la page</title>
  <link href="style.css" rel="stylesheet" type="text/css" />
</head>
```

### 10.6.2 Inclusion dans l'entête du fichier XHTML

Les règles de style peuvent directement incluses dans un élément style de l'entête, c'est-à-dire contenu dans l'élément head. Il est préférable de protéger ces règles par des balises <!-- et --> de commentaires.

```
<head>
  <title>Titre de la page</title>
  <style type="text/css"><!--
  /* Contenu de la feuille de style */
  /* Ce contenu est inclus dans un commentaire XML */
  body { background-color: white; }
  --></style>
</head>
```

### 10.6.3 Inclusion dans un attribut d'un élément

Chaque élément peut aussi avoir un attribut style qui contient uniquement des couples propriété/valeur séparés par des virgules. Les sélecteurs sont inutiles puisque ces règles s'appliquent implicitement à cet élément.

## 11.1 Un premier exemple



Rendu PNG et SVG

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  width="200" height="100" version="1.0">
  <defs>
    <path id="textpath"
      d="M 15,80 C 35,65 40,65 50,65 C 60,65 80,75 95,75
        C 110,75 135,60 150,60 C 165,60 170,60 185,65"
      style="fill:none;stroke:black;" />
  </defs>
  <text style="font-family:Verdana; font-size:28;
    font-weight:bold; fill:red">
    <textPath xlink:href="#textpath">
      Hello, SVG!
    </textPath>
  </text>
  <use xlink:href="#textpath" y="10"/>
</svg>
```

## 11.2 Éléments de dessins

### 11.2.1 Formes élémentaires

### 10.7.3 Boîtes et positionnement

Propriété	Valeur
width	auto, dimension ou pourcentage
height	auto, dimension ou pourcentage
padding	dimension ou pourcentage
padding-top	
padding-right	
padding-bottom	
padding-left	
border-style	none, dotted, dashed, solid, double, groove, ridge, inset ou outset
border-width	medium, thin, thick ou une dimension
border-color	couleur
margin	auto, dimension ou pourcentage
margin-top	
margin-right	
margin-bottom	
margin-left	
position	static, relative, absolute ou fixed
top	auto, dimension ou pourcentage
right	
bottom	
left	
float	none, left ou right
clear	none, left, right ou both
overflow	visible, hidden, scroll ou auto
visibility	visible ou hidden

### 10.7.4 Listes

Propriété	Valeur
list-style	Combinaison des trois propriétés list-style-*
list-style-image	image
list-style-position	outside ou inside
list-style-type	none, disc, circle, square, decimal, upper-Roman, lower-Roman, upper-alpha ou lower-alpha

Ligne verticale <path d="M x1 y1 V y2"/>		
Ligne <path d="M x1 y1 L x2 y2"/>		
Courbe de Bézier quadratique <path d="M x1 y1 Q cx cy x2 y2"/>		
Courbe de Bézier quadratique avec partage de point de contrôle <path d="M x1 y1 Q cx cy x2 y2 T x3 y3"/>		
Courbe de Bézier cubique <path d="M x1 y1 C cx1 cy1 cx2 cy2 x2 y2"/>		
Courbe de Bézier cubique avec partage de point de contrôle <path d="M x1 y1 C cx1 cy1 cx2 cy2 x2 y2 S cx3 cy3 x3 y3"/>		
Fermeture du chemin par une ligne <path d="M x1 y1 Q cx cy x2 y2 Z"/>		

Pour une introduction aux courbes de Bézier, on peut se référer à cette page.

Style dans un attribut style

### 11.4.2 Attributs spécifiques

```
<svg width="200" height="100">
  <rect x="10" y="10" width="180" height="80" stroke="black" fill="none"/>
  <ellipse cx="100" cy="50" rx="90" ry="40" stroke="black"/>
  <ellipse class="circle" cx="100" cy="50" rx="40" ry="40" fill="red"/>
</svg>
```



Style dans des attributs spécifiques stroke et fill

### 11.4.3 Élément style

```
<svg width="200" height="100">
  <style type="text/css">
    rect { stroke: red; }
    ellipse { fill: red; }
    ellipse.circle { fill: white; }
  </style>
  <rect x="10" y="10" width="180" height="80"/>
  <ellipse cx="100" cy="50" rx="90" ry="40" style="stroke:black;fill:red"/>
  <ellipse class="circle" cx="100" cy="50" rx="40" ry="40"/>
</svg>
```



