L'essentiel de XML

Cours XML

Olivier Carton

L'essentiel de XML: Cours XML

Olivier Carton

Version du 04/09/2012

Résumé

Support du cours XML en M2 Pro à l'Université Paris Diderot.

Ce document est le support d'un cours XML donné en M2 Pro à l'Université Paris Diderot. L'objectif est de présenter les aspects essentiels de XML de manière concise et illustrée par de nombreux exemples. Les principaux thèmes abordés sont la syntaxe de XML, la validation de documents par des DTD, des schémas et des schematrons, le langage XPath, la transformation de document par XSLT ainsi que la programmation.

Ce support de cours est actuellement en cours de rédaction. Il contient encore beaucoup d'erreurs et d'omissions. Certaines parties méritent d'être développées et/ou reprises. Une certaine indulgence est donc demandée au lecteur. Toutes les corrections, même les plus mineures, suggestions et encouragements sont les bienvenus. Ils participent à l'amélioration de ce document pour le bien de tous.

Table des matières

Chapitre 1. Présentation de XML

Le langage XML (eXtended Markup Language) est un format général de documents orienté texte. Il s'est imposé comme un standard incontournable de l'informatique. Il est aussi bien utilisé pour le stockage de document que pour la transmission de données entre applications. Sa simplicité, sa flexibilité et ses possibilités d'extension ont permis de l'adapter à de multiples domaines allant des données géographiques au dessin vectoriel en passant par les échanges commerciaux. De nombreuses technologies se sont développées autour de XML et enrichissent ainsi son environnement.

Le langage XML dérive de SGML (Standard Generalized Markup Language) et de HTML (HyperText Markup Language). Comme ces derniers, il s'agit d'un langage orienté texte et formé de *balises* qui permettent d'organiser les données de manière structurée.

1.1. Historique

L'historique suivant retrace les grandes étapes qui ont conduit à la naissance de XML. L'ancêtre de XML est le langage SGML qui a été introduit en 1986 par C. Goldfarb. SGML a été conçu pour des documentations techniques de grande ampleur. Sa grande complexité a freiné son utilisation en dehors des projets de grande envergure. En 1991, T. Berners-Lee a défini le langage HTML pour le WEB. Ce langage est une version simplifiée à l'extrême de SGML, destinée à une utilisation très ciblée. XML est, en quelque sorte, intermédiaire entre SGML et HTML. Il évite les aspects les plus complexes de SGML tout en gardant suffisamment de souplesse pour une utilisation généraliste. La version 1.0 de XML a été publiée en 1998 par le consortium W3C (World Wide Web Consortium). Une redéfinition XHTML de HTML 4.0 à travers XML a été donnée en 1999. Une seconde version 1.1, qui est simplement une mise à jour pour les caractères spéciaux en lien avec Unicode, a, ensuite, été publiée en 2004.

1.2. Intérêts

XML est devenu omniprésent dans le monde de l'informatique. De nombreux standards sont apparus et permettent à des applications différentes de stocker mais surtout de partager des documents. L'exemple le plus emblématique et le plus connu est le format OpenDocument qui est utilisé par OpenOffice, maintenant appelé LibreOffice, mais aussi par d'autres suites bureautiques comme KOffice. Un autre exemple est le format de dessins vectoriels SVG [Chapitre 11] utilisé par Inkscape. Ce succès de XML est en grande partie du à ses qualités. Nous allons d'abord énumérer ces caractéristiques essentielles qui ont conduit à ce développement puis nous allons les détailler plus en profondeur.

- Séparation stricte entre contenu et présentation
- Simplicité, universalité et extensibilité
- Format texte avec gestion des caractères spéciaux
- Structuration forte
- Modèles de documents (DTD [Chapitre 3] et Schémas XML [Chapitre 5])
- Format libre

Une des idées directrices de XML est la séparation entre contenu et présentation. Il faut bien distinguer le contenu d'un document et la présentation qui en est donnée. Un même contenu peut être rendu de façons très différentes. Cet ouvrage peut, par exemple, se présenter comme un livre imprimé ou comme une collections de pages WEB. Le contenu est constitué, au départ, de textes et d'illustrations mais aussi de liens entre ces éléments. L'organisation du texte en chapitres, sections et sous-sections ainsi que les renvois entre chapitres font partie intégrante du contenu du document. La présentation est au contraire la façon de présenter ce contenu au lecteur. Un des premiers principes de XML est d'organiser le contenu de manière indépendante de la présentation. Ce principe de séparation est déjà présent dans HTML. Le rendu d'une page HTML est globalement confié au navigateur. Les paragraphes des documents HTML sont, par exemple, écrits au kilomètre, sans indication de fins de lignes. Il appartient au navigateur de les découper en lignes en fonction de la taille de la page. C'est très différent d'un document PDF

où le découpage en pages et en lignes est figée par le document. La séparation n'est cependant pas totale en HTML. Certaines balises comme `<ul>` et `<li>` sont destinées à la structuration du document. Pour ces deux balises, il s'agit d'écrire une énumération. La présentation de cette énumération (marges, symbole marquant chaque entrée, …) est déléguée au navigateur. D'autre balises comme `<i>` ou `<b>` donnent davantage des indications de présentation. Cette séparation entre contenu et présentation a été accrue en HTML par l'introduction de CSS [Chapitre 10]. Une feuille de style CSS est chargée de donner au navigateur toutes les informations relatives à la présentation. Le document HTML peut alors se recentrer sur la structuration du contenu de façon indépendante de la présentation.

Cette séparation entre contenu et présentation est difficile à obtenir. Le rédacteur d'un document a souvent une présentation en tête et il est tentant pour lui d'essayer de l'imposer.

La séparation est encore plus marquée en XML car la signification des balises n'est pas figée comme en HTML. La règle est alors de choisir les balises pour organiser le document en privilégiant la structure de celui-ci par rapport à une éventuelle présentation.

Un second principe de XML est une structuration forte du document. Pour illustrer ce principe, la même adresse est donnée ci-dessous dans un style HTML puis dans un style XML.

```
Olivier Carton<br/>
175, rue du Chevaleret<br/>
75013 Paris<br/>
<tt>Olivier.Carton@liafa.jussieu.fr</tt>
```

```
<address>
  <personname>
    <firstname>Olivier</firstname> <surname>Carton</surname>
  </personname>
  <street>
    <streetnumber>175</streetnumber> <streetname>rue du Chevaleret</streetname>
  </street>
  <zipcode>75013</zipcode><city>Paris</city>
  <email>Olivier.Carton@liafa.jussieu.fr</email>
</address>
```

Les deux adresses contiennent les mêmes informations purement textuelles. La suppression des balises comme `<br/>` ou `<firstname>` donne le même résultat pour les deux adresses. Les balises `<br/>` présentes dans la première adresse permettent le découpage en ligne et la balise `<tt>` indique une police de caractères appropriée pour l'adresse électronique. Le rôle de ces balises est seulement d'assurer un rendu correct de l'adresse. Les balises dans la première adresse relèvent de la présentation alors que les balises dans la seconde adresse comme `<firstname>` ont un rôle sémantique. Ces dernières structurent les données textuelles et ajoutent ainsi de l'information. Le nom est, par exemple, décomposé en prénom et nom. Cette information supplémentaire facilite le traitement des données. Il devient, par exemple, très facile de mettre le nom en majuscule. En revanche, le rendu exact de l'adresse électronique est à la charge de l'application traitant l'adresse. L'adresse électronique peut, par exemple, être supprimée si l'adresse est utilisée pour une lettre.

Une des caractéristiques essentielles de XML est son extensibilité et sa flexibilité. Contrairement à HTML, le vocabulaire, c'est-à-dire l'ensemble des balises autorisées, n'est pas figé. La norme HTML fixe les balises pouvant apparaître dans un document ainsi que leur imbrication possible. À titre d'exemple, la balise `<li>` peut uniquement apparaître dans le contenu d'une balise `<ul>` ou `<ol>`. En revanche, les noms des balises XML sont libres. Il appartient aux auteurs de documents de fixer les balises utilisées. Il est seulement nécessaire que les auteurs s'entendent sur le vocabulaire, c'est-à-dire la liste des balises utilisées, lorsque des documents sont échangés. Cette liberté dans les noms de balises permet de définir des vocabulaires particuliers adaptés aux différentes applications. Il existe ainsi des vocabulaires pour décrire des dessins vectoriels, des échanges commerciaux ou des programmes de télévision. Ces vocabulaires particuliers sont appelés *dialectes* XML. Il en existe des centaines voire des milliers pour couvrir tous les champs d'application de XML.

La liberté dans le choix des noms de balises implique une contrepartie. Il devient nécessaire de fixer des règles que doivent respecter les documents. Sans ces règles, il n'est pas possible d'échanger et de traiter de manière automatique ces documents. Ces règles doivent d'abord fixer le vocabulaire mais aussi les relations entre les

balises. Les règles peuvent, par exemple, imposer qu'une balise `<address>` (cf. exemple ci-dessus) contiennent exactement une balise `<zipcode>` et une balise `<city>` sans pour autant fixer l'ordre de ces deux balises. Ces ensembles de règles portant sur les documents XML sont appelés *modèles de documents*. Plusieurs langages ont été développés pour décrire ces modèles. Le premier de ces langages est celui des DTD (Document Type Definition) qui est hérité de SGML. Des langages plus puissants, parmi lesquels les schémas ou relax NG, ont été introduits depuis pour remplacer les DTD. L'intérêt principal de ces modèles de documents est de pouvoir décrire explicitement les règles à respecter pour un document et de pouvoir vérifier si un document donné les respecte effectivement. Avant ces modèles de documents, il n'était pas rare que les données à fournir à un logiciel ne fussent pas décrites de façon très précise. Il n'était alors pas toujours facile de prévoir si des données seraient acceptées par le logiciel, et si ce n'était pas le cas de déterminer lequel des données ou du logiciel était en cause.

Les modèles de document tels que les DTD ou les schémas peuvent servir à une vérification automatique des documents. Il existe plusieurs implémentations de ces modèles. Cela autorise la vérification qu'un document donné satisfait ou non les contraintes spécifiées par une DTD ou un schéma. Lorsque les entrées possibles d'un logiciel sont décrites par un tel modèle, il est possible de vérifier de façon indépendante du logiciel que les données sont correctes. Cette possibilité est particulièrement intéressante lors d'échanges de documents.

Bien que les données présentes dans un document XML soient fortement structurées, le format XML est un format basé sur du texte. Il est ainsi possible de manipuler un document XML à l'aide d'un simple éditeur de texte. Il n'est pas nécessaire d'utiliser un logiciel spécialisé. Il existe bien sûr des logiciels spécialement conçus pour l'édition de documents XML. Ceux-ci peuvent grandement faciliter la tâche de l'auteur mais ils ne sont pas indispensables. Cet ouvrage a, par exemple, été rédigé avec l'éditeur Emacs.

Un des atouts d'XML est sa prise en charge native des caractères spéciaux grâce à Unicode. De plus, il est possible d'utiliser les différents codages (UTF-8, Latin-1, …) possibles puisque l'entête d'un document spécifie le codage.

De nombreuses technologies se sont développées autour de XML et en facilitent l'utilisation. Un des champs d'application est la manipulation et la transformation de documents XML. Il s'agit, par exemple, de réorganiser un document ou d'en extraire des fragments ou de le transformer complètement dans un autre format comme PDF. Comme tous les documents XML partagent la même syntaxe quel que soit leur vocabulaire, des outils permettent de manipuler ces documents de manière générale en s'affranchissant des aspects syntaxiques. Des langages de haut niveau comme XSLT [Chapitre 8] autorisent la description simple et concise de transformations. Le grand intérêt de ces langages est qu'ils sont indépendants du vocabulaire utilisés et qu'ils s'adaptent à tous les dialectes XML. Le langage XSLT manipule chaque document XML sous la forme d'un arbre issu de la structure des données. Les transformations sont décrites en XSLT par des règles qui s'appliquent aux fragments délimités par les balises. Ce langage XSLT est doublement intéressant. D'une part, il constitue un outil à la fois pratique et puissant et donc d'une grande utilité face à de nombreux problèmes concrets. D'autre part, il représente une approche originale de la programmation car il n'est pas basé sur la programmation impérative ou fonctionnelle de la grande majorité des langages. À ce titre, il est digne d'intérêt en soi. Le langage XSLT est lui-même un dialecte XML car il utilise la syntaxe XML. Des programmes XSLT peuvent être eux-mêmes manipulés et créés par d'autres programmes XSLT. Ce mécanisme est d'ailleurs mis en œuvre par les schematrons [Chapitre 7].

Le langage XML est totalement libre car il est développé par le W3C. Chacun peut l'utiliser sans devoir acheter une quelconque licence. Cette absence de droits favorise le développement de logiciels libres mis à disposition de la communauté. Il existe ainsi une très grande variété de logiciels libres autour de XML qui en couvrent les différents aspects.

1.3. Langages apparentés

Un des atouts indéniables de XML est le nombre de technologies et de langages qui se sont développés autour de XML. Ceux-ci enrichissent les outils pour la manipulation des documents XML. La liste ci-dessous énumère les principaux langages qui font partie de l'environnement XML.

XLink [W] et XPointer [W] (liens entre documents)

> XML contient déjà un mécanisme pour matérialiser des liens entre des éléments d'un document. XLink et XPointer permettent d'établir des liens entre documents et plus particulièrement entre un élément d'un document et un fragment d'un autre document. Ils généralisent les liens hypertextes des documents HTML en autorisant des liens entre plusieurs documents.

XPath [W] (langage de sélection)

XPath est un langage d'expressions permettant de sélectionner des éléments dans un document XML. Il est la pierre angulaire du langage XSLT pour la transformation de documents. Il est abordé au chapitre 6 de cet ouvrage.

XQuery [W] (langage de requête)

XQuery est un langage permettant d'extraire des informations à partir d'un ou plusieurs documents XML et de synthétiser de nouvelles informations à partir de celles extraites. Il s'apparente à un langage d'interrogation de bases de données et joue le rôle de SQL pour les documents XML.

Schémas XML [W] (modèles de documents)
: Les schémas XML remplacent les DTD héritées de SGML pour décrire des modèles de documents. Ils sont beaucoup plus souples et beaucoup plus puissants que les DTD. Ils sont abordés en détail au chapitre 5 de cet ouvrage.

XSLT [W] (transformation de documents)
: XSLT est un langage permettant d'exprimer facilement des transformations complexes entre documents XML. Il s'appuie sur la structuration forte des documents XML vus comme des arbres. Chaque transformation est décrite par des règles pour chacun des éléments du document. Il est étudié en profondeur au chapitre 8 de cet ouvrage.

1.4. Dialectes

De très nombreux dialectes ont été définis pour appliquer XML à des domaines très variés. Le grand avantage est que ces différents dialectes partagent la même syntaxe de base et que tous les outils XML peuvent être utilisés pour spécifier et manipuler ces documents. Il n'y a nul besoin de développer des outils spécifiques à ces différents dialectes. La liste ci-dessous énumère quelques uns de ces dialectes.

RSS [W] (Really Simple Syndication)

Abonnement à des flux de données

XUL [W] (XML-based User interface Language)

Langage de description d'interfaces graphiques développé par le projet Mozilla.

SVG [W] (Scalable Vector Graphics)

Description de dessins vectoriels

SMIL [W] (Synchronized Multimedia Integration Language)

Description de contenus multimédia

MathML [W] (Mathematical Markup Language)

Description de formules mathématiques

WSDL [W] (Web Services Description Language)

Description de services WEB

OpenStreetMap [W]

Cartes libres

XML Signature [W]
: Format pour les signatures électroniques

SAML [W] (Security Assertion Markup Language)

Langage d'échange d'authentifications et d'autorisations

UBL [W] (Universal Business Language)

Bibliothèque de documents standards pour les échanges commerciaux

OpenDocument [W]

Format de document pour les applications bureautiques développé au départ pour OpenOffice mais aussi utilisé par d'autres logiciels libres comme KOffice

DocBook [W]

Format de documentation technique

De nombreux projets informatiques, comme Ant ou Android utilisent XML pour le stockage de données et en particulier pour les fichiers de configuration.

1.5. DocBook

DocBook est un exemple typique d'utilisation de XML. Il s'agit d'un format pour écrire des documents techniques. Il est particulièrement adapté à la rédaction de documentations de logiciels. Il est d'ailleurs utilisé par de nombreux projets de logiciels libres, éventuellement de grande ampleur, comme le projet KDE. Cet ouvrage a été rédigé en utilisant DocBook. L'intégralité du texte est répartie en plusieurs fichiers XML. Afin d'obtenir une mise en page de qualité, les documents XML sont convertis, avec le langage XSLT, en un document LaTeX qui peut alors produire un document PDF.

DocBook était au départ basé sur SGML mais il s'appuie maintenant sur XML dont il est un des dialectes. Il contient de nombreuses balises permettant de décrire et de structurer le contenu de façon très précise. Il existe ensuite différents outils permettant de traduire un document DocBook, en une seule page HTML, en plusieurs pages HTML avec des liens ou encore en un document PDF.

DocBook met l'accent sur l'organisation et la structure du document. Son vocabulaire contient de très nombreuses balises permettant de transcrire très finement la sémantique de chaque fragment, à la manière de la seconde adresse donnée au début de cette introduction. Cet exemple est, en fait, très inspiré de DocBook. En revanche, DocBook ne permet pas de spécifier le rendu du document. Il n'est pas possible de donner, par exemple, la couleur ou la police de caractères à utiliser pour le texte. L'idée directrice est qu'un document DocBook doit permettre la production de plusieurs documents finaux à partir d'un même document original : document PDF, pages WEB. Comme les contraintes de ces différents média sont très diverses, il est impossible de pouvoir les spécifier dans le document. Le choix de DocBook est de donner suffisamment d'indications sur le contenu aux applications qui réalisent les transformations pour obtenir un résultat de qualité. Les documents DocBook font souvent partie d'un ensemble de documentations, comme celle de KDE, dont la présentation est uniforme et donc déterminée de manière globale.

1.6. Conventions

Certaines conventions sont utilisées tout au long de cet ouvrage afin d'en faciliter la lecture. Tous les exemples et plus généralement, tous les fragments de texte pouvant apparaître dans un document XML sont écrits en utilisant une police de caractères fixe comme l'exemple d'entête ci-dessous. Les noms des balises sont en particulier écrits avec cette police.

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

Lorsqu'un fragment de texte comporte des parties génériques qui doivent être remplacées pour obtenir un véritable exemple, ces parties sont écrites avec une police de caractères fixe et italique. L'exemple de déclaration de DTD ci-dessous signifie qu'une telle déclaration doit commencer par `<!DOCTYPE` suivi du nom de l'élément racine du document qui peut être un nom quelconque, différent de `root-element`.

```
<!DOCTYPE root-element ... >
```

L'écriture `...` signifie qu'une partie sans importance a été omise pour rendre la description plus concise et plus claire. Un exemple concret est obtenu en remplaçant *`root-element`* par `simple` et en complétant la partie omise.

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

Les références croisées entre les différents chapitres et sections de cet ouvrage jouent un rôle important. À chaque fois qu'un concept important est mentionné, le numéro de section où celui-ci est introduit est indiqué entre crochets de cette façon [Section 2.2]. Ces indications de navigation facilitent la compréhension des liens entre les différentes notions. Il a quelques fois été pris la liberté de mentionner des liens avec des concepts introduits plus tard dans l'ouvrage. Ces indications peuvent être ignorées dans une première lecture.

Les nombres sont généralement écrits en base décimale comme `123` ou `8364`. Lorsque l'écriture d'un nombre est en base hexadécimale, celle-ci commence par `x` ou `0x` comme `0x231` ou `x20AC`. L'exception à cette règle est l'écriture des points de code des caractères Unicode [Section 2.2] qui sont toujours écrits en hexadécimal précédés de `U+` comme `U+2023` ou `U+20AC`.

Chapitre 2. Syntaxe de XML

La syntaxe de XML est relativement simple. Elle nécessite un effort très modéré pour son apprentissage. Elle est constituée de quelques règles pour l'écriture d'une entête et des balises pour structurer les données. Ces règles sont très similaires à celles du langage HTML utilisé pour les pages WEB mais elles sont, en même temps, plus générales et plus strictes. Elles sont plus générales car les noms des balises sont libres. Elles sont aussi plus strictes car elles imposent qu'à toute balise ouvrante corresponde une balise fermante.

2.1. Premier exemple

Le langage XML est un format orienté texte. Un document XML est simplement une suite de caractères respectant quelques règles. Il peut être stocké dans un fichier et/ou manipulé par des logiciel en utilisant un codage des caractères. Ce codage précise comment traduire chaque caractère en une suite d'octets réellement stockés ou manipulés. Les codages possibles et leurs incidences sont décrits plus loin dans ce chapitre [Section 2.2]. Comme un document XML est seulement du texte, il peut être écrit comme l'exemple ci-dessous.

On commence par donner un premier exemple de document XML comme il peut être écrit dans un fichier `bibliography.xml`. Ce document représente une bibliographie de livres sur XML. Il a été tronqué ci-dessous pour réduire l'espace occupé. Ce document contient une liste de livres avec pour chaque livre, le titre, l'auteur, l'éditeur (publisher en anglais), l'année de parution, le numéro ISBN et éventuellement une URL.

```
<?xml version="1.0" encoding="iso-8859-1"?>❶
<!-- Time-stamp: "bibliography.xml   3 Mar 2008 16:24:04" -->❷
<!DOCTYPE bibliography SYSTEM "bibliography.dtd" >❸
<bibliography>❹
  <book key="Michard01" lang="fr">❺
    <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>❻
```

❶ Entête XML avec la version `1.0` et l'encodage `iso-8859-1` des caractères.
❷ Commentaire délimité par les chaînes de caractères `<!--` et `-->`.
❸ Déclaration de DTD externe dans le fichier `bibliography.dtd`.
❹ Balise ouvrante de l'élément racine `bibliography`
❺ Balise ouvrante de l'élément `book` avec deux attributs de noms `key` et `lang` et de valeurs `Michard01` et `fr`
❻ Balise fermante de l'élément racine `bibliography`

2.2. Caractères

Un document XML est une suite de caractères. Les caractères qui peuvent être utilisés sont ceux définis par la norme Unicode ISO 10646 [W] aussi appelée UCS pour *Universal Character Set*. Cette norme recense tous les

caractères des langues connues et tous les symboles utilisés dans les différentes disciplines. Elle nomme tous ces caractères et symboles et leur attribue un code sur 32 bits (4 octets) appelé simplement *code Unicode* ou *point de code* dans la terminologie Unicode. Dans la pratique, tous les points de code attribués à des caractères se situent dans l'intervalle de 0 à 0x10FFFF et ils utilisent donc au plus 21 bits. Cette longueur maximale ne sera pas dépassée avant longtemps car il reste encore de nombreux points de code non attribués dans cet intervalle pour des usages futurs. Unicode peut être vu comme un catalogue de tous les caractères disponibles. Un caractère dont le point de code est *n* est désigné par U+*n* où le nombre *n* est écrit en hexadécimal. L'écriture hexadécimale de *n* n'est pas préfixée du caractère 'x' car c'est implicite après les deux caractères 'U+'. Le caractère Euro '€' est, par exemple, désigné par U+20AC car son point de code est 8364 = 0x20AC. Le sous-ensemble des caractères Unicode dont les points de code tiennent sur 16 bits (2 octets), c'est-à-dire entre 0 et 65535 = 0xFFFF est appelé BMP pour *Basic Multilingual Plane*. Il couvre largement la très grande majorité des langues usuelles et les symboles les plus courants.

2.2.1. Caractères spéciaux

Les cinq caractères '<' U+2C, '>' U+2E, '&' U+26, ''' U+27 et '"' U+22 ont une signification particulière dans les documents XML. Les deux caractères '<' et '>' servent à délimiter les balises [Section 2.7.1], ainsi que les commentaires [Section 2.7.5] et les instructions de traitement [Section 2.7.6]. Le caractère '&' marque le début des références aux entités générales [Section 3.5.1]. Pour introduire ces caractères dans le contenu du document, il faut utiliser des sections littérales [Section 2.7.2] ou les entités prédéfinies [Section 3.5.1.2]. Les caractères ''' et '"' servent également de délimiteurs, en particulier pour les valeurs des attributs [Section 2.7.3]. Dans ces cas, il faut encore avoir recours aux entités prédéfinies pour les introduire.

2.2.2. Caractères d'espacement

Le traitement XML des caractères d'espacement est à la fois simple dans une première approche et subtil et source de surprises dans un second temps. Les caractères d'espacement sont l'espace ' ' U+20, la tabulation U+09 ('\t' en notation du langage C), le saut de ligne U+0A ('\n' en C) et le retour chariot U+0D ('\r' en C). Le traitement de ces caractères est indiqué aux applications par l'attribut xml:space [Section 2.7.4.2].

Les fins de lignes sont normalisées par l'analyseur lexical (*parser* en anglais). Ceci signifie que les différentes combinaisons de fin de ligne sont remplacées par un seul caractère U+0A avant d'être transmises à l'application. Cette transformation garantit une indépendance vis à vis des différents systèmes d'exploitation. Les combinaisons remplacées par cette normalisation sont les suivantes.

la suite des deux caractères U+0D U+0A
la suite des deux caractères U+0D U+85
le caractère U+85 appelé *Next Line*
le caractère U+2028 appelé *Line Separator*
le caractère U+0D non suivi de U+0A ou U+85

Les deux caractères U+85 et U+2028 ne peuvent être correctement décodés qu'après la déclaration de l'encodage des caractères par l'entête [Section 2.6.1]. Leur usage dans l'entête est donc déconseillé.

2.2.3. Jetons et noms XML

Les identificateurs sont utilisés en XML pour nommer différents objets comme les éléments, les attributs, les instructions de traitement. Ils servent aussi à identifier certains éléments par l'intermédiaire des attributs de type ID. XML distingue deux types d'identificateurs appelés *jetons* (*name token* abrégé en NMTOKEN dans la terminologie XML) et *noms XML* dans cet ouvrage.

Les caractères autorisés dans les identificateurs sont tous les caractères alphanumériques, c'est-à-dire les lettres minuscules [a-z], majuscules [A-Z] et les chiffres [0-9] ainsi que le tiret '-' U+2D, le point '.' U+2E, les deux points ':' U+3A et le tiret souligné '_' U+5F. Un *jeton* est une suite quelconque de ces caractères. Un *nom XML* est un jeton qui, en outre, commence par une lettre [a-zA-Z], le caractère ':' ou le caractère '_'. Les deux caractères '-' et '.' ne peuvent pas apparaître au début des noms. Il n'y a pas, a priori, de limite à la taille des identificateurs mais certains logiciels peuvent en imposer une dans la pratique.

Le caractère `':'` est réservé à l'utilisation des espaces de noms [Chapitre 4]. De fait, il ne peut apparaître qu'une seule fois pour séparer un préfixe du nom local dans les noms des éléments et des attributs. Les espaces de noms amènent à distinguer les noms ayant un caractère `':'`, appelés *noms qualifiés* et les autres, appelés par opposition *noms non qualifiés*.

Les noms commençant par les trois lettres `xml` en minuscule ou majuscule, c'est-à-dire par une chaîne de `[xX][mM][lL]` sont réservés aux usages internes de XML. Ils ne peuvent pas être utilisés librement dans les documents mais ils peuvent cependant apparaître pour des utilisations spécifiques prévues par la norme. Les noms commençant par `xml:` comme `xml:base` font partie de l'espace de noms XML [Section 4.7].

Quelques exemples d'identificateurs sont donnés ci-dessous.

Noms XML valides :
`name`, `id-42`, `xsl:template`, `sec.dtd-3.1` et `_special_`

Jetons qui ne sont pas des noms :
`-name`, `42`, `42-id` et `.sect.`

Noms réservés :
`xml:id` et `xml-stylesheet`

La norme XML 1.1 prévoit que tout caractère Unicode de catégorie lettre peut apparaître dans les identificateurs. Il est, par exemple, possible d'avoir des noms d'éléments avec des caractères accentués. Il est cependant conseillé de se limiter aux caractères ASCII de `[a-zA-Z]` pour assurer une meilleure compatibilité. Beaucoup de logiciels ne gèrent pas les autres caractères dans les identificateurs.

2.2.4. Codage

Chaque caractère possède un point de code sur 32 bits mais un document ne contient pas directement ces points de code des caractères. Ce codage serait inefficace puisque chaque caractère occuperait 4 octets. Chaque document utilise un codage pour écrire les points de code des caractères. Il existe différents codages dont le codage par défaut UTF-8. Certains codages permettent d'écrire tous les points de code alors que d'autres permettent seulement d'écrire un sous-ensemble comme le BMP. Le codage utilisé par un document est indiqué dans l'entête [Section 2.6.1] de celui-ci. Les principaux codages utilisés par les documents XML sont décrits ci-dessous.

US-ASCII

Ce codage permet uniquement de coder les points de code de 0 à 0x7F des caractères ASCII.

UCS-4 ou UTF-32 [W]

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

UCS-2 [W]

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

UTF-16 [W]

Ce codage coïncide essentiellement avec UCS-2 à l'exception de la plage de 2048 positions de 0xD800 à 0xDFFF qui permet de coder des caractères en dehors du BMP dont le point de code utilise au plus 20 bits. L'exclusion de cette plage ne pose aucun problème car elle ne contient aucun point de code attribué à un caractère. Un point de code ayant entre 17 et 20 bits est scindé en deux blocs de 10 bits répartis sur une paire de mots de 16 bits. Le premier bloc de 10 bits est préfixé des 6 bits `110110` pour former un premier mot de 16 bits et le second bloc de 10 bits est préfixé des 6 bits `110111` pour former un second mot de 16 bits.

Représentation UTF-16	**Signification**
`xxxxxxxx xxxxxxxx`	2 octets codant 16 bits
`110110``zz zzxxxxxx`	4 octets codant 20 bits

Représentation UTF-16	Signification
110111xx xxxxxxxx	yyyy xxxxxxxx xxxxxxxx où zzzz = yyyy-1

Tableau 2.1. Codage UTF-16

Le symbole de l'Euro '€' U+20AC, est, par exemple, codé par les deux octets x20 xAC = 00100000 10101100 puisqu'il fait partie du BMP. Le symbole de la croche '𝅘𝅥𝅯' U+1D160 est codé par les 4 octets xD8 x34 xDD x60 = **11011**000 00110100 **11011**101 01100000.

UTF-8 [W]

Ce codage est le codage par défaut de XML. Chaque caractère est codé sur un nombre variable de 1 à 4 octets. 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 peut uniquement coder des points de code ayant au maximum 21 bits mais tous les points de code attribués ne dépassent pas cette longueur.

Représentation UTF-8	Signification
0xxxxxxx	1 octet codant 7 bits
110xxxxx **10**xxxxxx	2 octets codant 8 à 11 bits
1110xxxx **10**xxxxxx **10**xxxxxx	3 octets codant 12 à 16 bits
11110xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx	4 octets codant 17 à 21 bits

Tableau 2.2. Codage UTF-8

Le symbole de l'Euro '€' U+20AC, est, par exemple, codé par les trois octets xE2 x82 xAC = **1110**0010 **10**000010 **10**101100. Le symbole de la croche '𝅘𝅥𝅯' U+1D160 est, quant à lui, codé par les 4 octets xF0 x9D x85 xA0 = **11110**000 **10**011101 **10**000101 **10**100000. Ce codage a l'avantage d'être relativement efficace pour les langues européennes qui comportent beaucoup de caractères ASCII. Il est, en revanche, peu adapté aux langues asiatiques dont les caractères nécessitent 3 octets alors que 2 octets sont suffisants avec UTF-16.

ISO-8859-1 (appelé Latin-1) [W]

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

ISO-8859-15 (appelé Latin-9 ou Latin-0) [W]

Ce codage est une mise à jour du codage ISO-8859-1 dont il diffère uniquement en 8 positions. Les caractères '€', 'Š', 'š', 'Ž', 'ž', 'Œ', 'œ' et 'Ÿ' remplacent les caractères '¤' U+A4, '¦' U+A6, '¨' U+A8, '´' U+B4, '¸' U+B8, '¼' U+BC, '½' U+BD et '¾' U+BE moins utiles pour l'écriture des langues européennes. Le symbole de l'Euro '€' U+20AC, est, par exemple, codé par l'unique octet xA4 = 10100100.

Le tableau suivant donne la suite d'octets pour le mot Hétérogène pour quelques codages classiques. Comme les points de code de x00 à xFF d'Unicode coïncident avec le codage des caractères de ISO-8859-1, le codage en UTF-16 est obtenu en insérant un octet nul 00 avant chaque octet du codage en ISO-8859-1. Le codage en UTF-32 est obtenu en insérant dans le codage en UTF-16 deux octets nuls avant chaque paire d'octets.

Codage	Séquence d'octets en hexadécimal pour Hétérogène
UTF-8	48 C3 A9 74 C3 A9 72 6F 67 C3 A8 6E 65
ISO-8859-1	48 E9 74 E9 72 6F 67 E8 6E 65
UTF-16	00 48 00 E9 00 74 00 E9 00 72 … 00 E8 00 6E 00 65
UTF-32	00 00 00 48 00 00 00 E9 00 00 00 74 … 00 00 00 65

Tableau 2.3. Comparaison des codages

Les logiciels manipulant des documents XML doivent obligatoirement gérer les codages UTF-8 et UTF-16. Les autres codages sont facultatifs. Il est essentiel que le codage d'un document soit indiqué dans l'entête [Section 2.6.1] du document. Si un navigateur interprète, par exemple, la suite d'octets du codage UTF-8 du mot `Hétérogène` comme un codage en ISO-8859-1, il affiche la chaîne `HÃ©tÃ©rogÃ¨ne`.

Bien que certains codages ne permettent pas de coder tous les points de code, il est néanmoins possible d'insérer n'importe quel caractère Unicode en donnant explicitement son point de code avec une des deux syntaxes suivantes. Ces syntaxes peuvent être utilisées pour n'importe quel point de code même si celui-ci peut être écrit avec le codage du document. Elles sont, en particulier, pratiques lorsque les éditeurs de texte affichent mal certains caractères. Les deux syntaxes prennent les formes `&#`*`point de code décimal`*`;` ou `&#x`*`point de code hexadécimal`*`;`. Le caractère Euro '€' peut par exemple être inséré par `&#8364;` ou `&#x20AC;`. Pour ces deux syntaxes, c'est le point de code du caractère qui est écrit en décimal ou en hexadécimal et non pas sa transcription dans le codage du document.

2.2.4.1. Marque d'ordre des octets

Pour les codages non basés sur les octets comme UTF-16 ou UCS-2, il est important de savoir dans quel ordre sont placés les deux octets de poids fort et de poids faible d'un mots de 16 bits. Il existe deux modes appelés *gros-boutiste* et *petit-boutiste* (*big endian* et *little endian* en anglais). Dans le mode gros-boutiste, l'octet de poids fort est placé avant l'octet de poids faible alors que dans le mode petit-boutiste, l'octet de poids fort est placé après l'octet de poids faible.

Afin de savoir quel est le mode utilisé dans un document XML, le document commence par le caractère `U+FEFF`. Ce caractère est appelé *espace insécable de largeur nulle* (*zero-width no-break space* en anglais) mais il a été remplacé, pour cet emploi, par le caractère `U+2060`. Il est maintenant uniquement utilisé comme marque d'ordre des octets (*Byte order mark* ou *BOM* en anglais). Cette marque est sans ambiguïté car il n'existe pas de caractère de point de code 0xFFFE. Le tableau suivant récapitule les séquences des premiers octets d'un document en fonction du codage et du mode gros-boutiste ou petit-boutiste. Les valeurs 0x3C, 0x3F et 0x78 sont les points de code des trois premiers caractères `'<'`, `'?'` et `'x'` de l'entête [Section 2.6.1].

Codage	**Mode**	**Séquence d'octets en hexadécimal**
UTF-8 sans BOM		`3C 3F 78`
UTF-8 avec BOM		`EF BB BF 3C 3F 78`
UTF-16 ou UCS-2	gros-boutiste	`FE FF 00 3C 00 3F`
UTF-16 ou UCS-2	petit-boutiste	`FF FE 3C 00 3F 00`
UTF-32	gros-boutiste	`00 00 FE FF 00 00 00 3C`
UTF-32	petit-boutiste	`FE FF 00 00 3C 00 00 00`

Tableau 2.4. Marques d'ordre des octets

Bien que la marque d'ordre des octets puisse être mise dans un document codé en UTF-8, celle-ci est inutile et il est déconseillé de la mettre. La marque d'ordre des octets ne peut pas être mise dans un document codé en ISO-8859-1.

2.2.5. Collations

Certaines ligatures comme le caractère `'œ' U+153` sont considérées par Unicode comme un seul caractère plutôt que comme la fusion des deux caractères `'oe'`. Il s'ensuit que les deux mots `cœur` et `coeur` sont, a priori, considérés comme distincts. Ce problème est résolu par l'utilisation de *collations* lors du traitement des documents. Une collation est une collection de règles qui établissent des équivalences entre des caractères ou des suites de caractères. Une collation peut, par exemple, déclarer que le caractère `'œ' U+153` est équivalent aux deux caractères `'oe'` ou que la lettre `'ß' U+DF` est équivalente aux deux lettres `'ss'`. Une collation établit aussi l'ordre des caractères utilisé pour l'ordre lexicographique. Elle peut, par exemple, déclarer que le caractère `'é'` se place entre les caractères `'e'` et `'f'`. La collation par défaut est basée sur les points de code des caractères. Le caractère `'é' U+E9` se trouve, pour cette collation, après le caractère `'z' U+7A` et le mot `zèbre` est donc avant le mot `étalon` dans l'ordre lexicographique.

2.2.6. Normalisation

Le même caractère peut avoir plusieurs points de code. Cette ambiguïté provient du fait qu'Unicode a été construit en fusionnant plusieurs codages et qu'il tente de rester compatible avec chacun d'eux. Le caractère `'µ'` est en même temps le caractère `U+B5` qui provient de Latin-1 et le caractère `U+3BC` qui provient du bloc des caractères grecs. D'autres caractères peuvent avoir un point de code mais peuvent, en même temps, correspondre à une suite de plusieurs points de code. Le caractère `'ï'` est, par exemple, le caractère `U+EF` mais il correspond également à la suite `U+69 U+A8` formée du caractère `'i'` suivi du caractère spécial tréma `'¨' U+A8`. Ce codage multiple conduit à des problèmes, en particulier pour la comparaison des chaînes de caractères. Pour palier à ce problème, Unicode introduit des *normalisations* qui transforment les différents codages en un codage canonique. La normalisation la plus standard est la normalisation C. Celle-ci transforme, par exemple, la suite de caractères `U+69 U+A8` en le caractère `'ï' U+EF`. La normalisation d'une chaîne de caractères peut être obtenue avec la fonction XPath `normalize-unicode()` [Section 6.3.3].

2.3. URI, URL et URN

XML et toutes les technologies autour de XML font un grand usage des URI et plus particulièrement des URL. Ceux-ci sont, par exemple, employés pour référencer des documents externes comme des DTD [Chapitre 3] ou pour identifier des espaces de noms [Chapitre 4]. Les URL sont bien connues car elles sont utilisées au quotidien pour naviguer sur le WEB. La terminologie XML distingue également les URI et les URN. Les significations exactes de ces trois termes dans la terminologie XML sont les suivantes.

URI
: Uniform Resource Identifier

URL
: Uniform Resource Locator

URN
: Uniform Resource Name

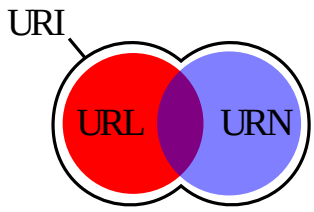


Figure 2.1. Relations entre URI, URL et URN

La notion la plus générale est celle d'URI. Les URI comprennent les URL et les URN même si certains URI peuvent être simultanément des URL et des URN. Les liens entre ces différents termes sont illustrés à la figure. Un URI est un identifiant qui permet de désigner sans ambiguïté un document ou plus généralement une ressource. Cet identifiant doit donc être unique de manière universelle. Une URL identifie un document en spécifiant un mécanisme pour le retrouver. Elle est composée d'un protocole suivi d'une adresse permettant de récupérer le document avec le protocole. Un URN est, au contraire, un nom donné à un document indépendamment de la façon d'accéder au document. Un exemple typique d'URN est l'URN formé à partir du numéro ISBN d'un livre comme `urn:isbn:978-2-7117-2077-4`. Cet URN identifie le livre *Langages formels, calculabilité et complexité* mais n'indique pas comment l'obtenir.

La syntaxe générale des URI prend la forme *`scheme:ident`* où *`scheme`* est un *schéma* d'URI et où *`ident`* est un identifiant obéissant à une syntaxe propre au schéma *`scheme`*. Chaque schéma définit un sous-espace des URI. Dans le cas d'une URL, le schéma est un protocole d'accès au document comme `http`, `sip`, `imap` ou `ldap`. Le schéma utilisé pour tous les URN est `urn`. Il est généralement suivi de l'identificateur d'un espace de noms comme `isbn`. Des exemples d'URI sont donnés ci-dessous. Les deux derniers URI de la liste sont des URN.

```
http://www.liafa.jussieu.fr/~carton/
```

```
sftp://carton@liafa.jussieu.fr
tel:+33-1-57-27-92-54
sip:0957279254@freephonie.net
file://home/carton/Enseignement/XML/Cours/XSLT
urn:oasis:names:tc:docbook:dtd:xml:docbook:5.1
urn:publicid:-:W3C:DTD+HTML+4.0:EN
```

Dans la pratique, la plupart des URI utilisées sont des URL et les deux termes peuvent (presque) être considérés comme synonymes.

2.3.1. Résolution d'URI

Un URI appelé *URI de base* est souvent attaché à un document ou à un fragment d'un document XML. Cet URI est généralement une URL. Il sert à résoudre les URL contenues dans le fragment de document, qu'elles soient relatives ou absolues. Cette *résolution* consiste à combiner l'URL de base avec ces URL pour obtenir des URL absolues qui permettent de désigner des documents externes.

Pour comprendre comment une URL de base se combine avec une URL, il faut d'abord comprendre la structure d'une URL. La description donnée ci-dessous se limite aux aspects indispensables pour appréhender la résolution des URL. Chaque URL se décompose en trois parties.

Protocole d'accès
: Une URL commence obligatoirement par le nom d'un protocole d'accès suivi du caractère `':'`. Les principaux protocoles sont `http`, `https`, `ftp` et `file`.

Adresse Internet
: Le protocole est suivi d'une adresse Internet qui commence par les deux caractères `'//'`. Cette adresse est absente dans le cas du protocole `file`.

Chemin d'accès
: L'URL se termine par un chemin d'accès dans l'arborescence des fichiers. Ce chemin se décompose lui-même en le nom du répertoire et le nom du fichier. Ce dernier est formé de tous les caractères après le dernier caractère `'/'`.

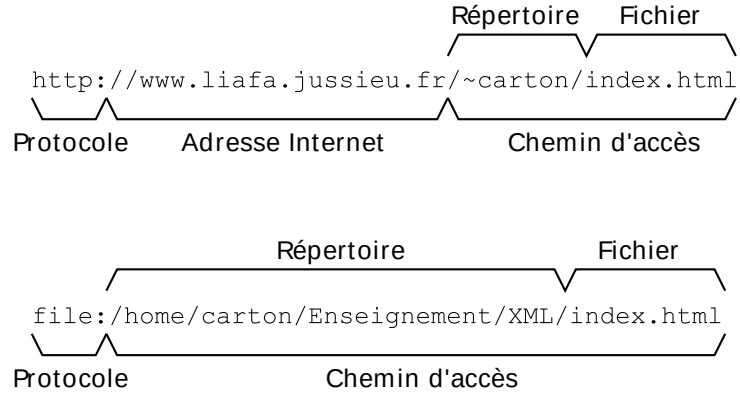


Figure 2.2. Structure d'une URL

La combinaison d'une URL `url` avec une URL de base `base` pour former une URL complète est réalisée de la façon suivante qui dépend essentiellement de la forme de l'URL `url`.

1. Si l'URL `url` est elle-même une URL complète qui commence par un protocole, le résultat de la combinaison est l'URL `url` sans tenir compte de l'URL `base`.

2. Si l'URL `url` est un chemin absolu commençant par le caractère `'/'`, le résultat est obtenu en remplaçant la partie chemin de l'URL `base` par l'URL `url`. L'URL `url` est donc ajoutée après la partie adresse Internet de l'URL `base`.

3. Si l'URL `url` est un chemin relatif ne commençant pas par le caractère `'/'`, le résultat est obtenu en remplaçant le nom du fichier de l'URL `base` par l'URL `url`. Le chemin relatif est donc concaténé avec le nom du répertoire.

Les exemples ci-dessous illustrent les différents cas. On suppose que l'URL `base` est fixée égale à l'URL `http://www.somewhere.org/Teaching/index.html`. Pour chacune des valeurs de l'URL `url`, on donne la valeur de l'URL obtenue par combinaison de l'URL `base` avec l'URL `url`.

`url=""` (chaîne vide)
   `http://www.somewhere.org/Teaching/index.html`

`url="XML/chapter.html"` (chemin relatif)
   `http://www.somewhere.org/Teaching/XML/chapter.html`

`url="XML/XPath/section.html"` (chemin relatif)
   `http://www.somewhere.org/Teaching/XML/XPath/section.html`

`url="/Course/section.html"` (chemin absolu)
   `http://www.somewhere.org/Course/section.html`

`url="http://www.elsewhere.org/section.html"` (URL complète)
   `http://www.elsewhere.org/section.html`

2.4. Syntaxe et structure

Il y a, en français, l'orthographe et la grammaire. La première est constituée de règles pour la bonne écriture des mots. La seconde régit l'agencement des mots dans une phrase. Pour qu'une phrase en français soit correcte, il faut d'abord que les mots soient bien orthographiés et, ensuite, que la phrase soit bien construite. Il y aurait encore le niveau sémantique mais nous le laisserons de côté. XML a également ces deux niveaux. Pour qu'un document XML soir correct, il doit d'abord être *bien formé* et, ensuite, être *valide*. La première contrainte est de nature *syntaxique*. Un document bien formé doit respecter certaines règles syntaxiques propres à XML qui sont explicitées dans ce chapitre. Il s'agit, en quelque sorte, de l'orthographe d'XML. La seconde contrainte est de nature *structurelle*. Un document valide doit respecter un *modèle de document*. Un tel modèle décrit de manière rigoureuse comment doit être organisé le document. Un modèle de documents peut être vu comme une grammaire pour des documents XML. La différence essentielle avec le français est que la grammaire d'XML n'est pas figée. Pour chaque application, il est possible de choisir la grammaire la plus appropriée. Cette possibilité d'adapter la grammaire aux données confère une grande souplesse à XML. Il existe plusieurs langages pour écrire des modèles de document. Les DTD (*Document Type Description*), héritées de SGML, sont simples mais aussi assez limitées. Elles sont décrites au chapitre suivant [Chapitre 3]. Les schémas XML sont beaucoup plus puissants. Ils sont décrits dans un autre chapitre [Chapitre 5].

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 [Section 3.5.1.4] ou XInclude [Section 2.9]. Les fichiers contenant des documents dans un dialecte XML peuvent avoir une autre extension qui précise le format. Les extensions pour les schémas XML [Chapitre 5], les feuilles de style XSLT [Chapitre 8], les dessins en SVG [Chapitre 11] sont, par exemple, `.xsd`, `.xsl` et `.svg`.

Un document XML est, la plupart du temps, stocké dans un fichier mais il peut aussi être dématérialisé et exister indépendamment de tout fichier. Il peut, par exemple, exister au sein d'une application qui l'a construit. Une chaîne de traitement de documents XML peut produire des documents intermédiaires qui sont détruits à la fin. Ces documents existent uniquement pendant le traitement et sont jamais mis dans un fichier.

2.5. Composition globale d'un document

La composition globale d'un document XML est immuable. Elle comprend toujours les constituants suivants.

Prologue
   Il contient des déclarations facultatives.

Corps du document
: C'est le contenu même du document.

Commentaires et instructions de traitement
: Ceux-ci peuvent apparaître partout dans le document, dans le prologue et le corps.

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-element>          ⎫
  ...                   ⎪   Corps
</root-element>         ⎭
```

Dans l'exemple donné au début de ce chapitre, le prologue comprend les trois premières lignes du fichier. La première ligne est l'entête XML et la deuxième est simplement un commentaire utilisé par Emacs pour mémoriser le nom du fichier et sa date de dernière modification. La troisième ligne est la déclaration d'une DTD externe contenue dans le fichier `bibliography.dtd`. Le corps du document commence à la quatrième ligne du fichier avec la balise ouvrante `<bibliography>`. Il se termine à la dernière ligne de celui-ci avec la balise fermante `</bibliography>`.

2.6. Prologue

Le prologue contient deux déclarations facultatives mais fortement conseillées ainsi que des commentaires [Section 2.7.5] et des instructions de traitement [Section 2.7.6]. La première déclaration est l'entête XML qui précise entre autre la version de XML et le codage du fichier. La seconde déclaration est la déclaration du type du document (DTD) qui définit la structure du document. La déclaration de type de document est omise lorsqu'on utilise des schémas XML [Chapitre 5] ou d'autres types de modèles qui remplacent les DTD. La structure globale du prologue est la suivante. Dans le prologue, tous les caractères d'espacement [Section 2.2.2] sont interchangeables mais l'entête est généralement placée, seule, sur la première ligne du fichier.

```
<?xml ... ?>                ⎫   Entête XML      ⎫
<!DOCTYPE root-element [    ⎫                   ⎪   Prologue
  ...                       ⎪   DTD             ⎪
]>                          ⎭                   ⎭
```

Les différentes parties du prologue sont détaillées dans les sections suivantes.

2.6.1. Entête XML

L'entête utilise une syntaxe `<?xml ... ?>` semblable à celle des instructions de traitement [Section 2.7.6] bien qu'elle ne soit pas véritablement une instruction de traitement. L'entête XML a la forme générale suivante.

```
<?xml version="..." encoding="..." standalone="..."?>
```

L'entête doit se trouver au tout début du document. Ceci signifie que les trois caractères `'<?x'` doivent être les trois premiers caractères du document, éventuellement précédés d'une marque d'ordre des octets [Section 2.2.4.1].

Cette entête peut contenir trois attributs `version`, `encoding` et `standalone`. Il ne s'agit pas véritablement d'attributs [Section 2.7.3] car ceux-ci sont réservés aux éléments mais la syntaxe identique justifie ce petit abus de langage. Chaque attribut a une valeur délimitée par une paire d'apostrophes `'''` ou une paire de guillemets `'"'`.

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 [Section 2.2.4] 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 comme `iso-8859-1` ou `utf-8`. L'attribut `standalone` précise si le fichier est autonome, c'est-à-dire s'il existe des déclarations externes qui affectent le document. La valeur de cet attribut peut être `yes` ou `no` et sa valeur par défaut est `no`. Les déclarations externes peuvent provenir d'une DTD externe [Section 3.2.2] où d'entités paramètres [Section 3.5.2]. Elles peuvent affecter le contenu du document en donnant, par exemple, des valeurs par défaut [Section 3.7.3] à des attributs. La valeur de l'attribut `standalone` influence également la prise en compte des caractères d'espacement dans les contenus purs [Section 3.6.1.1] lors de la validation par une DTD.

L'attribut `version` est obligatoire et l'attribut `encoding` l'est aussi dès que le codage des caractères n'est pas le codage par défaut UTF-8. Quelques exemples d'entête XML sont donnés ci-dessous.

```
<?xml version="1.0"?>
<?xml version='1.0' encoding='UTF-8'?>
<?xml version="1.1" encoding="iso-8859-1" standalone="no"?>
```

Lorsqu'un document est scindé en plusieurs fragments dans différents fichiers inclus par des entités externes [Section 3.5.1.4] ou par XInclude [Section 2.9], chacun des fragments peut commencer par une entête. L'intérêt est de pouvoir spécifier un codage des caractères différent.

2.6.2. Déclaration de type de document

La déclaration de type définit la structure du document. Elle précise en particulier quels éléments peut contenir chacun des éléments. Cette déclaration de type peut prendre plusieurs formes suivant que la définition du type est interne, c'est-à-dire incluse dans le document ou externe. Elle a la forme générale suivante qui utilise le mot clé `DOCTYPE`.

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

La forme précise de cette déclaration est explicitée au chapitre consacré aux DTD [Chapitre 3].

2.7. Corps du document

Le corps du document est constitué de son contenu qui est organisé de façon hiérarchique à la manière d'un système de fichiers à l'exception qu'aucune distinction n'est faite entre fichiers et répertoires. L'unité de cette organisation est l'*élément*. Chaque élément peut contenir du texte simple, comme un fichier, d'autres éléments, comme un répertoire, ou encore un mélange des deux.

Comme dans une arborescence de fichiers, il y a un élément appelé *élément racine* qui contient l'intégralité du document.

2.7.1. Éléments

Figure 2.3. Composition d'un élément

Un *élément* est formé d'une balise ouvrante, d'un contenu et de la balise fermante correspondante. La *balise ouvrante* prend la forme <*name*> formée du caractère `'<'` U+3C, du nom *name* de l'élément et du caractère `'>'` U+3E. Des attributs [Section 2.7.3] peuvent éventuellement être ajoutés entre le nom et le caractère `'>'`. La *balise fermante* prend la forme </*name*> formée des deux caractères `'</'` U+3C et U+2F, du nom *name* de l'élément et du caractère `'>'`. Les noms des éléments sont des noms XML [Section 2.2.3] quelconques. Ils ne sont pas limités à un ensemble fixé de noms prédéfinis comme en HTML. Le *contenu* d'un élément est formé de tout ce qui se trouve entre la balise ouvrante et la balise fermante (cf. figure). Il peut être constitué de texte, d'autres éléments, de commentaires [Section 2.7.5] et d'instructions de traitement [Section 2.7.6].

Dans la balise ouvrante, le caractère `'<'` doit être immédiatement suivi du nom de l'élément. En revanche, il peut y avoir des espaces entre le nom et le caractère `'>'`. La balise fermante ne peut pas contenir d'espace.

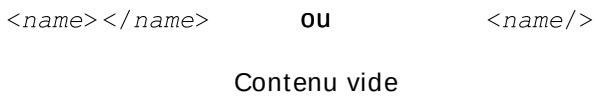


Figure 2.4. Élément avec un contenu vide

Lorsque le contenu est vide, c'est-à-dire lorsque la balise fermante suit immédiatement la balise ouvrante, les deux balises peuvent éventuellement se contracter en une seule balise de la forme `<name/>` formée du caractère `'<'`, du nom *name* et des deux caractères `'/>'`. Cette contraction est à privilégier lorsque l'élément est déclaré vide par une DTD [Section 3.6.4].

Figure 2.5. Imbrication des éléments

Comme chaque élément possède une balise ouvrante et une balise fermante, les balises vont nécessairement par paire. À toute balise ouvrante correspond une balise fermante et inversement. L'imbrication des balises doit, en outre, être correcte. Si deux éléments `tag1` et `tag2` ont un contenu commun, alors l'un doit être inclus dans l'autre. Autrement dit, si la balise ouvrante `<tag2>` se trouve entre les deux balises `<tag1>` et `<tag1/>`, alors la balise fermante `</tag2>` doit aussi se trouver entre les deux balises `<tag1>` et `<tag1/>` (cf. figure).

```
<parent>
  <sibling1> ... </sibling1>
  <sibling2> ... </sibling2>
  <self>
    <child1> ... <desc1></desc1> ... <desc2></desc2> ... </child1>
    <child2> ... </child2>
    <child3> ... <desc3><desc4> ... </desc4></desc3> ... </child3>
  </self>
  <sibling3> ... </sibling3>
</parent>
```

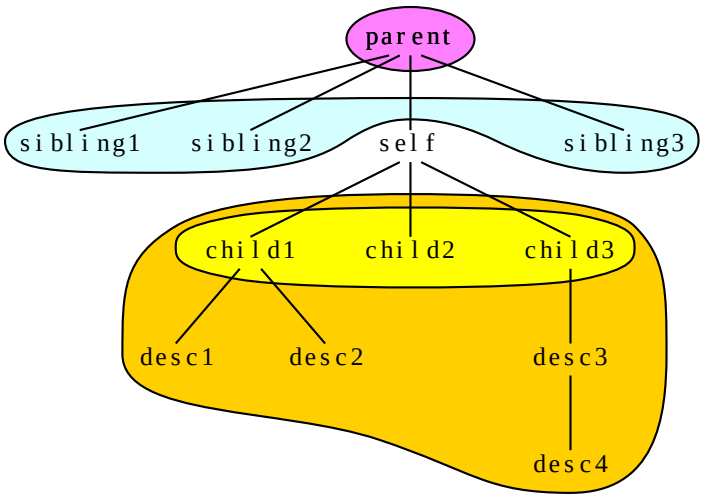


Figure 2.6. Liens de parenté

Dans l'exemple ci-dessus, le contenu de l'élément `self` s'étend de la balise ouvrante `<child1>` jusqu'à la balise fermante `</child3>`. Ce contenu comprend tous les éléments `child1`, `child2` et `child3` ainsi que les éléments `desc1`, `desc2`, `desc3` et `desc4`. Tous les éléments qu'il contient sont appelés *descendants* de l'élément `self`. Parmi ces descendants, les éléments `child1`, `child2` et `child3` qui sont directement inclus dans `self` sans élément intermédiaire sont appelés les *enfants* de l'élément `self`. Inversement, l'élément `parent` qui contient directement `self` est appelé le *parent* de l'élément `self`. Les autres éléments qui contiennent l'élément `self` sont appelés les *ancêtres* de l'élément `self`. Les autres enfants `sibling1`, `sibling2` et `sibling3` de l'élément `parent` sont appelés les *frères* de l'élément `self`. Ces relations de parenté entre les éléments peuvent être visualisées comme un arbre généalogique (cf. figure).

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 [Section 3.2] si celle-ci est présente. L'élément `bibliography` est l'élément racine de l'exemple donné au début du chapitre.

```
  ...            ]  Commentaires et instructions de traitement  ]
<root-element>   ]  Balise ouvrante de l'élément racine         |   Corps
  ...            ]  Éléments, commentaires et                   |    du
  ...            ]  instructions de traitement                  |  document
</root-element>  ]  Balise fermante de l'élément racine         |
  ...            ]  Commentaires et instructions de traitement  ]
```

2.7.2. Sections littérales

Les caractères spéciaux `'<'`, `'>'` et `'&'` ne peuvent pas être inclus directement dans le contenu d'un document. Ils peuvent être inclus par l'intermédiaire des entités prédéfinies [Section 3.5.1.2].

Il est souvent fastidieux d'inclure beaucoup de caractères spéciaux à l'aide des entités. Les *sections littérales*, appelées aussi *sections CDATA* en raison de leur syntaxe, permettent d'inclure des caractères qui sont recopiés à l'identique. Une section littérale commence par la chaîne de caractères `'<![CDATA['` et se termine par la chaîne `']]>'`. Tous les caractères qui se trouvent entre ces deux chaînes font partie du contenu du document, y compris les caractères spéciaux.

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

Une section CDATA ne peut pas contenir la chaîne de caractères `']]>'` qui permet à l'analyseur lexical de détecter la fin de la section. Il est en particulier impossible d'imbriquer des sections CDATA.

2.7.3. Attributs

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

```
<tag attribute="value"> ... </tag>
<tag attribute1="value1" attribute2="value2"> ... </tag>
```

Voici ci-dessous d'autres exemples concrets de balises ouvrantes avec des attributs. Ces exemples sont respectivement tirés d'un document XHTML, d'un schéma XML [Chapitre 5] et d'une feuille de style XSLT [Chapitre 8].

```
<body background='yellow'>
<xsd:element name="bibliography" type="Bibliography">
<a href="#{$node/@idref}">
```

Lorsque le contenu de l'élément est vide et que la balise ouvrante et la balise fermante sont contractées en une seule balise [Section 2.7.1], celle-ci peut contenir des attributs comme la balise ouvrante.

```
<hr style="color:red; height:15px; width:350px;" />
<xsd:attribute name="key" type="xsd:NMTOKEN" use="required"/>
```

Le nom de chaque attribut doit être un nom XML [Section 2.2.3]. La valeur d'un attribut peut être une chaîne quelconque de caractères délimitée par une paire d'apostrophes `'''` ou une paire de guillemets `'"'`. Elle peut contenir les caractères spéciaux `'<'`, `'>'`, `'&'`, `'''` et `'"'` mais ceux-ci doivent nécessairement être introduits par les entités prédéfinies [Section 3.5.1.2]. Si la valeur de l'attribut est délimitée par des apostrophes `'''`, les guillemets `'"'` peuvent être introduits directement sans entité et inversement.

```
<xsl:value-of select="key('idchapter', @idref)/title"/>
<xsl:if test='@quote = "&apos;"'>
```

Comme des espaces peuvent être présents dans la balise après le nom de l'élément et entre les attributs, l'indentation est libre pour écrire les attributs d'une balise ouvrante. Aucun espace ne peut cependant séparer le caractère `'='` du nom de l'attribut et de sa valeur. Il est ainsi possible d'écrire l'exemple générique suivant.

```
<tag attribute1="value1"
     attribute2="value2"
     ...
     attributeN="valueN">
  ...
</tag>
```

L'ordre des attributs n'a pas d'importance. Les attributs d'un élément doivent avoir des noms distincts. Il est donc impossible d'avoir deux occurrences du même attribut dans une même balise ouvrante.

Le bon usage des attributs est pour les meta-données plutôt que les données elles-mêmes. Ces dernières 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 [W] spécifie la représentation numérique de la date et de l'heure.

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

C'est une question de style de mettre les données dans les attributs ou dans les contenus des éléments. Le nom complet d'un individu peut, par exemple, être réparti entre des éléments `firstname` et `surname` regroupés dans un élément `personname` comme dans l'exemple ci-dessous.

```
<personname id="I666">
  <firstname>Gaston</firstname>
  <surname>Lagaffe</surname>
</personname>
```

Les éléments `firstname` et `surname` peuvent être remplacés par des attributs de l'élément `personname` comme dans l'exemple ci-dessous. Les deux solutions sont possibles mais la première est préférable.

```
<personname id="I666" firstname="Gaston" surname="Lagaffe"/>
```

2.7.4. 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 [Section 4.7]. Lors de l'utilisation de schémas, ces attributs peuvent être déclarés en important [Section 5.14] le schéma à l'adresse `http://www.w3.org/2001/xml.xsd`.

Contrairement à l'attribut `xml:id`, les trois autres attributs `xml:lang`, `xml:space` et `xml:base` s'appliquent au contenu de l'élément. Pour cette raison, la valeur de cet attribut est héritée par les enfants et, plus généralement, les descendants. Ceci ne signifie pas qu'un élément dont le père a, par exemple, un attribut `xml:lang` a également un attribut `xml:lang`. Cela veut dire qu'une application doit prendre en compte la valeur de l'attribut `xml:lang`

pour le traitement l'élément mais aussi de ses descendants à l'exception, bien sûr, de ceux qui donnent une nouvelle valeur à cet attribut. Autrement dit, la valeur de l'attribut `xml:lang` à prendre en compte pour le traitement d'un élément est celle donnée à cet attribut par l'ancêtre (y compris l'élément lui-même) le plus proche. Pour illustrer le propos, le document suivant contient plusieurs occurrences de l'attribut `xml:lang`. La langue du texte est, à chaque fois, donnée par la valeur de l'attribut `xml:lang` le plus proche.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<book xml:lang="fr">
  <title>Livre en Français</title>
  <chapter>
    <title>Chapitre en Français</title>
    <p>Paragraphe en Français</p>
    <p xml:lang="en">Paragraph in English</p>
  </chapter>
  <chapter xml:lang="en">
    <title>Chapter in English</title>
    <p xml:lang="fr">Paragraphe en Français</p>
    <p>Paragraph in English</p>
  </chapter>
</book>
```

Ce qui a été expliqué pour l'attribut `xml:lang` vaut également pour deux autres attributs `xml:space` et `xml:base`. C'est cependant un peu différent pour l'attribut `xml:base` car la valeur à prendre en compte doit être calculée à partir de toutes les valeurs des attributs `xml:base` des ancêtres.

2.7.4.1. Attribut `xml:lang`

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 [W] (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 [W] séparé du code de langue par un caractère tiret `'-'`. L'attribut `xml:lang` est du type `xsd:language` [Section 5.5.1.2] qui est spécialement prévu pour cet attribut.

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

Dans le document donné en exemple au début du chapitre, chaque élément `book` a un attribut `lang`. Ce n'est pas l'attribut `xml:lang` qui a été utilisé car celui-ci décrit la langue des données contenues dans l'élément alors que l'attribut `lang` décrit la langue du livre référencé.

2.7.4.2. Attribut `xml:space`

L'attribut `xml:space` permet d'indiquer à une application le traitement des caractères d'espacement [Section 2.2.2]. Les deux valeurs possibles de cet attribut sont `default` et `preserve`.

L'analyseur lexical transmet les caractères d'espacement aux applications sans les modifier. La seule transformation effectuée est la normalisation des fins de lignes. Il appartient ensuite aux applications de traiter ces caractères de façon appropriée. La plupart d'entre elles considèrent de façon équivalente les différents caractères d'espacement. Ceci signifie qu'une fin de ligne est vue comme un simple espace. Plusieurs espaces consécutifs sont aussi considérés comme un seul espace. Ce traitement est généralement le traitement par défaut des applications. Si l'attribut `xml:space` a la valeur `preserve`, l'application doit, au contraire, respecter les différents caractères d'espacement. Les fins de ligne sont préservées et les espaces consécutifs ne sont pas confondus. L'attribut `xml:space` intervient, en particulier, dans le traitement des espaces par XSLT [Section 8.4.1].

