



Université d'IBN KHALDOUN de TIARET
Faculté des Mathématiques et de l'Informatique
Département d'Informatique

==--==--==--==--==--==--==--==--==--==--==--==--==--==--==--==
XML AVANCÉ & WEB 2.0
==--==--==--==--==--==--==--==--==--==--==--==--==--==--==--==

Partie I : XML Avancé
(Synthèse de cours et exercices corrigés)

Niveau : Master-II
Spécialité : Genie Logiciel

Réalisé par :
Dr. Laouni DJAFRI
==--==--==--==--==--==

Année universitaire : 2022-2023

AVANT-PROPOS

Bienvenue dans ce polycopié de cours : "XML avancé & Web 2.0"

Ce polycopié de cours fournit aux étudiants en Master-II Génie Logiciel des éléments de réflexion pour concevoir et conceptualiser de nouvelles applications stratégiques basées sur la technologie XML.

Dans la première partie (XML avancé), qui est entre nos mains, nous vous présentons les mystères du langage de balisage extensible (XML). En fait, de nombreux experts pensent que XML représente une sorte de "un langage extensible" qui peut représenter des informations sous à peu près n'importe quelle forme imaginable, plus accessible que jamais.

Lisez ce polycopié de cours, et vous vous sentirez plus à l'aise!!!!...

XML représente un mouvement. Il est de nature similaire au mouvement «ouvert» tels que : systèmes d'exploitation-ouverts, interfaces ouvertes, open source, etc. En fait, XML est le prochain pilier de ce mouvement. Il peut être considéré comme le mouvement ouvert étendu à Internet. C'est pour cette raison que, en tant qu'ensemble de normes pures, XML est pris en charge par le World Wide Web Consortium. Cependant, le monde entier parle XML, et les ramifications de son adoption universelle commencent seulement à apparaître. Des modèles économiques entiers en seront affectés ; de tout nouveaux intérêts commerciaux sont poursuivis à cause de cela. L'industrie du logiciel elle-même en sera secouée plus qu'elle ne veut l'admettre. En effet, XML nous emmène sur la terre promise des composants. Prenez des composants basés sur XML, ajoutez Internet et vous obtenez des services Web. Un mélange très explosif en effet !

Ce polycopié de cours se compose de cinq chapitres, et chaque chapitre se termine par des exemples pratiques, des exercices résolus et des exercices ouverts.

Le présent polycopié commence par un chapitre consacré aux principes de XML qui donne une vue d'ensemble et l'historique de XML, ses objectifs et des briques de base. Nous procédons ensuite en deuxième chapitre à la description des formats de base XML, de la bonne formation XML et de la validation XML par rapport aux DTD et aux schémas. Le troisième chapitre XPath nous vous montrons comment utiliser le langage d'expression XPath pour décrire l'emplacement précis des éléments, des attributs ainsi que leurs valeurs dans un document XML. Ensuite, le chapitre quatre XSLT illustre la transformation et la mise en forme de données XML à l'aide de XML. Enfin, le chapitre cinq fournit une vue d'ensemble de haut niveau et une présentation approfondie de XQuery. Ce chapitre inclut également des fonctions plus évoluées pour les étudiants qui souhaitent un apprentissage de haut niveau dans le langage de requête "XQuery".

Nous vous souhaitons une bonne lecture!!!!...

TIARET, LE : 19-11-2022

Table des matières

I	Partie I : XML Avancé	
1	Les principes de XML	9
1.1	Introduction	9
1.2	Genèse et objectif du XML	9
1.2.1	La petite histoire du XML	10
1.2.2	Les objectifs du XML	10
1.3	Les briques de base	12
1.3.1	Structure d'un document XML	12
1.4	Exercices résolus	19
1.5	Solutions	20
1.5.1	Solution de l'exercice 1.1	20
1.5.2	Solution de l'exercice 1.2	20
1.6	Exercices ouverts	21
1.7	Conclusion	22
2	Validation de documents XML	23
2.1	Introduction	23
2.2	Validation de documents XML par DTD	24
2.2.1	Déclaration d'élément	24
2.2.2	Déclaration d'attribut	27
2.2.3	Les entités	31

2.2.4	Les DTD internes et externes	36
2.3	Validation de documents XML par Schéma XML	38
2.3.1	Structure de base d'un Schéma XML	38
2.3.2	Les déclarations d'éléments et d'attributs	39
2.4	Les types simples	40
2.4.1	Les types simples définis dans la recommandation	40
2.4.2	Les types simples définis par le concepteur	45
2.5	Les types complexes	54
2.5.1	Types complexes à contenu simple	55
2.5.2	Types complexes à contenu standard	56
2.5.3	Types complexes à contenu mixte	60
2.6	Exercices résolus	65
2.7	Solutions	66
2.7.1	Solution de l'exercice 2.1	66
2.7.2	Solution de l'exercice 2.2	68
2.8	Exercices ouverts	70
2.9	Conclusion	72
3	XPath	73
3.1	Introduction	73
3.2	Les chemins de localisation	73
3.2.1	Les sélecteurs de nœuds	76
3.3	Les Fonctions	80
3.3.1	Fonctions sur les nombres	80
3.3.2	Fonctions sur les booléens	80
3.3.3	Fonctions sur les nœuds	80
3.3.4	Fonctions sur les chaînes de caractères	81
3.4	Conclusion	83
4	XSLT	85
4.1	Introduction	85
4.2	Structure d'un document XSLT	86
4.2.1	Le prologue	87
4.2.2	L'élément racine	87
4.2.3	L'élément de production	88
4.3	Les éléments indispensables de XSLT	89
4.3.1	<xsl:document>	89
4.3.2	<xsl:element>	89
4.3.3	<xsl:function>	89
4.3.4	<xsl:template>	90
4.3.5	<xsl:apply-templates>	90
4.3.6	<xsl:call-template>	91

4.3.7	<xsl :processing-instruction>	91
4.3.8	<xsl :sequence>	91
4.3.9	<xsl :fallback>	91
4.3.10	<xsl :sort>	92
4.3.11	<xsl :perform-sort>	92
4.3.12	<xsl :value-of>	92
4.3.13	<xsl :param>	93
4.3.14	<xsl :with-param>	93
4.3.15	<xsl :if>	94
4.3.16	<xsl :choose>	94
4.3.17	<xsl :for-each>	94
4.3.18	<xsl :for-each-group>	95
4.3.19	<xsl :copy>	95
4.3.20	<xsl :copy-of>	96
4.3.21	<xsl :variable>	96
4.3.22	<xsl :attribute>	96
4.3.23	<xsl :attribute-set>	97
4.4	XPath, XSLT et HTML	98
4.5	Exercices résolus	99
4.6	Solutions	100
4.6.1	Solution de l'exercice 4.1	100
4.6.2	Solution de l'exercice 4.2	100
4.7	Conclusion	102
5	XQuery	103
5.1	Introduction	103
5.2	Fonctionnalités de XQuery	104
5.3	Formes d'une requête XQuery	105
5.3.1	Expressions de chemins (XPath)	105
5.3.2	Expressions FLOWR	106
5.4	XQuery avancé	113
5.4.1	Fonctions personnalisées sur les éléments et les attributs	113
5.5	Exercices ouverts	119
5.6	Conclusion	121
	Bibliographie	123
	Livres	123
	Articles	124

Partie I : XML Avancé

1 Les principes de XML 9

- 1.1 Introduction
- 1.2 Genèse et objectif du XML
- 1.3 Les briques de base
- 1.4 Exercices résolus
- 1.5 Solutions
- 1.6 Exercices ouverts
- 1.7 Conclusion

2 Validation de documents XML .. 23

- 2.1 Introduction
- 2.2 Validation de documents XML par DTD
- 2.3 Validation de documents XML par Schéma XML
- 2.4 Les types simples
- 2.5 Les types complexes
- 2.6 Exercices résolus
- 2.7 Solutions
- 2.8 Exercices ouverts
- 2.9 Conclusion

3 XPath 73

- 3.1 Introduction
- 3.2 Les chemins de localisation
- 3.3 Les Fonctions
- 3.4 Conclusion

4 XSLT 85

- 4.1 Introduction
- 4.2 Structure d'un document XSLT
- 4.3 Les éléments indispensables de XSLT
- 4.4 XPath, XSLT et HTML
- 4.5 Exercices résolus
- 4.6 Solutions
- 4.7 Conclusion

5 XQuery 103

- 5.1 Introduction
- 5.2 Fonctionnalités de XQuery
- 5.3 Formes d'une requête XQuery
- 5.4 XQuery avancé
- 5.5 Exercices ouverts
- 5.6 Conclusion

Bibliographie 123

- Livres
- Articles



1. Les principes de XML

1.1 Introduction

Le langage de balisage extensible, ou XML, est actuellement le langage le plus prometteur pour le stockage et l'échange d'informations sur le World Wide Web.

Bien que le langage de balisage hypertexte (HTML) soit actuellement le langage le plus couramment utilisé pour créer des pages Web, HTML a une capacité limitée pour stocker des informations. En revanche, comme XML vous permet de créer vos propres éléments, attributs et structure de document, vous pouvez l'utiliser pour décrire pratiquement tout type d'informations, d'une simple recette à une base de données complexe. De plus, un document XML en conjonction avec une feuille de style ou une page HTML conventionnelle peut être facilement affiché dans un navigateur Web. Parce qu'un document XML organise et étiquette si efficacement les informations qu'il contient, le navigateur peut rechercher, extraire, trier, filtrer, organiser et manipuler ces informations de manière très flexible.

XML fournit ainsi une solution idéale pour gérer la quantité et la complexité croissantes des informations qui doivent être livrées sur le Web.

1.2 Genèse et objectif du XML

Aux débuts d'Internet, les ordinateurs et les programmes échangeaient des données en utilisant des fichiers. Malheureusement, ces fichiers avaient bien souvent des règles de formatage qui leur étaient propres. Pour faire face à ce problème, un langage de balisage extensible appelé XML a été développé. Il est conçu pour améliorer la fonctionnalité du Web en fournissant une identification des informations plus flexible et adaptable. Il est appelé extensible car il ne s'agit pas d'un format fixe comme HTML. Au lieu de cela, XML est en fait un « métalangage » qui vous permet de concevoir vos propres langages de balisage personnalisés pour différents types de documents illimités.

1.2.1 La petite histoire du XML

L'histoire de langage de balisage extensible (XML) commence avec le développement du langage de balisage généralisé standardisé (SGML :Standardised Generalised Markup Language) par Charles Goldfarb, avec Ed Mosher et Ray Lorie dans les années 1970, il a été développé par plusieurs centaines de personnes à travers le monde jusqu'à ce qu'il soit finalement adopté comme norme ISO 8879 en 1986. Malgré son nom, SGML n'est pas un langage de balisage à part entière, mais c'est un langage utilisé pour spécifier des langages de balisage. Le but de SGML était de créer des vocabulaires qui pourraient être utilisés pour baliser des documents avec des balises structurales. On imaginait à l'époque que certains documents lisibles par machine devraient rester lisibles par machine pendant peut-être des décennies. Pour cette raison SGML est considéré comme trop complexe pour une utilisation générale. XML comble cette lacune en étant lisible à la fois par l'homme et par la machine, tout en étant suffisamment flexible pour prendre en charge l'échange de données indépendant de la plateforme et de l'architecture [Har04].

Le plus grand succès de SGML était HTML, qui est une application SGML. Cependant, HTML n'est qu'une application SGML. Il n'a pas ou n'offre nulle part la pleine puissance de SGML lui-même. Puisqu'il restreint les auteurs à un ensemble fini de balises conçues pour décrire les pages Web et les décrit d'une manière assez axée sur la présentation, c'est vraiment un peu plus qu'un langage de balisage traditionnel qui a été adopté par les navigateurs Web. 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. Mais, lorsqu'il s'agit de stockage et d'échange de données, HTML est une mauvaise solution, car il était à l'origine conçu comme une technologie de présentation [You02].

La version 1.0 de XML a été publiée le 10 février 1998 par le consortium W3C (World Wide Web Consortium). Depuis, les spécifications du langage ont évolué, et la version 1.1 a été publiée le 4 février 2004. C'est pourtant la version 1.0 du langage XML qui est encore la plus utilisée à ce jour [You02].

1.2.2 Les objectifs du XML

XML est devenu omniprésent dans le monde de l'informatique, son objectif est de faciliter les échanges de données entre les machines et les logiciels. Il décrit également les données de manière aussi bien compréhensible par les hommes qui écrivent les documents XML que par les machines qui les exploitent. Donc, XML est un langage extensible et configurable pour décrire tout type de données.

Nous allons d'abord énumérer ces principaux objectifs qui ont conduit à son développement puis nous allons les détailler plus en profondeur.

1. **XML peut être directement utilisable sur Internet** : XML a été conçu principalement pour stocker et diffuser des informations sur le Web et pour prendre en charge des applications distribuées sur Internet.
2. **XML peut prendre en charge une grande variété d'applications** : Bien que son utilisation principale soit l'échange d'informations sur Internet, XML a également été conçu pour être utilisé par des programmes qui ne sont pas sur Internet, tels que des outils logiciels pour créer des documents et pour filtrer, traduire ou formater des informations.
3. **Compatibilité de XML avec SGML** : XML a été conçu pour être un sous-ensemble de SGML, de sorte que chaque document XML valide soit également un document SGML conforme, et pour avoir essentiellement la même capacité d'expression que SGML. L'un des avantages d'atteindre cet objectif est que les programmeurs peuvent facilement adapter les outils logiciels de SGML pour mieux travailler avec des documents XML.
4. **Facilité d'écriture de programmes qui traitent des documents XML** : Pour qu'un langage de balisage pour le Web soit pratique et soit universellement accepté, il doit être facile dans l'écriture des programmes qui traitent les documents. En fait, la principale raison de la définition du sous-ensemble XML de SGML était la lourdeur de l'écriture de programmes pour traiter les documents SGML.
5. **Les documents XML doivent être lisibles par l'homme et raisonnablement clairs** : Les humains peuvent facilement lire un document XML car il est écrit en texte brut et possède une structure arborescente logique. Vous pouvez améliorer la lisibilité de XML en choisissant des noms significatifs pour les éléments, les attributs et les entités de votre document ; en arrangeant soigneusement et en indentant le texte pour montrer clairement la structure logique du document en un coup d'œil ; et éventuellement, en ajoutant des commentaires utiles.
6. **Extensibilité et Flexibilité de XML** : Contrairement à HTML fixe les balises pouvant apparaître dans un document ainsi que leur imbrication possible, en XML le vocabulaire (l'ensemble des balises autorisées) n'est pas figé.



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. Ces ensembles de règles portant sur les documents XML sont appelés modèles de documents tels que les DTD ou les schémas.

* **Questions de cours** : *XML remplace-t-il HTML ?*

Réponse : Actuellement, la réponse à cette question est non. HTML est toujours le

principal langage utilisé pour indiquer aux navigateurs comment afficher des informations sur le Web. Plutôt que de remplacer HTML, XML est actuellement utilisé en conjonction avec HTML et étend considérablement la capacité des pages Web à :

- Livrez pratiquement n'importe quel type de document.
- Triez, filtrez, réorganisez, recherchez et manipulez les informations de manière différente.
- Présentez des informations très structurées.

1.3 Les briques de base

Maintenant, les choses sérieuses commencent, nous allons entrer dans la partie pratique. Dans ce chapitre, nous allons découvrir ensemble les bases du XML.

1.3.1 Structure d'un document XML

La structure de données représentée dans un fichier XML est hiérarchisée dans une arborescence (un élément racine unique ; chaque élément peut contenir d'autres éléments qui peuvent à leur tour contenir du texte ou d'autres éléments). On peut donc représenter les données d'un fichier XML sous forme d'arbre comme suit :

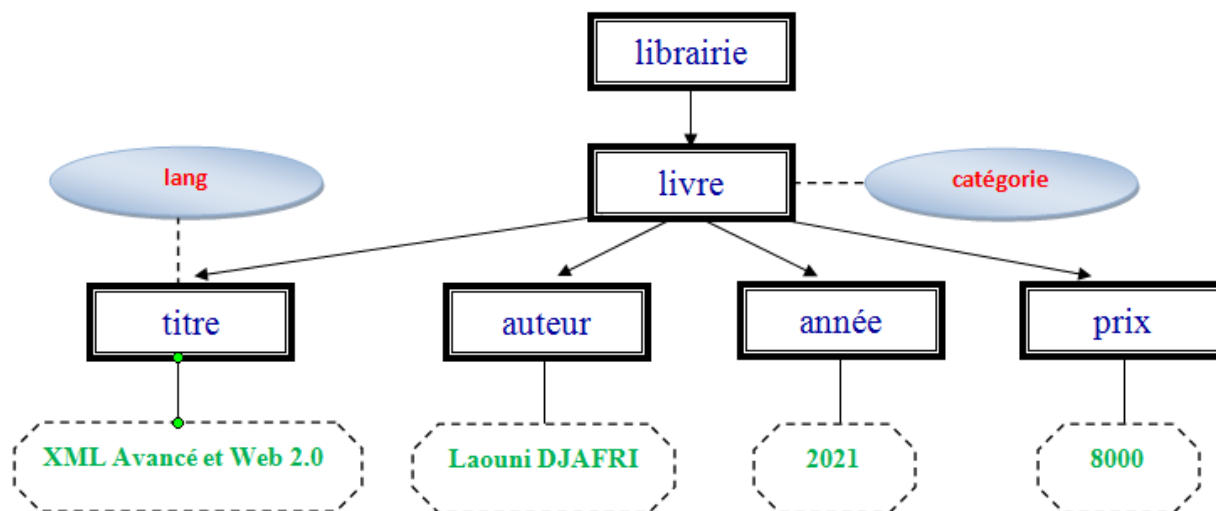


FIGURE 1.1 – Arborescence des nœuds dans un document XML.

Exemple de Document XML

Lorsque vous créez un document XML, plutôt que d'utiliser un ensemble limité d'éléments prédéfinis, vous créez vos propres éléments et vous leur attribuez les noms de votre choix, d'où le terme extensible dans le langage de balisage extensible.

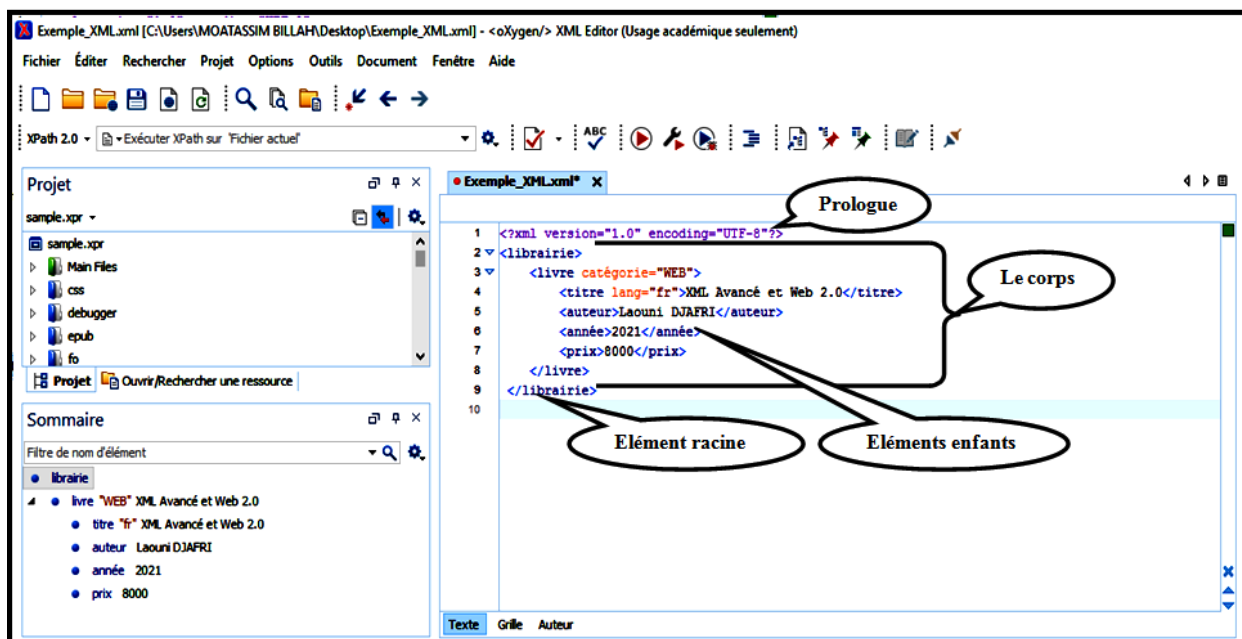


FIGURE 1.2 – Document XML représentant l'exemple de la figure précédente.

Un document XML se compose de deux parties principales : le prologue et le corps.

Le prologue

Le prologue correspond à la première ligne de votre document XML. Il indique *la version* du langage XML utilisé et *le codage des caractères* dans le document.

Comme vous pouvez le remarquer dans la figure 1.2, le prologue est une balise unique qui commence par `<?xml` et qui se termine par `?>`.

Restez avec nous, nous sommes au point de départ...

- La version

Dans le prologue, on commence généralement en indiquant la version de XML que l'on utilise pour décrire nos données. Pour rappel, il existe actuellement deux versions : `version 1.0` et `version 1.1`.

À noter : le prologue n'est obligatoire que depuis la version 1.1, mais il est plus que conseillé de l'ajouter quand même lorsque vous utilisez la version 1.0.

La différence entre les deux versions est une amélioration dans le support des différentes versions de l'Unicode. Sauf si vous souhaitez utiliser des caractères chinois dans vos documents XML, il conviendra d'utiliser la version 1.0 qui est encore aujourd'hui la version la plus utilisée.

- Le jeu de caractères

La seconde information de mon prologue est `encoding="UTF-8"`.

Il s'agit du jeu de caractères utilisé dans mon document XML. Par défaut, l'encodage de XML est l'UTF-8, mais si votre éditeur de texte enregistre vos documents en ISO8859-1, il suffit de la changer dans le prologue.

Le corps

Le corps d'un document XML est constitué de l'ensemble des balises qui décrivent les données (voir Figure 1.2). Il y a cependant une règle très importante à respecter dans la constitution du corps : une balise en paires unique doit contenir toutes les autres balises, cette balise en paires est appelée **élément racine** du corps. Il existe un et un seul élément de ce type dans un document ; c'est un élément père qui contient tous les autres éléments.

En revanche, d'autres éléments peuvent être également éléments pères. Dans le cas, de notre exemple, l'élément `livre` est en effet père de l'élément `titre` (qui est donc l'élément enfant de l'élément `livre`). Pour les éléments sans fils (comme l'élément `auteur`) on placera, comme dans l'exemple, la balise ouvrante `<auteur>`, le contenu "Laouni DJAFRI" et la balise fermante `</auteur>` sur la même ligne.

- Éléments et attributs XML [Joe12]

Parce que XML est conçu pour décrire des données et des documents, la recommandation W3C, qui peut être trouvée enfouie dans les liens sur <http://www.w3.org/> XML, est très stricte sur un petit noyau d'exigences de format qui font la différence entre un document texte contenant un tas de balises et un document XML réel. Les documents XML qui répondent aux recommandations de formatage (W3C) des documents XML sont décrits comme étant des documents XML bien formés. Les documents XML bien formés peuvent contenir des éléments, des attributs et du texte.

1- Éléments

Définition 1.3.1 Les éléments ressemblent à ceci (`<E> . . . </E>`) et ont toujours une balise d'ouverture et de fermeture : `<élément> contenu </élément>`.

Le contenu d'un élément est formé de tout ce qui se trouve entre la balise ouvrante et la balise fermante. Il peut être constitué de texte, d'autres éléments, de commentaires et d'**instructions de traitement**.

- **Texte** : Il s'agit simplement de mettre ce que vous voulez comme texte : un nombre, une lettre, une chaîne, etc.

-**Sous éléments** : peut être un élément qui possède un ou plusieurs éléments enfants (fils).

- **Les commentaires** : Pour insérer des commentaires dans un document, placez-les entre «<! - ->» et «- ->». Les commentaires sont utilisés pour les notes, l'indication de la propriété, etc. Ils sont destinés au lecteur humain et ils sont ignorés par le processeur XML. Les commentaires ne peuvent pas être insérés dans le balisage. Ils doivent apparaître avant ou après le balisage.

- **Les instructions de traitement** sont un mécanisme permettant d'insérer des instructions non XML, telles que des scripts, dans le document. Les instructions de traitement sont délimitées par les chaînes de caractères '<?' et '?>'. Les deux caractères '<?' sont immédiatement suivis du nom XML 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.

R Il existe quelques règles de base pour les éléments de document XML. Les noms d'éléments peuvent contenir des lettres, des chiffres, des tirets, des tirets bas, des points et des deux-points. Les noms d'éléments ne peuvent pas contenir d'espaces ; les tirets bas sont généralement utilisés pour remplacer les espaces. Les noms d'éléments peuvent commencer par une lettre, un tiret bas ou bien deux-points, mais ne peuvent pas commencer par d'autres caractères non alphabétiques ou un nombre, ou les lettres XML.