Style dans un élément style

### 11.4.4 Feuille de style attachée

```
<?xml-stylesheet href="stylesvg.css" type="text/css"?>
<svg width="200" height="100">
  <rect x="10" y="10" width="180" height="80"/>
  <ellipse cx="100" cy="50" rx="90" ry="40"/>
  <ellipse class="circle" cx="100" cy="50" rx="40" ry="40"/>
</svg>
```



Élément	Rendu (PNG)	Rendu (SVG)
Ligne <line x1=... y1=... x2=... y2=... />		
Rectangle <rect x=... y=... width=... height=... />		
Ellipse <ellipse cx=... cy=... rx=... ry=... />		
Ligne polygonale <polyline points="x1 y1 x2 y2 x3 y3 ..."/>		
Polygone <polygon points="x1 y1 x2 y2 x3 y3 ..."/>		

### 11.2.2 Chemins

Élément	Rendu (PNG)	Rendu (SVG)
Point de départ <path d="M x1 y1"/>		
Ligne horizontale <path d="M x1 y1 H x2 y1"/>		

### 11.2.3 Remplissage

Élément	Rendu (PNG)	Rendu (SVG)
Règle evenodd <path d="..." fill="evenodd"/>		
Règle nonzero <path d="..." fill="nonzero"/>		

### 11.3 Transformations

L'élément g permet de grouper plusieurs éléments graphiques. Il est ainsi possible d'associer simultanément à plusieurs éléments des règles de style communes.

L'élément g permet aussi d'appliquer des transformations affines sur les éléments graphiques. L'attribut transform de l'élément g contient une suite de transformations appliquées successivement. Les transformations possibles sont les suivantes.

Transformation	Action
translate(dx,dy)	Translation (déplacement)
scale(x) ou scale(x,y)	Changement d'échelle
rotate(a) ou scale(a,cx,cy)	Rotation
skewX(a) et skewY(a)	

### 11.4 Indications de style

#### 11.4.1 Attribut style

```
<svg width="200" height="100">
  <rect x="10" y="10" width="180" height="80" style="stroke:black;fill:none"/>
  <ellipse cx="100" cy="50" rx="90" ry="40" style="stroke:black;fill:red"/>
  <ellipse class="circle" cx="100" cy="50" rx="40" ry="40"/>
</svg>
```



## 11.5 Courbes de Bézier et B-splines

### 11.5.1 Courbes de Bézier

Les courbes de Bézier sont des courbes de degré 3. Elles sont donc déterminées par quatre points de contrôle. La courbe déterminée par les points  $P_1, P_2, P_3$  et  $P_4$  va de  $P_1$  à  $P_4$  et ses dérivées en  $P_1$  et  $P_4$  sont respectivement  $3(P_2 - P_1)$  et  $3(P_3 - P_4)$ . Ceci signifie en particulier que la courbe est tangente en  $P_1$  et  $P_4$  aux droites  $P_1P_2$  et  $P_3P_4$ .

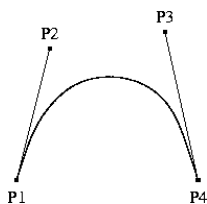


Fig. 1 : Courbe de Bézier

Si les coordonnées des points de contrôle sont  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$  et  $(x_4, y_4)$ , l'équation de la courbe est

$$x(t) = (1-t)^3x_1 + 3(1-t)^2x_2 + 3t^2(1-t)x_3 + t^3x_4 \quad \text{pour } 0 \leq t \leq 1$$

$$y(t) = (1-t)^3y_1 + 3(1-t)^2y_2 + 3t^2(1-t)y_3 + t^3y_4 \quad \text{pour } 0 \leq t \leq 1$$

L'applette ci-dessous permet d'expérimenter la forme de la courbe de Bézier en fonction de la position des points de contrôle. La courbe évolue lorsque les points de contrôle sont déplacés à la souris.