2.7.4.3. Attribut `xml:base`

À chaque élément d'un document XML est associée une URI [Section 2.3] appelée *URI de base*. Celle-ci est utilisée pour résoudre les URL des entités externes, qui peuvent être, par exemple des fichiers XML ou des fichiers

multimédia (images, sons, vidéos). Dans le fragment de document XHTML ci-dessous, l'élément `img` référence un fichier image `element.png` par son attribut `src`.

```
<img src="element.png" alt="Élément"/>
```

L'attribut de `xml:base` permet de préciser l'URI de base d'un élément. Par défaut, l'URI de base d'un élément est hérité de son parent. L'URI de base de la racine du document est appelée *URI de base du document*. Elle est souvent fixée par l'application qui traite le document mais elle peut aussi provenir d'un attribut `xml:base` de l'élément racine. Lorsque le document provient d'un fichier local, c'est souvent le chemin d'accès à celui-ci dans l'arborescence des fichiers, comme `file:/home/carton/Teaching/XML/index.html`. Lorsque le document est, au contraire, téléchargé, l'URI de base du document est l'adresse Internet de celui-ci comme `http://www.liafa.jussieu.fr/~carton/index.html`.

Pour chaque élément, l'attribut `xml:base` permet de fixer une URI de base de façon absolue ou, au contraire, de la construire à partir de l'URI de base du parent. Le comportement dépend de la forme de la valeur de l'attribut. La valeur est combinée avec l'URI de base du parent en suivant les règles de combinaison de celles-ci [Section 2.3.1]. L'attribut `xml:base` est de type `xsd:anyURI` [Section 5.5.1.2].

L'attribut `xml:base` est indispensable pour réaliser des inclusions de fichiers externes avec XInclude [Section 2.9] lorsque ces fichiers sont situés dans un répertoire différent de celui réalisant l'inclusion.

Le document suivant illustre les différents cas pour la combinaison d'une URI avec une adresse. Pour chacun des éléments, l'URI de base est donnée.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<book xml:base="http://www.somewhere.org/Teaching/index.html">❶
  <chapter xml:base="XML/chapter.html">❷
    <section xml:base="XPath/section.html"/>❸
    <section xml:base="/Course/section.html"/>❹
    <section xml:base="http://www.elsewhere.org/section.html"/>❺
  </chapter>
</book>
```

❶ `http://www.somewhere.org/Teaching/index.html`
❷ `http://www.somewhere.org/Teaching/XML/chapter.html`
❸ `http://www.somewhere.org/Teaching/XML/XPath/section.html`
❹ `http://www.somewhere.org/Course/section.html`
❺ `http://www.elsewhere.org/section.html`

L'URI de base d'un élément est retournée par la fonction XPath `base-uri()`.

2.7.4.4. Attribut `xml:id`

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.

Comme les applications qui traitent les documents XML ne prennent pas en compte les modèles de document, sous forme de DTD ou de schéma, elles ne peuvent pas déterminer le type des attributs. Il leur est en particulier impossible de connaître les attributs de type `ID` qui permettent d'identifier et de référencer les éléments. L'attribut `xml:id` résout ce problème puisqu'il est toujours du type `xsd:ID` [Section 5.5.1.4] qui remplace le type `ID` dans les schémas XML.

2.7.5. Commentaires

Les commentaires sont délimités par les chaînes de caractères `'<!--'` et `'-->'` comme en HTML. Ils ne peuvent pas contenir la chaîne `'--'` formée de deux tirets `'-'` et ils ne peuvent donc pas être imbriqués. Ils peuvent être présents dans le prologue et en particulier dans la DTD [Section 3.4]. Ils peuvent aussi être placés dans le contenu de n'importe quel élément et après l'élément racine. En revanche, ils ne peuvent jamais apparaître à l'intérieur

d'une balise ouvrante ou fermante. Un exemple de document XML avec des commentaires partout où ils peuvent apparaître 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 du document -->
<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 du document -->
```

Les caractères spéciaux `'<'`, `'>'` et `'&'` peuvent apparaître dans les commentaires. Il est en particulier possible de mettre en commentaire des éléments avec leurs balises comme dans l'exemple ci-dessous.

```
<!-- <tag type="comment">Élément mis en commentaire</tag> -->
```

2.7.6. Instructions de traitement

Les *instructions de traitement* sont destinées aux applications qui traitent les documents XML. Elles sont l'analogue des directives `#...` du langage C qui s'adressent au compilateur. Elles peuvent apparaître aux mêmes endroits que les commentaires à l'exception du contenu de la DTD.

Les instructions de traitement sont délimitées par les chaînes de caractères `'<?'` et `'?>'`. Les deux caractères `'<?'` sont immédiatement suivis du *nom XML* [Section 2.2.3] de l'instruction. Le nom de l'instruction est ensuite suivi du *contenu*. Ce contenu est une chaîne quelconque de caractères ne contenant pas la chaîne `'?>'` utilisée par l'analyseur lexical pour déterminer la fin de l'instruction. Le nom de l'instruction permet à l'application de déterminer si l'instruction lui est destinée.

Bien que le contenu d'une instruction puisse être quelconque, il est souvent organisé en une suite de paires *`param`*`="`*`value`*`"` avec une syntaxe imitant celle des attributs [Section 2.7.3]. Il incombe cependant à l'application traitant l'instruction de parser le contenu de celle-ci pour en extraire la liste des paires.

Les fichiers sources DocBook [http://www.docbook.org] de cet ouvrage contiennent des instructions de traitement de la forme suivante. Ces instructions indiquent le nom du fichier cible à utiliser par les feuilles de styles pour la conversion en HTML.

```
<?dbhtml filename="index.html"?>
```

Une feuille de style XSLT [Chapitre 8] peut être attachée à un document XML par l'intermédiaire d'une instruction de traitement de nom `xml-stylesheet` comme ci-dessous.

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

L'entête XML [Section 2.6.1] `<?xml version=... ?>` ressemble à une instruction de traitement de nom `xml` avec des paramètres `version`, `encoding` et `standalone`. Elle utilise en effet la même syntaxe. Elle n'est pourtant pas une instruction de traitement et elle ne fait pas partie du document.

2.8. Exemples minimaux

Quelques exemples minimalistes de documents XML sont donnés ci-dessous.

2.8.1. Exemple minimal

L'exemple suivant contient uniquement un prologue avec la l'entête XML et un élément de contenu vide. Les balises ouvrante `<tag>` et fermante `</tag>` ont été contractées en une seule balise `<tag/>` [Section 2.7.1]. Ce document n'a pas de déclaration de DTD.

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

L'exemple aurait pu encore être réduit en supprimant l'entête XML [Section 2.6.1] mais celle-ci est fortement conseillée. Le retour à la ligne après l'entête aurait aussi pu être supprimé sans changer le contenu du document.

2.8.2. Exemple simple avec une DTD

Cet exemple contient une déclaration de DTD qui permet de valider le document. Cette DTD déclare l'élément `simple` avec un contenu purement textuel.

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

2.9. XInclude

Il est possible de répartir un gros document en plusieurs fichiers afin d'en rendre la gestion plus aisée. Il existe essentiellement deux méthodes pour atteindre cet objectif. Le point commun de ces méthodes est de scinder le document en différents fichiers qui sont *inclus* par un fichier principal. Les deux méthodes se différencient par leurs façons de réaliser l'inclusion.

La méthode la plus ancienne est héritée de SGML et elle est basée sur les entités externes [Section 3.5.1.4]. La méthode, plus récente, basée sur XInclude [W] est à utiliser de préférences aux entités externes. XInclude définit un élément `xi:include` dans un espace de noms [Chapitre 4] associé à l'URL `http://www.w3.org/2001/XInclude`. Cet élément a un attribut `href` qui contient le nom du fichier à inclure et un attribut `parse` qui précise le type des données. Cet attribut peut prendre les valeurs `xml` ou `text`. Le fichier source principal de cet ouvrage inclut, par exemple, les fichiers contenant les différents chapitres grâce à des éléments `include` comme ci-dessous.

```
<book version="5.0"
      xmlns="http://docbook.org/ns/docbook"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  ...
  <!-- Inclusion des différents chapitres -->
  <xi:include href="introduction.xml"   parse="xml"/>
  <xi:include href="Syntax/chapter.xml" parse="xml"/>
  ...
</book>
```

Le fragment de document contenu dans un fichier inclus doit être bien formé. Il doit, en outre, être entièrement contenu dans un seul élément qui est l'élément racine du fragment.

Il faut prendre garde au fait que certaines applications ne gèrent pas XInclude. La solution est d'ajouter à la chaîne de traitement une étape consistant à construire un document global entièrement contenu dans un seul fichier. Le logiciel `xmllint` peut, par exemple, réaliser cette opération. Avec l'option `--xinclude`, il écrit sur la sortie standard un document où les éléments `xi:include` sont remplacés par le contenu des fichiers référencés. Cette option peut être combinée avec l'option `--noent` pour supprimer les entités [Section 3.5] définies dans la DTD.

L'opération consistant à remplacer un élément `xi:include` par le contenu du fichier doit mettre à jour l'attribut `xml:base` [Section 2.7.4.3] de l'élément racine du document dans le fichier. Cet attribut contient une URL qui

permet de résoudre les liens relatifs. Le chemin d'accès au fichier doit donc être ajouté à la valeur de l'attribut `xml:base`. Il faut, en particulier, ajouter cet attribut s'il est absent et si le chemin d'accès est non vide. Le chemin d'accès au fichier est récupéré dans l'attribut `href` de l'élément `xi:include`.

La mise à jour des attributs `xml:base` garde une trace des inclusions et permet aux liens relatifs de rester valides. La prise en compte des valeurs de ces attributs `xml:base` incombe en revanche aux applications qui traitent le document et utilisent ces liens.

Si chacun des fichiers `introduction.xml` et `Syntax/chapter.xml` a comme élément racine un élément `chapter` sans attribut `xml:base`, le résultat de l'inclusion de ces fichiers doit donner un document ressemblant à ceci.

```
<book version="5.0"
      xmlns="http://docbook.org/ns/docbook"
      xmlns:xi="http://www.w3.org/2001/XInclude">

  ...
  <!-- Inclusion des différents chapitres -->
  <chapter xml:id="chap.introduction" xml:base="introduction.xml">
    ...
  </chapter>
  <chapter xml:id="chap.syntax" xml:base="Syntax/chapter.xml">
    ...
  </chapter>
  ...
</book>
```

Chapitre 3. 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 [Chapitre 5] permettent de décrire de façon plus précise encore la structure d'un document. Ils sont plus sophistiqués mais plus difficiles à mettre en œuvre. Les DTD sont donc particulièrement adaptées pour des petits modèles de documents. En revanche, leur manque de modularité les rend plus difficiles à utiliser pour des modèles plus conséquents.

3.1. Un premier exemple

On reprend la petite bibliographie du fichier `bibliography.xml` déjà utilisée au chapitre précédent. La troisième ligne de ce fichier est la déclaration de la DTD qui référence un fichier externe `bibliography.dtd`. Le nom `bibliography` de l'élément racine du document apparaît dans cette déclaration juste après le mot clé `DOCTYPE`.

```
<!DOCTYPE bibliography SYSTEM "bibliography.dtd">
```

On présente maintenant le contenu de ce fichier `bibliography.dtd` qui contient la DTD du fichier `bibliography.xml`. La syntaxe des DTD est héritée de SGML et elle est différente du reste du document XML. Il n'y a pas de balises ouvrantes et fermantes. La DTD contient des déclarations d'éléments et d'attributs délimitées par les chaînes de caractères `'<!'` et `'>'`. Un mot clé juste après la chaîne `'<!'` indique le type de la déclaration. La syntaxe et la signification précise de ces déclarations sont explicitées dans ce chapitre.

```
<!ELEMENT bibliography (book)+>❶
<!ELEMENT book (title, author, year, publisher, isbn, url?)>❷
<!ATTLIST book key   NMTOKEN  #REQUIRED>❸
<!ATTLIST book lang (fr | en) #REQUIRED>❹
<!ELEMENT title     (#PCDATA)>❺
<!ELEMENT author    (#PCDATA)>
<!ELEMENT year      (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT isbn      (#PCDATA)>
<!ELEMENT url       (#PCDATA)>
```

❶ Déclaration de l'élément `bibliography` devant contenir une suite non vide d'éléments `book`.
❷ Déclaration de l'élément `book` devant contenir les éléments `title`, `author`, …, `isbn` et `url`.
❸❹ Déclarations des attributs obligatoires `key` et `lang` de l'élément `book`.
❺ Déclaration de l'élément `title` devant contenir uniquement du texte.

3.2. 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 vers 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. La déclaration de la DTD est introduite par le mot clé `DOCTYPE` et a la forme générale suivante où *`root-element`* est le nom de l'élément racine du document.

```
<!DOCTYPE root-element ... >
```

Le nom de l'élément racine est suivi du contenu de la DTD dans le cas d'une DTD interne ou de l'URL du fichier contenant la DTD dans le cas d'une DTD externe.

3.2.1. DTD interne

Lorsque la DTD est incluse dans le document, sa déclaration prend la forme suivante où son contenu est encadré par des caractères crochets `'['` et `']'`.

```
<!DOCTYPE root-element [ declarations ]>
```

Les déclarations *`declarations`* constituent la définition du type du document. Dans l'exemple suivant de DTD, 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.

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

3.2.2. DTD externe

Lorsque la DTD est externe, celle-ci est contenue dans un autre fichier dont l'extension est généralement `.dtd`. Le document XML se contente alors de donner l'adresse de sa DTD pour que les logiciels puissent y accéder. L'adresse de de la DTD peut être donnée explicitement par une URL ou par un FPI (*Formal Public Indentifier*). Les FPI sont des noms symboliques donnés aux documents. Ils sont utilisés avec des catalogues qui établissent les correspondances entre ces noms symboliques et les adresses réelles des documents. Lorsqu'un logiciel rencontre un FPI, il parcourt le catalogue pour le *résoudre*, c'est-à-dire déterminer l'adresse réelle du document. Les catalogues peuvent contenir des adresses locales et/ou des URL. Ils constituent donc une indirection qui facilite la maintenance. Lorsqu'un document, une DTD par exemple, est déplacé, il suffit de modifier les catalogues plutôt que tous les documents qui référencent le document.

3.2.2.1. Adressée par FPI

Les FPI (*Formal Public Identifier*) sont des identifiants de document hérités de SGML. Ils sont plutôt remplacés en XML par les URI [Section 2.3] qui jouent le même rôle. Ils sont constitués de quatre parties séparées par des chaînes `'//'` et ils obéissent donc à la syntaxe suivante.

```
type//owner//desc//lang
```

Le premier caractère *`type`* du FPI est soit le caractère `'+'` si le propriétaire est enregistré selon la norme ISO 9070 soit le caractère `'-'` sinon. Le FPI continue avec le propriétaire *`owner`* et la description *`desc`* du document. Cette description est formée d'un mot clé suivi d'un texte libre. Ce mot clé est `DTD` pour les DTD mais il peut aussi être `DOCUMENT`, `ELEMENTS` ou `ENTITIES`. Le FPI se termine par un code de langue *`lang`* de la norme ISO 639 [Section 2.7.4.1]. Lorsque le document appartient à une norme ISO, le premier caractère `'+'` ainsi que la chaîne `'//'` suivante sont supprimés. Des exemples de FPI sont donnés ci-dessous.

```
-//W3C//DTD XHTML 1.0 Strict//EN
-//OASIS//DTD Entity Resolution XML Catalog V1.0//EN
ISO/IEC 10179:1996//DTD DSSSL Architecture//EN
```

Un FPI peut être converti en URI en utilisant l'espace de noms `publicid` des URN et en remplaçant chaque chaîne `'//'` par le caractère `':'` et chaque espace par le caractère `'+'` comme dans l'exemple ci-dessous.

```
urn:publicid:-:W3C:DTD+XHTML+1.0+Strict:EN
```

La déclaration d'une DTD externe peut utiliser un FPI pour désigner la DTD. La référence à un FPI est introduite par le mot clé `PUBLIC` suivi du FPI et d'une URL délimitée par des apostrophes `'''` ou des guillemets `'"'`. L'URL est utilisée dans le cas où le FPI ne permet pas à l'application de retrouver la DTD.

```
<!DOCTYPE root-element PUBLIC "fpi" "url">
```

L'exemple suivant est la déclaration typique d'une page HTML qui utilise une des DTD de XHTML.

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

3.2.2.2. Adressée par URL

La référence à une URL est introduite par le mot clé `SYSTEM` suivi de l'URL délimitée par des apostrophes `'''` ou des guillemets `'"'`.

```
<!DOCTYPE root-element SYSTEM "url">
```

L'URL *`url`* peut être soit une URL complète commençant par `http://` ou `ftp://` soit plus simplement le nom d'un fichier local comme dans les exemples suivants.

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

<!DOCTYPE bibliography SYSTEM "bibliography.dtd">
```

3.2.3. DTD mixte

Il est possible d'avoir simultanément une DTD externe adressée par URL ou FPI et des déclarations internes. La DTD globale est alors formée des déclarations internes suivies des déclarations externes. La déclaration prend alors une des deux formes suivantes On retrouve un mélange de la syntaxe des DTD externes avec les mots clés `SYSTEM` et `PUBLIC` et de la syntaxe des DTD internes avec des déclarations encadrées par les caractères `'['` et `']'`.

```
<!DOCTYPE root-element SYSTEM "url" [ declarations ]>
<!DOCTYPE root-element PUBLIC "fpi" "url" [ declarations ]>
```

Il n'est pas possible de déclarer plusieurs fois le même élément dans une DTD. Lorsque la DTD est mixte, tout élément doit être déclaré dans la partie interne ou dans la partie externe mais pas dans les deux. En revanche, il est possible de déclarer plusieurs fois le même attribut. La première déclaration a priorité. Il est ainsi possible de donner une nouvelle déclaration d'un attribut dans la partie interne puisque celle-ci est placée avant la partie externe.

Les entités paramètres [Section 3.5.2] peuvent également être déclarées plusieurs fois dans une même DTD et c'est encore la première déclaration qui l'emporte. La DTD suivante déclare l'entité paramètre `book.entries` égale à la chaîne vide. Cette entité est ensuite utilisée dans la déclaration de l'élément `book`. Cette déclaration laisse la possibilité au document principal d'ajouter des enfants à l'élément `book` en donnant une nouvelle valeur à l'entité `book.entries`. La première ligne de cette DTD est une entête XML [Section 2.6.1] qui permet de déclarer le codage des caractères.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- DTD externe -->
<!-- Entité paramètre pour les enfants à ajouter à l'élément book -->
<!ENTITY % book.entries "">
<!ELEMENT bibliography (book)+>
<!ELEMENT book (title, author, year, publisher, isbn %book.entries;)>
<!ATTLIST book key NMTOKEN #REQUIRED>
<!ATTLIST book lang (fr | en) #IMPLIED>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT author    (#PCDATA)>
```

```
<!ELEMENT year      (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT isbn      (#PCDATA)>
```

Le document suivant a une DTD mixte dont la partie externe est la DTD précédente. La partie interne de la DTD contient la déclaration de l'élément `url`. Elle contient également des nouvelles déclarations de l'entité paramètre `book.entries` et de l'attribut `lang` de l'élément `book`, qui devient ainsi obligatoire.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE bibliography SYSTEM "mixed.dtd" [
  <!-- Ajout d'un enfant url à l'élément book -->
  <!ENTITY % book.entries ", url?">
  <!-- Redéclaration de l'attribut key de l'élément book -->
  <!-- L'attribut devient obligatoire -->
  <!ATTLIST book lang (fr | en) #REQUIRED>
  <!-- Déclaration de l'élément url -->
  <!ELEMENT url (#PCDATA)>
]>
<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="fr">
    <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>
```

Comme la valeur donnée à l'entité paramètre `book.entries` est la chaîne `", url?"` (sans les guillemets `'"'`), la déclaration de l'élément `book` dans le fichier `mixed.dtd` devient équivalente à la déclaration suivante qui autorise un enfant `url` optionnel.

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

3.3. Contenu de la DTD

Une DTD est constituée de déclarations d'éléments, d'attributs et d'entités. Elle peut aussi contenir des déclarations de notations mais celles-ci ne sont pas abordées dans cet ouvrage. Chacune de ces déclarations commence par la chaîne `'<!'` suivi d'un mot clé qui indique le type de déclaration. Les mots clés possibles sont `ELEMENT`, `ATTLIST` et `ENTITY`. La déclaration se termine par le caractère `'>'`.

3.4. Commentaires

Une DTD peut contenir des commentaires qui utilisent la syntaxe des commentaires XML [Section 2.7.5] délimités par les chaînes de caractères `'<!--'` et `'-->'`. Ceux-ci sont placés au même niveau que les déclarations d'éléments, d'attributs et d'entités. Ils ne peuvent pas apparaître à l'intérieur d'une déclaration.

```
 <!-- DTD pour les bibliographies -->
 <!ELEMENT bibliography (book)+>
```

```
<!-- Déclaration de l'élément book avec des enfants title, ..., isbn et url -->
<!ELEMENT book (title, author, year, publisher, isbn, url?)>
...
```

3.5. Entités

Les *entités* constituent un mécanisme hérité de SGML. Elles sont des *macros* semblables aux `#define` du langage C. Elles permettent également de réaliser des inclusions de documents comme la directive `#include` du langage C. Les entités sont définies dans la DTD du document. Il existe deux types d'entités. Les *entités générales*, appelées simplement entités dans cet ouvrage, sont destinées à être utilisées dans le corps du document. Les *entités paramètres* sont destinées à être utilisées au sein de la DTD.

Une entité est, en quelque sorte, un nom donné à un fragment de document. Ce fragment peut être donné explicitement à la définition de l'entité dans la DTD. Il peut également provenir d'un fichier externe. Dans ce cas, la définition de l'entité donne un FPI et/ou une URL permettant d'accéder au document. Le fragment de document peut être inséré dans le document en utilisant simplement le nom de l'entité. Lorsque le fragment provient d'un autre fichier, l'utilisation de l'entité provoque l'inclusion du fichier en question. Le nom de chaque entité générale ou paramètre doit être un nom XML [Section 2.2.3].

3.5.1. Entités générales

Les entités générales sont les entités les plus courantes et les plus utiles puisqu'elles peuvent être utilisées dans le corps du document.

3.5.1.1. Déclaration et référence

La déclaration d'une entité commence par `<!ENTITY` suivi du nom de l'entité. Elle prend une des trois formes suivantes où *`name`* est le nom de l'entité, *`fragment`* est un fragment de document, *`fpi`* et *`url`* sont un FPI et une URL.

```
<!ENTITY name "fragment">
<!ENTITY name SYSTEM "url">
<!ENTITY name PUBLIC "fpi" "url">
```

Lorsque l'entité est déclarée avec la première syntaxe, elle est dite *interne* car le fragment est explicitement donné dans la DTD du document. Lorsqu'elle est déclarée avec une des deux dernières syntaxes, elle est dite *externe* car le fragment provient d'un autre document. Ces deux dernières syntaxes sont semblables à la déclaration d'une DTD externe [Section 3.2.2] dans un document XML. Les règles pour l'utilisation des apostrophes `'''` et des guillemets `'"'` sont identiques à celles pour les valeurs d'attributs [Section 2.7.3]. Le fragment, le FPI et l'URL doivent être délimités par une paire d'apostrophes ou de guillemets. Si le fragment est délimité par des apostrophes, les guillemets peuvent être introduits directement sans entité et inversement.

Une entité de nom *`name`* est référencée, c'est-à-dire utilisée, par *`&name;`* où le nom *`name`* de l'entité est encadré par les caractères `'&'` et `';'`. Lorsque le document est traité, la référence à une entité est remplacée par le fragment de document correspondant. Une entité interne peut être référencée dans les contenus d'éléments et dans les valeurs d'attribut alors qu'une entité externe peut seulement être référencée dans les contenus d'éléments.

```
<tag meta="attribute: &name;">Content: &name;</tag>
```

3.5.1.2. Entités prédéfinies

Il existe des entités prédéfinies permettant d'inclure les caractères spéciaux [Section 2.2.1] `'<'`, `'>'`, `'&'`, `'''` et `'"'` dans les contenus d'éléments et dans les valeurs d'attributs. Ces entités sont les suivantes.

Entité	Caractère
<	<

Entité	Caractère
>	>
&	&
'	'
"	"

Tableau 3.1. Entités prédéfinies

Les trois entités `<`, `>` et `&` doivent être utilisées aussi bien dans le contenus des éléments que dans les valeurs des attributs puisque les caractères `'<'`, `'>'` et `'&'` ne peuvent pas y apparaître directement. Les deux entités `&apos;` et `&quot;` sont uniquement nécessaires pour inclure le caractère `'''` ou le caractère `'"'` dans la valeur d'un attribut délimitée par le même caractère. Lors de l'utilisation XPath [Chapitre 6] avec XSLT [Chapitre 8], il n'est pas rare d'inclure ces deux caractères dans la valeur d'un attribut. Une des deux entités `&apos;` ou `&quot;` devient alors indispensable. L'exemple suivant est extrait d'une feuille de style XSLT. La valeur de l'attribut `test` est une expression XPath qui teste si la valeur de la variable `string` contient le caractère `'"'`.

```
    <xsl:when test="contains($string, '&quot;')">
```

Les nombreuses entités prédéfinies en XHTML comme `&euro;` pour le symbole '€' n'existent pas en XML. La seule façon d'inclure ce caractère est d'utiliser les notations `&#`*`point de code décimal`*`;` ou `&#x`*`point de code hexadécimal`*`;`. Il est cependant possible de définir ces propres entités (cf. ci-dessous).

3.5.1.3. Entités internes

La valeur d'une entité interne est le fragment de document associé à celle-ci lors de sa déclaration. Cette valeur peut contenir des caractères ainsi que des éléments avec leurs balises. Lorsqu'elle contient des éléments, le fragment doit être bien formé [Section 2.7.1]. À toute balise ouvrante doit correspondre une balise fermante et l'imbrication des balises doit être correcte. Quelques exemples d'entités internes sons donnés ci-dessous. Le dernier exemple utilise des apostrophes `'''` pour délimiter le fragment de document qui contient des guillemets `'"'` pour encadrer les valeurs des attributs.

```
<!ENTITY aka   "also known as">
<!ENTITY euro  "&#x20AC;">
<!ENTITY rceil '<phrase condition="html">&#x2309;</phrase>
                <phrase condition="fo" role="symbolfont">&#xF8F9;</phrase>'>
```

Si la DTD contient les déclarations d'entités ci-dessus, Il est possible d'inclure le texte `also known as` en écrivant seulement `&aka;` ou d'inclure un symbole en écrivant `&euro;`. Les entités internes peuvent être référencées dans les contenus d'éléments mais aussi dans les valeurs d'attributs.

```
<price currency="Euro (&euro;)">30 &euro;</price>
```

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 l'analyseur de l'application. Les définitions récursives sont bien sûr interdites.

```
<!DOCTYPE book [
  <!-- Entités internes -->
  <!ENTITY mci "Michel Colucci &aka; 'Coluche'">
  <!ENTITY aka "also known as">
]>
<book>&mci;</book>
```

Il faut faire attention au fait que certaines applications ne gèrent pas ou gèrent mal les entités définies. La solution est d'ajouter à la chaîne de traitement une première étape consistant à substituer les entités par leurs valeurs pour obtenir un document intermédiaire sans entités. Le logiciel `xmllint` peut par exemple réaliser cette opération. Avec l'option `--noent`, il écrit sur la sortie standard le document en remplaçant chaque entité par sa valeur.

3.5.1.4. Entités externes

Une entité peut désigner un fragment de document contenu dans un autre fichier. Ce mécanisme permet de répartir un même document sur plusieurs fichiers comme dans l'exemple suivant. La déclaration utilise alors le mot clé `SYSTEM` suivi d'une URL qui peut, simplement, être le nom d'un fichier local.

Les entités externes peuvent êtres utilisées pour scinder un document en plusieurs fichiers. Le fichier principal inclut les différentes parties en définissant une entité externe pour chacune de ces parties. Les entités sont alors utilisées pour réaliser l'inclusion comme dans l'exemple ci-dessous.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE book [
  <!-- Entités externes -->
  <!ENTITY chapter1 SYSTEM "chapter1.xml">
  <!ENTITY chapter2 SYSTEM "chapter2.xml">
]>
<book>
  <!-- Inclusion du fichier chapter1.xml -->
  &chapter1;
  <!-- Inclusion du fichier chapter2.xml -->
  &chapter2;
</book>
```

Chacun des fichiers contenant une entité externe peut avoir une entête [Section 2.6.1]. Celle-ci permet par exemple de déclarer un encodage des caractères différents du fichier principal. Ce mécanisme pour répartir un document en plusieurs fichiers est à abandonner au profit de XInclude [Section 2.9] qui est plus pratique.

3.5.2. Entités paramètres

Les *entités paramètres* sont des entités qui peuvent uniquement être utilisées à l'intérieur de la DTD. La terminologie est historique et provient de SGML. Ces entités ont le même rôle que les entités générales. Elles sont surtout utilisées pour apporter de la modularité aux DTD. La déclaration d'une entité paramètre prend une des trois formes suivantes où *`name`* est le nom de l'entité, *`fragment`* est un fragment de document, *`fpi`* et *`url`* sont un FPI et une URL.

```
<!ENTITY % name "fragment">
<!ENTITY % name SYSTEM "url">
<!ENTITY % name PUBLIC "fpi" "url">
```

La seule différence avec la déclaration d'une entité générale est la présence du caractère `'%'` entre le mot clé `ENTITY` et le nom de l'entité déclarée. Comme pour les entités générales, l'entité est dite *interne* lorsqu'elle est déclarée avec la première syntaxe. Elle est dite *externe* lorsqu'elle est déclarée avec une des deux dernières syntaxes. Les règles pour l'utilisation des apostrophes `'''` et des guillemets `'"'` sont identiques à celles pour les valeurs d'attributs [Section 2.7.3] ou les entités générales [Section 3.5.1].

L'entité *`name`* ainsi déclarée peut être référencée, c'est-à-dire utilisée, par *`%name;`* où le nom de l'entité est encadré les caractères `'%'` et `';'`. Les entités paramètres peuvent uniquement être utilisée au sein de la DTD. Lorsque l'entité est interne, elle peut être utilisée dans les déclarations d'éléments, d'attributs et d'autres entités. Cette utilisation est limitée à la partie externe de la DTD. Lorsque l'entité est externe, elle est utilisée en dehors des déclarations pour inclure des déclarations provenant du document référencé par l'entité. L'exemple suivant définit deux entités paramètres `idatt` et `langatt` permettant de déclarer des attributs `id` et `xml:lang` facilement.

```
<!-- Déclaration de deux entités paramètres -->
<!ENTITY % idatt   "id ID #REQUIRED">
<!ENTITY % langatt "xml:lang NMTOKEN 'fr'">

<!-- Utilisation des deux entités paramètres -->
<!ATTLIST chapter %idatt; %langatt;>
```

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

Les entités paramètres ajoutent de la modularité qui est surtout nécessaire dans l'écriture de DTD de grande taille. Dans l'exemple précédent, l'attribut `id` pourrait être remplacé partout par un attribut `xml:id` en changeant uniquement la définition de l'entité paramètre `idatt`. Un autre exemple d'utilisation des entités paramètres est donné avec les DTD mixtes [Section 3.2.3]. Les entités externes permettent d'inclure une partie de DTD provenant d'un document externe comme dans l'exemple suivant.

```
<!-- Entité paramètre pour inclure la DTD principale -->
<!ENTITY % main SYSTEM "main.dtd">
<!-- Inclusion du fichier main.dtd -->
%main;
```

3.6. Déclaration d'élément

Les déclarations d'éléments constituent le cœur des DTD car elles définissent la structure des documents valides. Elles spécifient quels doivent être les enfants de chaque élément et l'ordre de ces enfants.

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 nom de l'élément doit être un nom XML [Section 2.2.3] et 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 *`element`* et *`type`* sont respectivement le nom et le type de l'élément. Le type de l'élément détermine quels sont ses contenus autorisés.

```
<!ELEMENT element type>
```

3.6.1. Contenu pur d'éléments

Le contenu d'un élément est *pur* lorsqu'il ne contient aucun texte et qu'il est uniquement constitué d'éléments qui sont ses enfants. Ces enfants peuvent, à leur tour, avoir des contenus textuels, purs ou mixtes. Un élément de contenu pur peut donc indirectement contenir du texte si celui-ci fait partie du contenu d'un de ses descendants. La déclaration d'un élément de contenu pur détermine quels peuvent être ses enfants et dans quel ordre ils doivent apparaître. Cette description est indépendante du contenu de ces enfants. Elle dépend uniquement des noms des enfants. Le contenu de chaque enfant est décrit par sa propre déclaration. Une déclaration d'élément prend la forme suivante.

```
<!ELEMENT element regexp>
```

Le nom de l'élément est donné par l'identifiant *`element`*. L'expression rationnelle *`regexp`* décrit les suites autorisées d'enfants dans le contenu de l'élément. Cette expression rationnelle est construite à partir des noms d'éléments en utilisant les opérateurs `','`, `'|'`, `'?'`, `'*'` et `'+'` ainsi que les parenthèses `'('` et `')'` pour former des groupes. Les opérateurs `','` et `'|'` sont binaires alors que les opérateurs `'?'`, `'*'` et `'+'` sont unaires et postfixés. Ils se placent juste après leur opérande, c'est-à-dire derrière le groupe auquel ils s'appliquent.

Le nom d'un élément signifie que cet élément doit apparaître dans le contenu. Les opérateurs principaux sont les deux opérateurs `','` et `'|'` qui expriment la mise en séquence et le choix. Un contenu est valide pour une expression de la forme *`block-1, block-2`* s'il est formé d'un contenu valide pour *`block-1`* suivi d'un contenu valide pour *`block-2`*. Un contenu est valide pour une expression de la forme *`block-1 | block-2`* s'il est formé d'un contenu valide pour *`block-1`* ou d'un contenu valide pour *`block-2`*. Les trois opérateurs `'?'`, `'*'` et `'+'` permettent d'exprimer des répétitions. Un contenu est valide pour une expression de la forme *`block`*`?` s'il est vide ou formé d'un contenu valide pour *`block`*. Un contenu est valide pour une expression de la forme *`block`*`*` (respectivement *`block`*`+`) s'il est formé d'une suite éventuellement vide (respectivement suite non vide) de contenus valides pour *`block`*. La signification des cinq opérateurs est récapitulée par la table suivante.

Opérateur	**Signification**
`,`	Mise en séquence

Opérateur	Signification
|	Choix
?	0 ou 1 occurrence
*	Répétition d'un nombre quelconque d'occurrences
+	Répétition d'un nombre non nul d'occurrences

Tableau 3.2. Opérateurs des DTD

Ces définitions sont illustrées 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.

`<!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`.

3.6.1.1. Caractères d'espacement

La prise en compte des caractères d'espacement [Section 2.2.2] lors de validation d'un élément de contenu pur dépend du caractère interne ou externe de la DTD et de la valeur de l'attribut `standalone` de l'entête [Section 2.6.1]. Bien que le contenu d'un élément soit pur, il est, néanmoins, possible d'insérer des caractères d'espacement entre ses enfants. La validation ignore ces caractères lorsque la DTD est interne ou lorsque la valeur de l'attribut `standalone` est `no`. Ce traitement particulier rend possible l'indentation du document. Si la DTD est externe et que la valeur de l'attribut `standalone` est `yes`, la validation devient stricte et elle n'autorise aucun caractère d'espacement dans un contenu pur.

La DTD suivante déclare un élément `list` de contenu pur. Il contient une suite non vide d'éléments `item` contenant chacun du texte.

```
<!-- Fichier "standalone.dtd" -->
<!ELEMENT list (item)+>
<!ELEMENT item (#PCDATA)>
```

Le document suivant est valide pour la DTD précédente car la valeur de l'attribut `standalone` de l'entête est `no`. Si cette valeur était `yes`, le document ne serait plus valide car l'élément `list` contient des caractères d'espacement entre ses enfants `item`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE list SYSTEM "standalone.dtd">
<list>
  <item>Item 1</item>
  <item>Item 2</item>
</list>
```

3.6.1.2. Contenu déterministe

Afin de simplifier la validation de documents, les expressions rationnelles qui décrivent les contenus purs des éléments doivent être déterministes. Cette contrainte signifie qu'en cours d'analyse du contenu, le nom d'un élément ne peut apparaître que sur une seule branche de l'expression. Un exemple typique d'expression ne respectant pas cette règle est la suivante.

```
<!ELEMENT item ((item1, item2) | (item1, item3))>
```

Lors de l'analyse du contenu de l'élément `item`, le validateur ne peut pas déterminer en lisant l'élément `item1` si celui-ci provient de la première alternative `(item1, item2)` ou de la seconde alternative `(item1, item3)`. L'expression précédente peut, cependant, être remplacée par l'expression équivalente suivante qui a l'avantage d'être déterministe.

```
<!ELEMENT item (item1, (item2 | item3))>
```

Le non-déterminisme d'une expression peut aussi provenir d'un des deux opérateurs `'?'` ou `'*'`. L'expression `(item1?, item1)` n'est pas déterministe car il n'est pas possible de déterminer immédiatement si un élément `item1` correspond à la première ou à la seconde occurrence de `item1` dans l'expression. Cette expression est équivalente à l'expression `(item1, item1?)` qui est déterministe. Il existe toutefois des expressions sans expression déterministe équivalente comme l'expression suivante.

```
<!ELEMENT item ((item1, item2)*, item1)>
```

3.6.2. 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 [Section 2.7.2].

```
<!ELEMENT element (#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)>
]>
<texts>
  <text>Du texte simple</text>
  <text>Une <![CDATA[ Section CDATA avec < et > ]]></text>
  <text>Des entités &lt; et &gt;</text>
</texts>
```

3.6.3. Contenu mixte

La déclaration de la forme suivante indique qu'un élément peut uniquement contenir du texte et les éléments `element1`, …, `elementN`. 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, le mot clé `#PCDATA` doit apparaître en premier avant tous les noms des éléments.

```
<!ELEMENT element (#PCDATA | element1 | ... | elementN)*>
```

Dans l'exemple suivant, l'élément `book` possède un contenu mixte. Il peut contenir du texte et des éléments `em` et `cite` en nombre quelconque et dans n'importe quel ordre.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE book [
  <!ELEMENT book (#PCDATA | em | cite)*>
  <!ELEMENT em   (#PCDATA)>
  <!ELEMENT cite (#PCDATA)>
]>
<book>
  Du <em>texte</em>, une <cite>citation</cite> et encore du <em>texte</em>.
</book>
```

3.6.4. Contenu vide

La déclaration suivante indique que le contenu de l'élément `element` est nécessairement vide. Cet élément peut uniquement avoir des attributs. Les éléments vides sont souvent utilisés pour des liens entre éléments.

```
<!ELEMENT element EMPTY>
```

Des exemples d'utilisation d'éléments de contenu vide sont donnés à la section traitant des attributs de type `ID`, `IDREF` et `IDREFS` [Section 3.7.2].

```
<!ELEMENT ref EMPTY>
<!ATTLIST ref idref IDREF #REQUIRED>
```

Dans un souci de portabilité, il est conseillé de contracter les balises ouvrante et fermante lorsqu'un élément est déclaré de contenu vide et de le faire uniquement dans ce cas.

3.6.5. Contenu libre

La déclaration suivante n'impose aucune contrainte sur le contenu de l'élément *`element`*. En revanche, 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 afin de procéder à des essais de validation. En revanche, ces déclarations sont déconseillées dans une DTD terminée car elles conduisent à des modèles de document trop laxistes.

```
<!ELEMENT element ANY>
```

3.7. Déclaration d'attribut

Pour qu'un document soit valide, tout attribut présent dans la balise ouvrante d'un élément doit être déclaré. La déclaration d'attribut prend la forme générale suivante où *`attribut`* est le nom de l'attribut et *`element`* le nom de l'élément auquel il appartient. Cette déclaration comprend également le type *`type`* et la valeur par défaut *`default`* de l'attribut. Le nom de l'attribut doit être un nom XML [Section 2.2.3].

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

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 element attribut1 type1 default1
                  attribut2 type2 default2
                  ...
                  attributN typeN defaultN>
```

Les différents types possibles pour un attribut ainsi que les valeurs par défaut autorisées sont détaillés dans les sections suivantes.

3.7.1. Types des attributs

Le type d'un attribut détermine quelles sont ses valeurs possibles. Les DTD proposent uniquement un choix fini de types pour les attributs. Le type doit en effet être pris dans la liste suivante. Les types les plus utilisés sont `CDATA`, `ID` et `IDREF`.

`CDATA`
: Ce type est le plus général. Il n'impose aucune contrainte à la valeur de l'attribut. Celle-ci peut être une chaîne quelconque de caractères.

(*value-1* | *value-2* | ... | *value-N*)

: La valeur de l'attribut doit être un des jetons [Section 2.2.3] *value-1*, *value-2*, ... *value-N*. Comme ces valeurs sont des jetons, celles-ci ne sont pas délimitées par des apostrophes `'''` ou des guillemets `'"'`.

`NMTOKEN`
: La valeur de l'attribut est un jeton [Section 2.2.3].

`NMTOKENS`
: La valeur de l'attribut est une liste de jetons séparés par des espaces.

`ID`
: La valeur de l'attribut est un nom XML [Section 2.2.3]. 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 est une liste de références séparées 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

Le type le plus général est `CDATA` puisque toutes les valeurs correctes d'un point de vue syntaxique sont permises. Cet type est très souvent utilisé car il est approprié dès qu'il n'y a aucune contrainte sur la valeur de l'attribut.

Les types `NMTOKEN` et `NMTOKENS` imposent respectivement que la valeur de l'attribut soit un jeton [Section 2.2.3] ou une suite de jetons séparés par des espaces. Il est aussi possible d'imposer que la valeur de l'attribut soit dans une liste fixe de jetons. Il est impossible, avec une DTD, de restreindre les valeurs d'un attribut à une liste fixe de valeurs qui ne sont pas des jetons.

L'utilisation des trois types `NOTATION`, `ENTITY` et `ENTITIES` est réservée à l'usage des entités externes non XML et elle n'est pas détaillée dans cet ouvrage. L'utilisation des trois types `ID`, `IDREF` et `IDREFS` est développée à la section suivante.

3.7.2. Attributs de type `ID`, `IDREF` et `IDREFS`

Il est fréquent qu'il existe des liens entre les données d'un document XML. Il peut s'agir, par exemple, de références à d'autres parties du document. Les attributs de types `ID`, `IDREF` et `IDREFS` s'utilisent conjointement pour matérialiser ces liens au sein d'un document. Un attribut de type `ID` permet d'identifier de façon unique un élément du document. Les éléments ainsi identifiés peuvent alors être référencés par d'autres éléments grâce aux attributs de types `IDREF` et `IDREFS`. Ces attributs créent ainsi des liens entre des éléments ayant les attributs de types `IDREF` ou `IDREFS` et des éléments ayant les attributs de type `ID`. Ce mécanisme permet uniquement de créer des liens

entre des éléments d'un même document. La norme XLink généralise ce mécanisme. Elle permet de créer des liens entre deux, voire même plus de fragments de documents XML provenant éventuellement de documents différents.

La valeur d'un attribut de type `ID` doit être un nom XML [Section 2.2.3]. La valeur de cet attribut doit être unique dans tout le document. Ceci signifie qu'un autre attribut de type `ID` d'un autre élément ne peut pas avoir la même valeur pour que le document soit valide. Un élément ne peut avoir qu'un seul attribut de type `ID`.

Les attributs `id` de type `ID` sont utilisés par la fonction XPath `id()`.

La valeur d'un attribut de type `IDREF` doit être un nom XML. Ce nom doit, en outre, être la valeur d'un attribut de type `ID` d'un (autre) élément pour que le document soit valide. La valeur d'un attribut de type `IDREFS` doit être une suite de noms séparés par des espaces. Chacun de ces noms doit, en outre, être la valeur d'un attribut de type `ID` d'un élément pour que le document soit valide.

Le document suivant illustre l'utilisation des attributs de type `ID`, `IDREF` et `IDREFS` qui est faite par DocBook pour les références internes. Son contenu est scindé en sections délimitées par les éléments `section`. Chacun de ces éléments a un attribut `id` de type `ID`. Le contenu des éléments `section` est constitué de texte et d'éléments `ref` et `refs` ayant respectivement un attribut `idref` de type `IDREF` et un attribut `idrefs` de type `IDREFS`. Ces éléments permettent, dans le contenu d'une section, de référencer une (par `ref`) ou plusieurs (par `refs`) autres sections. Il faut remarquer que les éléments `ref` et `refs` n'ont jamais de contenu. Ils sont déclarés vides en utilisant le mot clé `EMPTY`. Il appartient à l'application qui génère le document final d'ajouter du contenu qui peut être par exemple le numéro ou le titre de la section référencée.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!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">Une référence <ref idref="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>
```

Les attributs de type `ID` et `IDREF` permettent également de structurer un document. Si l'adresse et d'autres informations sont ajoutées à l'éditeur dans le document `bibliography.xml`, celles-ci sont recopiées dans chaque livre publié par l'éditeur. Cette duplication de l'information est bien sûr très mauvaise. Une meilleure approche consiste à scinder la bibliographie en deux parties. Une première partie contient les livres et une seconde partie les éditeurs avec les informations associées. Ensuite, chaque livre se contente d'avoir une référence sur son éditeur. Un attribut `id` de type `ID` est ajouté à chaque élément `publisher` de la seconde partie. Chaque élément `publisher` contenu dans un élément `book` est remplacé par un élément `published` ayant un attribut `by` de type `IDREF`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<bibliography>
  <books>
    <book key="Michard01" lang="fr">
      <title>XML langage et appplications</title>
      <author>Alain Michard</author>
      <year>2001</year>
      <isbn>2-212-09206-7</isbn>
      <url>http://www.editions-eyrolles/livres/michard/</url>
      <published by="id2680397"/>
```

```
    </book>
    <book key="Marchal00" lang="en">
      <title>XML by Example</title>
      <author>Benoît Marchal</author>
      <year>2000</year>
      <isbn>0-7897-2242-9</isbn>
      <published by="id2680427"/>
    </book>
    ...
  </books>
  <publishers>
    <publisher id="id2680397">
      <name>Eyrolles</name>
      <address>Paris</address>
    </publisher>
    <publisher id="id2680427">
      <name>Macmillan Computer Publishing</name>
      <address>New York</address>
    </publisher>
    ...
  </publishers>
</bibliography>
```

Beaucoup d'applications ne prennent pas en compte la DTD pour traiter un document. Il leur est alors impossible de savoir quels attributs sont de type `ID`, `IDREF` ou `IDREFS`. Elles utilisent souvent la convention qu'un attribut de nom `id` est implicitement de type `ID`. Une meilleure solution consiste à utiliser l'attribut `xml:id` [Section 2.7.4.4] qui est toujours de type `ID` (de type `xml:ID` [Section 5.5.1.4] en fait).

3.7.3. Valeur par défaut

Chaque déclaration d'attribut précise une valeur par défaut pour celui-ci. Cette valeur par défaut peut prendre une des quatre formes suivantes.

`"value"` ou `'value'`
: La valeur *`value`* est une chaîne quelconque de caractères délimitée par des apostrophes `'''` ou des guillemets `'"'`. Si l'attribut est absent pour un élément du document, sa valeur est implicitement la chaîne *`value`*. Cette valeur doit, bien sûr, être du type donné à l'attribut.

`#IMPLIED`
: L'attribut est *optionnel* et il n'a pas de valeur par défaut. Si l'attribut est absent, il n'a pas de valeur.

`#REQUIRED`
: L'attribut est *obligatoire* et il n'a pas de valeur par défaut.

`#FIXED "value"` ou `#FIXED 'value'`
: La valeur *`value`* est une chaîne quelconque de caractères délimitée par des apostrophes `'''` ou des guillemets `'"'`. La valeur de l'attribut est fixée à la valeur *`value`* donnée. Si l'attribut est absent, sa valeur est implicitement *`value`*. Si l'attribut est présent, sa valeur doit être *`value`* pour que le document soit valide. Cette valeur doit, bien sûr, être du type donné à l'attribut.

Beaucoup d'applications ne prennent pas en compte la DTD pour traiter un document XML. Ce comportement pose problème avec les valeurs par défaut et les valeurs fixes (utilisation de `#FIXED`) des attributs. Si l'attribut est absent pour un élément du document, l'application va considérer que l'attribut n'a pas de valeur bien que la DTD déclare une valeur par défaut. L'utilisation des valeurs par défaut est donc à éviter pour une portabilité maximale.

3.7.4. Exemples

Voici quelques exemples de déclarations d'attributs avec, pour chacune d'entre elles, des valeurs valides et non valides pour l'attribut.

`<!ATTLIST tag meta CDATA "default">`

La valeur de l'attribut `meta` peut être une chaîne quelconque et sa valeur par défaut est la chaîne `default` Les exemples illustrent l'utilisation des entités prédéfinies [Section 3.5.1.2] pour inclure des caractères spéciaux dans les valeurs d'attribut.

Attribut dans le document	**Valeur**
`<tag meta="Hello World!">`	`Hello World!`
`<tag>`	Valeur par défaut `default`
`<tag meta="">`	Chaîne vide
`<tag meta="=='&quot;=="">`	`=='"==`
`<tag meta='==&apos;"=='>`	`=='"==`
`<tag meta="==&lt;&amp;&gt;==">`	`==<&>==`
`<tag meta="==&#60;&#38;&#62==">`	`==<&>==`

`<!ATTLIST book lang (fr | en) "fr">`

La valeur de l'attribut `lang` peut être soit le jeton `fr` soit le jeton `en` et sa valeur par défaut est le jeton `fr`.

Attribut dans le document	**Valeur**
`<book>`	Valeur par défaut `fr`
`<book lang="fr">`	Valeur par défaut `fr`
`<book lang="en">`	Jeton `en`
`<book lang="de">`	*non valide* car la valeur `de` n'est pas énumérée

`<!ATTLIST book name NMTOKEN #IMPLIED>`

L'attribut `name` est optionnel et sa valeur doit être un jeton. Il n'a pas de valeur par défaut.

Attribut dans le document	**Valeur**
`<book>`	Pas de valeur
`<book name="en">`	Jeton `en`
`<book name="-id234"/>`	Jeton `-id234`
`<book name="Hello World!">`	*non valide* car `Hello World!` n'est pas un jeton

`<!ATTLIST entry id ID #REQUIRED>`

L'attribut `id` est obligatoire et sa valeur doit être un nom unique. Il n'a pas de valeur par défaut.

Attribut dans le document	**Valeur**
`<entry>`	*non valide* attribut absent
`<entry id="id-234">`	Nom `id-234`
`<entry id="Hello World!">`	*non valide* car `Hello World!` n'est pas un nom

`<!ATTLIST xref ref IDREF #REQUIRED>`

L'attribut `ref` est obligatoire et sa valeur doit être un nom XML. Pour que le document soit valide, ce nom doit être la valeur d'un attribut de type `ID` d'un (autre) élément. Cet attribut n'a pas de valeur par défaut.

Attribut dans le document	**Valeur**
`<xref ref="id-234"/>`	`id-234`
`<xref ref="-id234"/>`	*non valide* car `-id234` n'est pas nom.

`<!ATTLIST xrefs refs IDREFS #REQUIRED>`

L'attribut `refs` est obligatoire et sa valeur doit être une suite de noms XML séparés par des espaces. Pour que le document soit valide, chacun des noms de la liste doit être la valeur d'un attribut de type `ID` d'un (autre) élément. Il n'a pas de valeur par défaut.

Attribut dans le document	Valeur
`<xrefs refs="id-234"/>`	Nom `id-234`
`<xrefs refs="id-234 id-437"/>`	Noms `id-234` et `id-437`

`<!ATTLIST rule style CDATA #FIXED "free">`
La valeur de l'attribut `style` de l'élément `rule` est fixée à la valeur `free`.

Attribut dans le document	Valeur
`<rule>`	Valeur fixée `free`
`<rule style="free">`	Valeur fixée `free`
`<rule style="libre">`	*non valide* car valeur différente

3.8. Outils de validation

Il existe plusieurs outils permettant de valider un document XML par rapport à une DTD. Il existe d'abord des sites WEB dont l'avantage est une interface conviviale. L'utilisateur peut fournir son document par un simple copier/coller ou en le téléchargeant. Comme la validation est réalisée sur une machine distante, la DTD doit être, soit accessible via une URL, soit directement incluse dans document.

- Page de validation du W3C [http://validator.w3.org/]
- Page de validation du Scholarly Technology Group de l'université de Brown [http://www.stg.brown.edu/service/xmlvalid/]

L'inconvénient majeur de ces sites WEB est la difficulté de les intégrer à une chaîne de traitement automatique puisqu'ils requièrent l'intervention de l'utilisateur. Dans ce cas, il est préférable d'utiliser un logiciel comme `xmllint`. Avec l'option `--valid`, il réalise la validation du document passé en paramètre. Avec cette option, la DTD soit être précisée par une déclaration de DTD [Section 2.6.2] dans le prologue du document. Sinon, il faut donner explicitement la DTD après l'option `--dtdvalid`. La DTD peut être donnée par le nom d'un fichier local ou par une URL qui permet à `xmllint` d'accéder à celle-ci.

Chapitre 4. Espaces de noms

4.1. Introduction

Les *espaces de noms* ont été introduits en XML afin de pouvoir mélanger plusieurs vocabulaires au sein d'un même document. De nombreux dialectes XML ont été définis pour des utilisations diverses et il est préférable de les réutiliser au maximum. Il est, en effet, fastidieux de redéfinir plusieurs fois les mêmes vocabulaires. Le recyclage des dialectes fait d'ailleurs partie des objectifs de XML.

Le mélange de plusieurs vocabulaires au sein d'un même document ne doit pas empêcher la validation de celui-ci. Il devient indispensable d'identifier la provenance de chaque élément et de chaque attribut afin de le valider correctement. Les espaces de noms jouent justement ce rôle. Chaque élément ou attribut appartient à un espace de noms qui détermine le vocabulaire dont il est issu. Cette appartenance est marquée par la présence dans le nom d'un préfixe associé à l'espace de noms.

Le mélange de plusieurs vocabulaires est illustré par l'exemple suivant. Afin d'insérer des métadonnées dans des documents, il est nécessaire de disposer d'éléments pour présenter celles-ci. Il existe déjà un standard, appelé Dublin Core, pour organiser ces métadonnées. Il comprend une quinzaine d'éléments dont `title`, `creator`, `subject` et `date` qui permettent de décrire les caractéristiques principales d'un document. Il est préférable d'utiliser le vocabulaire Dublin Core, qui est un standard international, plutôt que d'introduire un nouveau vocabulaire. Le document suivant est le document principal d'un livre au format DocBook. Les métadonnées sont contenues dans un élément `metadata`. Celui-ci contient plusieurs éléments du Dublin Core dont les noms commencent par le préfixe `dc`. L'élément `include` de XInclude fait partie d'un autre espace de noms marqué par le préfixe `xi`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book version="5.0" xml:lang="fr"
      xmlns="http://docbook.org/ns/docbook"
      xmlns:dc="http://purl.org/dc/elements/1.1/"
      xmlns:xi="http://www.w3.org/2001/XInclude">

 <!-- Titre DocBook du document -->
 <title>Langages formels, calculabilité et complexité</title>

 <!-- Métadonnées -->
 <metadata>
   <dc:title>Langages formels, calculabilité et complexité</dc:title>
   <dc:creator>Olivier Carton</dc:creator>
   <dc:date>2008-10-01</dc:date>
   <dc:identifier>urn:isbn:978-2-7117-2077-4</dc:identifier>
 </metadata>

 <!-- Import des chapitres avec XInclude -->
 <xi:include href="introduction.xml"  parse="xml"/>
 <xi:include href="chapter1.xml"      parse="xml"/>
 <xi:include href="chapter2.xml"      parse="xml"/>
 <xi:include href="chapter3.xml"      parse="xml"/>
 <xi:include href="chapter4.xml"      parse="xml"/>
 <index/>
</book>
```

Comme en C++, les espaces de noms évitent les conflits de noms entre différents vocabulaires. Le dialecte DocBook dispose d'un élément `title` de même nom que l'élément `title` du Dublin Core. Ces deux éléments ne sont pas confondus dans le document précédent, car l'élément `title` du Dublin Core a le préfixe `dc`.

4.2. Identification d'un espace de noms

Un *espace de noms* est identifié par un URI [Section 2.3] appelé *URI de l'espace de noms*. Cet URI est très souvent une URL mais il est sans importance que l'URL pointe réellement sur un document. Cet URI garantit seulement que l'espace de noms soit identifié de manière unique. Dans la pratique, l'URL permet aussi souvent d'accéder à un document qui décrit l'espace de noms. Une liste [Section 4.9] des URI associés aux principaux espaces de noms est donnée à la fin du chapitre.

4.3. Déclaration d'un espace de noms

Un espace de noms déclaré par un pseudo attribut de forme `xmlns:`*`prefix`* dont la valeur est une URL qui identifie l'espace de noms. Le préfixe *`prefix`* est un nom XML [Section 2.2.3] ne contenant pas le caractère `':'`. Il est ensuite utilisé pour *qualifier* les noms d'éléments. Bien que la déclaration d'un espace de noms se présente comme un attribut, celle-ci n'est pas considérée comme un attribut. Le langage XPath [Chapitre 6] distingue en effet les attributs des déclarations d'espaces de noms. Ces dernières sont manipulées de façon particulière.

Un *nom qualifié* d'élément prend la forme *`prefix`*`:`*`local`* où *`prefix`* est un préfixe associé à un espace de noms et *`local`* est le *nom local* de l'élément. Ce nom local est également un nom XML ne contenant pas le caractère `':'`. Dans la terminologie XML, les noms sans caractère `':'` sont appelés `NCNAME` qui est l'abréviation de *No Colon Name* et les noms qualifiés sont appelés `QNAME` qui est, bien sûr, l'abréviation de *Qualified Name*.

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

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

Il est habituel d'associer l'espace de noms XHTML au préfixe `html` plutôt qu'à `hns`. L'exemple precédent devient alors l'exemple suivant qui est un document équivalent.

```
<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, `xsd` ou `xs` pour les schémas XML et `xsl` pour les feuilles de style XSL.

Il est bien sûr possible de déclarer plusieurs espaces 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 `mml`.

```
<html:html xmlns:html="http://www.w3.org/1999/xhtml"
           xmlns:mml="http://www.w3.org/1998/Math/MathML">
  <html:head>
```

```
    <html:title>Espaces de noms</html:title>
  </html:head>
  <html:body>
    ...
    <mml:math>
      <mml:apply>
        <mml:eq/>
        ...
      </mml:apply>
    </mml: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 `surname` font partie du même espace de noms. L'exemple suivant est uniquement donné pour illustrer le propos mais il n'est pas à suivre. C'est une mauvaise pratique d'associer deux préfixes au même URI.

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

4.4. Portée d'une déclaration

La portée d'une déclaration d'un espace de noms est l'élément dans lequel elle est faite. L'exemple précédent aurait pu aussi être écrit de la manière suivante. Il faut remarquer que la portée de la déclaration comprend les balises de l'élément qui la contient. Il est ainsi possible d'utiliser le préfixe `html` dans l'élément `html` pour obtenir le nom qualifié `html:html`.

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

4.5. Espace de noms par défaut

Il existe un *espace de noms par défaut* associé au préfixe vide. Son utilisation permet d'alléger l'écriture des documents XML en évitant de mettre un préfixe aux éléments les plus fréquents. Lorsque plusieurs espaces de noms coexistent au sein d'un document, il faut, en général, réserver l'espace de noms par défaut à l'espace de noms le plus utilisé. Dans le cas des schémas [Section 5.13], il est souvent pratique de prendre pour espace de noms par défaut l'espace de noms cible.

L'espace de noms par défaut peut être spécifié par un pseudo attribut de nom `xmlns` dont la valeur est l'URI de l'espace de noms. Lorsque celui a été spécifié, les éléments dont le nom n'est pas qualifié font partie de l'espace de noms par défaut. 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>
    ...
    <mml:math xmlns:mml="http://www.w3.org/1998/Math/MathML">
      <mml:apply>
        <mml:eq/>
        ...
      </mml:apply>
    </mml: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 -->
  <!-- Tous les éléments html, head, title, body, ...
       appartiennent à l'espace de noms par défaut. -->
  <head>
   <title>Espaces de noms</title>
  </head>
  <body>
    ...
    <name xmlns="">
      <!-- L'espace de noms par défaut n'est plus spécifié -->
      <!-- Les trois éléments name, firstname et surname
           n'appartiennent à aucun espace de noms. -->
      <firstname>Gaston<firstname>
      <surname>Lagaffe<surname>
    </name>
```

```
    ...
  </body>
</html>
```

Une conséquence de la remarque précédente est que dans un document XML sans déclaration d'espace de noms, tous les éléments ne font partie d'aucun espace de noms. Ce comportement assure une compatibilité des applications avec les documents sans espace de noms.

4.6. Attributs

Les attributs peuvent également avoir des noms qualifiés formés d'un préfixe et d'un nom local. Ils font alors partie de l'espace de noms auquel est associé le préfixe. Dans l'exemple suivant, l'attribut `noNamespaceSchemaLocation` fait partie de l'espace de noms des instances de schémas identifié par l'URI `http://www.w3.org/2001/XMLSchema-instance`. Le nom de l'attribut `noNamespaceSchemaLocation` doit donc avoir un préfixe associé à cette URI. La déclaration de l'espace de noms peut avoir lieu dans le même élément, comme dans l'exemple ci-dessous, puisque la portée de celle-ci est l'élément tout entier.

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

En revanche, les attributs dont le nom n'est pas qualifié ne font jamais partie de l'espace de noms par défaut. Cette règle s'applique que l'espace de noms par défaut soit spécifié ou non. Dans l'exemple ci-dessous, l'élément `book` appartient à l'espace de noms DocBook puisque celui-ci est déclaré comme l'espace de noms par défaut. L'attribut `id` appartient à l'espace de noms XML et l'attribut `version` n'appartient à aucun espace de noms.