En outre, les règles de base, il est important de penser à utiliser des tirets ou des points dans les noms d'éléments. Ils peuvent être considérés comme faisant partie de documents XML bien formés. D'autres systèmes, tels que les systèmes de bases de données relationnelle qui utilisent les données dans le nom de l'élément, ont souvent du mal à travailler avec des tirets ou des points dans les identificateurs de données, les confondant souvent avec autre chose qu'une partie du nom.

Définition 1.3.2

Un document XML bien formé : Quand vous entendrez parler de XML, vous entendrez souvent parler de document XML bien formé ou **well-formed** en anglais qui est un document XML dont la syntaxe est correcte.

R On peut résumer un document bien formé à un document XML avec une syntaxe correcte, c'est-à-dire :

- Le document XML ne possède qu'une seule balise racine.
- Le nom des balises et des attributs est conforme aux règles de nommage (namespace).
- Toutes les balises en paires sont correctement fermées.
- Toutes les valeurs des attributs sont entre guillemets simples ou doubles.
- Les balises de votre document XML ne se chevauchent pas, il existe une arborescence dans votre document.

2- Attributs

Définition 1.3.3

Il est possible d'attacher des informations supplémentaires aux éléments sous forme d'attributs. 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. Les noms suivent les mêmes règles que les noms d'éléments.

Pour que ce soit un peu plus parlant, voici tout de suite un exemple :

```
<prix monnaie="dinar"> 8000 </prix>
```

Dans l'exemple ci-dessus, l'information principale est le prix. L'attribut `monnaie` nous permet d'apporter des informations supplémentaires sur ce prix, mais ce n'est pas l'information principale que souhaite transmettre la balise `<prix/>`.

Chaque balise ouvrante peut contenir zéro, une ou plusieurs associations de valeurs à des attributs comme dans l'exemple suivant :

```
<prix monnaie="dinar" moyen_de_paiement="CIB"> 8000 </prix>
```

Pouvons-nous traduire les attributs en sous éléments ?

* Oui, nous pouvons, et l'exemple précédent devient le suivant :

```
<prix>
  <monnaie> dinars </monnaie>
  <moyen_de_paiement> CIB </moyen_de_paiement>
  <valeur> 8000 </valeur>
</prix>
```



Vous pouvez affecter n'importe quelle valeur littérale à un attribut, à condition de respecter ces règles :

- La chaîne ne peut pas contenir le même caractère de guillemet utilisé pour la délimiter.
- Dans une balise, un attribut ne peut-être présent qu'une seule fois.
- La chaîne ne peut pas inclure le caractère esperluette (&), sauf pour commencer un caractère ou une référence d'entité. La chaîne ne peut pas inclure le caractère chevron gauche (<).

Exemple pratique 1.3.1***TP : Structuration des données en utilisant XML***

Nous vous invitons à commencer la pratique en utilisant tous les éléments théoriques que vous avez lus au cours de ce chapitre. Cela me semble indispensable pour s'assurer que vous avez bien compris toutes les notions abordées jusqu'à maintenant.

Alors, l'objectif de ce TP est de vous montrer une utilisation concrète de structuration de données via XML.

Pour ce faire, créer un document XML bien formé qui contient au moins 2 personnes. Chaque personne est connue par les informations suivantes :

- Son sexe (homme ou femme).
- Son nom.
- Son prénom.
- Son adresse.
- Un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.).
- Une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).



Nous ne vous donnons aucune indication concernant le choix des balises, des attributs et de l'arborescence à choisir pour une raison très simple : lorsque l'on débute en XML, le choix des attributs, des balises et de l'arborescence est assez difficile.

L'objectif est vraiment de vous laisser chercher et vous pousser à vous poser les bonnes questions sur l'utilité d'une balise, d'un attribut, etc.

Solution proposée :

Nous partagerons notre solution avec vous ! Notez bien que ce n'est qu'une solution parmi les multiples solutions possibles !

Vous pouvez faire la solution de plusieurs manières, par exemple, vous pouvez utiliser des éléments avec attributs ou utiliser des éléments seuls (sans attributs).

Allons donc ensemble voir la solution proposée ci-dessous, puis nous fournirons quelques explications sur cette solution proposée.

```
<?xml version="1.0" encoding="UTF-8"?>
<Personnes>
  <personne sexe="masculin">  <!-- Première personne -->
    <nom>Ali</nom>
    <prenom>Mohamed</prenom>
    <adresse>
      <numero>104</numero>
      <rue>Amir AEK</rue>
      <codePostal>14000</codePostal>
      <ville>Tiaret</ville>
      <pays>Algerie</pays>
    </adresse>
    <telephones>
      <telephone type="fixe">046 75 80 10 </telephone>
      <telephone type="portable">07 76 00 83 55</telephone>
    </telephones>
    <emails>
      <email type="personnel">P1@gmail.com</email>
    </emails>
  </personne>
  <personne sexe="feminin">  <!-- deuxième personne -->
    <nom>Fella</nom>
    <prenom>Meriem</prenom>
    <adresse>
      <numero>11</numero>
      <rue>1 novembre 1954</rue>
      <codePostal>14000</codePostal>
      <ville>Tiaret</ville>
      <pays>Algerie</pays>
    </adresse>
    <telephones>
      <telephone type="fixe">046 75 10 22 13</telephone>
    </telephones>
    <emails>
      <email type="professionnel">fella@univ-tiaret.dz</email>
    </emails>
  </personne>
</Personnes>
```

Explication de l'exemple pratique

* **Le sexe** : Comme vous pouvez le voir, nous avons choisi de renseigner le sexe dans un attribut de la balise `<personne/>` et de ne pas en faire une balise complète. En fait, nous pensons que cette information est plus utile pour l'ordinateur qui va lire le document XML que pour quiconque lit ce document. De plus, contrairement à la Machine, nous avons la capacité de déduire que "Mohamed" est un homme et "Meriem" est une femme.

* **L'adresse** : Il est important que vos documents XML aient une arborescence logique. C'est pourquoi nous avons décidé de représenter "adresse" par une balise complète (avec sous éléments ; c'est-à-dire : sans attributs) qui contient les informations détaillées de l'adresse de la personne, y compris le numéro, la rue, le code postal, la ville, le pays, etc.

* **Numéros de téléphone et adresses e-mails** : Encore une fois, dans un souci d'arborescence logique, nous avons décidé de créer les blocs `<telephones/>` et `<emails/>` qui contiennent respectivement l'ensemble des numéros de téléphone et des adresses e-mail. Pour chacune des balises `<telephone/>` et `<email/>`, nous avons décidé d'y mettre un attribut `type`. Cet attribut permet de renseigner si l'adresse e-mail ou le numéro de téléphone est par exemple professionnel ou personnel.

R Bien qu'indispensable aussi bien aux êtres humains qu'aux machines, cette information est placée dans un attribut car ce n'est pas l'information principale que l'on souhaite transmettre. Ici, l'information principale reste le numéro de téléphone ou l'adresse email et non leur type.

1.4 Exercices résolus

Exercice 1.1

Le paragraphe suivant contient de l'information "en vrac". Réorganisez-la de manière à mettre en évidence sa structure logique, en appliquant la mise en forme XML ?

Une fille possède trois robes et deux pantalons. La première robe est une robe rose d'été, la deuxième est une robe verte d'été, la troisième est une robe bleue d'hiver, le premier pantalon est un pantalon d'été blanc, le second est beige et se porte en hiver .

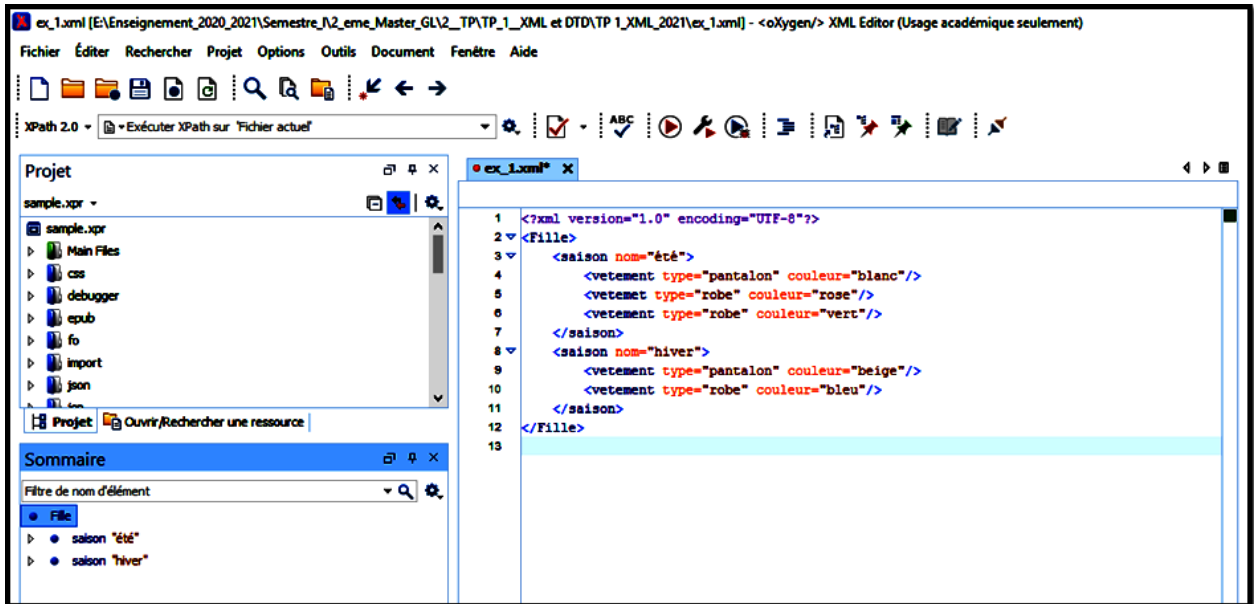
Exercice 1.2

Un polyèdre (c'est-à-dire : une forme géométrique à trois dimensions) est dit régulier s'il est constitué de faces toutes identiques et régulières. Parmi lesquels on compte le tétraèdre (à quatre faces), le cube, l'icosaèdre . . .

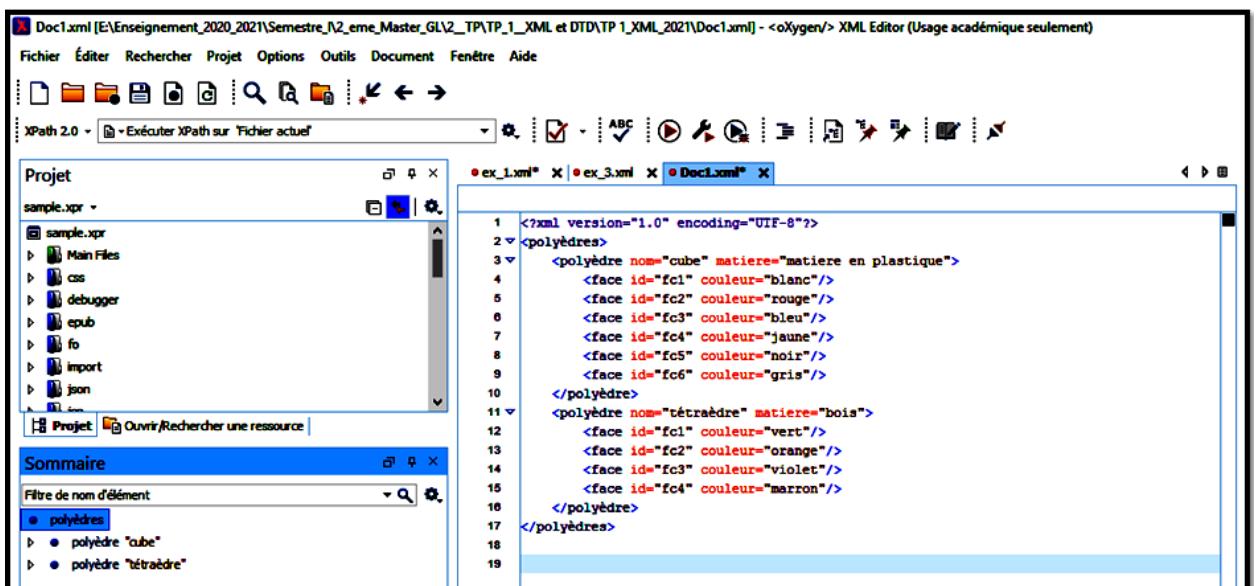
- Créer un document XML sous l'éditeur oXygen nommé 'Doc1' qui permet de faire une liste de polyèdres réguliers dans des matières a priori inconnues. pour chacun d'eux, précisez son nom, la matière dans laquelle il est réalisé, sa couleur ainsi que le nombre de ses faces. sachant que chaque face puisse avoir une couleur différente.

1.5 Solutions

1.5.1 Solution de l'exercice 1.1



1.5.2 Solution de l'exercice 1.2



1.6 Exercices ouverts

Exercice 1.3

- Modéliser et décrire (sous forme d'arbre, doc XML à l'aide de l'éditeur oXygen) la programmation d'une conférence internationale.

Pour ce faire, suivez les points suivants :

- Premièrement, nous avons besoin d'informations générales sur la conférence (nom, domaine, commission, etc.) et d'un texte présentant le lieu.
- Ensuite, nous voulons décliner une liste de conférence en spécifiant :
 1. Un titre ;
 2. Une date ;
 3. Une description ;
 4. Un tour de présentation orale, c'est-à-dire une liste des articles/auteurs (en général, il n'y aura qu'un auteur par présentation orale, mais nous devons pouvoir traiter le cas de duos où des auteurs se succédant dans le cas d'un travail commun par exemple).
- Un auteur sera décrit par son nom, son prénom, sa date de naissance, éventuellement une photo, une adresse de site web si existant, et une biographie.
- Pour chaque article, on voudra disposer d'un titre, des auteurs, et les affiliations des auteurs.
- S'il ya des articles présents dans la conférence et ont été publiés par les mêmes auteurs, dans ce cas on distinguera le titre, le résumé, le travail proposé et la conclusion.

Exercice 1.4

- Modéliser et décrire (sous forme d'arbre, doc XML à l'aide de l'éditeur oXygen) le texte ci-dessous :

Une bouteille d'eau en verre de 150 cl contient par litre 7.1 mg d'ions positifs calcium, et 5.5 mg d'ions positifs magnésium. On y trouve également des ions négatifs comme des chlorures à 20 mg par litre et des nitrates avec 1 mg par litre. Elle est recueillie à la source "Saida", dans le département de chimie. Son code barre est 6134080005003 et son pH est de 7,45. Comme la bouteille est sale, quelques autres matériaux comme du fer s'y trouvent en suspension.

Une seconde bouteille d'eau en verre a été, elle, recueillie à la source "ifri". La concentration en ions calcium est de 98 mg/l, et en ions magnésium de 4 mg/l. Il y a 3,6 mg/l d'ions chlorure et 2 mg/l de nitrates, pour un pH de 7,4. Le code barre de cette bouteille de 50 cl est 6138840001008.

Une bouteille de même contenance est de marque "nestlé", c'est bien connu pour ses sources donnant un pH neutre de 7. Elle comprend 11,5 mg/l d'ions calcium, 8,0 mg/l d'ions magnésium, 13,5 mg/l d'ions chlorures et 6,3 mg/l d'ions nitrates. Elle contient également des particules de silice. Son code barre est 6137640117008.

1.7 Conclusion

Dans ce chapitre, vous avez appris suffisamment de syntaxe XML pour pouvoir lire ou écrire des documents XML. Les documents XML contiennent des éléments et des attributs et offrent un moyen flexible et puissant d'échanger des données entre les applications et les organisations. Vous avez également appris que XML ne prédéfinit pas les éléments, que c'est à l'application de définir les éléments qui ont du sens.

Comme nous vous l'avons montré jusqu'à présent dans ce chapitre, il existe des règles très strictes pour la structure et la syntaxe de base des documents XML bien formés. Chaque document XML doit être bien formé, ce qui signifie qu'il doit répondre aux exigences minimales d'un document XML qui sont données dans la spécification XML. Si un document XML n'est pas bien formé, il ne peut pas être considéré comme un document XML. Il existe également plusieurs formats dans les limites de la syntaxe du document XML bien formé qui fournissent des moyens normalisés de représenter des types de données spécifiques.

Dans le chapitre suivant, vous apprendrez à valider des documents XML bien formé à l'aide d'une DTD (*Document Type Definition*) ou Schéma XML. Les DTDs ou les Schéma XML sont des outils de modélisation importants pour les développeurs XML, et ils sont utilisés pour mieux servir les auteurs XML.

2. Validation de documents XML

2.1 Introduction

Lors de la réalisation d'un document XML, la liberté donnée au rédacteur est très grande car le seul impératif est que le document soit *bien formé*. Peu de règles sont définies sur le nom des balises ou l'existence d'attribut. Cette liberté est problématique lors de l'utilisation de document XML dans le cadre d'échanges d'informations entre applications. Si des balises attendues par le système d'information sont manquantes, un dysfonctionnement pourra avoir lieu. Les DTDs et schéma XML permettent de résoudre ce problème. Alors, la validité d'un document XML est déterminée par une DTD ou un schéma XML. Vous avez le choix entre plusieurs formats de validation des documents XML. Cependant, les formats les plus courants et officiellement utilisés par le W3C sont les DTDs et les schéma XML, sur lesquels nous nous concentrerons dans ce chapitre.

Les documents XML sont comparés aux règles spécifiées dans une DTD ou un schéma. Un document XML bien formé qui répond à toutes les exigences d'une ou plusieurs spécifications est appelé un document XML valide. Autrement dit, *un document XML est valide, si et seulement si, est bien formé et respecte toutes les règles définies dans une DTD ou un schéma XML.*

- R** Il n'est pas courant de voir à la fois des références DTD et schéma XML dans un même document qui vérifient les mêmes règles structurelles, mais c'est un bon exemple du fait que vous pouvez combiner des références et schéma XML dans un seul document. Des références à une DTD et à un schéma XML peuvent se produire lorsqu'un document XML se compose de deux documents source ou plus. Les références DTD et de schéma XML conservent toutes les règles de structure présentes dans le document d'origine. Les références doubles peuvent également être utilisées lorsque des caractères XML illégaux sont représentés dans un document XML par des références d'entité.

2.2 Validation de documents XML par DTD

Une DTD est le moyen original de valider la structure d'un document XML et d'appliquer un formatage spécifique du texte sélectionné. Bien que l'affichage de la déclaration XML en haut de la DTD porte à croire qu'il s'agit d'un document XML, les DTD sont en fait des documents XML mal formés. En effet, ils suivent les règles de syntaxe DTD plutôt que la syntaxe de document XML. La référence est à la DTD (externe ou interne) située dans le premier élément sous la déclaration de document XML. Ceci est réalisé avec une définition DOCTYPE comme suit :

```
<!DOCTYPE element racine SYSTEM 'MonDoc.dtd'>
```

Une DTD est un bloc de balisage XML que vous ajoutez au prologue d'un document XML valide. Il peut aller n'importe où dans le prologue, en dehors des autres balises, en suivant la déclaration XML. (Rappelez-vous que si vous incluez la déclaration XML, elle doit être au tout début du document.)

Le rôle d'une DTD 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.

2.2.1 Déclaration d'élément

Dans un document XML valide créé à l'aide d'une DTD, vous devez déclarer explicitement le type de chaque élément que vous utilisez dans le document dans une déclaration de type d'élément dans la DTD. Une déclaration de type d'élément indique le nom du type d'élément et le contenu autorisé de l'élément (spécifiant souvent l'ordre dans lequel les éléments enfants peuvent apparaître). Prises ensemble, les déclarations de type d'élément dans la DTD tracent l'intégralité du contenu et de la structure logique du document. En effet, les déclarations d'éléments constituent le cœur des DTDs 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 et le type détermine les contenus valides de l'élément. On distingue : les contenus purs constitués uniquement d'autres éléments ; c'est-à-dire d'autres éléments imbriqués les uns dans les autres. Les contenus textuels constitués uniquement de texte (Dans le cas des chaînes de caractères non analysées) sont appelés "CDATA", et de même, les contenus mixtes qui mélangent éléments et texte (Dans le cas des chaînes de caractères non analysées) sont appelés (PCDATA) [GN07]. Une déclaration de type d'élément a la forme générale suivante :

```
<!ELEMENT nom de l'élément (modèle de contenu)>
```


*Le modèle de contenu peut être :***Contenu pur d'éléments**

Le contenu d'un élément est pur lorsqu'il ne contient pas de texte et se compose uniquement de ses éléments enfants. Ces enfants, à leur tour, peuvent avoir un contenu textuel, pur ou mixte. Par exemple, regardons la règle suivante :

```
<!ELEMENT personne (nom)>
```

Nous pouvons maintenant compléter notre DTD en ajoutant une règle pour la balise <nom/>. Par exemple, si l'on souhaite que cette balise contienne une valeur simple (PCDATA), on écrira :

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

Au final, la DTD de notre document XML est donc la suivante :

```
<!ELEMENT personne (nom)>
<!ELEMENT nom (#PCDATA)>
```

Cette règle signifie que la balise <personne/> contient la balise <nom/>, qui à son tour est un texte. Le document XML respectant cette règle ressemble donc à cela :

```
<personne>
  <nom> DJAFRI </nom>
</personne>
```

Le paramètre *personne* est le nom de l'élément devant satisfaire la règle. Les modèles de contenu peuvent être ANY ou EMPTY :

ANY : type d'éléments pouvant contenir n'importe quel élément défini dans la DTD. C'est-à-dire qu'un élément de ce type peut contenir zéro ou plusieurs éléments enfants de n'importe quel type déclaré, dans n'importe quel ordre ou nombre de répétitions, avec ou sans données de caractères (PCDATA) intercalées. Il s'agit de la spécification de contenu la plus laxiste et crée un type d'élément sans contraintes de contenu. Voici un exemple de déclaration :

```
<!ELEMENT personne (nom)>
<!ELEMENT nom (ANY)>
```

EMPTY : type d'élément vide. Seuls des attributs d'éléments sont alors autorisés. Prenons les règles suivantes :

```
<!ELEMENT personne (nom)>
<!ELEMENT Num de Tel (EMPTY)>
```

Les éléments suivants seraient des éléments "Num de Tel" valides que vous pourriez entrer dans votre document :

```
<Num de Tel></Num de Tel>
  Ou bien
<Num de Tel/>
```

- R** Bien que le mot-clef ANY existe, il est souvent déconseillé de l'utiliser afin de restreindre le plus possible la liberté de rédaction du document XML.

Comme pour le mot-clef EMPTY, l'usage des parenthèses n'est pas obligatoire pour le mot-clef ANY !

Contenu mixte

Avec ce type de spécification de contenu, l'élément peut contenir n'importe quelle quantité de données de caractères. De plus, si un ou plusieurs types d'éléments enfants sont spécifiés dans la déclaration, les données de caractères peuvent être intercalées avec n'importe quel nombre de ces éléments enfants, dans n'importe quel ordre.

La cardinalité

Lors de la description d'un modèle de contenu, grâce à des opérateurs de cardinalité, il est possible de préciser le nombre d'éléments enfants admis. Les opérateurs de cardinalité sont les suivants :

- 1- "+" : Un élément suivi d'un caractère « + » doit apparaître un ou plusieurs fois (au moins une seule fois) dans l'élément en cours de définition. L'élément peut se répéter.
- 2- "*" : Un élément suivi d'un caractère « * » peut apparaître zéro ou plusieurs fois dans l'élément en cours de définition. L'élément est facultatif mais, s'il est inclus, il peut se répéter indéfiniment.
- 3- "?" : Un élément suivi d'un caractère " ? " peut apparaître une fois ou pas du tout dans l'élément en cours de définition. Il indique que l'élément est facultatif et, s'il est inclus, ne peut pas être répété.

Les éléments personne et adresse ont un modèle de contenu qui utilise un indicateur d'occurrence :

```
<!ELEMENT personne (nom,adresse*,tél*,fax*,email*)>
<!ELEMENT adresse (rue,région?,code postal,ville,pays)>
```

Les enfants acceptables pour personne sont le nom, l'adresse, le tél, le fax et l'email. À l'exception du nom, ces enfants sont facultatifs et peuvent être répétés.

Les enfants acceptables pour l'adresse sont la rue, la région, le code postal, la ville et le pays. Aucun des enfants ne peut redoubler mais la région est facultative.