La méthode de Casteljau permet la construction géométrique de points de la courbe. Soient  $P_1, P_2, P_3$  et  $P_4$  les points de contrôle et soient  $L_2, H$  et  $R_3$  les milieux des segments  $P_1P_2, P_2P_3$  et  $P_3P_4$ . Soient  $L_3$  et  $R_2$  les milieux des segments  $L_2H$  et  $HR_3$  et soit  $L_4 = R_1$  le milieu du segment  $L_3R_2$  (cf. Figure 2). Le point  $L_4 = R_1$  appartient à la courbe de Bézier et il est obtenu pour  $t = 1/2$ . De plus la courbe se décompose en deux courbes de Bézier : la courbe de points de contrôle  $L_1 = P_1, L_2, L_3$  et  $L_4$  et la courbe de points de contrôle  $R_1, R_2, R_3$  et  $R_4 = P_4$ . Cette décomposition permet de poursuivre récursivement la construction de points de la courbe.

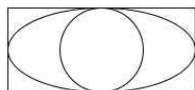
On remarque que chaque point de la courbe est barycentre des points de contrôle affectés des poids  $(1-t)^3, 3t(1-t)^2, 3t^2(1-t)$  et  $t^3$ . Comme tous ces poids sont positifs, la courbe se situe entièrement dans l'enveloppe convexe des points de contrôle.

Si dans la construction précédente, les milieux sont remplacés par les barycentres avec les poids  $t$  et  $1-t$ , on obtient le point de la courbe de coordonnées  $x(t), y(t)$ .

Style dans une feuille de style attachée

### 11.4.5 Au niveau d'un groupe

```
<svg width="200" height="100">
  <g style="stroke: black; fill: none">
    <rect x="10" y="10" width="180" height="80"/>
    <ellipse cx="100" cy="50" rx="90" ry="40"/>
    <ellipse class="circle" cx="100" cy="50" rx="40" ry="40"/>
  </g>
</svg>
```



Style dans le groupe

### 11.5.3 Conversion

Puisque seules les courbes de Bézier sont présentes en SVG, il est nécessaire de savoir passer d'une B-Spline à une courbe de Bézier. La B-spline de points de contrôle  $P_1, P_2, P_3$  et  $P_4$  est en fait la courbe de Bézier dont les points de contrôle  $P'_1, P'_2, P'_3$  et  $P'_4$  sont calculés de la manière suivante. Si les coordonnées des points  $P_1, P_2, P_3$  et  $P_4$  sont  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$  et  $(x_4, y_4)$ , les coordonnées  $(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3)$  et  $(x'_4, y'_4)$  des points  $P'_1, P'_2, P'_3$  et  $P'_4$  sont données par les formules suivantes pour les premières coordonnées et des formules similaires pour la seconde.

$$x'_1 = 1/6(x_1 + 4x_2 + x_3)$$

$$x'_2 = 1/6(4x_2 + 2x_3)$$

$$x'_3 = 1/6(2x_2 + 4x_3)$$

$$x'_4 = 1/6(x_2 + 4x_3 + x_4)$$

Les formules suivantes permettent la transformation inverse

$$x_1 = 6x'_1 - 7x'_2 + 2x'_3$$

$$x_2 = 2x'_2 - x'_3$$

$$x_3 = -x'_2 + 2x'_3$$

$$x_4 = 2x'_2 - 7x'_3 + 6x'_4$$

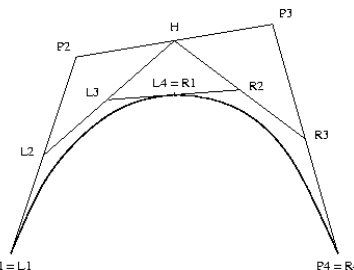


Fig. 2 : Construction de Casteljau

### 11.5.2 B-splines

Les courbes B-splines sont aussi des courbes de degré 3. Elles sont donc aussi déterminées par quatre points de contrôle. Contrairement à une courbe de Bézier, une B-spline ne passe par aucun de ses points de contrôle. Par contre, les B-splines sont adaptées pour être mises bout à bout afin de former une courbe ayant de multiples points de contrôle.

Soient  $n+3$  points  $P_1, \dots, P_{n+3}$ . Ils déterminent  $n$  B-splines  $s_1, \dots, s_n$  de la manière suivante. Chaque B-spline  $s_i$  est déterminée par les points de contrôle  $P_i, P_{i+1}, P_{i+2}$  et  $P_{i+3}$ .