```
<book version="5.0"
      xml:id="course.xml"
      xmlns="http://docbook.org/ns/docbook">
```

4.7. Espace de noms XML

Le préfixe `xml` est toujours implicitement lié à l'espace de noms XML identifié par l'URI est `http://www.w3.org/XML/1998/namespace`. Cet espace de noms n'a pas besoin d'être déclaré. Les quatre attributs particuliers [Section 2.7.4] `xml:lang`, `xml:space`, `xml:base` et `xml:id` font partie de cet espace de noms.

Ces quatre attributs sont déclarés par le schéma XML [Chapitre 5] qui se trouve à l'URL `http://www.w3.org/2001/xml.xsd`. Ce schéma peut être importé [Section 5.14] par un autre schéma pour ajouter certains de ces attributs à des éléments.

4.8. Espaces de noms et DTD

Les DTD ne prennent pas compte les espaces de noms. Il est cependant possible de valider, avec une DTD, un document avec des espaces de noms au prix de quelques entorses à l'esprit des espaces de noms. Il y a, en effet, quelques contraintes. D'une part, la déclaration d'espace de noms est vue comme un attribut de nom commençant par `xmlns:`. Il est donc nécessaire de le déclarer comme tout autre attribut. D'autre part, les noms qualifiés des éléments sont considérés comme des noms contenant le caractère `':'`. Il faut donc déclarer les éléments avec leur nom qualifié.

Les éléments du document suivant font partie de l'espace de noms identifié par l'URL `http://www.liafa.jussieu.fr/~carton/` qui est associé au préfixe `tns`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE list SYSTEM "valid.dtd">
```

```
<tns:list xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <tns:item>Item 1</tns:item>
  <tns:item>Item 2</tns:item>
</tns:list>
```

La DTD suivante valide le document précédent. Elle déclare un attribut `xmlns:tns` pour la déclaration d'espace de noms. De plus, les éléments sont déclarés avec leurs noms qualifiés `tns:list` et `tns:item`.

```
<!-- Fichier "valid.dtd" -->
<!ELEMENT tns:list (tns:item)+>
<!ATTLIST tns:list xmlns:tns CDATA #REQUIRED>
<!ELEMENT tns:item (#PCDATA)>
```

En revanche, le document suivant n'est pas valide pour la DTD précédent alors qu'il est équivalent au document précédent. Le préfixe `tns` a simplement été remplacé par le préfixe `ons`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE list SYSTEM "valid.dtd">
<ons:list xmlns:ons="http://www.liafa.jussieu.fr/~carton/">
  <ons:item>Item 1</ons:item>
  <ons:item>Item 2</ons:item>
</ons:list>
```

4.9. Quelques espaces de noms classiques

XML [Section 4.7]
`http://www.w3.org/XML/1998/namespace`

XInclude [Section 2.9]

`http://www.w3.org/2001/XInclude`

XLink

`http://www.w3.org/1999/xlink`

MathML
`http://www.w3.org/1998/Math/MathML`

XHTML
`http://www.w3.org/1999/xhtml`

SVG [Chapitre 11]
`http://www.w3.org/2000/svg`

Schémas [Chapitre 5]
`http://www.w3.org/2001/XMLSchema`

Instances de schémas
`http://www.w3.org/2001/XMLSchema-instance`

Schematron [Chapitre 7]
`http://purl.oclc.org/dsdl/schematron`

XSLT [Chapitre 8]
`http://www.w3.org/1999/XSL/Transform`

XSL-FO [Chapitre 9]
`http://www.w3.org/1999/XSL/Format`

DocBook [Section 1.5]

```
http://docbook.org/ns/docbook
```

Dublin Core

```
http://purl.org/dc/elements/1.1/
```

Chapitre 5. Schémas XML

5.1. Introduction

Les *schémas XML* permettent, comme les DTD [Chapitre 3], de définir des modèles de documents. Il est ensuite possible de vérifier qu'un document donné est valide pour un schéma, c'est-à-dire respecte les contraintes du schéma. Les schémas ont été introduits pour combler certaines lacunes des DTD.

5.1.1. Comparaison avec les DTD

La première différence entre les schémas et les DTD est d'ordre syntaxique. La syntaxe des DTD est une syntaxe héritée de SGML qui est différente de la syntaxe XML pour le corps des documents. En revanche, la syntaxe des schémas est une syntaxe purement XML. Un schéma est, en effet, un document XML à part entière avec un élément racine `xsd:schema` et un espace de noms.

Les DTD manquent cruellement de précision dans la description des contenus des éléments. Cette lacune se manifeste surtout au niveau des contenus textuels et des contenus mixtes. Il est, par exemple, impossible d'imposer des contraintes sur les contenus textuels [Section 3.6.2] des éléments. Le seul type possible pour les contenus textuels est `#PCDATA` qui autorise toutes les chaînes de caractères. Les types pour les attributs sont un peu plus nombreux mais ils restent encore très limités. À l'inverse, les schémas possèdent une multitude de types prédéfinis [Section 5.5.1] pour les contenus textuels. Ces types couvrent les chaînes de caractères, les nombres comme les entiers et les flottants ainsi que les heures et les dates. Ces types peuvent, en outre, être affinés par des mécanismes de restriction et d'union. Il est possible de définir, à titre d'exemple, des types pour les entiers entre `1` et `12`, les flottants avec deux décimales ou les chaînes d'au plus 16 caractères ne comportant que des chiffres et des tirets `'-'`.

Les DTD sont encore plus limitées dans la description des contenus mixtes [Section 3.6.3]. La seule possibilité est d'exprimer que le contenu d'un élément est un mélange, sans aucune contrainte, de texte et de certains éléments. Les schémas comblent cette lacune en permettant d'avoir des contenus mixtes [Section 5.5.4] aussi précis que les contenus purs qui décrivent l'ordre et les nombres d'occurrences des enfants d'un élément.

Dans une DTD, le contenu pur d'un élément est décrit directement par une expression rationnelle. Les schémas procèdent en deux étapes. Ils définissent des types qui sont ensuite associés aux éléments. Les schémas se distinguent des DTD par leurs possibilités de définir de nouveaux types. Il est d'abord possible de construire des types [Section 5.6] explicitement à la manière des DTD. Il existe ensuite des mécanismes permettant de définir un nouveau type à partir d'un autre type, soit prédéfini, soit déjà défini dans le schéma. Ce nouveau type est obtenu soit par extension [Section 5.8] soit par restriction [Section 5.9] du type de départ. L'extension consiste à enrichir le type en ajoutant du contenu et des attributs. La restriction consiste, à l'inverse, à ajouter des contraintes pour restreindre les contenus valides. Ces deux mécanismes permettent ainsi de construire une véritable hiérarchie de types semblable à l'approche orientée objet des langages comme Java ou C++.

Les schémas autorisent des facilités impossibles avec les DTD. La première est la possibilité d'avoir plusieurs éléments locaux [Section 5.4.5] avec des noms identiques mais avec des types et donc des contenus différents. Dans une DTD, un élément a une seule déclaration qui décrit ses contenus possibles pour toutes ses occurrences dans un document. Une deuxième facilité est formée des mécanismes de substitution de types et d'éléments [Section 5.10]. À titre d'exemple, un schéma peut prévoir qu'un élément puisse se substituer à un autre élément. Ces substitutions fonctionnent de paire avec la hiérarchie des types.

Les DTD ont une modularité très limitée et l'écriture de DTD d'envergure est un exercice difficile. Les seuls dispositifs mis à disposition des auteurs de DTD sont l'import de DTD externes [Section 3.2.2] et les entités paramètres [Section 3.5.2]. Les schémas possèdent plusieurs mécanismes destinés à une plus grande modularité. Le premier d'entre eux est la possibilité, pour les schémas, de définir des types par extension et restriction. Il existe également les groupes d'éléments et les groupes d'attributs [Section 5.11].

Les DTD proviennent de SGML et sont antérieures aux espaces de noms. Pour cette raison, elles ne les prennent pas en compte. La déclaration d'un élément se fait en donnant le nom qualifié de l'élément avec le préfixe et le nom local. Ceci impose, dans les documents, d'associer l'espace de noms à ce même préfixe. Ceci est contraire à

l'esprit des espace de noms où le préfixe est juste une abréviation interchangeable pour l'URI de l'espace de noms. Les schémas, au contraire, prennent en compte les espaces de noms [Section 5.13]. Un schéma déclare d'abord un espace de noms cible. Les éléments et les attributs sont ensuite déclarés, dans le schéma, avec leur nom local. Un document qui mélange des éléments et des attributs provenant d'espaces de noms différents peut encore être validé à l'aide des différents schémas pour les espaces de noms.

5.2. Un premier exemple

Voici un exemple de schéma XML définissant le type de document de la bibliographie [bibliography.xml]. Ce schéma est volontairement rudimentaire pour un premier exemple. Il n'est pas très précis sur les contenus de certains éléments. Un exemple plus complet peut être donné pour la bibliographie.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">❶
  <xsd:annotation>❷
    <xsd:documentation xml:lang="fr">
      Schéma XML pour bibliography.xml
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="bibliography" type="Bibliography"/>❸

  <xsd:complexType name="Bibliography">❹
    <xsd:sequence>
      <xsd:element name="book" minOccurs="1" maxOccurs="unbounded">❺
        <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:attribute name="key"  type="xsd:NMTOKEN" use="required"/>❻
          <xsd:attribute name="lang" type="xsd:NMTOKEN" use="required"/>❼
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

❶ Élément racine `xsd:schema` avec la déclaration de l'espace de noms des schémas associé au préfixe `xsd`.
❷ Documentation du schéma.
❸ Déclaration de l'élément `bibliography` avec le type `Bibliography`.
❹ Début de la définition du type `Bibliography`.
❺ Déclaration de l'élément `book` dans le contenu du type `Bibliography`.
❻❼ Déclaration des attributs `key` et `lang` de l'élément `book` avec le type `xsd:NMTOKEN`.

Ce schéma déclare l'élément `bibliography` du type `Bibliography` qui est ensuite introduit par l'élément `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`.

Dans tout ce chapitre, la convention suivante est appliquée. Les noms des éléments sont en minuscules alors que les noms des types commencent par une majuscule comme les classes du langage Java. Les noms de l'élément et de son type ne se différencient souvent que par la première lettre comme `bibliography` et `Bibliography` dans l'exemple précédent.

Comme pour les DTD, il existe des sites WEB permettant de valider un document vis à vis d'un schéma. Cette validation peut également être effectuée avec le logiciel `xmllint`. L'option `--schema` permet de passer en paramètre un schéma en plus du document XML.

5.3. Structure globale d'un schéma

Un schéma XML se compose essentiellement de déclarations d'éléments et d'attributs et de définitions de types. Chaque élément est déclaré avec un type qui peut être, soit un des types prédéfinis, soit un nouveau type défini dans le schéma. Le type spécifie quels sont les contenus valides de l'élément ainsi que ses attributs. Un nouveau type est obtenu soit par *construction*, c'est-à-dire une description explicite des contenus qu'il autorise, soit par *dérivation*, c'est-à-dire modification d'un autre type. Un schéma peut aussi contenir des imports d'autres schémas, des définitions de groupes d'éléments et d'attributs et des contraintes de cohérences.

L'espace de noms des schémas XML est identifié par l'URL `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 `xsd:schema`. 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 définitions de types -->
  ...
</xsd:schema>
```

Les éléments, attributs et les types peuvent être *globaux* ou *locaux*. Ils sont globaux lorsque leur déclaration ou leur définition est enfant direct de l'élément `xsd:schema`. Sinon, ils sont locaux. Ces objets et eux seuls peuvent être référencés dans tout le schéma pour être utilisé. Dans le premier exemple de schéma donné ci-dessus, l'élément `bibliography` est global alors que l'élément `title` est local. La déclaration de ce dernier intervient au sein de la définition du type `Bibliography` qui est global. Seuls les types globaux sont nommés. Dans le schéma ci-dessus, le type de l'élément `book` est anonyme. Les éléments et attributs sont toujours nommés, qu'ils soient globaux ou locaux. Ils doivent, bien sûr, avoir un nom pour apparaître dans un document.

Les objets globaux et locaux se comportent différement vis à vis de l'espace de noms cible du schéma [Section 5.13]. Les objets globaux appartiennent toujours à cet espace de noms. Les objets locaux, au contraire, appartiennent ou n'appartiennent pas à l'espace de noms cible suivant les valeurs des attributs `form`, `elementFormDefault` et `attributeFormDefault`.

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

5.3.1. Attributs de l'élément `xsd:schema`

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

`targetNamespace`

> La valeur de cet attribut est l'URI qui identifie l'espace de noms cible [Section 5.13], c'est-a-dire l'espace de noms des éléments et types définis par le schéma. Si cet attribut est absent, les élements 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` [Section 5.13] pour respectivement les éléments et les attributs. Les valeurs possibles sont `qualified` et `unqualified`. La valeur par défaut est `unqualified`.

`blockDefault` et `finalDefault`

> Ces deux attributs donnent la valeur par défaut des attributs `block` et `final` [Section 5.10.4]. 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`.

Dans l'exemple suivant, le schéma déclare que l'espace de noms cible est `http://www.liafa.jussieu.fr/~carton` et que tous les éléments doivent être qualifiés dans les documents valides. L'espace de noms par défaut est déclaré égal à l'espace de noms cible afin de simplifier l'écriture du schéma.

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

5.3.2. 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'URI `http://www.w3.org/2001/XMLSchema-instance`. L'attribut `schemaLocation` est utilisé lors de l'utilisation d'espaces de noms alors que l'attribut `noNamespaceSchemaLocation` est utilisé lorsque le document n'utilise pas d'espace de noms

La valeur de l'attribut `schemaLocation` est une suite d'URI séparées par des espaces. Ces URI vont par paires et le nombre d'URI doit donc être pair. La première URI de chaque paire identifie un espace de noms et la seconde donne l'adresse du schéma à utiliser pour les éléments et attributs dans cet espace de noms. L'espace de noms identifié par la première URI doit donc être l'espace de noms cible du schéma donné par la seconde. La valeur de l'attribut `schemaLocation` prend donc la forme générale suivante

```
schemaLocation="namespace1 schema1 namespace2 ... namespaceN schemaN"
```

où `namespacei` est l'espace de noms cible du schéma `schemai`.

Le logiciel qui effectue la validation se base sur la valeur de l'attribut `schemaLocation` pour chercher la définition de chaque élément ou attribut dans le schéma correspondant à son espace de noms.

La valeur de l'attribut `noNamespaceSchemaLocation` est simplement l'URL d'un unique schéma qui doit permettre de valider l'intégralité du document. Il n'est, en effet, pas possible de distinguer les éléments qui n'ont pas d'espace de noms.

Dans l'exemple suivant, le document déclare que le schéma se trouve dans le fichier local `bibliography.xsd` et que l'espace de noms cible de ce schéma est identifié par l'URL `http://www.liafa.jussieu.fr/~carton/`.

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

5.3.3. Documentation

L'élément `xsd:annotation` permet d'ajouter des commentaires dans un schéma. Il peut être enfant de l'élément `xsd:schema` pour des commentaires globaux. Il peut également être enfant des éléments `xsd:element`, `xsd:attribute` pour ajouter des commentaires aux déclarations d'éléments et d'atttributs ainsi que de `xsd:simpleType` et `xsd:complexType` pour ajouter des commentaires aux définitions de type. Contrairement aux commentaires XML [Section 2.7.5], ces commentaires font partie à part entière du schéma XML et constituent sa documentation.

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

5.4. Déclarations d'éléments

Pour qu'un document soit valide pour un schéma, tout élément apparaissant dans le document doit être déclaré dans le schéma. Cette déclaration lui donne un type qui détermine, d'une part, les contenus possibles et, d'autres part, les attributs autorisés et obligatoires. Contrairement aux DTD, les attributs ne sont pas directement associés aux éléments. Ils font partie des types qui sont donnés aux éléments.

Le type donné à un élément peut être soit un type nommé soit un type anonyme. Dans le premier cas, le type est soit un type prédéfini dont le nom fait partie de l'espace de noms des schémas soit un type défini globalement dans le schéma. Dans le second cas, le type est défini explicitement à la déclaration de l'élément.

5.4.1. Type nommé

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

```
<xsd:element name="element" type="type"/>
```

où *element* 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="tns:Title"/>
```

5.4.2. 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 cela donner des valeurs aux attributs `default` ou `fixed` de l'élément `xsd:element`.

```
<xsd:element name="title" type="xsd:string" default="Titre par défaut"/>

<xsd:element name="title" type="xsd:string" fixed="Titre fixe"/>
```

5.4.3. 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 *element* est le nom de l'élément déclaré.

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

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

5.4.4. 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 `xsd: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>
```

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.

5.4.5. Éléments locaux

Deux éléments définis non globalement dans un schéma peuvent avoir le même nom tout en ayant des types différents. Il s'agit en fait d'éléments différents mais ayant le même nom. Cette possibilité est absente des DTD où tous les éléments sont globaux et ne peuvent avoir qu'un seul type. Le schéma suivant définit deux éléments de même nom `local` mais de types `xsd:string` et `xsd:integer`. Le premier élément apparaît uniquement dans le contenu de l'élément `strings` alors que le second apparaît uniquement dans le contenu de l'élément `integers`. C'est donc le contexte qui permet de distinguer les deux éléments de nom `local`.

```
<?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 précédent 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>
```

5.5. Définitions 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.

Les schémas permettent de définir une hiérarchie de types qui sont obtenus par *extension* ou *restriction* 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"/>
```

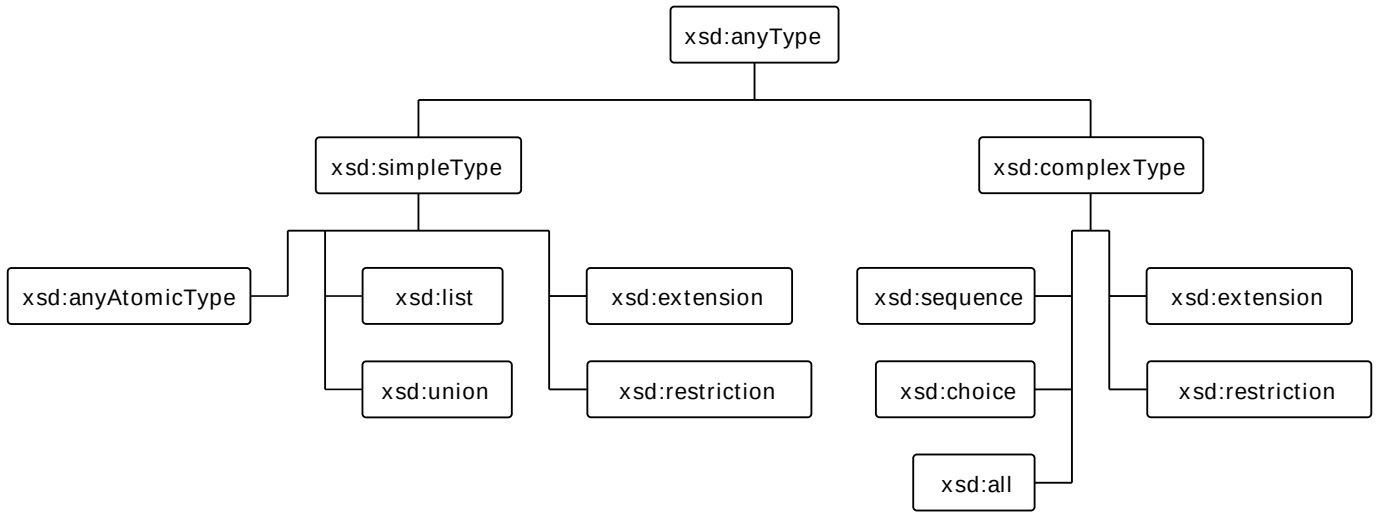


Figure 5.1. Hiérarchie de construction des types

5.5.1. Types prédéfinis

Les schémas possèdent de nombreux types prédéfinis. Certains, comme `xsd:int` et `xsd:float`, proviennent des langages de programmation, certains, comme `xsd:date` et `xsd:time`, sont inspirés de normes ISO (ISO 8601 dans ce cas) et d'autres encore, comme `xsd:ID`, sont hérités des DTD. Ces types autorisent l'écriture de schémas concis et très précis. Beaucoup d'entre eux pourraient être redéfinis par restriction [Section 5.9] de type de base mais leur présence comme type de base simplifie le travail.

5.5.1.1. Types numériques

Beaucoup de types numériques sont prédéfinis pour les nombres entiers et flottants. Certains types comme `xsd:int` ou `xsd:double` correspondent à un codage précis et donc à une précision fixée alors que d'autres types comme `xsd:integer` ou `xsd:decimal` autorisent une précision arbitraire. Ces derniers types sont à privilégier sauf quand le schéma décrit des données avec un codage bien déterminé.

`xsd:boolean`

Valeur booléenne avec `true` ou `1` pour *vrai* et `false` ou `0` pour *faux*

`xsd:byte`

Nombre entier signé sur 8 bits

`xsd:unsignedByte`

Nombre entier non signé sur 8 bits

`xsd:short`

Nombre entier signé sur 16 bits

`xsd:unsignedShort`

Nombre entier non signé sur 16 bits

`xsd:int`

Nombre entier signé sur 32 bits

`xsd:unsignedInt`

Nombre entier non signé sur 32 bits

`xsd:long`

Nombre entier signé sur 64 bits. Ce type dérive du type `xsd:integer`.

`xsd:unsignedLong`

Nombre entier non signé sur 64 bits

`xsd:integer`

Nombre entier sans limite de précision. Ce type n'est pas primitif et dérive du type `xsd:decimal`.

`xsd:positiveInteger`

Nombre entier strictement positif sans limite de précision

`xsd:negativeInteger`

Nombre entier strictement négatif sans limite de précision

`xsd:nonPositiveInteger`

Nombre entier négatif ou nul sans limite de précision

`xsd:nonNegativeInteger`

Nombre entier positif ou nul sans limite de précision

`xsd:float`

Nombre flottant sur 32 bits conforme à la norme IEEE 754 [W]

`xsd:double`

Nombre flottant sur 64 bits conforme à la norme IEEE 754 [W]

`xsd:decimal`

Nombre décimal sans limite de précision

5.5.1.2. Types pour les chaînes et les noms

Les schémas possèdent bien sûr un type pour les chaînes de caractères ainsi que quelques types pour les noms qualifiés et non qualifiés.

`xsd:string`

Chaîne de caractères composée de caractères Unicode

`xsd:normalizedString`

Chaîne de caractères normalisée, c'est-à-dire ne contenant pas de tabulation `U+09`, de saut de ligne `U+0A` ou de retour chariot `U+0D` [Section 2.2.2].

`xsd:token`

Chaîne de caractères normalisée (comme ci-dessus) et ne contenant pas en outre des espaces en début ou en fin ou des espaces consécutifs

`xsd:Name`

Nom XML [Section 2.2.3]

`xsd:QName`

Nom qualifié [Chapitre 4]

`xsd:NCName`

Nom non qualifié [Chapitre 4], c'est-à-dire sans caractère `':'`

`xsd:language`

Code de langue sur deux lettres de la norme ISO 639 comme `fr` ou `en` éventuellement suivi d'un code de pays de la norme ISO 3166 comme `en-GB`. C'est le type de l'attribut particulier `xml:lang` [Section 2.7.4.1] auquel il est spécialement destiné.

`xsd:anyURI`

Un URI comme `http://www.liafa.jussieu.fr/~carton/`. C'est le type de l'attribut particulier `xml:base` [Section 2.7.4.3].

`xsd:base64Binary`

Données binaires représentées par une chaîne au format Base 64.

`xsd:hexBinary`

Données binaires représentées par une chaîne au format Hex.

Les types `xsd:normalizedString` et `xsd:token` méritent quelques explications. D'une part, il faut bien distinguer le type `xsd:token` du type `xsd:NMTOKEN`. Le type `xsd:token` accepte des valeurs contenant éventuellement des espaces alors que le type `xsd:NMTOKEN` accepte uniquement un jeton [Section 2.2.3] qui ne contient jamais d'espace. D'autre part, les deux types `xsd:normalizedString` et `xsd:token` ne restreignent pas les valeurs possibles pour un document valide mais modifient le traitement des caractères d'espacement [Section 2.2.2] à l'analyse lexicale. Le type `xsd:normalizedString` n'interdit pas de mettre des caractères d'espacement autres que des espaces. En revanche, tous les caractères d'espacement sont convertis en espaces par l'analyseur lexical. De la même façon, le type `xsd:token` n'interdit pas de mettre des caractères d'espacement en début ou en fin ou des caractères d'espacement consécutifs. En revanche, les caractères d'espacement en début ou en fin sont supprimés et les suites de caractères d'espacement sont remplacées par un seul espace par l'analyseur lexical.

5.5.1.3. Types pour les dates et les heures

Quelques types des schémas imposent des formats standards pour écrire les dates et les heures. Ils s'appuient sur la norme ISO 8601. Leur grand intérêt est d'imposer une écriture normalisée pour les dates et les heures et d'en faciliter ainsi le traitement.

`xsd:time`

Heure au format `hh:mm:ss[.sss][TZ]`, par exemple `14:07:23`. La partie fractionnaire `.sss` des secondes est optionnelle. Tous les autres champs sont obligatoires. Les nombres d'heures, minutes et de secondes doivent être écrits avec deux chiffres en complétant avec `0`. L'heure peut être suivie d'un décalage horaire `TZ` qui est soit `Z` pour le temps universel soit un décalage commençant par `+` ou `-` comme `-07:00`.

`xsd:date`

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

`xsd:dateTime`

Date et heure au format `YYYY-MM-DDThh:mm:ss` comme `2008-01-16T14:07:23`. Tous les champs sont obligatoires.

`xsd:duration`

Durée au format `PnYnMnDTnHnMnS` comme `P1Y6M`, `P1M12DT2H` et `P1YD3H10S`.

`xsd:dayTimeDuration`

Durée au format `PnDTnHnMnS` comme `P7DT4H3M2S`.

`xsd:yearMonthDuration`

Durée au format `PnYnM` comme `P1Y6M`.

`xsd:gYear`

Année du calendrier grégorien au format `YYYY` comme `2011`.

`xsd:gYearMonth`

Année et mois du calendrier grégorien au format `YYYY-MM` comme `1966-06` pour juin 1966.

`xsd:gMonth`

Mois du calendrier grégorien au format `MM` comme `01` pour janvier.

`xsd:gMonthDay`

Jour et mois du calendrier grégorien au format `MM-DD` comme `12-25` pour le jour de Noël.

`xsd:gDay`

Jour (dans le mois) du calendrier grégorien au format `DD` comme `01` pour le premier de chaque mois.

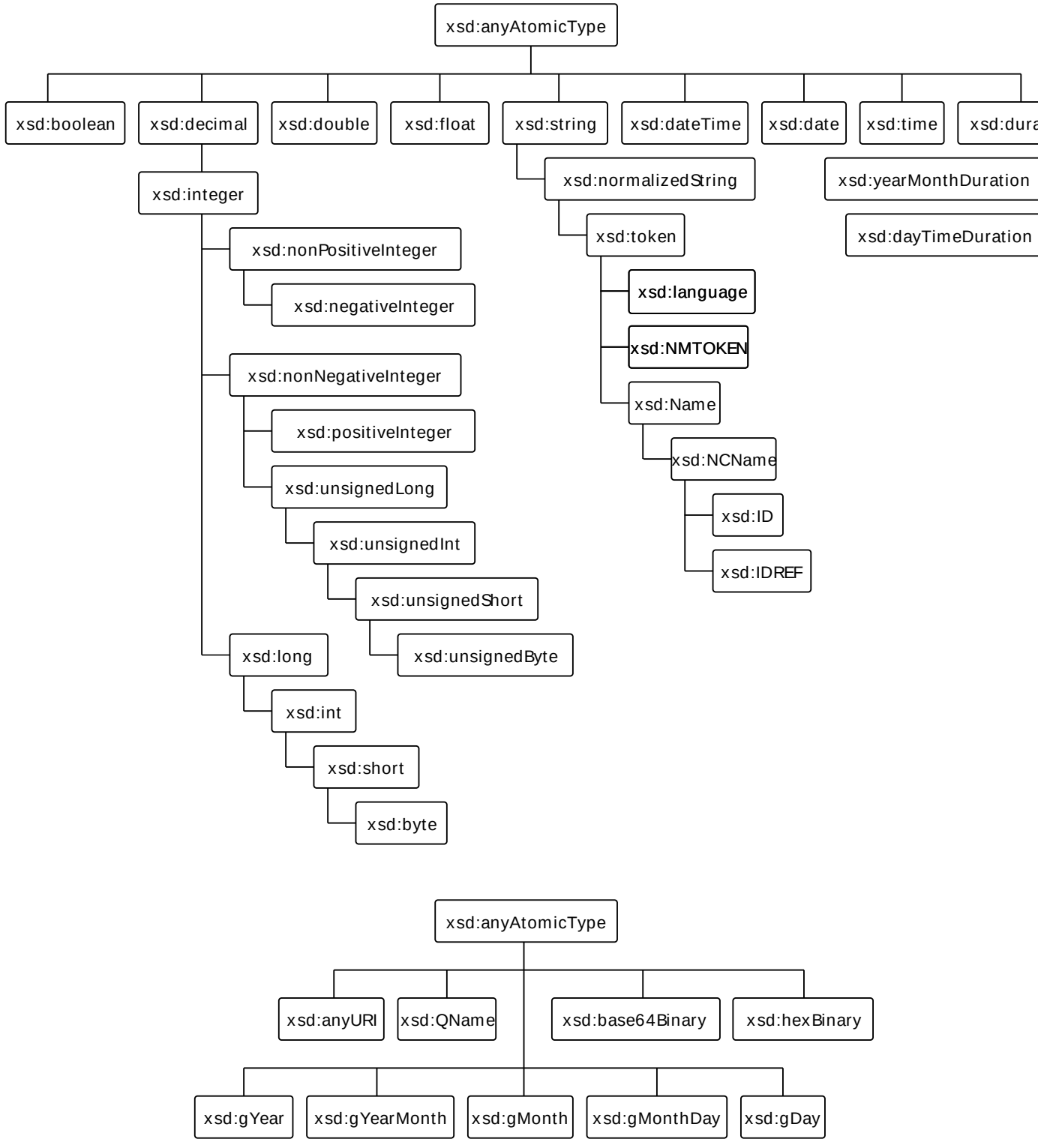


Figure 5.2. Hiérarchie des types atomiques

5.5.1.4. Types hérités des DTD

Les schémas XML reprennent certains types des DTD [Section 3.7.1] pour les attributs. Ces types facilitent la traduction automatique des DTD en schémas. Pour des raisons de compatibilité, ces types sont réservés aux attributs.

`xsd:ID`

nom XML identifiant un élément

`xsd:IDREF`

référence à un élément par son identifiant

`xsd:IDREFS`

liste de références à des éléments par leurs identifiants

`xsd:NMTOKEN`

jeton [Section 2.2.3]

`xsd:NMTOKENS`

liste de jetons [Section 2.2.3] separés par des espaces

`xsd:ENTITY`

entité externe non XML

`xsd:ENTITIES`

liste d'entités externes non XML separés par des espaces

`xsd:NOTATION`

notation

5.5.2. Types simples

Les types simples définissent uniquement des contenus textuels. Ils peuvent être utilisé pour les éléments ou les attributs. Ils 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>
```

5.5.3. Types complexes

Les types complexes définissent des contenus purs (constitués uniquement d'éléments), des contenus textuels ou des contenus mixes. Tous ces contenus peuvent comprendre des attributs. Les types complexes peuvent seulement

être utilisés 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 [Section 5.8] ou restriction [Section 5.9].

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>
```

Si le type est obtenu par extension ou restriction d'un autre type, l'élément `xsd:complexType` doit contenir un élément `xsd:simpleContent` ou `xsd: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>
```

5.5.4. Contenu mixte

Le contenu d'un élément est *pur* lorsqu'il ne contient que des éléments qui, eux-mêmes, peuvent à leur tour contenir du texte et/ou des éléments. Il est, au contraire, *mixte* lorsqu'il contient du texte autre que des caractères d'espacement en dehors de ses enfants. La construction de types pour les contenus mixtes est très facile et très souple avec les schémas. L'attribut `mixed` de l'élément `xsd:complexType` permet de construire un type avec du contenu mixte. Il faut, pour cela, lui donner la valeur `true`.

Le contenu d'un élément est valide pour un type déclaré mixte si le contenu devient valide pour le type non mixte correspondant lorsque tout le texte en dehors des enfants est supprimé. Dans l'exemple suivant, l'élément `person` doit contenir, un élément `firstname` et un élément `lastname` dans cet ordre. Puisque son type est déclaré mixte, il peut en outre contenir du texte comme ci-dessous.

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

Le document suivant est valide pour le schéma précédent. En revanche, il ne le serait pas sans la valeur `true` donnée à l'attribut `mixed` dans le schéma. Le contenu de l'élément `person` est valide car si on retire le texte en dehors de ses enfants `firstname` et `lastname` (texte en gras ci-dessous), on obtient un contenu valide pour

le type sans `mixed="true"`. Il serait, par exemple, impossible de changer l'ordre des enfants `firstname` et `lastname` car ce type spécifie qu'ils doivent apparaître dans cet ordre.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<person>
  Prénom : <firstname>Albert</firstname>,
  Nom : <lastname>Einstein</lastname>.
</person>
```

Le schéma précédent n'a pas d'équivalent dans les DTD. Dès que le contenu d'un élément est déclaré mixte dans une DTD [Section 3.6.3], il n'y a plus aucun contrôle sur l'ordre et le nombre d'occurrences de ses enfants. Le schéma suivant donne un exemple de contenu mixte équivalent à un contenu mixte d'une DTD.

```
<xsd:element name="book">
  <xsd:complexType mixed="true">
    <xsd:choice minOccurs="0" 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)* >
```

5.6. 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 DTD [Section 3.6.1].

5.6.1. Élément vide

Si un type complexe déclare uniquement des attributs, le contenu de l'élément doit être 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>
```

Un fragement de document valide peut être le suivant.

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

5.6.2. 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 DTD [Section 3.6.1]. 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)>
```

Un fragment de document ci-dessous est valide pour la déclaration de l'élément `book`.

```
<book>
  <title>XML langage et applications</title>
  <author>Alain Michard</author>
  <year>2001</year>
  <publisher>Eyrolles</publisher>
</book>
```

Un autre exemple de type construit avec l'opérateur `xsd:sequence` est donné comme exemple de type mixte [Section 5.5.4]. Le nombre d'occurences de chaque élément dans la séquence est 1 par défaut mais il peut être modifié par les attributs `minOccurs` et `maxOccurs` [Section 5.6.7].

5.6.3. 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 DTD [Section 3.6.1]. 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)>
```

Un autre exemple de type construit avec l'opérateur `xsd:choice` est donné comme exemple de type mixte [Section 5.5.4]. Il est bien sûr possible d'imbriquer les opérateurs `xsd:sequence` et `xsd:choice`. Dans l'exemple suivant, l'élément `book` contient soit un seul élément `author` soit un élément `authors` contenant au moins deux éléments `author`. Cette dernière contrainte est imposée par la valeur `2` de l'attribut `minOccurs` [Section 5.6.7].

```
<xsd:element name="book" minOccurs="1" maxOccurs="unbounded">
  <xsd:complexType>
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:choice>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="authors">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="author" type="xsd:string"
                         minOccurs="2" maxOccurs="unbounded"/>
```

```
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
    <xsd:element name="year" type="xsd:string"/>
    ...
  </xsd:sequence>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<bibliography>
  <!-- Livre à un seul auteur -->
  <book key="Michard01" lang="fr">
    <title>XML langage et applications</title>
    <author>Alain Michard</author>
    <year>2001</year>
    <publisher>Eyrolles</publisher>
  </book>
  <!-- Livre à deux auteurs -->
  <book key="ChagnonNolot07" lang="fr">
    <title>XML</title>
    <authors>
      <author>Gilles Chagnon</author>
      <author>Florent Nolot</author>
    </authors>
    <year>2007</year>
    <publisher>Pearson</publisher>
  </book>
</bibliography>
```

5.6.4. Opérateur d'ensemble

L'opérateur `xsd:all` n'a pas d'équivalent dans les DTD. 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.

```
<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>
```

Le document suivant est un document valide où l'élément `book` a la déclaration précédente. L'ordre des enfants de l'élément `book` peut être variable.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<bibliography>
  <book key="Michard01" lang="fr">
    <author>Alain Michard</author>
    <title>XML langage et applications</title>
    <publisher>Eyrolles</publisher>
    <year>2001</year>
  </book>
```

```
  <book key="Zeldman03" lang="en">
    <title>Designing with web standards</title>
    <author>Jeffrey Zeldman</author>
    <year>2003</year>
    <publisher>New Riders</publisher>
  </book>
  ...
</bibliography>
```

L'utilisation de l'élément `xsd:all` doit respecter quelques contraintes qui limitent fortement son intérêt. Un opérateur `xsd:all` ne peut pas être imbriqué avec d'autres constructeurs `xsd:sequence`, `xsd:choice` ou même `xsd:all`. D'une part, les seuls enfants possibles de `xsd:all` sont des éléments `xsd:element`. D'autre part, l'élément `xsd:all` est toujours enfant de `xsd:complexType` ou `xsd:complexContent`.

Les attributs `minOccurs` et `maxOccurs` 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. Les attributs `minOccurs` et `maxOccurs` peuvent aussi apparaître comme attribut de `xsd:all`. Dans ce cas, leurs valeurs s'appliquent à tous les éléments `xsd:element` enfants de `xsd:all`. Les valeurs autorisées pour `minOccurs` sont `0` et `1` et la seule valeur autorisée pour `maxOccurs` est `1`.

Dans la déclaration suivante de l'élément `book`, ses enfants peuvent apparaître dans n'importe quel ordre et chacun d'eux peut avoir 0 ou 1 occurrence.

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:all minOccurs="0">
      <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>
```

5.6.5. 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 équivalence 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>
```

5.6.6. Opérateur de liste

L'opérateur `xsd:list` définit un nouveau type simple dont les valeurs sont les listes de valeurs du type simple donné par l'attribut `itemType`. Il ne s'agit pas de listes générales comme dans certains langages de programmation. Il s'agit uniquement de listes de valeurs séparées par des espaces. Ces listes sont souvent utilisées comme valeurs d'attributs. Les valeurs du type simple donné par `itemType` ne peuvent pas comporter de caractères d'espacement [Section 2.2.2] qui pertuberaient la séparation entre les valeurs. L'exemple suivant définit des types pour les listes d'entiers et pour les listes de 5 entiers.

```
<!-- Type pour les listes d'entiers -->
<xsd:simpleType name="IntList">
  <xsd:list itemType="xsd:integer"/>
</xsd:simpleType>
<!-- Type pour les listes de 5 entiers -->
<xsd:simpleType name="IntList5">
  <xsd:restriction base="IntList">
    <xsd:length value="5"/>
  </xsd:restriction>
</xsd:simpleType>
```

5.6.7. 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 DTD. 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 `maxOccurs` 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 | elem3), elem4) >
```

```
   <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>
```

```
<!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>
```

5.6.8. Joker `xsd:any`

L'opérateur `xsd:any` permet d'introduire dans un document un ou des éléments externes au schéma, c'est-à-dire non définis dans le schéma. Le nombre d'éléments externes autorisés peut-être spécifié avec les attributs `minOccurs` et `maxOccurs` [Section 5.6.7]. La validation de ces éléments externes est contrôlée par l'attribut `processContents` qui peut prendre les valeurs `strict`, `lax` et `skip`. La valeur par défaut est `strict`. Lorsque la valeur est `strict`, les éléments externes doivent être validés par un autre schéma déterminé par l'espace de noms de ces éléments pour que le document global soit valide. Lorsque la valeur est `skip`, les éléments externes ne sont pas validés. La valeur `lax` est intermédiaire entre `strict` et `skip`. La validation des éléments externes est tentée mais elle peut échouer.

Le schéma suivant autorise zéro ou un élément externe dans le contenu de l'élément `person` après l'élément `lastname`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="person">
```

```
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string"/>
        <xsd:element name="lastname"  type="xsd:string"/>
        <xsd:any processContents="lax" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="utf-8"?>
<person>
  <firstname>Elizabeth II Alexandra Mary</firstname>
  <lastname>Windsor</lastname>
  <title>Queen of England</title>
</person>
```

L'attribut `namespace` de l'élément `xsd:any` permet de préciser les espaces de noms auxquels doivent appartenir les éléments externes. Sa valeur doit être une suite d'URI identifiant des espaces de noms séparés par des espaces. Les valeurs particulières `##any`, `##other`, `##local` et `##targetNamepspace` peuvent aussi apparaître dans la liste des espace de noms. La valeur par défaut `##any` n'impose aucune restriction sur l'espace de noms des éléments externes. Les valeurs `##targetNamepspace` et `##other` autorisent respectivement des éléments dont l'espace de noms est égal ou différent de l'espace de noms cible du schéma. La valeur `##local` autorise des éléments ayant des noms non qualifiés et n'appartenant à aucun espace de noms.

Dans le schéma suivant, l'élément externe ajouté dans le contenu de l'élément `person` doit appartenir à l'espace de noms XHTML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string"/>
        <xsd:element name="lastname"  type="xsd:string"/>
        <xsd:any processContents="lax" namespace="http://www.w3.org/1999/xhtml"
                 minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="utf-8"?>
<person>
  <firstname>Victor</firstname>
  <lastname>Hugo</lastname>
  <html:ul xmlns:html="http://www.w3.org/1999/xhtml">
    <li>Romancier</li>
    <li>Poète</li>
    <li>Dramaturge</li>
  </html:ul>
</person>
```

Il existe également un élément `xsd:anyAttribute` [Section 5.7.4] pour autoriser des attributs externes au schéma.

5.6.9. Validation

Cette section aborde deux aspects techniques de la validation du contenu d'éléments purs. Ces deux aspects sont la présence des caractères d'espacement [Section 2.2.2] et le déterminisme des contenus. Ils ont déjà été abordés pour les DTD [Section 3.6.1.1].

Lorsque le contenu d'un élément d'un élément pur est validé, les caractères d'espacement qui se trouvent hors de ses enfants sont ignorés. Ceci autorise l'indentation des documents XML sans perturber la validation.

Le contenu des éléments purs et mixtes doit être déterministe. Ceci signifie que la présence d'une balise ouvrante dans le contenu doit complètement déterminer d'où provient celle-ci dans la définition du type. Cette restriction est identique à celle portant sur les expressions définissant le contenu d'un élément pur dans une DTD. Le schéma suivant n'est pas valable car le contenu de l'élément `item` n'est pas déterministe. Une balise ouvrante `<item1>` peut provenir de la première ou de la seconde déclaration d'élément `item1`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="item">
    <xsd:complexType>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="item1" type="xsd:string"/>
          <xsd:element name="item2" type="xsd:string"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="item1" type="xsd:string"/>
          <xsd:element name="item3" type="xsd:string"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

5.7. Déclarations d'attributs

La déclaration d'un attribut est semblable à la déclaration d'un élément mais elle utilise l'élément `xsd:attribute` au lieu de l'élément `xsd:element`. Les attributs `name` et `type` de `xsd:attribute` spécifient respectivement le nom et le type de l'attribut. Le type d'un attribut est nécessairement un type simple puisque les attributs ne peuvent contenir que du texte. La déclaration d'un attribut prend la forme suivante.

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

L'exemple suivant déclare un attribut `format` de type `xsd:string`.

```
<xsd:attribute name="format" type="xsd:string"/>
```

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>
```

5.7.1. Localisation

Les déclarations d'attributs se placent normalement dans les définitions de types complexes qui peuvent être globaux ou locaux. Les types simples ne peuvent pas avoir d'attributs. La définition d'un type complexe se compose de la description du contenu suivie de la déclaration des attributs. L'ordre de déclarations des attributs est sans importance puisque l'ordre des attributs dans une balise n'est pas fixe.

Dans l'exemple suivant, le type complexe `List` déclare deux attributs `form` et `lang`. Les déclarations de ces deux attributs se situent après la description du contenu du type `List` constitué d'une suite d'éléments `item`. Le type de l'attribut `form` est le type prédéfini `xsd:string` alors que le type de l'attribut `lang` est le type global et simple `Lang` défini dans la suite du schéma.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="List">
    <!-- Contenu du type List -->
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="item" type="xsd:string"/>
    </xsd:sequence>
    <!-- Déclaration des attributs locaux form et lang du type List -->
    <xsd:attribute name="form" type="xsd:string"/>
    <xsd:attribute name="lang" type="Lang"/>
  </xsd:complexType>
  <!-- Type global et simple Lang pour l'attribut lang -->
  <xsd:simpleType name="Lang">
    <xsd:restriction base="xsd:string">
      <xsd:length value="2"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="list" type="List"/>
</xsd:schema>
```

Un attribut peut aussi être global lorsque sa déclaration est enfant direct de l'élément `xsd:schema`. Cet attribut peut alors être ajouté à différents types complexes. La définition du type utilise l'élément `xsd:attribute` avec un attribut `ref` qui remplace les deux attributs `name` et `type`. Cet attribut `ref` contient le nom de l'attribut global à ajouter. La déclaration globale d'un attribut est justifiée lorsque celui-ci a des occurrences multiple. Elle accroît la modularité en évitant de répéter la même déclaration dans plusieurs types.

Le schéma suivant déclare un attribut global `lang` de type `xsd:language`. Cet attribut est ajouté à deux reprises dans le type global `Texts` et dans le type anonyme de l'élément `text`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Déclaration de l'attribut global lang -->
  <xsd:attribute name="lang"  type="xsd:language"/>
  <xsd:element   name="texts" type="Texts"/>
  <xsd:complexType name="Texts">
    <xsd:sequence>
      <xsd:element name="text" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:string">
              <!-- Ajout de l'attribut lang au type anonyme -->
              <xsd:attribute ref="lang"/>
            </xsd:extension>
          </xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
```

```
    </xsd:sequence>
    <!-- Ajout de l'attribut lang au type Texts -->
    <xsd:attribute ref="lang"/>
  </xsd:complexType>
</xsd:schema>
```

Lorsqu'un schéma déclare un espace de noms cible [Section 5.13], les attributs globaux appartiennent automatiquement à cet espace de noms. Ceci signifie d'abord qu'ils doivent être référencés par leur nom qualifié dans le schéma. L'ajout d'un attribut de nom `name` à un type complexe prend alors la forme suivante où le préfixe `tns` est associé à l'espace de noms cible du schéma.

```
  <xsd:attribute ref="tns:name"/>
```

Cela signifie aussi qu'ils doivent avoir un nom qualifié dans les documents instance comme dans l'exemple suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<tns:texts tns:lang="fr" xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <text>Texte en français</text>
  <text tns:lang="en">Text in english</text>
</tns:texts>
```

5.7.2. Attribut optionnel, obligatoire ou interdit

Par défaut, un attribut est optionnel. Il peut être présent ou absent. Il peut aussi être rendu *obligatoire* ou *interdit* en donnant la valeur `required` ou `prohibited` à l'attribut `use` de l'élément `xsd:attribute`. L'attribut `use` peut aussi prendre la valeur `optional`. Cette valeur est très peu utilisée car c'est la valeur par défaut. La valeur `prohibited` est utile dans les restrictions de types pour modifier l'utilisation d'un attribut.

Les valeurs `optional` et `required` de l'attribut `use` sont donc équivalentes à `#IMPLIED` et `#REQUIRED` utilisés dans les déclarations d'attributs des DTD [Section 3.7]. Dans l'exemple suivant, les attributs `lang`, `xml:id` et `dummy` sont respecivement optionnel, obligatoire et interdit.

```
<xsd:attribute name="lang"   type="xsd:NMTOKEN" use="optional"/>
<xsd:attribute name="xml:id" type="xsd:ID"      use="required"/>
<xsd:attribute name="dummy"  type="xsd:string"  use="prohibited"/>
```

Le schéma suivant donne une utilisation réaliste de la valeur `prohibited` pour l'attribut `use`. Le type `Date` déclare un attribut `format` optionnel. Le type `Date-8601` est une restriction [Section 5.9] du type `Date`. L'attribut `format` devient interdit et ne peut plus apparaître.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Date">
    <xsd:simpleContent>
      <xsd:extension base="xsd:date">
        <!-- Attribut format optionnel dans le type Date -->
        <xsd:attribute name="format" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="Date-8601">
    <xsd:simpleContent>
      <xsd:restriction base="Date">
        <!-- Attribut format interdit dans le type Date-8601 -->
        <xsd:attribute name="format" type="xsd:string" use="prohibited"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:element name="date" type="Date-8601"/>
</xsd:schema>
```

5.7.3. Valeur par défaut et valeur fixe

Comme pour les éléments, il est possible de donner une valeur par défaut ou une valeur fixe à un attribut. La valeur de l'attribut `default` ou de l'attribut `fixed` de l'élément `xsd:attribute` permet de spécifier cette valeur. Il va de soi qu'une valeur par défaut n'est autorisée que si l'attribut est optionnel. Il est également interdit de donner simultanément une valeur par défaut et une valeur fixe. L'attribut `fixed` est équivalent à `#FIXED` utilisé dans les déclarations d'attribut des DTD. Dans le premier exemple suivant, la valeur par défaut de l'attribut `lang` est `fr` et dans le second exemple, sa valeur est fixée à la valeur `en`.

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

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

5.7.4. Joker `xsd:anyAttribute`

De même qu'il existe un élément `xsd:any` [Section 5.6.8] pour autoriser des éléments externes, il existe aussi un élément `xsd:anyAttribute` pour autoriser des attributs externes au schéma. Il possède également les attributs `processContents` et `namespace` pour contrôler la validation et l'espace de noms des attributs externes ajoutés dans les documents.

Dans le schéma suivant, l'élément `person` peut avoir des attributs appartenant à l'espace de noms identifié par l'URI `http://www.liafa.jussieu.fr/~carton/`.

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

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="utf-8"?>
<person tns:id="id42" xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <firstname>Victor</firstname>
  <lastname>Hugo</lastname>
</person>
```

5.8. Extension de types

L'*extension* est la première façon d'obtenir un type dérivé à partir d'un type de base. L'idée générale de l'extension est de rajouter du contenu et des attributs. Elle s'apparente à la dérivation de types des langages de programmation orientés objets comme Java ou C++. Les contenus du type dérivé ne sont généralement pas valides pour le type de base car des éléments et/ou des attributs nouveaux peuvent apparaître. L'extension s'applique aux types simples et aux types complexes mais elle donne toujours un type complexe.

L'extension d'un type est introduite par l'élément `xsd:extension` dont l'attribut `base` donne le nom du type de base. Celui-ci peut être un type prédéfini ou un type défini dans le schéma. Le contenu de l'élément `xsd:extension` explicite le contenu et les attributs à ajouter au type de base. L'élément `xsd:extension` est enfant d'un élément `xsd:simpleContent` ou `xsd:complexContent`, lui-même enfant de l'élément `xsd:complexType`.

5.8.1. Types simples

L'extension d'un type simple ne permet pas de changer le contenu mais permet uniquement d'ajouter des attributs pour donner un type complexe à contenu simple. C'est en fait la seule façon d'obtenir un tel type s'il on exclut l'extension ou la restriction d'un type qui est déjà dans cette catégorie. Lors d'une extension d'un type simple, l'élément `xsd:extension` est toujours enfant d'un élément `xsd:simpleContent`. Les déclarations des attributs qui sont ajoutés sont placées dans le contenu de l'élément `xsd:extension`.

Le fragment de schéma suivant définit un type `Price` qui étend le type prédéfini `xsd:decimal` en lui ajoutant un attribut `currency` de type `xsd:string`

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

Un fragment de document valide peut être le suivant.

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

5.8.2. Types complexes à contenu simple

Il est possible d'étendre un type complexe à contenu simple pour lui ajouter de nouveaux attributs. On obtient alors un nouveau type complexe à contenu simple qui possède les attributs du type de base et ceux déclarés par l'extension. L'extension d'un tel type est similaire à l'extension d'un type simple. L'élément `xsd:extension` est encore enfant d'un élément `xsd:simpleContent`. Les déclarations des attributs qui sont ajoutés sont placées dans le contenu de l'élément `xsd:extension`.

Dans le schéma suivant, le type `Price` est étendu en un type `LocalType` qui possède un nouvel attribut `country` de type `xsd:string`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base complexe à contenu simple -->
  <xsd:complexType name="Price">
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <!-- Attribut ajouté au type xsd:decimal -->
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <!-- Extension du type de base -->
  <xsd:complexType name="LocalPrice">
    <xsd:simpleContent>
      <xsd:extension base="Price">
        <!-- Attribut ajouté au type Price -->
        <xsd:attribute name="country" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:element name="price" type="LocalPrice"/>
</xsd:schema>
```

Un fragment de document valide peut être le suivant.

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

5.8.3. Types complexes à contenu complexe

L'extension d'un type complexe à contenu complexe consiste à ajouter du contenu et/ou des attributs. Le contenu est ajouté après le contenu du type de base. L'ajout d'attribut est semblabe au cas des types complexes à contenu simple.

Dans le schéma suivant le type `Fullname` étend le type `Name` en lui ajoutant un élément `title` et un attribut `id`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base -->
  <xsd:complexType name="Name">
    <xsd:sequence>
      <xsd:element name="firstname" type="xsd:string"/>
      <xsd:element name="lastname"  type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Extension du type de base -->
  <xsd:complexType name="Fullname">
    <xsd:complexContent>
      <xsd:extension base="Name">
        <xsd:sequence>
          <!-- Ajout de l'élément title après firstname et lastname -->
          <xsd:element name="title" type="xsd:string"/>
        </xsd:sequence>
        <!-- Ajout de l'attribut id -->
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  ...
</xsd:schema>
```

L'élément `title` est ajouté après le contenu du type `Name` qui est constitué des deux éléments `firstname` et `lastname`. Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<names>
  <fullname id="id40">
    <firstname>Alexander III Alexandrovich</firstname>
    <lastname>Romanov</lastname>
    <title>Tsar of Russia</title>
  </fullname>
  <fullname id="id52">
    <firstname>Elizabeth II Alexandra Mary</firstname>
    <lastname>Windsor</lastname>
    <title>Queen of England</title>
  </fullname>
</names>
```

5.9. Restriction de types

La *restriction* est la deuxième façon d'obtenir un type dérivé à partir d'un type de base. L'idée générale de la restriction est de définir un nouveau type dont les contenus au sens large sont des contenus du type de base. Par contenus au sens large, on entend les contenus proprement dits ainsi que les valeurs des attributs. La restriction s'applique aux types simples et aux types complexes mais elle prend des formes différentes suivant les cas.

La restriction d'un type est introduite par l'élément `xsd:restriction` dont l'attribut `base` donne le nom du type de base. Celui-ci peut être un type prédéfini ou un type défini dans le schéma. Le contenu de l'élément `xsd:restriction` explicite les restrictions au type de base. Dans le cas d'un type simple, l'élément `xsd:restriction` est enfant direct de l'élément `xsd:simpleType`. Dans le cas d'un type complexe, il est enfant d'un élément `xsd:simpleContent` ou `xsd:complexContent`, lui-même enfant de l'élément `xsd:complexType`.

5.9.1. Types simples

Les schémas définissent un certain nombre de types de base [Section 5.5.1]. Tous les autres types simples sont obtenus par restriction directe ou multiple de ces différents types de base. La restriction des types simples est effectuée par l'intermédiaire de *facettes* qui imposent des contraintes aux contenus. Toutes les facettes ne s'appliquent pas à tous les types simples. On donne d'abord quelques exemples de restrictions classiques à l'aide des principales facettes puis une liste exhaustive des facettes [Section 5.9.1.4].

5.9.1.1. Restriction par intervalle

Il est possible de restreindre les contenus en donnant une valeur minimale et/ou une valeur maximale avec les éléments `xsd:minInclusive`, `xsd:minExclusive`, `xsd:maxInclusive` et `xsd:maxExclusive`. Ces contraintes ne s'appliquent qu'aux types numériques pour lesquels elles ont un sens. 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>
```

La restriction par intervalle peut aussi s'appliquer aux dates et aux heures comme le montre l'exemple suivant.

```
<xsd:attribute name="date">
  <xsd:simpleType>
    <xsd:restriction base="xsd:date">
      <!-- Date après le 1er janvier 2001 exclus -->
      <xsd:minExclusive value="2001-01-01"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

5.9.1.2. Restriction par énumération

Il est possible de donner explicitement une liste des valeurs possibles d'un type prédéfini ou déjà défini avec l'élément `xsd:enumeration`. 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`.

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

5.9.1.3. Restriction par motif

Il est possible de restreindre les valeurs en donnant, avec l'élément `xsd:pattern`, une expression rationnelle [Section 5.15] qui décrit les valeurs possibles d'un type prédéfini ou déjà défini. Le contenu est valide s'il est conforme à l'expression rationnelle. Dans l'exemple suivant, le type `ISBN` décrit explicitement toutes 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 `Identifier` définit un type pour les noms XML [Section 2.2.3]. Il aurait aussi pu être décrit avec l'expression rationnelle `\i\c*`.

```
<xsd:simpleType name="Identifier">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[:_A-Za-z][-.:_0-9A-Za-z]*"/>
  </xsd:restriction>
</xsd:simpleType>
```

Pour que la contenu soit valide, il faut que le contenu, pris dans son intégralité, soit conforme à l'expression rationnelle. Il ne suffit pas qu'un fragment (une sous-chaîne) de celui-ci soit conforme. Le contenu `abc123xyz` n'est, par exemple, pas conforme à l'expression `\d{3}` bien que le fragment `123` le soit. Les ancres `'^'` et `'$'` [Section 6.3.4] sont implicitement ajoutées à l'expression. Pour avoir une expression qui accepte éventuellement un fragment du contenu, il suffit d'ajouter `.*` au début et à la fin de celle-ci. Le contenu `abc123xyz` est, par exemple, conforme à l'expression `.*\d^{3}.*`.

5.9.1.4. Liste des facettes

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.

La liste suivante décrit toutes les facettes. Pour chacune d'entre elles sont donnés les types sur lesquels elle peut s'appliquer.

`xsd:enumeration`

Cette facette permet d'énumérer explicitement les valeurs autorisées. Elle s'applique à tous les types simples y compris les types construits avec `xsd:union` [Section 5.6.5] et `xsd:list` [Section 5.6.6].

`xsd:pattern`

Cette facette permet de donner une expression rationnelle [Section 5.15] pour contraindre les valeurs. Elle ne s'applique pas uniquement aux types dérivés de `xsd:string` mais à tous les types simples y compris les types numériques et les types contruits avec `xsd:union` et `xsd:list`. L'utilisation avec `xsd:decimal` permet de restreindre, par exemple, aux nombres ayant 4 chiffres pour la partie entière et 2 pour la partie fractionnaire. Lorsque cette facette est appliquée à un type construit avec `xsd:list`, la contrainte porte sur les items de la liste et non sur la liste elle-même.

`xsd:length`, `xsd:minLength` et `xsd:maxLength`

Ces trois facettes donnent respectivement une longueur fixe ou des longueurs minimale et maximale. Elle s'appliquent aux types dérivés de `xsd:string` [Section 5.5.1.2] ainsi qu'aux types construits avec l'opérateur `xsd:list` [Section 5.6.6].

`xsd:minInclusive`, `xsd:minExclusive`, `xsd:maxInclusive` et `xsd:maxExclusive`

Ces quatre facettes donnent des valeurs minimale et maximale en incluant ou non la borne donnée. Ces facettes s'appliquent à tous les types numériques [Section 5.5.1.1] ainsi qu'à tous les types de date et d'heure [Section 5.5.1.3].

`xsd:fractionDigits` et `xsd:totalDigits`

Ces deux facettes fixent respectivement le nombre maximal de chiffres de la partie fractionnaire (à droite de la virgule) et le nombre maximal de chiffres en tout. Il s'agit de valeurs maximales. Il n'est pas possible de spécifier des valeurs minimales. De même, il n'est pas possible de spécifier le nombre maximal de chiffres de la partie entière (à gauche de la virgule). Ces deux facettes s'appliquent uniquement aux types numériques dérivés de `xsd:decimal`. Ceci inclut tous les types entiers mais exclut les types `xsd:float` et `xsd:double`.

`xsd:whiteSpace`

Cette facette est particulière. Elle ne restreint pas les valeurs valides mais elle modifie le traitement des caractères d'espacement [Section 2.2.2] à l'analyse lexicale. Cette facette peut prendre les trois valeurs `preserve`, `replace` et `collapse` qui correspondent à trois modes de fonctionnement de l'analyseur lexical.

`preserve`
: Dans ce mode, les caractères d'espacement sont laissés inchangés par l'analyseur lexical.

`replace`
: Dans ce mode, chaque caractère d'espacement est remplacé par un espace `#x20`. Le résultat est donc du type prédéfini `xsd:normalizedString` [Section 5.5.1.2].

`collapse`
: Dans ce mode, le traitement du mode précédent `replace` est d'abord appliqué puis les espaces en début et en fin sont supprimés et les suites d'espaces consécutifs sont remplacées par un seul espace. Le résultat est donc du type prédéfini `xsd:token` [Section 5.5.1.2]

Cette facette ne s'applique qu'aux types dérivés de `xsd:string`. Une dérivation ne peut que renforcer le traitement des caractères d'espacement en passant d'un mode à un mode plus strict (`preserve` → `replace` → `collapse`). Les changements dans l'autre sens sont impossibles.

5.9.2. Types complexes à contenu simple

Les types complexes à contenu simple sont toujours obtenus par extension d'un type simple en lui ajoutant des attributs. La restriction d'un de ces types peut porter sur le type simple du contenu ou/et sur les attributs. Il est possible de remplacer le type du contenu par un type obtenu par restriction. Il est aussi possible de changer le type d'un attribut ou de modifier son utilisation. Un attribut optionnel peut, par exemple, devenir obligatoire. La restriction d'un type complexe à contenu simple donne toujours un type complexe à contenu simple.

Par défaut, le nouveau type complexe défini est identique au type de base. Pour modifier le type du contenu, l'élément `xsd:restriction` contient un élément `xsd:simpleType` qui donne explicitement le nouveau type du contenu. Ce type doit être obtenu par restriction du type qui définit le contenu du type de base.

Dans le schéma suivant, un type `Base` est défini par extension du type simple `xsd:string` en lui ajoutant un attribut `format`. Le type `Derived` est ensuite obtenu en restreignant le type du contenu aux chaînes d'au plus 32 caractères.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base -->
  <xsd:complexType name="Base">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="format" type="xsd:string"/>
      </xsd:extension>
```

```
    </xsd:simpleContent>
  </xsd:complexType>
  <!-- Restriction du type de base -->
  <xsd:complexType name="Derived">
    <xsd:simpleContent>
      <xsd:restriction base="Base">
        <!-- Nouveau type pour le contenu du type Derived -->
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="32"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  ...
</xsd:schema>
```

La restriction peut aussi changer les types des attributs et leur utilisation. Les attributs dont certaines propriétés changent sont redéclarés dans le nouveau type. Les autres restent implicitement inchangés. Le type d'un attribut peut être remplacé par un type obtenu par restriction. Ce type peut, bien sûr, être nommé ou anonyme. L'utilisation des attributs peut aussi être restreinte. Un attribut optionnel peut devenir interdit avec `use="prohibited"` ou obligatoire avec `use="required"`. L'inverse est en revanche interdit. Il est également impossible d'ajouter de nouveaux attributs. Si un attribut possède une valeur par défaut ou une valeur fixe, celle-ci ne peut être ni modifiée ni supprimée.

Dans le schéma suivant, le type de base `Base` possède plusieurs attributs dont le type dérivé `Derived` modifie l'utilisation.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base -->
  <xsd:complexType name="Base">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="decimal"  type="xsd:decimal"/>
        <xsd:attribute name="string"   type="xsd:string"/>
        <xsd:attribute name="optional" type="xsd:string"/>
        <xsd:attribute name="required" type="xsd:string" use="required"/>
        <xsd:attribute name="fixed"    type="xsd:string" fixed="Fixed"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="Derived">
    <xsd:simpleContent>
      <xsd:restriction base="Base">
        <!-- Restriction du type de l'attribut -->
        <xsd:attribute name="decimal" type="xsd:integer"/>
        <!-- Le nouveau type doit être dérivé du type initial -->
        <xsd:attribute name="decimal" type="xsd:string"/>
        <!-- Restriction du type de l'attribut avec un type anonyme -->
        <xsd:attribute name="string">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="32"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <!-- Restriction de l'utilisation de l'attribut -->
```

```
        <xsd:attribute name="optional" type="xsd:string" use="required"/>
        <!-- Impossible d'étendre l'utilisation de l'attribut -->
        <xsd:attribute name="required" type="xsd:string"/>
        <!-- Impossible de changer ou supprimer la valeur fixe -->
        <xsd:attribute name="fixed"    type="xsd:string"/>
        <!-- Impossible d'ajouter un nouvel attribut -->
        <xsd:attribute name="newattr"  type="xsd:string"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  ...
</xsd:schema>
```

Il est encore possible de changer simultanément le type du contenu et certaines propriétés des attributs. Dans le schéma suivant, le type est restreint au chaînes d'au plus 32 caractères et le type de l'attribut `decimal` est changé en le type `xsd:integer`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base -->
  <xsd:complexType name="Base">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="decimal"   type="xsd:decimal"/>
        <xsd:attribute name="unchanged" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="Derived">
    <xsd:simpleContent>
      <xsd:restriction base="Base">
        <xsd:simpleType>
          <!-- Nouveau type pour le contenu du type Derived -->
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="32"/>
          </xsd:restriction>
        </xsd:simpleType>
        <!-- Restriction du type de l'attribut -->
        <xsd:attribute name="decimal"  type="xsd:integer"/>
        <!-- Attribut unchanged inchangé -->
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  ...
</xsd:schema>
```

5.9.3. Types complexes à contenu complexe

La restriction d'un type complexe permet d'imposer des contraintes aussi bien au contenu qu'aux attributs. La restriction doit rester fidèle au principe que tous les contenus possibles du type restreint doivent être valides pour le type de base. Il est, par exemple, possible de changer le type d'un élément en un type restreint ou de changer le nombre d'occurrences d'un éléments ou d'un bloc avec les attributs `minOccurs` et `maxOccurs` [Section 5.6.7]. Les restrictions portant sur les attributs sont identiques à celles possibles pour un type complexe à contenu simple.

Le nouveau type est défini en écrivant sa définition comme s'il s'agissait d'une première définition. Dans le schéma suivant, le type `Shortname` est obtenu par restriction du type `Name`. La valeur de l'attribut `maxOccurs` pour l'élément `firstname` passe de `unbounded` à `1`. L'attribut `id` devient obligatoire.

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

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="names">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="name"      type="Name"/>
        <xsd:element name="shortname" type="Shortname"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
  <!-- Type de base -->
  <xsd:complexType name="Name">
    <xsd:sequence>
      <!-- Nombre illimité d'occurrences de l'élément firstname -->
      <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>
  <!-- Restriction du type Name -->
  <xsd:complexType name="Shortname">
    <xsd:complexContent>
      <xsd:restriction base="Name">
        <xsd:sequence>
          <!-- Nombre limité d'occurrences de l'élément firstname -->
          <xsd:element name="firstname" type="xsd:string" maxOccurs="1"/>
          <xsd:element name="lastname"  type="xsd:string"/>
        </xsd:sequence>
        <!-- Attribut id obligatoire -->
        <xsd:attribute name="id" type="xsd:ID" use="required"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<names xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <name>
    <firstname>Elizabeth II</firstname>
    <firstname>Alexandra</firstname>
    <firstname>Mary</firstname>
    <lastname>Windsor</lastname>
  </name>
  <shortname id="id-42">
    <firstname>Bessiewallis</firstname>
    <lastname>Warfield</lastname>
  </shortname>
</names>
```

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>
```

Une restriction d'un type complexe à contenu complexe peut aussi supprimer un des choix possibles dans un élément `xsd:choice`. Dans l'exemple suivant, le choix `integer` a été supprimé dans le type `Float`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base -->
  <xsd:complexType name="Number">
    <xsd:choice>
      <xsd:element name="integer" type="xsd:integer"/>
      <xsd:element name="float"   type="xsd:float"/>
      <xsd:element name="double"  type="xsd:double"/>
    </xsd:choice>
  </xsd:complexType>
  <!-- Restriction du type de base -->
  <xsd:complexType name="Float">
    <xsd:complexContent>
      <xsd:restriction base="Number">
        <xsd:choice>
          <!-- Suppression de l'élément integer -->
          <xsd:element name="float"   type="xsd:float"/>
          <xsd:element name="double"  type="xsd:double"/>
        </xsd:choice>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
  ...
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent. Il utilise une substitution de type [Section 5.10.2] avec l'attribut `xsi:type` pour changer le type de l'élément `number` en `Float`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<numbers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <number>
    <integer>42</integer>
  </number>
  <number xsi:type="Float">
    <!-- Élément integer impossible -->
    <integer>42</integer>
    <float>3.14</float>
  </number>
</numbers>
```

5.10. Substitutions

Les schémas XML prévoient plusieurs mécanismes de substitution au niveau des types et des éléments. Dans la substitution de type, un document peut changer explicitement le type associé à un élément afin d'y placer un contenu différent de celui prévu par le schéma. La substitution d'éléments va encore plus loin. Un document peut remplacer un élément par un autre élément.

Les substitutions ne sont pas toujours possibles. Une première condition pour qu'elles puissent s'effectuer est que les types soient compatibles. Il faut que le type de substitution soit un type dérivé du type initial. Les substitutions sont donc étroitement liées aux différentes façons d'obtenir des types dérivés par extension ou restriction.

Une seconde condition pour rendre possible les substitutions est que celles-ci doivent être autorisées par le schéma. Les schémas possèdent différents outils pour contrôler les substitutions [Section 5.10.4].

5.10.1. Annihilation

L'*annihilation* est un mécanisme qui permet de mettre aucun contenu à un élément alors que le type de l'élément prévoit que le contenu est normalement non vide. Cette notion correspond à l'absence de valeur telle qu'elle peut apparaître dans les bases de données.

Le schéma doit d'abord autoriser le mécanisme en donnant la valeur `true` à l'attribut `nillable` de l'élément `xsd:element`. La valeur par défaut de cet attribut est `false`. Le document doit ensuite, explicitement, déclarer que l'élément n'a pas de valeur en donnant la valeur `true` à l'attribut `xsi:nil` qui est dans l'espace de noms des instances de schémas. Le contenu de l'élément doit alors être vide. Dans le schéma suivant, l'élément `item` est déclaré annihilable.