Les listes ordonnées et de choix

Un modèle de contenu peut avoir l'une des deux formes de base suivantes :

- **Séquence** : la forme de séquence du modèle de contenu indique que l'élément doit contenir une séquence spécifique de types d'éléments enfants. Vous séparez les noms des types d'éléments enfants par des **virgules**. Par exemple, la DTD suivante indique qu'un élément de document "annuaire" doit avoir au moins un élément enfant "personne", l'élément enfant "personne" doit avoir un élément enfant "nom" suivi d'un élément enfant "prenom", suivi d'un élément enfant "email" et suivi d'un élément enfant "tel" :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT annuaire (personne)+>
<!ELEMENT personne (nom,prenom,email+,tel*)>
<!ATTLIST personne type (etudiant | professeur | administrateur) "etudiant">
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
<!ATTLIST tel type (fixe | portable) #REQUIRED>
```

- **Choix** : la forme de choix du modèle de contenu indique que l'élément/attribut peut avoir n'importe lequel d'une série de types d'éléments enfants possibles, qui sont séparés à l'aide de "|" (barre verticale). Par exemple, la DTD précédente spécifie qu'un attribut "type" peut avoir la valeur "etudiant", ou la valeur "professeur", ou la valeur "administrateur", de même pour les éléments.

Ci-dessous le document XML correspondant à la DTD ci-dessus :

2.2.2 Déclaration d'attribut

De la même manière, comme pour les éléments, des attributs peuvent être définis pour chaque élément. Cela se fait en utilisant le mot-clé "ATTLIST", qui répertorie tous les attributs d'un élément. Vous définissez tous les attributs associés à un élément particulier en utilisant un type de déclaration de balisage DTD connu sous le nom de déclaration de liste d'attributs. Cette déclaration fait ce qui suit :

- Elle définit les noms des attributs associés à l'élément. Dans un document valide, vous pouvez inclure dans une balise de début d'élément uniquement les attributs définis pour cet élément.
- Elle spécifie le type de données de chaque attribut.
- Elle spécifie pour chaque attribut ; si cet attribut est obligatoire. S'il n'est pas obligatoire, la déclaration de liste d'attributs indique également ce que le processeur doit faire si l'attribut est omis. (La déclaration peut, par exemple, fournir une valeur d'attribut par défaut que le processeur transmettra à l'application.)

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE annuaire SYSTEM "ex_05.dtd">
<annuaire>
  <personne type="etudiant">
    <nom>malik</nom>
    <prenom>ali</prenom>
    <email>c@gmail.com</email>
    <tel type="fixe">501</tel>
  </personne>
  <personne type="professeur">
    <nom>said</nom>
    <prenom>mustapha</prenom>
    <email>g@yahoo.fr</email>
  </personne>
  <personne type="administrateur">
    <nom>saleh</nom>
    <prenom>mahmoud</prenom>
    <email>m_s@hotmail.com</email>
    <tel type="fixe">213</tel>
  </personne>
  <personne type="etudiant">
    <nom>abdelwaheb</nom>
    <prenom>ali</prenom>
    <email>ab_a@yahoo.fr</email>
  </personne>
</annuaire>

```

Alors, chaque définition d'attribut se compose d'un nom d'attribut, d'un type d'attribut et d'une valeur par défaut. La déclaration d'attribut prend la forme générale suivante :

```
<!ATTLIST nom_element nom_attribut type valeur par default>
```

- R** Vous pouvez déclarer des éléments et des attributs dans n'importe quel ordre dans une DTD. Par exemple, vous pouvez déclarer la spécification de la liste d'attributs pour un élément particulier avant de déclarer cet élément.

Les valeurs par défaut

Le paramètre `valeur_par défaut` peut prendre l'une des quatre valeurs suivantes :

1. **#REQUIRED** : Lorsqu'on souhaite qu'un attribut soit *obligatoirement* renseigné, on utilise le mot clef (`#REQUIRED`). c'est-à-dire : la présence de l'attribut est obligatoire.

Par exemple, si l'on souhaite que le sexe d'une personne soit renseigné, on utilisera la règle suivante :

```
<!ATTLIST personne sexe (masculin|féminin) #REQUIRED>
```

L'attribut `sexe` doit prendre soit la valeur `"masculin"` ou la valeur `"féminin"`, l'une de ces valeurs est obligatoire.

2. **#IMPLIED** : Si on souhaite indiquer qu'un attribut n'est pas obligatoire, on utilise le mot clef `#IMPLIED`. c'est-à-dire : la présence de l'attribut est optionnelle.

Si l'on reprend l'exemple précédent, on peut indiquer qu'il n'est pas obligatoire de renseigner le `sexe` d'une personne par la règle suivante :

```
<!ATTLIST personne sexe CDATA #IMPLIED>
```

L'attribut `"sexe"` est optionnel et il n'a pas de valeur par défaut (on peut, par exemple, écrire `sexe="1"`). Si l'attribut est absent, il n'a pas de valeur.

3. **#FIXED** : Il est possible de rendre obligatoire un attribut et de fixer sa valeur grâce au mot clef `#FIXED` suivi de ladite valeur. Cette situation peut, par exemple, se rencontrer lorsque l'on souhaite travailler dans une `"taille"` bien précise et que l'on souhaite qu'elle apparaisse dans le document. La règle suivante permet d'indiquer que la `"taille"` doit obligatoirement apparaître et a pour seule valeur possible le `"centimètre"`.

```
<!ATTLIST objet taille CDATA #FIXED "centimètre">
```

Dans ce cas, la présence de l'attribut `"taille"` est optionnelle, mais, s'il est présent, sa valeur doit être identique à la valeur précisée derrière le paramètre `#FIXED`.

4. Il est également possible d'indiquer une valeur par défaut pour un attribut. Il suffit tout simplement d'écrire cette valeur `"en dur"` dans la règle. c'est-à-dire : une valeur qui sera utilisée comme valeur par défaut en cas d'absence de l'attribut.

Par exemple, il est possible d'indiquer qu'une personne dont l'attribut `sexe` n'est pas renseigné, soit de `sexe masculin` par défaut grâce à la règle suivante :

```
<!ATTLIST personne sexe CDATA "masculin">
```



Les sections `CDATA` sont définies comme des blocs de textes et un type de nœud XML reconnu par les langages de balisage mais ne sont pas analysés par les parseurs. `CDATA` est destiné uniquement au groupe de texte spécifiant des caractères de type balisage. Seuls les délimiteurs de section `CDATA` `<![CDATA[et]]>` sont classés comme balisage. Le texte entre les délimiteurs est une donnée textuelle.

Voici ci-dessous, un exemple de section `CDATA` :

```
<![CDATA[
Ici, vous pouvez saisir n'importe quel caractère.
]]>
```

Types d'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 citée ci-après. Les types les plus utilisés sont CDATA, ID, IDREF, IDREFS, NMTOKEN et NMTOKENS [Hos13] [Kah09].

- **CDATA** : est le type le plus utilisé pour définir un attribut dont la valeur est une donnée textuelle.
- **ID** : Comme son nom le laisse deviner, un ID représente un identifiant. Il doit donc contenir des valeurs uniques. A ce titre, il est impossible de lui définir une valeur fixe ou par défaut. Comme pour d'autres types vu précédemment, un ID ne doit être utilisé qu'avec les attributs afin d'assurer une compatibilité entre les Schémas XML et les DTD.
- **IDREF** : La valeur de l'attribut doit correspondre à la valeur d'un attribut de type ID dans un élément du document. En d'autres termes, ce type d'attribut fait référence à l'identifiant unique d'un autre attribut.

R Les types d'attributs ID et IDREF peuvent être utilisés pour établir des relations entre les éléments, permettant la création de structures de documents de type réseau qui ne pourraient pas être capturées dans des structures arborescentes. En particulier, il est possible de définir des documents qui modélisent des tables relationnelles en utilisant les attributs de type ID et IDREF. ID agit comme une clé primaire, tandis que IDREF agit comme une clé étrangère. Certains analyseurs XML prennent en charge l'emplacement des éléments par ID.

- **IDREFS** : Le type IDREFS représente une liste de IDREF séparés par un espace. Afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type IDREFS que pour un attribut.
- **NMTOKEN** : Le type NMTOKEN est basé sur le type TOKEN et représente une chaîne de caractères "simple", c'est-à-dire : une chaîne de caractères sans espace qui ne contient que les symboles suivants :
 - Des lettres.
 - Des chiffres.
 - Les caractères spéciaux "(.)" "(-)" "(_)" et "(:)"

Si la chaîne de caractères contient des espaces au début ou à la fin, ils seront automatiquement supprimés.

- **NMTOKENS** : Le type NMTOKENS représente une liste de NMTOKEN séparés par un espace. Une nouvelle fois, afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type NMTOKENS que seulement pour un attribut.

Rappel 2.2.1 Les attributs de type ID doivent respecter les règles ci-dessous :

- Un seul attribut de type ID peut être attribué par élément.
- La valeur d'un attribut ID ne doit apparaître qu'une unique fois dans un document XML.
- Les attributs de type ID doivent d'être associés à la valeur par défaut #REQUIRED ou #IMPLIED.

Les attributs de type IDREF ou IDREFS doivent respecter la règle suivante : la valeur d'un tel attribut doit obligatoirement correspondre à celle d'un attribut de type ID.

2.2.3 Les entités

Une entité peut être considérée comme un alias permettant de réutiliser des informations au sein du document XML ou de la définition DTD. Il existe plusieurs types d'entités dont les trois principales sont les entités générales, utilisable dans un document XML, et les entités paramètres, utilisables uniquement dans les DTD et les entités prédéfinies. Les entités générales peuvent être internes, c'est-à-dire : définies dans la DTD, ou bien externe, c'est-à-dire : définies dans un fichier externe. Dans cette partie, nous aborderons trois types d'entités : les entités générales, les entités paramètres et les entités prédéfinies [Har04].

Entités générales

Les entités générales (analysables) sont les entités les plus simples, les plus courantes et les plus utiles puisqu'elles peuvent être utilisées dans le corps du document. Elles permettent également d'associer un alias à une information afin de l'utiliser dans le document XML.

Une entité générale est un contenu bien formé, appelé "texte de remplacement", qui peut être appelée par une référence d'entité. Les références d'entités (analysables) utilisent l'esperlutte "&" et le point-virgule ";" respectivement comme délimiteurs de début et de fin. Pour utiliser une entité générale dans notre document XML, il suffit d'utiliser la syntaxe suivante :

```
<!ENTITY abréviation "Ceci est le texte à abrégé">
```

Le code XML est :

```
&abréviation;
```

Les entités internes

Les entités internes sont des entités déclarées dans une DTD. Voici la syntaxe pour la déclaration d'entité interne :

```
<!ENTITY nom_entité "valeur_entité">
```

- nom_entité est le nom de l'entité suivi de sa valeur entre doubles ou simples guillemets.
- valeur_entité contient la valeur du nom de l'entité.
- La valeur d'entité de l'entité interne est dé-référencée en ajoutant le préfixe "&" au nom de l'entité, comme suit : &nom_entité.



- Une DTD interne est une DTD qui est écrite dans le même fichier que le document XML. Elle est généralement spécifique au document XML dans lequel elle est écrite.
- Une DTD interne s'écrit dans ce qu'on appelle le DOCTYPE. On le place sous le prologue du document et au-dessus du contenu XML.

Exemple :

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "yes"?>
<!DOCTYPE personne [
  <!ELEMENT personne (#PCDATA)>
  <!ENTITY nom "DJAFRI">
  <!ENTITY prenom "Laouni">
  <!ENTITY num_tel "213.767.243.855">
]>

<personne>
  &nom;
  &prenom;
  &num_tel;
</personne>
```

Dans l'exemple ci-dessus, les noms d'entité respectifs nom, prenom et num_tel sont remplacés par leurs valeurs dans le document XML. Les valeurs d'entité sont dé-référencées en ajoutant le préfixe & au nom de l'entité.

Les entités externes

Si une entité est déclarée en dehors d'une DTD, elle est appelée entité externe. Vous pouvez faire référence à une entité externe en utilisant le mot clé SYSTEM ou le mot clé PUBLIC. Voici la syntaxe pour la déclaration d'entité externe :

```
<!ENTITY nom_entité SYSTEM ou PUBLIC "URI ou URL">
```

Vous pouvez vous référer à une DTD externe en utilisant :

1- Le mot SYSTEM qui permet de spécifier l'emplacement d'un fichier externe contenant des déclarations DTD. Comme vous pouvez le voir, il contient le mot-clé SYSTEM et une référence URI pointant vers l'emplacement du document. La syntaxe est la suivante :

```
<!DOCTYPE nom_entité SYSTEM "personne.dtd" [...]>
```

2- Le mot PUBLIC qui permet de fournir un mécanisme pour localiser les ressources DTD. Comme vous pouvez le voir, il commence par le mot-clé PUBLIC, suivi d'un identifiant spécialisé.

```
<!ENTITY nom_entité PUBLIC "URL">
```

Exemple :

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<!DOCTYPE address SYSTEM "personne.dtd">

<personne>
  <nom>
    DJAFRI
  </nom>

  <prenom>
    Laouni
  </prenom>

  <num_tel>
    213.767.243.855
  </num_tel>
</personne>
```

Vous pouvez, ensuite, trouver le contenu du fichier DTD "personne.dtd" comme suit :

```

<!ELEMENT personne (nom, prenom, num_tel)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT num_tel (#PCDATA)>

```

Entités paramètres

Contrairement aux entités générales utilisées dans les documents XML, les entités paramètres ne peuvent être utilisées que dans les DTD. Elles permettent d'associer un alias à une partie de la déclaration de la DTD. Le but d'une entité de paramètre est de vous permettre de créer des sections réutilisables de texte de remplacement. Voyons tout de suite la syntaxe d'une entité paramètre :

```
<!ENTITY % nom "valeur">
```

Pour utiliser une entité paramètre dans notre DTD, il suffit d'utiliser la syntaxe suivante :
%nom;

Prenons par exemple ce cas où des téléphones ont pour attribut une marque :

```

<telephone marque="Samsung" />
<telephone marque="Apple" />

```

Normalement, pour indiquer que l'attribut marque de la balise <telephone/> est obligatoire et qu'il doit contenir la valeur Samsung ou Apple, nous devons écrire la règle suivante :

```
<!ATTLIST telephone marque (Samsung|Apple) #REQUIRED>
```

À l'aide d'une entité paramètre, cette même règle s'écrit de la façon suivante :

```

<!ENTITY % listeMarques "marque (Samsung|Apple) #REQUIRED">
<!ATTLIST telephone %listeMarques; >

```

Encore une fois, au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives.

Entités prédéfinies

Tous les formats de référence de caractère définis précédemment incluent une esperluette (&). Alors, comment représentez-vous une esperluette dans des documents XML ? Pour accommoder les esperluettes et quatre autres caractères spéciaux qui font partie de la syntaxe principale XML, des références de caractères spéciaux réservés sont définies. Les symboles inférieurs et supérieurs (qui sont utilisés pour définir les éléments XML) et les

Entité prédéfinie	Caractère inséré	Caractère équivalent
&	&	&
<	<	<
>	>	>
'	'	'
"	“	"

TABLE 2.1 – Entités prédéfinies

guillemets (qui sont utilisés pour définir les valeurs d'attribut) sont pris en charge via des substitutions de caractères spéciales prédéfinies sans aucune référence Entité, Unicode ou Hex. Une esperluette (&) est utilisée au début et un point-virgule (;) est placé à la fin de la référence. Le Tableau 2.1 montre l'entité de caractère réservé et sa référence.

L'insertion d'une de ces références d'entité prédéfinies équivaut à l'insertion de la référence de caractère correspondante. Les références d'entités prédéfinies sont simplement plus faciles à mémoriser et à comprendre lorsque vous les voyez dans un document.

Les entités prédéfinies fonctionnent comme les autres entités internes générales, sauf que vous pouvez utiliser des références à celles-ci sans déclarer les entités. Vous pouvez insérer des entités prédéfinies aux mêmes endroits que les entités internes générales, à savoir :

- Dans le contenu d'un élément.
- Dans la valeur d'un attribut (la valeur par défaut dans une déclaration d'attribut ou la valeur d'attribut attribuée dans la balise de début d'un élément).
- Dans la valeur littérale d'une déclaration d'entité interne.

Dans les trois exemples suivants, des références d'entité prédéfinies sont utilisées pour insérer des caractères qu'il serait illégal d'insérer littéralement.

Dans le premier exemple, < est utilisé pour insérer un chevron gauche (<) dans le contenu d'un élément :

```
<Titre> &lt; XML avancé & Web 2.0> </Titre>
```

Dans le deuxième exemple, & est utilisé pour insérer une esperluette (&) dans une valeur d'attribut :

```
<Produit entreprise="Polyma & Plastic"> Sacs Bleus </Produit>
```

Dans le troisième exemple, " est utilisé pour insérer un guillemet double (") dans une valeur d'entité (ce qui serait illégal d'entrer littéralement car c'est le même caractère utilisé pour délimiter la chaîne) :

```
<!ENTITY Titre "XML &quot;Avancé&quot; & &quot;Web 2.0&quot; Livre">
```

2.2.4 Les DTD internes et externes

Les DTD internes

Définition 2.2.1 Une DTD interne est une DTD qui est écrite dans le même fichier que le document XML. Elle est généralement spécifique au document XML dans lequel elle est écrite.

Une DTD interne s'écrit dans ce qu'on appelle le DOCTYPE. On le place sous le prologue du document et au-dessus du contenu XML. Voyons plus précisément la syntaxe :
<!DOCTYPE racine []>

La DTD interne est ensuite écrite entre des crochets []. Dans ce DOCTYPE, le mot racine doit être remplacé par le nom de la balise qui forme la racine du document XML.

Les DTD externes

Définition 2.2.2 Une DTD externe est une DTD qui est écrite dans un autre document que le document XML. Si elle est écrite dans un autre document, c'est que souvent, elle est commune à plusieurs documents XML qui l'exploitent.

De manière générale, afin de bien séparer le contenu XML de sa définition DTD, on prendra l'habitude de créer plusieurs fichiers afin de les séparer.

L'étude de la syntaxe d'une DTD externe est l'occasion de vous révéler qu'il existe en réalité deux types de DTD : les DTD externes PUBLIC et les DTD externes SYSTEM (Exactement, comme les entités générales). Dans les deux cas, et comme pour une DTD interne, c'est dans le DOCTYPE que cela se passe. Voyons plus précisément la syntaxe :

```
<!DOCTYPE racine SYSTEM/PUBLIC "URI/URL">
```

R Dans le prologue, si et seulement si :

1- standalone="yes" : Cette information indique que votre DTD est interne, c'est-à-dire : Cette information permet de savoir si votre document XML est autonome ou si un autre document lui est rattaché.

2- standalone="no" : Dans le cas d'une DTD externe, nos documents XML ne sont plus autonomes, en effet, ils font référence à un autre fichier qui fournit la DTD. Afin que le document contenant la DTD soit bien pris en compte, nous devons l'indiquer en passant simplement la valeur de l'attribut standalone à "no".

Exemple pratique 2.2.2

Un document XML est utilisé pour pouvoir échanger entre deux agendas électroniques les informations de leur annuaire téléphonique respectif. Malheureusement, ces agendas n'utilisent pas la même application. Ainsi, pour éviter de vérifier dans chaque application la conformité du fichier XML, on utilise une DTD pour valider ces documents XML échangés.

- Ecrivez cette DTD sous l'éditeur oXygen, de façon la plus précise possible, qui respecte les règles suivantes :

- L'élément racine aura pour nom « agenda » et ne contiendra que des éléments « personne ».
- Une réponse est représentée par l'élément personne qui contiendra obligatoirement dans cet ordre :
 - Un élément « nom » pour le nom de la personne ;
 - Un élément « prénom » pour le prénom de la personne ;
 - Un élément « adresse » qui comprendra d'autre élément ;
 - Un élément « téléphone » pour le numéro de téléphone de la personne.
- L'élément adresse devra contenir dans cet ordre les éléments « rue », « code_postal », « ville ».

Solution :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT agenda (personne*)>
<!ELEMENT personne (nom,prénom,adresse,téléphone)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prénom (#PCDATA)>
<!ELEMENT adresse (rue,code_postal,ville)>
<!ELEMENT téléphone (#PCDATA)>
<!ELEMENT rue (#PCDATA)>
<!ELEMENT code_postal (#PCDATA)>
<!ELEMENT ville (#PCDATA)>
```

Inconvénients de la DTD

- Les DTD ne sont pas au format XML. Nous avons dû apprendre un nouveau langage avec sa propre syntaxe et ses propres règles.
- Les DTD ne permettent pas de typer des données. Comme vous avez pu le constater, on se contente d'indiquer qu'une balise contient des données, mais impossible de préciser si l'on souhaite que ça soit un nombre entier, un nombre décimal, une date, une chaîne de caractères, etc.
- Il n'est pas orienté objet. Par conséquent, le concept d'héritage ne peut pas être appliqué sur les DTD.
- Possibilités limitées d'exprimer la cardinalité des éléments.

2.3 Validation de documents XML par Schéma XML

Les DTD sont un moyen simple de définir les éléments et attributs autorisés lors de la rédaction d'un document XML. Néanmoins, leur usage est parfois frustrant car ce langage ne permet pas d'exploiter à sa pleine mesure les possibilités de XML. Le W3C a donc développé un autre langage de définition de contenu, le schéma XML. Un fichier écrit selon ce format est lui-même un document XML valide. Le schéma XML offre davantage de possibilités que les DTD. Cette section de ce chapitre présente les avantages ainsi que la syntaxe de base, débutant par les grandes lignes de la déclaration d'éléments et d'attributs, avant d'exploiter la définition et l'utilisation des types de données possibles avec les schémas XML.

Un schéma et un document XML décrit par ce schéma sont stockés dans des fichiers séparés. À cet égard, un schéma est similaire à une DTD externe, qui est stockée séparément du document qu'elle contraint. Le schéma lui-même est en fait un type spécial de document XML, en particulier, c'est un document XML qui est écrit selon les règles données dans la spécification W3C Schéma XML. Ces règles constituent un langage, connu sous le nom de langage de définition de schéma XML -*XML Schema Definition*-, qui est une application spécifique de XML, d'où les lettres (**xsd**) ou tout simplement (**xs**) utilisées par convention à la fois pour le préfixe d'espace de noms Schéma XML et l'extension de fichier de schéma.

Un schéma XML décrit les éléments et les attributs qui peuvent être contenus dans un document conforme et les manières dont les éléments peuvent être organisés dans la structure hiérarchique du document. Les schémas XML permettent également, comme les DTD, 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. C'est pour pallier les lacunes des DTD que les schémas XML ont été créés. Si ces derniers proposent au minimum les mêmes fonctionnalités que les DTD, ils en apportent également de nouvelles. Il va donc falloir vous concentrer de plus en plus !

2.3.1 Structure de base d'un Schéma XML

Un schéma XML a tendance à être "verbal", c'est-à-dire : à prendre beaucoup plus de place que la syntaxe équivalente pour une DTD. Il est donc particulièrement important de respecter une organisation interne rigoureuse. Dans la suite de ce polycopié, nous adopterons la structure générale ci-dessous, qui simplifie les déclarations d'éléments et d'attributs. Chaque élément est déclaré avec un type qui peut être, soit un des types prédéfinis (*primitifs*), soit un nouveau type (*simple* ou *complexe*) défini dans le schéma [Wal12] [Bou21]. 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>
```

2.3.2 Les déclarations d'éléments et d'attributs

Le cœur de l'utilisation d'un schéma lors de la rédaction d'un document XML consiste bien entendu en ses éléments et attributs. Tout comme une DTD, il faut les déclarer.

Déclaration d'éléments

Un élément se déclare à l'aide de l'élément «element». Il faut spécifier le nom de l'élément à l'aide de l'attribut «name», ainsi que son type à l'aide de l'attribut «type». 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. La déclaration la plus simple d'un élément prend la forme suivante :

```
<xs:element name="prénom" type="xs:string"/>
<xs:element name="NumDeTel" type="xs:TypeNumDeTel"/>
```

Dans la spécification XML de base, le terme type d'élément fait référence à une classe d'éléments portant le même nom et que vous avez éventuellement déclaré à l'aide d'une déclaration de type d'élément dans une DTD. Cependant, la spécification schéma XML utilise le terme type d'élément ou type d'attribut dans un sens un peu plus étroit pour désigner spécifiquement le type de données de l'élément ou de l'attribut, c'est-à-dire le contenu et les attributs autorisés d'un élément ou les valeurs autorisées de un attribut. La spécification de type n'est qu'une partie d'une déclaration d'élément ou d'attribut.

Déclaration d'attributs

À la différence des éléments, un attribut ne peut être que de type simple ; il ne peut contenir ni élément enfant, ni attribut (ce qui est conforme aux DTD et à la notion de document XML bien formé).

Un attribut se déclare de manière comparable à un élément , à l'aide du mot clé attribute et des attributs name et type. L'exemple suivant illustre ainsi la déclaration d'un attribut "mise_à_jour" de type "date" .Il s'agit d'un autre type simple que l'on verra plus loin, et qui indique la date de dernière mise à jour de la liste des contacts.

```
<xs:attribute name="mise_à_jour" type="xs:date"/>
```

R Les éléments pouvant contenir des éléments enfants sont de **type complexes**, les autres sont de **type simple**.