Si les coordonnées des points de contrôle sont  $(x_1, y_1), \dots, (x_{n+3}, y_{n+3})$ , l'équation de la spline  $s_i$  est

$$x_i(t) = 1/6((1-t)^3x_i + (3t^3 - 6t^2 + 4)x_{i+1} + (-3t^3 + 3t^2 + 3t + 1)x_{i+2} + t^3x_{i+3}) \quad \text{pour } 0 \leq t \leq 1$$

$$y_i(t) = 1/6((1-t)^3y_i + (3t^3 - 6t^2 + 4)y_{i+1} + (-3t^3 + 3t^2 + 3t + 1)y_{i+2} + t^3y_{i+3}) \quad \text{pour } 0 \leq t \leq 1$$

À partir des équations définissant les splines  $s_i$ , on vérifie facilement les formules suivantes qui montrent que la courbe obtenue en mettant bout à bout les courbes  $s_i$  est de classe  $C^2$ , c'est-à-dire deux fois dérivable.

$$s_i(1) = s_{i+1}(0) = 1/6(P_{i+1} + 4P_{i+2} + P_{i+3})$$

$$s'_i(1) = s'_{i+1}(0) = 1/2(P_{i+3} - P_{i+1})$$

$$s''_i(1) = s''_{i+1}(0) = P_{i+3} - 2P_{i+2} + P_{i+1}$$

L'applette ci-dessous permet d'expérimenter la forme des B-splines en fonction de la position des points de contrôle. Le bouton gauche permet de déplacer les points de contrôle et le bouton droit d'en ajouter.

```

* Ce handler se contente d'afficher les balises ouvrantes et fermantes.
* @author O. Carton
* @version 1.0
*/

```

```

class TrivialSAXHandler extends DefaultHandler {
    public void setDocumentLocator(Locator locator) {
        System.out.println("Location : " +
            "publicId=" + locator.getPublicId() +
            " systemId=" + locator.getSystemId());
    }
    public void startDocument() {
        System.out.println("Debut du document");
    }
    public void endDocument() {
        System.out.println("Fin du document");
    }
    public void startElement(String namespace,
        String localname,
        String qualname,
        Attributes attrs) {
        System.out.println("Balise ouvrante : " +
            "namespace=" + namespace +
            " localname=" + localname +
            " qualname=" + qualname);
    }
    public void endElement(String namespace,
        String localname,
        String qualname) {
        System.out.println("Balise fermante : " +
            "namespace=" + namespace +
            " localname=" + localname +
            " qualname=" + qualname);
    }
    public void characters(char[] ch, int start, int length) {
        System.out.print("Caractères : ");
        for(int i = start; i < start+length; i++)
            System.out.print(ch[i]);
        System.out.println();
    }
}

```

Lecture d'un fichier XML

```

// IO
import java.io.InputStream;
import java.io.FileInputStream;
// SAX
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;

/**
 * Lecture triviale d'un document XML avec SAX
 * @author O. Carton
 * @version 1.0
 */
class TrivialSAXRead {
    public static void main(String [] args)
        throws Exception
    {
        // Création de la fabrique de parsers
        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
    }
}

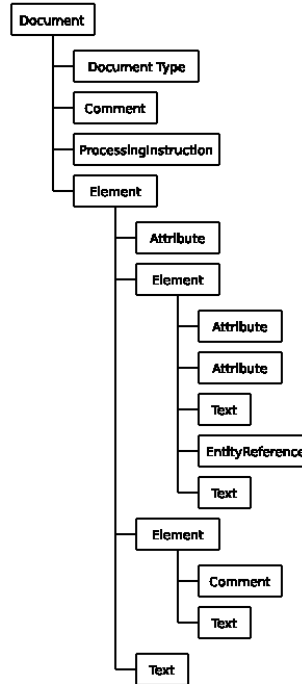
```

```

Du texte
</item>
et encore du texte.
</table>

```

La figure ci-dessous représente sous forme d'arbre le document XML présenté ci-dessus.