```
<?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="item" nillable="true"
                     maxOccurs="unbounded" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Le document suivant est valide pour ce schéma. L'élément `item` peut avoir un contenu vide si on lui ajoute un attribut `xsi:nil` avec la valeur `true`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<list xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Contenu normal -->
  <item>123</item>
  <!-- Contenu vide non autorisé -->
  <item></item>
  <!-- Contenu annihilé -->
  <item xsi:nil="true"></item>
  <!-- Contenu nécessairement vide -->
  <item xsi:nil="true">  </item>
  <item>789</item>
</list>
```

5.10.2. Substitution de type

Un type peut remplacer dans une instance de document un type dont il dérive directement ou non. Soit, par exemple, un élément `elem` déclaré d'un type `BaseType`. Si un type `ExtentedType` 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 `ExtentedType`. 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 `id`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema targetNamespace="http://www.liafa.jussieu.fr/~carton/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            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"?>
<tns:names xmlns:tns="http://www.liafa.jussieu.fr/~carton/"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Élément name avec le type tns:Name -->
  <tns:name>
    <firstname>Bessiewallis</firstname>
    <lastname>Warfield</lastname>
  </tns:name>
  <!-- Élément name avec le type tns:Fullname -->
  <tns:name id="id52" xsi:type="tns:Fullname">
    <firstname>Elizabeth II Alexandra Mary</firstname>
    <lastname>Windsor</lastname>
    <title>Queen of England</title>
  </tns:name>
</tns:names>
```

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 targetNamespace="http://www.liafa.jussieu.fr/~carton/"
            xmlns="http://www.liafa.jussieu.fr/~carton/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <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. Il est, en effet, nécessaire de déclarer trois espaces de noms : celui des éléments du document, celui des schémas pour le type `xsd:nonNegativeInteger` et celui des instances de schémas pour l'attribut `xsi:type`.

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

5.10.3. Groupes de substitution

Il est possible de spécifier qu'un élément peut être remplacé par un autre élément dans les documents instance. 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.

La substitution d'élément se distingue en plusieurs points de la substitution de type. Elle est évidemment beaucoup plus forte car elle affecte les éléments qui peuvent apparaître dans les documents. Pour cette raison, elle doit explicitement être prévue par le schéma par l'intermédiaire de groupes de substitution qui décrivent quel élément peut être remplacé et par quels éléments. En revanche, le document ne signale pas la substitution comme il le fait pour une substitution de type avec l'attribut `xsi:type`.

Ce mécanisme est mis en œuvre en créant un *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">
  <!-- 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>
```

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

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
```

```
<integers>
  <integer>0</integer>
  <!-- Élément positive se substituant à un élément integer -->
  <positive>1</positive>
  <!-- Élément negative se substituant à un élément integer -->
  <negative>-1</negative>
  <integer>-42</integer>
</integers>
```

Un élément donné est nécessairement le chef d'un unique groupe puisque chaque groupe est identifié par son chef. De même, un même élément ne peut appartenir qu'à au plus un groupe puisque l'attribut `substitutionGroup` de `xsd:element` ne peut contenir qu'un seul nom d'élément. La version 1.1 des schémas autorise l'attribut `substitutionGroup` à contenir plusieurs noms d'éléments. Un élément peut ainsi appartenir à plusieurs groupes de substitution.

Les substitutions peuvent être utilisées en cascade. Un élément membre d'un groupe de substitution peut lui-même être chef d'un autre groupe de substitution. Les membres de ce dernier groupe peuvent bien sûr remplacer leur chef de groupe mais aussi son chef de groupe. Dans le schéma suivant, l'élément `head` est le chef d'un groupe comprenant l'élément `subs`. Cet élément `subs` est, à son tour, chef d'un groupe de substitution comprenant l'élément `subsubs`. Cet élément `subsubs` peut dont remplacer l'élément `subs` mais aussi l'élément `head`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- L'élément chef de groupe -->
  <xsd:element name="head" type="xsd:string"/>
  <!-- Un élément subs pouvant se substituer à head -->
  <xsd:element name="subs" type="xsd:string" substitutionGroup="head"/>
  <!-- Un élément subsubs pouvant se substituer à subs et à head -->
  <xsd:element name="subsubs" type="xsd:string" substitutionGroup="subs"/>
  <xsd:element name="heads">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="head" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<heads>
  <head>Élément head original</head>
  <subs>Substitution de head par subs</subs>
  <subsubs>Substitution de head par subsubs</subsubs>
  <head>Autre élément head original</head>
</heads>
```

Les définitions circulaires de groupes de substitution sont interdites. Un chef de groupe ne peut pas être membre d'un groupe dont le chef serait lui même membre d'un de ses groupes. Les déclarations suivantes ne sont donc pas valides.

```
  <xsd:element name="head" type="xsd:string" substitutionGroup="subs"/>
  <xsd:element name="subs" type="xsd:string" substitutionGroup="head"/>
```

Les groupes de substitution permettent, quelquefois, de compenser les lacunes de l'opérateur `xsd:all` [Section 5.6.4]. Il est, en effet, possible de simuler un opérateur `xsd:choice` [Section 5.6.3] avec un élément abstrait et un groupe de substitution.

Dans le schéma suivant, l'élément `number` est abstrait. Il doit nécessairement être remplacé, dans un document, par un élément de son groupe de substitution, c'est-à-dire par un élément `integer` ou par un élément `double`. Il y a donc le choix entre un élément `integer` ou un élément `double`. Le type de l'élément `number` est `xsd:anyType` pour que les types `xsd:integer` et `xsd:double` des éléments `integer` et `double` puissent en dériver.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="properties">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="property" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="property">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="key" type="xsd:string"/>
        <xsd:element ref="number"/>
        <xsd:element name="condition" type="xsd:string" minOccurs="0"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="number"  type="xsd:anyType" abstract="true"/>
  <xsd:element name="integer" type="xsd:integer" substitutionGroup="number"/>
  <xsd:element name="double"  type="xsd:double"  substitutionGroup="number"/>
</xsd:schema>
```

5.10.4. Contrôle des substitutions et dérivations

Il existe différents moyens de contrôler les substitutions de types et d'éléments. Les types et éléments abstraits introduits par l'attribut `abstract` permettent de forcer une substitution en empêchant un type ou un élément d'apparaître dans un document. Les attributs `block` et `final` permettent, au contraire, de limiter les substitutions et les définitions de types dérivés.

Les trois attributs `abstract`, `block` et `final` s'appliquent aussi bien aux declarations d'éléments qu'aux définitions de types. Il faut prendre garde au fait que leurs significations dans ces deux cas sont proches mais néanmoins différentes.

Le tableau suivant récapitule les utilisations des trois attributs `abstract`, `block` et `final` pour les types et les éléments.

Attribut	Type	Élément
`abstract`	bloque l'utilisation du type dans les documents	bloque la présence de l'élément dans les documents
`block`	bloque la substitution du type dans les documents	bloque la substitution de type pour cet élément dans les documents
`final`	bloque la dérivation de types dans le schéma	bloque l'ajout d'éléments dans le groupe de substitution dans le schéma

5.10.4.1. Facette fixée

Les types simples sont obtenus par restrictions successives des types prédéfinis en utilisant des facettes [Section 5.9.1]. Il est possible d'imposer, avec l'attribut `fixed`, qu'une facette ne puisse plus être modifiée dans une restriction ultérieure.

L'attribut `fixed` peut être utilisé dans toutes les facettes `xsd:minLength`, `xsd:maxLength`, `xsd:minInclusive`, …. Sa valeur par défaut est la valeur `false`. Lorsqu'il prend la valeur `true`, la valeur de la facette est bloquée et elle ne peut plus être modifiée.

Dans le schéma suivant, le type `ShortString` est obtenu par restriction du type `xsd:string`. Il impose une longueur maximale à la chaîne avec la facette `xsd:maxLength`. Cette facette est fixée avec `fixed="true"`. Le type `VeryShortString` est obtenu par restriction du type `ShortString`. Il ne peut pas donner une nouvelle valeur à `xsd:maxLength`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base : restriction du type xsd:string -->
  <xsd:simpleType name="ShortString">
    <xsd:restriction base="xsd:string">
      <!-- Facette fixée -->
      <xsd:maxLength value="32" fixed="true"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- Restriction du type ShortString -->
  <xsd:simpleType name="VeryShortString">
    <xsd:restriction base="ShortString">
      <!-- Facette modifiée -->
      <xsd:minLength value="2"/>
      <!-- Facette impossible à modifier -->
      <xsd:maxLength value="16"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
...
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<strings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <string>Une chaîne assez courte</string>
  <string xsi:type="VeryShortString">Très courte</string>
</strings>
```

5.10.4.2. Type abstrait

Un type complexe peut être déclaré abstrait en donnant la valeur `true` à l'attribut `abstract` de l'élément `xsd:complexType`. Un type simple déclaré avec `xsd:simpleType` ne peut pas être abstrait.

Ce mécanisme est assez semblable à la notion de classe abstraite des langages de programmation orientés objet comme Java ou C++. Dans ces langages, un type déclaré abstrait peut être utilisé pour dériver d'autres types mais il ne peut pas être instancié. Ceci signifie qu'aucun objet de ce type ne peut être créé. Il est, en revanche, possible de créer des objets des types dérivés.

Lorsqu'un type est déclaré abstrait dans un schéma, celui-ci peut encore être utilisé dans la déclaration d'un élément. En revanche, l'élément ne pourra pas avoir ce type dans un document. Un document valide doit nécessairement opérer une substitution de type par l'intermédiaire de l'attribut `xsi:type` ou une substitution d'élément par l'intermédiaire d'un groupe de substitution.

Dans l'exemple suivant, on définit un type abstrait `Price` et un type dérivé `InternPrice`. L'élément `price` est du type `Price`. Il peut être substitué par l'élément `internprice` qui est de type `InternPrice`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base abstrait -->
  <xsd:complexType name="Price" abstract="true">
```

```
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <!-- Type dérivé concret -->
  <xsd:complexType name="InternPrice">
    <xsd:simpleContent>
      <xsd:restriction base="Price">
        <xsd:attribute name="currency" type="xsd:string" use="required"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  <!-- Élément price de type abstrait -->
  <xsd:element name="price" type="Price"/>
  <!-- Élément interprice de type concret substituable à price -->
  <xsd:element name="internprice" type="InternPrice"
                                  substitutionGroup="price"/>
  <xsd:element name="prices">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="price" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Le document ci-dessous est valide pour le schéma donné ci-dessus. L'élément `price` n'apparaît pas avec le type `Price` qui est abstrait. Soit le type `Price` est remplacé par le type dérivé `InternPrice`, soit l'élément `price` est remplacé par l'élément `internprice`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<prices xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Élément de type Price non autorisé -->
  <price>78.9</price>
  <!-- Substitution de type -->
  <price xsi:type="InternPrice" currency="euro">12.34</price>
  <!-- Substitution d'élément -->
  <internprice currency="dollar">45.56</internprice>
</prices>
```

Dans l'exemple suivant, on définit un type abstrait `AbstractType` sans contrainte. Ce type est alors équivalent au type `xsd:anyType`. On dérive ensuite deux types par extension `Derived1` et `Derived2`. Le premier type `Derived1` déclare un attribut `att` de type `xsd:string` et un élément `string` comme unique contenu. Le second type `Derived2` ne déclare aucun attribut mais il déclare un contenu constitué d'un élément `string` suivi d'un élément `integer`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema targetNamespace="http://www.liafa.jussieu.fr/~carton/"
            xmlns="http://www.liafa.jussieu.fr/~carton/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <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:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </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 `value` apparaît deux fois dans le document mais avec respectivement 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"?>
<tns:values xmlns:tns="http://www.liafa.jussieu.fr/~carton/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Élément value de type Abstract impossible -->
  <tns:value xsi:type="tns:Derived1" att="avec un attribut">
    <string>Une chaîne</string>
  </tns:value>
  <tns:value xsi:type="tns:Derived2">
    <string>Un entier</string>
    <integer>-1</integer>
  </tns:value>
</tns:values>
```

5.10.4.3. Élément abstrait

Un élément peut être déclaré *abstrait* en donnant la valeur `true` à l'attribut `abstract` de l'élément `xsd:element`. Un élément déclaré abstrait peut être utilisé dans la construction d'un type pour un autre élément. En revanche, il ne peut pas apparaître dans un document instance. L'élément doit nécessairement être remplacé par un autre élément. Cette substitution est uniquement possible lorsque l'élément abstrait est le chef d'un groupe de substitution. Il peut alors être remplacé par n'importe quel membre du groupe.

La contrainte imposée en rendant un élément abstrait est plus forte que celle imposée en rendant un type abstrait. Il n'est en effet plus possible de remplacer le type. Il faut nécessairement remplacer l'élément.

Dans le schéma suivant, l'élément `value` est déclaré abstrait. Il est le chef d'un groupe qui comprend uniquement l'élément `other` qui peut donc le remplacer.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Un élément value abstrait -->
  <xsd:element name="value" type="xsd:string" abstract="true"/>
```

```
  <!-- Un élément other pouvant se substituer à value -->
  <xsd:element name="other" type="String27" substitutionGroup="value"/>
  <!-- Type obtenu par restriction de xsd:string -->
  <xsd:simpleType name="String27">
    <xsd:restriction base="xsd:string">
      <xsd:length value="27"/>
    </xsd:restriction>
  </xsd:simpleType>
  ...
```

Le document suivant est valide pour le schéma précédent. L'élément abstrait `value` n'apparaît pas. Il est systématiquement remplacé par l'élément `other`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<values xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- Élément value impossible -->
  <value>Une chaîne d'une autre longueur</value>
  <!-- Élément value impossible même avec une substitution de type -->
  <value xsi:type="String27">Une chaîne de 27 caractères</value>
  <!-- Substitution d'élément -->
  <other>Une chaîne de même longueur</other>
</values>
```

5.10.4.4. Type bloqué

Il est possible, dans un schéma de limiter dans les documents les 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` et `restriction`. 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 `xsd:schema`.

Lorsque `restriction` apparaît, par exemple, dans la valeur de l'attribut `block` de la définition d'un type complexe, celui-ci ne peut pas être remplacé dans un document par un type obtenu par restriction. Cette contrainte s'applique aux substitutions de types et d'éléments. Il n'est pas possible de changer le type d'un élément avec l'attribut `xsi:type`. Il n'est pas possible non plus de substituer l'élément par un autre élément dont le type est obtenu par restriction.

Dans le schéma suivant, les types `Extension` et `Restriction` sont respectivement obtenus par extension et restriction du type `Base`. La définition de ce type `Base` contient `block="#all"`. Ceci impose que l'élément `value` ne peut pas changer son type en le type `Restriction` ou `Restriction` avec l'attribut `xsi:type`. L'élément `subs` ne peut pas se substituer à l'élément `value` car son type est `Extension`. En revanche, l'élément `sametype` peut se substituer à l'élément `value` car son type est `Base`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="value" type="Base"/>
  <!-- Élément du même type dans le groupe de substitution -->
  <xsd:element name="sametype" type="Base"   substitutionGroup="value"/>
  <!-- Élément d'un type dérivé dans le groupe de substitution -->
  <xsd:element name="subst" type="Extension" substitutionGroup="value"/>
  ...
  <!-- Type de base ne pouvant pas être substitué dans les documents -->
  <xsd:complexType name="Base" block="#all">
    <xsd:sequence>
      <xsd:element name="integer" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Type obtenu par extension du type de base -->
```

```
  <xsd:complexType name="Extension">
    <xsd:complexContent>
      <xsd:extension base="Base">
        <xsd:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- Type obtenu par restriction du type de base -->
  <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 est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<values xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <value>
    <integer>-1</integer>
  </value>
  <!-- Substitution autorisée avec le même type -->
  <sametype>
    <integer>-1</integer>
  </sametype>
  <!-- Élément substitué d'un type dérivé impossible -->
  <subst att="Un attribut">
    <integer>-1</integer>
  </subst>
  <!-- Élément value de type Extension impossible -->
  <value xsi:type="Extension" att="Un attribut">
    <integer>1</integer>
  </value>
  <!-- Élément value de type Restriction impossible -->
  <value xsi:type="Restriction">
    <integer>1</integer>
  </value>
</values>
```

L'attribut `block` bloque la substitution par un type dont une des étapes de dérivation et pas seulement la première étape est mentionnée dans ses valeurs. Si, par exemple, l'attribut `block` d'une définition de type contient `extension`, seuls les types obtenus par restrictions successives de ce type peuvent le remplacer.

Dans le schéma suivant, le type `List` bloque sa substitution par un type obtenu par extension. Le type `ShortList` est obtenu par restriction du type `List` et le type `AttrShortList` est obtenu par extension du type `ShortList`. Le type `ShortList` peut se substituer au type `List`. Au contraire, le type `AttrShortList` ne peut pas se substituer au type `List` car il y a une dérivation par extension entre le type `List` et le type `AttrShortList`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base ne pouvant pas être substitué par une extension -->
  <xsd:complexType name="List" block="extension">
    <xsd:sequence>
```

```
      <xsd:element name="item" type="xsd:string" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Restriction du type de base -->
  <xsd:complexType name="ShortList">
    <xsd:complexContent>
      <xsd:restriction base="List">
        <xsd:sequence>
          <!-- Nombre limité d'éléments item -->
          <xsd:element name="item" type="xsd:string" maxOccurs="8"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- Extension de la restriction du type de base -->
  <xsd:complexType name="AttrShortList">
    <xsd:complexContent>
      <xsd:extension base="ShortList">
        <!-- Ajout d'un attribut -->
        <xsd:attribute name="length" type="xsd:integer"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="list" type="List"/>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<!-- Type ShortList possible mais type AttrShortList impossible -->
<list xsi:type="ShortList"
      xsi:type="AttrShortList" length="3"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <item>Premier item</item>
  <item>Deuxième item</item>
  <item>Troisième item</item>
</list>
```

5.10.4.5. Élément bloqué

L'attribut `block` peut aussi apparaître dans la déclaration d'un élément. L'attribut `block` de l'élément `xsd:element` permet d'empêcher que cet élément 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 `xsd:schema`.

Dans le schéma suivant, l'élément `integer` bloque toutes les substitutions de types dans les documents avec `block="restriction  extension"`. Ce blocage empêche de changer le type en un type dérivé avec l'attribut `xsi:type`. Il empêche également l'élément `positive` de se substituer à l'élément `integer` car son type est obtenu par restriction du type `xsd:integer`. En rechanche, l'élément `sametype` dont le type est aussi `xsd:integer` peut rempalcer l'élément `integer`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="integer"  type="xsd:integer"
                               block="restriction extension"/>
  <!-- Élément avec le même type -->
  <xsd:element name="sametype" type="xsd:integer"
```

```
                                substitutionGroup="integer"/>
  <!-- Élément avec un type obtenu par restriction -->
  <xsd:element name="positive" type="xsd:positiveInteger"
                                substitutionGroup="integer"/>
  ...
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<integers xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <integer>0</integer>
  <!-- Substitution par un élément de même type -->
  <sametype>1</sametype>
  <!-- Substitution de type impossible -->
  <integer xsi:type="xsd:positiveInteger">1</integer>
  <!-- Substitution d'élément avec un type dérivé impossible -->
  <positive>1</positive>
</integers>
```

L'attribut `block` de l'élément `xsd:element` peut aussi contenir la valeur `substitution`. Cette valeur a un effet très proche de l'attribut `final` avec la valeur `#all`. Elle empêche les éléments du groupe de substitution de se substituer dans les documents instance. Cela anihile l'intéret d'avoir des éléments dans le groupe de substitution puisque ceux-ci ne peuvent pas réellement se substituer à leur chef de groupe.

Dans le schéma suivant, les éléments `sametype` et `positive` appartiennent au groupe de substitution de l'élément `integer`. En rechanche, ils ne peuvent pas se substituer à cet élément en raison de la valeur `substitution` de l'attribut `block`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="integer"  type="xsd:integer" block="substitution"/>
  <!-- Élément avec le même type -->
  <xsd:element name="sametype" type="xsd:integer"
                                substitutionGroup="integer"/>
  <!-- Élément avec un type obtenu par restriction -->
  <xsd:element name="positive" type="xsd:positiveInteger"
                                substitutionGroup="integer"/>
  ...
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<integers xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <integer>0</integer>
  <!-- Substitution de type -->
  <integer xsi:type="xsd:positiveInteger">1</integer>
  <!-- Substitution d'élément avec le même type impossible -->
  <sametype>1</sametype>
  <!-- Substitution d'élément avec un type dérivé impossible -->
  <positive>1</positive>
</integers>
```

5.10.4.6. Type final

Il est possible, dans un schéma, de restreindre l'utilisation d'un type pour définir d'autres types. Ce mécanisme s'apparente à la possiblité des langages de programmation orientés objet de bloquer la dérivation d'une classe

avec le qualificatif `final`. Le mécanisme des schémas est plus précis car il permet de bloquer sélectivement les différentes dérivations : restriction, extension, union et liste.

L'attribut `final` des éléments `xsd:simpleType` et `xsd:complexType` permet d'empêcher que le type défini puisse servir de type de base à des constructions ou à des dérivations de types. Pour un type simple, la valeur de cet attribut est soit la chaîne `#all` soit une liste de valeurs parmi les valeurs `restriction`, `list` et `union`. Il est donc impossible de bloquer les extensions d'un type simple. Pour un type complexe, la valeur de cet attribut est soit la chaîne `#all` soit une liste de valeurs parmi les valeurs `extension`, `restriction`. 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 `xsd:schema`.

Le schéma suivant n'est pas correct car les définitions des types `Extension` et `Restriction` sont impossibles en raison de la valeur `#all` de l'attribut `final` dans la définition du type `Base`. Si la valeur de cet attribut `final` est changée en `restriction`, la définition du type `Restriction` reste incorrecte mais la définition du type `Extension` devient correcte.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
  <!-- L'attribut final="#all" empêche les restrictions et extensions -->
  <xsd:complexType name="Base" final="#all">
    <xsd:sequence>
      <xsd:element name="integer" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Extension du type Base impossible -->
  <xsd:complexType name="Extension">
    <xsd:complexContent>
      <xsd:extension base="Base">
        <xsd:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- Restriction du type Base impossible -->
  <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 bloquage imposé par l'attribut `final` n'opère que sur la dérivation directe de types. Dans le schéma suivant, le type `List` bloque les extensions avec `final="extension"`. Le type `ShortList` est dérivé par restriction du type `List`. Ce type peut être étendu en un type `AttrShortList`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Type de base bloquant les extensions -->
  <xsd:complexType name="List" final="extension">
    <xsd:sequence>
      <xsd:element name="item" type="xsd:string" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Restriction du type de base -->
  <xsd:complexType name="ShortList">
```

```
    <xsd:complexContent>
      <xsd:restriction base="List">
        <xsd:sequence>
          <!-- Nombre limité d'éléments item -->
          <xsd:element name="item" type="xsd:string" maxOccurs="8"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
  <!-- Extension de la restriction du type de base -->
  <xsd:complexType name="AttrShortList">
    <xsd:complexContent>
      <xsd:extension base="ShortList">
        <!-- Ajout d'un attribut -->
        <xsd:attribute name="length" type="xsd:integer"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="list" type="List"/>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<list length="3" xsi:type="AttrShortList"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <item>Premier item</item>
  <item>Deuxième item</item>
  <item>Troisième item</item>
</list>
```

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

5.10.4.7. Élément final

Il est possible, dans un schéma, de limiter les éléments susceptibles de se substituer à un élément donné. Il est en effet possible d'empêcher sélectivement les éléments d'appartenir à un groupe de substitution en fonction de leur type.

L'attribut `final` de l'élément `xsd:element` permet de sélectioner quels éléments peuvent appartenir au groupe de substitution de l'élément. La valeur de cet attribut est soit la chaîne `#all` soit une liste de valeurs parmi les valeurs `restriction` et `extension`. Les valeurs énumérées ou toutes pour `#all` bloquent les éléments dont le type est obtenu par la dérivation correspondante.

Dans le schéma suivant, l'élément `integer` empêche les éléments dont le type est dérivé par extension de son type `xsd:integer` d'appartenir à son groupe de substitution. Comme le type `xsd:positiveInteger` est obtenu par restriction du type `xsd:integer`, l'élément `positive` peut appartenir au groupe de substitution de `integer`. En revanche, l'élément `attributed` ne pourrait pas appartenir à ce groupe de substitution car son type `Attributed` est obtenu par extension du type `xsd:integer`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="integer"  type="xsd:integer" final="extension"/>
  <!-- Élément avec un type obtenu par restriction de xsd:integer -->
  <xsd:element name="positive" type="xsd:positiveInteger"
                               substitutionGroup="integer"/>
  <!-- Élément avec un type obtenu par extension de xsd:integer -->
  <!-- Impossible dans le groupe de substitution de integer -->
```

```
  <xsd:element name="attributed" type="Attributed"
                                  substitutionGroup="integer"/>
  <!-- Type obtenu par extension de xsd:integer -->
  <xsd:complexType name="Attributed">
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  ...
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent. L'élément `attributed` ne peut pas se substituter à l'élément `integer`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<integers>
  <integer>0</integer>
  <!-- Substitution élément -->
  <positive>1</positive>
  <!-- Élément attributed impossible -->
  <attributed att="Un attribut">-1</attributed>
</integers>
```

5.11. 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 des schémas. Les groupes d'éléments et d'attributs sont respectivement définis par les éléments `xsd:group` et `xsd:attributeGroup`.

Les groupes d'éléments ne doivent pas être confondus avec les groupes de substitution [Section 5.10.3] qui permettent de remplacer un élément par un autre.

5.11.1. Groupe d'éléments

L'élément `xsd:group` permet de définir un groupe d'éléments dont le nom est donné par l'attribut `name`. L'élément `xsd:group` doit être enfant de l'élément racine `xsd:schema` du schéma. Ceci signifie que la portée de la définition du groupe est le schéma dans son intégralité. Le contenu de l'élément `xsd:group` est un fragment de type nécessairement inclus dans un élément `xsd:sequence`, `xsd:choice` ou `xsd:all`.

Un groupe peut être employé dans la définition d'un type ou la définition d'un autre groupe. L'utilisation d'un groupe est équivalente à l'insertion de son contenu. L'intérêt d'un groupe est de pouvoir l'utiliser à plusieurs reprises et de factoriser ainsi les parties communes à plusieurs types. L'utilisation d'un groupe est introduite par un élément `xsd:group` avec un attribut `ref` qui donne le nom du groupe à insérer.

Dans le schéma suivant, le groupe `FirstLast` est défini puis utilisé dans la définition du groupe `Name` et du type `Person` ainsi que dans la définition du type anonyme de l'élément `character`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:group name="FirstLast">
    <xsd:sequence>
      <xsd:element name="firstname" type="xsd:string"/>
      <xsd:element name="lastname"  type="xsd:string"/>
    </xsd:sequence>
  </xsd:group>
  <xsd:group name="Name">
```

```
    <xsd:choice>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:group   ref="FirstLast"/>
    </xsd:choice>
  </xsd:group>
  <xsd:complexType name="Person">
    <xsd:sequence>
      <xsd:element name="surname" type="xsd:string"  minOccurs="0"/>
      <xsd:group   ref="Name"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="characters">
    <xsd:complexType>
       <xsd:sequence>
  <xsd:element name="character" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group   ref="Name"/>
        <xsd:element name="creator" type="Person"
       minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
       </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<characters>
  <character>
    <firstname>Gaston</firstname>
    <lastname>Lagaffe</lastname>
    <creator>
      <firstname>André</firstname>
      <lastname>Franquin</lastname>
    </creator>
  </character>
  <character>
    <name>Astérix</name>
    <creator>
      <surname>Al Uderzo</surname>
      <firstname>Albert</firstname>
      <lastname>Uderzo</lastname>
    </creator>
    <creator>
      <firstname>René</firstname>
      <lastname>Goscinny</lastname>
    </creator>
  </character>
```

Un groupe est en fait un *fragment* de type qui peut être utilisé à l'intérieur de la définition de n'importe quel type. En revanche, il ne peut pas servir comme type dans la déclaration d'un l'élément. À l'inverse, un type peut servir dans la déclaration d'éléments mais il ne peut pas être directement inclus par un autre type.

Les groupes sont en fait un mécanisme d'abréviation. Ils permettent d'accroître la modularité des schémas en évitant de recopier plusieurs fois le même fragment dans la définition de différents types.

5.11.2. Groupe d'attributs

Les groupes d'attributs jouent, pour les attributs, un rôle similaire aux groupes d'éléments. Ils permettent de regrouper plusieurs déclarations d'attributs dans le but d'une réutilisation. L'élément `xsd:attributeGroup` permet de définir un groupe d'attributs dont le nom est donné par l'attribut `name`. L'élément `xsd:attributeGroup` doit être enfant de l'élément racine `xsd:schema` du schéma. Ceci signifie que la portée de la définition du groupe est le schéma dans son intégralité. Le contenu de l'élément `xsd:attributeGroup` est constitué de déclarations d'attributs introduites par l'élément `xsd:attribute`.

Dans l'exemple suivant, le groupe d'attributs `LangType` regroupe les déclaration des deux attributs `lang` et `type`.

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

Un groupe d'attributs peut être employé dans la définition d'un type ou la définition d'un autre groupe d'attributs. L'utilisation d'un groupe est équivalente à l'insertion de son contenu. L'utilisation d'un groupe est introduite par un élément `xsd:attributeGroup` avec un attribut `ref` qui donne le nom du groupe à insérer.

Le groupe d'attributs `LangType` peut être employé de la façon suivante dans la définition d'un type complexe nommé ou anonyme. Tout élément du type `SomeType` défini ci-dessous pourra avoir les attributs `lang` et `type` déclarés dans le groupe `LangType`.

```
<xsd:complexType name="SomeType">
  <!-- Contenu -->
  ...
  <xsd:attributeGroup ref="LangType"/>
</xsd:complexType>
```

Il est possible d'utiliser successivement plusieurs groupes d'attributs pour déclarer les attributs d'un type mais il faut une occurrence de `xsd:attributeGroup` pour chaque groupe utilisé.

```
<xsd:attributeGroup ref="AttrGroup1"/>
<xsd:attributeGroup ref="AttrGroup2"/>
```

Un groupe d'attributs peut aussi être utilisé dans la définition d'un autre groupe d'attributs. Le nouveau groupe défini contient tous les attributs du ou des groupes référencés en plus des attributs qu'il déclare explicitement. Ce mécanisme est semblable à l'héritage des classes dans les langages de programmation objet.

```
<xsd:attributeGroup name="LangTypeClass">
  <xsd:attributeGroup ref="LangType"/>
  <xsd:attribute name="class" type="xsd:string"/>
</xsd:attributeGroup>
```

Le schéma à l'adresse `http://www.w3.org/2001/xml.xsd` [Section 5.14] déclare les quatre attributs particuliers [Section 2.7.4] `xml:lang`, `xml:space`, `xml:base` et `xml:id`. Il définit également un groupe d'attributs `xml:specialAttrs` permettant de déclarer simultanément ces quatre attributs. Cet exemple montre que les noms des groupes d'éléments et des groupes d'attributs sont des noms qualifiés dans l'espace de noms cible du schéma.

5.12. Contraintes de cohérence

Les schémas permettent de spécifier des contraintes globales de cohérence. Celles-ci doivent être vérifiées par un document pour que celui-ci soit valide. Elles ressemblent aux contraintes des DTD portant sur les attributs des

types `ID`, `IDREF` et `IDREFS` [Section 3.7.2] mais elles sont beaucoup plus générales. Elle peuvent porter sur des éléments ou des attributs. La portée de ces contraintes peut être n'importe quel contenu d'élément et non pas l'intégralité du document comme dans les DTD.

Ces contraintes sont de deux types. Elles peuvent être des contraintes d'*unicité* comme celle des attributs de type `ID` des DTD ou des contraintes d'*existence* comme celle des attributs de type `IDREF` et `IDREFS` des DTD. Les contraintes utilisent des expressions XPath [Chapitre 6] mais une connaissance superficielle de ce langage suffit pour les utiliser.

5.12.1. Contraintes d'unicité

Une contrainte d'*unicité* spécifie que dans le contenu d'un élément donné, il ne peut exister qu'un seul élément ayant une propriété fixée. Cette propriété est très souvent la valeur d'un attribut mais elle peut aussi être formée des valeurs de plusieurs enfants ou attributs. Cette notion est similaire à la notion de *clé* des bases de données. Elle généralise les attributs de types `ID` dont la valeur est unique dans tout le document.

Une contrainte d'unicité est donnée par un élément `xsd:key` ou `xsd:unique`. Les contraintes introduites par ces deux éléments se présentent de la même façon et ont des sémantiques très proches. L'élément `xsd:key` ou `xsd:unique` doit être enfant d'un élément `xsd:element` qui déclare un élément. Cet élément qui contient la contrainte définit la *portée* de celle-ci. Les contraintes d'unicité ainsi que les contraintes d'existence doivent être placées après le type de la déclaration.

Chaque élément `xsd:key` ou `xsd:unique` possède un attribut `name` uniquement utilisé par les contraintes d'existence introduites par `xsd:keyref` et qui peut donc être ignoré pour l'instant. Il contient un élément `xsd:selector` et des éléments `xsd:field` possédant chacun un attribut `xpath`. L'élément `xsd:selector` détermine sur quels éléments porte la contrainte. La valeur de son attribut `xpath` est une expression XPath qui sélectionne des éléments concernés. Les éléments `xsd:field` déterminent quelle est la valeur qui doit être unique. Cette valeur est constituée de plusieurs *champs* à la manière d'un objet dans les langages de programmation. La valeur de l'attribut `xpath` de chacun des éléments `xsd:selector` spécifie un champ de la valeur de la clé d'unicité. La contrainte donnée par un élément `xsd:key` impose que chacun des champs déterminé par les éléments `xsd:field` soit présent et que la valeur ainsi constituée soit unique pour les éléments sélectionnés par `xsd:selector` dans le contenu de l'élément définissant la portée. Au contraire, la contrainte donnée par un élément `xsd:key` n'impose pas que chacun des champs déterminé par les éléments `xsd:field` soit présent. Elle impose seulement que les éléments ayant tous les champs aient une valeur 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`.

```
<!-- Déclaration de l'élément bibliography de type Bibliography -->
<xsd:element name="bibliography" type="Bibliography">
  <!-- Unicité des attributs key des éléments book dans 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 [Section 5.10.1]. Une contrainte décrite avec `xsd:unique` est au contraire seulement vérifiée pour les éléments dont tous les champs spécifiés dans la contrainte sont présents.

5.12.1.1. Portée des contraintes

Le schéma suivant illustre la notion de portée. Il contient deux exemples de contrainte d'unicité. Une première contrainte `group.num` porte sur les attributs `num` des éléments `group`. Cette contrainte est déclarée dans l'élément `groups` qui est l'élément racine du document ci-dessous. Deux éléments `group` du document ne peuvent pas avoir la même valeur d'attribut `num`. La seconde contrainte `person.id` porte sur les éléments `person` contenus dans un élément `group`. Comme cette contrainte est déclarée dans l'élément `group`, deux éléments `person` contenus dans le même élément `group` ne peuvent pas avoir la même valeur d'attribut `id`. En

revanche, deux éléments `person` contenus dans des éléments `group` différents peuvent avoir la même valeur d'attribut `id`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="groups">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="group" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
    <!-- Unicité des attributs num des éléments group -->
    <xsd:unique name="group.num">
      <xsd:selector xpath="group"/>
      <xsd:field    xpath="@num"/>
    </xsd:unique>
  </xsd:element>
  <xsd:element name="group">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="person" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="firstname" type="xsd:string"/>
              <xsd:element name="lastname"  type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:string"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="num" type="xsd:integer"/>
    </xsd:complexType>
    <!-- Unicité des attributs id des éléments person -->
    <xsd:key name="person.id">
      <xsd:selector xpath="person"/>
      <xsd:field    xpath="@id"/>
    </xsd:key>
  </xsd:element>
</xsd:schema>
```

Le document suivant est valide pour le schéma précédent. Deux éléments `person` contenus respectivement dans le premier et le deuxième élément `group` ont la même valeur `AC` pour l'attribut `id`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<groups>
  <group num="1">
    <person id="AC">
      <firstname>Albert</firstname>
      <lastname>Cohen</lastname>
    </person>
    <person id="VH">
      <firstname>Victor</firstname>
      <lastname>Hugo</lastname>
    </person>
  </group>
  <group num="2">
    <person id="AC">
      <firstname>Anders</firstname>
      <lastname>Celsius</lastname>
```

```
    </person>
    <person id="SH">
      <firstname>Stephen</firstname>
      <lastname>Hawking</lastname>
    </person>
  </group>
</groups>
```

5.12.1.2. Valeurs à champs multiples

La valeur qui détermine l'unicité peut être constituée de plusieurs champs. Il suffit pour cela de mettre plusieurs éléments `xsd:field` dans l'élément `xsd:key` ou `xsd:unique`. Deux valeurs sont alors considérées comme différentes si elles diffèrent en au moins un champ.

La contrainte `person.names` ci-dessous peut remplacer la contrainte `person.id` du schéma précédent. Elle impose alors que la valeur formée des contenus des deux éléments `fitstname` et `lastname` soit différente pour chacun des éléments `person`. Deux éléments `person` contenus dans un même élément `group` peuvent avoir le même contenu textuel pour l'élément `firstname` ou pour l'élément `lastname` mais pas pour les deux en même temps.

```
    <xsd:key name="person.names">
      <xsd:selector xpath="person"/>
      <xsd:field    xpath="firstname"/>
      <xsd:field    xpath="lastname"/>
    </xsd:key>
```

La contrainte ci-dessus illustre aussi que la valeur peut aussi être donnée par des éléments et pas seulement par des attributs. Le document suivant vérifie la contrainte ci-dessus bien que deux éléments `person` dans le même élément `group` aient la même valeur `Albert` pour l'élément `firstname`. Deux éléments ayant exactement la même valeur pour l'attribut `id` sont aussi dans le même élément `group` mais la contrainte ne porte plus sur cet attribut.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<groups>
  <group num="1">
    <person id="AC">
      <firstname>Albert</firstname>
      <lastname>Cohen</lastname>
    </person>
    <person id="VH">
      <firstname>Victor</firstname>
      <lastname>Hugo</lastname>
    </person>
    <person id="AC">
      <firstname>Anders</firstname>
      <lastname>Celsius</lastname>
    </person>
    <person id="AE">
      <firstname>Albert</firstname>
      <lastname>Einstein</lastname>
    </person>
  </group>
</groups>
```

Il est bien sûr possible de mettre plusieurs contraintes dans un même élément. Les deux contraintes `person.id` et `person.names` pourraient être mises simultanément dans l'élément `group` comme ci-dessous.

```
    <xsd:key name="person.id">
      <xsd:selector xpath="person"/>
```

```
  <xsd:field    xpath="@id"/>
</xsd:key>
<xsd:key name="person.names">
  <xsd:selector xpath="person"/>
  <xsd:field    xpath="firstname"/>
  <xsd:field    xpath="lastname"/>
</xsd:key>
```

5.12.1.3. Différence entre `xsd:key` et `xsd:unique`

Le schéma précédent illustre également la différence entre les contraintes introduites par les éléments `xsd:key` et `xsd:unique`. Une contrainte introduite par `xsd:key` impose que tous les champs de la valeur soient présents. La contrainte `person.id` impose donc que l'attribut `id` soit présent dans chaque élément `person` même si cet attribut est déclaré optionnel [Section 5.7.2]. Au contraire, une contrainte introduite par `xsd:unique` n'impose pas que tous les champs de la valeurs soient présents. Seuls les éléments ayant tous les champs sont pris en compte dans la vérification de la contrainte. Deux éléments ayant tous les champs ne peuvent avoir tous les champs égaux.

5.12.1.4. Expressions XPath

Les valeurs des attributs `xpath` des éléments `xsd:selector` et `xsd:field` sont des expressions XPath restreintes [Chapitre 6]. L'expression XPath de `xsd:selector` est relative à l'élément dont la déclaration contient l'élément `xsd:unique` ou `xsd:key`. Elle sélectionne uniquement des éléments descendants de cet élément. L'expression XPath de `xsd:field` est relative aux éléments sélectionnés par `xsd:selector`. Elle sélectionne uniquement des éléments ou des attributs descendants de ces éléments.

Les seuls opérateurs autorisés dans les expressions XPath des attributs `xpath` de `xsd:selector` et `xsd:field` sont l'opérateur d'union `'|'` [Section 6.2.4] et l'opérateur de composition de chemins `'/'` [Section 6.2.3]. L'opérateur `'|'` peut apparaître au niveau global mais pas à l'intérieur d'une expression de chemins avec l'opérateur `'/'`. Les seuls axes [Section 6.2.1.1] autorisés dans ces expressions XPath sont les axes `child::` et `attribute::` dans leurs syntaxes abrégées [Section 6.7] `' '` et `'@'`. L'axe `descendant::` peut, en outre, apparaître au début des expressions de chemins dans sa syntaxe abrégée `'.//'`. Les filtres ne sont pas permis dans ces expressions. La contrainte suivante impose, par exemple, que tous les enfants ainsi que tous les enfants de ses enfants `group` aient des valeurs d'attribut `id` différentes.

```
<xsd:unique name="all.id">
  <xsd:selector xpath="* | group/*"/>
  <xsd:field    xpath="id"/>
</xsd:unique>
```

5.12.2. Contraintes d'existence

Une contrainte d'*existence* spécifie que dans le contenu d'un élément donné, il doit exister un élément ayant une propriété fixée. Comme pour les contraintes d'unicité, cette propriété est très souvent la valeur d'un attribut mais elle peut aussi être formée des valeurs de plusieurs enfants ou attributs. L'idée générale est de référencer un élément par une valeur appelée *clé* et que cet élément doit exister. Cette idée généralise les attributs de types `IDREF` et `IDREFS` des DTD.

Ces contraintes sont introduites par un élément `xsd:keyref` qui doit être enfant d'un élément `xsd:element`. Comme pour les contraintes d'unicité, cet élément dans lequel se trouve la contrainte définit la portée de celle-ci.

Chaque élément `xsd:keyref` possède des attributs `name` et `refer`. L'attribut `name` donne le nom de la contrainte. La valeur de l'attribut `refer` doit être le nom, c'est-à-dire la valeur de l'attribut `name`, d'une contrainte d'unicité qui est associée à cette contrainte d'existence. L'élément `xsd:keyref` contient un élément `xsd:selector` et des éléments `xsd:field` possédant chacun un attribut `xpath`. L'élément `xsd:selector` sélectionne sur quels éléments porte la contrainte. La valeur de son attribut `xpath` est une expression XPath qui sélectionne des éléments concernés. Les éléments `xsd:field` déterminent les différents champs de la valeur servant de clé. La contrainte donnée par un élément `xsd:keyref` impose que pour chaque élément sélectionné, il existe un élément sélectionné par la contrainte d'unicité associée qui a la même valeur.

Dans l'exemple suivant, la contrainte d'unicité `idchapter` impose que la valeur d'un attribut `id` d'un élément `chapter` soit unique. La contrainte d'existence `idref` utilise cette contrainte `idchapter` pour imposer que la valeur d'un attribut `idref` de tout élément `ref` soit aussi la valeur d'un attribut `id` d'un élément `chapter`. Ceci signifie que tout élément `ref` référence, par son attribut `idref`, un chapitre qui existe bien dans le document.

```
    <!-- Unicité des attributs id des éléments chapter -->
    <xsd:key name="idchapter">
      <xsd:selector xpath="chapter"/>
      <xsd:field    xpath="@id"/>
    </xsd:key>
    <!-- Existence des références idref des éléments ref -->
    <xsd:keyref name="idref" refer="idchapter">
      <xsd:selector xpath=".//ref"/>
      <xsd:field    xpath="@idref"/>
    </xsd:keyref>
```

Dans l'exemple précédent, la valeur d'un des attributs `xpath` est l'expression `.//ref` qui sélectionne tous les descendants de nom `ref` de l'élément courant. Cette expression est en fait une abréviation [Section 6.7] de l'expression `./descendant-or-self::node()/ref`.

5.12.3. Exemple complet

Voici un exemple de document 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-ZA" 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-ZA">Product n° 2</product>
    <product serial="102-XY">Product n° 3</product>
    <product serial="102-XA">Product n° 4</product>
  </catalog>
</list>
```

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 -->
  ...
```

5.13. Espaces de noms

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

5.13.1. Schéma 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, que les noms des éléments du document ne soit pas qualifiés (sans le caractère `':'`) et que l'espace de noms par défaut [Section 4.5] ne soit pas spécifié.

Si l'attribut `targetNamespace` de l'élément `xsd: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`, …).

5.13.2. Espace de noms cible

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

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 `xsd:element` est directement fils de l'élément `xsd: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 targetNamespace="http://www.liafa.jussieu.fr/~carton/"
            elementFormDefault="unqualified"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <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"?>
<tns:name xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <firstname>Gaston</firstname>
  <lastname>Lagaffe</lastname>
</tns: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 targetNamespace="http://www.liafa.jussieu.fr/~carton/"
            elementFormDefault="qualified"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <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"?>
<tns:name xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <tns:firstname>Gaston</tns:firstname>
  <tns:lastname>Lagaffe</tns:lastname>
</tns:name>
```

Le comportement pour les attributs est identique mais il est gouverné par l'attribut `attributeFormDefault` de l'élément `xsd: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 dicté par `elementFormDefault` et `attributeFormDefault` en utilisant l'attribut `form` des éléments `xsd:element` et `xsd:attribute`.

Cet attribut peut prendre les valeurs `qualified` ou `unqualified`. Le schéma suivant spécifie que l'élément `firstname` doit être qualifié. Tous les autres éléments locaux comme `lastname` n'ont pas à être qualifiés car la valeur par défaut de l'attribut `elementFormDefault` est `unqualified`.

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

5.13.3. Noms qualifiés

Lorsqu'un élément, un attribut, un groupe d'éléments, un groupe d'attributs ou encore 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.

Les éléments, attributs, groupes et types doivent être nommés avec un nom non qualifié quand ils sont déclarés ou définis. Ils sont à ce moment implicitement qualifiés par l'espace de nom cible. Ceci signifie que les noms apparaissant dans l'attribut `name` de `xsd:element`, `xsd:attribute`, `xsd:group`, `xsd:attributeGroup`, `xsd:simpleType` et `xsd:complexType` sont toujours des noms locaux, c'est-à-dire sans préfixe.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsd:schema targetNamespace="http://www.liafa.jussieu.fr/~carton/"
            elementFormDefault="qualified"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <!-- Référence au type Name par son nom qualifié -->
  <!-- Le nom name de l'élément déclaré n'est pas qualifié -->
  <xsd:element name="name" type="tns:Name" />

  <!-- Le nom Name du type défini n'est pas qualifié -->
  <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>
```

Dans l'exemple précédent, le type `Name` est référencé par son nom qualifié dans la déclaration de l'élément `name`. De la même façon, toute référence à un élément déclaré globalement ou à un groupe d'éléments ou d'attributs utilise un nom qualifié. Dans l'exemple suivant, l'élément `name` apparaît dans la définition d'un autre type `Tree`

qui pourrait être ajoutée au schéma précédent. La définition de ce type est récursive et la référence à lui-même utilise bien sûr le nom qualifié.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema targetNamespace="http://www.liafa.jussieu.fr/~carton/"
     xmlns:tns="http://www.liafa.jussieu.fr/~carton/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Déclaration des éléments globaux name et tree -->
  <xsd:element name="name" type="xsd:string"/>
  <!-- Référence au type Tree par son nom qualifié -->
  <xsd:element name="tree" type="tns:Tree"/>
  <xsd:complexType name="Tree">
    <xsd:sequence>
      <!-- Référence à l'élément global name par son nom qualifié -->
      <xsd:element ref="tns:name"/>
      <!-- Référence récursive au type Tree par son nom qualifié -->
      <!-- Le nom child de l'élément déclaré n'est pas qualifié -->
      <xsd:element name="child" type="tns:Tree" minOccurs="0" maxOccurs="2"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

L'utilisation de `minOccurs` avec la valeur `0` pour l'élément `child` est indispensable pour terminer la récursivité. Sinon, aucun document valide ne peut avoir d'élément de type `Tree`. Le document suivant est valide pour le schéma précédent.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<tns:tree xmlns:tns="http://www.liafa.jussieu.fr/~carton/">
  <tns:name>Racine</tns:name>
  <child>
    <tns:name>Fils Gauche</tns:name>
    <child><tns:name>Petit fils</tns:name></child>
  </child>
  <child><tns:name>Fils Droit</tns:name></child>
</tns:tree>
```

Il est souvent assez lourd de qualifier chacun des noms des objets définis dans le schéma. Une alternative 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 ne pas utiliser l'espace de noms par défaut pour les éléments des schémas comme il pourrait être tentant de le faire. Dans la pratique, on associe l'espace de noms par défaut à l'espace de noms cible et on déclare également un préfixe pour cet espace de noms afin de pouvoir y faire référence de façon explicite. Dans l'exemple suivant, l'espace de noms cible `http://www.liafa.jussieu.fr/~carton/` est déclaré comme l'espace de noms par défaut et il est également associé au préfixe `tns`.

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

  <!-- Définition du 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>
```

5.14. Imports d'autres schémas

Dans un souci de modularité, il est possible d'importer d'autres schémas dans un schéma à l'aide des éléments `xsd:include` et `xsd:import`. L'élément `xsd:include` est employé lorsque l'espace de noms cible est identique pour le schéma importé. L'élément `xsd:import` est employé lorsque l'espace de noms cible du schéma importé est différent de celui qui réalise l'import. Les deux éléments `xsd:include` et `xsd:import` possèdent un attribut `schemaLocation` pour donner l'URL du schéma. L'élément `xsd:import` a, en outre, un attribut `namespace` pour spécifier l'URI qui identifie l'espace de noms cible du schéma importé.

Le schéma à l'adresse `http://www.w3.org/2001/xml.xsd` contient les définitions des quatre attributs particuliers [Section 2.7.4] `xml:lang`, `xml:space`, `xml:base` et `xml:id` de l'espace de noms XML [Section 4.7]. 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>
```

Un document valide pour ce schéma est le suivant. L'espace de noms XML est toujours associé au préfixe `xml` et n'a pas besoin d'être déclaré.

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

5.15. Expressions rationnelles

Une expression rationnelle désigne un ensemble de chaînes de caractères. Une chaîne donnée est *conforme* à une expression si elle fait partie des chaînes décrites. Les expressions rationnelles sont construites à partir des caractères et des opérateurs d'union `'|'`, de concaténation et de répétition `'?'`, `'*'` et `'+'`.

La syntaxe des expressions rationnelles des schémas est inspirée de celle du langage Perl avec cependant quelques variations. Elle diffère de la syntaxe utilisée par les DTD pour décrire les contenus purs d'éléments [Section 3.6.1] car la concaténation n'est plus marquée par la virgule `','`. Ces expressions rationnelles sont utilisées par la facette `xsd:pattern` [Section 5.9.1.3] pour définir des restrictions de types simples. Elles sont également utilisées, en dehors des schémas, par les fonctions XPath `matches()`, `replace()` et `tokenize()` [Section 6.3.3] ainsi que l'élément XSLT `xsl:analyze-string` [Section 8.14].

La plupart des caractères se désignent eux-mêmes dans une expression. Ceci signifie que l'expression `'a'` (sans les apostrophes) désigne l'ensemble réduit à l'unique chaîne `'a'` (sans les apostrophes). Certains caractères, dits *spéciaux*, ont une signification particulière dans les expressions. La liste des caractères spéciaux comprend les

caractères `'|'`, `'?'`, `'*'`, `'+'`, `'.'`, `'\'`, `'('`, `')'`, `'['`, `']'`, `'{'`, `'}'`. Pour supprimer le caractère spécial d'un de ces caractères et l'inclure dans une expression, il faut le faire précéder du caractère d'échappement `'\'`. Quelques caractères d'espacement [Section 2.2.2] bénificient de notations spécifiques. La tabulation `U+09`, le saut de ligne `U+0A` et le retour chariot `U+0D` peuvent être désignés par `'\t'`, `'\n'` et `'\r'`. Ces notations ne sont pas indispensables puisque ces trois caractères peuvent être insérés dans un document XML avec la notation `&#...`.

5.15.1. Opérateurs

Les principaux opérateurs des expressions rationnelles sont l'union désignée par le caractère `'|'`, la concaténation notée par simple juxtaposition et les opérateurs de répétition `'?'`, `'*'`, `'+'`. Comme pour les expressions arithmétiques, on peut utiliser les parenthèses `'('` et `')'` pour former des groupes. Une expression de la forme *`expr-1`*`|`*`expr-2`* désigne l'union des ensembles de chaînes désignés par *`expr-1`* et *`expr-2`*. L'expression `a|b` désigne, par exemple, l'ensemble contenant les deux chaînes `'a'` et `'b'` (sans les apostrophes). Une expression de la forme *`expr-1expr-2`* désigne l'ensemble des chaînes obtenue en concaténant (c'est-à-dire en mettant bout à bout) une chaîne conforme à l'expression *`expr-1`* et une chaîne conforme à l'expression *`expr-2`*. L'expression `ab` désigne, par exemple, l'ensemble contenant l'unique chaîne `ab`. Il est, bien sûr, possible de combiner ces opérateurs. L'expression `aba|ba` désigne, par exemple, l'ensemble formé des deux chaînes `aba` et `ba`. L'expression `a(a|b)bc` désigne l'ensemble contenant les deux chaînes `aabc` et `abbc`.

5.15.2. Répétitions

Les opérateurs `'?'`, `'*'`, `'+'` permettent de répéter des groupes. Ils utilisent une notation postfixée : ils se placent après leur opérande. S'ils sont placés après un caractère, ils s'appliquent uniquement à celui-ci mais s'ils sont placés après un groupe entre parenthèses, ils s'appliquent à tout le groupe. Ils autorisent, respectivement, la répétition de leur opérande, zéro ou une fois, un nombre quelconque de fois et un nombre positif de fois. Une expression de la forme *`expr`*`?` désigne l'ensemble constitué de la chaîne vide et des chaînes conformes à l'expression *`expr`*. Une expression de la forme *`expr`*`*` désigne l'ensemble constitué de toutes les chaînes obtenues en concaténant plusieurs (éventuellement zéro) chaînes conformes à l'expression *`expr`*. L'expression `a?` désigne, par exemple, l'ensemble formé de la chaîne vide et de la chaîne `'a'`. Les expressions `a*` et `(a|b)*` désignent, respectivement, les ensembles constitués des chaînes formées uniquement de `'a'` et les chaînes formées de `'a'` et de `'b'`.

Les accolades `'{'` et `'}'` permettent d'exprimer des répétitions dont le nombre est, soit un entier fixé, soit dans un intervalle donné. Une expression de la forme *`expr`*`{`*`n`*`}`, où *`n`* est un entier, désigne l'ensemble constitué de toutes les chaînes obtenues en concaténant exactement *`n`* chaînes conformes à l'expression *`expr`*. Une expression de la forme *`expr`*`{`*`m`*`,`*`n`*`}`, où *`m`* et *`n`* sont des entiers, désigne l'ensemble constitué de toutes les chaînes obtenues en concaténant un nombre de chaînes conformes à l'expression *`expr`* compris, au sens large, entre *`m`* et *`n`*. L'entier *`n`* peut être omis pour donner une expression de la forme *`expr`*`{`*`m`*`,}`. Le nombre de répétitions doit seulement être supérieur à *`m`*. L'expression `(a|b){6}` désigne, par exemple, les chaînes de longueur 6 formées de `'a'` et de `'b'`. L'expression `(a|b){3,8}` désigne les chaînes de longueur comprise entre 3 et 8 formées de `'a'` et de `'b'`.

5.15.3. Ensembles de caractères

Il est souvent nécessaire de désigner des ensembles de caractères pour construire des expressions rationnelles. Il existe plusieurs façons de décrire ces ensembles.

Le caractère spécial `'.'` désigne tout caractère autre qu'un retour à la ligne. L'expression `a.b` désigne, par exemple, l'ensemble de toutes les chaînes de trois caractères dont le premier est `'a'` et le dernier est `'b'`. L'expression `.*` désigne l'ensemble de toutes les chaînes ne contenant aucun retour à la ligne.

Un ensemble fini de caractères peut simplement être décrit en donnant ces caractères encadrés par des crochets `'['` et `']'`. L'expression `[aeiouy]` désigne l'ensemble des voyelles minuscules et elle est équivalente à l'expression `a|e|i|o|u|y`. Cette syntaxe est pratique car elle permet d'inclure facilement un intervalle en plaçant un tiret `'-'` entre les deux caractères qui le délimitent. L'expression `[0-9]` désigne, par exemple, l'ensemble des chiffres. Il est possible mettre des caractères et plusieurs intervalles. L'expression `[:a-zA-F]` désigne l'ensemble des lettres minuscules et majuscules et du caractère `':'`. L'ordre des caractères entre les crochets est sans importance. Pour inclure un tiret `'-'` dans l'ensemble, il faut le placer en premier ou en dernier. Pour inclure un crochet fermant `']'`, il faut le placer en premier, juste après le crochet ouvrant. L'expression `[-az]` désigne l'ensemble des trois caractères `'-'`, `'a'` et `'z'`.

Lorsque le premier caractère après le crochet ouvrant est le caractère `'^'`, l'expression désigne le complémentaire de l'ensemble qui aurait été décrit sans le caractère `'^'`. L'expression `[^0-9]` désigne, par exemple, l'ensemble des caractères qui ne sont pas un chiffre. Pour inclure un caractère `'^'` dans l'ensemble, il faut le placer à une autre place que juste après le crochet ouvrant. L'expression `[0-9^]` désigne, par exemple, l'ensemble formé des chiffres et du caractère `'^'`.

Un ensemble de caractères peut être décrit comme la différence de deux ensembles entre crochet. La syntaxe prend la forme `[...-[...]]` où `...` est remplacé par une suite de caractères et d'intervalles comme dans les exemples précédents. L'expression `[a-z-[aeiouy]]` désigne, par exemple, l'ensemble des consonnes minuscules.

Certains ensembles de caractères fréquemment utilisés peuvent être décrits par le caractère d'échappement `'\'` suivi d'une lettre. Les différentes combinaisons possibles sont données ci-dessous.

`\s`

caractères d'espacement [Section 2.2.2], c'est-à-dire l'ensemble `[ \t\n\r]`

`\S`

caractères autres que les caractères d'espacement, c'est-à-dire l'ensemble `[^ \t\n\r]`

`\d`

chiffres, c'est-à-dire `[0-9]`

`\D`

caractères autres que les chiffres, c'est-à-dire `[^0-9]`

`\w`

caractères alphanumériques et le tiret `'-'`, c'est-à-dire `[-0-9a-zA-Z]`

`\W`

caractères autres que les caractères alphanumériques et le tiret, c'est-à-dire `[^-0-9a-zA-Z]`

`\i`

premiers caractères des noms XML [Section 2.2.3], c'est-à-dire `[:_a-zA-Z]`

`\I`

caractères autres que les premiers caractères des noms XML, c'est-à-dire `[^:_a-zA-Z]`

`\c`

caractères des noms XML, c'est-à-dire `[-.:_a-zA-Z]`

`\C`

caractère autres que les caractères des noms XML, c'est-à-dire `[^-.:_a-zA-Z]`

L'expression `\s*\d+\s*` désigne, par exemple, une suite non vide de chiffres encadrée, éventuellement, par des caractères d'espacement. L'expression `\i\c*` désigne l'ensemble des noms XML.

Il est aussi possible de décrire un ensemble de caractères en utilisant une des catégories de caractères Unicode. La syntaxe prend la forme `\p{`*`cat`*`}` pour l'ensemble des caractères de la catégorie *`cat`* et `\P{`*`cat`*`}` pour son complémentaire. L'expression `\p{Lu}` désigne, par exemple, toutes les lettres majuscules. Les principales catégories de caractères Unicode sont les suivantes.

Catégorie	Signification	
L	lettres	
Lu	lettres majuscules	
Ll	lettres minuscules	
N	chiffres	
P	caractère de ponctuation	
Z	Séparateurs	

Tableau 5.1. Catégories Unicode

Chapitre 6. XPath

XPath est un langage permettant de sélectionner des parties d'un document XML. Il est utilisé dans de nombreux dialectes XML. Dans cet ouvrage, il est déjà apparu dans les contraintes de cohérence [Section 5.12] des schémas XML. Les schématrons [Chapitre 7] du chapitre suivant sont essentiellement basés sur XPath. Le langage XSLT [Chapitre 8] fait également un usage intensif de XPath pour désigner les parties à traiter.

Le langage XPath n'est pas un langage autonome. C'est un langage d'expressions utilisé au sein d'un autre langage hôte. Il ressemble, dans cet aspect, aux expressions rationnelles, appelées aussi expressions régulières qui est abrégé en *regex* telles qu'elles sont utilisées dans les langages de script tels que Perl ou Python.

La syntaxe de XPath n'est pas une syntaxe XML car les expressions XPath apparaissent en général comme valeurs d'attributs de documents XML. C'est en particulier le cas pour les schémas, les schématrons et XSLT.

XPath était au départ un langage permettant essentiellement de décrire des ensembles de nœuds dans un document XML. La version 1.0 de XPath comprenait quelques fonctions pour la manipulation de nombres et de chaînes de caractères. L'objectif était alors de pouvoir comparer les contenus de nœuds. La version 2.0 de XPath a considérablement enrichi le langage. Il est devenu un langage beaucoup plus complet capable, par exemple, de manipuler des listes de nœuds et de valeurs atomiques.

XPath est uniquement un langage d'expressions dont l'évaluation donne des valeurs sans effet de bord. Il n'est pas possible dans XPath de mémoriser un résultat. Il n'existe pas de variables propres à XPath mais une expression XPath peut référencer des variables du langage hôte. Les valeurs de ces variables sont alors utilisées pour évaluer l'expression. L'affectation de valeurs à ces variables se fait uniquement au niveau du langage hôte. Le langage XPath utilise aussi des variables propres particulières qui servent à parcourir des listes. Ces variables s'apparentent aux variables du langage hôte car elles ne sont jamais affectées explicitement. Leur portée est en outre toujours limitée à un opérateur d'itération comme `for` [Section 6.6.2].

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

6.1. Données et environnement

Une expression XPath est généralement évaluée par rapport à un document XML pour en sélectionner certaines parties. Le document XML est vu comme un arbre formé de nœuds. Les principaux nœuds de cet arbre sont les éléments, les attributs et le texte du document mais les commentaires et les instructions de traitement apparaissent aussi comme des nœuds. L'objet central de XPath est donc le nœud d'un document XML. XPath prend aussi en compte les contenus des éléments et les valeurs des attributs pour effectuer des sélections. Dans ce but, XPath manipule aussi des valeurs atomiques des types prédéfinis [Section 5.5.1] des schémas : `xsd:string`, `xsd:integer`, `xsd:decimal`, `xsd:date`, `xsd:time` ….

6.1.1. Arbre d'un document XML

Il a été vu au chapitre sur la syntaxe [Chapitre 2] que les éléments d'un document XML sont reliés par des liens de parenté. Un élément est le parent d'un autre élément s'il le contient. Ces relations de parenté constituent l'arbre des éléments. Cette structure d'arbre est étendue à tous les constituants d'un document pour former l'*arbre du document* qui inclut les éléments et leurs contenus, les attributs, les instructions de traitement et les commentaires. C'est sous cette forme d'arbre que le document XML est manipulé par XPath et XSLT.

L'arbre d'un document est formé de nœuds de différents types qui correspondent aux différents constituants du document. Ces types de nœuds font partie du système de types [Section 6.1.4] de XPath. Ils sont les suivants.

document node (root node)

> Le nœud racine de l'arbre d'un document est un nœud particulier appelé *document node* ou *root node* dans la terminologie de XPath 1.0. Ce nœud ne doit pas être confondu avec l'élément racine qui est un enfant de ce document node.

element node
: Chaque élément du document est représenté par un nœud de cette sorte. Le contenu texte de l'élément n'est pas contenu dans ce nœud mais dans des nœuds textuels.

attribute node
: Chaque attribut est représenté par un nœud de cette sorte dont le parent est le nœud de l'élément ayant cet attribut. La valeur de l'attribut est contenue dans le nœud.

comment node

: Chaque commentaire du document est représenté par un nœud de cette sorte qui contient le texte du commentaire.

processing instruction node

: Chaque instruction de traitement est représentée par un nœud de cette sorte qui contient le texte de l'instruction.

text node
: Ces nœuds dits *textuels* encapsulent le contenu texte des éléments. Chaque fragment de texte non intérrompu par un élément, une instruction de traitement ou un commentaire est contenu dans un tel nœud. Le contenu textuel d'un élément est réparti dans plusieurs nœuds de cette sorte lorsque l'élément contient aussi d'autres éléments, des commentaires ou des instructions de traitement qui scindent le contenu textuel en plusieurs fragments.

namespace node
: Les nœuds de cette sorte représentaient en XPath 1.0 les espaces de noms déclarés dans un élément. Il sont obsolètes et et ne doivent plus être utilisés.

Afin d'illustrer ces sortes de nœuds, voici ci-dessous un document XML très simple ainsi que son arbre.