Si vous ne comprenez pas jusqu'à maintenant, pas de panique, nous allons simplifier, décortiquer, et disséquer toutes les choses ensemble pas à pas...!!!

2.4 Les types simples

Définition 2.4.1 On définit un type simple à l'aide de l'élément «simpleType» dans le schéma en indiquant son nom grâce à l'attribut «name». Les types simples peuvent être des restrictions au type initial, des listes ou des unions [Kah09].

Exemple :

```
<xs:simpleType name="Byte">
  <!--définition du types: restrictions, listes ou unions-->
</xs:simpleType>
```

Pour déclarer un élément (ou attribut) avec un type simple, vous pouvez utiliser un type simple intégré, c'est-à-dire un type défini dans le cadre du langage de définition de schéma XML. Vous pouvez également utiliser un nouveau type simple que vous définissez en le dérivant d'un type simple existant.

2.4.1 Les types simples définis dans la recommandation

La spécification des W3C définit plusieurs catégories de types [Ste03], disponibles tels quels dans un schéma :

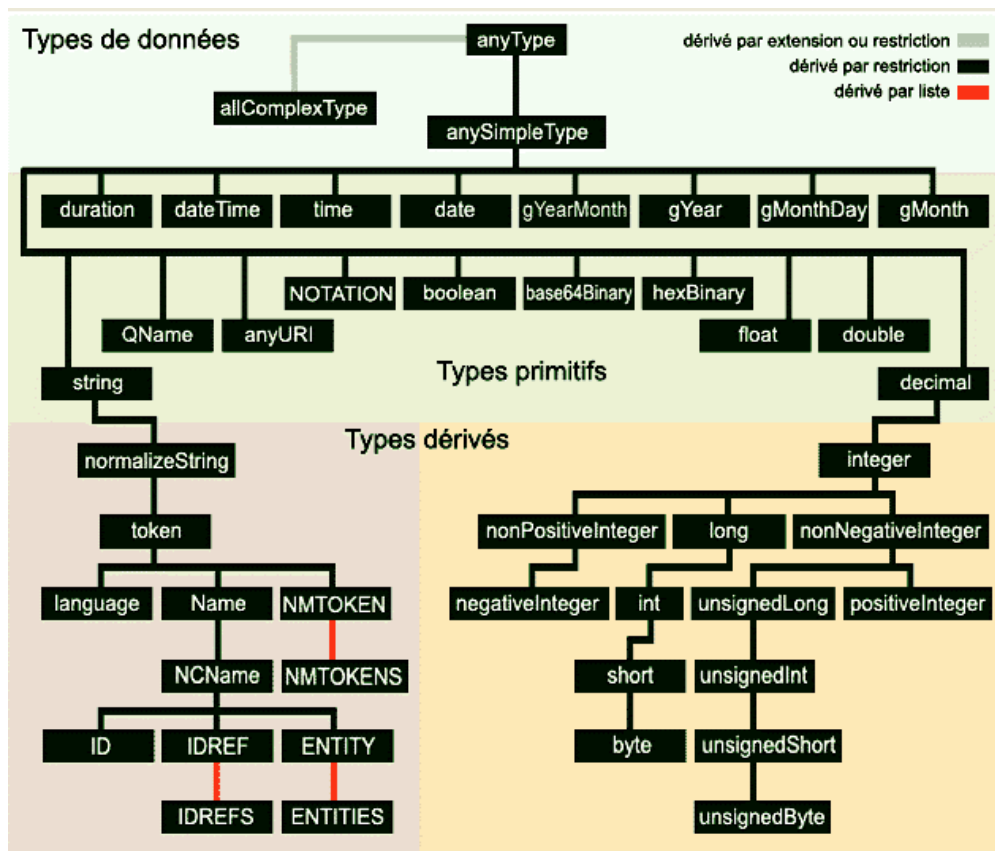


FIGURE 2.1 – Hiérarchie des types atomiques.

- R** Un attribut d'une balise peut être considéré comme un élément simple. En effet, la valeur d'un attribut est un type simple. Les types primitifs et les types dérivés sont également considérés comme des types simples dans un schéma XML.

Nous allons déterminer par la suite les types de données que ce soit de types primitifs ou bien de types dérivés, comme le montre la Figure 2.1.

- **Les types primitifs** qui servent de base à l'ensemble des autres types.
- **Les types dérivés** des types primitifs.

Rappel 2.4.1 Les types primitifs sont des types qui ne sont pas dérivés d'autres types, tels que les axiomes en mathématiques. Tandis que les types dérivés sont obtenus par **dérivation** directe à partir des types primitifs.

Les types primitifs

Les types primitifs sont les suivants :

- **string** : une séquence de l'un des caractères XML légaux.
- **boolean** : il peut valoir true ou false, 0 ou 1.
- **decimal** : représente un nombre décimal sans limite de précision. Ce nombre peut être positif ou négatif et donc être précédé du symbole + ou -. Dans le cas d'un nombre où la partie entière est égale à zéro, il n'est pas obligatoire de l'écrire.
- **float** : représente un nombre flottant sur 32 bits conforme à la norme IEEE 754. Il représente un nombre flottant, c'est-à-dire un nombre entier ou décimal, se trouvant entre les valeurs 3.4×10^{-38} et 3.4×10^{38} .
- **double** : représente un nombre flottant sur 64 bits conforme à la norme IEEE 754. Le type double est très proche du type float, Concrètement, la différence entre le type double et le type float se traduit par le fait qu'un nombre de type double se trouvant entre les valeurs 1.7×10^{-308} et 1.7×10^{308} .
- **date** : Le type date permet d'exprimer une date. A l'image du type duration, une date s'exprime selon une expression bien spécifique à savoir **YYYY-MM-DD**. Une nouvelle fois, nous vous proposons de décortiquer tout ça :

YYYY : représente l'année (year) sur 4 chiffres ou plus.

MM : représente le mois (month) sur 2 chiffres.

DD : représente le jour (day) également sur 2 chiffres.

- **time** : Le type time permet d'exprimer une heure. Encore une fois, une expression bien spécifique doit être respectée : **hh:mm:ss**. Pour continuer avec nos bonnes habitudes, décortiquons ensemble cette expression :

hh : représente les heures (hour) sur 2 chiffres.

mm : représente les minutes (minute) sur 2 chiffres.

ss : représente les secondes (second) sur 2 chiffres entiers ou à virgule.

- **dateTime** : Le type `dateTime` peut être considéré comme un mélange entre le type **date** et le type **time**. Ce nouveau type permet donc de représenter une date et une heure. Une nouvelle fois, une expression particulière doit être respectée : **YYYY-MM-DDThh :mm :ss**.
- **gDay** : représente un jour sur 2 chiffres précédés du symbole `—DD`.
- **gMonth** : représente un mois sur 2 chiffres précédés du symbole `—MM`.
- **gMonthDay** : représente un mois et un jour. Une nouvelle fois, une expression particulière doit être utilisée afin d'exprimer ce nouveau type : `—MM-DD`.
- **gYear** : représente une année sur 4 chiffres ou plus. Dans le cas où l'on souhaite exprimer une année avant Jésus-Christ, un signe `-` peut être placé devant l'expression.
- **gYearMonth** : représente une année et un mois. Comme pour tous les types que nous venons de voir dans ce chapitre, le type `gYearMonth` doit respecter une expression particulière : `YYYY-MM`.
- **hexBinary** : représente une donnée binaire au format hexadécimal, par exemple de la forme `0EF8`, ou `bc1`.
- **base64Binary** : représente une donnée binaire au format Base 64.
- **anyURI** : une URI (Unique Resource Identifier), sorte d'URL étendue indiquant l'emplacement des ressources sur internet. 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 ci-dessous :

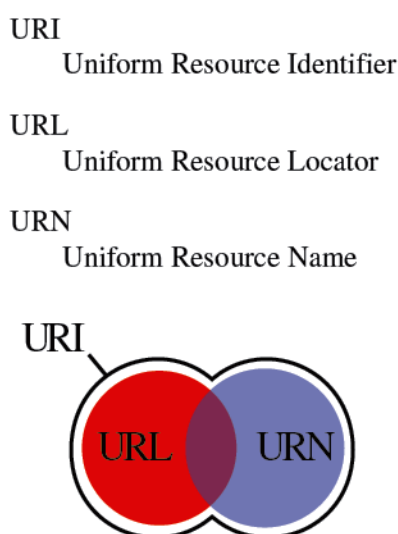


FIGURE 2.2 – Relations entre URI, URL et URN.

- **QName** : qui est l'abréviation de *-Qualified Name-*, il représente un nom qualifié comprenant par exemple la référence à un espace de noms.
- **NOTATION** : un type de données analogue au type d'attributs notation dans une DTD.

Les types dérivés

Les types dérivés des types primitifs sont :

- **normalizedString** : ce type est dérivé de type **string**. Il s'agit d'une 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.
- **token** : ce type est dérivé de type **normalizedString**. Il s'agit d'une chaîne de caractères normalisée et ne contenant pas en outre des espaces en début ou en fin ou des espaces consécutifs.
- **language** : il s'agit d'une chaîne de caractère représentant les définitions internationales de langue (norme RFC 3066). Par exemple, **fr** pour le français, **de** pour l'allemand, ou **en/us** pour l'anglais parlé aux États-Unis. Il est basé sur le type **token**.
- **Name** : représente une chaîne de caractère commençant par une lettre ou un plusieurs signes de ponctuation et continuant par des lettres, des chiffres, ou des signes de ponctuation, ou bien les caractères tiret bas (**_**), deux points (**:**), ou un point (**.**). Cependant, les chaînes commençant par **xml** : (en minuscules ou en majuscules) sont interdites.
- **NCName** : qui est l'abréviation de *-No Colon Name-*, il représente un nom XML sans le caractère deux points (**:**). Il est également basé sur le type **Name**.
- **integer** : ce types est basé sur le type **decimal**. Il représente un nombre entier sans limite de précision. Le signe (**+**) en tête est interdit, ainsi que les espaces au début.
- **long** : ce type est dérivé de type **integer**. Il représente un nombre entier signé sur 64 bits dont l'intervalle est -9223372036854775808 - 9223372036854775807.
- **int** : le type **int** est basé sur le type **long**. Il représente un nombre entier signé sur 32 bits dont l'intervalle est -2147483648 - 2147483647.
- **short** : le type **short** est basé sur le type **int**. Il représente un nombre entier court dont l'intervalle est -32768 - 32767.

- **byte** : le type `byte` est basé sur le type `short`. Il représente un entier dont l'intervalle est -128 - 127.
- **nonPositiveInteger** : ce type est basé sur le type `integer`, il représente un nombre entier négatif incluant le zéro.
- **negativeInteger** : ce type est basé sur le type `nonPositiveInteger`, il représente un nombre entier négatif dont la valeur maximum est -1 (strictement inférieur à 0).
- **nonNegativeInteger** : ce type est basé sur le type `integer`, il représente un nombre entier positif incluant le zéro (strictement supérieur à 0).
- **positiveInteger** : ce type est basé sur le type `nonNegativeInteger`, il représente un nombre entier positif commençant à 1.
- **unsignedLong** : ce type est basé sur le type `nonNegativeInteger` et représente un nombre entier long non-signé dont l'intervalle est 0 - 18446744073709551615.
- **unsignedInt** : le type `unsignedInt` est basé sur le type `unsignedLong` et représente un nombre entier non-signé dont l'intervalle est 0 - 4294967295.
- **unsignedShort** : le type `unsignedShort` est basé sur le type `unsignedInt` et représente un nombre entier court non-signé dont l'intervalle est 0 - 65535.
- **unsignedByte** : le type `unsignedByte` est basé sur le type `unsignedShort` et représente un nombre entier non-signé dont l'intervalle est 0 - 255.
- **NMTOKEN** : le type `NMTOKEN` est basé sur le type `token` applicable uniquement aux attributs, et représente une chaîne de caractère "simple". Le type d'attribut `NMTOKEN` utilisé dans les DTD.
- **NMTOKENS** : représente une liste de `NMTOKEN`, il est également applicable uniquement aux attributs. Le type d'attribut `NMTOKEN` est utilisé dans les DTD.
- **ID** : représente un identifiant unique, il est basé sur le type `NCName` applicable uniquement aux attributs. Le type d'attribut `ID` est utilisé dans les DTD.
- **IDREF** : référence à un identifiant, il est basé sur le type `NCName` applicable uniquement aux attributs. Le type d'attribut `IDREF` est utilisé dans les DTD.
- **IDREFS** : référence une liste d'identifiants, applicable uniquement aux attributs. Le type d'attribut `IDREFS` est utilisé dans les DTD.
- **ENTITY** : permet de faire référence à une entité le plus souvent non XML et déclaré dans des fichiers DTD. Ce type est basé sur le type `NCName`. Le type d'attribut `ENTITY` est également utilisé dans les DTD.

- **ENTITIES** : permet de faire référence à une liste d'ENTITY séparés par un espace. Puisque c'était déjà le cas pour le type ENTITY, le type ENTITIES n'échappe pas à la règle et ne doit être utilisé qu'avec un attribut. Le type d'attribut ENTITIES est aussi utilisé dans les DTD.

2.4.2 Les types simples définis par le concepteur

La définition d'un nouveau type simple nécessite de partir d'un type simple déjà existant, qu'il soit prédéfini ou déjà défini par le constructeur dans le schéma (voir la figure 2.3). Il faut ensuite dériver le nouveau type en appliquant des *restrictions* au type initial, ou bien en établissant une *liste* ou une *énumération* de valeurs possibles. Les restrictions autorisées dépendent du types original [Ste03][All02].

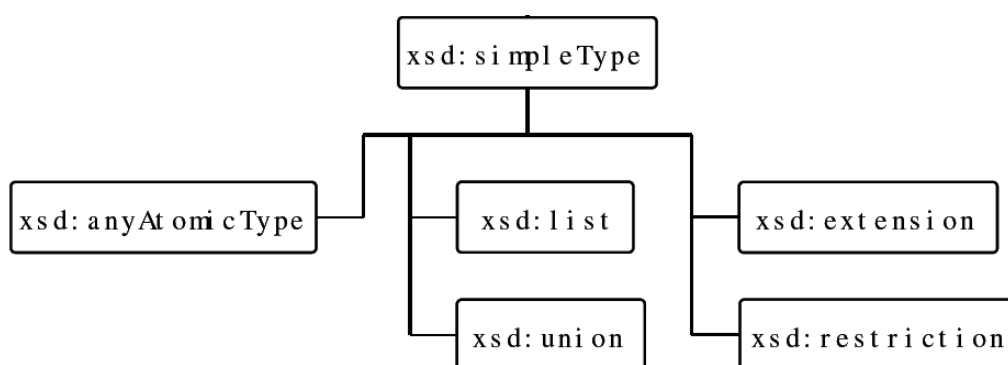


FIGURE 2.3 – Hiérarchie de construction de type simple.

Les restrictions

Un type simple par restriction est effectuée à l'aide de facettes¹ qui imposent des contraintes aux contenus. Les facettes permettent de modifier les formats autorisés pour un type donné. Le terme de restriction qui provient de la recommandation, n'est pas explicite car il est possible d'appliquer une facette qui permettra d'étendre les formats autorisés pour un type. On peut appliquer plusieurs facettes à un même types .

On définit une restriction à l'aide de l'élément « restriction », on indique à l'aide de l'attribut « base ». Il existe trois types de restrictions :

a. Restriction par intervalle

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

1. traduction du terme original anglais *facet*. Une facette permet de placer une contrainte sur l'ensemble des valeurs que peut prendre un tube de base.

Exemple :

Dans l'exemple suivant, le type donné à l'élément `year` est un entier entre 2000 et 3000 inclus. Le type utilisé dans cet exemple est un type anonyme.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="year">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="2000"/>
        <xs:maxInclusive value="3000"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

Dans l'exemple précédent, `minInclusive` et `maxInclusive` place respectivement une borne inférieure et une borne supérieure (inclusive) à la plage de valeurs autorisées.



Les facettes autorisées pour les chaînes de caractères sont : `length`, `minLength` et `maxLength`.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:attribute name="date">
    <xs:simpleType>
      <xs:restriction base="xs:date">
        <!-- Date après le 1er janvier 2020 exclus -->
        <xs:minExclusive value="2020-01-01"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:schema>
```



L'espace de noms doit aussi être spécifié sur le type prédéfini duquel est dérivé le type que l'on définit.

b. Restriction par énumération

Il est possible de restreindre les valeurs de type simple en donnant, avec l'élément `<xsd:enumeration>`, une expression rationnelle qui décrit *une liste des valeurs possibles* d'un type prédéfini ou déjà défini. Autrement dit, La restriction enumeration, comme son nom le laisse deviner permet d'énumérer la liste des valeurs possibles pour un élément ou un attribut, sa syntaxe est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xsd:element name="mon_nom">
    <xsd:simpleType>
      <xsd:restriction base="type de base">
        <xsd:enumeration value="valeur1" />
        <xsd:enumeration value="valeur2" />
        <!-- liste des valeurs... -->
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Vous pouvez limiter le contenu autorisé à un ensemble de chaînes spécifiques en dérivant le type de l'élément de `xsd:string` à l'aide des facettes `xsd:enumeration`, comme indiqué ici :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xsd:element name="RELIURE">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="couverture rigide"/>
        <xsd:enumeration value="papier grand public"/>
        <xsd:enumeration value="papier commercial"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

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`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <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>
</xsd:schema>
```

c. Restriction par motif

Il est possible de restreindre les valeurs de type simple en donnant, avec l'élément `xsd:pattern`, une expression rationnelle² (également appelées expressions régulières) qui *décrit les modèles possibles* d'un type prédéfini ou déjà défini. Le contenu est valide s'il est conforme à l'expression rationnelle. Ainsi, à l'aide des expressions rationnelles, il est possible de dire que le contenu attendu doit forcément débuter par une majuscule, se terminer par un point, débuter par un caractère spécial, ne pas contenir la lettre "z", etc. Voici sa syntaxe ci-dessous :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="mon_nom">
    <xs:simpleType>
      <xs:restriction base="type_de_base">
        <xs:pattern value="ma_valeur" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

Dans l'exemple suivant, le type ISBN décrit explicitement toutes les formes possibles des numéros ISBN.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:simpleType name="ISBN">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d-\d{2}-\d{6}-[\dX]"/>
      <xs:pattern value="\d-\d{3}-\d{5}-[\dX]"/>
      <xs:pattern value="\d-\d{4}-\d{4}-[\dX]"/>
      <xs:pattern value="\d-\d{5}-\d{3}-[\dX]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Question : Peut-on définir un type limité aux chaînes de caractères minuscules ?

Réponse : Oui on peut !

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:simpleType name="typeChaineMinuscules">
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]*" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

2. Les expressions rationnelles sont un langage à part entière sur lequel nous n'allons pas revenir dans ce cours, mais ne vous inquiétez pas, vous trouverez facilement sur Internet de nombreuses ressources francophones sur le sujet.

Les listes

Les listes sont des suites de type simple. Elles permettent d'indiquer comme format valable une suite 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'espace qui perturberaient la séparation entre les valeurs. L'exemple suivant définit des types pour des listes d'entiers et pour des listes de 5 entiers.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <!-- Type pour les listes d'entiers -->
  <xs:simpleType name="IntList">
    <xs:list itemType="xs:integer"/>
  </xs:simpleType>
  <!-- Type pour les listes de 5 entiers -->
  <xs:simpleType name="IntList5">
    <xs:restriction base="IntList">
      <xs:length value="5"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Les unions

L'opérateur `<xsd:union>` définit un nouveau type simple, soit une chaîne de caractère, soit un nombre, dont les valeurs sont celles des types listés dans l'attribut `memberTypes`. Par conséquent, si on désire qu'un type autorise soit une chaîne de caractère soit un nombre. Il est possible de le faire à l'aide d'une déclaration d'union. Par exemple, si l'on a défini au préalable un type `typeCodePostalAlgérie` limité aux codes postaux Algérie, mais que l'on souhaite qu'un élément "ville" puisse accepter au choix le code postal ou le nom de la commune, alors on peut recourir à une union.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:simpleType name="typeVille">
    <xs:union memberTypes="typeCodePostalAlgérie xs:string"/>
  </xs:simpleType>
</xs:schema>
```

Les éléments suivants (code XML) sont des instances validées de cette déclaration :

* Autres facettes utiles

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

xsd:fractionDigits et xsd:totalDigits

1- La restriction `totalDigits` s'applique à un élément de type numérique et permet de définir le nombre exact de chiffres qui composent le nombre. Sa valeur doit obligatoirement être supérieure à zéro.

```
<?xml version="1.0" encoding="UTF-8"?>
<Villes>
  <ville>Tiaret</ville>
  <ville>14000</ville>
  <ville>Mascara</ville>
  <ville>29000</ville>
</Villes>
```

Exemple :

Prenons l'âge d'une personne. Imaginons un contexte dans lequel l'âge d'une personne doit obligatoirement être compris entre 10 et 99. Il doit donc être obligatoirement composé de 2 chiffres :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="personne">
    <xs:attribute name="age">
      <xs:simpleType>
        <xs:restriction base="xs:nonNegativeInteger">
          <xs:totalDigits value="2" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

Le code XML est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<personne_age>
  <!-- valide -->
  <personne age="18" />
  <!-- valide -->
  <personne age="43" />
  <!-- invalide -->
  <personne age="4" />
</personne_age>
```

2- La restriction `fractionDigits` s'applique à un élément de type numérique et permet de définir le nombre maximal de chiffres qui composent une décimale. Sa valeur peut-être supérieure ou égale à zéro, sa syntaxe est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="mon_nom">
    <xs:simpleType>
      <xs:restriction base="type de base">
        <xs:fractionDigits value="ma_valeur" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```



Les deux facettes `xsd:fractionDigits` et `xsd:totalDigits` 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 et exclut les types `xsd:float` et `xsd:double`.

`xsd:whiteSpace`

Cette facette est particulière, elle permet de spécifier le comportement à adopter par un élément lorsqu'il contient des espaces. Les espaces peuvent être de différentes formes comme par exemple les tabulations, les retours à la ligne, etc. La syntaxe est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="mon_nom">
    <xs:simpleType>
      <xs:restriction base="type_de_base">
        <xs:fractionDigits value="ma_valeur" />
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

La facette `xsd:whiteSpace` ne restreint pas les valeurs valides mais elle modifie le traitement des caractères d'espacement 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.

Les trois valeurs possibles sont :

- **Preserve** : cette valeur permet de *préserver* tous les espaces. En d'autres termes, les caractères d'espacement sont laissés inchangés par l'analyseur lexical.
- **replace** : cette valeur permet de *remplacer* tous les espaces (tabulations, retour à la ligne, etc.) par des espaces "simples". En d'autres termes, chaque caractère d'espacement est remplacé par un espace #x20. Le résultat est donc du type prédéfini `xsd:normalizedString`.
- **Collapse** : cette valeur permet de *supprimer* les espaces en début et en fin de chaîne, de remplacer les tabulations et les retours à la ligne par un espace "simple" et de remplacer les espaces multiples par un espace "simple". 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`.

- R** La facette *xsd:whiteSpace* 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.

Dans cette partie du chapitre, nous avons vu les éléments simples et les différentes familles de types simples. Malheureusement, toutes ces connaissances ne sont pas suffisantes si l'on souhaite pouvoir décrire toutes les structures que les documents XML offrent.

Nous avons également vu comment déclarer un élément simple ainsi qu'un attribut. Cependant nous n'avons pas vu comment déclarer un attribut dans un élément. En effet, un élément qui possède un attribut n'est plus un élément simple. On parle alors d'**élément complexe**.

Exemple pratique 2.4.2

À l'aide de l'éditeur oXygen, déterminez les types simples suivants :

- Une heure comprise entre 2 h 30 du matin et 16 h 50 ;
- Un nombre réel en précision simple, supérieur ou égal à -3476,4 et strictement inférieur à 5 ;
- Une chaîne de quatre caractères ;
- Une liste limitée aux chaîne « jpg », « gif », « png » ;
- Un type de numéro composé de 13 chiffres.