Arbre d'un document

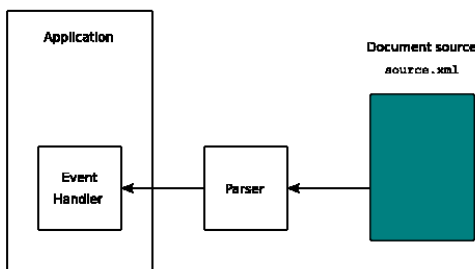
## 12 Programmation XML

Page des sources des exemples

Il existe deux méthodes essentielles appelées SAX et DOM pour lire un document XML dans un fichier. Il en existe en fait une troisième appelée StAX qui n'est pas abordée ici.

### 12.1 SAX

#### 12.1.1 Principe



Principe de SAX

SAX est une API permettant de lire un fichier XML sous forme de flux. Le principe de fonctionnement est le suivant. L'application crée un parseur et elle enregistre auprès de ce parseur son gestionnaire d'événements. Au cours de la lecture du fichier contenant le document XML, le gestionnaire reçoit les événements générés par le parseur. Le document XML n'est pas chargé en mémoire.

La tâche du programmeur est légère puisque le parseur est fourni par l'environnement Java. Seul le gestionnaire d'événements doit être écrit par le programmeur. Cette écriture est facilitée par les gestionnaires par défaut qu'il est facile de dériver pour obtenir un gestionnaire.

#### 12.1.2 Lecture d'un fichier XML avec SAX

Handler minimal

```

import org.xml.sax.Attributes;
import org.xml.sax.Locator;
import org.xml.sax.helpers.DefaultHandler;

/**
 * Handler trivial pour SAX

```

```

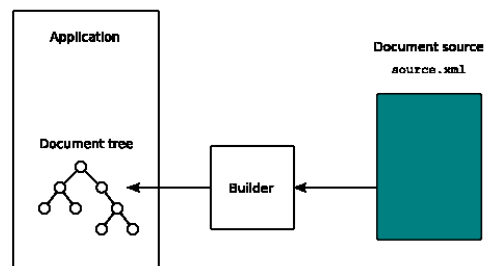
// Création du parser
SAXParser parser = parserFactory.newSAXParser();

// Lecture de chaque fichier passé en paramètre
for(int i = 0; i < args.length; i++) {
    // Flux d'entrée
    InputStream is = new FileInputStream(args[i]);
    parser.parse(is, new TrivialSAXHandler());
}
}
}

```

### 12.2 DOM

#### 12.2.1 Principe



Principe de DOM

DOM est une API permettant de charger un document XML sous forme d'un arbre qu'il est ensuite possible de manipuler. Le principe de fonctionnement est le suivant. L'application crée un constructeur qui lit le document XML et construit une représentation du document XML sous forme d'un arbre.

#### 12.2.2 Arbre document

```

<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Time-Stamp: tree.xml 14 Feb 2008 09:29:00 -->
<?xml-stylesheet href="tree.xsl" type="text/xsl"?>
<table type="technical">
  <item key="id001" lang="fr">
    XML &amp; Co
  </item>
</table>
<!-- Un commentaire inutile -->

```

```

{
  // Création de la fabrique de constructeur de documents
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
  // Création du constructeur de documents
  DocumentBuilder documentBuilder = dbf.newDocumentBuilder();

  // Lecture de chaque fichier passé en paramètre
  for(int i = 0; i < args.length; i++) {
    // Flux d'entrée
    InputStream is = new FileInputStream(args[i]);
    // Construction du document
    Document doc = documentBuilder.parse(is);
    // Exploitation du document ...
    System.out.println(doc);
  }
}

```

12.3 Comparaison

La grande différence entre les API SAX et DOM est que la première ne charge pas le document en mémoire alors que la seconde construit en mémoire une représentation arborescente du document. La première est donc particulièrement adaptée aux (très) gros documents. Par contre, elle offre des facilités de traitement plus réduites. Le fonctionnement par événements rend difficiles des traitements non linéaires du document. Au contraire, l'API DOM rend plus faciles des parcours de l'arbre.