```
<?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"?>
<list type="technical">
 <item key="id001" lang="fr">
   XML &amp; Co
 </item>
 <item>
   <!-- Un commentaire inutile -->
   Du texte
 </item>
 et encore du texte à la fin.
</list>
```

Dans la figure ci-dessous, chaque nœud de l'arbre est représenté par un rectangle contenant trois informations. La première information est la sorte du nœud, la deuxième est le nom éventuel du nœud et la troisième est la valeur textuelle du nœud.

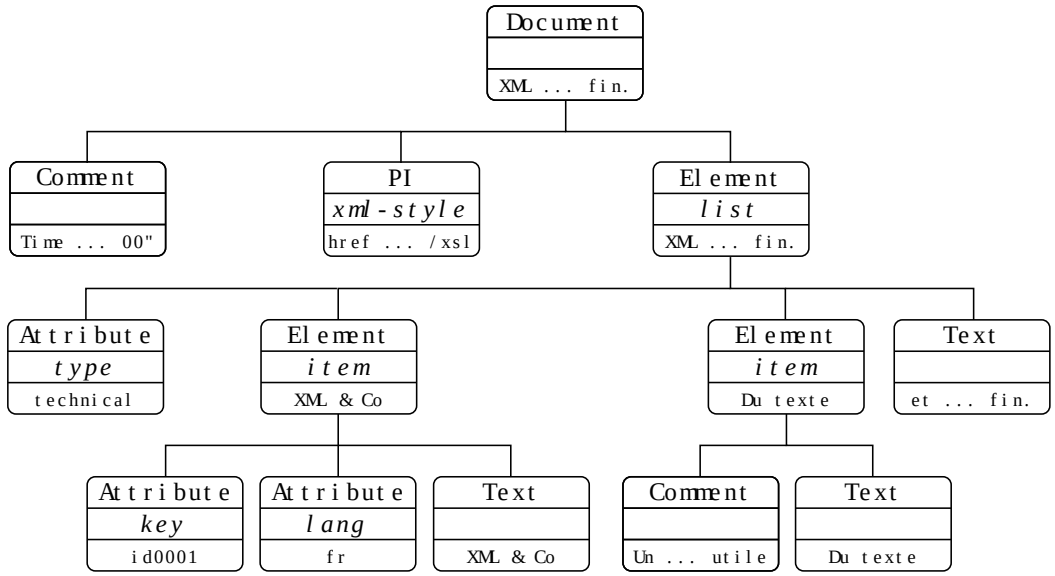


Figure 6.1. Arbre d'un document XML

6.1.1.1. Propriétés des nœuds

Les nœuds de l'arbre d'un document sont caractérisés par un certain nombre de propriétés. Ces propriétés interviennent de façon essentielle dans la sélection des nœuds effectuée par Xpath. Suivant les sortes de nœuds, certaines propriétés n'ont pas de sens et elles n'ont alors pas de valeur. À titre d'exemple, le nœud d'un commentaire n'a pas de nom.

nom
: Pour le nœud d'un élément, d'un attribut ou d'une instruction de traitement, le nom est bien sûr le nom de l'élément, de l'attribut ou de l'instruction. Les nœuds des commentaires et les nœuds textuels n'ont pas de nom. Le nœud racine n'a également pas de nom.

parent
: Tout nœud à l'exception du nœud racine a un parent qui est soit le nœud racine soit le nœud de l'élément qui le contient. La relation parent/enfant n'est pas symétrique en XML. Bien que le parent d'un attribut soit l'élément qui le contienne, l'attribut n'est pas considéré comme un enfant de cet élément (cf. propriété suivante).

enfants
: Seuls le nœud racine et les nœuds des éléments peuvent avoir des enfants. Les enfants d'un nœud sont les nœuds des éléments, des instructions de traitement et des commentaires ainsi que les nœuds textuels qu'il contient. Les attributs ne font pas partie des enfants d'un élément. L'élément racine est un enfant du nœud racine qui peut aussi avoir des instructions de traitement et des commentaires comme enfants.

valeur textuelle
: Chaque nœud de l'arbre du document XML a une valeur qui est une chaîne de caractères Unicode [Section 2.2]. Pour un élément, cette valeur textuelle est le résultat de la concaténation des contenus de tous les nœuds textuels qui sont descendants du nœud. Ce texte comprend donc tout le texte contenu dans le nœud, y compris le texte contenu dans ses descendants. De manière imagée, cette chaîne est aussi obtenue en supprimant du contenu de l'élément les balises de ses descendants, les instructions de traitement et les commentaires. La valeur textuelle de l'élément `p` du fragment de document XML

```
<p>Texte en <i>italique</i> et en <b>gras <i>italique</i></b></p>
```

est donc la chaîne `Texte en italique et en gras italique`. La valeur textuelle est retournée par l'instruction XSLT `xsl:value-of` [Section 8.6.5].

Pour un attribut, la valeur textuelle est la valeur de l'attribut. Pour une instruction de traitement, c'est le texte qui suit le nom de l'instruction. Pour un commentaire, c'est le texte du commentaire sans les délimiteurs `<!--` et `-->`. La valeur textuelle d'un nœud textuel est juste le texte qu'il contient.

type
: Cette propriété n'existe que si le document a été un validé par un schéma [Chapitre 5]. Il s'agit du type attribué à l'élément ou à l'attribut lors de la validation par le schéma. Ce type peut être un type prédéfini ou un type défini dans le schéma.

valeur typée
: Cette valeur n'existe que si le document a été un validé par un schéma [Chapitre 5]. Il s'agit de la valeur obtenue en convertissant la valeur textuelle dans le type du nœud. Si le type d'un attribut est, par exemple, `xsd:double` et sa valeur textuelle est la chaîne `1.0e-3`, sa valeur est le nombre flottant `0.001`. Cette conversion n'est possible que si le type du nœud est un type simple.

Chaque nœud a une valeur qui est utilisée à chaque fois qu'un nœud apparaît là où une valeur atomique est requise. Beaucoup de fonctions et d'opérateurs XPath prennent en paramètre des valeurs atomiques. Lorsqu'un nœud est passé en paramètre à de tels fonctions ou opérateurs, celui-ci est converti automatiquement en sa valeur. Ce processus se déroule de la façon suivante. Si le nœud a une valeur typée (c'est-à-dire lorsque le document est traité avec un schéma), la valeur est la valeur typée et son type est celui de la valeur typée. Sinon, la valeur est la valeur textuelle du nœud et son type est `xsd:untypedAtomic`. Cette conversion d'un nœud en valeur est appelée *atomisation*.

L'atomisation d'un nœud peut donner une liste de valeurs atomiques éventuellement vide. C'est le cas lorsque le type du nœud est un type de listes comme le type prédéfini `xsd:IDREFS` ou les types construits avec l'opérateur `xsd:list` [Section 5.6.6].

Lorsqu'une liste contenant des nœuds doit être convertie en une liste de valeurs, chacun des nœuds est atomisé et sa valeur le remplace dans la liste. lorsque la valeur du nœud comporte plusieurs valeurs atomiques, celles-ci s'insèrent dans la liste à la place du nœud.

6.1.1.2. Fonctions

Il existe des fonctions XPath permettant de récupérer les propriétés d'un nœud. Les fonctions sont décrites avec leur type de retour et le type de leurs paramètres. Les types utilisés sont ceux du système de typage [Section 6.1.4] de XPath. Lorsque le type du paramètre est `node()?`, la fonction prend en paramètre un nœud ou rien. Dans ce dernier cas, le paramètre est implicitement le nœud courant.

Fonctions sur les nœuds

`xsd:string name(node()? node)`

: retourne le nom complet du nœud `node` ou du nœud courant si `node` est absent.

`xsd:string local-name(node()? node)`

: retourne le nom local du nœud `node` ou du nœud courant si `node` est absent. Le nom local est la partie du nom après le caractère `':'` qui la sépare du préfixe.

`xsd:QName node-name(node()? node)`

: retourne le nom qualifié du nœud `node` ou du nœud courant si `node` est absent.

`xsd:string string(node()? node)`

: retourne la valeur textuelle du nœud `node` ou du nœud courant si `node` est absent. Cette fonction ne doit pas être confondue avec la fonction `xsd:string()` [Section 6.1.6] qui convertit une valeur en chaîne de caractères.

`xsd:anyAtomicType* data(node()* list)`

retourne les valeurs typées des nœuds de la liste `list`.

`node() root(node()? node)`

retourne la racine de l'arbre qui contient le nœud `node` ou le nœud courant si `node` est absent.

`node()* id(xsd:string s)`

la chaîne `s` doit être une liste de noms XML séparés par des espaces. La fonction retourne la liste des nœuds dont la valeur de l'attribut de type `ID` [Section 3.7.2] ou `xsd:ID` [Section 5.5.1.4] est un des noms de la chaîne `s`. La valeur passée en paramètre est souvent la valeur d'un attribut de type `IDREF` ou `IDREFS`. Pour qu'un attribut soit de type `ID`, il doit être déclaré de ce type dans une DTD. Il est aussi possible d'utiliser l'attribut `xml:id` [Section 2.7.4.4] qui est de type `xsd:ID`.

`xsd:anyURI base-uri(node()? node)`

retourne l'URI de base [Section 2.7.4.3] du nœud `node` ou du nœud courant si `node` est absent.

`xsd:anyURI document-uri(node()? node)`

retourne l'URI de base [Section 2.7.4.3] du document contenant le nœud `node` ou le nœud courant si `node` est absent.

`xsd:anyURI resolve-uri(xsd:string? url, xsd:string? base)`

combine l'URI `uri` avec l'URI de base `base` et retourne le résultat. Si l'URI `base` est omise et que le langage hôte est XSLT, c'est l'URI de base de la feuille de style qui est utilisée.

Les espaces de noms [Chapitre 4] sont manipulés en XPath 1.0 à travers l'axe `namespace`. Cet axe est obsolète en XPath 2.0. Il est remplacé par des fonctions permettant d'accéder aux espaces de noms.

`xsd:string* namespace-uri(node()? node)`

retourne l'URI qui définit l'espace de noms dans lequel se trouve le nœud `node` ou le nœud courant si `node` est absent.

`xsd:string* in-scope-prefixes(node() node)`

retourne la liste des préfixes associés à un espace de noms dans le nœud `node`. Si l'espace de noms par défaut est défini, la liste contient la chaîne vide. La liste contient toujours le préfixe `xml` associé à l'espace de noms XML.

`xsd:string namespace-uri-for-prefix(xsd:string prefix, node() node)`

retourne l'URI qui définit l'espace de noms auquel est associé le préfixe `prefix` dans le nœud `node`.

- `for $i in in-scope-prefixes(.) return concat($i, '=', namespace-uri-for-prefix($i, .))`

  donne une chaîne formée d'un bloc de la forme *`prefixe=uri`* pour chaque espace de noms déclaré dans le nœud courant.

6.1.2. Ordre du document

Tous les nœuds d'un document XML sont classés suivant un ordre appelé *ordre du document*. Pour les éléments, cet ordre est celui du parcours préfixe de l'arbre du document. Ceci correspond à l'ordre des balises ouvrantes dans le document. Un élément est toujours placé après ses ancêtres et ses frères gauches et avant ses enfants et ses frères droits. Les attributs et les déclarations d'espaces de noms sont placés juste après leur élément et avant tout autre élément. Les déclarations d'espaces de noms sont placées avant les attributs. L'ordre relatif des déclarations d'espaces de noms et l'ordre relatif des attributs sont arbitraires mais ils restent fixes tout au long du traitement du document par une application.

L'ordre du document peut être manipulé explicitement par les opérateurs XPath `'<<'` et `'>>'` [Section 6.5.3]. Il intervient aussi de façon implicite dans l'évaluation de certains opérateurs, comme par exemple `'/'`, qui ordonnent les nœuds de leur résultat suivant cet ordre.

6.1.3. Modèle de données

Les expressions XPath manipulent des *valeurs* qui sont soit des nœuds de l'arbre d'un document XML, soit des *valeurs atomiques*. Les principales valeurs atomiques sont les entiers, les nombres flottants et les chaînes de caractères. La donnée universelle de XPath est la *liste* de valeurs. Ces listes ne peuvent pas être imbriquées. Une liste contient uniquement des valeurs et ne peut pas contenir une autre liste. La *longueur* de la liste est le nombre de valeurs qu'elle contient. Les valeurs de la liste sont ordonnées et chaque valeur a une position allant de 1 (et non pas 0) pour la première valeur à la longueur de la liste pour la dernière valeur.

Les listes peuvent être construites explicitement avec l'opérateur XPath `','` (virgule). L'opérateur XSLT `xsl:sequence` [Section 8.6.8] permet également de construire explicitement des séquences. Beaucoup d'opérateurs et de fonctions XPath retournent des listes comme valeurs.

Toute valeur est considérée par XPath comme une liste de longueur 1. Inversement, toute liste de longueur 1 est assimilée à l'unique valeur qu'elle contient.

Les valeurs atomiques comprennent les chaînes de caractères, les entiers, les nombres flottants et tous les types prédéfinis [Section 5.5.1] des schémas XML. Les principaux types pour les valeurs atomiques sont les suivants. Pour chacun d'entre eux, un exemple de valeur est donné.

`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 et double précision

`xsd:integer`
: Entier : `42`

`xsd:duration`, `xsd:yearMonthDuration` et `xsd:dayTimeDuration`
: Durée : `P6Y4M2DT11H22M44S` de 6 ans, 4 mois, 2 jours, 11 heures 22 minutes et 44 secondes.

`xsd:date`, `xsd:time` et `xsd:dateTime`
: Date et heure : `2009-03-02T11:44:22`

`xsd:anyURI`
: URL : `http://www.liafa.jussieu.fr/~carton/`

`xsd:anyType`, `xsd:anySimpleType` et `xsd:anyAtomicType`
: Types racine de la hiérarchie

`xsd:untyped` et `xsd:untypedAtomic`
: Nœud et valeur atomique non typée

6.1.4. Typage

XPath possède un système de typage assez rudimentaire qui permet de décrire les types des paramètres et le type de retour des fonctions. Ce système de typage est également utilisé par XSLT pour donner le type d'une variable [Section 8.9] lors de sa déclaration et pour les types de retour et des paramètres des fonctions d'extension

[Section 8.10]. Les types des valeurs influent aussi sur les coportement de certains opérateurs et en particulier des comparaisons [Section 6.5].

Ce système est organisé en une hiérarchie dont la racine est le type `item()` (avec les parenthèses). Tous les autres types dérivent de ce type qui est le plus général. Toute valeur manipulée par XPath est donc de ce type.

Il y a ensuite des types pour les nœuds et des types pour les valeurs atomiques. Pour les nœuds, il y a d'abord un type générique `node()` ainsi que des types pour chacun des types de nœuds [Section 6.1.1]. Ces types sont `document-node()`, `element()`, `attribute()`, `text()`, `processing-instruction()` et `comment()` (avec les parenthèses). Les types pour les valeurs atomiques sont les types prédéfinis [Section 5.5.1] des schémas comme `xsd:integer`.

Toute valeur atomique extraite du document par atomisation est typée. Lorsque le document est validé par un schéma avant traitement par une feuille de style XSLT, les types des valeurs atomiques des contenus d'éléments et des valeurs d'attributs sont donnés par le schéma. Lorsque le document n'est pas validé par un schéma, toutes ces valeurs atomiques provenant du document sont considérées du type `xsd:untypedAtomic`.

Le système de type possède trois opérateurs `'?'`, `'*'` et `'+'`, en notation postfixée, pour construire de nouveaux types. Ils s'appliquent aux types pour les nœuds et les valeurs atomiques décrits précédemment. L'opérateur `'?'` désigne un nouveau type autorisant l'absence de valeur. Ces types sont surtout utilisés pour décrire les paramètres optionnels des fonctions. La fonction XPath `name()` [Section 6.1.1.1] a, par exemple, un paramètre de type `node()?`. Ceci signifie que son paramètre est soit un nœud soit rien. L'opérateur `'*'` construit un nouveau type pour les listes. Le type `xsd:integer*` est, par exemple, le type des listes d'entiers éventuellement vides. L'opérateur `'+'` construit un nouveau type pour les listes non vides. Le type `xsd:integer+` est, par exemple, le type des listes non vides d'entiers.

6.1.5. Contexte

L'évaluation d'une expression XPath se fait dans un *contexte* qui est fourni par le langage hôte. Le contexte se décompose en le *contexte statique* et le *contexte dynamique*. Cette distinction prend du sens lorsque les programmes du langage hôte sont susceptibles d'être compilé. Le contexte statique comprend tout ce qui peut être déterminé par une analyse statique du programme hôte. Tout ce qui dépend du document fait, au contraire, partie du contexte dynamique.

Les expressions XPath sont beaucoup utilisées dans les feuilles de style XSLT. Comme le langage XSLT est sans effet de bord, l'analyse statique des feuilles de styles XSLT est relativement facile et permet de déterminer beaucoup d'information.

6.1.5.1. Contexte statique

On rappelle que la portée d'une déclaration est l'élément qui contient cette déclaration. Pour une déclaration d'espace de noms [Section 4.2], c'est l'élément contenant l'attribut `xmlns`. Pour une déclaration de variable XSLT [Section 8.9], la portée est l'élément parent de l'élément `xsl:variable`. La portée d'une définition d'une fonction d'extension [Section 8.10] est toute la feuille de style car ces définitions sont enfants de l'élément racine. On dit qu'un objet est *en portée* dans une expression XPath si l'expression est incluse dans la portée de l'objet en question. Le contexte statique comprend les objets suivants

Espaces de noms

Tous les espaces de noms [Chapitre 4] en portée, y compris éventuellement l'espace de noms par défaut. Le contexte comprend les associations entre les préfixes et les URL qui identifient les espaces de noms. Ceci permet de prendre en compte les noms qualifiés.

Variables

Toutes les variables en portée et leur type éventuel. Les valeurs de ces variables font partie du contexte dynamique.

Fonctions

Toutes les fonctions disponibles. Ceci comprend les fonctions standards de XPath, les fonctions de conversion de types et les fonctions d'extension définies au niveau du langage hôte.

Collations

Toutes les collations [Section 2.2.5] en portée.

URI de base

URI de base [Section 2.7.4.3] de l'élément.

6.1.5.2. Contexte dynamique

La partie importante du contexte dynamique est appelée le *focus* et elle comprend trois valeurs appelées *objet courant* (*context item* dans la terminologie du W3C), *position dans le contexte* et *taille du contexte*. Le focus peut évoluer au cours de l'évaluation d'une expression.

L'objet courant est indispensable puisqu'il permet de résoudre toutes les expressions relatives très souvent utilisées. C'est un peu l'équivalent du *current working directory* d'un shell Unix. L'expression `@id` retourne, par exemple, l'attribut `id` de l'objet courant. L'objet courant est très souvent un nœud du document mais il peut aussi être une valeur atomique. Lorsque l'objet courant est un nœud, il est appelé le *nœud courant*. L'évaluation de certaines expressions, comme par exemple `@id`, provoque une erreur si l'objet courant n'est pas un nœud. L'objet courant est retourné par l'expression XPath `'.'` (point).

Il est fréquent qu'une expression XPath soit évaluée pour différentes valeurs de l'objet courant parcourant une liste d'objets. Cette liste peut être déterminée avant l'évaluation ou au cours de l'évaluation de l'expression XPath. La position du contexte donne la position de l'objet courant dans cette liste implicite. Cette position est retournée par la fonction XPath `position()`. La taille du contexte est la longueur de cette liste implicite et elle est retournée par la fonction XPath `last()`.

Variables locales

Certains opérateurs XPath comme `for`, `some` et `every` [Section 6.6.2] introduisent des variables locales qui font partie du contexte dynamique.

Valeurs des variables
Les valeurs de toutes les variables [Section 8.9] en portée.

Définition des fonctions
Toutes les définitions des fonctions d'extension [Section 8.10].

6.1.6. Fonctions de conversion

XSLT 1.0 possède une fonction `number()` pour convertir une valeur quelconque en un entier.

Pour chaque type de base des schémas comme `xsd:boolean` ou `xsd:string`, il existe une fonction XPath de conversion ayant le même nom. Celle-ci convertit une valeur quelconque en une valeur du type en question. La fonction `xsd:boolean()` convertit, par exemple, une valeur en une valeur booléenne [Section 6.3.1].

6.1.7. Fonctions externes

Certaines fonctions pouvant être utilisées dans les expressions XPath sont, en réalité, fournies par le langage hôte. Le langage XSLT [Chapitre 8] fournit, en particulier, quelques fonctions très utiles. Il permet également de définir de nouvelles fonctions [Section 8.10].

```
xsd:string format-number(number n, xsd:string format, xsd:ID id)
```

retourne le nombre `n` formaté avec le format `format` où l'interprétation de ce format est donné par l'élément `xsl:decimal-format` [Section 8.6.12] identifié par `id`.

```
xsd:string generate-id(node()? node)
```

retourne un identifiant pour le nœud `node` ou le nœud courant si `node` est absent. Au cours d'un même traitement, plusieurs appels à `generate-id` avec le même paramètre donnent le même identifiant. En

revanche, des appels dans des traitements différents peuvent conduire à des résultats différents. Cette fonction est souvent utilisée pour produire une valeur destinée à un attribut de type `ID` ou `xsd:ID` comme `xml:id` [Section 2.7.4.4].

`node() current()`

retourne le nœud courant auquel s'applique la règle définie par `xsl:template` [Section 8.5.1].

`node()* current-group()`

retourne la liste des nœuds du groupe courant lors de l'utilisation de `xsl:for-each-group` [Section 8.7.4].

`xsd:string current-grouping-key()`

retourne la clé courante lors de l'utilisation de `xsl:for-each-group` [Section 8.7.4].

`xsd:string regexp-group(xsd:integer n)`

retourne le fragment de texte entre une paire de parenthèses de l'expression rationnelle lors de l'utilisation `xsl:matching-substring` [Section 8.14].

6.2. Expressions de chemins

Le premier objectif de XPath est, comme son nom l'indique, d'écrire des chemins dans l'arbre d'un document XML. Ces chemins décrivent des ensembles de nœuds du document qu'il est ainsi possible de manipuler. Le cœur de XPath est constitué des opérateurs de chemins qui permettent l'écriture des chemins.

Le type fondamental de XPath est la liste mais les opérateurs de chemins retournent des listes où les nœuds sont dans l'ordre du document et où chaque nœud a au plus une seule occurrence. Ces listes représentent en fait des ensembles de nœuds. Ce comportement assure une compatibilité avec XPath 1.0 qui manipule des ensembles de nœuds plutôt que des listes.

Il existe deux syntaxes, une explicite et une autre abrégée pour les expressions de chemins. La première facilite la compréhension mais la seconde, beaucoup plus concise, est généralement utilisée dans la pratique. La syntaxe abrégée est décrite en [Section 6.7] mais la plupart des exemples sont donnés avec les deux syntaxes pour une meilleure familiarisation avec les deux syntaxes.

6.2.1. Expressions de cheminement

Les expressions de cheminement permettent de se déplacer dans l'arbre d'un document en passant d'un nœud à d'autres nœuds. Une expression de cheminement a la forme *`axe`*`::`*`type`*. Elle retourne l'ensemble des nœuds du type *`type`* qui sont reliés au nœud courant par la relation *`axe`*. Parmi tous les nœuds, l'axe effectue une première sélection basée sur la position des nœuds dans l'arbre par rapport au nœud courant. Le type raffine ensuite cette sélection en se basant sur le type et le nom des nœuds. La sélection peut encore être affinée par des filtres [Section 6.2.2]. Si l'objet courant n'est pas un nœud, l'évaluation d'une telle expression provoque une erreur. Les différents types correspondent aux différents nœuds pouvant apparaître dans l'arbre d'un document. Les relations entre le nœud courant et les nœuds retournés sont appelées *axes* dans la terminologie XML.

L'axe par défaut est l'axe `child` qui sélectionne les enfants du nœud courant.

`node()`
: tous les enfants du nœud courant

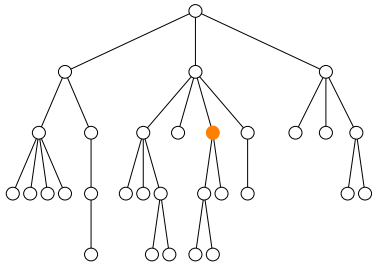
`ancestor::*`
: tous les ancêtres stricts du nœud courant

6.2.1.1. Axes

Chacun des axes donne une relation qui relie les nœuds sélectionnés au nœud courant. Les axes qu'il est possible d'utiliser dans les expressions XPath sont les suivants.

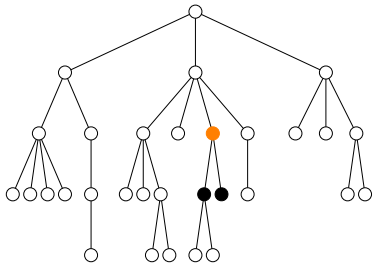
`self`

Le nœud lui-même (égalité)



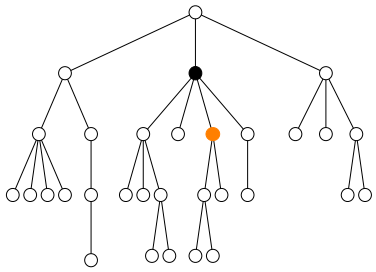
`child`

Enfant direct



`parent`

Parent

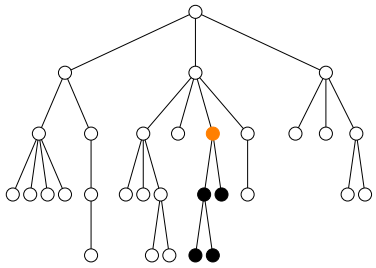


`attribute`

Attribut du nœud

`descendant`

Descendant strict

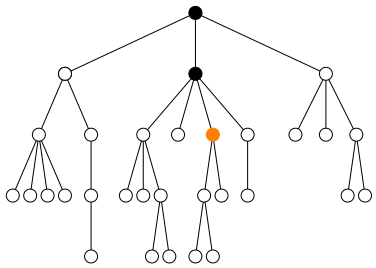


`descendant-or-self`

Descendant ou le nœud lui-même

`ancestor`

Ancêtre strict

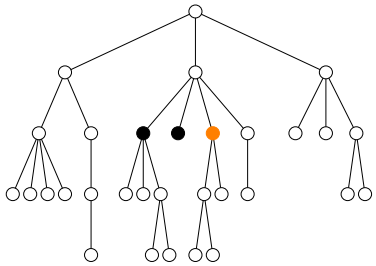


`ancestor-or-self`

Ancêtre ou le nœud lui-même

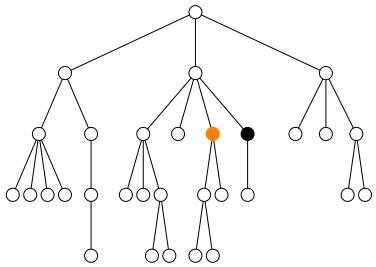
`preceding-sibling`

Frère gauche (enfant du même parent)



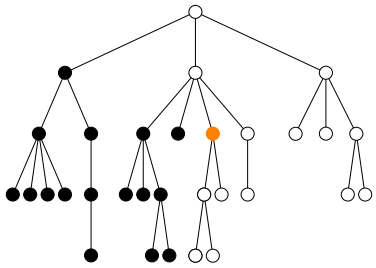
`following-sibling`

Frère droit (enfant du même parent)



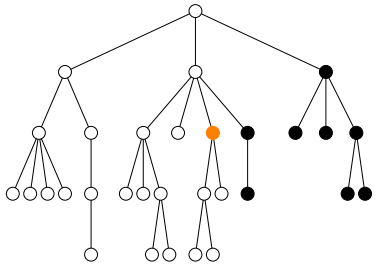
`preceding`

À gauche



`following`

À droite



`namespace`

Espace de noms du nœud

L'axe `namespace` est un héritage de XPath 1.0 qui doit être considéré comme obsolète. Il est conseillé d'utiliser les fonctions XPath pour accéder aux espaces de noms d'un 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.

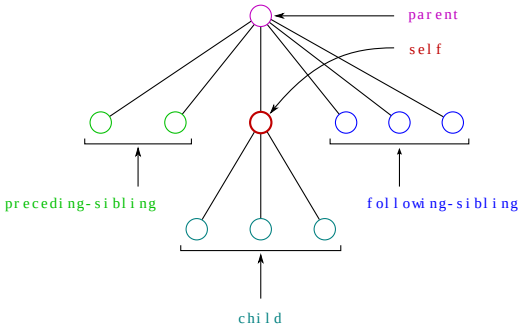


Figure 6.2. Nœuds proches

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.

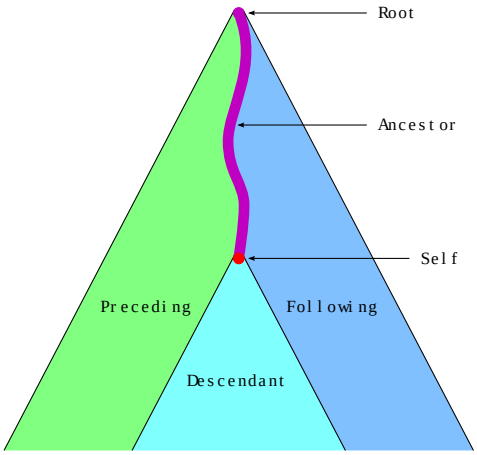


Figure 6.3. Partitionnement des éléments selon les cinq axes

6.2.1.2. 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 utilisé

`ns``:*`
tous les éléments ou tous les attributs (suivant l'axe) dans l'espace de noms associé au préfixe *`ns`*

`*:`*`local`*
tous les éléments ou tous les attributs (suivant l'axe) de nom local *`local`*

`name`
les nœuds de nom *`name`* (éléments ou attributs suivant l'axe utilisé)

`node()`

tous les nœuds

`text()`

tous les nœuds textuels

`comment()`

tous les commentaires

`processing-instruction()` ou `processing-instruction(`*`name`*`)`

toutes les instructions de traitement ou les instructions de traitement de nom *`name`*

Les types `node()`, `text()`, `comment()` et `processing-instruction()` se présentent comme des pseudo fonctions sans paramètre. Les parenthèses sont indispensables car les expressions `text` et `text()` s'évaluent différemment. L'expression `text` retourne les éléments de nom `text` qui sont enfants du nœud courant alors que l'expression `text()` retourne les nœuds textuels qui sont enfants du nœud courant. Il faut, également, distinguer l'expression `string` qui s'évalue en une liste de nœuds de nom `string` des expressions `'string'` et `"string"` qui s'évaluent en une chaîne de caractères.

`child::*` ou `*`

tous les éléments qui sont enfants du nœud courant

`attribute::*` ou `@*`

tous les attributs du nœud courant

`attribute::id` ou `@id`
attribut `id` du nœud courant

`child::node()` ou `node()`
tous les enfants du nœud courant

`child::text` ou `text`
tous les éléments de nom `text` qui sont enfants du nœud courant

`child::text()` ou `text()`

tous les nœuds textuels qui sont enfants du nœud courant

`descendant::comment()`

tous les commentaires qui sont descendants du nœud courant, c'est-à-dire contenus dans le nœud courant

`following::processing-instruction()`

toutes les instructions de traitement qui suivent le nœud courant.

`child::processing-instruction('xml-stylesheet')`

toutes les instructions de traitement de nom `xml-stylesheet` qui sont enfants du nœud courant.

6.2.2. Filtres des expressions de cheminement

Les filtres [Section 6.4.2] permettent de sélectionner dans une liste les objets qui satisfont une condition. Ils apparaissent beaucoup dans les expressions de chemin pour restreindre le résultat d'une expression de cheminement. Nous donnons ici quelques exemples d'utilisation de ces filtres. Lorsque ces expressions XPath apparaissent comme valeur d'attributs, il est nécessaire de remplacer les caractères spéciaux `'<'` et `'>'` par les entités prédéfinies [Section 3.5.1.2].

`child::*[position() < 4]` ou `*[position() < 4]`

les trois premiers éléments enfants du nœud courant

`attribute::*[name() != 'id']` ou `@*[name() != 'id']`

tous les attributs autres que `id`. Cette expression peut aussi être écrite `@* except @id`.

`child::node()[position() = 4]` ou `node()[4]`

le quatrième enfant du nœud courant

`child::section[position() = last()]` ou `section[position() = last()]`

dernier enfant `section` du nœud courant

`descendant::item[attribute::id]` ou `.//item[@id]`

tous les descendants du nœud courant qui sont un élément `item` ayant un attribut `id`

`ancestor::*[@type='base']`

tous les ancêtres du nœud courant dont l'attribut `type` existe et vaut `base`.

`chapter[count(child::section) > 1]`

élément `chapter` ayant au moins deux éléments `section` comme enfants.

6.2.3. Opérateur `'/'` de composition de chemins

L'opérateur `'/'` permet de composer des expressions de cheminement pour créer de véritables chemins dans un arbre XML. C'est un des opérateurs clé de XPath. Sa sémantique est proche du même opérateur `'/'` des shell Unix mais il est plus général et pas toujours facile à appréhender

Une expression de la forme *`expr1/expr2`* est évaluée de la façon suivante. L'expression *`expr1`* est d'abord évaluée pour donner une liste d'objets. Pour chacun des objets de cette liste, l'expression *`expr2`* est évaluée en modifiant dynamiquement le focus [Section 6.1.5.2] de la façon suivante. L'objet courant est fixé à l'objet choisi dans la liste, la position dans le contexte est fixée la position de cet objet dans la liste et la taille du contexte est également fixée à la taille de la liste. Les listes retournées par chacune des évaluations de *`expr2`*, sont fusionnées pour donner une liste de nœuds dans l'ordre du document et avec au plus une occurrence de chaque nœud dans la liste. Cette liste est alors le résultat de l'évaluation de *`expr1/expr2`*. Si l'évaluation de *`expr1`* retourne la liste vide, l'évaluation de *`expr1/expr2`* retourne aussi la liste vide. Le focus reprend, après l'évaluation, sa valeur initiale.

Supposons, par exemple, que l'expression *expr1* retourne la liste (n1, n2, n3) formée de trois nœuds. L'expression *expr2* est donc évaluée trois fois. Elle est évaluée une première fois en prenant le nœud courant égal à n1, la position du contexte égale à 1 et la taille du contexte égale à 3. Elle est évaluée une deuxième fois en prenant le nœud courant égal à n2, la position du contexte égale à 2 et la taille du contexte égale à 3. Elle est évaluée une troisième et dernière fois en prenant le nœud courant égal à n3, la position du contexte égale à 3 et la taille du contexte égale à 3. Si ces trois évaluations retournent respectivement les listes (n0, n2, n5), (n1) et (n0, n1, n2, n4, n6) et que l'ordre du document est n0, n1, n2, n4, n5, n6, le résultat de l'évaluation est la liste (n0, n1, n2, n4, n5, n6).

`self::node()/child::*` ou `./*`
: tous les élements qui sont enfants du nœud courant

`child::*/child::*` ou `*/*`
: tous les éléments qui sont des enfants des enfants du nœud courant

`child::p/child::em` ou `p/em`
: tous les éléments em enfants d'un élément p enfant du nœud courant

`child::*/attribute::*` ou `*/@*`
: tous les attributs des éléments qui sont enfants du nœud courant

`parent::*/parent::*` ou `../..`
: le parent du parent du nœud courant

`parent::*/child::*` ou `../*`
: tous éléments qui sont frères du nœud courant, y compris le nœud courant

`child::*/parent::*` ou `*/..`
: le nœud courant s'il contient au moins un élément ou aucun nœud sinon

`descendant::p/child::em` ou `.//p/em`
: tous les élements em qui sont enfants d'un élément p descendant du nœud courant

`descendant::*/descendant::*` ou `descendant::*`
: tous les éléments descendants du nœud courant

`ancestor::*/descendant::*`
: tous les éléments du document

`descendant::*/ancestor::*`
: tous les ancêtres et tous les descendants du nœud courant ayant au moins un élément comme enfant si le nœud courant a au moins un élément comme enfant et aucun nœud sinon.

`descendant::p/following-sibling::em[position()=1]` ou `.//p/following-sibling::em[1]`
: premier frère em d'un élément p descendant du nœud courant.

L'opérateur '/' peut être cascadé et il est associatif à gauche. Une expression de la forme *expr1/expr2/expr3* est implicitement parenthésée (*expr1/expr2*)/*expr3*. Ce parenthésage est très souvent inutile car l'opérateur '/' est associatif à l'exception de quelques cas pathologiques.

`child::*/child::*/attribute::*` ou `*/*/@*`
: tous les attributs des éléments qui sont des enfants des enfants du nœud courant

`parent::*/child::*/descendant::text()`
: tous les nœuds textuels descendants d'un frère du nœud courant

L'opérateur '/' peut aussi être employé dans des expressions de la forme */expr2* qui sont l'analogue des chemins absolus dans les systèmes de fichiers.

Une expression de la forme */expr2* est évaluée de la façon suivante. L'expression *expr2* est évaluée en ayant, au préalable, fixé le nœud courant à la racine de l'arbre contenant le nœud courant, la position dans le contexte à

1 et la taille du contexte à 1. Une telle expression est donc équivalente à une expression `root(.)/expr2`. La fonction `root()` retourne la racine de l'arbre contenant son paramètre.

Le nœud courant appartient très souvent à l'arbre d'un document XML. Dans ce cas, la racine de cet arbre est un nœud de la sorte `document node`. Il est également possible que le nœud courant appartienne à un fragment de document et que la racine de cet arbre soit un nœud quelconque.

Les expressions de la forme `/expr2` se comportent comme des chemins absolus puiqu'elles s'évaluent en partant d'une racine. Elles ne sont toutefois pas complètement absolues car la racine choisie dépend du nœud courant. La racine choisie est celle de l'arbre contenant le nœud courant.

`/`
: le nœud `document node` racine de l'arbre

`/child::*` ou `/*`
: l'élément racine du document

`/child::book` ou `/book`
: l'élément racine du document si son nom est `book` et aucun nœud sinon

`/child::book/child:chapter` ou `/book/chapter`
: les éléments `chapter` enfant de l'élément racine du document si son nom est `book` et aucun nœud sinon

`/descendant::section` ou `//section`
: tous les éléments `section` du document

6.2.4. Expressions ensemblistes

Les opérateurs sur les ensembles réalisent les opérations d'union, d'intersection et de différences sur les ensembles de nœuds représentés par des listes classées dans l'ordre du document. L'opérateur d'union est très souvent utilisé alors que les deux autres opérateurs le sont plus rarement.

L'opérateur d'union est noté par le mot clé `union` ou par le caractère `'|'`. Les opérateurs d'intersection et de différence sont notés respectivement par les mots clé `intersect` et `except`.

`child::* union attribute::*` ou `* | @*`
: tous les attributs et tous les éléments enfants du nœud courant

`*:* except xsl:*`
: tous les éléments enfants du nœud courant qui ne sont pas dans l'espace de noms associé au préfixe `xsl`

`@* except @id`
: tous les attributs excepté l'attribut `id`

Bien que les listes manipulées par XPath peuvent contenir des nœuds et des valeurs atomiques, ces opérateurs fonctionnent uniquement avec des listes représentant des ensembles de nœuds. Ils ne fonctionnent pas avec des listes contenant des valeurs atomiques. Les listes de nœuds retournées par ces opérateurs sont triées dans l'ordre du document et ne comportent pas de doublon.

6.3. Valeurs atomiques

6.3.1. Expressions booléennes

Le type booléen contient les deux valeurs `true` et `false`. Ces deux valeurs sont retournées par les fonctions `true()` et `false()` sans paramètre. Les parenthèses sont obligatoires car l'expression XPath `true` donne les éléments `true` enfants du nœud courant.

Les valeurs booléennes peuvent être manipulées à l'aide des opérateurs `and` et `or` et de la fonction `not()`.

```
true() and false()
   false

true() or false()
   true

not(false())
   true
```

Les valeurs des autres types peuvent être converties explicitement en des valeurs booléennes par la fonction `xsd:boolean()`. Elles sont aussi converties implicitement dès qu'une valeur booléenne est requise. C'est le cas, par exemple, des filtres [Section 6.4.2] et de la structure de contrôle `if` [Section 6.6.1] de XPath ou des structures de contrôle `xsl:if` [Section 8.7.1] et `xsl:when` [Section 8.7.2] de XSLT. Les règles qui s'appliquent alors sont les suivantes.

- Une liste vide est convertie en `false`.
- Une liste non vide dont le premier élément est un nœud est convertie en `true`.
- Si la liste contient une seule valeur atomique, les règles suivantes s'appliquent. Rappelons qu'une valeur atomique est considérée comme une liste contenant cet objet et inversement.
  - Un nombre est converti en `true` sauf s'il vaut `0` ou `NaN`.
  - Une chaîne de caractères est convertie en `true` sauf si elle est vide.
- Les autres cas provoquent une erreur.

`xsd:boolean(())`
: donne `false` car la liste est vide

`xsd:boolean((/,0))`
: donne `true` car le premier objet de la liste est un nœud

`xsd:boolean(0)`
: donne `false` car l'entier est égal à `0`

`xsd:boolean(7)`
: donne `true` car l'entier est différent de `0`

`xsd:boolean('')`
: donne `false` car la chaîne de caractères est vide

`xsd:boolean('string')`
: donne `true` car la chaîne de caractères n'est pas vide

6.3.2. Nombres

Les seuls nombres existant en XPath 1.0 étaient les nombres flottants. XPath 2.0 manipule les nombres des trois types `xsd:integer`, `xsd:decimal` et `xsd:double` des schémas XML [Section 5.5.1.1].

Fonctions sur les nombres

Opérateurs `'+'`, `'-'` et `'*'`
: Ces opérateurs repectivement calculent la somme, la différence et le produit de deux nombres entiers, décimaux ou flottants.

  - `2+3, 2-3, 2*3`
    `5, -1, 6`

Opérateur `div`

Cet opérateur calcule le quotient de la division de deux nombres entiers, décimaux ou flottants.

- `3 div 2, 4.5 div 6.7`
  `1.5, 0.671641791044776119`

Opérateurs `idiv` et `mod`

Ces opérateurs calculent respectivement le quotient et le reste de la division entière de deux entiers.

- `3 idiv 2, 3 mod 2`
  `2, 1`

`number abs(number x)`
retourne la valeur absolue d'un nombre entier ou flottant. La valeur retournée est du même type que la valeur passée en paramètre.

- `abs(-1), abs(2.3)`
  `1, 2.3`

`number floor(number x)`
retourne la valeur entière approchée par valeur inférieure d'un nombre décimal ou flottant.

- `floor(1), floor(2.5), floor(2,7), floor(-2.5)`
  `1, 2, 2, -3`

`number ceiling(number x)`
retourne la valeur entière approchée par valeur supérieure d'un nombre décimal ou flottant.

- `ceiling(1), ceiling(2.5), ceiling(2.3), ceiling(-2.5)`
  `1, 3, 3, -2`

`number round(number x)`
retourne la valeur entière approchée la plus proche d'un nombre décimal ou flottant. Si le paramètre est égal à n+1/2 pour un entier n, la valeur retournée est l'entier n+1.

- `round(1), round(2.4), round(2.5), round(-2.5)`
  `1, 2, 3, -2`

`number round-half-to-even(number x, xsd:integer? precision)`
retourne le multiple de $10^{-precision}$ le plus proche du nombre décimal ou flottant `x`. La valeur par défaut de `precision` est `0`. Dans ce cas, la fonction retourne l'entier le plus proche de `x`. Si `x` est à égale distance de deux multiples, c'est-à-dire si sa valeur est de la forme $(n+1/2)\times10^{-precision}$, la fonction retourne le multiple pair.

- `round-half-to-even(12.34,-1), round-half-to-even(12.34,1)`
  `10, 12.3`

- `round-half-to-even(2.5), round-half-to-even(3.5)`
  `2, 4`

`xsd:anyAtomicType min(xsd:anyAtomicType* list, xsd:string? col)`
retourne le minimum d'une liste de valeurs qui sont comparables pour l'ordre <. Le paramètre optionnel `col` spécifie la collation à utiliser pour comparer des chaînes de caractères.

- `min(1 to 6), min(('Hello', 'new', 'world'))`
  `1, 'Hello'`

`xsd:anyAtomicType max(xsd:anyAtomicType* list, xsd:string? col)`
retourne le maximum d'une liste de valeurs qui sont comparables pour l'ordre `lt`. Le paramètre optionnel `col` spécifie la collation à utiliser pour comparer des chaînes de caractères.

`number sum(xsd:anyAtomicType* list)`
retourne la somme d'une liste de nombres. Les valeurs qui ne sont pas des nombres sont converties au préalable en flottants avec `xsd:double`.

- `sum(1 to 6)`
  `21`

`number avg(xsd:anyAtomicType* list)`
retourne la moyenne d'une liste de nombres. Les valeurs qui ne sont pas des nombres sont converties au préalable en flottants avec `xsd:double`.

- `avg(1 to 6)`
  `3.5`

6.3.3. Chaînes de caractères

Les chaînes de caractères XPath contiennent, comme les documents XML, des caractères Unicode.

Une chaîne littérale est délimitée par une paire d'apostrophes `'''` ou une paire de guillemets `'"'`. Lorsqu'elle est délimitée par une paire d'apostrophes, les guillemets sont autorisés à l'intérieur et les apostrophes sont incluses en les doublant. Lorsqu'elle est, au contraire, délimitée par une paire de guillemets, les apostrophes sont autorisées à l'intérieur et les guillemets sont inclus en les doublant.

- `'Une chaîne'`
  donne `Une chaine`

- `'Une apostrophe '' et un guillemet "'`
  donne `Une apostrophe ' et un guillemet "`

- `"Une apostrophe ' et un guillemet """`
  donne `Une apostrophe ' et un guillemet "`

Les expressions XPath sont très souvent utilisées commme valeurs d'attributs d'un dialecte XML comme les schemas XML [Chapitre 5] ou XSLT [Chapitre 8]. Dans ce cas, les apostrophes ou les guillemets qui délimitent la valeur de l'attribut doivent être introduits avec les entités prédéfinies [Section 3.5.1.2] que cela soit comme délimiteur ou à l'intérieur d'une chaîne.

Il existe de nombreuses fonctions permettant de manipuler les chaînes de caractères. Dans les exemples ci-dessous, les chaînes apparaissant dans les valeurs des expressions sont écrites avec délimiteurs `'''` bien que ceux-ci ne fassent pas partie des chaînes.

Fonctions sur les chaînes de caractères

`xsd:integer string-length(xsd:string s)`

retourne la longueur de la chaîne de caractères, c'est-à-dire le nombre de caractères qui la composent.

- `string-length('Hello world')`
  `11`

`xsd:string concat(xsd:string s1, xsd:string s2, xsd:string s3, ...)`

retourne la concaténation des chaînes de caractères `s1`, `s2`, `s3`, …. Le nombre de paramètres de cette fonction est variable.

- `concat('Hello', 'new', 'world')`
  `'Hellonewworld'`

`xsd:string string-join(xsd:string* list, xsd:string sep)`

retourne la concaténation des chaînes de caractères de la liste en insérant la chaîne `sep` entre elles. Contrairement à la fonction `concat`, le nombre de paramètres de cette fonction est fixé à 2 mais le premier paramètre est une liste.

- `string-join(('Hello', 'new', 'world'), ' ')`
  `'Hello new world'`

`xsd:integer compare(xsd:string s1, xsd:string s2, xsd:anyURI? col)`

retourne la comparaison des deux chaînes de caractères `s1` et `s2` en utilisant la collation optionnelle identifiée par l'URI `col`. La valeur de retour est `-1`, `0` ou `1` suivant que `s1` est avant `s2` pour l'ordre lexicographique, égale à `s2` ou après `s2`. Si `col` est absente, la collation par défaut est utilisée. Celle-ci est basée sur les codes Unicode.

- `compare('Hello', 'world')`
  `-1`

- `compare('hello', 'World')`
  `1`

`xsd:boolean starts-with(xsd:string s, xsd:string prefix)`

retourne `true` si la chaîne `s` commence par la chaîne `prefix` et `false` sinon.

`xsd:boolean ends-with(xsd:string s, xsd:string suffix)`

retourne `true` si la chaîne `s` se termine par la chaîne `suffix` et `false` sinon.

`xsd:boolean contains(xsd:string s, xsd:string factor)`

retourne `true` si la chaîne `factor` apparaît comme sous-chaîne dans la chaîne `s` et `false` sinon.

- `contains('Hello', 'lo')`
  `true`

`xsd:boolean matches(xsd:string s, xsd:string regexp, xsd:string? flags)`

retourne `true` si une partie de la chaîne `s` est conforme à l'expression rationnelle `regexp` [Section 5.15] et `false` sinon. La chaîne optionnelle `flags` précise comment doit être effectuée l'opération.

- `matches('Hello world', '\w+')`
  `true`

- `matches('Hello world', '^\w+$')`
  `false`

- `matches('Hello', '^\w+$')`
  `true`

`xsd:string substring(xsd:string s, xsd:double start xsd:double length)`

retourne la sous-chaîne commençant à la position `start` et de longueur `length` ou moins si la fin de la chaîne `s` est atteinte. Les positions dans la chaîne `s1` sont numérotées à partir de `1`. Les paramètres `start` et `length` sont des flottants par compatibilité avec XPath 1.0. Ils sont convertis en entiers avec `round()`.

- `substring('Hello world', 3, 5)`
  `'llo w'`

- `substring('Hello world', 7, 10)`
  `'world'`

`xsd:string substring-before(xsd:string s1, xsd:string s2)`

retourne la sous-chaîne de `s1` avant la première occurrence de la chaîne `s2` dans `s1`.

- `substring-before('Hello world', 'o')`
  `'Hell'`
- `substring-before('Hello world', 'ol')`
  `''`

`xsd:string substring-after(xsd:string s1, xsd:string s2)`

retourne la sous-chaîne de `s1` après la première occurrence de la chaîne `s2` dans `s1`.

- `substring-after('Hello world', 'o')`
  `' world'`
- `substring-after('Hello world', 'ol')`
  `''`

`xsd:string translate(xsd:string s, xsd:string from, xsd:string to)`

retourne la chaîne obtenue en remplaçant dans la chaîne `s` chaque caractère de la chaîne `from` par le caractère à la même position dans la chaîne `to`. Si la chaîne `from` est plus longue que la chaîne `to`, les caractères de `from` sans caractères en correspondance dans `to` sont supprimés de la chaîne `s`.

- `translate('Hello world', 'lo', 'ru')`
  `'Herru wurrd'`
- `translate('01-44-27-45-19', '-', '')`
  `'0144274519'`

`xsd:string replace(xsd:string s, xsd:string regexp, xsd:string repl, xsd:string? flags)`

retourne la chaîne de obtenue en remplaçant dans la chaîne `s` l'occurrence de l'expression rationnelle `regexp` [Section 5.15] par la chaîne `repl`. L'expression `regexp` peut délimiter des blocs avec des paires de parenthèses `'('` et `')'` qui peuvent ensuite être utilisés dans la chaîne `repl` avec la syntaxe `$1`, `$2`, ….

- `replace('Hello world', 'o', 'u')`
  `'Hellu wurld'`
- `replace('="\&apos;\"=', '(["\\])', '\\$1')`
  `'=\"\\'\\\"='`
- `replace('(code,1234)', '\(([^,]*),([^\)]*)\)', '($2,$1)')`
  `'(1234,code)'`

`xsd:string* tokenize(xsd:string s, xsd:string regexp)`

retourne la liste des chaînes obtenues en découpant la chaîne `s` à chaque occurence de l'expression `regexp` [Section 5.15] qui ne peut pas contenir la chaîne vide.

- `tokenize('Hello new world', '\s+')`
  `('Hello', 'new', 'world')`
- `tokenize('Hello&#x20;&#x20;world', '\s')`
  `('Hello', '', 'world')`

`xsd:string normalize-space(xsd:string s)`

supprime les caractères d'espacement [Section 2.2.2] en début et en fin de chaîne et remplace chaque suite de caractères d'espacement consécutifs par un seul espace.

- `normalize-space(' Hello &x0A; world ')`
  `'Hello world'`

`xsd:string lower-case(xsd:string s)`

retourne la chaîne `s` mise en minuscule.

- `lower-case('Hello world')`
  `'hello world'`

`xsd:string upper-case(xsd:string s)`

retourne la chaîne `s` mise en majuscule.

- `upper-case('Hello world')`
  `'HELLO WORLD'`

`xsd:string codepoints-to-string(xsd:integer* list)`

convertit une liste de points de code [Section 2.2] en une chaîne de caractères.

- `codepoints-to-string((65, 8364, 48))`
  `'A€0'`

`xsd:integer* string-to-codepoints(xsd:string s)`

convertit une chaîne de caractères en une liste de points de code [Section 2.2].

- `string-to-codepoints('A&#8364;0')`
  `(65, 8364, 48)`

`xsd:string normalize-unicode(xsd:string s, xsd:string? norm)`

retourne la normalisation [Section 2.2.6] de la chaîne `s` avec la normalisation spécifiée par la chaîne `norm`. Cette dernière peut prendre les valeurs `NFC`, `NFD`, `NFKC` et `NFKD` si le processeur implémente chacune de ces normalisations. Si `norm` est absente, la normalisation C (`NFC`) par défaut et toujours implémentée est utilisée.

- `string-to-codepoints(normalize-unicode(codepoints-to-string((105, 776))))`
  `(239)`

6.3.4. Expressions rationnelles

Quelques fonctions XPath comme `matches()`, `replace()` et `tokenize()` prennent en paramètre une expression rationnelle. La syntaxe de ces expressions est identique à celle des expressions rationnelles des schémas [Section 5.15] avec seulement quelques différences.

La principale différence avec les expressions rationnelles des schémas est l'ajout des deux ancres `'^'` et `'$'`. Ces deux caractères deviennent spéciaux. Ils désignent la chaîne vide placée, respectivement, au début et à la fin de la chaîne. Ces deux caractères sont utiles car la fonction `matches()` retourne `true` dès qu'un fragment de la chaîne est conforme à l'expression rationnelle. Pour forcer la chaîne, prise dans son intégralité, à être conforme, il faut ajouter les caractères `'^'` et `'$'` au début et à la fin de l'expression rationnelle.

- `matches('Hello world', '\w+')`
  donne `true` car les fragments `Hello` ou `world` sont conformes à l'expression `\w+`.

- `matches('Hello world', '^\w+$')`
  donne `false` car la chaîne contient un espace.

Le fonctionnement des ancres `'^'` et `'$'` est modifié par le modificateur `'m'`.

6.3.4.1. Modificateurs

Les fonctions `matches()` et `replace()` prennent un dernier paramètre optionnel `flag` qui modifie leur comportement. Ce paramètre doit être une chaîne de caractères pouvant contenir les caractères `'i'`, `'m'` et `'s'` dans n'importe quel ordre. Chacun de ces caractères joue le rôle d'un *modificateur* qui influence un des aspects du fonctionnement de ces fonctions.

Le caractère `'i'` indique que la chaîne est comparée à l'expression sans tenir compte de la casse des lettres. Ceci signifie que les lettres minuscules et majuscules ne sont plus distinguées.

- `matches('Hello world', 'hello')`
  donne `false`.

- `matches('Hello world', 'hello', 'i')`
  donne `true`.

Le caractère `'m'` modifie la signification des ancres `'^'` et `'$'`. Il indique que la chaîne est comparée à l'expression en mode *multi-ligne*. Dans ce mode, la chaîne est vue comme plusieurs chaînes obtenues en découpant à chaque saut de ligne marqué par le caractère `U+0A` [Section 2.2.2]. Ceci signifie que le caractère `'^'` désigne alors la chaîne vide placée au début de la chaîne ou après un saut de ligne et que le caractère `'$'` désigne alors la chaîne vide placée à la fin de la chaîne ou avant un saut de ligne.

- `matches('Hello
world', '^Hello$')`
  donne `false`.

- `matches('Hello
world', '^Hello$', 'm')`
  donne `true`.

Le caractère `'s'` modifie la signification du caractère spécial `'.'`. Il désigne normalement tout caractère autre qu'un saut de ligne. Avec le modificateur `'s'`, il désigne tout caractère, y compris le saut de ligne `U+0A` [Section 2.2.2].

- `matches('Hello
world', '^.*$')`
  donne `false`.

- `matches('Hello
world', '^.*$', 's')`
  donne `true`.

6.4. Listes

La liste est la structure de données fondamentale de XPath. Il existe plusieurs opérateurs permettant de construire et de manipuler des listes. La restriction importante des listes XPath est qu'elles ne peuvent pas être imbriquées. Une liste XPath ne peut pas contenir d'autres listes. Elle peut uniquement contenir des nœuds et des valeurs atomiques. Ainsi, l'expression `((1,2),(3,4,5))` ne donne pas une liste contenant deux listes de deux et trois entiers. Elle donne la liste de cinq entiers que donne également l'expression `(1,2,3,4,5)`.

6.4.1. Constructeurs

L'opérateur essentiel de construction de listes est `','` (virgule) qui permet de concaténer, c'est-à-dire mettre bout à bout, des listes. Il est aussi bien utilisé pour écrire des listes constantes comme `(1,2,3,4,5)` que pour concaténer les résultats d'autres expressions. Contrairement aux opérateurs de chemins [Section 6.2], l'opérateur `','` ne réordonne pas les éléments des listes et ne supprime pas les doublons. Le résultat de l'expression *`expr1,expr2`* est la nouvelle liste formée des valeurs du résultat de *`expr1`* suivis des valeurs du résultat l'expression *`expr2`*. Si une valeur apparaît dans les deux résultats, elle a plusieurs occurrences dans le résultat final. Par exemple, le résultat de l'expression `(1,2),(1,3)` est bien la liste `(1,2,1,3)` avec deux occurrences de l'entier `1`.

Le fait qu'une valeur soit assimilée à la liste (de longueur 1) contenant cette valeur simplifie l'écriture des expressions. Ainsi les deux expressions `(1),(2)` et `1,2` sont équivalentes. Cette identification entre une valeur et une liste ne crée pas d'ambiguïté car les listes XPath ne peuvent pas être imbriquées. Il n'y a pas de différence entre les deux expressions `((1),(2))` et `(1,2)`.

`title, author`
: donne la liste des enfants de nom `title` puis des enfants de nom `author` du nœud courant.

`1, 'Two', 3.14, true()`
: donne la liste `(1, 'Two', 3.14, true)` constituée d'un entier, d'une chaîne de caractères, d'un nombre flottant et d'une valeur booléenne.

`(1, 2), 3, (4, (5))`
: donne la liste `(1, 2, 3, 4, 5)` sans imbrication.

`(1, 2), 2, 1, 2`
: donne la liste `(1, 2, 2, 1, 2)` avec répétitions.

L'opérateur `to` permet de créer une liste contenant une suite d'entiers consécutifs. L'expression *`n1 to n2`* donne la liste *`n1,n1+1,n1+2,…,n2-1,n2`* des entiers de *`n1`* à *`n2`* compris. Cet opérateur est surtout utile avec l'opérateur `for` [Section 6.6.2] pour itérer sur une liste d'entiers.

`1 to 5`
: donne la liste `(1, 2, 3, 4, 5)`

`1, 2 to 4, 5`
: donne la liste `(1, 2, 3, 4, 5)`

6.4.2. Filtres

Un filtre permet de sélectionner dans une liste les objets qui satisfont une condition. Un filtre se présente comme une expression entre des crochets `'['` et `']'` placée après la liste à filtrer.

Une expression de la forme *`expr1`*`[`*`expr2`*`]` est évaluée de la façon suivante. L'expression *`expr1`* est d'abord évaluée pour donner une liste *`l`* d'objets. Pour chaque objet *`o`* de la liste *`l`*, l'expression *`expr2`* est évaluée en modifiant, au préalable, le focus [Section 6.1.5.2] de la manière suivante. L'objet courant est fixé à l'objet *`o`*, la position du contexte est fixée à la position de l'objet *`o`* dans la liste *`l`* et la taille du contexte est fixée à la taille de *`l`*. Le résultat de cette évaluation est ensuite converti en une valeur booléenne en utilisant les règles de conversion [Section 6.3.1]. Le résultat final de l'évaluation de *`expr1`*`[`*`expr2`*`]` est la liste des objets de *`l`* pour lesquels *`expr2`* s'est évaluée en la valeur `true`. Les objets sélectionnés restent bien sûr dans l'ordre de la liste *`l`*. La liste résultat est en fait construite en supprimant de la liste *`l`* les objets pour lesquels *`expr2`* s'évalue en `false`.

Lors de l'évaluation de l'expression suivante, l'objet courant qui est retourné par `'.'` prend les valeurs successives `1`, `2`, `3`, `4` et `5`. Seules les valeurs paires satisfont la condition et sont conservées.

`(1 to 5)[. mod 2 = 0]`

: donne la liste `(2,4)` des entiers pairs de la liste

Les filtres sont beaucoup utilisés dans les expression de chemin [Section 6.2.2] pour sélectionner des nœuds.

Plusieurs conditions peuvent être combinées à l'aide des opérateurs booléens `and` et `or`. Les filtres peuvent aussi être enchaînés en les mettant l'un après l'autre.

`text[position() > 1 and position() < 4]`
: donne les enfants `text` du nœud courant aux positions 2 et 3

`text[position() > 1][position() < 4]`
: donne les enfants `text` du nœud courant aux positions 2, 3 et 4

Le second exemple montre que les filtres peuvent être enchaînés. Il ne donne pas le même résultat que le premier exemple car les positions retournées par la fonction `position()` du second filtre sont celles dans la liste obtenue après le premier filtre. Comme le premier enfant `text` a été supprimé, il y a un décalage d'une unité.

Les expressions de chemins retournent des listes de nœuds triés dans l'ordre du document et sans doublon. Au contraire, l'opérateur `','` ne supprime pas les doublons.

`p[@align or @type]`

donne la liste des enfants `p` du nœud courant ayant un attribut `align` ou `type`.

`p[@align], p[@type]`

donne la liste des enfants `p` du nœud courant ayant un attribut `align` suivis des enfants `p` ayant un attribut `type`. Si un nœud `p` possède les deux attributs, il apparaît deux fois dans la liste.

6.4.3. Fonctions

Il existe des fonctions XPath permettant de manipuler les listes.

`xsd:integer count(item()* l)`

retourne la longueur de la liste `l`, c'est-à-dire le nombre de nœuds ou valeurs atomiques qui la composent.

- `count('Hello world', 1, 2, 3)`
  `4`

`xsd:boolean empty(item()* l)`

retourne `true` si la liste `l` est vide et `false` sinon.

- `empty(1 to 5)`
  `false`

`xsd:boolean exists(item()* l)`

retourne `true` si la liste `l` est non vide et `false` sinon.

- `exists(1 to 5)`
  `true`

`item()* distinct-values(item()* l)`

retourne une liste des valeurs distinctes de la liste `l` en supprimant les valeurs égales pour l'opérateur `eq`.

- `distinct-values((1, 'Hello', 0, 1, 'World'))`
  `(1, 'Hello', 0, 'World')`

`xsd:integer* index-of(item()* l, item() value)`

retourne la liste des positions de la valeur `value` dans la liste `l`.

- `index-of((1, 'Hello', 0, 1, 'World'), 1)`
  `(1, 4)`

`item()* subsequence(item()* l, xsd:double start xsd:double length)`

retourne la sous-liste commençant à la position `start` et de longueur `length` ou moins si la fin de la liste `l` est atteinte.

- `subsequence((1, 'Hello', 0, 1, 'World'), 2, 3)`
  `('Hello', 0, 1)`

```
item()* remove(item()* l, xsd:integer pos)
```

retourne la liste obtenue en supprimant de la liste `l` la valeur à la position `pos`.

- `remove(1 to 5, 3)`
  `(1, 2, 4, 5)`

```
item()* insert-before(item()* l1, xsd:integer pos, item()* l2)
```

retourne la liste obtenue en insérant la liste `l2` dans la liste `l1` à la position `pos`.

- `insert-before(1 to 5, 3, 1 to 3)`
  `(1, 2, 1, 2, 3, 3, 4, 5)`

```
item()* reverse(item()* l)
```

retourne la liste obtenue en inversant l'ordre

- `reverse(1 to 5)`
  `(5, 4, 3, 2, 1)`

6.5. Comparaisons

Les comparaisons sont un aspect important mais délicat de XPath. Elles jouent un rôle important en XPath car elles permettent d'affiner la sélection des nœuds en prenant en compte leurs contenus. Il est, par exemple, possible de sélectionner des éléments d'un document dont la valeur d'un attribut satisfait une condition comme dans les expressions `item[@type='free']` et `list[@length < 5]`. Les comparaisons sont aussi délicates à utiliser car leur sémantique est source de pièges conduisant aisément à des programmes erronés.

Il existe deux types d'opérateurs pour effectuer des comparaisons entre valeurs. Les premiers opérateurs dits *généraux* datent de la première version de XPath. Ils permettent de comparer deux valeurs quelconques, y compris des listes, avec des résultats parfois inattendus. Les seconds opérateurs ont été introduits avec la version 2.0 de XPath. Ils autorisent uniquement les comparaisons entres les valeurs atomiques de même type. Ils sont plus restrictifs mais leur comportement est beaucoup plus prévisible.

Il existe aussi l'opérateur `is` et les deux opérateurs `<<` et `>>` permettant de tester l'égalité et l'ordre de nœuds dans le document.

6.5.1. Opérateurs de comparaisons atomiques

Les opérateurs de comparaison pour les valeurs atomiques sont les opérateurs `eq`, `ne`, `lt`, `le`, `gt` et `ge`. Ils permettent respectivement de tester l'égalité, la non-égalité, l'ordre strict et l'ordre large (avec égalité) entre deux valeurs de même type. L'ordre pour les entiers et les flottants est l'ordre naturel alors que l'ordre pour les chaînes de caractères est l'ordre lexicographique du dictionnaire. Cet ordre lexicographique prend en compte les collations.