J'espère que vous suivez toujours !!!...

file:/E:/Enseignement_2020_2021/Semestre_I/2_eme_Master_GL/2_TP/TP_2_XSD/ex_01.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   elementFormDefault="qualified">
4   <!--1ere question -->
5     <xs:simpleType name="typeHeure">
6       <xs:restriction base="xs:time">
7         <xs:minInclusive value="02:30:00"></xs:minInclusive>
8         <xs:maxInclusive value="16:50:00"></xs:maxInclusive>
9       </xs:restriction>
10    </xs:simpleType>
11
12   <!--2eme question -->
13     <xs:simpleType name="typeNombre">
14       <xs:restriction base="xs:float">
15         <xs:minInclusive value="-3476.4"/>
16         <xs:maxExclusive value="5"/>
17       </xs:restriction>
18     </xs:simpleType>
19
20   <!--3eme question -->
21     <xs:simpleType name="typeChaine4">
22       <xs:restriction base="xs:string">
23         <xs:minLength value="4"/>
24         <xs:maxLength value="4"/>
25       </xs:restriction>
26     </xs:simpleType>
27
28   <!--4eme question -->
29     <xs:simpleType name="typeImage">
30       <xs:restriction base="xs:string">
31         <xs:enumeration value="jpg"/>
32         <xs:enumeration value="gif"/>
33         <xs:enumeration value="png"/>
34       </xs:restriction>
35     </xs:simpleType>
36
37   <!--5eme question -->
38     <xs:simpleType name="typeNum">
39       <xs:restriction base="xs:string">
40         <xs:pattern value="[0-9]{13}"/>
41       </xs:restriction>
42     </xs:simpleType>
43 </xs:schema>
44
45
```

2.5 Les types complexes

Définition 2.5.1 Un type complexe est défini à l'aide de l'élément `complexType`. Il permet de définir des séquences d'éléments, des types de choix, ou un ensemble d'éléments.

Syntaxe :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="mon_nom">
    <xs:complexType>
      <!-- contenu ici -->
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Les définitions de types complexes vont de la simple définition d'un type d'élément contenant un attribut à la définition plus complexe de l'élément racine du document. De plus, les types complexes décrivent des contenus formés uniquement d'éléments ou des contenus mixtes. Ils peuvent être utilisés uniquement pour déclarer des éléments. Seuls les types complexes peuvent définir des attributs. Autrement dit, un élément complexe est un élément qui contient d'autres éléments ou des attributs. Bien évidemment les éléments contenus dans un élément peuvent également contenir des éléments ou des attributs [Har04][Ste03][All02].

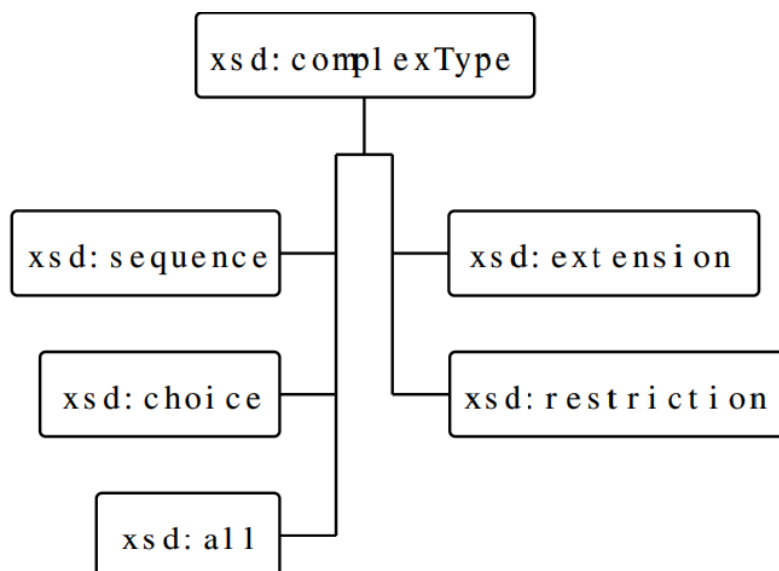


FIGURE 2.4 – Hiérarchie de construction de type complexe.

2.5.1 Types complexes à contenu simple

On appelle contenu simple, le contenu d'un élément complexe qui n'est composé que d'attributs et d'un texte de type simple. 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`.

Pour déclarer un élément complexe faisant référence à une balise contenant des attributs et du texte, voici la syntaxe à utiliser :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="mon_nom">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="mon_type">
          <xs:attribute name="mon_nom" type="mon_type" />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Dans le cas où la balise que l'on cherche à décrire contient plusieurs attributs, il convient tout simplement de les lister entre les balises `<xsd:extension/>`. Par exemple, dans le schéma suivant, le type `voiture` possède un attribut `marque` de type `xsd:string`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="voiture">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="marque" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Voici alors le code XML associé :

```
<?xml version="1.0" encoding="UTF-8"?>
<voiture marque="Renault">Clio campus</voiture>
```

2.5.2 Types complexes à contenu standard

Après les contenus simples, nous allons monter la barre d'un cran et s'attaquer aux contenus "standards". Ce que j'appelle contenu "standard", c'est le contenu d'un élément complexe qui n'est composé que d'autres éléments (simples ou complexes) ou uniquement d'attributs.

Voyons maintenant les deux cas de contenu "standard" :

Cas1 : balise contenant un ou plusieurs attributs

Dans ce cas, l'élément complexe ne contient que des attributs. Le Schéma XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="voiture">
    <xs:complexType>
      <xs:attribute name="marque" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Le code XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<voiture marque="Renault">Clio campus</voiture>
```

- R** Vous remarquez dans cet exemple qu'il en est de même pour les types complexes à contenu simple, mais gardez à l'esprit que nous n'utiliserons pas de mots-clés : `<xsd:extension/>` et `<xsd:simpleContent/>`.

Il n'y a, pour le moment, rien de compliqué. On se contente d'imbriquer une balise `<xsd:attribute/>` dans une balise `<xsd:complexType/>`. Si l'on tente de complexifier un petit peu les choses, nous allons nous rendre compte que, dans le fond, rien ne change. Prenons par exemple le cas d'une balise contenant plusieurs attributs, alors, le schéma XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="voiture">
    <xs:complexType>
      <xs:attribute name="marque" type="xs:string" />
      <xs:attribute name="modele" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Cas2 : balise contenant d'autres éléments

Il est maintenant temps de passer à la suite et de jeter un coup d'œil aux balises qui contiennent d'autres éléments et de les détailler plus en plus.

Une séquence d'éléments

Dans une DTD, il est possible d'indiquer qu'un élément doit contenir une suite de sous éléments dans un ordre déterminé. On peut bien sûr en faire autant avec un schéma XML. Pour cela, lors de la définition du type de l'élément, on utilise le mot clé : "sequence". En d'autres termes, plus précis, une séquence est utilisée lorsque l'on souhaite spécifier que les éléments contenus dans un type complexe doivent apparaître dans un ordre précis. Voici maintenant un exemple montrant comment se déclare une séquence au niveau d'un Schéma XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="personne">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="nom" type="xs:string"/>
        <xs:element name="prenom" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="sexe" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

La définition précédente indique l'élément <personne/> qui possède l'attribut "sexe", contient les balises <nom/> et <prenom/> dans cet ordre. Le code XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- valide -->
<personnes>
  <personne sexe="masculin">
    <nom>DJAFRI</nom>
    <prenom>LAOUNI</prenom>
  </personne>
  <!-- invalide => les balises nom et prenom sont inversées -->
  <personne sexe="masculin">
    <prenom>LAOUNI</prenom>
    <nom>DJAFRI</nom>
  </personne>
</personnes>
```

- R** L'élément "sequence" dans le schéma XML a le même effet que l'opérateur virgule (,) dans une DTD.

Une alternative d'éléments

On peut vouloir modifier la définition précédente si l'on souhaite indiquer un choix. Un choix est utilisé lorsque l'on veut spécifier qu'un élément contenu dans un type complexe soit choisi dans une liste prédéfinie. Pour ce faire, on a recours à l'élément ou bien le mot clé : "choice". Voici le même exemple que précédemment, mais nous allons maintenant voir comment une alternative d'éléments (choix d'éléments) est déclarée au niveau du schéma XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="personne">
    <xs:complexType>
      <xs:choice>
        <xs:element name="nom" type="xs:string"/>
        <xs:element name="prenom" type="xs:string"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

La définition précédente de l'élément `<personne/>` qui possède l'attribut "sexe", contient soit la balise `<nom/>`, soit la balise `<prenom/>`. Le code XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<personnes>
  <!-- valide: l'apparence de l'élément "nom" seulement. -->
  <personne sexe="masculin">
    <nom>DJAFRI</nom>
  </personne>
  <!-- valide: l'apparence de l'élément "prenom" seulement. -->
  <personne sexe="masculin">
    <prenom>LAOUNI</prenom>
  </personne>
  <!-- invalide => les 2 balises prenom et nom ne peuvent pas apparaître en même temps -->
  <personne sexe="masculin">
    <nom>DJAFRI</nom>
    <prenom>LAOUNI</prenom>
  </personne>
</personnes>
```

- R** Toute combinaison des éléments "choice" et "sequence" est autorisée, ce qui entraîne plus de souplesse dans la définition. Cela facilite la tâche par rapport aux déclarations des DTD. Cet élément a le même effet que l'opérateur " | " dans une DTD.

Un ensemble d'éléments

Un ensemble d'éléments est une innovation par rapport aux DTD. Il indique que les éléments enfants doivent apparaître une fois et dans n'importe quel ordre. Dans ce cas, on utilise l'élément "all". Voyons tout de suite un exemple montrant comment se déclare le type "all" au niveau d'un Schéma XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="personne">
    <xs:complexType>
      <xs:all>
        <xs:element name="nom" type="xs:string"/>
        <xs:element name="prenom" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Cet extrait signifie donc que la balise `<personne/>` contient les balises `<nom/>` et `<prenom/>` dans n'importe quel ordre.

Le code XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<personnes>
  <!-- valide -->
  <personne sexe="masculin">
    <nom>DJAFRI</nom>
    <prenom>LAOUNI</prenom>
  </personne>
  <!-- valide -->
  <personne sexe="masculin">
    <prenom>LAOUNI</prenom>
    <nom>DJAFRI</nom>
  </personne>
</personnes>
```

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

R L'élément "all" impose l'apparition obligatoires de tous les éléments composant le schéma XML, au moins une fois et dans n'importe quel ordre. Cet élément est inexistant en DTD.

CAS D'UN TYPE COMPLEXE ENCAPSULANT UN TYPE COMPLEXE

Avant de terminer cette partie, il nous reste un cas à voir : il s'agit d'un type complexe encapsulant également un type complexe. Prenons par exemple le document XML suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<personne>
  <identite>
    <nom>DJAFRI</nom>
    <prenom>LAOUNI</prenom>
  </identite>
</personne>
```

Ce document XML permet d'identifier une personne via son nom et son prénom. Voyons alors le Schéma XML qui définit notre document XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="personne">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="identite">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="nom" type="xs:string"/>
              <xs:element name="prenom" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

En soit, il n'y a rien de compliqué. Il convient juste de repérer que lorsque l'on place un élément complexe au sein d'un autre élément complexe, dans notre cas, il s'agit de l'identité d'une personne, il convient d'utiliser une séquence, un choix ou un type all.

D'une manière générale, pour déclarer un élément avec un contenu d'élément, vous définissez le type de l'élément à l'aide de `xs:complexType` et y incluez un modèle de contenu qui décrit les éléments enfants autorisés, la disposition autorisée de ces éléments et les règles de leurs occurrences. Vous créez le modèle de contenu en utilisant l'élément de schéma `xs:sequence`, `xs:choice` ou `xs:all`, ou une combinaison de ces éléments. Chacun de ces éléments de schéma ajoute un groupe de déclarations d'éléments au modèle de contenu. La signification du groupe dépend de l'élément de schéma que vous utilisez.

2.5.3 Types complexes à contenu mixte

Le contenu mixte comprend un élément complexe qui est composé d'attributs, d'éléments et de texte. Pour déclarer un élément complexe au contenu mixte, on utilise le mot clef `mixed`. Voici sa syntaxe :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="mon_nom">
    <xs:complexType mixed="true">
      <!-- liste des éléments -->
    </xs:complexType>
    <!-- liste des attributs -->
  </xs:element>
</xs:schema>
```

Exemple :

Écrire un schéma XML qui vous permet de valider une facture fictive dans laquelle on souhaite identifier l'acheteur et la somme qu'il doit payer ?

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="facture">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="acheteur" type="xs:string" />
        <xs:element name="somme" type="xs:int" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Le code XML associé est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<facture>
  <acheteur>Mahmoud</acheteur>, doit payer
  <somme>3000</somme>DA
</facture>
```

Dans le schéma XML ci-dessus, nous avons utilisé la balise `<xsd:sequence/>` pour encapsuler la liste des balises contenues. dans la balise `<facture/>`, mais vous pouvez bien évidemment adapter à votre cas de figure et choisir parmi les balises que nous avons vu dans ce chapitre, à savoir : `<xsd:sequence/>`, `<xsd:choice/>` et `<xsd:all/>`.

Rappel 2.5.1 Un élément complexe:

- est un élément qui contient d'autres éléments ou des attributs.
- est décrit grâce à la balise `<xsd:complexType/>`.
- a trois types de contenus possibles : les contenus simples, standards et mixtes.

* INDICATEURS D'OCCURRENCES (pour les éléments)

Dans une DTD un indicateur d'occurrence ne peut prendre que les valeurs 0 et 1 où l'infini. On peut forcer un élément "Elt" à être présent 100 fois, mais il faut alors écrire (Elt1, Elt2, ...Elt100) 100 fois. Schéma XML permet de déclarer directement une telle occurrence, car tout nombre entier non négatif peut être utilisé. Pour vous aider à bien comprendre cette notion, nous vous proposons d'étudier un morceau de schéma XML que nous avons déjà vu. Il s'agit de celui d'une personne qui possède un nom et un prénom. Voici maintenant le schéma correspondant :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="personne">
    <xs:sequence>
      <xs:element name="nom" type="xs:string"/>
      <xs:element name="prenom" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Comme nous l'avons vu précédemment, cet extrait signifie que la balise `<personne/>` contient les balises `<nom/>` et `<prenom/>` dans cet ordre.

La notion d'occurrence va nous permettre de préciser si les balises, dans le cas de notre exemple `<nom/>` et `<prenom/>`, peuvent apparaître plusieurs fois, voire pas du tout.

1- Le nombre minimum d'occurrences

Pour indiquer le nombre minimum d'occurrences d'un élément, on utilise l'attribut `minOccurs`.

2- Le nombre maximum d'occurrences

Pour indiquer le nombre maximum d'occurrences d'un élément, on utilise l'attribut `maxOccurs`.



- Lorsque le nombre d'occurrences n'est pas précisé, la balise doit apparaître une et une seule fois, sa valeur par défaut est 1.
- Pour déclarer qu'un élément peut-être présent un nombre illimité de fois, on utilise la valeur "unbounded".

Exemple :

Pour indiquer qu'un élément devra être utilisé 2, 3, 4 fois, ou $+\infty$, on écrira le code ci-dessous :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="personne">
    <xs:sequence>
      <xs:element name="nom" type="xs:string" minOccurs="3" maxOccurs="4"/>
      <xs:element name="prenom" type="xs:string" minOccurs="2" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

* CONTRAINTES D'OCCURRENCES (pour les attributs)

Tout comme dans une DTD, un attribut pour avoir un indicateur d'occurrence.

L'élément "**attribute**" d'un schéma peut posséder trois attributs optionnels : `use`, `default` et `fixed`. Des combinaisons de ces trois attributs permettent de paramétrer ce qui est acceptable ou non dans le fichier XML final (obligatoire, optionnel ou interdit).

L'attribut **use** peut prendre trois valeurs : `required`, `prohibited` ou `optional`. Les valeurs `optional` et `required` de l'attribut `use` sont donc équivalentes à `#IMPLIED` et `#REQUIRED` utilisés dans les déclarations d'attributs des DTD. Dans l'exemple suivant, les attributs `lang`, `xml:id` et `dummy` sont respectivement optionnel, obligatoire et interdit.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="MonType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="lang" type="xs:NMTOKEN" use="optional"/>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="dummy" type="xs:string" use="prohibited"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Il est possible de donner une valeur par défaut ou une valeur fixe à un attribut dont la valeur possible est "texte". La valeur de l'attribut `default` ou de l'attribut `fixed` de l'élément `xs: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 l'exemple suivant, la valeur par défaut de l'attribut lang1 est ar et de l'attribut lang2 sa valeur est fixée à la valeur en.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="MonType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="lang1" type="xs:NMTOKEN" default="ar"/>
        <xs:attribute name="lang2" type="xs:NMTOKEN" fixed="en"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Rappel 2.5.2

- NBR d'occurrences s'exprime grâce aux mots clés **minOccurs** et **maxOccurs**.
- Le mot clé **ref** permet de faire référence à des éléments dans le but de les réutiliser plusieurs fois au sein du Schéma XML.
- L'héritage permet de réutiliser des éléments d'un XSD pour en construire de nouveaux. Il existe 2 types d'héritages : l'héritage par : **restriction** et par **extension**.

Exemple pratique 2.5.3

À l'aide de l'éditeur oXygen, déterminez les types complexes suivants :

- Un type d'élément pouvant contenir du texte et un attribut optionnel « quantité » qui doit être un nombre entier ;
- Modifier le type précédent afin que «quantité» possède comme val par défaut «1» ;
- Un type d'élément pouvant contenir la suite des éléments « e1 », « e2 », « e3 », « e4 » sachant que :
 1. e1 : étant facultatif et présent au maximum une fois,
 2. e2 : pouvant être présent deux ou trois fois,
 3. e3, e4 : étant présents une seule et unique fois.
- Modifier le type précédent de manière à ce que l'on ait choix entre « e3 », « e4 ».

SOLUTION DE L'EXERCICE PRATIQUE :

file:/E:/Enseignement_2020_2021/Semestre_I/2_eme_Master_GL/2_TP/TP_2_XSD/Solution_XSD/ex_02.xsd

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2  elementFormDefault="qualified">
3
4  <!-- 1ere question -->
5      <xs:element name="type_element1" >
6          <xs:complexType mixed="true">
7              <xs:attribute name="quantité" use="optional" type="xs:integer"/>
8          </xs:complexType>
9      </xs:element>
10
11 <!-- 2eme question -->
12 <xs:element name="type_element2" >
13 <xs:complexType mixed="true">
14 <xs:attribute name="quantité" use="optional" type="xs:integer"
14 default="1"/>
15 </xs:complexType>
16 </xs:element>
17
18 <!-- 3eme question -->
19 <xs:element name="liste_element1">
20 <xs:complexType>
21 <xs:sequence>
22 <xs:element name="e1" minOccurs="0" maxOccurs="1"/>
23 <xs:element name="e2" minOccurs="2" maxOccurs="3"/>
24 <xs:element name="e3"/>
25 <xs:element name="e4"/>
26 </xs:sequence>
27 </xs:complexType>
28 </xs:element>
29
30 <!-- 4eme question -->
31 <xs:element name="liste_element2">
32 <xs:complexType>
33 <xs:sequence>
34 <xs:element name="e1" minOccurs="0" maxOccurs="1"/>
35 <xs:element name="e2" minOccurs="2" maxOccurs="3"/>
36 <xs:choice>
37 <xs:element name="e3"/>
38 <xs:element name="e4"/>
39 </xs:choice>
40 </xs:sequence>
41 </xs:complexType>
42 </xs:element>
43 </xs:schema>
44
```


2.6 Exercices résolus

```
<?xml version=" 1 . 0 " encoding="UTF-8"?>
<Univ_Ins>
  <Faculté>
  <Faculté>
    <Dep_Fac>dep_1</Dep_Fac>
    <Dep_Fac>dep_2</Dep_Fac>
  </Faculté>
  <Institut>
    <Dep_Ins>dep_3</Dep_Ins >
  </Institut>
</Univ_Ins>
<!-- doc1.xml -->
```

```
<?xml version=" 1 . 0 " encoding="UTF-8"?>
<Univ_Ins>
  <Faculté>
    <Dep_Fac>dep_4</Dep_Fac>
  </Faculté>
  <Institut>
    <Dep_Ins>dep_5</Dep_Ins>
    <Dep_Ins>dep_6</Dep_Ins>
  </Institut>
  <Institut>
    <Dep_Ins>dep_7</Dep_Ins>
</Univ_Ins>
<!-- doc2.xml -->
```

Exercice 2.1

Soient les deux documents XML, doc1.xml et doc2.xml ci-dessus :

- Définir un schéma XSD qui valide :
 1. Le document doc1.xml mais pas le document doc2.xml ?
 2. Le document doc2.xml mais pas le document doc1.xml ?
 3. Les deux documents doc1.xml et doc2.xml ?

Exercice 2.2

À l'aide de l'éditeur oXygen, produisez un document valide de type XML Schéma (XSD) qui respecte les contraintes suivantes :

- L'élément racine est l'élément « annuaire » ;
- L'annuaire est constitué d'au moins un élément « personne » ;
- Chaque élément personne possède un attribut obligatoire « nom » et un attribut facultatif « prénom », qui sont des chaînes de caractères.
- Chaque élément « personne » possède les éléments enfants suivants :
 - Un élément « dateAnniversaire » qui est de type date ;
 - Un élément « idéesCadeaux » qui est chaîne de caractères ;
 - Un élément « téléphone » qui possède :
 - Un attribut « numéro » qui est une liste de nombres entiers strictement positifs et strictement inférieurs à 100 ;
 - Un élément enfant « heuresDAppel » qui doit posséder un attribut « heureDébut » et un attribut « heureFin », les deux attributs sont obligatoires.

2.7 Solutions

2.7.1 Solution de l'exercice 2.1

* XSD qui valide le document doc1.xml mais pas le document doc2.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="type_Univ_Ins" mixed="true">
    <xs:sequence>
      <xs:element name="Faculté" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="Institut" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="type_Faculté" mixed="true">
    <xs:sequence>
      <xs:element name="Dep_Fac" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="type_Institut" mixed="true">
    <xs:sequence>
      <xs:element name="Dep_Ins" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Univ_Ins" type="type_Univ_Ins"/>
  <xs:element name="Faculté" type="type_Faculté"/>
  <xs:element name="Institut" type="type_Institut"/>
</xs:schema>
```

* XSD qui valide le document doc2.xml mais pas le document doc1.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="type_Univ_Ins" mixed="true">
    <xs:sequence>
      <xs:element name="Faculté" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="Institut" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="Dep_Ins" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="type_Faculté" mixed="true">
    <xs:sequence>
      <xs:element name="Dep_Fac" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="type_Institut" mixed="true">
    <xs:sequence>
      <xs:element name="Dep_Ins" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Univ_Ins" type="type_Univ_Ins"/>
  <xs:element name="Faculté" type="type_Faculté"/>
  <xs:element name="Institut" type="type_Institut"/>
</xs:schema>
```

* XSD qui valide les deux documents doc1.xml et doc2.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:complexType name="type_Univ_Ins" mixed="true">
    <xs:sequence>
      <xs:element name="Faculté" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="Institut" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="Dep_Ins" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="type_Faculté" mixed="true">
    <xs:sequence>
      <xs:element name="Dep_Fac" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="type_Institut" mixed="true">
    <xs:sequence>
      <xs:element name="Dep_Ins" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Univ_Ins" type="type_Univ_Ins"/>
  <xs:element name="Faculté" type="type_Faculté"/>
  <xs:element name="Institut" type="type_Institut"/>
</xs:schema>
```

2.7.2 Solution de l'exercice 2.2

 file:/E:/Enseignement_2020_2021/Semestre_l/2_eme_Master_GL/2_TP/TP_2_XSD/Solution_XSD/ex_04.xsd

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4      <!-- declaration des types simples -->
5
6
7  <!-- declaration les "ref" des elements des types simples -->
8
9      <xs:element name="date_anniversaire" type="xs:date"/>
10     <xs:element name="idees_cadeaux" type="xs:string"/>
11 <!-- declaration les "ref" des elements des types complexes -->
12 <xs:element name="annuaire" type="type_annuaire"></xs:element>
13 <xs:element name="personne" type="type_personne"/>
14 <xs:element name="telephone" type="type_telephone"></xs:element>
15 <xs:element name="heure_appel" type="type_heure_appel"></xs:element>
16
17
18 <!--Un attribut «numéro» qui est une liste de nombres entiers:>0 et <100 -->
19
20 <xs:simpleType name="numero">
21     <xs:restriction base="xs:byte">
22         <xs:minInclusive value="0"/>
23         <xs:maxInclusive value="99"/>
24     </xs:restriction>
25 </xs:simpleType>
26 <xs:simpleType name="liste_numero">
27     <xs:list itemType="numero"/>
28 </xs:simpleType>
29
30
31
32     <!-- declaration des types complexes -->
33
34
35 <!-- L'annuaire est constitué d'au mois un élément « personne » -->
36
37 <xs:complexType name="type_annuaire">
38     <xs:sequence>
39         <xs:element ref="personne" minOccurs="1"/>
40     </xs:sequence>
41 </xs:complexType>
42
43
  
```

file:/E:/Enseignement_2020_2021/Semestre_I/2_eme_Master_GL/2_TP/TP_2_XSD/Solution_XSD/ex_04.xsd

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4
5
6
7         <!-- Suite de l'exercice 2.2 -->
8
9
10
11
12 <!-- Chaque élément « personne » possède les éléments enfants suivants: -->
13
14     <xs:complexType name="type_personne" mixed="true">
15         <xs:all>
16             <xs:element ref="date_anniversaire"/>
17             <xs:element ref="idees_cadeaux"/>
18         </xs:all>
19         <xs:attribute name="nom" use="required"/>
20         <xs:attribute name="prénom" use="optional"/>
21     </xs:complexType>
22
23
24 <!-- Un élément « téléphone » qui possède : -->
25
26     <xs:complexType name="type_telephone">
27         <xs:sequence>
28             <xs:element ref="heure_appel"/>
29         </xs:sequence>
30         <xs:attribute name="Numero" type="numero"/>
31     </xs:complexType>
32
33
34 <!-- Un élément enfant « heuresDAppel » -->
35
36     <xs:complexType name="type_heure_appel">
37         <xs:attribute name="heure_debut" type="xs:time" use="required"/>
38         <xs:attribute name="heure_fin" type="xs:time" use="required"/>
39     </xs:complexType>
40 </xs:schema>
```

2.8 Exercices ouverts

Exercice 2.3

À l'aide de l'éditeur oXygen, produisez un document valide de type XML Schéma (XSD) pour une bibliographie.