12.4 AJAX

Cette page donne un petit historique de l'objet XMLHttpRequest.

```

<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr">
<head>
  <title>
    Chargement dynamique
  </title>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
  <meta name="Author" content="Olivier Carton" />
  <script type="text/javascript">
    // Dans la première version de Google Suggest, la fonction
    // s'appelait sendRPCDone. Elle s'appelle maintenant
    // window.google.ac.Suggest_apply
    window.google = new Object();
    window.google.ac = new Object();
    window.google.ac.Suggest_apply = sendRPCDone;

    var XMLHttpRequestObject = false;
    // Création du gestionnaire de connexion
    if (window.XMLHttpRequest) {
      XMLHttpRequestObject = new XMLHttpRequest();
    } else {
      XMLHttpRequestObject = new ActiveXObject("Microsoft.XMLHTTP");
    }

    // Fonction de chargement des données
    function getData(uri) {

```

- Éditeurs
  - emacs
  - kxmleditor
- Programmes de validation
  - nsgmls (projet OpenJade)
  - xmllint (partie de la librairie libxml2)
  - xerces (projet XML d'Apache)
- Application de feuilles de style XSLT
  - xsltproc (partie de la librairie libxslt)
  - xalan (projet XML d'Apache)
  - saxon
  - xt de James Clark
  - sablotron
- Transformation de document XML-FO en PDF
  - fop d'Apache)

13.1 Memento logiciels

13.1.1 Validation DTD

13.1.1.1 Xmllint

Validation d'un document avec DTD interne ou locale

```
xmllint --valid bibliography.xml
```

13.1.1.2 Nsgmls

```

export SP_CHARSET_FIXED="yes"
export SP_ENCODING="XML"
export SGML_CATALOG_FILES="/usr/share/sgml/opensp-1.5.2/OpenSP/xml.soc"
nsgmls -s bibliography.xml

```

13.1.2 Validation par un schéma

13.1.2.1 Xmllint

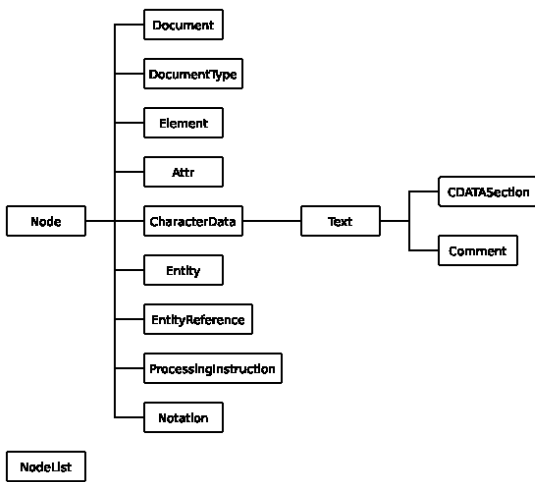
```
xmllint --noout --schema bibliography.xsd bibliography.xml
```

13.1.3 Application de transformation XSLT

13.1.3.1 xsltproc

```
xsltproc --stringparam param value helloworld.xml bibliography.xml
```

12.2.3 Principales classes



Classes du DOM

12.2.4 Lecture d'un fichier XML avec DOM

Lecture d'un fichier XML

```

// IO
import java.io.InputStream;
import java.io.FileInputStream;
// DOM
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;

/**
 * Lecture triviale d'un document XML avec DOM
 * @author O. Carton
 * @version 1.0
 */
class TrivialDOMRead {
  public static void main(String [] args)
    throws Exception