```
2 ne 3
```
donne `true`

```
2 lt 3
```
donne `true`

```
'chaine' ne 'string'
```
donne `true`

```
'chaine' lt 'string'
```
donne `true`

Ces opérateurs de comparaison exigent que leurs deux paramètres soient du même type. Les constantes présentes dans le programme sont automatiquement du bon type. En revanche, les contenus des éléments et les valeurs

des attibuts doivent être convertis explicitement à l'aide des fonctions de conversion lorsque le document est traité indépendamment d'un schéma. Pour tester si la valeur d'un attribut `pos` vaut la valeur `1`, il est nécessaire d'écrire `xsd:integer(@pos) eq 1` où la valeur de l'attribut `pos` est convertie en entier par la fonction `xsd:integer`. Les opérateurs généraux de comparaison évitent ces conversions fastidieuses car ils effectuent eux-mêmes des conversions implicites.

La contrainte d'égalité des types des valeurs n'est pas stricte. Il est possible de comparer une valeur d'un type avec une valeur d'un type obtenu par restriction [Section 5.9]. Il est également possible de comparer des valeurs des différents types numériques `xsd:integer`, `xsd:decimal`, `xsd:float` et `xsd:double`.

6.5.2. Opérateurs généraux de comparaisons

Les opérateurs généraux de comparaison sont les opérateurs `'='`, `'!='`, `'<'`, `'<='`, `'>'` et `'>='`. Ils permettent respectivement de tester l'égalité, la non-égalité, l'ordre strict et l'ordre large de deux valeurs de types quelconques.

Les objets à comparer sont d'abord atomisés [Section 6.1.1.1], ce qui signifie que les nœuds présents dans les listes sont remplacés par leur valeur pour obtenir uniquement des valeurs atomiques. Ensuite, la comparaison est effectuée de façons différentes suivant que les objets sont des listes composées d'une seule valeur (considérées alors comme une simple valeur) ou de plusieurs valeurs.

6.5.2.1. Comparaisons de valeurs atomiques

La façon de réaliser une comparaison entre deux valeurs atomiques dépend du type [Section 6.1.4] de ces deux valeurs. Suivant les types de celles-ci, certaines conversions sont effectuées au préalable puis elles sont comparées avec l'opérateur de comparaison atomique correspondant donné par la table suivante.

Opérateur général	Opérateur atomique
=	eq
!=	ne
<	lt
<=	le
>	gt
>=	ge

Lorsque les deux valeurs sont de type `xsd:untypedAtomic`, celle-ci sont comparées comme des chaînes de caractères. Lorsqu'une seule des deux valeurs est de type `xsd:untypedAtomic`, celle-ci est convertie dans le type de l'autre valeur avant de les comparer. Quand le type de l'autre valeur est un type numérique, la valeur de type `xsd:untypedAtomic` est convertie en une valeur de type `xsd:double` plutôt que dans le type de l'autre valeur. Ceci évite qu'une valeur décimale comme `1.2` soit convertie en entier avant d'être comparée à la valeur `1`. Si les deux valeurs sont de types incompatibles, la comparaison échoue et provoque une erreur.

Pour illustrer ces comparaisons, on considère le petit document suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<list>
  <item type="1">1</item>
  <item type="01">2</item>
  <item type="03">3</item>
  <item type="1.2">4</item>
  <!-- Erreur car 'str' ne peut être convertie en nombre flottant -->
  <item type="str">5</item>
</list>
```

Les expressions suivantes sont évaluées sur le document précédent en supposant, à chaque fois, que le nœud courant est l'élément racine `list` du document. Pour chacune des expressions, le résultat est une liste d'enfants

`item` de l'élément `list`. Il est décrit en donnant les positions de ces éléments `item` sélectionnés. Le résultat `item[1], item[2]` de la première expression signifie, par exemple, qu'elle sélectionne le premier et le deuxième enfants `item`.

`item[@type=1]`
: donne `item[1], item[2]` car la valeur de l'attribut `type` est convertie en nombre flottant avant d'être comparée à `1`. La valeur `'01'` est donc convertie en `'1'`.

`item[@type='1']`
: donne `item[1]` car la valeur de l'attribut `type` est convertie en chaîne de caractères avant d'être comparée à `'1'`.

`item[@type=.]`
: donne `item[1]` car la valeur de l'attribut `type` et le contenu de l'élément `item` sont convertis en chaînes de caractères avant d'être comparés.

`item[@type=1.2]`
: donne `item[4]` car la valeur de l'attribut `type` est convertie en nombre flottant avant d'être comparée à `1.2`.

`item[xsd:double(.)=xsd:double(@type)]`
: donne `item[1], item[3]` car la valeur de l'attribut `type` et le contenu de l'élément `item` sont convertis en nombres flottants avant d'être comparés.

Il faut faire attention au fait que les comparaisons peuvent échouer et provoquer des erreurs lorsque les types ne sont pas compatibles. Ce problème renforce l'intérêt de la validation des documents avant de les traiter.

6.5.2.2. Comparaisons de listes

Les comparaisons entre listes ont une sémantique très particulière qui est parfois pratique mais souvent contraire à l'intuition. Ce cas s'applique dès qu'un des deux objets comparés est une liste puisqu'une valeur est identifiée à une liste de longueur 1. La comparaison entre deux listes `l1` et `l2` pour un des opérateurs =, `!=`, <, <=, > et >= est effectuée de la façon suivante. Chaque valeur de la liste `l1` est comparée avec chaque valeur de la liste `l2` pour le même opérateur, comme décrit à la section précédente. Le résultat global est égal à `true` dès qu'au moins une des comparaisons donne la valeur `true`. Il est égal à `false` sinon. Cette stratégie implique en particulier que le résultat est `false` dès qu'une des deux listes est vide quel que soit l'opérateur de comparaison.

`() = ()`
: donne `false` car une des deux listes est vide.

`() != ()`
: donne `false` car une des deux listes est vide.

L'exemple précédent montre que l'opérateur `!=` n'est pas la négation de l'opérateur =, ce qui n'est pas très intuitif.

`() != (1)`
: donne `false` car une des deux listes est vide.

`(1) = (1)`
: donne `true` car la valeur `1` de la liste `l1` est égale à la valeur `1` de la liste `l2`.

`(1) != (1)`
: donne `false` car l'unique valeur `1` de la liste `l1` n'est pas différente de l'unique valeur `1` de la liste `l2`.

`(1) = (1, 2)`
: donne `true` car la valeur `1` de la liste `l1` est égale à la valeur `1` de la liste `l2`.

`(1) != (1, 2)`
: donne `true` car la valeur `1` de la liste `l1` est n'est pas égale à la valeur `2` de la liste `l2`.

Dès que la comparaison de deux valeurs des listes `l1` et `l2` échoue, la comparaison globale entre les listes `l1` et `l2` échoue également. L'ordre des comparaisons entre les valeurs des deux listes est laissé libre par XPath et

chaque logiciel peut les effectuer dans l'ordre qui lui convient. Lorsqu'une de ces comparaisons donne la valeur `true` et qu'une autre de ces comparaisons échoue, le résultat de la comparaison des deux listes est imprévisible. Il est égal à `true` si une comparaison donnant `true` est effectuée avant toute comparaison qui échoue mais la comparaison globale échoue dans le cas contraire.

La sémantique des comparaisons de listes permet d'écrire simplement certains tests. Pour savoir si une valeur contenue, par exemple, dans une variable `$n` est égale à une des valeurs de la liste `(2, 3, 5, 7)`, il suffit d'écrire `$n = (2, 3, 5, 7)`.

6.5.3. Opérateurs de comparaisons de nœuds

L'opérateur `is` compare deux nœuds et retourne `true` s'il s'agit du même nœud. C'est donc plus un test d'identité que d'égalité. Il s'apparente plus à l'opérateur `'=='` de Java qu'à la méthode `equals` du même langage.

Les deux opérateurs `'<<'` et `'>>'` permettent de tester si un nœud se trouve avant ou après un autre nœud dans l'ordre du document [Section 6.1.2].

Pour illustrer ces trois opérateurs, on considère le document minimaliste suivant.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<list length="2">
  <item type="1">1</item>
  <item type="1">2</item>
</list>
```

Les expressions suivantes sont évaluées sur le document précédent en supposant, à chaque fois, que le nœud courant est la racine du document.

`list is list/item/parent::*`
: donne `true` car il s'agit du même nœud qui est l'élément racine du document.

`list/item[1]/@type is list/item[2]/@type`
: donne `false` car il s'agit de deux nœuds différents bien que les deux attributs aient même nom et même valeur.

`list << list/item[1]`
: donne `true` car le père est placé avant ses enfants dans l'ordre du document.

`list/@length << list/item[1]`
: donne `true` car les attributs sont placés avant les éléments enfants.

`list/item[1] << list/item[2]`
: donne `true`.

Un exemple pertinent d'utilisation de l'opérateur `is` est donné par la feuille de style avec indexation [Section 8.12] pour regrouper les éditeurs et supprimer leurs doublons dans le document `bibliography.xml`.

6.6. Structures de contrôle

Des structures de contrôle sont apparues avec la version 2.0 de XPath. L'objectif n'est pas d'écrire des programmes complets en XPath. Il s'agit plutôt de remplacer par des expressions XPath simples des constructions plus lourdes des langages hôtes. Il est, par exemple, plus concis d'écrire le fragment XSLT [Chapitre 8] suivant avec un test dans l'expression XPath. Dans l'exemple suivant, l'élément `a` contient une expression XPath en attribut [Section 8.6.3] dont le résultat devient la valeur de son attribut `id`. Cette expression retourne la valeur de l'attribut `xml:id` de l'élément courant si cet attribut existe ou génère un nouvel identifiant sinon.

```
<a id="{if (@xml:id) then @xml:id else generate-id()}"/>
```

Elle est équivalente au fragment XSLT suivant où le test est réalisé par un élément `xsl:choose` [Section 8.7.2].

```
<a>
  <xsl:choose>
    <xsl:when test="@xml:id">
      <xsl:attribute name="id" select="@xml:id"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:attribute name="id" select="generate-id()"/>
    </xsl:otherwise>
  </xsl:choose>
</a>
```

6.6.1. Conditionnelle

L'opérateur `if` permet d'effectuer un test. Sa syntaxe est la suivante.

`if (`*`test`*`) then `*`expr1`*` else `*`expr2`*

La sémantique est celle de `if` dans tous les langages de programmation. L'expression *`test`* est évaluée et le résultat est converti en une valeur booléenne [Section 6.3.1]. Si cette valeur booléenne est `true`, l'expression *`expr1`* est évaluée et le résultat est celui de l'expression globale. Sinon, l'expression *`expr2`* est évaluée et le résultat est celui de l'expression globale.

La partie `else` *`expr2`* est obligatoire et ne peut pas être omise. Lorsque cette seconde partie est inutile, on met simplement `else ()`.

`if (contains($url, ':')) then substring-before($url, ':') else ''`
: l'évaluation de cette expression retourne le protocole d'une URL placé avant le caractère `':'` si celle-ci en contient un.

L'opérateur `if` est parfois remplacé par un filtre [Section 6.4.2]. L'expression `if (@xml:id) then @xml:id else generate-id()` est en effet équivalente à l'expression plus concise `(@xml:id, generate-id())[1]`.

6.6.2. Itération

L'opérateur `for` permet de parcourir des listes pour construire une nouvelle liste. Il ne s'agit pas d'une structure de contrôle pour des itérations quelconques comme le `for` ou le `while` des langages C ou Java. Il s'apparente plus à l'opérateur `map` des langages fonctionnels comme ML qui permet d'appliquer une fonction à chacun des objets d'une liste.

La syntaxe de l'opérateur `for` est la suivante où *`var`* est une variable et *`expr1`* et *`expr2`* sont deux expressions. La variable *`var`* peut uniquement apparaître dans l'expression *`expr2`*.

`for `*`var`*` in `*`expr1`*` return `*`expr2`*

L'évaluation d'une telle expression est réalisée de la façon suivante. L'expression *`expr1`* est d'abord évaluée pour donner une liste de valeurs. Pour chacune de ces valeurs, celle-ci est affectée à la variable *`var`* et l'expression *`expr2`* est évaluée. Le résultat global est la liste obtenue en concaténant les listes obtenues pour chacune des évaluations de l'expression *`expr2`*.

Le résultat de l'expression *`expr2`* est une liste qui peut donc contribuer à plusieurs valeurs de la liste finale. Si, au contraire, ce résultat est la liste vide, il n'y a aucune contribution à la liste finale.

La variable *`var`* introduite par l'opérateur `for` est une variable muette. Sa portée est réduite à l'expression *`expr2`* après le mot clé `return`. Aucune valeur ne peut lui être affectée directement.

`for $i in 1 to 5 return $i * $i`
: l'évaluation de cette expression donne la liste `(1,4,9,16,25)` des cinq premiers carrés

`for $i in 1 to 3 return (2 * $i, 2 * $i + 1)`
l'évaluation de cette expression donne la liste `(2,3,4,5,6,7)` qui est la concaténation des trois listes `(2,3)`, `(4,5)` et `(6,7)`

`for $i in 1 to 5 return if ($i mod 2) then () else $i * $i`
l'évaluation de cette expression donne la liste `(4,16)` des carrés des deux nombres pairs 2 et 4 pour lesquels `$i mod 2` donne `0`

Il est possible d'imbriquer plusieurs opérateurs `for`. Il y a d'ailleurs une syntaxe étendue qui permet une écriture concise de ces itérations imbriquées. Cette syntaxe prend la forme suivante.

`for` *`var-1`* `in` *`expr-1`*`, …, ` *`var-N`* `in` *`expr-N`* `return` *`expr`*

Cette expression est, en fait, équivalente à l'expression suivante écrite avec la première syntaxe.

`for` *`var-1`* `in` *`expr-1`* `return for … return for` *`var-N`* `in` *`expr-N`* `return` *`expr`*

`for $i in 0 to 2, $j in 0 to 2 return $i * 3 + $j`
l'évaluation de cette expression donne la liste `(0,1,2,3,4,5,6,7,8)` qui est la concaténation des trois listes `(0,1,2)`, `(3,4,5)` et `(6,7,8)`

6.6.3. Quantification existentielle

L'opérateur `some` permet de vérifier qu'au moins un des objets d'une liste satisfait une condition. Sa syntaxe est la suivante.

`some` *`var`* `in` *`expr1`* `satisfies` *`expr2`*

L'évaluation d'une telle expression est réalisée de la façon suivante. L'expression *`expr1`* est d'abord évaluée pour donner une liste de valeurs. Pour chacune de ces valeurs, celle-ci est affectée à la variable *`var`* et l'expression *`expr2`* est évaluée. Le résultat de l'expression globale est `true` si au moins une des évaluations de *`expr2`* donne une valeur qui se convertit en `true`. Il est égal à `false` sinon.

`some $i in 0 to 5 satisfies $i > 4`
l'évaluation de cette expression donne la valeur `true` car la condition `$i > 4` est satisfaite pour `$i = 5`

6.6.4. Quantification universelle

L'opérateur `every` permet de vérifier que tous les objets d'une liste satisfont une condition. Sa syntaxe est la suivante.

`every` *`var`* `in` *`expr1`* `satisfies` *`expr2`*

L'évaluation d'une telle expression est réalisée de la façon suivante. L'expression *`expr1`* est d'abord évaluée pour donner une liste de valeurs. Pour chacune de ces valeurs, celle-ci est affectée à la variable *`var`* et l'expression *`expr2`* est évaluée. Le résultat de l'expression globale est `true` si toutes les évaluations de *`expr2`* donnent une valeur qui se convertit en `true`. Il est égal à `false` sinon.

`every $i in 0 to 5 satisfies $i > 4`
l'évaluation de cette expression donne la valeur `false` car la condition `$i > 4` n'est pas satisfaite pour `$i = 0` ou `$i = 1`

`every $i in 0 to 5 satisfies some $j in 0 to 5 satisfies $i + $j eq 5`
l'évaluation de cette expression donne la valeur `true`

6.7. 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::node()`.

`@`

est l'abréviation de `attribute::`.

`//`

est l'abréviation de `/descendant-or-self::node()/`.

`[`*`n`*`]`

est l'abréviation de `[position()=`*`n`*`]` où *n* est un entier.

En XPath 1.0, la formule `'.'` était une abréviation pour `self::node()` et elle désignait toujours un nœud. En XPath 2.0, elle désigne l'objet courant qui peut être une valeur atomique ou un nœud.

Il faut remarquer que `'//'` n'est pas une abbréviation de `descendant-or-self::`. Les deux expressions `//section[1]` et `descendant-or-self::section[1]` donnent des résultats différents. La première expression donne la liste des éléments `section` qui sont le premier enfant `section` de leur parent alors que la seconde donne le premier élément `section` du document.

6.8. Motifs

Les *motifs* XPath sont des expressions XPath simplifiées permettant de sélectionner de façon simple des nœuds dans un document. Ils sont beaucoup utilisés par XSLT [Chapitre 8], d'abord pour spécifier des nœuds auxquels s'applique une règle [Section 8.5.1] mais aussi pour la numérotation [Section 8.6.11] et l'indexation [Section 8.12]. Ils répondent à une exigence d'efficacité. Le cœur de XSLT est l'application de règles à des nœuds du document source. Il est important de déterminer rapidement quelles sont les règles susceptibles de s'appliquer à un nœud. Dans ce but, les motifs n'utilisent qu'un sous-ensemble très limité de XPath. De surcroît, les motifs sont évalués de façon particulière, sans utiliser le nœud courant.

Les seuls opérateurs utilisés dans les motifs sont les opérateurs `'|'`, `'/'` et `'//'` et leur imbrication est très restreinte. Un motif prend nécessairement la forme suivante

```
expr-1 | expr-2 | ... | expr-N
```

où les expressions *`expr-1`*, ..., *`expr-N`* ne contiennent que des opérateurs `'/'` et `'//'`. En outre, les seuls axes possibles dans les motifs sont les deux axes `child::`, `attribute::`. L'axe `descendant-or-self::` est implicite dans l'utilisation de l'opérateur `'//'` mais c'est la seule façon de l'utiliser dans un motif. En revanche, les tests portant sur les types de nœuds comme `text()` ou `comment()` et les filtres sont autorisés dans les motifs.

L'évaluation d'un motif XPath est réalisée de façon différente d'une expression XPath classique. Celle-ci ne nécessite pas de nœud courant. Un motif est toujours évalué à partir de la racine du document. De plus, un motif commence toujours implicitement par l'opérateur `'//'`. Le motif `section` est, en fait, évalué comme l'expression `//section`. Il sélectionne tous les éléments `section` du document. Des exemples de motifs avec les nœuds qu'ils sélectionnent sont donnés ci-dessous.

`/`

la racine du document

`*`

tous les éléments

`section`
tous les éléments `section`

`chapter|section`
tous les éléments `chapter` et `section`

`chapter/section`
tous les éléments `section` enfants d'un élément `chapter`

`chapter//section`
tous les éléments `section` descendants d'un élément `chapter`

`section[1]`
tous les éléments `section` qui sont le premier enfant de leur parent

`section[@lang]`
tous les éléments `section` ayant un attribut `lang`

`section[@lang='fr']`
tous les éléments `section` dont l'attribut `lang` a la valeur `fr`

`@*`

tous les attributs

`section/@lang`
tous les attributs `lang` des éléments `section`

`text()`

tous les nœuds textuels

6.9. Utilisation interactive de xmllint

Le logiciel `xmllint` possède un mode interactif avec lequel il est possible de se déplacer dans un document XML comme s'il s'agissait d'un système de fichiers. Les nœuds du documents XML sont assimilés à des répertoires (dossiers) et fichiers. Ce mode interactif s'apparente à un interpréteur de commandes (*shell*) dont le répertoire de travail devient le nœud courant. Les commandes permettent de se déplacer dans le document en changeant le nœud courant et en évaluant des expressions XPath par rapport à ce nœud courant. L'intérêt de ce mode interactif réside bien sûr dans la possibilité d'expérimenter facilement les expressions XPath.

Le mode interactif de `xmllint` est activé avec l'option `--shell`. Les principales commandes disponibles sont alors les suivantes.

`help`
affiche un récapitulatif des commandes disponibles.

`cd` *`path`*
change le nœud courant en le nœud retourné par l'expression XPath *`path`*.

`pwd`
affiche le nœud courant.

`xpath` *`path`*
affiche le résultat de l'évalution de l'expression XPath *`path`*.

`cat [`*`node`*`]`
affiche le contenu du nœud *`node`*.

`dir [node]`
affiche les informations relatives au nœud *`node`*.

`grep string`
affiche les occurrences de la chaîne *`string`* dans le contenu du nœud courant.

Une session d'expérimentation du mode interactif de `xmllint` avec le fichier de bibliographie `bibliography.xml` est présentée ci-dessous.

```
bash $ xmllint --shell bibliography.xml
/ > grep XML
/bibliography/book[1]/title : t--   27 XML langage et applications
/bibliography/book[2]/title : t--   14 XML by Example
/bibliography/book[5]/title : t--   46 Modélisation et manipulation ...
/bibliography/book[6]/title : t--   25 XML Schema et XML Infoset
/bibliography/book[7]/title : ta-    3 XML
/ > xpath bibliography/book[@lang='en']
Object is a Node Set :
Set contains 2 nodes:
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 >
```

6.10. Récapitulatif des opérateurs 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`

Opérateur	Action	Syntaxe	Exemples
/	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

Chapitre 7. Schematron

7.1. Introduction

Schematron est un autre formalisme permettant de spécifier la structure d'un document XML. C'est donc au départ une alternative aux schémas [Chapitre 5] mais il est plutôt complémentaire des schémas. Ce formalisme n'est pas très adapté pour définir l'imbrication des éléments comme le font les schémas en donnant une grammaire. En revanche, il permet d'imposer des contraintes sur le document qu'il est difficile, voire impossible, d'exprimer avec les schémas. Il est fréquent d'utiliser les deux formalismes conjointement. Un schéma définit la structure globale du document et un schematron la complète en ajoutant des contraintes supplémentaires que doit satisfaire le document pour être valide. Il existe d'ailleurs des mécanismes permettant d'inclure un schematron au sein d'un schéma.

Schematron est basé sur XPath [Chapitre 6]. Un schematron est constitué de règles écrites avec des expressions XPath qui expriment la présence ou l'absence de motifs dans le document. Ce mécanisme rend schematron très puissant puisqu'il est possible de mettre en relation des éléments et des attributs qui sont éloignés dans le document. Schematron reste cependant très simple car le vocabulaire est restreint. L'écriture d'un schematron requiert l'utilisation de seulement quelques éléments.

7.2. Premier exemple

On donne ci-dessous un premier exemple de schematron très simple. Il contient une seule règle qui s'applique aux éléments `list` du document. Pour chacun de ces éléments, la valeur de l'attribut `length` doit être égale au nombre d'enfants.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">❶
  <sch:title>Premier exemple de schematron</sch:title>❷
  <sch:pattern>
    <sch:rule context="list">❸
      <sch:assert test="@length = count(*)">❹
        L'attribut length doit être égal au nombre d'enfants.❺
      </sch:assert>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

❶ Élément racine `sch:schema` du schematron avec la déclaration de l'espace de noms des schematrons.
❷ Titre informatif du schematron.
❸ Règle s'appliquant à tous les éléments `list` du document cible.
❹ Contrainte proprement dite exprimée par une expression XPath.
❺ Texte utilisé pour la fabrication du rapport.

Si le schematron précédent est appliqué au document XML suivant, le rapport va contenir le texte `"L'attribut ... d'éléments."` car la contrainte entre la valeur de l'attribut `length` et le nombre d'enfants de l'élément `list` n'est pas satisfaite par le deuxième élément `list` du document.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<lists>
  <list length="3">
    <item>A</item><item>B</item><item>C</item>
  </list>
  <list length="4">
    <item>1</item><item>2</item><item>3</item>
  </list>
</lists>
```

Le résultat de la vérification du document ci-dessus avec le schematron donné précédemment est donné à la section suivante.

7.3. Fonctionnement

Le principe de fonctionnement de schematron est le suivant. Un document cible est validé avec un schematron en utilisant une application appropriée. Cette validation produit un rapport qui retrace les différentes étapes de la validation. Ce rapport contient des messages destinés à l'utilisateur. Ces messages peuvent provenir d'éventuelles erreurs mais ils peuvent aussi être positifs et confirmer que le document satisfait bien certaines contraintes. Ces différents messages sont produits par l'application de validation ou sont issus du schematron. Le schematron associe en effet des messages aux différentes contraintes qu'il contient. Ce rapport est souvent lui même un document XML.

La validation d'un document avec un schematron est souvent réalisée en deux phases. Dans une première phase, le schematron est transformé en une version compilée qui est indépendante du document. Dans une seconde phase, la version compilée est utilisée pour produire le rapport. La première phase est parfois, elle-même, scindée en plusieurs étapes afin de prendre en compte les inclusions et les blocs abstraits.

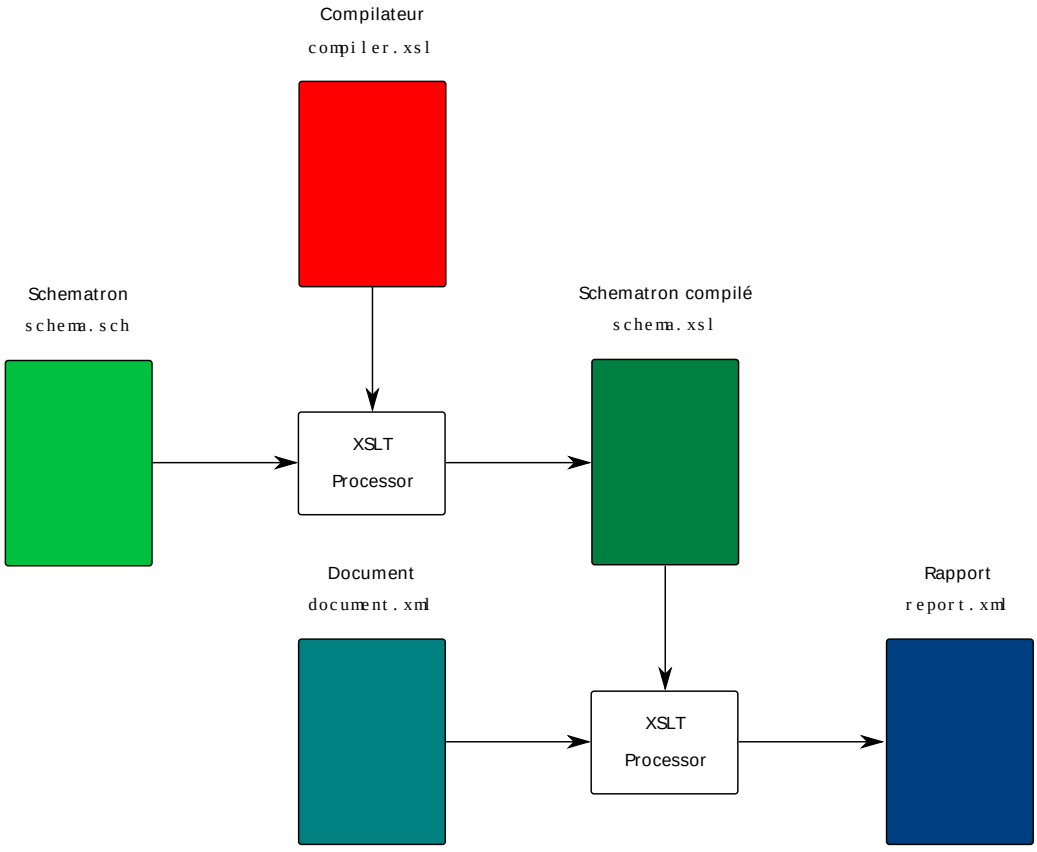


Figure 7.1. Validation avec un schematron

Il existe plusieurs implémentations de schematron souvent basées sur XSLT [Chapitre 8]. Les deux phases sont réalisées par l'application de feuilles de style XSLT. Une première feuille de style XSLT transforme le schematron en une autre feuille de style XSLT qui constitue la version compilée du schematron. Cette dernière feuille de style est appliquée au document pour produire le rapport.

La composition exacte du rapport dépend de l'application qui réalise la validation du document avec le schematron. Il contient des fragments de texte issus du schematron mais aussi du texte produit par l'application de validation. Ce

rapport peut être un simple fichier texte mais il est souvent un document XML, en particulier lorsque la validation est réalisée via XSLT. Le format SVRL est un dialecte XML conçu spécialement pour les rapports de validation avec des schematrons. Il est en particulier utilisé par l'implémentation standard de schematron disponible à l'adresse `http://www.schematron.com/`.

Le rapport SVRL obtenu par la validation du document XML avec le schematron donné ci-dessus est le suivant.

```
<?xml version="1.0" standalone="yes"?>
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl"
                        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                        xmlns:sch="http://purl.oclc.org/dsdl/schematron"
                        title="Comparaison attribut/nombre d'enfants"
                        schemaVersion="ISO19757-3">
  <svrl:active-pattern/>
  <svrl:fired-rule context="list"/>
  <svrl:fired-rule context="list"/>
  <svrl:failed-assert test="@length = count(*)" location="/lists/list[2]">
    <svrl:text>L'attribut length doit être
               égal au nombre d'enfants.</svrl:text>
  </svrl:failed-assert>
</svrl:schematron-output>
```

7.4. Structure globale d'un schematron

L'ensemble d'un schematron est contenu dans un élément `sch:schema`. L'espace de noms [Chapitre 4] des schematrons est identifié par l'URL `http://purl.oclc.org/dsdl/schematron`. Il existe des versions antérieures des schematrons qui utilisaient un autre espace de noms. Le préfixe généralement associé à l'espace de noms est `sch` comme dans cet ouvrage ou aussi `iso` pour bien distinguer la version ISO actuelle des schematrons des anciennes versions.

Un schematron est essentiellement constitué de règles regroupées en blocs. Il contient également du texte permettant de décrire les opérations effectuées. Il peut aussi déclarer des espaces de noms et des clés XSLT [Section 8.12].

7.4.1. Espaces de noms cible

Lorsque les éléments des documents à valider appartiennent à un ou des espaces de noms, il est nécessaire de les déclarer dans le schematron et de leur associer des préfixes. Les préfixes sont nécessaires pour nommer correctement les éléments dans les expressions XPath des règles. Il faut en effet utiliser des noms qualifiés.

La déclaration d'un espace de noms se fait par l'élément `sch:ns` dont les attributs `prefix` et `uri` donnent respectivement le préfixe et l'URI qui identifie l'espace de noms.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>Comparaison attribut/nombre d'enfants</sch:title>
  <!-- Déclaration de l'espace de noms cible associé au prefix tns -->
  <sch:ns prefix="tns" uri="http://www.liafa.jussieu.fr/~carton/"/>
  <sch:pattern>
    <!-- Le motif XPath utilise le nom qualifié de l'élément -->
    <sch:rule context="tns:list">
      <sch:assert test="@length = count(*)">
        L'attribut length doit être égal au nombre d'éléments.
      </sch:assert>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

7.4.2. Règles et blocs

Les règles d'un schematron sont regroupées en blocs. Chacun de ces blocs est introduit par un élément `sch:pattern`.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  ...
  <sch:pattern>
    ...
  </sch:pattern>
  <sch:pattern>
    ...
  </sch:pattern>
  ...
</sch:schema>
```

Le contenu de chaque élément `sch:pattern` est composé de règles. Chaque règle est donnée par un élément `sch:rule` dont l'attribut `context` détermine à quels éléments la règle s'applique. Chaque règle contient ensuite des tests introduits par les éléments `sch:assert` et `sch:report`.

```
<sch:pattern>
  <sch:rule context="...">
     ...
  </sch:rule>
  <sch:rule context="...">
     ...
  </sch:rule>
  ...
</sch:pattern>
```

La validation d'un document avec un schematron consiste à traiter séquentiellement chacun des blocs de règles. Pour chaque bloc et pour chaque élément du document cible est déterminée la règle à appliquer. Les différents tests de cette règle sont réalisés et des messages sont ajoutés au rapport en fonction des résultats de ces tests. Même si plusieurs règles peuvent s'appliquer à un élément, une seule des règles du bloc est réellement appliquée. Il faut donc éviter la situation où plusieurs règles d'un même bloc s'appliquent potentiellement à un même élément.

7.4.3. Titres et commentaires

Le schematron ainsi que chacun de ses blocs peuvent être commentés. Les éléments `sch:schema` et `sch:pattern` peuvent avoir comme enfant un élément `sch:title` pour donner un titre. Ils peuvent aussi contenir des éléments `sch:p` prévus pour donner des descriptions plus longues. Le contenus des éléments `sch:title` et `sch:p` sont souvent repris pour la construction du rapport.

7.5. Règles

Chaque règle est matérialisée par un élément `sch:rule` qui contient un ou plusieurs tests. L'élément `sch:rule` possède un attribut `context` dont la valeur doit être un motif XPath [Section 6.8]. Ce motif détermine sur quels nœuds s'applique la règle.

Les tests d'une règle sont introduits par les éléments `sch:assert` et `sch:report`. Ces deux éléments prennent la même forme. Ils possèdent un attribut `test` dont la valeur est une expression XPath et ils contiennent du texte qui est éventuellement utilisé pour construire le rapport. Ces deux éléments se distinguent par leurs sémantiques qui sont à l'opposé l'une de l'autre.

```
<sch:rule context="...">
```

```
  <sch:assert test="...">
     ...
  </sch:assert>
  <sch:report test="...">
     ...
  </sch:report>
  ...
</sch:rule>
```

Un test introduit par `sch:assert` est réalisé de la façon suivante. L'expression XPath contenue dans l'attribut `test` est évaluée en prenant le nœud sélectionné comme contexte et le résultat de l'évaluation est converti en une valeur booléenne. Si le résultat est `false`, le texte contenu dans l'élément `sch:assert` est ajouté au rapport. Sinon rien n'est ajouté au rapport.

Dans l'exemple ci-dessous, le message d'erreur est ajouté au rapport si la condition n'est pas vérifiée, c'est-à-dire si l'élément `book` n'a aucun des attributs `id` ou `key`.

```
<sch:rule context="book">
  <sch:assert test="@id|@key">
    L'élément book doit avoir un attribut id ou key
  </sch:assert>
</sch:rule>
```

Un test introduit par `sch:report` est, au contraire, réalisé de la façon suivante. L'expression XPath contenue dans l'attribut `test` est évaluée en prenant le nœud sélectionné comme contexte et le résultat de l'évaluation est converti en une valeur booléenne. Si le résultat est `true`, le texte contenu dans l'élément `sch:report` est ajouté au rapport. Sinon rien n'est ajouté au rapport.

Dans l'exemple ci-dessous, le message d'erreur est ajouté au rapport si la condition est vérifiée, c'est-à-dire si l'élément `book` a simultanément les deux attributs `id` et `key`.

```
<sch:rule context="book">
  <sch:report test="count(@id|@key) &gt; 1">
    L'élément book doit avoir un seul des attributs id ou key
  </sch:report>
</sch:rule>
```

Chaque règle peut bien sûr contenir plusieurs éléments `sch:assert` et `sch:report` comme dans l'exemple ci-dessous. L'ordre de ces différents éléments est sans importance.

```
<sch:rule context="tns:pattern[@is-a]">
  <sch:assert test="key('patid', @is-a)">
    L'attribut is-a doit référencer un bloc abstrait.
  </sch:assert>
  <sch:report test="@abstract = 'true'">
    Un bloc avec un attribut is-a ne peut pas être abstrait.
  </sch:report>
  <sch:report test="rule">
    Un bloc avec un attribut is-a ne peut pas contenir de règle.
  </sch:report>
</sch:rule>
```

Le texte contenu dans les éléments `sch:assert` et `sch:report` peut aussi contenir des éléments `sch:name` et `sch:value-of` qui permettent d'ajouter du contenu dynamique. Lors de la construction du rapport, ces éléments sont évalués et ils sont remplacés par le résultat de l'évaluation. L'élément `sch:name` s'évalue en le nom de l'élément sur lequel est appliqué la règle. Il est particulièrement utile dans les règles abstraites [Section 7.6] et les blocs abstraits [Section 7.7] où le nom du contexte n'est pas fixé. L'élément `sch:value-of` a un attribut `select` contenant une expression XPath. L'évaluation de cette expression remplace l'élément `sch:value-of` dans le rapport. Un exemple d'utilisation de cet élément est donné à la section suivante.

7.5.1. Variables locales

Une règle peut définir des variables locales introduites par des éléments `sch:let`. Celles-ci mémorisent le résultat d'un calcul intermédiaire pour des utilisations dans la règle. Chaque élément `sch:let` a des attributs `name` et `value` pour spécifier le nom et la valeur de la variable. L'attribut `value` doit contenir une expression XPath qui donne sa valeur à la variable. Cette valeur ne peut plus ensuite être modifiée. Les éléments `sch:let` doivent être les premiers enfants de l'élément `sch:rule`. La variable ainsi déclarée est disponible dans toute la règle.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>Tests sur les heures</sch:title>
  <sch:pattern>
    <sch:rule context="time">
      <sch:let name="hour" value="number(substring(.,1,2))"/>
      <sch:let name="min"  value="number(substring(.,4,2))"/>
      <sch:let name="sec"  value="number(substring(.,7,2))"/>

      <!-- Test si l'heure est de la forme HH:MM:SS -->
      <sch:assert test="string-length(.) = 8 and substring(.,3,1) = ':' and
                                                 substring(.,6,1) = ':'">
        L'heure <sch:value-of select="."/> doit être au format HH:MM:SS.
      </sch:assert>
      <sch:assert test="$hour >= 0 and $hour &lt;= 23">
        Le nombre d'heures doit être compris entre 0 et 23.
      </sch:assert>
      <sch:assert test="$min >= 0 and $min &lt;= 59">
        Le nombre de minutes doit être compris entre 0 et 59.
      </sch:assert>
      <sch:assert test="$sec >= 0 and $sec &lt;= 59">
        Le nombre de secondes doit être compris entre 0 et 59.
      </sch:assert>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

7.6. Règles abstraites

Le principe général des règles abstraites est d'avoir des règles génériques susceptibles de s'appliquer à différents contextes. Le contexte d'une règle, c'est-à-dire l'ensemble des nœuds sur lesquels elle s'applique est normalement donné par son attribut `context`. Il est possible de définir une règle sans contexte qui définit seulement des contraintes. Elle ne peut pas être utilisée directement mais d'autres règles l'utilisent en spécifiant le contexte.

Une règle est déclarée abstraite avec un attribut `abstract` ayant la valeur `true`. Elle n'a pas d'attribut `context`. Elle possède, en revanche, un attribut `id` qui permet de la désigner pour l'utiliser. Une autre règle peut utiliser une règle abstraite en lui fournissant un contexte. Elle fait appel à la règle abstraite grâce à l'élément `sch:extends` dont l'attribut `rule` donne l'identifiant de la règle abstraite.

Dans l'exemple suivant une règle abstraite de nom `has-title` est définie. Elle vérifie que le nœud contexte possède un enfant `title` et que celui-ci est le premier enfant. Deux règles utilisent ensuite cette règle pour les éléments `book` et `chapter`.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>Utilisation de règles abstraites</sch:title>
  <sch:pattern>
```

```
    <!-- Règle abstraite qui teste si le premier enfant est title -->
    <sch:rule abstract="true" id="has-title">
      <sch:assert test="*[1][self::title]">
        L'élément <sch:name/> doit avoir un enfant title qui
        doit être le premier enfant.
      </sch:assert>
    </sch:rule>
    <!-- Utilisation de la règle abstraite pour les éléments book -->
    <sch:rule context="book">
      <sch:extends rule="has-title"/>
      <sch:assert test="chapter">
        Le livre soit contenir au moins un chapitre.
      </sch:assert>
    </sch:rule>
    <!-- Utilisation de la règle abstraite pour les éléments chapter -->
    <sch:rule context="chapter">
      <sch:extends rule="has-title"/>
      <sch:assert test="para">
        Le chapire soit contenir au moins un paragraphe.
      </sch:assert>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

Ce schematron permet de vérifier que le document suivant n'est pas correct. L'élément `title` n'est pas le premier enfant du second `chapter` et le troisième `chapter` n'a pas d'enfant `title`.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<book>
  <title>Titre du livre</title>
  <chapter>
    <title>Premier chapitre</title>
    <para>Ceci est le premier chapitre ...</para>
  </chapter>
  <chapter>
    <para>Paragraphe avant le titre ...</para>
    <title>Titre mal placé</title>
    <para>Second paragraphe du second chapitre après le titre ...</para>
  </chapter>
  <chapter>
    <para>Chapitre sans titre</para>
  </chapter>
</book>
```

7.7. Blocs abstraits

Les blocs abstraits généralisent le principe des règles abstraites. Ils déclarent des règles qui peuvent s'appliquer à différentes situations. Leur principe de fonctionnement est proche de celui des fonctions de n'importe quel langage de programmation. Un bloc abstrait contient des règles qui utilisent des paramètres. Ce bloc est alors utilisé par d'autres blocs qui instancient les paramètres en leur donnant des valeurs explicites.

Un bloc est déclaré abstrait avec un attribut `abstract` ayant la valeur `true`. Le bloc qui utilise un bloc abstrait doit avoir un attribut `is-a` qui donne l'identifiant du bloc abstrait. Il ne doit pas contenir de règles mais seulement des éléments `sch:param` qui permettent d'instancier les paramètres. L'élément `sch:param` a des attributs `name` et `value` qui donnent respectivement le nom du paramètre et la valeur qui lui est affectée.

Le fonctionnement des blocs abstraits est semblable au passage de paramètres des éléments `xsl:apply-templates` et `xsl:call-template` [Section 8.9.2] de XSLT. En revanche, l'élément `sch:param` des

schematrons est l'analogue de l'élément `xsl:with-param` de XSLT. L'élément `xsl:param` de XSLT n'a pas d'équivalent dans les schematrons car les paramètres des blocs abstraits ne sont pas déclarés.

Le schematron suivant définit un bloc abstrait `uniq` qui contient deux règles dépendant des paramètres `elem` et `desc`. La première règle vérifie que l'élément `elem` a au moins un descendant `desc`. La seconde vérifie au contraire qu'il n'a pas plus d'un descendant `desc`. Ces deux règles conjuguées vérifient donc que l'élément `elem` a exactement un seul descendant `desc`.

Le bloc abstrait `uniq` est ensuite utilisé par les deux blocs `uniq-id` et `uniq-title`. Le premier bloc donne les valeurs `book` et `@id|@key` aux deux paramètres `elem` et `desc`. Il vérifie donc que chaque élément `book` possède exactement un seul des deux attributs `id` et `key`. Le second bloc donne les valeurs `book` et `title` aux paramètres `elem` et `desc`. Il vérifie donc que chaque élément `book` possède exactement un seul enfant `title`.

La vérification effectuée par le premier bloc n'est pas faisable avec les DTD [Chapitre 3] et les schémas XML [Chapitre 5]. Les déclarations d'attributs de ces deux langages se font sur chacun des attributs de façon indépendante. Il n'est pas possible d'exprimer une contrainte qui met en relation deux attributs ou deux éléments.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>Utilisation de blocs abtraits</sch:title>
  <!-- Déclaration du bloc abstrait -->
  <sch:pattern abstract="true" id="uniq">
    <!-- Les règles utilisent les paramètres elem et desc -->
    <sch:rule context="$elem">
      <sch:assert test="$desc">
        L'élément <sch:name/> doit avoir un descendant $desc.
      </sch:assert>
      <sch:report test="count($desc) &gt; 1">
        L'élément <sch:name/> doit avoir un seul descendant $desc.
      </sch:report>
    </sch:rule>
  </sch:pattern>
  <!-- Utilisation du bloc abstrait -->
  <sch:pattern is-a="uniq" id="uniq-id">
    <sch:param name="elem" value="book"/>
    <sch:param name="desc" value="@id|@key"/>
  </sch:pattern>
  <sch:pattern is-a="uniq" id="uniq-title">
    <sch:param name="elem" value="book"/>
    <sch:param name="desc" value="title"/>
  </sch:pattern>
</sch:schema>
```

Le mécanisme des blocs abstraits est souvent implémenté comme les `#define` du langage C. Chaque bloc qui utilise un bloc abstrait est remplacé par une copie de celui-ci où les paramètres sont substitués par leurs valeurs. Le schematron précédent est en fait équivalent au schematron suivant. Le bloc abstrait `uniq` a disparu mais ses règles apparaissent dupliquées dans les deux blocs `uniq-id` et `uniq-title`.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>Substitution des blocs abstraits</sch:title>
  <sch:pattern id="uniq-id">
    <sch:rule context="book">
      <sch:assert test="@id|@key">
        L'élément <sch:name/> doit avoir un descendant @id|@key
      </sch:assert>
      <sch:report test="count(@id|@key) &gt; 1">
```

```
        L'élément <sch:name/> doit avoir un seul descendant @id|@key
      </sch:report>
    </sch:rule>
  </sch:pattern>
  <sch:pattern id="uniq-title">
    <sch:rule context="book">
      <sch:assert test="title">
        L'élément <sch:name/> doit avoir un descendant title
      </sch:assert>
      <sch:report test="count(title) &gt; 1">
        L'élément <sch:name/> doit avoir un seul descendant title
      </sch:report>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

L'exemple ci-dessous illustre la puissance des schematrons. Ce schematron exprime certaines contraintes que doivent satisfaire les schematrons pour être valides. Ces contraintes portent sur les liens entre les éléments `pattern` abstraits et ceux qui les utilisent.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <sch:title>Vérification des liens is-a des schematrons</sch:title>

  <sch:p>Ce schematron vérifie que, dans un schematron, tout bloc référencé
  par un autre bloc par l'attribut is-a est bien déclaré abstrait par
  l'attribut abstract avec la valeur true.</sch:p>

  <!-- Déclaration de l'espace de noms cible : celui des schematrons -->
  <!-- Ne pas utiliser le préfixe sch car cela pose problème -->
  <sch:ns prefix="tns" uri="http://purl.oclc.org/dsdl/schematron"/>

  <!-- Clé pour retrouver les éléments pattern par leur id -->
  <xsl:key name="patid" match="tns:pattern" use="@id"/>

  <sch:pattern>
    <sch:rule context="tns:pattern[@is-a]">
      <sch:assert test="key('patid', @is-a)">
        L'attribut is-a doit référencer un bloc abstrait.
      </sch:assert>
      <sch:report test="@abstract = 'true'">
        Un bloc avec un attribut is-a ne peut pas être abstrait.
      </sch:report>
      <sch:report test="rule">
        Un bloc avec un attribut is-a ne peut pas contenir de règle.
      </sch:report>
    </sch:rule>
  </sch:pattern>
  <sch:pattern>
    <sch:rule context="tns:pattern[@abstract = 'true']">
      <sch:assert test="@id">
        Un bloc abstrait doit avoir un attribut id.
      </sch:assert>
      <sch:report test="@is-a">
        Un bloc abstrait ne peut pas avoir un attribut is-a.
      </sch:report>
```

```
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

7.8. Phases de validations

Il est possible de regrouper les blocs en phases. Chaque phase est identifiée par un nom. Lors de la validation d'un document par un schematron, il est possible de spécifier la phase à effectuer. Seuls les blocs appartenant à cette phase sont alors pris en compte. Ce mécanisme permet de scinder un schematron en plusieurs parties et de procéder à une validation incrémentale d'un document. Chaque phase déclare les blocs qu'elle contient et un bloc peut appartenir à plusieurs phases. Lorsque la validation par schematron est implémentée avec XSLT, la phase est précisée en donnant un paramètre global [Section 8.9.2] à la feuille de style XSLT. Il existe une phase par défaut appelée `#ALL` qui comprend tous les blocs. Si aucune phase n'est spécifiée, la validation utilise tous les blocs du schematron.

Une phase est déclarée par un élément `sch:phase` ayant un attribut `id` permettant de l'identifier. Chaque bloc de cette phase est donné par un enfant `sch:active` ayant un attribut `pattern` qui précise l'identifiant du bloc.

Dans l'exemple minimal ci-dessous, il y a deux phases appelées `phase1` et `phase2`. Chacune de ces deux phases contient un seul bloc.

```
<?xml version="1.0" encoding="utf-8"?>
<sch:schema queryBinding="xslt" schemaVersion="ISO19757-3"
            xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>Utilisation de phases</sch:title>

  <!-- Phase 1 ne comprenant que le premier bloc -->
  <sch:phase id="phase1">
    <sch:active pattern="idkey"/>
  </sch:phase>
  <!-- Phase 2 ne comprenant que le second bloc -->
  <sch:phase id="phase2">
    <sch:active pattern="count"/>
  </sch:phase>
  <!-- Vérification des attributs id et key -->
  <sch:pattern id="idkey">
    <sch:rule context="book">
      <sch:assert test="@id|@key">
        L'élément book doit avoir un attribut id ou key
      </sch:assert>
    </sch:rule>
  </sch:pattern>
  <!-- Décompte du nombre de livres -->
  <sch:pattern id="count">
    <sch:rule context="bibliography">
      <sch:report test="book">
        Il y a <sch:value-of select="count(book)"/> livre(s).
      </sch:report>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

Chapitre 8. XSLT

Le langage XSL (pour *XML Stylesheet Language*) a été conçu pour transformer des documents XML en d'autres formats comme PDF ou des pages HTML. Au cours de son développement, le projet s'est avéré plus complexe que prévu et il a été scindé en deux unités distinctes XSLT et XSL-FO. Le langage XSLT (pour *XML Stylesheet Language Transformation*) est un langage de transformation de documents XML. Le langage XSL-FO [Chapitre 9] (pour *XML Stylesheet Language - Formatting Objets*) est un langage de mise en page de document. Le processus de transformation d'un document XML en un document imprimable, au format PDF par exemple, est donc découpé en deux phases. Dans la première phase, le document XML est transformé en un document XSL-FO à l'aide de feuilles de style XSLT. Dans la seconde phase, le document FO obtenu à la première phase est converti par un processeur FO en un document imprimable.

Même si le langage XSLT puise son origine dans la transformation de documents XML en document XSL-FO, il est adapté à la transformation d'un document de n'importe quel dialecte XML dans un document de n'importe quel autre dialecte XML. Il est souvent utilisé pour produire des documents XSL-FO ou XHTML, il peut aussi produire des documents SVG [Chapitre 11].

Ce chapitre est consacré à la partie XSLT de XSL. Le cours est essentiellement basé sur des exemples. Les différentes constructions du langage XSLT sont illustrées pas des fragments de programmes extraits des exemples.

8.1. Principe

Le principe de fonctionnent de XSLT est le suivant. Une feuille de style XSLT contient des règles qui décrivent des transformations. Ces règles sont appliquées à un document source XML pour obtenir un nouveau document XML résultat. Cette transformation est réalisée par un programme appelé *processeur XSLT*. La feuille de style est aussi appelée *programme* dans la mesure où il s'agit des instructions à exécuter par le processeur.

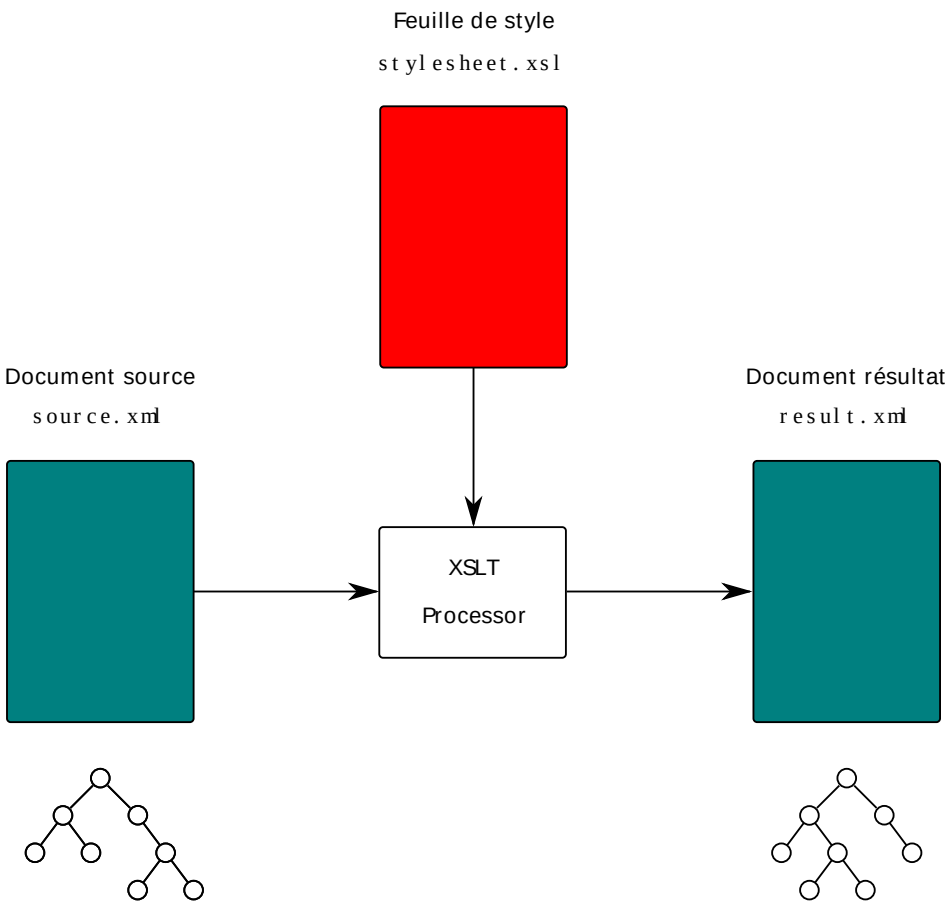


Figure 8.1. Principe de XSLT

La version 2.0 de XSLT a introduit un certain nombre d'évolutions par rapport à la version 1.0. La première évolution est l'utilisation de XPath [Chapitre 6] 2.0 à la place de XPath 1.0. La seconde évolution importante est la possibilité de traiter un document validé au préalable par un schéma XML [Chapitre 5].

L'intérêt de cette validation est d'associer un type à chaque contenu d'élément et et à chaque valeur d'attribut. Si le type d'un attribut est, par exemple, `xsd:integer` et que sa valeur `123`, celle-ci est interprétée comme un entier et non comme une chaîne de caractères à la manière de XSLT 1.0. La validation par un schéma n'est pas nécessaire pour utiliser XSLT 2.0. Il existe donc deux façons de traiter un document avec XSLT 2.0, soit sans schéma soit avec schéma. La première façon correspond au fonctionnement de XSLT 1.0. La seconde façon prend en compte les types associés aux nœuds par la validation.

Pour la version 1.0 de XSLT, il existe plusieurs processeurs libres dont le plus répandu est `xsltproc`. Il est très souvent déjà installé sur les machines car il fait partie de la librairie standard `libxslt`. En revanche, il n'implémente que la version 1.0 de la norme avec quelques extensions. Le logiciel `saxon` implémente la XSLT 2.0 mais la version gratuite n'implémente que le traitement sans schéma. Il n'existe pas actuellement de processeur libre implémentant le traitement avec schéma. Pour cette raison, ce chapitre se concentre sur le traitement sans schéma même si l'essentiel reste encore valable dans le traitement avec schéma.

Le langage XSLT est un dialecte XML. Ceci signifie qu'une feuille de style XSLT est un document XML qui peut lui-même être manipulé ou produit par d'autres feuilles de style. Cette possibilité est d'ailleurs exploité par schematron [Chapitre 7] pour réaliser une validation en plusieurs phases.

8.2. Premier programme : `Hello, World!`

On commence par la feuille de style XSLT la plus simple. Cette-ci se contente de produire un document XHTML affichant le message `Hello World!` dans un titre. Cette feuille de style a donc la particularité de produire un document indépendant du document source.

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

❶ Élément racine `xsl:stylesheet` de la feuille de style.
❷ Déclaration de l'espace de noms XSLT associé au préfixe `xsl`.
❸ Déclaration de l'espace de noms XHTML comme espace de noms par défaut.
❹ Définition d'une règle s'appliquant à la racine `'/'` du document source.
❺ Fragment de document XHTML retourné par la règle.

Cette feuille de style est constituée d'une seule règle introduite par l'élement `xsl:template` dont l'attribut `match` précise que cette règle s'applique à la racine du document source. L'élément `xsl:template` contient le document XHTML produit. En appliquant ce programme à n'importe quel document 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>
```

L'entête XML du résultat à été automatiquement mise par le processeur XSLT. Comme le codage [Section 2.2] du document résultat n'est pas spécifié par la feuille de style, c'est le codage par défaut de XML qui a été choisi. Le processeur a inséré la déclaration de l'espace de noms XHTML dans l'élément racine `html` du document. Le processeur a également ajouté l'élément `meta` propre à HTML pour préciser le codage des caractères.

8.3. Modèle de traitement

Le document XML source est transformé en un document XML résultat obtenu en appliquant les règles de la *feuille de style* à des nœuds du document source. Chaque application de règle à un nœud produit un fragment de document XML. Tous ces fragments sont assemblés pour former le document résultat.

Chaque fragment produit par l'application d'une règle est une suite de nœuds représentant des éléments, des attributs, des instructions de traitement et des commentaires. Il s'agit le plus souvent d'une suite d'éléments ou d'attributs. Lors de l'assemblage des fragments, ces nœuds viennent s'insérer à l'intérieur d'un autre fragment.

Chaque règle est déclarée par un élément `xsl:template`. Le contenu de cet élément est le fragment de document qui est produit par l'application de cette règle. Ce contenu contient des éléments de l'espace de noms XSLT et des éléments d'autres espaces de noms. Ces derniers éléments sont recopiés à l'identique pour former le fragment. Les éléments de XSLT sont des instructions qui sont exécutées par le processeur XSLT. Ces éléments sont remplacés dans le fragment par le résultat de leur exécution. L'élément essentiel est l'élément `xsl:apply-templates` qui permet d'invoquer l'application d'autres règles. Les fragments de document produits par ces applications de règles remplacent l'élément `xsl:apply-templates`. L'endroit où se trouve l'élément `xsl:apply-templates` constitue donc le point d'ancrage pour l'insertion du ou des fragments produits par son exécution.

La forme globale d'une règle est donc la suivante.

```
<xsl:template match="...">
  <!-- Fragment produit -->
  ...
  <!-- Application de règles -->
  <xsl:apply-templates .... />
  ...
  <!-- Application de règles -->
  <xsl:apply-templates .... />
  ...
<xsl:template/>
```

Chacune des règles est déclarée avec un élément `xsl:template` dont l'attribut `match` précise sur quels nœuds elle est susceptible d'être appliquée. Le processus de transformation consiste à appliquer des règles sur des nœuds *actifs* du documents source. Au départ, seule la racine est active et la première règle est donc appliquée à cette racine. L'application de chaque règle produit un fragment de document qui va constituer une partie du document résultat. Elle active d'autres nœuds avec des éléments `xsl:apply-templates` placés au sein du fragment de document. Des règles sont alors appliquées à ces nouveaux nœuds actifs. D'une part, elles produisent des fragments de documents qui s'insèrent dans le document résultat à la place des éléments `xsl:apply-templates` qui les ont provoquées. D'autre part, elles activent éventuellement d'autres nœuds pour continuer le processus. Ce dernier s'arrête lorsqu'il n'y a plus de nœuds actifs.

Le processus de transformation s'apparente donc à un parcours de l'arbre du document source. Il y a cependant une différence importante. Dans un parcours classique d'un arbre comme les parcours en largeur ou en profondeur, le traitement d'un nœud entraîne le traitement de ses enfants. L'application d'une règle XSLT active d'autres nœuds mais ceux-ci ne sont pas nécessairement les enfants du nœud sur lequel la règle s'applique. Les nœuds activés sont déterminés par l'attribut `select` des éléments `xsl:apply-templates`. Chaque attribut `select` contient une expression XPath dont l'évaluation donne la liste des nœuds activés.

La figure ci-dessous illustre la construction de l'arbre résultat par l'application des règles XSLT. Chacun des triangles marqués `template` représente un fragment de document produit par l'application d'une règle. Tous ces triangles sont de même taille sur la figure même si les fragments ne sont pas identiques. Les flèches marquées `apply-templates` symbolisent l'application de nouvelles règles par l'activation de nœuds. L'arbre du document résultat est, en quelque sorte, obtenu en contractant ces flèches marquées `apply-templates` et en insérant à leur point de départ le triangle sur lequel elles pointent. Les flèches partant d'un même triangle peuvent partir de points différents car un même fragment de document peut contenir plusieurs éléments `xsl:apply-templates`.

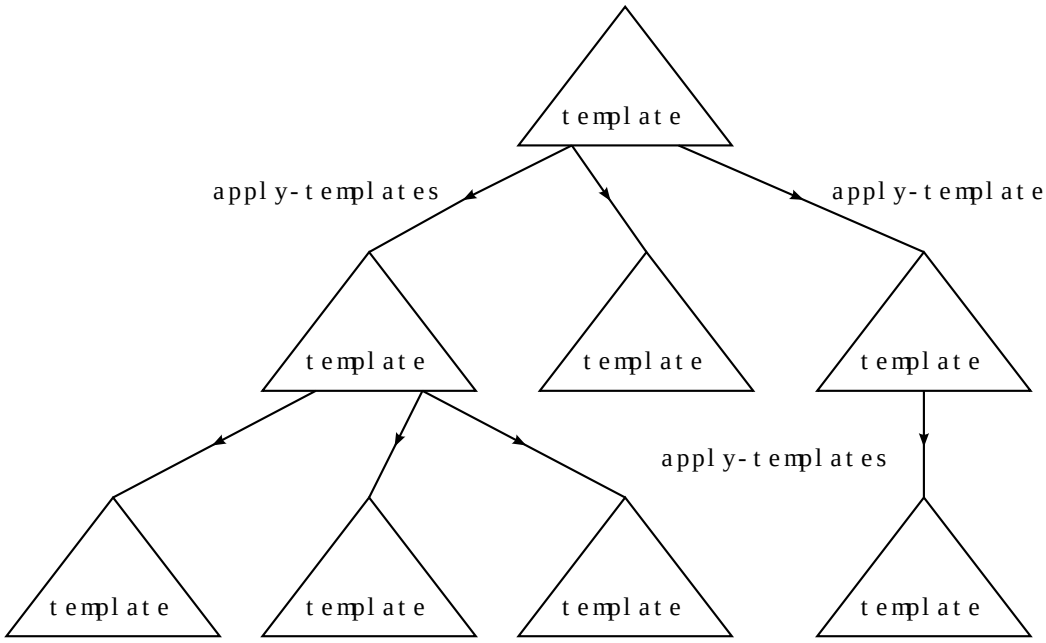


Figure 8.2. Traitement

Il est maintenant possible de revenir sur le premier programme `Hello, Word!` et d'en expliquer le fonctionnement. Ce programme contient une seule règle qui s'applique à la racine du document source. Comme le premier nœud actif au départ est justement la racine, le processus commence par appliquer cette règle. Le document résultat est construit en ajoutant à sa racine le contenu de la règle. Comme ce contenu ne contient aucun élément `xsl:apply-templates`, aucun nouveau nœud n'est rendu actif et le processus de transformation s'arrête après l'application de cette première règle.

8.4. Entête

Le programme ou *feuille de style* est entièrement inclus dans un élément `xsl:stylesheet` ou de façon complètement équivalente un élément `xsl:transform`. L'attribut `version` précise la version de XSLT utilisée. Les valeurs possibles sont `1.0` ou `2.0`. Un processeur XSLT 1.0 signale généralement une erreur lorsque la feuille de style n'utilise pas cette version. Un processeur XSLT 2.0 passe dans un mode de compatibilité avec la version 1.0 lorsqu'il rencontre une feuille de style de XSLT 1.0. L'espace de noms des éléments de XSLT doit être déclaré. Il est identifié par l'URI `http://www.w3.org/1999/XSL/Transform`. Le préfixe `xsl` est généralement associé à cet espace de noms. Dans tout ce chapitre, ce préfixe est utilisé pour qualifier les éléments XSLT. Il est aussi indispensable de déclarer les espaces de noms du document résultat si celui-ci en utilise. Ces déclarations d'espaces de noms sont importantes car le contenu de chaque règle contient un mélange d'éléments XSLT et d'éléments du document résultat.

Les processeurs XSLT ajoutent les déclarations nécessaires d'espaces de noms dans le document résultat. Il arrive que certains espaces de noms soient déclarés alors qu'ils ne sont pas indispensables. L'attribut `exclude-result-prefixes` de `xsl:stylesheet` permet d'indiquer que certains espaces de noms ne seront pas utilisés dans le document résultat et que leurs déclarations doivent être omises. Cet attribut contient une liste de préfixes séparés par des espaces. Dans l'exemple suivant, les espaces de noms associés aux préfixes `xsl` (XSLT) et `dbk` (DocBook) ne sont pas déclarés dans le document résultat.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                exclude-result-prefixes="xsl dbk"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:dbk="http://docbook.org/ns/docbook"
                xmlns="http://www.w3.org/1999/xhtml">
```

```
  ...
```

L'élément `xsl:output` doit être un enfant de l'élément `xsl:stylesheet`. Il 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 [Section 3.2.2]. Un exemple typique d'utilisation est le suivant.

```
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
<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"/>
```

8.4.1. Traitement des espaces

Avant d'appliquer la feuille de style au document source, est effectué un traitement préalable des caractères d'espacement [Section 2.2.2] dans le document. Ce traitement consiste à supprimer certains nœuds textuels ne contenant que des caractères d'espacement. Seuls les nœuds textuels contenant uniquement des caractères d'espacement sont concernés. Les nœuds contenant, au moins, un autre caractère ne sont jamais supprimés par ce traitement. L'attribut `xml:space` [Section 2.7.4.2] ainsi que les éléments `xsl:strip-space` et `xsl:preserve-space` contrôlent lesquels des nœuds textuels sont effectivement supprimés.

La sélection des nœuds textuels à supprimer est basée sur une liste de noms d'éléments du document source dont les caractères d'espacement doivent être préservés. Par défaut, cette liste contient tous les noms d'éléments et aucun nœud textuel contenant des caractères d'espacement n'est supprimé. L'élément `xsl:strip-space` permet de retirer des noms d'éléments de cette liste et l'élément `xsl:preserve-space` permet d'en ajouter. Ce dernier élément est rarement utilisé puisque la liste contient, au départ, tous les noms d'éléments. L'attribut `elements` des éléments `xsl:strip-space` et `xsl:preserve-space` contient une liste de noms d'éléments (à retirer ou à ajouter) séparés par des espaces. Cet attribut peut également contenir la valeur `'*'` ou des valeurs de la forme `tns:*`. Dans ces cas, sont retirés (pour `xsl:strip-space`) ou ajoutés (pour `xsl:preserve-space`) à la liste tous les noms d'éléments ou tous les noms d'éléments de l'espace de noms associé au préfixe `tns`.