- Cette bibliographie contient des livres ;
- Chaque livre possède : un titre, un ou plusieurs auteurs, un ou plusieurs tomes et une date de publication ;
- Chaque auteur doit porter un identifiant, ainsi qu'un nom et un prénom, il est également soit un homme, soit une femme ;
- Pour chaque tome leur nombre de pages.

Exercice 2.4

À l'aide de l'éditeur oXygen, définissez un schéma XML (avec l'utilisation de l'option « ref », est obligatoire) qui permet de produire un document XML pour un système solaire comme suit :

- L'élément racine est l'élément « système_solaire » ;
- Le système_solaire est constitué d'un ou plusieurs éléments « étoile » et d'au moins un élément « planète » ;
- Chaque élément « étoile » possède les éléments enfants suivants : « nom », « type_spectral » qui sont de type chaîne de caractères, et « âge » qui est de type date ;
- Un élément « planète » qui possède un attribut obligatoire « type » qui est une chaîne de caractères, et les éléments enfants suivants : « masse » et « distance » qui sont de type double, et « nom » qui est de type chaîne de caractères :
 - Un élément « masse » possède un attribut obligatoire « unit » qui est de type chaîne de caractères ;
 - Un élément « distance » possède un attribut obligatoire « unit » qui est de type chaîne de caractères ;
 - L'élément « distance » limité à des valeurs supérieurs à 0.4 UA et strictement inférieurs à 90 UA ;

UA : Unité Astronomique

2.9 Conclusion

Un schéma ou une description réelle peut-être très concis mais précis. L'art d'un bon caricaturiste est d'utiliser un minimum de traits tout en produisant un dessin qui ressemble vraiment à l'objet dessiné.

Les langages de schémas ont la réputation d'être un sujet complexe et cela, en soi, suffit à nous faire nous demander si le nom « schéma » est bien choisi. La raison en est que les schémas XML ne sont pas des schémas au sens commun du terme : leur objectif principal n'est pas de décrire des documents XML, mais de les « valider ». Tandis que, le format schéma XML offre plus de possibilités que les DTD pour contraindre les éléments et attributs autorisés dans un format XML données. Ces possibilités supplémentaires sont présentes au prix d'un format un peu plus difficile à analyser à l'oeil, car fortement modularisé. Un schéma XML permet également d'affecter un type aux données d'un document XML. Il facilite l'appel à des ensembles d'éléments et d'attributs définis dans plusieurs fichiers différents, et parfois épars sur le réseau. Le format rend possible la mise au point d'une documentation détaillée incluses "naturellement" dans le corps du fichier, au fil des décorations. Il offre aussi une grande souplesse dans la définition des combinaisons d'éléments possibles.

Enfin en tant que document XML lui-même, un fichier de schéma XML est un fichier qui peut être traité et manipulé automatiquement par les mêmes outils que ceux qui sont utilisés pour le traitement et la manipulation des fichiers qu'il définit : c'est une spécification du parc des outils, en même temps qu'un élargissement de l'éventail des applications possibles.



3. XPath

3.1 Introduction

XPath est une recommandation officielle du World Wide Web Consortium (W3C). XPath est une technologie qui permet d'extraire des informations (éléments, attributs, commentaires, etc...) d'un document XML. XPath fournit également divers types d'expressions qui peuvent être utilisées pour rechercher des informations pertinentes dans le document XML. XPath définit les parties d'un document XML comme les éléments, les attributs, les textes, l'espace de noms, les instructions de traitement, les commentaires et les nœuds de document XML [Bec22] [Hon21].

Dans ce chapitre, nous verrons les points énumérés ci-dessous :

1. **Les chemins de localisation** : XPath fournit des expressions de chemin puissantes pour sélectionner des nœuds ou une liste de nœuds dans les documents XML.
2. **Les Fonctions** : XPath fournit une riche bibliothèque de fonctions standards pour la manipulation de valeurs de chaînes, de valeurs numériques, de comparaison de date et d'heure, de manipulation de nœud et de QName, de manipulation de séquence, de valeurs booléennes, etc.

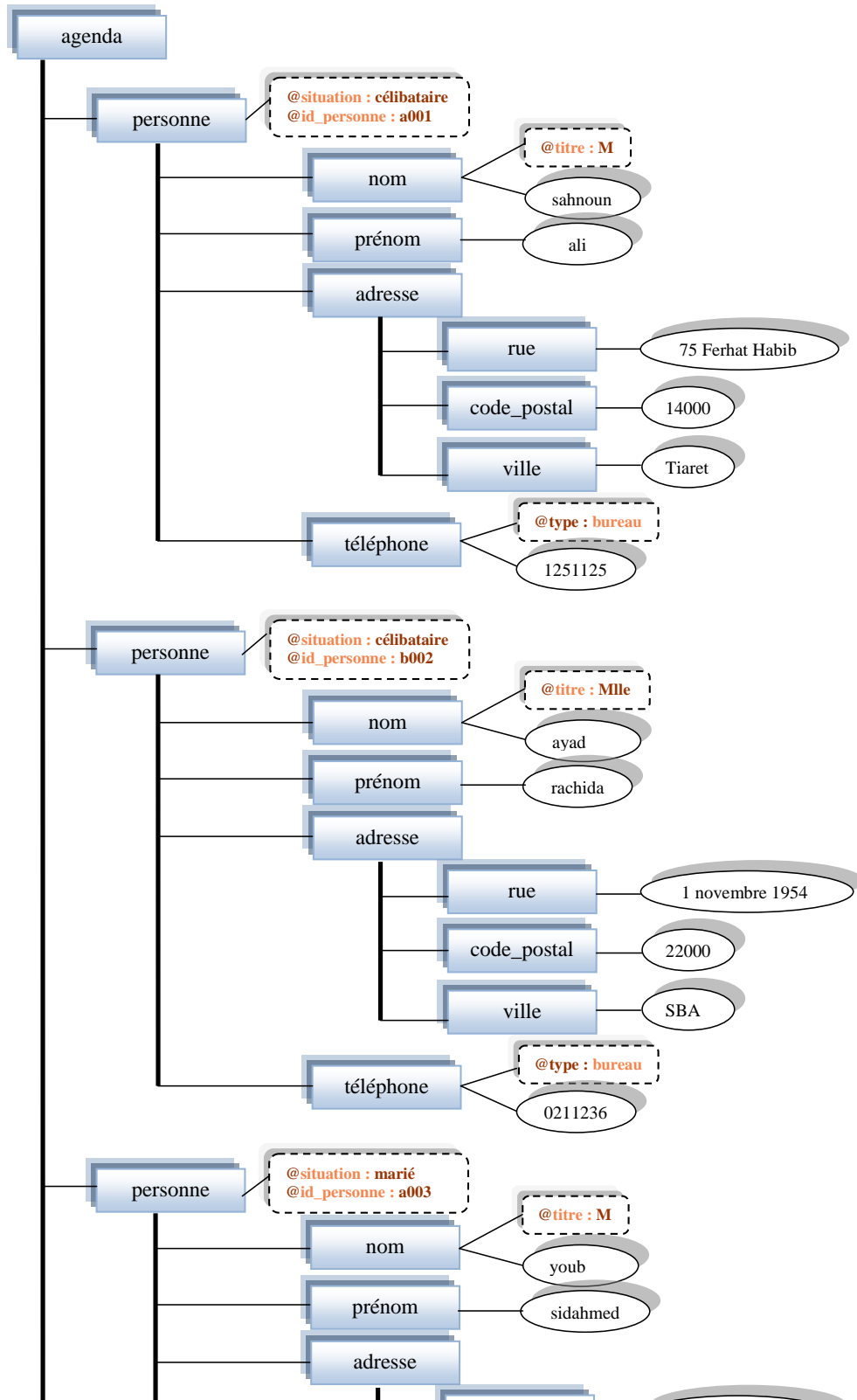
3.2 Les chemins de localisation

Avant d'étudier de manière plus approfondie comment écrire des expressions XPath, il convient de revenir sur quelques notions de vocabulaire qui seront indispensables pour bien comprendre la suite du XPath. Afin d'illustrer les concepts que nous verrons plus tard, nous vous demandons de produire le document XML de l'exercice pratique 2.2.2 que nous avons déjà vu dans le Chapitre 2 : *Validation de documents XML*.

file:///.../Semestre_I/2_eme_Master_GL/2_TP/TP_1_XML et DTD/TP_1_XML_2021/ex_07_rep.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <agenda>
3   <personne situation="célibataire" id_personne="a001">
4     <nom titre="M">sahnoun</nom>
5     <prénom>ali</prénom>
6     <adresse>
7       <rue>75 Ferhat Habib</rue>
8       <code_postal>14000</code_postal>
9       <ville>Tiaret</ville>
10    </adresse>
11    <téléphone Type="bureau">1251125</téléphone>
12  </personne>
13  <personne situation="célibataire" id_personne="b002">
14    <nom titre="Mlle">ayad</nom>
15    <prénom>rachida</prénom>
16    <adresse>
17      <rue>1 novembre 1954</rue>
18      <code_postal>22000</code_postal>
19      <ville>SBA</ville>
20    </adresse>
21    <téléphone Type="bureau">021123654</téléphone>
22  </personne>
23  <personne situation="marié" id_personne="a003">
24    <nom titre="M">youb</nom>
25    <prénom>sidahmed</prénom>
26    <adresse>
27      <rue>10 Amir AEK</rue>
28      <code_postal>31000</code_postal>
29      <ville>oran</ville>
30    </adresse>
31    <téléphone Type="perso">001255</téléphone>
32  </personne>
33  <personne situation="célibataire" id_personne="a004">
34    <nom titre="Mlle">salah</nom>
35    <prénom>aicha</prénom>
36    <adresse>
37      <rue>05 juillet 1962</rue>
38      <code_postal>16000</code_postal>
39      <ville>Alger</ville>
40    </adresse>
41    <téléphone Type="portable">897422258</téléphone>
42  </personne>
43 </agenda>
44
```

Reprenons également une illustration de son arbre :



Si on veut récupérer par exemple le numéro de téléphone du bureau, alors voici le chemin à parcourir :

- Étape 1 : nœud "agenda".
- Étape 2 : descendre au nœud enfant "personne".
- Étape 3 : descendre au nœud enfant "telephone" dont l'attribut Type="bureau".

Sans rentrer dans les détails, l'expression XPath correspondante ressemblera à quelque chose comme ça : /Étape 1/Étape 2/Étape 3

Restez toujours avec nous, dans le langage d'expression XPath, il est possible d'exprimer vos chemins de deux manières :

1. Un chemin relatif .
2. Un chemin absolu.

- **Un chemin relatif** est défini comme un type de XPath utilisé pour rechercher un nœud d'élément n'importe où sur la page Web. Il est spécifié par la notation à double barre oblique (//) qui commence au milieu de la structure DOM, et il n'est pas nécessaire d'ajouter un long XPath. Pas besoin de démarrer à partir du nœud racine, on peut commencer par le nœud que nous avons sélectionné. Le chemin relatif peut également avoir des enfants, et ce chemin relatif est préféré en raison de l'absence d'accès direct depuis le nœud racine.

Si on veut récupérer le numéro de téléphone en utilisant le chemin relatif, la syntaxe correspondante à notre exemple ressemblera alors à ça : //téléphone

- **Un chemin absolu** est le chemin complet à partir de la racine à l'élément que nous voulons identifier.

Si on veut par exemple récupérer le numéro de téléphone en utilisant le chemin absolu, la syntaxe correspondante à notre exemple ressemblera alors à ça : /agenda/personne/téléphone

3.2.1 Les sélecteurs de nœuds

Les sélecteurs de nœuds sont de la forme : `axe::filtre[condition1][condition2]...`

1. **Les axes** : indiquent un sens de recherche,
2. **Les filtres (Tests de nœuds)** : sélectionnent un nœud ou un type de nœud,
3. **Les conditions (prédicats)** : sélectionnent sur le contenu, on peut utiliser plusieurs prédicats (facultatif).

1. Les axes :

Les axes nous permettent de définir la direction de recherche pour que la première sélection soit faite en fonction de 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. Par exemple, si l'on souhaite se diriger vers un nœud enfant ou au contraire remonter vers un nœud parent voir un ancêtre.

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 indiqués dans la figure ci-dessous :

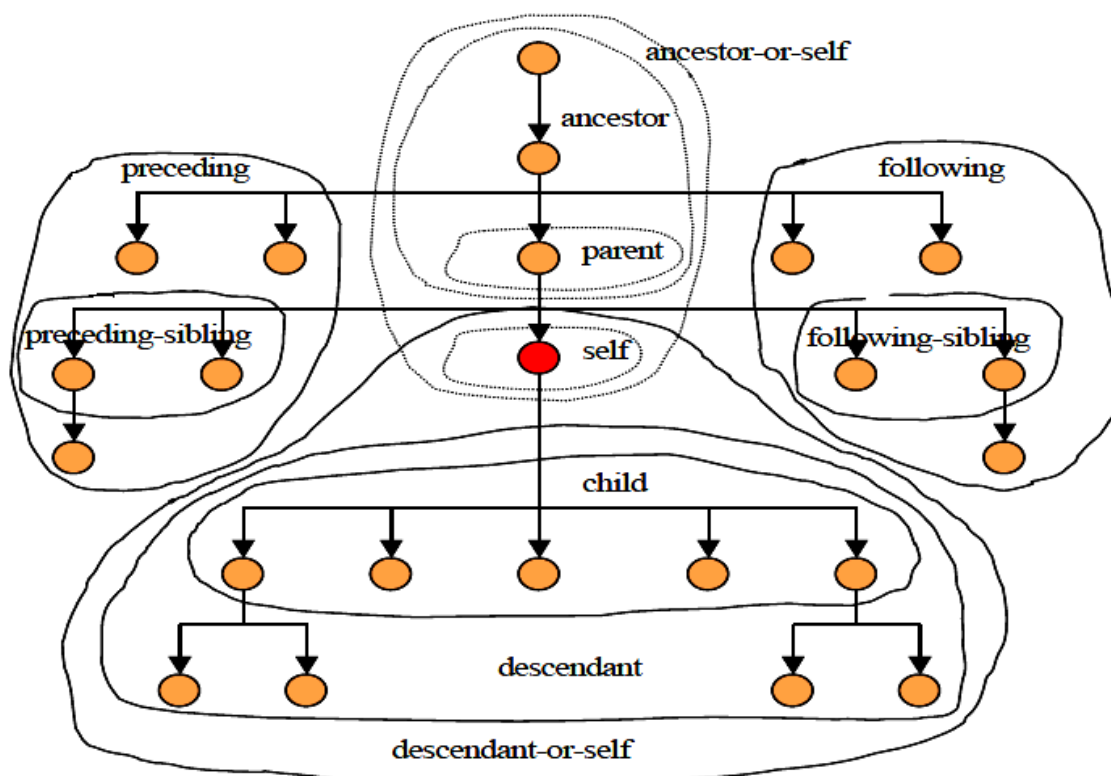


FIGURE 3.1 – Les axes de recherche en XPath.

- L'axe par défaut est l'axe `child` qui sélectionne les enfants du nœud courant. On écrit `node()` pour sélectionner tous les enfants du nœud courant.
- L'axe `self` est le nœud lui-même, il permet d'orienter la recherche vers le nœud courant (contextuel).
- L'axe `ancestor` oriente la recherche vers les ancêtres du nœud courant. On écrit `ancestor::*` pour sélectionner tous les ancêtres stricts du nœud courant.
- L'axe `ancestor-or-self` oriente la recherche vers le nœud courant et ses ancêtres.

- L'axe descendant oriente la recherche vers les descendants du nœud courant.
- L'axe descendant-or-self oriente la recherche vers le nœud courant (lui-même) et ses descendants.
- L'axe following oriente la recherche vers tous les nœuds gauches placés après le nœud courant.
- L'axe following-sibling oriente la recherche vers les nœuds frères gauche (enfants du même parent).
- L'axe parent oriente la recherche vers le père du nœud courant.
- L'axe preceding oriente la recherche vers tous les nœuds précédents (placés à droite) du nœud courant.
- L'axe preceding-sibling oriente la recherche vers les frères précédents (placés à droite) du nœud courant (enfants du même parent).
- L'axe attribute oriente la recherche vers les attributs du nœud courant.

2. Les filtres (Tests de nœuds) :

Le filtre va nous permettre de sélectionner dans une liste les nœuds qui satisfont une condition en indiquant explicitement le nom d'un nœud ou le type de nœud dont les informations nous intéressent.

- Filtrer les nœuds nommés => (*) : l'étoile permet d'orienter la recherche vers tous les nœuds. les nœuds de l'axe qui ont un nom (attribut ou élément), on écrit par exemple : //personne / attribute :: *.
- Filtrer les nœuds => node() : oriente la recherche vers tous les types de nœuds (éléments, commentaires, attributs, etc) sauf la racine, on écrit : //personne/nom/node() .
- Filtrer les textes => text() : permet de sélectionner tous les nœuds de type texte, on écrit : (//text()).
- Filtrer les commentaires => comment() : permet de sélectionner tous les nœuds de type commentaire, on écrit : //comment() .

3. Les prédicats :

Les prédicats sont facultatifs, leur nombre est illimité, ils agissent comme un filtre et vont nous permettre de gagner en précision dans nos recherches. Un prédicat se présente comme une expression entre des crochets '[' et ']' placée après la liste à filtrer (les nœuds). Ainsi, grâce aux prédicats, il sera par exemple possible de sélectionner les informations à une position précise.

- Condition d'existence: permet de sélectionner un nœud en fonction de son contenu : `/adresses/personne[type[attribute::id]]`
- Condition de position: permet de sélectionner un nœud en fonction de sa position.

Exemple : sélectionner les trois premiers éléments enfants du nœud courant.

Code XPath : `child::*[position()<4]` ou `*[position()<4]`

- Les conditions logiques: dans ce cas, les relations portent sur deux sous-expressions XPATH : **exp1 relation exp2**. Les relations possibles sont : =, !=, <, <=, >, >=, or, and, Not.

Exemple : `personne [prénom="med"]` : oriente la recherche vers les personnes qui ont un prénom med.

Nous vous présentons ci-dessous quelques instructions de base :

- `child::*` ou `*` oriente la recherche vers tous les éléments qui sont enfants du nœud courant.
- `attribute::*` ou `@*` sélectionne tous les attributs du nœud courant.
- `attribute::id` ou `@id` sélectionne un attribut id du nœud courant.
- `child::node()` ou `node()` oriente la recherche vers tous les enfants du nœud courant.
- `child::text` ou `text` oriente la recherche vers tous les éléments de nom `text` qui sont enfants du nœud courant.
- `child::text()` ou `text()` oriente la recherche vers tous les nœuds textuels qui sont enfants du nœud courant.
- `descendant::comment()` sélectionne tous les commentaires qui sont descendants du nœud courant, c'est-à-dire contenus dans le nœud courant.
- `following::processing-instruction()` sélectionne toutes les instructions de traitement qui suivent le nœud courant.
- `child::processing-instruction('xml-styleSheet')` sélectionne toutes les instructions de traitement de nom `xml-styleSheet` qui sont enfants du nœud courant.
- `tokenize` affiche un seul mot.
- `lower-case` affiche les mots minuscules.
- `upper-case` affiche les mots majuscules.

3.3 Les Fonctions

Cette section vous fournira une introduction aux fonctions XPath permettant de récupérer les propriétés d'un nœud, de sorte qu'elles renvoient un objet de l'un des quatre types différents :

3.3.1 Fonctions sur les nombres

XPath fournit quelques fonctions simples pour manipuler les nombres. Nous décrivons brièvement chacun d'eux avec un exemple simple illustrant leur utilisation.

- La fonction `ceiling()` renvoie le plus petit entier, qui est supérieur à l'argument de la fonction `plafond()`. Il s'agit essentiellement d'une fonction « arrondir vers le haut ».
- La fonction `floor()` renvoie le plus grand entier qui n'est pas supérieur à l'argument de la fonction `floor()`. Il s'agit essentiellement d'une fonction d'« arrondi vers le bas ».
- La fonction `number()`, cette fonction permet de convertir son argument en nombre.
- La fonction `round()`, cette fonction permet d'arrondir un nombre réel à l'entier le plus proche.
- La fonction `sum()` renvoie la somme de l'addition de la valeur de chaîne de chaque nœud dans un ensemble de nœuds, après conversion en nombre.

3.3.2 Fonctions sur les booléens

Le langage d'expression XPath fournit cinq fonctions booléennes.

- La fonction `boolean()` permet de convertir son argument en une valeur booléenne.
- La fonction `false()` : cette fonction renvoie la valeur booléenne "false" quel que soit son argument.
- La fonction `true()` renvoie la valeur booléenne "true".
- La fonction `not()` renvoie la valeur `false` si son argument est vrai, et renvoie la valeur `true` sinon.
- La fonction `lang()` renvoie `true` ou `false`, selon que la langue du nœud de contexte spécifiée par les attributs `xml:lang` est la même ou est une sous-langue de la langue spécifiée par la chaîne d'argument.

3.3.3 Fonctions sur les nœuds

XPath fournit un certain nombre de fonctions qui nous permettent d'affiner ou de filtrer une sélection dans un ensemble de nœuds, puis de renvoyer un autre ensemble de nœuds. Les fonctions sur les nœuds XPath sont souvent utilisées dans les prédicats des étapes de localisation. Examinons brièvement chacune des fonctions .

- La fonction `count()` renvoie le nombre de nœuds dans un ensemble de nœuds d'argument. Ainsi, avec l'exemple de l'exercice pratique 2.2.2 (Chapitre 2), nous pouvons utiliser la fonction `count()` pour calculer le nombre d'éléments `<personne>` contenus dans le document source ou bien l'élément racine `<agenda>`.
- La fonction `id()` sélectionne les nœuds en fonction de leur possession d'un attribut ID unique. Pour être désigné comme un attribut d'ID, l'attribut doit être déclaré comme tel dans une définition de type de document d'accompagnement.
- La fonction `last()` renvoie un nombre égal à la taille du contexte dans le contexte où l'évaluation a lieu. La fonction `last()` est particulièrement utile lorsque nous ne savons pas, ou ne souhaitons pas prendre le temps de découvrir, quelle est la taille du contexte. Quelle que soit la taille du contexte, la fonction `last()` renverra le nœud en dernière position.
- La fonction `local-name()` renvoie la partie locale du nom étendu du nœud dans l'ensemble de nœuds d'argument qui est le premier dans l'ordre du document.
- La fonction `name()` renvoie une chaîne contenant un QName représentant le nom étendu du nœud dans l'ensemble de nœuds d'argument qui est le premier dans l'ordre du document XML.
- La fonction `namespace-uri()` renvoie l'URI de l'espace de noms du ou des nœuds dans l'ensemble de nœuds d'argument.
- La fonction `position()` renvoie un nombre qui reflète la position contextuelle du nœud contextuel. Nous pouvons utiliser la fonction `position()` pour sélectionner uniquement certains nœuds ou, nous pouvons afficher explicitement la valeur renvoyée par la fonction `position()`. La fonction `position()` permet également d'utiliser les opérateurs `<>` (supérieur à) ou `<<` (inférieur à).

3.3.4 Fonctions sur les chaînes de caractères

XPath fournit quelques fonctions simples pour manipuler les chaînes de caractères. Maintenant, nous décrirons brièvement chacun d'eux avec un exemple simple illustrant leur utilisation.

- La fonction `concat(chaîne1, ..., chaîneN)` permet de concaténer les arguments (les chaînes) qui lui sont passés.
- La fonction `substring(string, number, number?)` retourne `string`: Elle retourne la sous-chaîne du premier argument commençant à la position spécifiée par le second argument et de longueur spécifiée par le troisième argument. Si le troisième argument n'est pas spécifié, la fonction retourne la sous-chaîne comprise entre la position de départ spécifiée et la fin de la chaîne de caractères initiale.
- La fonction `string-length(chaîne)` renvoie la longueur d'une chaîne.
- La fonction `normalize-space(chaîne)` renvoie une version normalisée (suppression des blancs au début et à la fin et remplacement de toute suite de blancs par un seul). `normalize-space(' AB CD E ')` = `'AB CD E'`

- La fonction `translate(chaîne1, chaîne2, chaîne3)` renvoie une copie de `chaîne1` dans laquelle les caractères présents dans `chaîne2` sont remplacés par les caractères de même position dans `chaîne3`. `translate('ABCD', 'AC', 'ac')` = `'aBcD'`
- La fonction `starts-with(chaîne1, chaîne2)` vraie si et seulement si `chaîne1` débute par `chaîne2`.
- La fonction `contains(chaîne1, chaîne2)` vraie si et seulement si `chaîne1` contient `chaîne2`.

Exemple pratique 3.3.1


- A partir de l'exercice pratique 2.2.2 Chapitre 2), en considérant que le nœud contextuel est «agenda», donnez les expressions (XPath) permettant de localiser les nœuds suivants :
 1. Tous les nœuds «personne» présents dans le document ;
 2. Tous les nœuds «nom» présents dans le document ;
 3. Tous les nœuds «téléphone» correspondants à des type du téléphone «bureau» ;
 4. Tous les attributs «titre» ;
 5. Le nom de la première personne qui est du titre Mlle ;
 6. Afficher tous les titres de noms en minuscules ;
 7. Afficher tous les titres de noms en majuscules ;
 8. Afficher uniquement le premier mot du prénom de la deuxième personne.

SOLUTION DE L'EXERCICE PRATIQUE :

1. `//personne`
2. `//nom`
3. `/agenda/personne/téléphone[@type="bureau"] or //téléphone[@type="bureau"]`.
4. `//nom/@titre`
5. `(//nom[@titre="Mlle"])[1]`
6. `//titre/lower-case(text())`
7. `//titre/upper-case(text())`
8. `tokenize(/agenda/personne[2]/prénom, ' ')[1]`

3.4 Conclusion

Dans ce chapitre, nous avons donné une vue générale de ce qu'est XPath et de ce qu'il fait. Dans le chapitre suivant, nous continuerons à détailler les aspects et les relations entre XPath et XSLT. Le XML Path Language (XPath) est conçu pour vous aider, ainsi qu'un processeur XSLT, à trouver votre chemin dans les documents XML, afin que vous puissiez sélectionner une partie définie d'un document XML à diverses fins. Les exemples peuvent être pour l'affichage en XML à l'aide de feuilles de style en cascade, la conversion de parties sélectionnées du XML en HTML pour l'affichage, la transformation en un autre type de document XML dans un scénario d'échange de données B2B, ou pour afficher le résultat d'une requête du document XML. XPath est d'une importance cruciale pour toutes les transformations XSLT. Si vous ne pouvez pas définir et localiser un élément particulier, ou, plus précisément, le nœud qui le représente, alors vous ne pouvez pas le traiter correctement pour produire la sortie souhaitée. De même, la bibliothèque de fonctions XPath prend en charge certaines fonctions numériques, de chaîne et booléennes ainsi que les fonctions d'ensemble de nœuds qui sont particulièrement pertinentes pour l'utilisation de XPath comme outil de navigation. Ces fonctions sont conçues pour vous aider à sélectionner les nœuds à traiter d'une manière particulière.



4. XSLT

4.1 Introduction

XSLT, eXtensible Stylesheet Language Transformations, est un langage déclaratif qui utilise des feuilles de style (parfois appelées feuilles de transformation) pour décrire une transformation d'un document source XML en un document résultat, qui peut-être un document XML structuré différemment, un document HTML, un document texte ou dans un autre format. Une transformation XSLT ne modifie ni ne transforme le document source, elle crée simplement un nouveau document de résultat. Dans les règles d'une feuille de style XSLT, vous déclarez ce que vous voulez que le processeur XSLT produise. C'est, en général, la façon dont le processeur XSLT traite un document source pour obtenir la sortie spécifiée. Afin de mieux comprendre XSLT, nous devons suivre ces étapes : nous avons d'abord une feuille de style XSLT contient des règles décrivant des transformations. Ces règles sont ensuite appliquées à un document source XML pour obtenir un nouveau document XML résultat (transformation). 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.

Plus précisément, il ne s'agit pas d'un « document » source qu'un processeur XSLT traite. Il traite en fait un arbre source conservé en mémoire et crée un arbre de sortie en mémoire basé sur les nœuds contenus dans l'arbre source et les règles de modèle contenues dans la feuille de style XSLT. L'arborescence source est la représentation hiérarchique basée sur des nœuds en mémoire d'un document XML source. L'arborescence de sortie peut, mais pas nécessairement, être ultérieurement enregistrée sous forme de fichier au format XML, HTML ou dans un autre format ou peut être traitée davantage sans jamais avoir été sérialisée (c'est-à-dire retransformée en syntaxe XML conventionnelle).

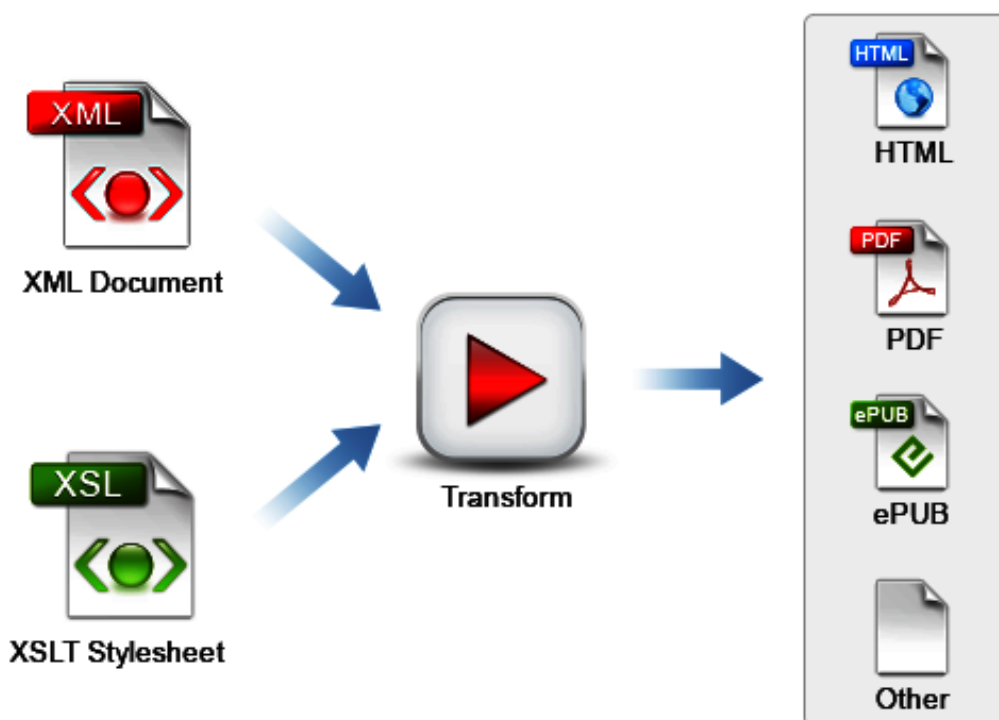


FIGURE 4.1 – Transformation XSLT

R Pourquoi les développeurs Web utilisent XSLT ?

- Indépendant de la programmation. Les transformations sont écrites dans un fichier xsl séparé qui est à nouveau un document XML.
- La sortie peut être modifiée en modifiant simplement les transformations dans le fichier xsl. Pas besoin de changer de code. Ainsi, les concepteurs Web peuvent modifier la feuille de style et voir rapidement le changement dans la sortie.

Comme d'habitude, nous allons débiter en douceur. Nous vous proposons de débiter notre apprentissage du XSLT en découvrant ensemble la structure d'un document XSLT [Tid09][Ama09], ainsi que les éléments indispensables de XSLT. Ce chapitre vise également à découvrir la relation forte entre **XSLT** et la technologie **XPath** du chapitre précédent et avec la technologie **XQuery** qui va nous occuper dans le prochain chapitre.

4.2 Structure d'un document XSLT

Comme ce fut le cas pour les DTD et les schémas XML, on s'habitue à écrire les documents XSLT dans un fichier distinct du document XML dont les données seront converties en d'autres documents dans d'autres formats. L'extension portée par les documents XSLT est ".xsl".