```

```

if (XMLHttpRequestObject) {
  // Mise en place de la requête
  XMLHttpRequestObject.open("GET", url);

  // Mise en place du handler
  XMLHttpRequestObject.onreadystatechange = function() {
    if (XMLHttpRequestObject.readyState == 4 &&
        XMLHttpRequestObject.status == 200) {
      // Évaluation du résultat
      eval(XMLHttpRequestObject.responseText);
    }
  }
  // Envoi de la requête
  XMLHttpRequestObject.send(null);
}

// Fonction appelée à chaque entrée de caractère
function getSuggest(keyEvent) {
  var keyEvent = (keyEvent) ? keyEvent : window.event;
  var input = (keyEvent.target) ? keyEvent.target : keyEvent.srcElement;
  // Utilisation du wrapper google.php
  var url = 'google.php?qu=';

  if (keyEvent.type == 'keyup') {
    if (input.value) {
      getData(url + input.value);
    } else {
      var target = document.getElementById("target");
      target.innerHTML = "<div></div>";
    }
  }
}

// Fonction appelée par la requête
function sendRPCDone(unused, term, results, unusedArray) {
  // Entête de la table de présentation des résultats
  var data = "<table align='left' border='1' " +
    "cellpadding='2' cellspacing='0'>";
  if (results.length != 0) {
    for (var i = 1; i < results.length-1; i = i + 2) {
      data += "<tr><td><a href='http://www.google.com/search?q=" +
        results[i] + "'>" + results[i+1] + "</a></td>" +
        "<td>" + results[i+1] + "</td></tr>";
    }
  }
  data += "</table>";
  var target = document.getElementById("target");
  target.innerHTML = data;
}
</script>
</head>

<body>
<h2>Google Live Search</h2>
<!-- Zone de saisie -->
<!-- Chaque touche lâchée provoque un appel à la fonction getSuggest -->
<input type="text" onkeyup="JavaScript:getSuggest(event)" /></p>
<!-- Div recevant le résultat -->
<div id="target"></div></p>
</body>
</html>

```

## 14 Acronymes XML

AJAX  
Asynchronous JavaScript and XML

API  
Application Programming Interface

BMP  
Basic Multilingual Plane

CSS  
Cascading Style Sheet

DOM  
Document Object Model

DTD  
Document Type Definition

FPI  
Formal Public Identifier

Infoset  
XML Information Set

NCName  
No Colon Name (Nom local)

PCDATA  
Parsed characters DATA

PNG  
Portable Network Graphics

PSVI  
Post-Schema Validation Infoset

QName  
Qualified Name

RDF  
Resource Description Framework

RSS  
Really Simple Syndication

SAX  
Simple API for XML

SGML  
Standard Generalized Markup Language

SMIL  
Synchronized Multimedia Integration Language

SOAP  
Simple Object Access Protocol

SVG  
Scalable Vector Graphics

UCS  
Universal Character Set

URI  
Uniform Resource Identifier

URL  
Uniform Resource Locator

- 108 -

## 15 Bibliographie

### 15.1 XML

1. A. Michard, *XML langage et applications*. Eyrolles, 2001. L'édition 2001 est beaucoup plus complète que l'édition 1999.
2. B. Marchal, *XML by Example*. Macmillan Couputer Publishing, 2000.
3. M. Morrison, *XML*. CampusPress, 2005. (Livre très superficiel).
4. F. Role, *Modélisation et manipulation de documents XML*. Lavoisier, 2005.

### 15.2 XPath

1. M. Kay, *XPath 2.0 Programmer's Reference*. Wiley Publishing Inc., 2004.

### 15.3 Schémas XML

1. V. Lamareille, *XML Schema et XML Infoset*. Cepaduès, 2006.

### 15.4 XSLT et XSL-FO

1. P. Drix, *XSLT fondamentale*. Eyrolles, 2002.
2. D. Tidwell, *XSLT*. O'Reilly, 2001.
3. M. Kay, *XSLT 2.0 Programmer's Reference*. Wiley Publishing Inc., 2004.

### 15.5 CSS

1. C. Aubry, *CSS 1 et CSS 2.1*. Editions ENI, 2006.

### 15.6 SVG

1. J. D. Eisenberg, *SVG Essentials*. O'Reilly, 2002.

- 110 -

#### 13.1.3.2 Saxon

```
saxon8 bibliography.xml helloworld.xml param=value
```

#### 13.1.3.3 Xt

```
xt bibliography.xml helloworld.xml param=value
```

#### 13.1.3.4 Sablotron

Le caractère \$ est un caractère spécial du shell.

```
sabcmd bibliography.xml helloworld.xml \${param=value}
```

#### 13.1.3.5 Xalan

```
xalan -IN bibliography.xml -XSL helloworld.xml -PARAM param value
```

#### 13.1.4 Création d'un fichier PDF avec fop

```
# Directement à partir d'un fichier FO
fop file.fo file.pdf
# Application d'une feuille de style pour produire le fichier FO
fop -xsl file.xsl -xml file.xml -pdf file.pdf
```