Un nœud textuel est retiré de l'arbre du document si les deux conditions suivantes sont vérifiées. Il faut d'abord que le nom de son parent n'appartienne pas à la liste des noms d'éléments dont les espaces doivent être préservés. Ceci signifie que le nom de son parent a été retiré de la liste grâce à un élément `xsl:strip-space`. Il faut ensuite que la valeur de l'attribut `xml:space` donné au plus proche parent contenant cet attribut ne soit pas la valeur `preserve`. La valeur par défaut de cet attribut est `default` qui autorise la suppression.

La feuille de style suivante recopie le document source mais les nœuds textuels contenant uniquement des caractères d'espacement sont supprimés du contenu des éléments `strip`. Afin d'observer les effets de `xsl:strip-space`, il est nécessaire que la valeur de l'attribut `indent` de l'élément `xsl:output` soit `no` pour qu'aucun caractère d'espacement ne soit ajouté pour l'indentation.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="iso-8859-1" indent="no"/>
  <xsl:strip-space elements="strip"/>
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Le document suivant contient des éléments `preserve` et des éléments `strip`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<list>
  <sublist>
    <preserve>
      <p>a b c</p>   <p> a b c </p>
    </preserve>
    <strip xml:space="preserve">
      <p>a b c</p>   <p> a b c </p>
    </strip>
    <strip>
      <p>a b c</p>   <p> a b c </p>
    </strip>
  </sublist>
  <sublist xml:space="preserve">
    <strip>
      <p>a b c</p>   <p> a b c </p>
    </strip>
    <strip xml:space="default">
      <p>a b c</p>   <p> a b c </p>
    </strip>
  </sublist>
</list>
```

Si la feuille de style précédente est appliquée au document précédent, certains espaces sont supprimés. Aucun des espaces contenus dans les éléments p n'est supprimé car ces éléments ont un seul nœud textuel comme enfant et ce nœud textuel contient des caractères autres que des espaces. En revanche, des retours à la ligne entre les balises <strip> et ainsi que les espaces entre les balises et disparaissent car les nœuds textuels qui les contiennent ne contiennent que des caractères d'espacement.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<list>
  <sublist>
    <preserve>
      <p>a b c</p>   <p> a b c </p>
    </preserve>
    <!-- La valeur "preserve" de l'attribut xml:space
         inhibe la suppression des espaces -->
    <strip xml:space="preserve">
      <p>a b c</p>   <p> a b c </p>
    </strip>
    <strip><p>a b c</p><p> a b c </p></strip>
  </sublist>
  <sublist xml:space="preserve">
    <!-- La valeur "preserve" de l'attribut xml:space du parent
         inhibe la suppression des espaces -->
    <strip>
      <p>a b c</p>   <p> a b c </p>
    </strip>
    <!-- La valeur "default" de l'attribut xml:space masque
         la valeur "preserve" de l'attribut du parent -->
    <strip xml:space="default"><p>a b c</p><p> a b c </p></strip>
  </sublist>
</list>
```

8.5. Définition et application de règles

Les deux éléments qui constituent le cœur de XSLT sont les éléments xsl:template et xsl:apply-templates qui permettent respectivement de définir des règles et de les appliquer à des nœuds. Chaque

définition de règle par un élément `xsl:template` précise, par un motif XPath, les nœuds sur lesquels elle s'applique. Chaque application de règles par un élément `xsl:apply-templates` précise les nœuds actifs auxquels doit être appliquée une règle mais ne spécifie pas du tout la règle à appliquer. Le choix de la règle à appliquer à chaque nœud est fait par le processeur XSLT en fonction de priorités entre les règles [Section 8.5.3]. Ces priorités sont déterminées à partir des motifs XPath de chaque règle. Ce principe de fonctionnement est le mode normal de fonctionnement de XSLT. Il est, cependant, utile à l'occasion d'appliquer une règle déterminée à un nœud. L'élément `xsl:call-template` permet l'appel explicite d'une règle par son nom.

Les définitions de règles sont globales. Tous les éléments `xsl:template` sont enfants de l'élément racine `xsl:stylesheet` et la portée des règles est l'intégralité de la feuille de style. Au contraire, les éléments `xsl:apply-templates` et `xsl:call-template` ne peuvent apparaître que dans le contenu d'un élément `xsl:template`.

8.5.1. Définition de règles

L'élément `xsl:template` permet de définir une règle. Cet élément est nécessairement enfant de l'élément racine `xsl:stylesheet`. L'attribut `match` qui contient un motif XPath [Section 6.8] définit le contexte de la règle, c'est-à-dire, les nœuds sur lesquels elle s'applique. L'attribut `name` donne le nom de la règle. Un de ces deux attributs doit être présent mais les deux peuvent être simultanément présents. L'attribut `match` permet à la règle d'être invoquée implicitement par un élément `xsl:apply-templates` alors que l'attribut `name` permet à la règle d'être appelée explicitement par un élément `xsl:call-template`. Le contenu de l'élément `xsl:template` est le fragment de document à insérer dans le document résultat lors de l'application de la règle. L'élément `xsl:template` peut aussi contenir un attribut `priority` pour fixer la priorité de la règle ainsi qu'un attribut `mode` pour préciser dans quels modes elle peut être appliquée.

Le fragment de programme suivant définit une règle. 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`.

```
<xsl:template match="author|year|publisher">
  ...
</xsl:template>
```

Lorsqu'une règle est appliquée à un nœud du document source, ce nœud devient le nœud courant du focus [Section 6.1.5.2]. Les expressions XPath sont alors évaluées à partir de ce nœud qui est retourné par l'expression `'.'`. Certains éléments comme `xsl:for-each` [Section 8.7.3] ou les filtres [Section 6.4.2] peuvent localement changer le focus et le nœud courant. Le nœud sur lequel est appliquée la règle peut encore être recupéré par un appel à la fonction XPath `current()`.

8.5.2. Application implicite

L'élément `xsl:apply-templates` provoque l'application de règles sur les nœuds sélectionnés par son attribut `select` qui contient une expression XPath. L'évaluation de cette expression par rapport au nœud courant donne une liste de nœuds du document source. À chacun de ces nœuds, une règle choisie par le processeur XSLT est

appliquée. Le résultat de ces application de règles remplace alors l'élément `xsl:apply-templates` dans le contenu de l'élément `xsl:template` pour former le résultat de la règle en cours d'application.

L'expression XPath de l'attribut `select` sélectionne, en général, plusieurs nœuds. Sur chacun de ces nœuds, est appliquée une seule règle dépendant du nœud. La règle est choisie parmi les règles où il y a concordance entre le motif XPath de l'attribut `match` et le nœud. Lors de l'application la règle choisie, chaque nœud sélectionné devient le nœud courant du focus. Pour appliquer plusieurs règles à un même nœud, il faut plusieurs éléments `xsl:apply-templates`. Il est, en particulier, possible d'avoir recours à des modes [Section 8.11] pour appliquer des règles différentes.

La valeur par défaut de l'attribut `select` est `node()` qui sélectionne tous les enfants du nœud courant, y compris les nœuds textuels, les commentaires et les instructions de traitement. Pour sélectionner uniquement les enfants qui sont des éléments, il faut mettre la valeur `'*'`. Pour sélectionner tous les attributs du nœud courant, il faut mettre la valeur `@*` qui est l'abréviation de `attribute::*`. Il est, bien sûr, possible de mettre toute expression XPath comme `ancestor-or-self::p[@xml:lang][1]/@xml:lang`. Une partie importante de la programmation XSLT réside dans les choix judicieux des valeurs des attributs `select` des éléments `xsl:apply-templates` ainsi que des valeurs des attributs `match` des éléments `xsl:template`.

Dans la feuille de style suivante, la première règle qui s'applique à la racine `'/'` crée la structure du document XHTML. Les enfants `li` de l'élément `ul` sont créés par les applications de règles sur les éléments `book` provoquées par le premier élément `xsl:apply-templates`. Les contenus des éléments `li` sont construits par les applications de règles sur les enfants des éléments `book` provoquées par le second élément `xsl:apply-templates`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    <html>
      <head><title>Bibliographie</title></head>
      <body>
 <h1>Bibliographie</h1>
 <ul><xsl:apply-templates select="bibliography/book"/></ul>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="book">
    <!-- Une entrée (item) de la liste par livre -->
    <li><xsl:apply-templates select="*"/></li>
  </xsl:template>
  ...
</xsl:stylesheet>
```

Les nœuds sur lesquels sont appliqués les règles sont très souvent des descendants et même des enfants du nœud courant mais ils peuvent également être très éloignés dans le document. Dans l'exemple suivant, les éléments sont retournés par la fonction XPath `id`. Il faut remarquer que l'élément `xsl:for-each` change le focus et que l'expression XPath `'.'` ne désigne plus le nœud courant sur lequel s'applique la règle.

```
<xsl:template match="dbk:co">
  <xsl:text>\includegraphics[scale=0.25]{images/callouts/</xsl:text>
  <xsl:number level="single" count="dbk:co" format="1"/>
  <xsl:text>.pdf}</xsl:text>
</xsl:template>
w<xsl:template match="dbk:callout">
  <xsl:text>\item[</xsl:text>
  <!-- Parcours des éléments référencés par l'attribut arearefs -->
  <xsl:for-each select="id(@arearefs)">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
```

```
  <xsl:text>]</xsl:text>
  <xsl:apply-templates/>
</xsl:template>
```

Les nœuds sélectionnés par l'expression XPath de l'attribut `select` sont normalement traités dans l'ordre du document [Section 6.1.2]. L'élément `xsl:apply-templates` peut, néanmoins, contenir un ou plusieurs éléments `sort` pour effectuer un tri [Section 8.8] des nœuds sélectionnés.

8.5.3. Priorités

Lorsque plusieurs règles peuvent s'appliquer à un même nœud, le processeur XSLT choisit la plus appropriée parmi celles-ci. Le choix de la règle est d'abord dicté par les priorités d'import [Section 8.15] entre les feuilles de style puis par les priorités entre les règles.

Le choix de la règle à appliquer s'effectue en deux étapes. La première étape consiste à ne conserver que les règles de la feuille de style de priorité d'import maximale parmi les règles applicables. Lorsqu'une feuille de style est importée par une autre feuille de style, elle a une priorité d'import inférieure à celle qui l'importe. L'ordre des imports a également une incidence sur les priorités d'import.

La seconde étape consiste à choisir, parmi les règles restantes, celle qui a la priorité maximale. C'est une erreur s'il y en a plusieurs de priorité maximale. Le processeur XSLT peut signaler l'erreur ou continuer en choisissant une des règles. La priorité d'une règle est un nombre décimal. Elle peut est fixée par un attribut `priority` de l'élément `xsl:template`. Sinon, elle prend une valeur par défaut qui dépend de la forme du motif [Section 6.8] contenu dans l'attribut `match`. Les priorités par défaut prennent des valeurs entre `-0.5` et `0.5`. Lorsque le motif est de la forme *`expr-1 | ... | expr-N`*, le processeur considère chacun des motifs *`expr-i`* de façon indépendante. Il fait comme si la règle était répétée pour chacun de ces motifs. L'axe n'a aucune incidence sur la priorité par défaut. Celle-ci est déterminée par les règles ci-dessous.

`-0.5`
: pour les motifs très généraux de la forme `*`, `@*`, `node()`, `text()`, `comment()` ainsi que le motif `/`,

`-0.25`
: pour les motifs de la forme `*:`*`name`* ou *`prefix`*`:*` comme `*:p` ou `dbk:*`,

`0`
: pour les motifs de la forme *`name`* ou `@`*`name`* comme `section` ou `@type`,

`0.5`
: pour tous les autres motifs comme `chapter/section` ou `section[@type]`.

8.5.4. Application de règles moins prioritaires

C'est normalement la règle de priorité maximale [Section 8.5.3] qui est appliquée à un nœud lorsque le processus est initié par un élément `xsl:apply-templates`. Il existe également les deux éléments `xsl:next-match` et `xsl:apply-imports` qui permettent d'appliquer une règle de priorité inférieure. Ces deux éléments appliquent implicitement une règle à l'élément courant et il n'ont donc pas d'attribut `select`.

L'élément `xsl:next-match` applique à l'élément courant la règle qui se trouvait juste avant la règle courante dans l'ordre des priorités croissantes. Cet élément est souvent utilisé pour définir une règle pour un cas particulier tout en réutilisant la règle pour le cas général. Dans l'exemple suivant, la règle spécifique pour les éléments `para` avec un attribut `role` égal à `right` ajoute un élément `div` avec un attribut `style`. Elle appelle la règle générale pour la règle pour les éléments DocBook `para`.

```
<!-- Paragraphes -->
<xsl:template match="dbk:para">
  <p xsl:use-attribute-sets="para"><xsl:apply-templates/></p>
</xsl:template>
<!-- Paragraphes alignés à droite -->
<xsl:template match="dbk:para[@role='right']">
```

```
  <div style="text-align: right">
    <xsl:next-match/>
  </div>
</xsl:template>
```

L'élément `xsl:apply-imports` permet d'appliquer au nœud courant une règle provenant d'une feuille de style importée [Section 8.15] par la feuille de style contenant la règle en cours. La règle appliquée est la règle ayant une priorité maximale parmi les règles provenant des feuilles de style importées. Cet élément est souvent utilisé pour redéfinir une règle d'une feuille de style importée tout en utilisant la règle redéfinie. Dans l'exemple suivant, la feuille de style `general.xsl` contient une règle pour les éléments DocBook `para`.

```
<!-- Feuille de style general.xsl -->
<!-- Paragraphes -->
<xsl:template match="dbk:para">
  <p xsl:use-attribute-sets="para"><xsl:apply-templates/></p>
</xsl:template>
```

La feuille de style `main.xsl` importe la feuille de style `general.xsl` [Section 8.15]. Elle contient une nouvelle règle pour les éléments DocBook `para`. Cette règle a une priorité d'import supérieure et elle est donc appliquée en priorité. Cette règle fait appel à la règle contenue dans `general.xsl` par l'élément `xsl:apply-imports`.

```
<!-- Feuille de style main.xsl -->
<!-- Import de la feuille de style general.xsl -->
<xsl:import href="general.xsl"/>
<!-- Paragraphes alignés à gauche -->
<xsl:template match="dbk:para">
  <div style="text-align: left">
    <xsl:apply-imports/>
  </div>
</xsl:template>
```

8.5.5. Application explicite

L'élément `xsl:call-template` provoque l'application de la règle dont le nom est donné par l'attribut `name`. Contrairement à l'élément `xsl:apply-templates`, le contexte et, en particulier, le nœud courant ne sont pas modifiés pour l'application de la règle nommée. Le résultat de l'application de la règle remplace alors l'élément `xsl:call-template` dans le contenu de l'élément `xsl:template` pour former le résultat de la règle en cours d'application.

L'utilisation de `xsl:call-template` est courant pour factoriser des parties communes à des règles. Lorsque plusieurs règles partagent un fragment, il est approprié de placer ce fragment dans une nouvelle règle nommée puis de l'utiliser en invoquant, à plusieurs reprises, cette règle avec `xsl:call-template`.

Dans la feuille de style suivante, les différentes règles pour traiter les éléments `title`, `url` et les autres enfants de l'élément `book` font appel à la règle `comma` pour ajouter une virgule si l'enfant n'est pas le dernier. Il est important que le focus soit préservé à l'appel de cette règle pour que le test `position() != last()` fonctionne correctement.

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

  <xsl:template match="book">
    <!-- Une entrée (item) de la liste par livre -->
    <li><xsl:apply-templates select="*"/></li>
  </xsl:template>
  <xsl:template match="title">
    <!-- Titre du livre en italique -->
```

```
    <i><xsl:apply-templates/></i>
    <xsl:call-template name="comma"/>
  </xsl:template>
  <xsl:template match="url">
    <!-- URL du livre en police fixe -->
    <tt><xsl:apply-templates/></tt>
    <xsl:call-template name="comma"/>
  </xsl:template>
  <xsl:template match="*">
    <xsl:apply-templates/>
    <xsl:call-template name="comma"/>
  </xsl:template>
  <xsl:template name="comma">
    <!-- Virgule si ce n'est pas le dernier -->
    <xsl:if test="position() != last()">
      <xsl:text>, </xsl:text>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Les deux éléments `xsl:apply-templates` et `xsl:call-template` peuvent contenir des éléments `xsl:with-param` pour spécifier les valeurs de paramètres [Section 8.9.2] que la règle appliquée peut avoir déclarés avec l'élément `xsl:param`.

8.6. Construction de contenu

Chaque application de règle de la feuille de style produit un fragment du résultat. Ce fragment est construit à partir du contenu de l'élément `xsl:template` et d'autres éléments permettant d'insérer d'autres nœuds calculés.

8.6.1. Contenu brut

Lorsqu'une règle est appliquée, tout le contenu de l'élément `xsl:template` est recopié dans le résultat à l'exception des éléments de l'espace de noms XSLT. Ces derniers sont, en effet, évalués par le processeur puis remplacés par leur valeur. Tout les autres éléments ainsi que leur contenu se retrouvent recopiés à l'identique dans le document résultat.

Cette fonctionnalité est utilisée dans le premier exemple `Hello, Word!` [Section 8.2]. L'unique élément `xsl:template` contient le document XHTML produit par la feuille de style. Les espaces de noms jouent ici un rôle essentiel puisqu'ils permettent de distinguer les directives de traitement pour le processeur XSLT (c'est-à-dire les éléments XSLT) des autres éléments.

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

L'insertion d'un contenu fixe, comme dans l'exemple précédent, est très limité. Le résultat attendu dépend généralement du document source. Dans la règle ci-dessous, le contenu de l'élément `h1` provient du document source. Ce texte est extrait du document source grâce à l'élément `xsl:value-of` décrit ci-dessous [Section 8.6.5]. Il est le contenu textuel de l'élément racine `text` du document source.

```
<xsl:template match="/">
```

```
  <html>
    <head>
      <title>Localized Hello World !</title>
    </head>
    <body>
      <h1><xsl:value-of select="text"/></h1>
    </body>
  </html>
</xsl:template>
```

Si la feuille de style ainsi modifiée est appliquée au document suivant, on obtient un document XHTML où le contenu de l'élément `h1` est maintenant `Bonjour !`.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<!-- Élément text contenant le texte inséré dans l'élément h1 -->
<text>Bonjour !</text>
```

Du contenu peut aussi être construit par un élément `xsl:apply-templates`. Ce contenu est alors le résultat de l'application de règles aux éléments sélectionnés par l'attribut `select` de `xsl:apply-templates`. On reprend le document `bibliography.xml` déjà utilisé au chapitre sur syntaxe [Chapitre 2].

La feuille de style suivante transforme le document `bibliography.xml` en un document XHTML qui présente la bibliographie sous forme d'une liste avec une mise en forme minimaliste Cette feuille de style fonctionne de la manière suivante. La première règle est appliquée à la racine du document source. Elle produit le squelette du document XHTML avec en particulier un élément `ul` pour contenir la liste des livres. Le contenu de cet élément `ul` est obtenu en appliquant une règle à chacun des éléments `book`. La seconde règle de la feuille de style est la règle qui est appliquée aux éléments `book`. Pour chacun de ces éléments, elle produit un élément `li` qui s'insère dans le contenu de l'élément `ul`. Le contenu de l'élément `li` est, à nouveau, produit en appliquant une règle aux enfants de l'élément `book`. C'est la dernière règle qui s'applique à chacun de ces éléments. Cette règle se contente de rappeler récursivement une règle sur leurs enfants. La règle par défaut recopie alors les contenus textuels de ces élément.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
<!-- Règle pour la racine qui construit le squelette -->
<xsl:template match="/">
  <html>
    <head>
      <title>Bibliographie</title>
    </head>
    <body>
      <h1>Bibliographie</h1>
      <ul><xsl:apply-templates select="bibliography/book"/></ul>
    </body>
  </html>
</xsl:template>
<!-- Règle pour les éléments book -->
<xsl:template match="book">
  <!-- Une entrée li de la liste par livre -->
  <li><xsl:apply-templates/></li>
</xsl:template>
<!-- Règle pour les autres éléments -->
<xsl:template match="*">
  <!-- Récupération du texte par la règle par défaut -->
  <xsl:apply-templates/>
</xsl:template>
</xsl:stylesheet>
```

En appliquant la feuille de syle précédente au document `bibliography.xml`, on obtient le document XHTML suivant dont l'indentation a été remaniée pour une meilleure présentation.

```
<?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>Bibliographie</title>
  </head>
  <body>
    <h1>Bibliographie</h1>
    <ul>
      <li>XML langage et appplications Alain Michard 2001 Eyrolles
        2-212-09206-7 http://www.editions-eyrolles/livres/michard/ </li>
      <li>Designing with web standards Jeffrey Zeldman 2003 New Riders
        0-7357-1201-8 </li>
      ...
    </ul>
  </body>
</html>
```

Dans la feuille de style précédente, plusieurs règles sont susceptibles de s'appliquer aux éléments `book` : la deuxième règle dont la valeur de l'attribut `match` est `book` et la troisième dont la valeur de l'attribut `match` est `'*'`. Le processeur XSLT choisit la deuxième en raison des priorités attribuées aux règles en fonction du motif contenu dans l'attribut `match`.

La présentation de la bibliographie en XHTML obtenue avec le feuille de style précédente est très sommaire. Il est possible de l'améliorer en mettant, par exemple, le titre en italique et l'URL en fonte fixe. Il suffit, pour cela, d'ajouter deux règles spécifiques pour les éléments `title` et `url`. Les deux nouvelles règles suivantes ont une priorité supérieure à la dernière règle de la feuille de style et elle est donc appliquée aux éléments `title` et `url`.

```
<!-- Règle pour les éléments title -->
<xsl:template match="title">
  <!-- Titre en italique -->
  <i><xsl:apply-templates/></i>
</xsl:template>
<!-- Règle pour les éléments url -->
<xsl:template match="url">
  <!-- URL en fonte fixe -->
  <tt><xsl:apply-templates/></tt>
</xsl:template>
```

Beaucoup de la programmation XSLT s'effectue en jouant sur les éléments sélectionnés par les attributs `match` des éléments `xsl:apply-templates`. La feuille de style suivante présente la bibliographie en XHTML sous la forme de deux listes, une pour les livres en français et une autre pour les livres en anglais. Le changement par rapport à la feuille de style précédente se situe dans la règle appliquée à la racine. Celle-ci contient maintenant deux éléments `xsl:apply-templates`. Le premier active les livres en français et le second les livres en anglais.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
  <html>
    <head>
      <title>Bibliographie Français/Anglais</title>
    </head>
    <body>
    <h1>Bibliographie en français</h1>
```

```
    <!-- Traitement des livres avec l'attribut lang="fr" -->
    <ul><xsl:apply-templates select="bibliography/book[@lang='fr']"/></ul>
    <h1>Bibliographie en anglais</h1>
    <!-- Traitement des livres avec l'attribut lang="en" -->
    <ul><xsl:apply-templates select="bibliography/book[@lang='en']"/></ul>
    </body>
  </html>
  </xsl:template>
  <!-- Les autres règles sont inchangées -->
  ...
</xsl:stylesheet>
```

8.6.1.1. Substitution d'espaces de noms

Il est parfois souhaité qu'une feuille de style XSLT produise une autre feuille de style comme document résultat. Ce mécanisme est, en particulier, mis en œuvre par les schematrons [Chapitre 7]. Le problème est que l'espace de noms XSLT est justement utilisé par le processeur pour distinguer les éléments XSLT qui doivent être exécutés des autres éléments qui doivent être recopiés dans le résultat. Il est alors, a priori, impossible de mettre des éléments XSLT dans le document résultat. L'élément `xsl:namespace-alias` permet de contourner cette difficulté. Il provoque la conversion d'un espace de noms de la feuille de style en un autre espace de noms dans le résultat. Cet élément doit être enfant de l'élément racine `xsl:stylesheet` de la feuille de de style. Ses attributs `stylesheet-prefix` et `result-prefix` donnent respectivement les préfixes associés à l'espace de noms à convertir et de l'espace de noms cible. Ces deux attributs peuvent prendre la valeur particulière `#default` pour spécifier l'espace de noms par défaut.

Pour produire des éléments XSLT dans le document résultat, la feuille de style déclare, d'une part, l'espace de noms XSLT associé, comme d'habitude, au préfixe `xsl` et elle déclare, d'autre part, un autre espace de noms associé à un préfixe arbitraire `tns`. Les éléments XSLT devant être recopiés dans le résultat sont placés dans l'espace de noms associé au préfixe `tns`. L'élément `xsl:namespace-alias` assure la conversion de cet autre espace de noms en l'espace de noms XSLT dans le résultat. L'exemple suivant est une feuille de style produisant une autre feuille de style produisant, à son tour, un document XHTML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:tns="http://www.liafa.jussieu.fr/~carton">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <!-- Le préfix tns est transformé en xsl dans la sortie -->
  <xsl:namespace-alias stylesheet-prefix="tns" result-prefix="xsl"/>
  <xsl:template match="/">
    <tns:stylesheet version="2.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <tns:output method="xhtml" encoding="iso-8859-1" indent="yes"/>
      <tns:template match="/">
        <html>
          <head>
            <title>Exemple d'expressions XPath</title>
          </head>
          <body>
            <h1>Exemple d'expressions XPath</h1>
            <xsl:apply-templates select="operators/operator"/>
          </body>
        </html>
      </tns:template>
    </tns:stylesheet>
  </xsl:template>
  ...
```

8.6.2. Règles par défaut

Il existe des règles par défaut pour traiter chacun des nœuds possibles d'un document. Celles-ci simplifient l'écriture de feuilles de style très simples en fournissant un traitement adapté à la plupart des nœuds. Ces règles ont la priorité la plus faible. Elles ne s'appliquent à un nœud que si la feuille de style ne contient aucune règle pouvant s'appliquer à ce nœud. L'effet de ces différentes règles est le suivant. Elles suppriment les instructions de traitement et les commentaires. Elles recopient dans le résultat le contenu des nœuds textuels et les valeurs des attributs. Le traitement d'un élément par ces règles est d'appliquer récursivement une règle à ses enfants, ce qui exclut les attributs.

La feuille de style suivante est la feuille de style minimale sans aucune définition de règles. Elle est néanmoins utile grâce à la présence des règles par défaut.

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

En appliquant cette feuille de style minimale au document `bibliography.xml`, on obtient le résultat suivant.

```
<?xml version="1.0" encoding="UTF-8"?>

    XML langage et applications
    Alain Michard
    2001
    Eyrolles
    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
    2002
    Eyrolles
    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.

```
<!-- Règle pour la racine et les éléments -->
<xsl:template match="/|*">
  <!-- Traitement récursif des enfants -->
  <!-- La valeur par défaut de l'attribut select est node() -->
  <xsl:apply-templates/>
</xsl:template>

<!-- Règle pour nœuds textuels et les attributs -->
<xsl:template match="text()|@*">
```

```
  <!-- La valeur textuelle est retournée -->
  <xsl:value-of select="."/>
</xsl:template>

<!-- Règle pour les intructions de traitement et les commentaires -->
<!-- Suppression de ceux-ci car cette règle ne retourne rien -->
<xsl:template match="processing-instruction()|comment()"/>
```

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.

8.6.3. 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` [Section 8.6.6] pour ajouter un attribut.

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

Pour insérer des accolades ouvrantes ou fermantes dans la valeur d'un attribut, il faut simplement doubler les accolades et écrire `'{{'` et `'}}'`. Si la règle XSLT suivante

```
<xsl:template match="*">
  <braces id="{@id}" esc="{{@id}}" escid="{{{@id}}}"/>
</xsl:template>
```

est appliquée à un élément `<braces id="JB007"/>`, on obtient l'élément suivant. La chaîne `{@id}` est remplacée par la valeur de l'attribut `id`. En revanche, la chaîne `{{@id}}` donne la valeur `{@id}` où chaque paire d'accolades donne une seule accolade. Finalement, la chaîne `{{{@id}}}` combine les deux comportements.

```
<braces id="JB007" esc="{@id}" escid="{JB007}"/>
```

Les expressions XPath en attribut avec des structures de contrôle XPath [Section 6.6] sont souvent plus concises que les constructions équivalentes en XSLT.

Les expressions XPath en attribut apparaissent normalement dans les attribut des éléments hors de l'espace de noms XSLT. Ils peuvent égalemement apparaître comme valeur de quelques attributs d'éléments XSLT. C'est, par exemple, le cas de l'attribut `name` des éléments `xsl:element` et `xsl:attribute` [Section 8.6.6].

8.6.4. Texte brut

L'élément `xsl:text` utilise son contenu pour créer un nœud texte dans le document résultat. Les caractères spéciaux [Section 2.2.1] `'<'`, `'>'` et `'&'` sont automatiquement remplacés par les entités prédéfinies [Section 3.5.1.2] correspondantes si la valeur de l'attribut `method` de l'élément `output` n'est pas `text`. Si la valeur cet attribut est `text`, les caractères spéciaux sont écrits tels quels.

```
<xsl:text>Texte et entités '&amp;lt;', '&amp;gt;' et '&amp;amp;'</xsl:text>
```

La présentation de la bibliographie en XHTML peut être améliorée par l'ajout de virgules `','` entre les propriétés titre, auteur, … de chacun des livres. L'élément `xsl:text` est alors nécessaire pour insérer les virgules. Il faut, en outre, gérer l'absence de virgule à la fin grâce à l'élément `xsl:if` [Section 8.7.1].

```
<!-- Règle pour les autres éléments -->
<xsl:template match="*">
  <xsl:apply-templates/>
  <!-- Virgule après ces éléments -->
  <xsl:if test="position() != last()">
    <xsl:text>, </xsl:text>
  </xsl:if>
</xsl:template>
```

Pour que le test `position() != last()` fonctionne correctement, il faut que la règle appliquée aux éléments `book` n'active pas les nœuds textuels contenant des caractères d'espacements. Il faut, pour cela, remplacer la valeur par défaut de l'attribut `select` par la valeur `'*'` qui ne sélectionne que les enfants qui sont des éléments et pas ceux de type `text()`.

```
<!-- Règle pour les éléments book -->
<xsl:template match="book">
  <!-- Une entrée li de la liste par livre -->
  <li><xsl:apply-templates select="*"/></li>
</xsl:template>
```

8.6.5. Expression XPath

L'élément `xsl:value-of` crée un un nœud texte dont le contenu est calculé. L'attribut `select` doit contenir une expression XPath qui est évaluée pour donner une liste de valeurs. Chacune de ces valeurs est convertie en une chaîne de caractères. Lorsqu'une valeur est un nœud, la conversion retourne la valeur textuelle [Section 6.1.1.1] de celui-ci. Le texte est alors obtenu en concaténant ces différentes chaînes. Un espace `' '` est inséré, par défaut, entre ces chaînes. Le caractère inséré peut être changé en donnant une valeur à l'attribut `separator` de l'élément `xsl:value-of`.

La feuille se style suivante collecte des valeurs des enfants de l'élément racine `list` pour former le contenu d'un élément `values`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <values><xsl:value-of select="list/*" separator=","/></values>
  </xsl:template>
</xsl:stylesheet>
```

Le document suivant comporte un élément racine avec des enfants `item` contenant des valeurs `1`, `2` et `3`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<list><item>1</item><item>2</item><item>3</item></list>
```

En appliquant la feuille de style précédente au document précédent, on obtient le document suivant où apparaissent les trois valeurs `1`, `2` et `3` séparées par des virgules `','`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<values>1,2,3</values>
```

Le fonctionnement de `xsl:value-of` de XSLT 1.0 est différent et constitue une incompatibilité avec XSLT 2.0. Avec XSLT 1.0, seule la première valeur de la liste donnée par l'évaluation de l'expression de l'attribut `select` est prise en compte. Les autres valeurs sont ignorées. En appliquant la feuille de style précédente au document précédent avec un processeur XSLT 1.0, on obtient le document suivant où seule la valeur `1` apparaît.

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

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"/>
```

8.6.6. Élément et attribut

Tout élément contenu dans un élément `xsl:template` et n'appartenant pas à l'espace de nom `xsl` des feuilles de style est recopié à l'identique lors de l'application de la règle. Ceci permet d'ajouter facilement des éléments et des attributs dont les noms sont fixes. Il est parfois nécessaire d'ajouter des éléments et/ou des attributs dont les noms sont calculés dynamiquement. Les éléments `xsl:element` et `xsl:attribute` permettent respectivement de construire un nœud pour un élément ou un attribut. Le nom de l'élément ou de l'attribut est déterminé dynamiquement par l'attribut `name` de `xsl:element` et `xsl:attribute`. Cet attribut contient une expression XPath en attribut [Section 8.6.3]. L'évaluation des expressions XPath délimitées par des accolades `'{'` et `'}'` fournit le nom de l'élément ou de l'attribut. Le contenu de l'élément ou la valeur de l'attribut sont donnés par l'attribut `select` ou par le contenu des éléments `xsl:element` et `xsl:attribute`. Ces deux possibilités s'excluent mutuellement. L'attribut `select` doit contenir une expression XPath.

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

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

L'utilisation des éléments `xsl:element` et `xsl:element` est pratique lorsque l'apparition de l'élément ou de l'attribut est conditionnel. 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.

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

8.6.6.1. Groupe d'attributs

Il est fréquent qu'une feuille de style ajoute systématiquement les mêmes attributs à des éléments. Les *groupes d'attributs* définissent des ensembles d'attributs qu'il est, ensuite, facile d'utiliser pour ajouter des attributs aux éléments construits. Cette technique accroît, en outre, la flexibilité des feuilles de style. Le groupe peut être redéfini au niveau global, permettant ainsi, d'adapter la feuille de style sans modifier les règles.

Un groupe d'attributs est introduit par l'élément `xsl:attribute-set` dont l'attribut `name` précise le nom. Cet éléments doit être un enfant de l'élément racine `xsl:stylesheet` de la feuille de style. L'élément `xsl:attribute-set` contient des déclarations d'attributs introduites par des éléments `xsl:attribute`. Le groupe d'attribut est ensuite utilisé par l'intermédiaire de l'attribut `use-attribute-sets`. Cet attribut peut apparaître dans certains éléments XSLT comme `xsl:element` mais il peut également apparaître dans des éléments hors de l'espace de noms XSLT. Dans ce dernier cas, son nom doit être qualifié pour qu'il fasse partie de l'espace de noms XSLT. L'attribut `use-attribute-sets` contient une liste de noms de groupes d'attributs séparés par des espaces.

Un groupe d'attributs peut réutiliser d'autres groupes d'attributs. Il contient alors tous les attributs de ces groupes en plus de ceux qu'il déclare explicitement. Les noms des groupes utilisés sont donnés par un attribut `use-attribute-sets` de l'élément `xsl:attribute-set`.

La feuille de style suivante permet la transformation d'un sous-ensemble, délibérément très réduit, de DocBook en XHTML. Le sous-ensemble est constitué des éléments `book`, `title`, `chapter`, `sect1` et `para`. La feuille de style définit trois groupes d'attributs de noms `text`, `title` et `para`. Le premier est prévu pour contenir des attributs généraux pour les éléments XHTML contenant du texte. Il est utilisé dans les définitions des deux autres

groupes `title` et `para`. Le groupe `title` est utilisé pour ajouter des attributs aux éléments `h1` et `h2` construits par la feuille de style. Le groupe `para` est utilisé pour ajouter des attributs aux éléments `p`.

Il faut remarquer que de nom l'attribut `use-attribute-sets` n'est pas qualifié lorsque celui-ci apparaît dans un élément XSLT comme `xsl:attribute-set` alors qu'il est qualifié lorsque celui-ci apparaît dans des éléments hors de l'espace de noms XSLT comme `h1` et `p`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dbk="http://docbook.org/ns/docbook"
                xmlns="http://www.w3.org/1999/xhtml">
  <xsl:output ... />
  <!-- Définition d'un groupe d'attribut text -->
  <xsl:attribute-set name="text">
    <xsl:attribute name="align">left</xsl:attribute>
  </xsl:attribute-set>
  <!-- Définition d'un groupe d'attribut para pour les paragraphes -->
  <xsl:attribute-set name="para" use-attribute-sets="text"/>
  <!-- Définition d'un groupe d'attribut title pour les titres -->
  <xsl:attribute-set name="title" use-attribute-sets="text">
    <xsl:attribute name="id" select="generate-id()"/>
  </xsl:attribute-set>

  <xsl:template match="/">
    <xsl:comment>Generated by dbk2html.xsl</xsl:comment>
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head><title><xsl:value-of select="dbk:book/dbk:title"/></title></head>
      <body><xsl:apply-templates/></body>
    </html>
  </xsl:template>
  <!-- Éléments non traités -->
  <xsl:template match="dbk:title|dbk:subtitle"/>
  <!-- Livre -->
  <xsl:template match="dbk:book">
    <xsl:apply-templates/>
  </xsl:template>
  <!-- Chapitres -->
  <xsl:template match="dbk:chapter">
    <h1 xsl:use-attribute-sets="title"><xsl:value-of select="dbk:title"/></h1>
    <xsl:apply-templates/>
  </xsl:template>
  <!-- Sections de niveau 1 -->
  <xsl:template match="dbk:sect1">
    <h2 xsl:use-attribute-sets="title"><xsl:value-of select="dbk:title"/></h2>
    <xsl:apply-templates/>
  </xsl:template>
  <!-- Paragraphes -->
  <xsl:template match="dbk:para">
    <p xsl:use-attribute-sets="para"><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

Un document produit par cette feuille de style contient de multiples occurrences des mêmes attributs avec les mêmes valeurs. Il est préférable d'utiliser CSS [Chapitre 10] qui permet de séparer le contenu de la présentation et de réduire, du même coup, la taille du fichier XHTML.

Une feuille de style peut importer la feuille de style précédente tout en redéfinissant certains groupes d'attributs pour en modifier le comportement.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Import de la feuille de style précédente -->
  <xsl:import href="dbk2html.xsl"/>
  <!-- Modification du groupe d'attributs text -->
  <xsl:attribute-set name="title">
    <!-- Nouvelle valeur de l'attribut align -->
    <xsl:attribute name="align">center</xsl:attribute>
  </xsl:attribute-set>
  <!-- Modification du groupe d'attributa text -->
  <xsl:attribute-set name="para">
    <!-- Ajout d'un d'attribut style -->
    <xsl:attribute name="style">font-family: serif</xsl:attribute>
  </xsl:attribute-set>
</xsl:stylesheet>
```

8.6.7. Commentaire et instruction de traitement

Les éléments `xsl:comment` et `xsl:processing-instruction` permettent respectivement de construire un nœud pour un commentaire [Section 2.7.5] ou une instruction de traitement [Section 2.7.6]. Le nom de l'instruction de traitement est déterminé dynamiquement par l'attribut `name` de `xsl:processing-instruction`. Cet attribut contient une expression XPath en attribut [Section 8.6.3]. Le texte du commentaire ou de l'instruction de traitement est donné par le contenu des éléments `xsl:comment` et `xsl:processing-instruction`.

```
<!-- Un commentaire dans la feuille de style -->
<xsl:comment>Un commentaire dans le document final</xsl:comment>
<xsl:processing-instruction name="dbhtml">
  filename="index.html"
</xsl:processing-instruction>
```

8.6.8. Liste d'objets

L'élément `xsl:sequence` construit une liste de valeurs déterminée par l'évaluation de l'expression XPath contenu dans l'attribut `select`.

Cet élément est souvent utilisé dans la déclaration d'une variable [Section 8.9] ou dans la définition d'une fonction d'extension [Section 8.10].

8.6.9. Copie superficielle

L'élément `xsl:copy` permet de copier le nœud courant dans le document résultat. Cette copie est dite *superficielle* car elle copie seulement le nœud ainsi que les déclarations d'espaces de noms nécessaires. L'élément `xsl:copy` n'a pas d'attribut `select` car c'est toujours le nœud courant qui est copié.

Les attributs (dans le cas où le nœud courant est un élément) et le contenu ne sont pas copiés. Ils doivent être ajoutés explicitement. Le contenu de l'élément `xsl:copy` définit le contenu de l'élément copié. C'est ici qu'il faut, par exemple, ajouter des éléments `xsl:copy-of` ou `xsl:apply-templates`.

La feuille de style suivante transforme le document `bibliography.xml` en remplaçant chaque attribut `lang` d'un élément `book` par un enfant `lang` ayant comme contenu la valeur de l'attribut. La feuille de style est constituée de deux règles, une pour les éléments `book` et une autre pour tous les autres éléments. Dans les deux règles, la copie de l'élément est réalisée par un élément `xsl:copy`. La copie des attributs est réalisée par un élément `xsl:copy-of` qui sélectionne explicitement les attributs. La copie des enfants est réalisée par un élément `xsl:apply-templates` pour un appel récursif des règles.

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

```
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <!-- Règle pour la racine et les éléments autres que book -->
  <xsl:template match="/|*">
    <xsl:copy>
      <!-- Copie explicite des attributs -->
      <xsl:copy-of select="@*"/>
      <!-- Copie explicite des enfants -->
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
  <!-- Règle pour les éléments book -->
  <xsl:template match="book">
    <xsl:copy>
      <!-- Copie explicite des attributs autres que 'lang' -->
      <xsl:copy-of select="@*[name() != 'lang']"/>
      <!-- Ajout de l'élément lang -->
      <lang><xsl:value-of select="@lang"/></lang>
      <!-- Copie explicite des enfants -->
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

En appliquant la feuille de style précédente au document `bibliography.xml`, on obtient le document suivant.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<bibliography>
  <book key="Michard01">
    <lang>fr</lang>
    <title>XML langage et appplications</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</lang>
    <title>Designing with web standards</title>
    <author>Jeffrey Zeldman</author>
    <year>2003</year>
    <publisher>New Riders</publisher>
    <isbn>0-7357-1201-8</isbn>
  </book>
  <!-- Fichier tronqué -->
  ...
</bibliography>
```

8.6.10. Copie profonde

L'élément `xsl:copy-of` permet de copier des nœuds sélectionnés ainsi que tous les descendants de ces nœuds dans le document résultat. Cette copie est dite *profonde* car tout le sous-arbre enraciné en un nœud sélectionné est copié. L'expression XPath contenue dans l'attribut `select` de `xsl:copy-of` détermine les nœuds à copier.

La feuille de style suivante transforme le document `bibliography.xml` en répartissant la bibliographie en deux parties dans des éléments `bibliodiv` contenant respectivement les livres en français et en anglais. L'unique règle de la feuille de style crée le squelette avec les deux éléments `bibliodiv` et copie les livres dans ces deux éléments grâce à deux éléments `xsl:copy-of`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="/">
    <bibliography>
      <bibliodiv>
        <xsl:copy-of select="bibliography/book[@lang='fr']"/>
      </bibliodiv>
      <bibliodiv>
        <xsl:copy-of select="bibliography/book[@lang='en']"/>
      </bibliodiv>
    </bibliography>
  </xsl:template>
</xsl:stylesheet>
```

8.6.11. Numérotation

Le rôle de l'élément `xsl:number` est double. Sa première fonction est de créer un entier ou une liste d'entiers pour numéroter 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 XPath `format-number()` [Section 8.6.12].

8.6.11.1. 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 recopiés 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, …`

Tableau 8.1. Formats de `xsl:number`

Il existe aussi des formats `w`, `W` et `Ww` permettant d'écrire les nombres 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.

8.6.11.2. Calcul du numéro

Le numéro calculé par `xsl:number` peut être donné de façon explicite par l'attribut `value` qui contient une expression XPath. L'évaluation de cette expression fournit le nombre résultat. Cette méthode permet d'avoir un contrôle total sur le numéro. Si l'attribut `value` est absent, le numéro est calculé grâce aux valeurs des attributs `level`, `count` et `from`. L'attribut `level` détermine le mode de calcul alors que les attributs `count` et `from` les éléments pris en compte. Chacun de ces trois attributs contient un motif XPath permettant de sélectionner des nœuds.

```
<xsl:number value="1 + count(preceding::*)" format="i"/>
```

L'attribut `level` peut prendre les valeurs `single`, `multiple` et `any`. Les modes de calcul `single` et `any` fournissent un seul entier alors que le mode `multiple` fournit une liste d'entiers. Dans ce dernier cas, le format

peut contenir plusieurs indicateurs de formats séparés par des caractères non alphanumériques comme `1.1.1` ou `[A-1-i]`.

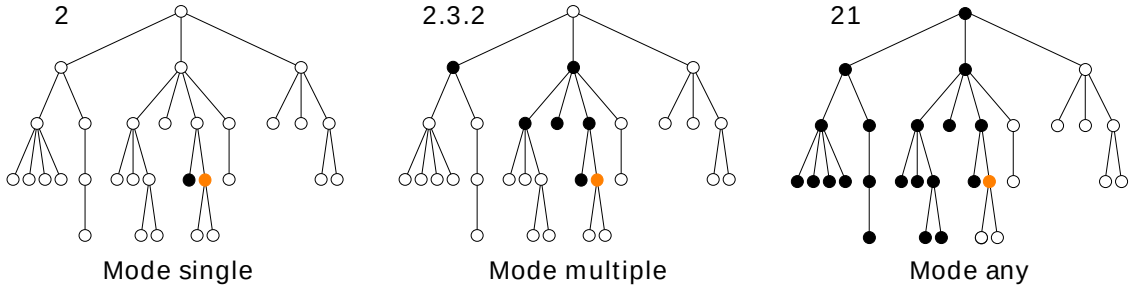


Figure 8.3. Modes de `xsl:number`

Dans le mode `single`, le numéro est égal au nombre (augmenté d'une unité pour commencer avec 1) de frères gauches du nœud courant qui satisfont le motif donné par `count`. Rappelons qu'un *frère* est un enfant du même nœud père et qu'il est *gauche* s'il précède le nœud courant dans le document.

La feuille de style suivante ajoute un numéro aux éléments `section`. Ce numéro est calculé dans le mode `single`.

```
<?xml version="1.0" encoding="us-ascii"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="*">
    <xsl:copy>
      <xsl:if test="name()='section'">
        <!-- format="1" est la valeur par défaut -->
        <xsl:number level="single" count="section" format="1"/>
      </xsl:if>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

On considère le document XML suivant qui représente le squelette d'un livre avec des chapitres, des sections et des sous-sections.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book>
  <chapter>
    <section>
      <section></section><section></section>
    </section>
    <section>
      <section></section><section></section>
    </section>
  </chapter>
  <chapter>
    <section>
      <section></section><section></section>
    </section>
    <section>
      <section></section><section></section>
    </section>
  </chapter>
</book>
```

En appliquant la feuille de style au document précédent, on obtient le document XML suivant. Chaque élément `section` contient en plus un numéro calculé par `xsl:number` en mode `single`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book>
  <chapter>
    <section>1
      <section>1</section><section>2</section>
    </section>
    <section>2
      <section>1</section><section>2</section>
    </section>
  </chapter>
  <chapter>
    <section>1
      <section>1</section><section>2</section>
    </section>
    <section>2
      <section>1</section><section>2</section>
    </section>
  </chapter>
</book>
```

Dans le mode `multiple`, l'élément `xsl:number` fournit une liste d'entiers qui est calculée de la façon suivante. Le *nœud de départ* est déterminé par l'attribut `from` qui contient un motif XPath. C'est l'ancêtre le plus proche du nœud courant qui satisfait le motif de l'attribut `from`. Ensuite, on considère chacun des ancêtres entre le nœud de départ et le nœud courant qui satisfait l'attribut `count`. Pour chacun de ces ancêtres, le nombre (plus une unité) de frères gauches qui satisfont le motif de `count` fournit un des entiers de la suite.

Si l'élément `xsl:number` de la feuille de style précédente est remplacé par l'élément suivant, on obtient le document ci-dessous. Comme le format est `A.1.i`, chaque section contient un numéro global formé d'un numéro de chapitre (`A`, `B`, …), d'un numéro de section (`1`, `2`, …) et d'un numéro de sous-section (`i`, `ii`, …). Ces différents numéros sont séparés par les points `'.'` qui sont repris du format.

```
 <xsl:number level="multiple" count="chapter|section" format="A.1.i"/>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book>
  <chapter>
    <section>A.1
      <section>A.1.i</section><section>A.1.ii</section>
    </section>
    <section>A.2
      <section>A.2.i</section><section>A.2.ii</section>
    </section>
  </chapter>
  <chapter>
    <section>B.1
      <section>B.1.i</section><section>B.1.ii</section>
    </section>
    <section>B.2
      <section>B.2.i</section><section>B.2.ii</section>
    </section>
  </chapter>
</book>
```

Dans le mode `any`, le *nœud de départ* est égal au dernier nœud avant le nœud courant qui vérifie le motif donné par l'attribut `from`. Par défaut le nœud de départ est la racine du document. Le numéro est égal au nombre (augmenté d'une unité pour commencer avec 1) de nœuds entre le nœud de départ et le nœud courant qui satisfont le motif donné par l'attribut `count`.

Si l'élément `xsl:number` de la feuille de style précédente est remplacé par l'élément suivant, on obtient le document ci-dessous. Chaque section contient son numéro d'ordre dans le document car la valeur par défaut de `from` est la racine du document.

```
<xsl:number level="any" count="section" format="1"/>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book>
  <chapter>
    <section>1
      <section>2</section><section>3</section>
    </section>
    <section>4
      <section>5</section><section>6</section>
    </section>
  </chapter>
  <chapter>
    <section>7
      <section>8</section><section>9</section>
    </section>
    <section>10
      <section>11</section><section>12</section>
    </section>
  </chapter>
</book>
```

L'élément `xsl:number` suivant utilise l'attribut `from` pour limiter la numérotation des éléments `section` aux contenus des éléments `chapter`. En appliquant la feuille de style précédente avec cet élément `xsl:number`, on obtient le document ci-dessous. Chaque section contient son numéro d'ordre dans le chapitre.

```
<xsl:number level="any" count="section" from="chapter" format="1"/>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book>
  <chapter>
    <section>1
      <section>2</section><section>3</section>
    </section>
    <section>4
      <section>5</section><section>6</section>
    </section>
  </chapter>
  <chapter>
    <section>1
      <section>2</section><section>3</section>
    </section>
    <section>4
      <section>5</section><section>6</section>
    </section>
  </chapter>
</book>
```

8.6.12. Formatage de nombres

L'élément `xsl:number` a des possibilités limitées pour formater un nombre de manière générale. Ils possède deux attributs `grouping-separator` et `grouping-size` dont les valeurs par défaut sont respectivement `','` et 3. Dans l'exemple suivant, l'entier 1234567 est formaté en `1.234.567`.

```
<xsl:number grouping-separator="." value="12345678"/>
```

La fonction XPath `format-number()` permet de formater un nombre entier ou décimal de façon plus précise. Le premier paramètre est le nombre à formater et le second est une chaîne de caractères qui décrit le formatage à effectuer. Cette fonction est inspirée de la classe `DecimalFormat` de Java et elle s'apparente, par les formats utilisés, à la fonction `printf` du langage C. Un troisième paramètre optionnel référence éventuellement un élément `xsl:decimal-format` pour changer la signification de certains caractères dans le format.

Lorsqu'aucun élément `xsl:decimal-format` n'est référencé, le format est une chaîne de caractères formée des caractères `'#'`, `'0'`, `'.'`, `','`, `'%'`, `'‰'` (de code hexadécimal `x2030`) et `';'`. La signification de ces différents caractères est donnée ci-dessous. La chaîne passée en second paramètre peut aussi contenir d'autres caractères qui sont recopiés inchangés dans le résultat.

`'#'`
: position pour un chiffre

`'0'`
: position pour un chiffre remplacé éventuellement par `0`

`'.'`
: position du point décimal

`','`
: position du séparateur de groupe (milliers, millions, …)

`';'`
: séparateur entre un format pour les nombres positifs et un format pour les nombres négatifs.

La chaîne de caractères passée en second paramètre à `format-number()` peut contenir deux formats séparés par un caractère `';'` comme `#000;-#00` par exemple. Le premier format `#000` est alors utilisé pour les nombres positifs et le second format `-#00` pour les nombres négatifs. Dans ce cas, le second format doit explicitement insérer le caractère `'-'` car c'est la valeur absolue du nombre qui est formatée. En formatant les nombres 12 et -12 avec ce format `#000;-#00`, on obtient respectivement `012` et `-12`. La table ci-dessous donne quelques exemples de résultats de la fonction `format-number()` avec des formats différents.

Nombre\Format	**##**	**####**	**#,#00.##**	**####.00**	**0000.00**
1	1	1	01	1.00	0001.00
123	123	123	123	123.00	0123.00
1234	1234	1234	1,234	1234.00	1234.00
12.34	12	12	12.34	12.34	0012.34
1.234	1	1	01.23	1.23	0001.23

Tableau 8.2. Résultats de `format-number()`

Les caractères `'%'` et `'‰'` permettent de formater une fraction entre 0 et 1 comme un pourcentage ou un millième. Le formatage du nombre 0.1234 avec les formats #%, #.##% et #‰ donne respectivement `12%`, `12.34%` et `123‰`.

L'élément `xsl:decimal-format` permet de changer les caractères utilisés dans le format. Cet élément déclare un objet qui définit l'interprétation des caractères dans le format utilisé par `format-number()`. L'attribut `name` donne le nom de l'objet qui est utilisé comme troisième paramètre de la fonction `format-number()`. Outre cet attribut, l'élément `xsl:decimal-format` possède plusieurs attributs permettant de spécifier les caractères utilisés pour marquer les différentes parties du nombre (point décimal, séparateur de groupe, etc …).

`decimal-separator`
: caractère pour marquer le point décimal (`'.'` par défaut)

`grouping-separator`
: caractère pour séparer les groupes (`','` par défaut)

`digit`
: caractère pour la position d'un chiffre (`'#'` par défaut)

`zero-digit`
caractère pour la position d'un chiffre remplacé par `'0'` (`'0'` par défaut)

`pattern-separator`
caractère pour séparer deux formats pour les nombres positifs et les nombres négatifs (`';'` par défaut)

`percent`
caractère pour formater les pourcentages (`'%'` par défaut)

`per-mille`
caractère pour formater les millièmes (`'‰'` par défaut)

Les éléments `xsl:decimal-format` doivent être enfants de l'élément racine `xsl:stylesheet` de la feuille de style et leur portée est globale. Ils peuvent être référencés par la fonction `format-number()` dans n'importe quelle expression XPath de la feuille de style.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- Format des nombres en englais -->
  <xsl:decimal-format name="en" decimal-separator="."
                                grouping-separator=","/>
  <!-- Format des nombres en français -->
  <xsl:decimal-format name="fr" decimal-separator=","
                                grouping-separator="."/>
  ...
  <price xml:lang="en-GB" currency="pound">
    <xsl:value-of select="format-number($price, '###,###,###.##', 'en')"/>
  </price>
  ...
  <price xml:lang="fr" currency="euro">
    <xsl:value-of select="format-number($price, '###.###.###,##', 'fr')"/>
  </price>
  ...
```

8.7. Structures de contrôle

Le langage XSLT propose, comme tout langage de programmation, des structures de contrôle permettant d'effectuer des tests et des boucles. Il contient les deux éléments `xsl:if` et `xsl:choose` pour les tests et les deux éléments `xsl:for-each` et `xsl:for-each-group` pour les boucles. L'élément `xsl:if` autorise un test sans alternative (pas d'élément `xsl:else`) alors que l'élément `xsl:choice` permet, au contraire un choix entre plusieurs alternatives. L'élément `xsl:for-each` pemet des boucles simples sur des nœuds sélectionnés. L'élément `xsl:for-each-group` permet de former des groupes à partir de nœuds sélectionnés puis de traiter successivement les différents groupes. Certaines constructions XSLT sont parfois réalisées de manière plus concise par des structures de contrôle XPath [Section 6.6] placées dans des attributs [Section 8.6.3].

8.7.1. Conditionnelle sans alternative

L'élément `xsl:if` permet de réaliser un test. De façon surprenante, cet élément ne propose pas d'alternative car il n'existe pas d'élément `xsl:else`. Lorsqu'une alternative est nécessaire, il faut utiliser l'élément `xsl:choose`.

La condition du test est une expression XPath contenue dans l'attribut `test` de `xsl:if`. Cette expression est évaluée puis convertie en valeur booléenne [Section 6.3.1]. Si le résultat est `true`, le contenu de l'élément `xsl:if` est pris en compte. Sinon, le contenu de l'élément `xsl:if` est ignoré.

```
<xsl:if test="not(position()=last())">
  <xsl:text>, </xsl:text>
</xsl:if>
```

8.7.2. Conditionnelle à alternatives multiples

L'élément `xsl:choose` permet de réaliser plusieurs tests consécutivement. Il contient des éléments `xsl:when` et éventuellement un élément `xsl:otherwise`. Chacun des éléments `xsl:when` possède un attribut `test` contenant une expression XPath servant de condition.

Les conditions contenues dans les attributs `test` des éléments `xsl:when` sont évaluées puis converties en valeur booléenne [Section 6.3.1] dans l'ordre des éléments `xsl:when` Le contenu du permier élément `xsl:when` dont la condition donne la valeur `true` est pris en compte et les contenus des autres `xsl:when` et d'un éventuel `xsl:otherwise` sont ignorés. Si aucune condition ne donne la valeur `true`, le contenu de l'élément `xsl:otherwise` est pris en compte et les contenus de tous les éléments `xsl:when` sont ignorés.

Le fragment de feuille de style retourne le contenu de l'enfant `title` de nœud courant si cet enfant existe ou construit un titre avec un numéro sinon.

```
<xsl:choose>
  <xsl:when test="title">
    <xsl:value-of select="title"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:text>Section </xsl:text>
    <xsl:number level="single" count="section"/>
  </xsl:otherwise>
</xsl:choose>
```

8.7.3. Itération simple

L'élément `xsl:for-each` permet de réaliser des boucles en XSLT. L'itération est déjà présente implicitement avec l'élément `xsl:apply-templates` puisqu'une règle est successivement appliquée à chacun des nœuds sélectionnés. L'élément `xsl:for-each` réalise une boucle de manière explicite.

L'attribut `select` détermine les objets traités. L'expression XPath qu'il contient est évaluée pour donner une liste `l` d'objets qui est, ensuite, parcourue. Le contenu de l'élément `xsl:for-each` est exécuté pour chacun des objets de la liste `l`. Le focus [Section 6.1.5.2] est modifié pendant l'exécution de `xsl:for-each`. À chaque itération, l'objet courant est fixé à un des objets de la liste `l`. La taille du contexte est également fixée à la longueur de `l` et la position dans le contexte est finalement fixée à la position de l'objet courant dans la liste `l`.

La feuille de style suivante présente la bibliographie `bibliography.xml` sous forme d'un tableau XHTML. Le tableau est construit par l'unique règle de la feuille de style. La première ligne du tableau avec des éléments `th` est ajoutée explicitement et les autres lignes avec des éléments `td` sont ajoutées par un élément `xsl:for-each` qui parcourt tous les éléments `book` de `bibliography.xml`. Le résultat de la fonction `position()` est formatée par l'élément `xsl:number` pour numéroter les lignes du tableau.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>Bibliographie en tableau</title>
      </head>
      <body>
        <h1>Bibliographie en tableau</h1>
        <table align="center" border="1" cellpadding="2" cellspacing="0">
          <tr>
            <th>Numéro</th>
            <th>Titre</th>
            <th>Auteur</th>
            <th>Éditeur</th>
            <th>Année</th>
          </tr>
```

```
          <xsl:for-each select="bibliography/book">
            <xsl:sort select="author" order="ascending"/>
            <tr>
              <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>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

En appliquant la feuille de style précédente au document `bibliography.xml`, on obtient le document suivant.

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Bibliographie en tableau</title>
  </head>
  <body>
    <h1>Bibliographie en tableau</h1>
    <table align="center" border="1" cellpadding="2" cellspacing="0">
      <tr>
        <th>Num&eacute;ro</th>
        <th>Titre</th>
        <th>Auteur</th>
        <th>&Eacute;diteur</th>
        <th>Ann&eacute;e</th>
      </tr>
      <tr>
        <td>1</td>
        <td>XML langage et appplications</td>
        <td>Alain Michard</td>
        <td>Eyrolles</td>
        <td>2001</td>
      </tr>
      <!-- Fichier tronqué -->
      ...
    </table>
  </body>
</html>
```

8.7.4. Itération sur des groupes

L'élément `xsl:for-each-group` est un ajout de XSLT 2.0. Il permet de grouper des nœuds du document source suivant différents critères puis de parcourir les groupes formés.

La feuille de style suivante donne un premier exemple simple d'utilisation de l'élément `xsl:for-each-group`. Elle présente la bibliographie en XHTML en regroupant les livres par années. Le regroupement est réalisé par l'élément `xsl:for-each-group` avec l'attribut `group-by` égal à l'expression XPath `year`. L'attribut `select` détermine que les éléments à regrouper sont les élément `book`. Le traitement de chacun des groupes est réalisé par le contenu de l'élément `xsl:for-each-group`. La clé du groupe est d'abord récupérée par la fonction `current-grouping-key()` pour construire le titre contenu dans l'élément XHTML `h2`. Les éléments `book` de chaque groupe sont ensuite traités, l'un après l'autre, grâce à un élément `xsl:for-each`.

L'attribut `select` de cet élément utilise la fonction `current-group()` qui retourne la liste des objets du groupe.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/xhtml">
<xsl:output method="xhtml" encoding="iso-8859-1" indent="yes"/>
<xsl:template match="/">
<html>
  <head>
    <title>Bibliographie par année</title>
  </head>
  <body>
  <h1>Bibliographie par année</h1>
  <!-- Regroupement des livres par années -->
  <xsl:for-each-group select="bibliography/book" group-by="year">
    <!-- Tri des groupes par années -->
    <xsl:sort select="current-grouping-key()"/>
    <!-- Titre avec l'année -->
    <h2>
      <xsl:text>Année </xsl:text>
      <xsl:value-of select="current-grouping-key()"/>
    </h2>
    <!-- Liste des livres de l'année -->
    <ul>
      <!-- Traitement de chacun des éléments du groupe -->
      <xsl:for-each select="current-group()">
        <!-- Tri des éléments du groupe par auteur puis publisher -->
        <xsl:sort select="author"/>
        <xsl:sort select="publisher"/>
        <xsl:apply-templates/>
      </xsl:for-each>
    </ul>
  </xsl:for-each-group>
  </body>
</html>
</xsl:template>
<!-- Règle pour les éléments title -->
<xsl:template match="title">
  <i><xsl:apply-templates/></i>
  <xsl:call-template name="separator"/>
</xsl:template>
<!-- Règle pour les autres éléments -->
<xsl:template match="*">
  <xsl:apply-templates/>
  <xsl:call-template name="separator"/>
</xsl:template>
<!-- Virgule après les éléments -->
<xsl:template name="separator">
  <xsl:if test="position() != last()">
    <xsl:text>, </xsl:text>
  </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

En appliquant la feuille de style précédente au document `bibliography.xml`, on obtient le document suivant.

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

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>Bibliographie par année</title>
  </head>
  <body>
    <h1>Bibliographie par année</h1>
    <h2>Année 2000</h2>
    <ul>
      <li><i>XML by Example</i>, Benoît Marchal, 2000,
        Macmillan Computer Publishing, 0-7897-2242-9</li>
      ...
    </ul>

    <h2>Année 2001</h2>
    ...
  </body>
</html>
```

La feuille de style suivante donne un exemple classique d'utilisation de l'élément `xsl:for-each-group`. Celle-ci effectue une transformation d'un document XHTML en un document DocBook. Pour simplifier, on se contente de sous-ensembles très restreints de ces deux dialectes XML. On considère uniquement les éléments `html`, `body`, `h1`, `h2` et `p` de XHTML et des éléments `book`, `chapter`, `sect1`, `title` et `para` de DocBook. Ces deux langages organisent un document de manières différentes. Dans un document XHTML, les chapitres et les sections sont uniquement délimités par les titres `h1` et `h2`. Au contraire, dans un document DocBook, les éléments `chapter` et `sect1` encapsulent les chapitres et les sections. Ces différences rendent plus difficile la transformation de XHTML vers DocBook. Pour trouver le contenu d'un chapitre, il est nécessaire de regrouper tous les éléments placés entre deux éléments `h1` consécutifs. L'élément `xsl:for-each-group` permet justement de réaliser facilement cette opération.