```
1 <?xml version="1.0" encoding="UTF-8"?> <!-- le prologue -->
2
3 <!-- l'élément racine -->
4 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5     xmlns:xs="http://www.w3.org/2001/XMLSchema"
5     exclude-result-prefixes="xs" version="2.0">
6
7     <!-- l'élément output -->
8     <xsl:output
9         method="html"
10        encoding="UTF-8"
11        doctype-public="-//W3C//DTD HTML 4.01//EN"
12        doctype-system="http://www.w3.org/TR/html4/strict.dtd"
13        indent="yes" />
14
15     <!-- reste du document XSLT -->
16
17 </xsl:stylesheet>
```

Document sans erreur U+000A 7:30 Mod

FIGURE 4.2 – Structure d'un document XSLT

4.2.1 Le prologue

Le prologue à ajouter en haut de la feuille de style XSL créée avec le texte XSL qui définit une règle d'autorisation booléenne. Si un prologue de feuille de style est spécifié, ce prologue est importé dans la feuille de style vide. Si aucun prologue n'est spécifié, une valeur de prologue par défaut est utilisée à la place. Tous les attributs du prologue requis sont spécifiés par défaut dans les entrées de ce dernier (voir la figure 4.2).

- R** Si l'un des attributs de prologue requis est modifié ou omis dans l'entrée, le client d'autorisation ne démarre pas et renvoie une erreur.

4.2.2 L'élément racine

L'élément racine de XSLT est `<xsl:stylesheet>`, ou son synonyme l'élément `<xsl:transform>`. Il forme le squelette de toutes les feuilles de style XSLT. Pour être conforme à la spécification XSLT, votre feuille de style doit toujours commencer par cet élément, codé exactement comme indiqué dans la figure 4.2.

Comme vous pouvez le constater sur la figure 4.2, il y'a des attributs dans cet élément racine. L'attribut `"xmlns:xsl"` nous permet de déclarer un espace de noms, et l'attribut `"version"` qui indique le numéro de version que l'on souhaite utiliser. Toutefois, la première version de XSLT reste encore aujourd'hui majoritairement utilisée.

4.2.3 L'élément de production

L'élément `<xsl:output>` est utilisé pour déterminer le format de sortie d'une transformation XSLT. Directement après l'élément racine, nous allons prendre l'habitude de placer l'élément de production : `<xsl:output>` (voir la figure 4.2). Conceptuellement, lors d'une transformation XSLT, un arbre de résultats est d'abord créé, puis cet arbre de résultats est sérialisé pour former, par exemple, un document XML. L'élément `<xsl:output>` agit sur le processus de sérialisation, pas sur la création de l'arbre résultat.

R Ce n'est pas une exigence de la recommandation XSLT qu'un processeur XSLT sérialise l'arbre de résultats qu'il peut rendre disponible pour le traitement, par exemple, une autre API. Dans ce cas, on peut s'attendre à ce qu'un élément `<xsl:output>` soit ignoré.

L'élément `<xsl:output>` a dix attributs, tous facultatifs. L'attribut `method` prend généralement l'une des trois valeurs suivantes : **xml**, **html** ou **text**. Ainsi, si vous souhaitez spécifier sans ambiguïté que la sortie d'une transformation doit être HTML, vous utiliserez "`<xsl:output method="html"/>`" éventuellement avec d'autres attributs autorisés. La sortie par défaut est XML ; par conséquent, lorsqu'aucun élément `<xsl:output>` n'est pas présent, la sortie est XML comme si les éléments suivants étaient spécifiés : "`<xsl:output method="xml"/>`". Une exception à cette règle se produit lorsque la racine de l'élément du document de sortie est **<html>**, auquel cas le processeur XSLT traitera probablement la sortie comme HTML (si aucun attribut de méthode n'est spécifié). L'attribut `version` spécifie la version de la méthode de sortie choisie qui doit être implémentée. Ainsi, pour sortir HTML 4.0, vous pouvez spécifier "`<xsl:output method="html" version="4.0"/>`".

Les processeurs XSLT n'ont pas besoin de prendre en charge toutes les versions des sorties possibles. L'attribut `encoding` spécifie quel schéma de codage de caractères doit être utilisé dans le document de sortie. L'attribut `omit-xml-declaration` s'applique lorsque la méthode de sortie est «xml». Il peut prendre les valeurs "oui" ou "non". Lorsque la valeur est « non », une déclaration XML est automatiquement produite au début d'un document résultat. Si une déclaration XML doit être sortie, l'attribut `standalone` de l'élément `<xsl:output>` indique, s'il est présent, qu'un attribut `standalone` doit être inclus dans la déclaration XML. L'attribut `autonomous` sur l'élément `<xsl:output>` a des valeurs autorisées de «oui» et «non». La valeur spécifiée est également la valeur de l'attribut `autonomous` dans la déclaration XML du document de sortie.

R Une feuille de style XSLT peut contenir plusieurs éléments `<xsl:output>`. Dans ce cas, c'est comme si les éléments présents étaient fusionnés en un élément composite `<xsl:output>`. S'il y a un conflit apparent de valeurs d'attributs, les valeurs explicitement déclarées ont priorité sur les valeurs par défaut et les valeurs avec une priorité d'importation plus élevée ont la priorité sur celles de moindre priorité.

4.3 Les éléments indispensables de XSLT

Cette section décrit plusieurs éléments XSLT et des concepts de base permettant d'être immédiatement utilisables. La syntaxe de chaque élément est affichée, suivie d'une description[Kay08].

4.3.1 <xsl:document>

L'instruction <xsl:document> crée un nœud de document. Le contenu du nœud de document est le résultat du traitement du constructeur de séquence que contient l'instruction <xsl:document>. Une fois que le nœud de document a été construit, son élément de document peut être validé et se voir attribuer une annotation de type basée sur les attributs de type ou de validation, dont un seul peut-être présent. Vous pouvez voir la syntaxe ci-dessous :

```
<xsl:document validation="strict | lax | preserve | strip"?  
type="nom du document"?>  
constructeur de séquence  
</xsl:document>
```

4.3.2 <xsl:element>

L'instruction <xsl:element> crée un élément. Le nom de l'élément créé est déterminé par ses attributs de nom et d'espace de noms. L'attribut name donne le nom qualifié de l'élément et l'attribut namespace donne l'espace de noms de l'élément. Si l'attribut namespace est manquant, le nom qualifié spécifié par l'attribut name est utilisé pour identifier l'espace de noms de l'élément : l'espace de noms associé au préfixe du nom qualifié ou l'espace de noms par défaut s'il n'a pas de préfixe. La syntaxe est la suivante :

```
<xsl:element name="nom de l'element"  
namespace="namespace-uri"?  
inherit-namespaces="yes | no"?  
validation="strict | lax | preserve | strip">  
constructeur de séquence  
</xsl:element>
```

4.3.3 <xsl:function>

La déclaration <xsl:function> définit une fonction de feuille de style. Elle est située au niveau supérieur de la feuille de style, en tant qu'enfant de <xsl:stylesheet>. Le nom de la fonction de feuille de style est spécifié via l'attribut name, qui doit être un nom qualifié avec un préfixe. Les paramètres de la fonction sont spécifiés via les éléments <xsl:param>; c'est une erreur d'avoir plus d'une définition de fonction pour une fonction avec le même nom et le même nombre de paramètres. Si le processeur a déjà une implémentation intégrée d'une fonction portant ce nom, la fonction de feuille de style sera utilisée à la place, sauf si l'attribut override est présent avec une valeur no. Veuillez voir la syntaxe suivante :

```
<xsl:function name="nom de la fonction"
  as="type de sequence"?
  override="yes | no"?>
  (xsl:param*, constructeur de séquence)
</xsl:function>
```

4.3.4 <xsl:template>

L'instruction `<xsl:template>` déclare un modèle. Les modèles peuvent être utilisés de deux manières dans une feuille de style, ils peuvent être appliqués aux nœuds ou ils peuvent être appelés par leur nom. Chaque modèle doit spécifier soit un attribut `match`, afin qu'il puisse être appliqué aux nœuds, soit un attribut `name`, afin qu'il puisse être appelé par son nom avec `<xsl:call-template>`. Un nom et un modèle de correspondance peuvent être spécifiés. Lorsque des modèles sont appliqués à une séquence de nœuds à l'aide de `<xsl:apply-templates>`, ils peuvent être appliqués dans un mode particulier; l'attribut `mode` sur `<xsl:template>` indique le mode dans lequel les modèles doivent être appliqués pour que ce modèle soit utilisé. Si la valeur est `#all`, alors le modèle sera utilisé quel que soit le mode. Vous pouvez voir la syntaxe ci-dessous :

```
<xsl:template match="expression XPath"?
  mode="nom mode | #default | #all"?
  priority="number"?
  name="nom du template"
  as="sequence-type"?>
  (xsl:param*, constructeur de séquence)
</xsl:template>
```

4.3.5 <xsl:apply-templates>

L'instruction `<xsl:apply-templates>`, permet de continuer la transformation des éléments enfants d'un template. `<xsl:apply-templates>` indique également au processeur de rassembler une séquence de nœuds et de traiter chacun des nœuds en trouvant un modèle qui lui correspond et en utilisant ce modèle. Les nœuds auxquels le processeur applique des modèles sont sélectionnés par l'expression contenue dans l'attribut `select`. Si l'attribut `select` est manquant, le processeur applique des modèles aux nœuds enfants du nœud actuel. Les nœuds sont traités dans l'ordre déterminé par les éléments `<xsl:sort>` contenus par l'instruction `<xsl:apply-templates>`. Pour plus de détails, voir la description de `<xsl:sort>`. S'il n'y a pas d'éléments `<xsl:sort>` dans l'instruction `<xsl:apply-templates>`, le processeur parcourt les nœuds dans l'ordre dans lequel ils apparaissent dans la séquence (qui est généralement l'ordre du document). Voir la syntaxe ci-dessous :

```
<xsl:apply-templates select="nœud-sequence-expression"?
  mode="qualified-name | #current | #default"?>
  (xsl:sort | xsl:with-param)*
</xsl:apply-templates>
```

4.3.6 <xsl:call-template>

L'instruction `<xsl:call-template>` appelle un modèle par son nom. Le nom du modèle appelé est contenu dans l'attribut `name`. Les éléments `<xsl:with-param>` contenus dans `<xsl:call-template>` sont utilisés pour définir les paramètres qui sont passés au modèle. Pour plus de détails, voir la description de `<xsl:with-param>`. Veuillez voir également la syntaxe suivante :

```
<xsl:call-template name="qualified-name">
(xsl:with-param*)
</xsl:call-template>
```

4.3.7 <xsl:processing-instruction>

L'instruction `<xsl:processing-instruction>` crée une instruction de traitement dont la cible est le nom contenu dans l'attribut `name`. La valeur de l'instruction de traitement créée est le résultat du traitement du contenu de l'instruction `<xsl:processing-instruction>` ou de l'évaluation de son attribut `select` (elle ne peut pas avoir les deux), comme vous pouvez le voir dans la syntaxe ci-dessous :

```
<xsl:processing-instruction name="unqualified-name"
select="expression XPath"?>
constructeur de séquence
</xsl:processing-instruction>
```

4.3.8 <xsl:sequence>

L'instruction `<xsl:sequence>` ajoute la séquence sélectionnée par son attribut `select` à la séquence en cours de construction en utilisant le constructeur de séquence auquel elle appartient. Il est principalement utilisé pour ajouter des nœuds existants à une telle séquence, car d'autres valeurs peuvent être ajoutées à l'aide de l'instruction `<xsl:copy-of>`. Voir la syntaxe ci-dessous :

```
<xsl:sequence select="expression">
(xsl:fallback*)
</xsl:sequence>
```

4.3.9 <xsl:fallback>

L'instruction `<xsl:fallback>` fournit un traitement alternatif dans le cas où une implémentation ne prend pas en charge un élément particulier. Lorsqu'un processeur XSLT tombe sur une instruction qu'il ne comprend pas (comme une instruction XSLT 2.0 ou un élément d'extension), il recherche un élément `<xsl:fallback>` parmi ses enfants. S'il en trouve un, il traite le contenu de cet élément `<xsl:fallback>`; si ce n'est pas le cas, il arrête le traitement et la transformation échoue. Voir la syntaxe ci-dessous :

```
<xsl:fallback>
constructeur de séquence
</xsl:fallback>
```

4.3.10 <xsl:sort>

L'instruction `<xsl:sort>` est une fonction qui permet de trier un ensemble d'éléments par ordre croissant ou décroissant. `<xsl:sort>` crée une clé de tri en évaluant l'expression contenue dans l'attribut `select` ou le constructeur de séquence dans le contenu de `<xsl:sort>` (vous ne pouvez pas avoir les deux) avec l'élément comme élément courant. Les éléments sont ensuite triés en fonction de cette clé de tri et des autres attributs de l'élément `<xsl:sort>`. L'instruction possède au moins un attribut `select` pour lequel il convient de renseigner une expression XPath permettant alors de sélectionner les informations à trier. Pour mieux comprendre, voir la syntaxe suivante :

```
<xsl:sort select="expression"?
data-type="text | number | qualified-name"?
order="ascending | descending"?
stable="yes | no"?
collation="collation-uri"?
lang="language-code"?
case-order="upper-first | lower-first"?
constructeur de séquence
</xsl:sort>
```

4.3.11 <xsl:perform-sort>

L'instruction `<xsl:perform-sort>` trie une séquence. La séquence à trier peut être sélectionnée à l'aide de l'attribut `Select` ou générée à l'aide du constructeur de séquence imbriqué (vous ne pouvez pas avoir les deux). Le genre lui-même est défini à l'aide des éléments `<xsl:sort>` dans l'élément `<xsl:perform-sort>`. Si vous souhaitez effectuer une seule chose avec la séquence triée (par opposition à son stockage dans une variable afin de la réutiliser), vous devez combiner le tri avec traitement à l'aide de `<xsl:for-each>` ou `<xsl:apply-templates>`. Veuillez voir la syntaxe ci-dessous :

```
<xsl:perform-sort select="expression"?>
(xsl:sort+, constructeur de séquence)
</xsl:perform-sort>
```

4.3.12 <xsl:value-of>

L'instruction `<xsl:value-of>` crée un nœud texte. La valeur du nœud de texte créé est le résultat du traitement du contenu de l'instruction `<xsl:value-of>` ou de l'évaluation de son attribut `Select` (vous ne pouvez pas avoir les deux). La séquence résultante est atomisée, chaque élément collé à une chaîne et ces chaînes concaténées avec la valeur de l'attribut séparateur utilisé comme séparateur entre chaque élément. Si l'attribut séparateur n'est pas présent, il prend par défaut à un seul espace si l'attribut `Select` est utilisé et à une chaîne vide si le contenu de l'élément `<xsl:value-of>` est utilisé. Si l'attribut `disable-output-escaping` est spécifié avec la valeur `yes`, la valeur est sortie sans caractères spéciaux tels que "<" et "&", de sorte qu'ils devraient être remplacés par "<" et "&" respectivement. La syntaxe de l'instruction `<xsl:value-of>` est la suivante :

```
<xsl:value-of select="expression"?
separator="string"?
disable-output-escaping="yes | no">
constructeur de séquence
</xsl:value-of>
```

4.3.13 <xsl:param>

L'élément `<xsl:param>` déclare un paramètre pour un modèle (s'il s'agit de `<xsl:template>`), une fonction de stylesheet (si elle est dans `<xsl:function>`) ou pour la feuille de style dans son ensemble (si c'est au niveau supérieur de la feuille de style). L'attribut `name` détient le nom du paramètre. L'attribut `as` spécifie le type de valeur attendue pour le paramètre. Pour les paramètres de modèles et de stylesheet, l'attribut `required` indique si une valeur doit être fournie pour le paramètre (la valeur par défaut est `no`, qui n'est pas requise). L'attribut `select` ou le contenu de l'élément `<xsl:param>` (vous ne pouvez pas avoir la fois) contient la valeur par défaut du paramètre, qui sera utilisée si aucune valeur n'est explicitement transmise à la feuille de style ou de ce modèle pour ce paramètre. Pour les paramètres de modèle, l'attribut `tunnel` indique si le paramètre est un paramètre de tunnellation. S'il a la valeur `yes`, le paramètre peut être défini à partir d'un `<xsl:with-param>` dont le propre attribut `tunnel` est `yes`, même s'il existe des modèles intermédiaires qui ne déclarent pas ou ne transmettent pas explicitement le paramètre. Voyons maintenant la syntaxe de l'instruction `<xsl:param>` ci-dessous :

```
<xsl:param name="nom"
select="expression XPath"?
as="type de séquence"?
required="yes | no"?
tunnel="yes | no"?>
constructeur de séquence
</xsl:param>
```

4.3.14 <xsl:with-param>

L'élément `<xsl:with-param>` spécifie la valeur qui doit être transmise à un paramètre de modèle (template). Il peut être utilisé lors de l'application de modèles avec `<xsl:apply-templates>` ou des modèles d'appel avec `<xsl:call-template>`. L'attribut `name` détient le nom du paramètre pour lequel la valeur est passée. La valeur du paramètre est déterminée soit par l'attribut `select` ou par le contenu de l'élément `<xsl:param>` (vous ne pouvez pas avoir les deux). L'attribut `as` indique le type attendu du paramètre. Si l'attribut est manquant et que vous définissez le paramètre à l'aide de son contenu, le paramètre contient le nœud de document d'un arbre temporaire. Sinon, le paramètre détient la séquence sélectionnée ou construite. L'attribut `tunnel` détermine si le paramètre est un paramètre de tunnel ou non. Alors même si un modèle ne déclare pas le paramètre, il sera transmis à travers ce modèle dans tous les cas. Les paramètres de tunnel ne peuvent être utilisés que dans des modèles qui déclarent des paramètres à l'aide d'un élément `<xsl:param>` avec un attribut `tunnel` égal à `yes`. Veuillez voir la

syntaxe suivante :

```
<xsl:with-param name="qualified-name"
select="expression"?
as="sequence-type"?
tunnel="yes | no"?>
constructeur de séquence
</xsl:with-param>
```

4.3.15 <xsl:if>

L'instruction `<xsl:if>` effectue un traitement conditionnel. Le contenu de `<xsl:if>` n'est traité que si l'expression contenue dans son attribut `test` est évaluée à une valeur booléenne effective de `true`.