- 107 -

URN  
Universal Resource Name

XML  
eXtensible Markup Language

XSD  
XML Schema Definition

XSL  
eXtensible Style Language

XSL-FO  
XSL Formating Object

- 109 -

6.16	Schémas et espaces de noms	48
6.17	Imports d'autres schémas	51
<b>XPath</b>		53
7	XPath	53
7.1	Arbre d'un document XML	53
7.2	Forme des expressions	53
7.3	Axes	54
7.4	Types	56
7.5	Prédicats	56
7.6	Syntaxe abrégée	56
7.7	Exemples	57
7.8	Utilisation de xml:lint	57
7.9	XPath 2.0	58
<b>Programmation XSL</b>		61
8	Programmation XSL	61
8.1	Principe	61
8.2	Premier programme : Hello, World!	61
8.3	Modèle de traitement	62
8.4	Entête	62
8.5	Définition et application de règles	62
8.6	Règles par défaut	63
8.7	Construction du résultat	64
8.8	Structures de contrôle	70
8.9	Tris	71
8.10	Variables et paramètres	72
8.11	Modes	75
8.12	Indexation	77
8.13	Documents multiples	77
<b>XSL-FO</b>		79
9	XSL-FO	79
9.1	Premier exemple	79
9.2	Structure globale	79
<b>Feuilles de style CSS</b>		80
10	Feuilles de style CSS	80
10.1	Principe	80
10.2	Règles	80
10.3	Héritage et cascade	84
10.4	Modèle de boîtes	85
10.5	Style et XML	86
10.6	Attachement de règles de style	87
10.7	Principales propriétés	88
<b>SVG</b>		90
11	SVG (Scalar Vector Graphics)	90
11.1	Un premier exemple	90
11.2	Éléments de dessins	90
11.3	Transformations	93
11.4	Indications de style	93
<b>Courbes de Bézier et B-splines</b>		96
11.5	Courbes de Bézier et B-splines	96

- II -

## Table des matières

<b>XML</b>		1
1	XML en M2	1
<b>Introduction à XML</b>		2
2	Introduction à XML	2
2.1	Présentation	2
2.2	Historique	2
2.3	Intérêts	2
2.4	Dialectes et extensions	2
2.5	Applications	3
<b>Syntaxe de XML</b>		4
3	Syntaxe	4
3.1	Premier exemple	4
3.2	Syntaxe et structure	4
3.3	Composition globale d'un document	4
3.4	Prologue	5
3.5	Corps du document	6
3.6	Exemples	9
<b>DTD</b>		11
4	DTD	11
4.1	Un premier exemple	11
4.2	Déclaration de la DTD	12
4.3	Contenu de la DTD	13
4.4	Outils de validations	20
<b>Espaces de noms</b>		21
5	Espaces de noms	21
5.1	Identification d'un espace de noms	21
5.2	Déclaration d'un espace de noms	21
5.3	Portée d'une déclaration	22
5.4	Espace de noms par défaut	23
5.5	Attributs	24
5.6	Espace de noms xml	24
<b>Schémas XML</b>		25
6	Schémas XML	25
6.1	Introduction	25
6.2	Un premier exemple	25
6.3	Structure globale d'un schéma	26
6.4	Commentaires	27
6.5	Déclarations d'éléments	27
6.6	Types prédéfinis	29
6.7	Déclarations de types	30
6.8	Constructions de types	32
6.9	Extension de types	36
6.10	Restriction de types	38
6.11	Groupes de substitution	41
6.12	Contrôle des dérivations et substitutions	42
6.13	Déclarations d'attributs	45
6.14	Groupes d'éléments et d'attributs	45
6.15	Contraintes de cohérence	46

<b>Programmation XML</b>		99
12	Programmation XML	99
12.1	SAX	99
12.2	DOM	101
12.3	Comparaison	104
12.4	AJAX	104
<b>Logiciels XML</b>		106
13	Logiciels XML	106
13.1	Memento logiciels	106
<b>Acronymes XML</b>		108
14	Acronymes XML	108
<b>Bibliographie</b>		110
15	Bibliographie	110
15.1	XML	110
15.2	XPath	110
15.3	Schémas XML	110
15.4	XSLT et XSL-FO	110
15.5	CSS	110
15.6	SVG	110