L'espace de noms par défaut de la feuille de style est celui de DocBook. Les noms des éléments XHTML doivent ainsi être qualifiés par le préfixe `html` associé à l'espace de noms de XHTML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:html="http://www.w3.org/1999/xhtml"
                xmlns="http://docbook.org/ns/docbook"
                exclude-result-prefixes="xsl html">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="/">
    <book>
      <xsl:apply-templates select="html:html/html:body"/>
    </book>
  </xsl:template>
  <xsl:template match="html:body">
    <!-- Regroupement des éléments avec l'élément h1 qui précède -->
    <xsl:for-each-group select="*" group-starting-with="html:h1">
      <chapter>
        <!-- Titre avec le contenu de l'élément h1 -->
        <title><xsl:value-of select="current-group()[1]"/></title>
        <!-- Traitement du groupe -->
        <!-- Regroupement des éléments avec l'élément h2 qui précède -->
        <xsl:for-each-group select="current-group()"
                            group-starting-with="html:h2">
          <xsl:choose>
            <xsl:when test="local-name(current-group()[1]) = 'h2'">
              <sect1>
```

```
              <!-- Titre avec le contenu de l'élément h2 -->
              <title><xsl:value-of select="current-group()[1]"/></title>
              <xsl:apply-templates select="current-group()"/>
            </sect1>
          </xsl:when>
          <xsl:otherwise>
            <xsl:apply-templates select="current-group()"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each-group>
    </chapter>
  </xsl:for-each-group>
 </xsl:template>
 <!-- Supression des éléments h1 et h2 -->
 <xsl:template match="html:h1|html:h2"/>
 <!-- Transformation des éléments p en éléments para -->
 <xsl:template match="html:p">
   <para><xsl:apply-templates/></para>
 </xsl:template>
</xsl:stylesheet>
```

Le document suivant est le document XHTML sur lequel est appliqué la tranformation. Celui-ci représente le squelette typique d'un document XHTML avec des titres de niveaux 1 et 2 et des paragraphes.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr">
  <head>
    <title>Fichier HTML exemple</title>
  </head>
  <body>
  <h1>Titre I</h1>
      <p>Paragraphe I.0.1</p>
    <h2>Titre I.1</h2>
      <p>Paragraphe I.1.1</p>
      <p>Paragraphe I.1.2</p>
    <h2>Titre I.2</h2>
      <p>Paragraphe I.2.1</p>
      <p>Paragraphe I.2.2</p>
  <h1>titre II</h1>
      <p>Paragraphe II.0.1</p>
    <h2>Titre II.1</h2>
      <p>Paragraphe II.1.1</p>
      <p>Paragraphe II.1.2</p>
    <h2>Titre II.2</h2>
      <p>Paragraphe II.2.1</p>
      <p>Paragraphe II.2.2</p>
  </body>
</html>
```

Le document suivant est le document DocBook obtenu par transformation par la feuille de style prédédente du document XHTML précédent. Les deux éléments `h1` du document XHTML donnent deux éléments `chapter` dans le document DocBook.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<book xmlns="http://docbook.org/ns/docbook">
  <chapter>
    <title>Titre I</title>
    <para>Para I.0.1</para>
    <sect1>
```

```
      <title>Titre I.1</title>
      <para>Para I.1.1</para>
      <para>Para I.1.2</para>
    </sect1>
    <sect1>
      <title>Titre I.2</title>
      <para>Para I.2.1</para>
      <para>Para I.2.2</para>
    </sect1>
  </chapter>
  <chapter>
    <title>titre II</title>
    <para>Para II.0.1</para>
    <sect1>
      <title>Titre II.1</title>
      <para>Para II.1.1</para>
      <para>Para II.1.2</para>
    </sect1>
    <sect1>
      <title>Titre II.2</title>
      <para>Para II.2.1</para>
      <para>Para II.2.2</para>
   </sect1>
  </chapter>
</book>
```

8.8. 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 `xsl:apply-templates`, `xsl:call-template`, `xsl:for-each` ou `xsl: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>
```

Le tri réalisé par `xsl:sort` est basé sur les valeurs retournées par l'expression XPath contenue dans l'attribut `select`. Cette expression est souvent le nom d'un enfant ou d'un attribut mais elle peut aussi être plus complexe. La feuille de style suivante réordonne les enfants des éléments `book`. L'expression contenue dans l'attribut `select` de `xsl:sort` retourne un numéro d'ordre en fonction du nom de l'élément. Ce numéro est calculé avec la fonction `index-of()` et une liste de noms dans l'ordre souhaité. Cette solution donne une expression concise. Elle a aussi l'avantage que l'ordre est donné par une liste qui peut être fixe ou calculée. Cette liste peut, par exemple, être la liste des noms des enfants du premier élément `book`.

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

  <!-- Liste des noms des enfants du premier élément book -->
  <xsl:variable name="orderlist" as="xsd:string*"
  select="/bibliography/book[1]/*/name()"/>
  ...
  <xsl:template match="book">
    <xsl:copy>
      <!-- Copie des attributs -->
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates>
 <!-- Tri des enfants dans l'ordre donné par la liste fixe -->
 <!-- Les noms absents de la liste sont placés à la fin -->
 <xsl:sort select="(index-of(('title', 'author', 'publisher',
                                    'year', 'isbn'), name()),10)[1]"/>
 <!-- Tri dans l'ordre des enfants du premier élément book -->
 <!-- <xsl:sort select="index-of($orderlist, name())"/> -->
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

8.8.1. Tri de listes

L'élément `xsl:perform-sort` permet d'appliquer un tri à une suite quelconque d'objets, en particulier avant de l'affecter à une variable. Ses enfants 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 version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsl:output method="text" encoding="iso-8859-1"/>
  <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>
```

8.9. 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 stoker 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`. L'élément `xsl:with-param` permet d'instancier un paramètre lors de l'appel à une règle.

8.9.1. Variables

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 *variables*. 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 enfant 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 variable. Les types possibles sont les types XPath [Section 6.1.4]. Dans l'exemple suivant, les deux variables `squares` et `cubes` sont déclarées de type `xsd:integer*`. Chacune d'elles contient donc une liste éventuellement vide d'entiers. La valeur de la variable `square` est donnée par l'élément `xsl:sequence` contenu dans l'élément `xsl:variable`. La valeur de la variable `cubes` est donnée par l'expression XPath de l'attribut `select`.

```
<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*"
       select="for $i in 1 to 5 return $i * $i * $i"/>
```

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

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

Une variable permet aussi de mémoriser un ou plusieurs nœuds. Il est parfois nécessaire de mémoriser le nœud courant dans une variable afin de pouvoir y accéder dans une expression XPath qui modifie contexte dynamique [Section 6.1.5.2]. Le fragment de feuille de style mémorise le nœud courant dans la variable `current`. Elle l'utilise ensuite pour sélectionner les éléments `publisher` dont l'attribut `id` est égal à l'attribut `by` du nœud courant.

```
<xsl:variable name="current" select="."/>
<xsl:xsl:copy-of select="//publisher[@id = $current/@by]"/>
```

XSLT 2.0 a introduit une fonction `current()` qui retourne le nœud courant. Il n'est plus nécessaire de le stocker dans une variable. L'exemple précédent pourrait être réécrit de la façon suivante.

```
<xsl:xsl:copy-of select="//publisher[@id = current()/@by]"/>
```

L'expression XPath `//publisher[@id = $current/@by]` n'est pas très efficace car elle nécessite un parcours complet du document pour retrouver le bon élément `publisher`. Elle peut avantageusement être remplacée par un appel à la fonction `key()`. Il faut au préalable créer avec `xsl:key` [Section 8.12] un index des éléments `publisher` par leur attribut `id`. Cette approche est développée dans l'exemple suivant.

Une variable peut aussi être utilisée, dans un souci d'efficacité pour mémoriser un résultat intermédiaire. Dans l'exemple suivant, le nœud retourné par la fonction `key()` est mémorisé dans la variable `result` puis utilisé à plusieurs reprises.

```
<!-- Indexation des éléments publisher par leur attribut id -->
<xsl:key name="idpublisher" match="publisher" use="@id"/>
...
<!-- Sauvegarde du noeud recherché -->
<xsl:variable name="result" select="key('idpublisher', @by)"/>
```

```
<publisher>
  <!-- Utilisation multiple du noeud -->
  <xsl:copy-of select="$result/@*[name() != 'id']"/>
  <xsl:copy-of select="$result/* | $result/text()"/>
</publisher>
```

L'élément `xsl:variable` permet également de déclarer des variables locales lors de la définition de fonctions d'extension XPath [Section 8.10].

8.9.2. Paramètres

Il existe des paramètres XSLT qui s'apparentent aux paramètres des fonctions des langages classiques comme C ou Java. Ils servent à transmettent des valeurs à la feuille de style et aux règles. Les paramètres sont déclarés par l'élément `xsl:param` qui permet également de donner une valeur par défaut comme en C++. Cet élément peut être enfant de l'élément racine `xsl:stylesheet` ou des éléments `xsl:template`. Dans le premier cas, le paramètre est *global* et dans le second cas, il est *local* à la règle déclarée par `xsl:template`.

Comme avec l'élément `xsl:variable`, l'attribut `name` de l'élément `xsl:param` détermine le nom du paramètre. La valeur par défaut est optionnelle. Elle est donnée soit par une expression XPath dans l'attribut `select` soit directement dans le contenu de l'élément `xsl:param`. Un attribut optionnel `as` peut spécifier le type du paramètre [Section 6.1.4]. Le fragment suivant déclare un paramètre `bg-color` avec une valeur par défaut égale à la chaîne de caractères `white`.

```
<xsl:param name="bg-color" select="'white'"/>
```

Les apostrophes `'''` sont nécessaires autour de la chaîne `white` car la valeur de l'attribut `select` est une expression XPath. La même déclaration peut également prendre la forme suivante sans les apostrophes.

```
<xsl:param name="bg-color">white<xsl:param/>
```

8.9.2.1. Paramètres globaux

Les paramètres globaux sont déclarés par un élément `xsl:param` enfant de l'élément `xsl:stylesheet`. Leur valeur est fixée au moment de l'appel au processeur XSLT. Leur valeur reste constante pendant toute la durée du traitemet 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. Les processeurs qui peuvent être utilisés en ligne de commande ont généralement une option pour donner une valeur à un paramètre. Le processeur `xstlproc` a, par exemple, des options `--param` et `--stringparam` dont les valeurs sont des expressions XPath. La seconde option ajoute implicitement les apostrophes `'''` nécessaires autour des chaînes de caractères.

La feuille de style suivante utilise un paramètre global `bg-color` pour la couleur de fond du document XHTML résultat. Sa valeur est utilisée pour donner une régle CSS [Chapitre 10] dans l'entête du document.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:dbk="http://docbook.org/ns/docbook"
                xmlns="http://www.w3.org/1999/xhtml">
  <xsl:output ... />
  <!-- Paramètre global pour la couleur du fond -->
  <xsl:param name="bg-color" select="'white'"/>

  <xsl:template match="/">
    <xsl:comment>Generated by dbk2html.xsl</xsl:comment>
    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title><xsl:value-of select="dbk:book/dbk:title"/></title>
      <style>
        <xsl:comment>
```

```
          body { background-color: <xsl:value-of select="$bg-color"/>; }
        </xsl:comment>
      </style>
    </head>
    <body><xsl:apply-templates/></body>
    </html>
  </xsl:template>
  ...
</xsl:stylesheet>
```

Pour changer la couleur de fond du document résultat, il faut donner une autre valeur au paramètre `bg-color` comme dans l'exemple suivant.

```
xsltproc --stringparam bg-color blue dbk2html.xsl dbk2html.xml
```

8.9.2.2. Paramètres locaux

La déclaration d'un paramètre d'une règle est réalisée par un élément `xsl:param` enfant de `xsl:template`. Les déclarations de paramètres doivent être les premiers enfants. Le passage d'une valeur en paramètre est réalisé par un élément `xsl:with-param` fils de `xsl:apply-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 [Section 6.1.4] 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 version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <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>
```

Dans l'exemple précédent, la valeur passée en paramètre est une chaîne de caractères mais elle peut aussi être un nœud ou une liste de nœuds. Dans l'exemple suivant, la règle `get-id` retourne un attribut `id`. La valeur de celui-ci est produite à partir des attributs `id` et `xml:id` du nœud passé en paramètre. Par défaut, ce nœud est le nœud courant.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="/">
    <xsl:apply-templates select="*"/>
  </xsl:template>
  <xsl:template match="*">
    <xsl:copy>
      <!-- Appel de la règle get-id avec la valeur par défaut du paramètre -->
      <xsl:call-template name="get-id"/>
      <xsl:apply-templates select="*"/>
```

```
    </xsl:copy>
  </xsl:template>
  <!-- Retourne un attribut id -->
  <xsl:template name="get-id">
    <!-- Paramètre avec le noeud courant comme valeur par défaut -->
    <xsl:param name="node" as="node()" select="."/>
    <xsl:attribute name="id"
                   select="($node/@id, $node/@xml:id, generate-id($node))[1]"/>
  </xsl:template>
</xsl:stylesheet>
```

L'élément `xsl:param` est également utilisé pour déclarer les paramètres des fonctions d'extension XPath [Section 8.10]. La règle `get-id` de la feuille de style précédente aurait aussi pu être remplacée par une fonction d'extension. Dans la feuille de style suivante, la fonction XPath `get-id()` calcule la valeur de l'attribut `id` à partir des attributs `id` et `xml:id` du nœud passé en paramètre.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:fun="http://www.liafa.jussieur.fr/~carton">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <!-- Définition de la fonction d'extension get-id -->
  <xsl:function name="fun:get-id" as="xsd:string">
    <xsl:param name="node" as="node()"/>
    <xsl:sequence select="($node/@id, $node/@xml:id, generate-id($node))[1]"/>
  </xsl:function>
  <xsl:template match="/">
    <xsl:apply-templates select="*"/>
  </xsl:template>
  <xsl:template match="*">
    <xsl:copy>
      <!-- Ajout de l'attribut id -->
      <xsl:attribute name="id" select="fun:get-id(.)"/>
      <xsl:apply-templates select="*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

8.9.3. Récursivité

XPath 2.0 a ajouté des fonctions qui facilitent le traitement des chaînes de caractères. Avec XSLT 1.0 et XPath 1.0, il est souvent nécessaire d'avoir recours à des règles récursives pour certains traitements. Les deux règles `quote.string` et `quote.string.char` données ci-dessous insèrent un caractère `'\'` avant chaque caractère `'\'` ou `'"'` d'une chaîne.

```
<!-- Insertion de '\' avant chaque caractère '"' et '\' du contenu -->
<xsl:template match="text()">
  <xsl:call-template name="quote.string">
    <xsl:with-param name="string" select="."/>
  </xsl:call-template>
</xsl:template>
<!-- Insertion de '\' avant chaque caractère '"' et '\' du paramètre string -->
<xsl:template name="quote.string">
  <!-- Chaîne reçue en paramètre -->
  <xsl:param name="string"/>
  <xsl:choose>
    <xsl:when test="contains($string, '&quot;')">
```

```
      <xsl:call-template name="quote.string.char">
        <xsl:with-param name="string" select="$string"/>
        <xsl:with-param name="char"   select="'&quot;'"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:when test="contains($string, '\')">
      <xsl:call-template name="quote.string.char">
        <xsl:with-param name="string" select="$string"/>
        <xsl:with-param name="char"   select="'\'"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$string"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<!-- Fonction auxiliare pour quote.string -->
<xsl:template name="quote.string.char">
  <!-- Chaîne reçue en paramètre -->
  <xsl:param name="string"/>
  <!-- Caractère reçu en paramètre -->
  <xsl:param name="char"/>
  <xsl:variable name="prefix">
    <xsl:call-template name="quote.string">
      <xsl:with-param name="string" select="substring-before($string, $char)"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="suffix">
    <xsl:call-template name="quote.string">
      <xsl:with-param name="string" select="substring-after($string, $char)"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:value-of select="concat($prefix, '\', $char, $suffix)"/>
</xsl:template>
```

Ce traitement pourrait être réalisé de façon plus concise avec la fonction XPath 2.0 `replace()`.

```
<!-- Insertion de '\' avant chaque caractère '"' et '\' du contenu -->
<xsl:template match="text()">
  <xsl:value-of select="replace(., '([&quot;\\])', '\\$1')">
</xsl:template>
```

8.9.4. 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 utilisées 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 `lunnet`. 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 au nœuds textuels déclare les paramètres `tunnel` et `lunnet`. La valeur de `tunnel` est effectivement la valeur donnée au départ à ce paramètre. Au contraire, la valeur du paramètre `lunnet` est celle par défaut car il n'a pas été transmis.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <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:with-param name="lunnet" select="'lunnet'"/>
    </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="lunnet" select="'default'"/>
    <!-- Produit la valeur 'tunnel' fournie au départ -->
    <xsl:value-of select="$tunnel"/>
    <!-- Produit la valeur 'default' déclarée par défaut -->
    <xsl:value-of select="$lunnet"/>
  </xsl:template>
</xsl:stylesheet>
```

8.10. Fonctions d'extension XPath

Avec XSLT 2.0, il est possible de définir des nouvelles fonctions qui peuvent être utilisées dans les expressions XPath. Ces nouvelles fonctions viennent complèter la librairie des fonctions XPath. Elles remplacent avantageusement des constructions lourdes de XSLT 1.0 avec des règles récursives. Le traitement des chaînes de caractères est un champ d'utilisation classique de ces fonctions.

La définition d'une fonction XPath est introduite par l'élément `xsl:function`. Cet élément a des attributs `name` et `as` qui donnent respectivement le nom et le type de retour de la fonction. Le nom de la fonction est un nom qualifié avec un espace de noms [Chapitre 4]. Les paramètres de la fonction sont donnés par des éléments `xsl:param` [Section 8.9.2] enfants de l'élément `xsl:function`. Chacun de ces éléments a aussi des attributs `name` et `as` qui donnent respectivement le nom et le type du paramètre. En revanche, l'élément `xsl:param` ne peut pas donner une valeur par défaut au paramètre avec un attribut `select` ou un contenu. Cette restriction est justifiée par le fait que les fonctions XPath sont toujours appelées avec un nombre de valeurs correspondant à leur arité. Les types possibles pour le type de retour et les paramètres sont les types XPath [Section 6.1.4]. La définition d'une fonction prend donc la forme générique suivante.

```
<xsl:function name="name" as="return-type">
  <!-- Paramètres de la fonction -->
  <xsl:param name="param1" as="type1"/>
  <xsl:param name="param2" as="type2"/>
  ...
  <!-- Corps de la fonction -->
  ...
</xsl:function>
```

L'élément `xsl:function` doit nécessairement être enfant de l'élément racine `xsl:stylesheet` de la feuille de style. Ceci signifie que la portée de la définition d'une fonction est la feuille de style dans son intégralité.

La fonction `url:protocol` de l'exemple suivant extrait la partie protocole [Section 2.3.1] d'une URL. Elle a un paramètre `url` qui reçoit une chaîne de caractères. Elle retourne la chaîne de caractères située avant le caractère `':'` si l'URL commence par un protocole ou la chaîne vide sinon.

```
<xsl:function name="url:protocol" as="xsd:string">
  <xsl:param name="url" as="xsd:string"/>
  <xsl:sequence select="
    if (contains($url, ':')) then substring-before($url, ':') else ''"/>
</xsl:function>
```

Une fois définie, la fonction `url:protocol` peut être utilisée comme n'importe quelle autre fonction XPath. L'exemple suivant crée un nœud texte contenant `http`.

```
<xsl:value-of select="url:protocol('http://www.liafa.jussieu.fr/')"/>
```

La fonction `url:protocol` peut être utilisée pour définir une nouvelle fonction `url:address` qui extrait la partie adresse internet d'une URL. Cette fonction utilise une variable locale `protocol` pour stocker le résultat de la fonction `url:protocol`.

```
<xsl:function name="url:address" as="xsd:string">
  <xsl:param name="url" as="xsd:string"/>
  <xsl:variable name="protocol" as="xsd:string" select="url:protocol($url)"/>
  <xsl:sequence select="
    if (($protocol eq 'file') or ($protocol eq ''))
    then ''
    else substring-before(substring-after($url, '://'), '/')"/>
</xsl:function>
```

L'expression XPath `url:address('http://www.liafa.jussieu.fr/~carton/')` s'évalue, par exemple, en la chaîne de caractères `www.liafa.jussieu.fr`.

Les fonctions définies par `xsl:function` peuvent bien sûr être récursives. La fonction récursive suivante `url:file` extrait le nom du fichier d'un chemin d'accès. C'est la chaîne de caractères située après la dernière occurrence du caractère `'/'`.

```
<xsl:function name="url:file" as="xsd:string">
  <xsl:param name="path" as="xsd:string"/>
  <xsl:sequence select="
    if (contains($path, '/'))
    then url:file(substring-after($path, '/'))
    else $path"/>
</xsl:function>
```

L'expression XPath `url:file('Enseignement/XML/index.html')` s'évalue, par exemple, en la chaîne de caractères `index.html`.

Une fonction XPath est identifiée par son nom qualifié et son arité (nombre de paramètres). Il est ainsi possible d'avoir deux fonctions de même nom pouvu qu'elles soient d'arités différentes. Ceci permet de simuler des paramètres avec des valeurs par défaut en donnant plusieurs définitions d'une fonction avec des nombres de paramètres différents. Dans l'exemple suivant, la fonction `fun:join-path` est définie une première fois avec trois paramètres. La seconde définition avec seulement deux paramètres permet d'omettre le troisième paramètre qui devient ainsi optionnel.

```
<!-- Définition d'une fonction join-path avec 3 paramètres -->
<xsl:function name="fun:join-path" as="xsd:string">
  <xsl:param name="path1" as="xsd:string"/>
  <xsl:param name="path2" as="xsd:string"/>
  <xsl:param name="sep"   as="xsd:string"/>
  <xsl:sequence select="concat($path1, $sep, $path2)"/>
</xsl:function>
<!-- Définition d'une fonction join-path avec 2 paramètres -->
<xsl:function name="fun:join-path" as="xsd:string">
  <xsl:param name="path1" as="xsd:string"/>
  <xsl:param name="path2" as="xsd:string"/>
  <xsl:sequence select="concat($path1, '/' , $path2)"/>
</xsl:function>
...
  <!-- Appel de la fonction à 3 paramètres -->
  <xsl:value-of select="fun:join-path('Directory', 'index.html', '/')"/>
  <!-- Appel de la fonction à 2 paramètres -->
```

```
  <xsl:value-of select="fun:join-path('Directory', 'index.html')"/>
```

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 appliquer des règles avec le même mode que celui de la règle en cours. La valeur par défaut est encore `#default`.

```
<!-- 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 version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <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.

```
<!-- Règle pour la racine -->
<xsl:template match="/">
  <html>
    <head>
      <title><xsl:value-of select="book/title"/></title>
    </head>
    <body>
      <!-- Fabrication de la table des matières -->
      <xsl:apply-templates mode="toc"/>
      <!-- Fabrication du corps du document -->
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
...
<!-- 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>
```

8.12. Indexation

La fonction `id()` [Section 6.1.1.2] permet de retrouver des nœuds dans un document à partir de leur attribut de type `ID` [Section 3.7.2]. Elle prend en paramètre une liste de noms séparés par des espaces. Elle retourne une liste contenant les éléments dont la valeur de l'attribut de type `ID` est un des noms passés en paramètre. Cette fonction est typiquement utilisée pour traiter des attributs de type `IDREF` ou `IDREFS`. Ces attributs servent à référencer des éléments du document en donnant une (pour `IDREF`) ou plusieurs (pour `IDREFS`) valeurs d'attibuts de type `ID`. La fonction `id()` permet justement de retrouver ces éléments référencés.

L'exemple suivant illustre l'utilisation classique de la fonction `id()`. On suppose avoir un document contenant une bibliographie où les éditeurs ont été placés dans une section séparée du document. Chaque élément `book` contient un élément `published` ayant un attribut `by` de type `IDREF` pour identifier l'élément `publisher` correspondant dans la liste des édideurs. La feuille de style suivante permet de revenir à une bibliographie où chaque élément `book` contient directement l'élément `publisher` correspondant. L'élément retourné par la fonction `id()` est stocké dans une variable [Section 8.9] pour l'utiliser à plusieurs reprises.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <!-- Règle pour la racine -->
```

```
  <xsl:template match="/">
    <bibliography>
      <xsl:apply-templates select="bibliography/books/book"/>
    </bibliography>
  </xsl:template>
  <!-- Règles pour les livres -->
  <xsl:template match="book">
    <book>
      <!-- Copie des attributs -->
      <xsl:copy-of select="@*"/>
      <!-- Copie des éléments autres que published -->
      <xsl:copy-of select="*[name() != 'published']"/>
      <!-- Remplacement des éléments published -->
      <xsl:apply-templates select="published"/>
    </book>
  </xsl:template>
  <!-- Règle pour remplacer published par le publisher référencé -->
  <xsl:template match="published">
    <!-- Elément publisher référencé par l'élément published -->
    <xsl:variable name="pubnode" select="id(@by)"/>
    <publisher>
      <!-- Recopie des attributs autres que id -->
      <!-- L'attribut id ne doit pas être recopié car sinon,
    on peut obtenir plusieurs éléments publisher avec
    la même valeur de cet attribut -->
      <xsl:copy-of select="$pubnode/@*[name() != 'id']"/>
      <xsl:copy-of select="$pubnode/* | $pubnode/text()"/>
    </publisher>
  </xsl:template>
</xsl:stylesheet>
```

Lorsque la fonction `id()` retourne plusieurs nœuds, il est possible de les traiter un par un en utilisant un élément `xsl:for-each` comme dans l'exemple suivant.

```
  <xsl:for-each select="id(@arearefs)">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
```

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'élément `xsl:key` doit être un enfant de l'élément racine `xsl:stylesheet`. L'attribut `name` de `xsl:key` fixe le nom de l'index pour son utilisation. L'attribut `match` contient un motif [Section 6.8] qui détermine les nœuds indexés. L'attribut `use` contient une expression XPath qui spécifie la clé d'indexation, c'est-à-dire la valeur qui identifie les nœuds et permet de les retrouver. Pour chaque nœud sélectionné par le motif, cette expression est évaluée en prenant le nœud sélectionné comme nœud courant. Pour indexer des éléments en fonction de la valeur de leur attribut `type`, on utilise l'expression `@type` comme valeur de l'attribut `use`. La valeur de l'attribut `use` peut être une expression plus complexe qui sélectionne des enfants et des attributs. Dans l'exemple suivant, tous les éléments `chapter` du document sont indéxés en fonction de leur attribut `id`.

```
<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. Dans l'exemple suivant, on utilise l'index créé à l'exemple précédent. La valeur de l'attribut `@idref` d'un élément contenu dans la variable `$node` sert pour retrouver l'élément dont c'est la valeur de l'attribut `id`. Le nom `idchapter` de l'index est un nom XML et il doit être placé entre apostrophes ou guillemets.

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

L'utilisation des index peut être contournée par des expressions XPath appropriée. L'expression ci-dessus avec la fonction `key` est équivalente à l'expression XPath ci-dessous qui utilise l'opérateur `'//'` [Section 6.7].

```
<xsl:value-of select="//chapter[@id = $node/@idref]/title"/>
```

L'inconvénient de cette dernière expression est d'imposer un parcours complet du document pour chacune de ses évaluations. Si l'expression est évaluée à de nombreuses reprises, ceci peut conduire à des problèmes d'efficacité.

Lors de la présentation des attributs de type `ID` et `IDREF` [Section 3.7.2], il a été observé que la bibliographie `bibliography.xml` doit être organisée de façon différente si les éléments `publisher` contiennent des informations autres que le nom de l'éditeur. Afin d'éviter de dupliquer ces informations dans chaque livre, il est préférable de regrouper les éléments `publisher` dans une section à part. Chaque élément `publisher` contenu dans un élément `book` est alors remplacé par un élément `published` avec un attribut `by` de type `IDREF` pour référencer l'élément `publisher` déplacé. La feuille de style suivante remplace les éléments `publisher` par des éléments `published` et les regroupe dans une liste en fin de document. Elle suppose que chaque élément `publisher` contient, au moins, un enfant `name` avec le nom de l'éditeur. Elle supprime également les doublons en évitant que deux éléments `publisher` avec le même nom, c'est-à-dire le même contenu de l'élément `name`, apparaissent dans la liste. Cette suppression des doublons est réalisée par une indexation des éléments `publisher` sur le contenu de `name`. Pour chaque élément `publisher`, l'indexation permet de retrouver efficacement tous les éléments `publisher` avec un contenu identique de `name`. Seul le premier de ces éléments `publisher` est ajouté à la liste des éditeurs dans le document résultat.

Pour savoir si un élément `publisher` est le premier de ces éléments avec le même nom, celui-ci est comparé avec le premier de la liste retournée par la fonction `key()` avec l'opérateur `is` [Section 6.5.3]. La copie de cet élément `publisher` est réalisée par un élément `xsl:copy` [Section 8.6.9] ainsi que par élement `xsl:copy-of` [Section 8.6.10] pour ses attributs et ses enfants.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
  <!-- Indexation des éléments publisher par leur nom -->
  <xsl:key name="pubname" match="publisher" use="name"/>
  <xsl:template match="/">
    <bibliography>
      <books>
        <!-- Livres -->
        <xsl:apply-templates select="bibliography/book"/>
      </books>
      <publishers>
        <!-- Éditeurs -->
        <xsl:apply-templates select="bibliography/book/publisher"/>
      </publishers>
    </bibliography>
  </xsl:template>
  <!-- Règles pour les livres -->
  <xsl:template match="book">
    <book>
      <!-- Copie des attributs et des enfants autres que publisher -->
      <xsl:copy-of select="@* | *[name() != 'publisher']"/>
      <!-- Transformation de l'élément publisher en élément published -->
      <xsl:if test="publisher">
        <published id="{generate-id(key('pubname', publisher/name)[1])}"/>
      </xsl:if>
    </book>
  </xsl:template>
  <!-- Copie d'un élément publisher en ajoutant un attribut id -->
```

```
  <xsl:template match="publisher">
   <!-- Test si l'élément publisher est le premier avec le même nom -->
    <xsl:if test=". is key('pubname', name)[1]">
      <xsl:copy>
        <!-- Ajout de l'attribut id -->
        <xsl:attribute name="id">
          <xsl:value-of select="generate-id()"/>
        </xsl:attribute>
        <!-- Copie des attributs et des enfants -->
        <xsl:copy-of select="@* | *"/>
      </xsl:copy>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

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"?>
<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 version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <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>
  </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 les deux itérations imbriquées et les 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.

8.14. Analyse de chaînes

Il arrive que le document source ne soit pas suffisament structuré et que la feuille de style ait besoin d'extraire des morceaux de texte. L'élément `xsl:analyze-string` permet d'analyser une chaîne de caractères et de la découper en fragments. Ces fragments peuvent ensuite être repris et utilisés. L'analyse est réalisée avec une expression rationnelle [Section 5.15].

La chaîne à analyser et l'expression rationnelle sont respectivement données par les attributs `select` et `regex` de l'élément `xsl:analyze-string`. Les deux enfants `xsl:matching-substring` et `xsl:non-matching-substring` de `xsl:analyze-string` donnent le résultat suivant que la chaîne est compatible ou non avec l'expression.

La fonction XPath `regex-group()` permet de récupérer un fragment de la chaîne correspondant à un bloc délimité par des parenthèses dans l'expression. L'entier fourni en paramètre donne le numéro du bloc. Les blocs sont numérotés à partir de 1.

Dans l'exemple suivant, le contenu de l'élément `name` est découpé à la virgule pour extraire le prénom et le nom de famille d'un nom complet écrit suivant la convention anglaise. Les deux parties du nom sont ensuite utilisées pour construire les enfants `firstname` et `lastname` de `name`. Lorsque le contenu ne contient pas de virgule, l'élément `name` est laissé inchangé.

```
<xsl:template match="name">
  <xsl:analyze-string select="." regex="([^,]*),\s*(.*)">
    <xsl:matching-substring>
      <name>
        <firstname><xsl:value-of select="regex-group(2)"/></firstname>
        <lastname><xsl:value-of  select="regex-group(1)"/></lastname>
      </name>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
      <name><xsl:value-of select="."/></name>
    </xsl:non-matching-substring>
  </xsl:analyze-string>
</xsl:template>
```

La feuille de style précédente peut être appliquée au document suivant.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<names>
  <name>Lagaffe, Gaston</name>
  <name>Talon, Achille</name>
  <name>Astérix</name>
```

```
</names>
```

On obtient le document suivant où les noms sont mieux structurés.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<names>
   <name><firstname>Gaston</firstname><lastname>Lagaffe</lastname></name>
   <name><firstname>Achille</firstname><lastname>Talon</lastname></name>
   <name>Astérix</name>
</names>
```

8.15. Import de feuilles de style

Les éléments `xsl:include` et `xsl:import` permettent d'inclure les règles d'une feuille de style au sein d'une autre feuille de style. La seule différence entre les deux éléments est la gestion des priorités entre les règles des deux feuilles de style. Ces priorités [Section 8.5.3] influencent les choix de règles à appliquer sur les nœuds. Ces deux éléments ont un attribut `href` contenant l'URL [Section 2.3] de la feuille de style à inclure. Cette URL est très souvent le nom relatif ou absolu d'un fichier local. Les deux éléments `xsl:include` et `xsl:import` doivent être enfants de l'élément racine `xsl:stylesheet` et ils doivent être placés avant toute définition de règle par un élément `xsl:template`. L'exemple suivant est la feuille de style principale pour la transformation de cet ouvrage au format DocBook en HTML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:dbk="http://docbook.org/ns/docbook">
  <!-- Import de la feuille de style docbook.xsl -->
  <xsl:import
    href="/usr/share/sgml/docbook/xsl-stylesheets/html/profile-docbook.xsl"/>
  <!-- Ajout des paramètres communs de configuration -->
  <xsl:include href="common.xsl"/>
  <!-- Feuille de style CSS -->
  <xsl:param name="html.stylesheet" select="'style.css'"/>
  <!-- Pour les paragraphes justifiés à droite -->
  <xsl:template match="dbk:para[@role = 'right']">
    <p align="right"><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

Lorsque un processeur XSLT choisit une règle à appliquer à un nœud du document source, il prend en compte deux paramètres qui sont la *priorité d'import* entre les feuilles de style et les priorités entre les règles. Lorsque la feuille de style est importée avec l'élément `xsl:include`, les deux feuilles de style ont même priorité d'import comme si les règles des deux feuilles se trouvaient dans une même feuille de style. Au contraire, lorsque la feuille de style est importée avec l'élément `xsl:import`, la feuille de style importée a une priorité d'import inférieure. Les règles de la feuille de style qui réalise l'import, c'est-à-dire celle qui contient l'élément `xsl:import`, sont appliquées en priorité sur les règles de la feuille importée. Ce mécanisme permet d'adapter la feuille de style importée en ajoutant des règles dans la feuille de style qui importe. Lorsqu'une règle ajoutée remplace une règles de la feuiile importée, elle peut encore utiliser la règle remplacée grâce à l'élément `xsl:apply-imports` [Section 8.5.4].

Une feuille de style peut importer plusieurs feuilles de style avec plusieurs éléments `xsl:import`. Dans ce cas, les feuilles de style importées en premier ont une priorité d'import inférieure. L'ordre des priorités d'import est l'ordre des éléments `xsl:import` dans la feuille de style qui importe. Il est également possible qu'une feuille de style importée par un élément `xsl:import` réalise elle-même des imports d'autres feuilles de style avec des éléments `xsl:import`. Lorsqu'il y a plusieurs niveaux d'import, l'ordre d'import entre les différentes feuilles de style est d'abord dicté par l'ordre des imports de premier niveau puis l'ordre des imports de second niveau et ainsi de suite. Ce principe est illustré par l'exemple suivant. Supposons qu'une feuille de style A importe, dans cet ordre, les feuilles de style B et C, que la feuille B importe, à son tour, les feuilles D et E et que la feuille C importe,

finalement, les feuilles F et G (cf. Figure). L'ordre des feuilles par priorité d'import croissante est D, E, B, F, G, C, A. Cet ordre correspond à un parcours suffixe de l'arbre des imports.

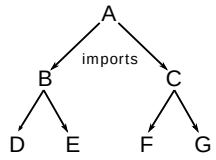


Figure 8.4. Priorités d'import

Chapitre 9. XSL-FO

XSL-FO est un *dialecte* de XML permettant de décrire le rendu de documents. Un document XSF-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 [Chapitre 10] 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 XSLT [Chapitre 8].

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 `helloworld.pdf` ou `helloworld.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.

Chapitre 10. 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 `':'`. Le caractère `';'` sépare les couples propriété/valeur de la règle. Il n'est donc pas indispensable après le dernier couple de la règle.

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 `'*/'`.

L'exemple ci-dessous est la feuille de style utilisée pour la présentation de cet ouvrage au format HTML.

```
/* 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;
    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 possibles 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
```

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;
}
```

L'élément unique dont l'attribut `id` a la valeur *name* peut être sélectionné par le sélecteur *#name* où la valeur de l'attribut est précédée d'un dièse `'#'`. Dans l'exemple suivant, le contenu de l'élément `h1` dont l'attribut `id` vaut `title` sera de couleur rouge.

```
h1#title { color: red }
```

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 }
```

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 enfant 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.

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`*.

`[`*`att`*`~=`*`value`*`]`
: L'élément est sélectionné s'il a un attribut *`att`* dont la valeur est une suite de chaînes séparées par des espaces dont l'une est exactement la chaîne *`value`*. Le sélecteur *`.classname`* est donc une abréviation de `[class~="`*`classname`*`"]`.

`[`*`att`*`|=`*`value`*`]`
: L'élément est sélectionné s'il a un attribut *`att`* dont la valeur est une suite de chaînes de caractères séparées par des tirets `'-'` dont la première chaîne est égale à la chaîne *`value`*.

Dans l'exemple suivant, la règle s'applique aux éléments `p` dont l'attribut `lang` commence par `en-` comme `en-GB` ou `en-US` mais pas `fr`. La propriété `quote` définit quels caractères doivent entourer les citations.

```
p[lang|="en"] {
    quotes: '"' '"' "'" "'";
}
```

Il est possible de mettre plusieurs prédicats portant sur les attributs comme dans l'exemple suivant. Le sélecteur suivant sélectionne les éléments `p` ayant un attribut `lang` de valeur `fr` et un attribut `type` de valeur `center`.

```
p[lang="fr"][type="center"] {
    ...
}
```

Il est possible de composer des sélecteurs avec les trois opérateurs `' '` (espace), `'>'` et `'+'` pour former des nouveaux sélecteurs.

Le sélecteur *`selector1 selector2`* sélectionne tous les éléments sélectionnés par *`selector2`* qui sont en outre descendants dans l'arbre du document (c'est-à-dire inclus) d'un élément sélectionné par *`selector1`*. Dans l'exemple suivant, seuls les éléments `em` contenus directement ou non dans un élément `p` auront leur texte en gras.

```
p em {
    font-weight: bold;
}
```

Cet opérateur peut aussi combiner plusieurs sélecteurs. Dans l'exemple suivant, sont sélectionnés les éléments de classe `sc` contenus dans un élément `em` lui même contenu dans un élément `p`. Il y a bien un espace entre `em` et `.sc`. Le sélecteur `p em.sc` est bien sûr différent.

```
p em .sc {
    font-variant: small-caps;
}
```

Le sélecteur *`selector1 > selector2`* sélectionne tous les éléments sélectionnés par *`selector2`* qui sont en outre enfant (c'est-à-dire directement inclus) d'un élément sélectionné par *`selector1`*.

Le sélecteur *`selector1 + selector2`* sélectionne tous les éléments sélectionnés par *`selector2`* qui sont en outre frère droit (c'est-à-dire qui suivent directement) d'un élément sélectionné par *`selector1`*.

10.2.3. Propriétés

Les propriétés qui peuvent être modifiées par une règle dépendent des éléments sélectionnés. Pour chaque élément de XHTML, il y a une liste des propriétés qui peuvent être modifiées.

10.2.4. Valeurs

Les valeurs possibles dépendent de la propriété. Certaines propriétés acceptent comme `display` uniquement un nombre déterminé de valeurs. D'autres encore prennent une couleur, un entier, un pourcentage, un nombre décimal ou une dimension avec une unité.

Pour les dimensions, il est nécessaire de faire suivre le nombre d'une unité parmi les unités suivantes.

Symbole	**Unité**
`px`	pixel
`pt`	point = 1/72 in
`pc`	pica = 12 pt
`em`	largeur du M dans la police
`ex`	hauteur du x dans la police

Symbole	Unité
`in`	pouce
`mm`	millimètre
`cm`	centimètre

Tableau 10.1. Unités des dimensions en CSS

10.3. Héritage et cascade

Certaines propriétés sont automatiquement héritées comme `color`. Cela signifie qu'un élément hérite de la valeur de cette propriété de son père sauf si une règle donne une autre valeur à cette propriété. D'autres propriétés comme `background-color` ne sont pas héritées. On peut donner la valeur `inherit` à n'importe quelle propriété pour forcer l'héritage de la valeur du père.

Comme la propriété `color` est héritée, la règle suivante impose que le texte de tout élément est noir à l'exception des éléments pour lesquels cette propriété est modifiée.

```
body {
    color: black;
}
```

10.3.1. Provenance de la valeur

La valeur d'une propriété pour un élément peut avoir les trois provenances suivantes par ordre de priorité décroissante.

1. La propriété est modifiée par au moins une règle qui sélectionne l'élément. La valeur de la propriété est alors donnée par la règle de plus grande priorité (cf. ci-dessous).

2. Si aucune règle ne donne la valeur de la propriété et si la propriété est héritée, la valeur est égale à la valeur de la propriété pour le père.

3. Si la propriété n'est pas héritée ou si l'élément est la racine du document, la valeur de la propriété est alors une valeur par défaut appelée *valeur initiale*.

10.3.2. Cascade

Plusieurs règles peuvent s'appliquer à un même élément. Il y a souvent plusieurs feuilles de style pour le même document : une feuille de style par défaut pour l'application, une autre fournie par l'auteur du document et éventuellement une feuille donnée par l'utilisateur. De plus, une même feuille peut contenir plusieurs règles qui sélectionnent le même élément.

La priorité d'une règle est d'abord déterminée par sa provenance. La feuille de style de l'auteur a priorité sur celle de l'utilisateur qui a elle-même priorité sur celle de l'application.

Pour les règles provenant de la même feuille de style, on applique l'algorithme suivant pour déterminer leur priorité. On calcule une *spécificité* de chaque sélecteur qui est un triplet (a,b,c) d'entiers. L'entier a est le nombre d'occurrences de prédicats sur l'attribut `id` de forme `#ident`. L'entier b est le nombre de prédicats sur les autres attributs y compris les attributs `class` de forme `classname`. Le nombre c est finalement le nombre de noms d'éléments apparaissant dans le sélecteur. Les spécificités (a,b,c) et (a',b',c') des sélecteurs des deux règles sont alors comparées par ordre lexicographique. Ceci signifie qu'on compare d'abord a et a', puis b et b' si a est égal à a', puis finalement c et c' si (a,b) est égal à (a',b').

Pour des règles dont les sélecteurs ont même spécificité, c'est l'ordre d'apparition dans la feuille de style qui détermine la priorité. La dernière règle apparue a une priorité supérieure.

10.4. Modèle de boîtes

Chaque élément (au sens usuel ou au sens XML) est mis dans une boîte au moment de la mise en page. Cette boîte englobe le contenu de l'élément. Elle a aussi un espacement intérieur (*padding* en anglais), une bordure (*border*) et une marge (*margin*) (cf. figure ci-dessous).

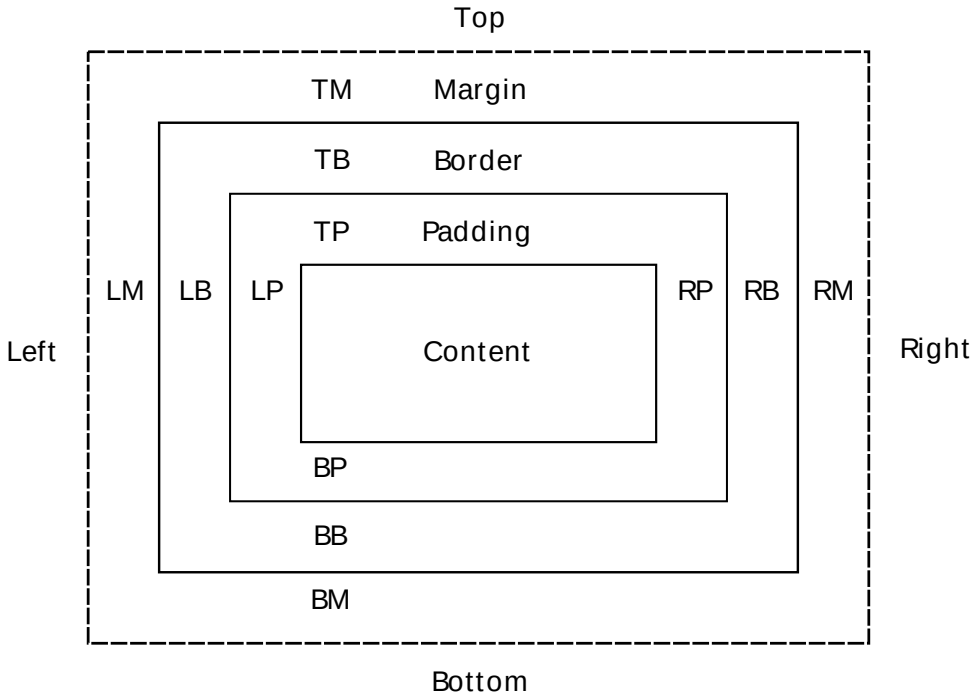


Figure 10.1. Modèle de boîte

Chaque boîte est positionnée à l'intérieur de la boîte englobante de son père. Le positionnement des éléments dans une boîte englobante est d'abord déterminé par le type de la boîte englobante. Les types principaux possibles pour une boîte sont *bloc* (*block*) ou *en ligne* (*inline*), *table*, *entrée de liste* (*list-item*). Les éléments d'une boîte de type bloc sont assemblés verticalement alors que ceux d'une boîte de type *en ligne* sont assemblés horizontalement.

Le type de la boîte est contrôlée par la propriété `display` de l'élément. Les principales valeurs que peut prendre cette propriété sont `block`, `inline`, `list-item` et `none`. Cette dernière valeur permet de ne pas afficher certains éléments. Il est ainsi possible de créer des pages dynamiques en modifiant avec des scripts la valeur de cette propriété.

La propriété `position` détermine comment l'élément est positionné dans la boîte englobante de l'élément père. La valeur `static` signifie que l'élément est positionné automatiquement par l'application dans le flux d'éléments. La valeur `relative` signifie que les valeurs des propriétés `top`, `right`, `bottom` et `left` donnent un déplacement par rapport à la position normale. Pour les valeurs `absolute` et `fixed`, les propriétés `top`, `right`, `bottom` et `left` fixent la position par rapport à la page ou par rapport à la fenêtre de l'application.

10.5. Style et XML

On reprend le document `bibliography.xml` avec la feuille de style CSS suivante attachée.

```
/* Feuille de style pour la bibliographie */
```

```
bibliography {
    display: block;
    border: 1px solid black;
    margin: 30px;
    padding: 20px;
}

/* Mode liste sans marque */
book {
    display: list-item;
    list-style: none;
}

/* Calcul de la marque */
book:before {
    content: "[" counter(c) "]";
    counter-increment: c;
}

/* Titre en italique */
title {
    font-style: italic;
}

/* Virgule après chaque champ */
title:after,  author:after,  year:after, publisher:after {
 content: ", ";
}

/* Fonte sans-serif pour l'url */
url {
    font-family: sans-serif;
}

/* 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.

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

<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 être directement incluses dans un élément `style` de l'entête, c'est-à-dire contenues 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 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.

```
<span style="text-decoration: overline">A</span>
```

10.7. Principales propriétés

10.7.1. Polices de caractères 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-attachement`	`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`

10.7.3. Boîtes et positionnement

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

Chapitre 11. SVG

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

11.1. Un premier exemple

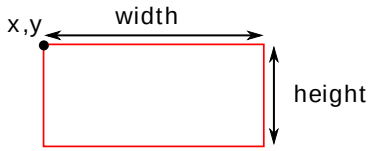
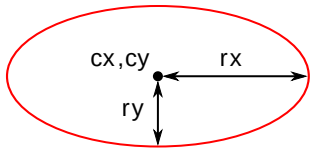
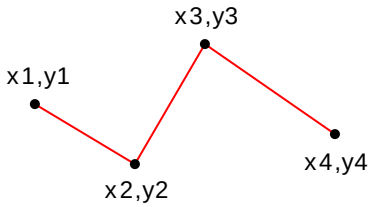
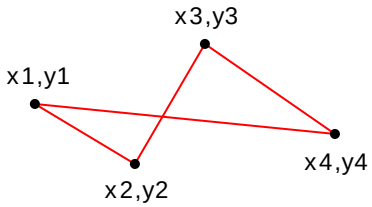
Figure 11.1. Rendu du document ci-dessous

```
<?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 version="1.0" width="200" height="100"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
 <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

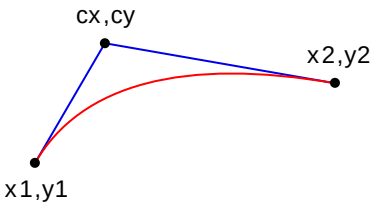
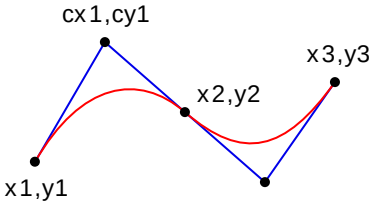
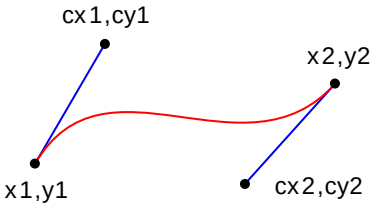
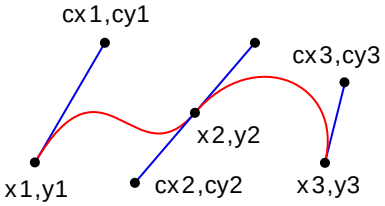
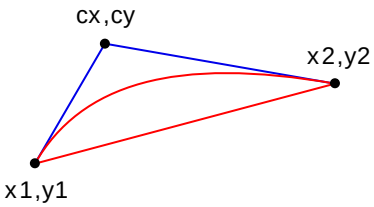
11.2.1. Formes élémentaires

Élément	Rendu
Ligne `<line x1=... y1=... x2=... y2=... />`	

Élément	Rendu
Rectangle `<rect x=... y=... width=... height=... />`	
Ellipse `<ellipse cx=... cy=... rx=... ry=... />`	
Ligne polygonale `<polyline points="x1 y1 x2 y2 x3 y3 ..."/>`	
Ligne polygonale fermée `<polygon points="x1 y1 x2 y2 x3 y3 ..."/>`	

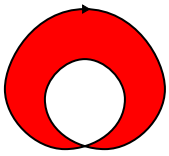
11.2.2. Chemins

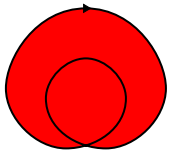
Élément	Rendu
Point de départ `<path d="M x1 y1"/>`	
Ligne horizontale `<path d="M x1 y1 H x2"/>`	
Ligne verticale `<path d="M x1 y1 V y2"/>`	

Élément	Rendu
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 Bezier, on peut se réferer à cette partie [Section 11.5].

11.2.3. Remplissage

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

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

11.3. Transformations

L'élément `g` permet de grouper plusieurs éléments graphiques. Il est ainsi possible d'associer simlutanément 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 successivments. Les transformations possibles sont les suivantes.

Transformation	Action
`translate(dx,dy)`	Translation (déplacement)
`scale(x)` ou `scale(x,y)`	Dilatation
`rotate(a)` ou `scale(a,cx,cy)`	Rotation
`skewX(x)` et `skewY(y)`	Inclinaisons

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>
```

11.4.2. 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>
```

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"/>
  <ellipse class="circle" cx="100" cy="50" rx="40" ry="40"/>
</svg>
```

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>
```

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>
```

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.

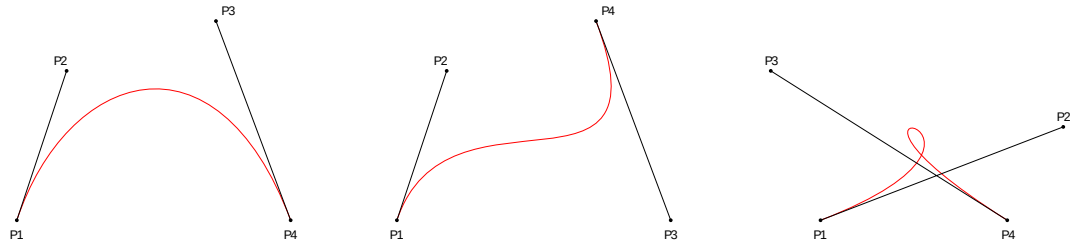


Figure 11.2. Courbes de Bézier

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), la courbe est décrites par les formules suivantes qui donnent la courbe sous forme paramétrée.

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

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

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). 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).

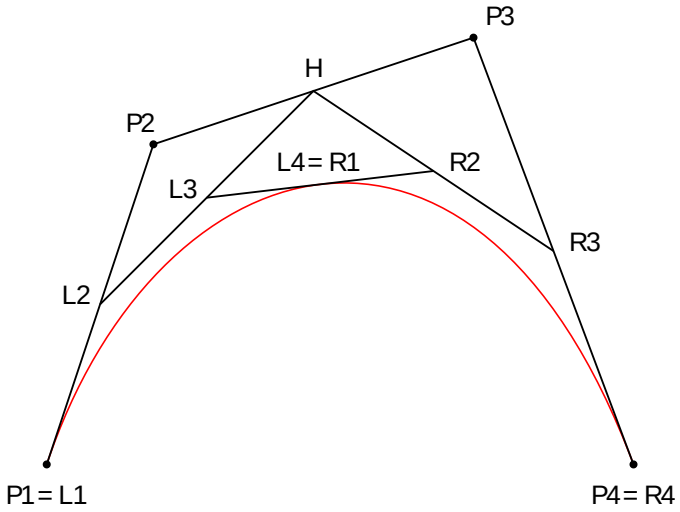


Figure 11.3. Construction de Casteljau

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 $P_1,\ldots,P_{n+3}$. Ils déterminent n B-spline $s_1,\ldots,s_n$ 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), …, (x_{n+3}, y_{n+3}), l'équation de la spline s_i est la suivante.

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

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

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.

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

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

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

11.5.3. Conversions

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

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

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

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

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

Chapitre 12. Programmation XML

Pages [Examples] des sources des examples.

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

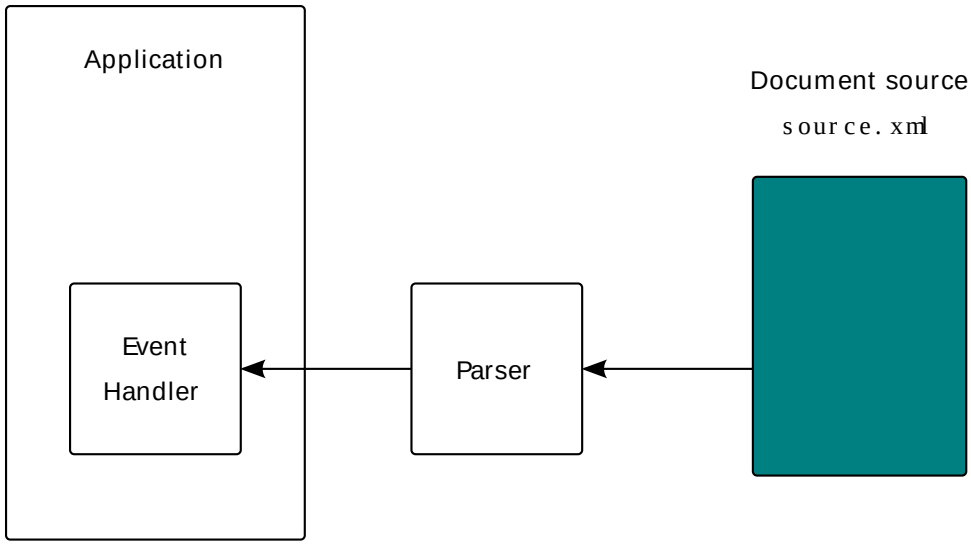


Figure 12.1. Principe de SAX

SAX est une API permettant de lire un fichier XML sous forme de flux. Le principe de fonctionnent 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 la 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 `TrivialSAXHandler.java` minimal

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

/**
 * Handler trivial pour SAX
 * 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  atts) {
       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 `TrivialSAXRead.java` 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();
        // 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

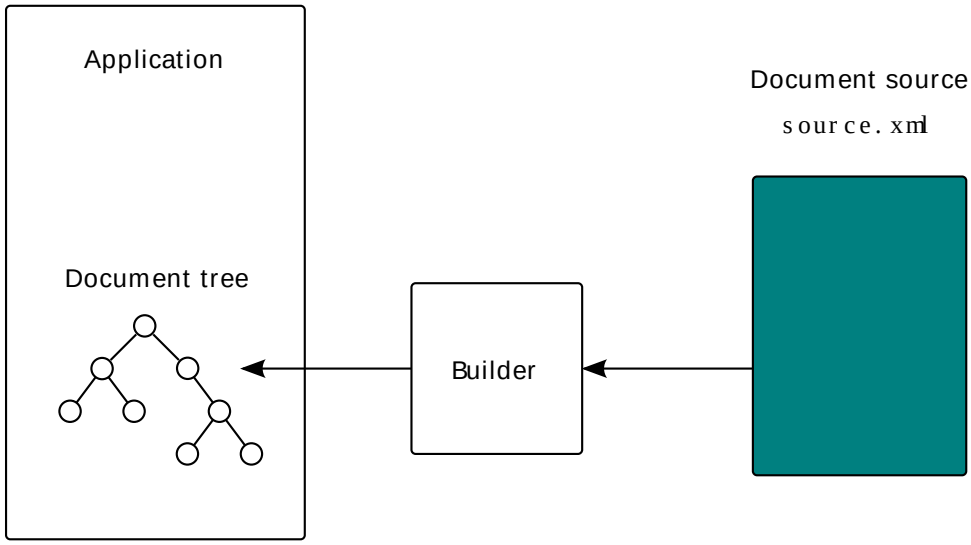


Figure 12.2. Principe de DOM

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 fonctionnent 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>
 <item>
   <!-- Un commentaire inutile -->
```

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

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

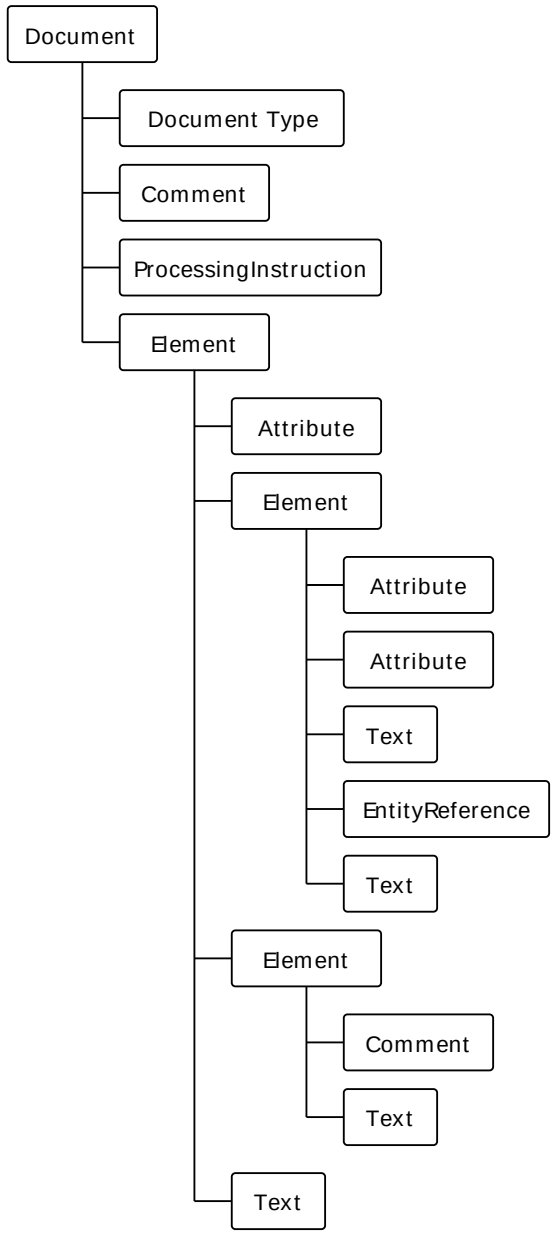


Figure 12.3. Arbre d'un document

12.2.3. Principales classes

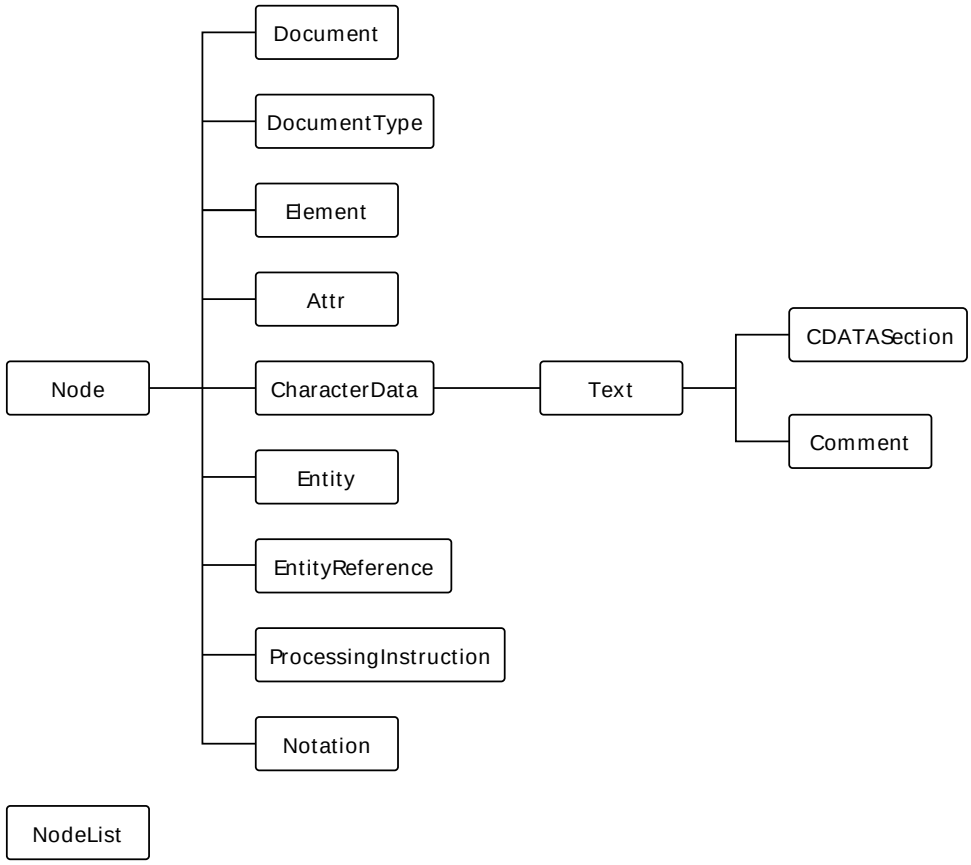


Figure 12.4. Classes du DOM

12.2.4. Lecture d'un fichier XML avec DOM

Lecture `TrivialDOMRead.java` 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
   {
       // 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 [http://developer.apple.com/internet/webcontent/xmlhttpreq.html] 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(url) {
     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] + "</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 -->
 <!-- La fonction getSuggest est appelée à chaque touche relachée -->
 <p><input type="text" onkeyup="JavaScript:getSuggest(event)" /></p>
 <!-- Div recevant le résultat -->
 <p><div id="target"></div></p>
</body>
```

```
</html>
```

Annexe A. Acronymes

AJAX [w]
: Asynchronous JavaScript and XML

API [w]
: Application Programming Interface

BMP [w]
: Basic Multilingual Plane [Section 2.2]

BOM [w]
: Byte Order Mark [Section 2.2.4]

CSS [w]
: Cascading Style Sheet [Chapitre 10]

DOM [w]
: Document Object Model

DTD [w]
: Document Type Definition [Chapitre 3]

FPI
: Formal Public Identifier [Section 3.2.2]

Infoset
: XML Information Set

ISBN [w]
: International Standard Book Number

NCName
: No Colon Name (Nom local) [Chapitre 4]

PCDATA
: Parsed characters DATA

PNG [w]
: Portable Network Graphics

PDF
: Portable Document Format

PSVI
: Post-Schema Validation Infoset

QName
: Qualified Name [Chapitre 4]

RDF [w]
: Resource Description Framework

RSS [w]
: Really Simple Syndication

SAML [w]
: Security Assertion Markup Language

SAX [w]
: Simple API for XML

SGML [w]
: Standard Generalized Markup Language [Chapitre 1]

SMIL [w]
: Synchronized Multimedia Integration Language

SOAP [w]
: Simple Object Access Protocol

SVG [w]
: Scalable Vector Graphics [Chapitre 11]

SVRL [http://www.schematron.com/validators.html]
: Schematron Validation Report Language [Section 7.3]

UBL [w]
: Universal Business Language

UCS [w]
: Universal Character Set [Section 2.2]

URI [w]
: Uniform Resource Identifier [Section 2.3]

URL [w]
: Uniform Resource Locator [Section 2.3]

URN [w]
: Uniform Resource Name [Section 2.3]

UTF [w]
: Universal Transformation Format [Section 2.2.4]

WSDL [w]
: Web Services Description Language

XML [w]
: eXtensible Markup Language

XSD [w]
: XML Schema Definition [Chapitre 5]

XSL [w]
: eXtensible Style Language [Chapitre 8]

XSL-FO [w]
: XSL Formating Object [Chapitre 9]

Bibliographie

XML

A. Michard. *XML langage et applications*. Eyrolles. Paris. 2001. 2-212-09206-7.

B. Marchal. *XML by Example*. Macmillan Couputer Publishing. 2000.

M. Morrison. *XML*. CampusPress. 2005.

F. Role. *Modélisation et manipulation de documents XML*. Lavoisier. 2005.

XPath

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

Schémas XML

V. Lamareille. *XML Schema et XML Infoset*. Cépaduès. 2006.

J.-J. Thomasson. *Schémas XML*. Eyrolles. 2003. 2-212-11195-9.

XSLT et XSL-FO

P. Drix. *XSLT fondamental*. Eyrolles. 2002.

D. Tidwell. *XSLT*. O'Reilly. 2001.

M. Kay. *XSLT 2.0 Programmer's Reference*. Wiley Publishing Inc.. 2004.

M. Kay. *XSLT 2.0 and XPath 2.0*. Wiley Publishing, Inc.. Indianapolis. 2008. 978-0-470-19274-0.

CSS

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

SVG

J. Eisenberg. *SVG Essentials*. O'Reilly. 2002. 2-596-00223-8.

Index

Symboles

A

S

T

U

V

W

X