```
<xsl:if test="expression booléenne">
constructeur de séquence
</xsl:if>
```

4.3.16 <xsl:choose>

L'instruction `<xsl:choose>` est une construction conditionnelle qui entraîne le traitement de différentes instructions dans différentes circonstances. Le processeur XSLT traite les instructions contenues dans le premier élément `<xsl:when>` dont l'attribut `test` est évalué comme vrai. Si aucun des attributs de test des éléments `<xsl:when>` n'est évalué comme vrai, le contenu de l'élément `<xsl:otherwise>`, s'il y en a un, est traité. Les conditions incluses dans la fonction `<xsl:choose/>` s'expriment grâce à la fonction `<xsl:when/>`. La fonction `<xsl:when/>` possède un attribut `test` auquel il convient de renseigner une condition, comme vous pouvez le voir dans la syntaxe ci-dessous :

```
<xsl:choose>
<xsl:when test="expression">
... suite de la transformation ...
</xsl:when>
<xsl:otherwise>
... suite de la transformation ...
</xsl:otherwise>
</xsl:choose>
```

4.3.17 <xsl:for-each>

L'instruction `<xsl:for-each>` indique au processeur XSLT de traiter les éléments dans une séquence un par un. Les éléments sont sélectionnés par l'expression contenue dans l'attribut `select`. S'il existe des éléments `<xsl:sort>`, ceux-ci modifient l'ordre dans lequel les éléments sont traités ; s'il n'y en a pas, ils sont traités dans l'ordre des documents. Chacun des nœuds est ensuite traité selon les instructions contenues dans le

```

<xsl:for-each> après tous les éléments <xsl:sort>.
<xsl:for-each select="expression">
(xsl:sort*, constructeur de séquence)
</xsl:for-each>

```

4.3.18 <xsl:for-each-group>

L'instruction `<xsl:for-each-group>` regroupe les éléments dans la séquence sélectionnée par son attribut `select`. Le groupement dépend des autres attributs qui sont présents. `group-by` regroupe les éléments en fonction d'une valeur calculée à l'aide de l'expression de l'attribut `group-by`. `group-adjacent` regroupe les éléments adjacents en fonction d'une valeur calculée à l'aide de l'expression dans l'attribut `group-adjacent`. `group-starting-with` et `group-ending-with` construit des groupes qui commencent ou se terminent par des nœuds correspondant aux modèles des attributs `group-starting-with` et `group-ending-with`. L'un de ces attributs doit être présent. Lors du regroupement par valeur de chaîne à l'aide de `group-by` ou `group-adjacent`, l'attribut `collation` spécifie la collation à utiliser pour comparer les valeurs.

```

<xsl:for-each-group select="expression"
group-by="expression"?
group-adjacent="expression"?
group-starting-with="pattern"?
group-ending-with="pattern"?
collation="collation-uri"?>
(xsl:sort*, constructeur de séquence)
</xsl:for-each-group>

```

4.3.19 <xsl:copy>

L'instruction `<xsl:copy>` crée une copie superficielle du nœud actuel. Pour la plupart des nœuds, cela équivaut à une copie complète, mais pour les nœuds d'élément et de document, c'est légèrement différent. Si le nœud courant est un nœud d'élément ou de document, les enfants de la copie sont les résultats du traitement du contenu de l'instruction `<xsl:copy>`. Si le nœud actuel est un élément, tous les nœuds d'espace de noms de l'élément d'origine sont copiés, à moins que l'attribut `copy-namespaces` ne soit présent avec la valeur `no`. Si l'attribut `inherit-namespaces` est présent avec la valeur `no`, les enfants du nouvel élément n'auront pas de copies des nœuds d'espace de noms de l'élément copié. Les attributs des ensembles d'attributs nommés dans `use-attribute-sets` sont ajoutés à l'élément copié (ainsi que tous les attributs pouvant être ajoutés via les instructions `<xsl:attribute>` dans `<xsl:copy>`). Si le nœud actuel est un nœud d'élément, d'attribut ou de document, les attributs `type` et `validation` contrôlent la validation de l'élément, de l'attribut ou du document. Pour plus de détails, consultez les descriptions dans `<xsl:element>`, `<xsl:attribute>` et `<xsl:document>`.

```

<xsl:copy use-attribute-sets="list-of-qualified-names"?

```

```

copy-namespaces="yes | no"?
inherit-namespaces="yes | no"?
type="qualified-name"?
validation="strict | lax | preserve | strip">
constructeur de séquence
</xsl:copy>

```

4.3.20 <xsl:copy-of>

L'instruction `<xsl:copy-of>` crée une copie complète de tout ce qui est sélectionné par l'expression contenue dans son attribut `select`. Une copie complète d'un nœud comprend des nœuds enfants, des attributs et des nœuds d'espace de noms (sauf si l'attribut `copy-namespaces` est présent avec la valeur `no`). Si le nœud à copier est un nœud d'élément, d'attribut ou de document, les attributs `type` et `validation` contrôlent la validation de la copie. Pour plus de détails, consultez les descriptions dans `<xsl:element>`, `<xsl:attribute>` et `<xsl:document>`. Veuillez voir la syntaxe ci-dessous :

```

<xsl:copy-of select="expression"
copy-namespaces="yes | no"?
type="qualified-name"?
validation="strict | lax | preserve | strip" />

```

4.3.21 <xsl:variable>

L'élément `<xsl:variable>` déclare une variable. Si l'élément `<xsl:variable>` apparaît au niveau supérieur de la feuille de style, en tant qu'enfant de l'élément de document `<xsl:stylesheet>`, il s'agit alors d'une variable globale dont la portée couvre la totalité de la feuille de style. Sinon, il s'agit d'une variable locale avec une portée de ses frères suivants et de leurs descendants. Pour pouvoir utiliser une variable et récupérer la valeur qu'elle contient, il convient de faire précéder son nom par le symbole "\$". Voir la syntaxe ci-dessous :

```

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

```

Grâce à la fonction `<xsl:value-of>`, il est ensuite très simple d'afficher le contenu de la variable.

4.3.22 <xsl:attribute>

L'instruction `<xsl:attribute>` crée un attribut. Lors de l'ajout d'attributs à un élément, toutes les instructions `<xsl:attribute>` doivent précéder toutes les instructions qui ajoutent du contenu à l'élément. L'instruction `<xsl:attribute>` peut également être incluse dans un élément `<xsl:attribute-set>`. Pour plus de détails, voir la description de `<xsl:attribute-set>`. Le nom de l'attribut créé est déterminé par ses attributs de nom et d'espace de noms. L'attribut `name` donne le nom qualifié de l'attribut et

L'attribut `namespace` donne l'espace de noms pour l'attribut. Si le nom contenu dans l'attribut `name` n'inclut pas de préfixe et que l'attribut `namespace` a une valeur, alors l'attribut généré recevra un préfixe généré par le processeur XSLT. La valeur de l'attribut créé est le résultat du traitement du contenu de l'instruction `<xsl:attribute>` ou de l'évaluation de son attribut `select` (`<xsl:attribute>` peut avoir un attribut ou un contenu `select`, mais pas les deux). La séquence résultante est atomisée, chaque élément converti en une chaîne et cette chaîne concaténée avec la valeur de l'attribut `separator` utilisé comme séparateur entre chaque élément. Si l'attribut `separator` n'est pas présent, il prend par défaut un espace unique si l'attribut `select` est utilisé, et une chaîne vide si le contenu de l'élément `<xsl:attribute>` est utilisé. Une fois la valeur construite, l'attribut peut être validé et affecté d'une annotation de type basée sur les attributs `type` ou `validation`, dont un seul peut-être présent. Si l'attribut `type` est présent, l'attribut est donc validé par rapport au type nommé. Si l'attribut `validation` est présent, l'attribut est donc validé en conséquence : `preserve` et `strip` entraînent une annotation de type `xdt:untypedAtomic`, tandis que `strict` et `lax` entraînent une annotation de type basée sur toutes les déclarations d'attribut de niveau supérieur qui ont été importées dans le feuille de style. Si ni l'attribut `type` ni l'attribut `validation` ne sont présents, la validation se poursuit en fonction de l'attribut `default-validation` sur l'élément `<xsl:stylesheet>` comme indiqué ci-dessous :

```
<xsl:attribute name="qualified-name"
namespace="namespace-URI"?
type="qualified-name"?
validation="strict | lax | preserve | strip"?
select="expression"?
separator="string"?>
constructeur de séquence
</xsl:attribute>
```

4.3.23 `<xsl:attribute-set>`

La déclaration `<xsl:attribute-set>` définit un ensemble d'attributs qui peuvent ensuite tous être ajoutés à un élément en une seule fois via l'attribut `use-attribute-sets` sur `<xsl:element>` ou `<xsl:copy>`. L'attribut `name` contient le nom qualifié de l'ensemble d'attributs, par lequel il peut être référencé ultérieurement. L'élément `<xsl:attribute-set>` contient un certain nombre d'éléments `<xsl:attribute>`. L'attribut `use-attribute-sets` contient une liste délimitée par des espaces de noms qualifiés d'autres ensembles d'attributs, de sorte que les attributs des ensembles d'attributs utilisés sont ajoutés à celui-ci.

```
<xsl:attribute-set name="qualified-name"
use-attribute-sets="list-of-qualified-names"?>
(xsl:attribute*)
</xsl:attribute-set>
```

4.4 XPath, XSLT et HTML

XPath et XSLT prennent bien en charge les opérations qui impliquent généralement une itération sur une séquence. Lorsqu'une opération n'est pas prise en charge, vous pouvez utiliser un modèle ou une fonction récursive qui utilise les prédicats pour extraire le premier élément sur lequel opérer et le reste des éléments sur lesquels effectuer une récurrence. Naturellement, au fur et à mesure que XSLT et XPath se développeront, ils incluront des instructions et des fonctions qui permettront aux utilisateurs de faire les choses dont ils ont besoin rapidement et facilement. Cependant, en attendant, les implémenteurs sont tenus de répondre à la pression des utilisateurs pour des fonctionnalités supplémentaires, tout comme Netscape et Microsoft l'ont fait dans leurs navigateurs Web respectifs lors du développement de HTML. Dans ces situations, avec différentes applications prenant en charge différentes fonctionnalités, il peut être difficile de reconnaître quand une fonctionnalité particulière fait partie du langage standard et quand elle est définie dans une implémentation particulière. Finalement, comme avec HTML, cela conduit les implémentations à diriger le processus de normalisation plutôt que l'inverse. Pour éviter la confusion qui pourrait survenir lorsque les implémenteurs étendent XSLT et XPath, les deux langages ont une manière standard de traiter les extensions, de sorte que vous pouvez facilement savoir quelles instructions et fonctions font partie des normes XSLT et XPath et lesquelles sont des extensions définies par l'implémenteur. XSLT vous permet de convertir XML en HTML, en d'autres types de XML ou simplement de récupérer tout le texte XML (voir Figure 4.3). Avec un peu de créativité et une bonne connaissance de XSLT, vous pouvez générer pratiquement n'importe quelle forme de sortie à partir de XML. XSLT fournit des solutions simples et rapides à tous les problèmes de transformation XML. Cependant, les concepteurs de XSLT n'avaient pas prévu que vous utilisiez la spécification sans aide supplémentaire.

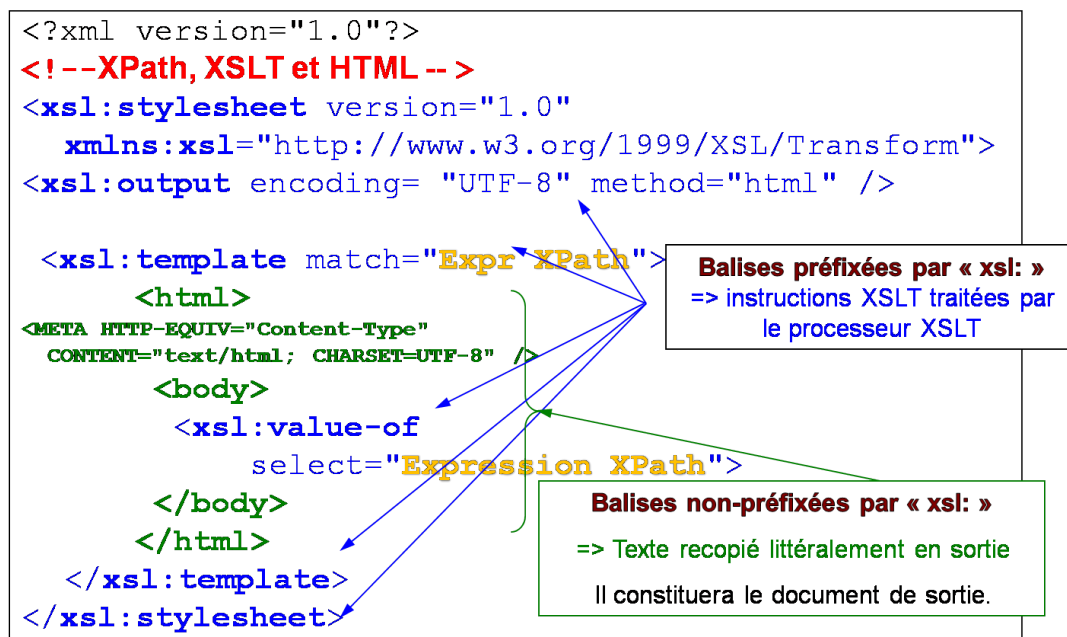


FIGURE 4.3 – Feuille de style récupère tout le texte d'un document XML.

4.5 Exercices résolus

```

<?xml version="1.0" encoding="ISO-8859-1"?> <!-- Exercice_1.xml -->
<?xml-stylesheet href="EXO.xsl" type="text/xsl"?>
<liste>
  <livre>
    <titre genre="Intelligence Artificielle">Images Satellitaires</titre>
    <auteur>Mohammed Mahmoud</auteur>
    <auteur>Ahmed Ali</auteur>
    <parution>2021</parution>
  </livre>
  <livre>
    <titre genre="Intelligence Artificielle">Données déséquilibrées</titre>
    <auteur>DJAFRI Laouni</auteur>
    <parution>2022</parution>
  </livre>
  <livre>
    <titre genre="Mathématique">Statistique inférentielle</titre>
    <auteur> DJAFRI Esseddik El-Moatassim Billah</auteur>
    <parution>2020</parution>
  </livre>
  <livre>
    <titre genre="Medecine"> Covid-19 en Algerie</titre>
    <auteur> DJAFRI Amina Rimes </auteur>
    <auteur>DJAFRI Anfal</auteur>
    <parution>2020</parution>
  </livre>
</liste>

<?xml version="1.0" encoding="UTF-8"?> <!-- EXO.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs" version="2.0">

  <xsl:template match="/liste/livre">
    Livre trouvé
  </xsl:template>

</xsl:stylesheet>

```

Exercice 4.1 . À l'aide de l'éditeur oXygen, écrivez la feuille de style la plus simple possible, à partir de l'exercice "Exercice_1.xml", qui affiche les titres de chaque ouvrage publié en 2020 et après.

Exercice 4.2 . À l'aide de l'éditeur oXygen, écrivez une feuille de style qui permet de récapituler dans un tableau toutes les informations figurant dans le fichier XML (Exercice_1.xml). Les livres les plus récents devront apparaître en haut du tableau et chaque titre devra s'afficher par ordre alphabétique.

4.6 Solutions

4.6.1 Solution de l'exercice 4.1

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="liste">
    <html>
      <head><title>exo1</title></head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="livre">
    <xsl:if test="parution[.]>=2020]">
      <xsl:value-of select="titre"/>
      <xsl:text> : paru en </xsl:text>
      <xsl:value-of select="parution"/>
      <br/>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

L'éditeur oXygen affiche le résultat suivant :

```
Image Satellitaires : paru en 2021
Données déséquilibrées : paru en 2022
Statistique inférentielle : paru en 2020
Covid-19 en Algerie : paru en 2020
```

4.6.2 Solution de l'exercice 4.2

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes=
"xs" version="2.0">
  <xsl:template match="/liste">
    <html>
      <body>
        <table border="1" width="80%">
          <tr>
            <td>Titre</td>
            <td>Auteur</td>
            <td>parution</td>
          </tr>
          <xsl:for-each select="livre">
            <xsl:sort select="parution"
              order="descending"
              data-type="number"
              lang="fr"/>
            <xsl:sort select="titre"
              order="ascending"
              data-type="text"
              lang="fr"/>
            <tr>
              <td><xsl:value-of select="titre"/></td>
              <td><xsl:value-of select="auteur"/></td>
              <td><xsl:value-of select="parution"/></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>

```

L'éditeur oXygen affiche le résultat suivant :

Titre	Auteur	parution
Données déséquilibrées	DJAFRI Laouni	2022
Image Satellitaires	Mohammed Mahmoud Ahmed Ali	2021
Covid-19 en Algerie	DJAFRI Amina Rimes DJAFRI Anfal	2020
Statistique inférentielle	DJAFRI Esseddik El-Moatassim Billah	2020

4.7 Conclusion

Le vieil adage dit que pour un homme avec un marteau, tout ressemble à un clou. Nous ne prétendons pas que XSLT résoudra tous les problèmes que vous rencontrerez. Dans ce chapitre, nous avons discuté des raisons d'utiliser XSLT et des décisions prises lors de la conception de XSLT ; Tout au long de ce chapitre, nous avons également vu des situations standard où XSLT est extrêmement puissant et utile. A cet effet, XSLT est considéré comme une technologie très puissante qui n'a trouvé que récemment un créneau pour elle-même. Comme pour de nombreuses nouvelles technologies, sa plus grande limitation est l'acceptation : de nombreux navigateurs ne prennent tout simplement pas en charge le traitement XSLT et ne le prendront probablement pas en charge dans un avenir prévisible. Pour suivre le rythme des évolutions technologiques, notamment dans le domaine du Web ; la spécification XSLT 2.0 a ajouté un certain nombre de nouvelles fonctionnalités importantes au langage, dont beaucoup sont liées aux modifications apportées à XPath 2.0. Nous avons essayé de manière exhaustive de vous présenter les instructions courantes de XSLT, celles qu'il vous sera probablement demandé d'utiliser pour résoudre des problèmes complexes à l'avenir. Alors gardez à l'esprit que maîtriser ce langage vous apportera pas mal de solutions, notamment dans le domaine du web social et le web sémantique. Dans le chapitre suivant, nous présenterons la technologie XQuery, qui est une technologie complémentaire aux autres technologies que nous avons étudiées dans les chapitres précédents.

Nous vous invitons à rester avec nous pour comprendre de plus en plus
le module de "XML Avancé & Web 2.0" ... !!!



5. XQuery

5.1 Introduction

Ces dernières années, une énorme quantité d'informations est stockée dans des bases de données XML et dans des documents sur un système de fichiers. Cela comprend des données hautement structurées, telles que les chiffres de vente, des données semi-structurées telles que des catalogues de produits, et des données relativement non structurées telles que des lettres, des images, des vidéos, etc. Toutes ces données sont utilisées à diverses fins. Par exemple, les chiffres des ventes peuvent être utiles pour compiler des états financiers qui peuvent être publiés sur le Web, communiquer les résultats aux autorités fiscales, calculer les factures des vendeurs, etc. Pour chacun de ces usages, nous nous intéressons à différents éléments de la donnée et attendons qu'elle soit formatée et transformée selon nos besoins. XQuery (XML Query) conçu par le W3C pour répondre à ces besoins, c'est parce que XQuery est un langage de requêtes puissant pour les documents et les bases de données XML, il utilise une partie du langage XPath afin de naviguer sur un arbre XML. Il vous permet de sélectionner les éléments de données XML qui vous intéressent, de les réorganiser et éventuellement de les transformer, et de renvoyer les résultats dans une structure de votre choix. Dans ce chapitre, nous commençons notre présentation détaillée de XQuery, en nous appuyant sur les technologies pré-requises que nous avons couvertes dans les chapitres précédents. Vous verrez qu'apprendre XQuery n'est pas comme faire un saut quantique dans une autre dimension, mais comme gravir un sommet qui ne devient visible qu'après avoir atteint le sommet des contreforts. Ce chapitre présente également de nombreuses informations clés dont vous aurez besoin pour résoudre les problèmes de bases de données et semi structurées, ainsi que pour surmonter les difficultés liées au web social et web sémantique dans la deuxième partie de ce polycopié. Nous vous encourageons donc à prendre votre temps au fur et à mesure que vous progressez dans ses sections de ce chapitre.

5.2 Fonctionnalités de XQuery

XQuery dispose d'un riche ensemble de fonctionnalités qui permettent d'effectuer de nombreux types d'opérations sur des données et des documents XML, notamment :

- Sélection des informations en fonction de critères spécifiques ;
- Filtrage des informations indésirables ;
- Rechercher des informations dans un document ou un ensemble de documents ;
- Joindre des données à partir de plusieurs documents ou collections de documents ;
- Trier, regrouper et agréger des données ;
- Transformer et restructurer les données XML en un autre vocabulaire ou structure XML ;
- Effectuer des calculs arithmétiques sur des nombres et des dates ;
- Manipulation de chaînes de caractères pour reformater le texte.

Comme vous pouvez le constater, XQuery peut être utilisé non seulement pour extraire des sections de documents XML, mais également pour manipuler et transformer les résultats. Le point idéal de XQuery consiste à interroger des corps de contenu XML stockés dans des bases de données. Pour cette raison, on l'appelle parfois le « SQL de XML ». Certaines des premières implémentations de XQuery se trouvaient dans des produits de base de données XML natifs. Le terme "base de données XML native" fait généralement référence à une base de données conçue pour le contenu XML à partir de zéro, par opposition à une base de données traditionnellement relationnelle. Plutôt que d'être orienté autour de tables et de colonnes, son modèle de données est basé sur des documents hiérarchiques et des collections de documents.

De plus, il y a autant de raisons d'interroger XML qu'il y a de raisons d'utiliser XML. Voici quelques exemples d'utilisations courantes du langage XQuery :

- Extraire des informations d'une base de données relationnelle pour les utiliser dans un service Web ;
- Génération de rapports sur les données stockées dans une base de données pour présentation sur le Web au format XHTML ;
- Recherche de documents textuels dans une base de données XML native et présentation des résultats ;
- Extraire des données de bases de données ou de progiciels et les transformer pour l'intégration d'applications ;
- Combinaison de contenu provenant de sources traditionnellement non XML pour mettre en œuvre la gestion et la diffusion de contenu ;
- Interrogation ad hoc de documents XML autonomes à des fins de test ou de recherche.

5.3 Formes d'une requête XQuery

Une requête XQuery est une composition d'**expressions** dont chaque expression retourne une valeur ou retourne une erreur [Cli20][Ari08][Pet12].

5.3.1 Expressions de chemins (XPath)

Le type de requête le plus simple sélectionne simplement des éléments ou des attributs à partir d'un document d'entrée. Ce type de requête est appelé expression de chemin. Pour ce faire, nous nous appuyons sur le document XML pour l'exercice pratique 2.2.2 (Chapitre 2) que nous avons déjà vu dans la section « Validation des documents XML ». Par exemple, l'expression XPath : `doc("agenda.xml")//personne/nom` sélectionnera tous les noms dans le document `agenda.xml`.

Les expressions de chemin sont utilisées pour parcourir une arborescence XML afin de sélectionner les éléments et les attributs d'intérêt. Ils sont similaires aux chemins utilisés pour les noms de fichiers dans de nombreux systèmes d'exploitation. Ils consistent en une série d'étapes, séparées par des barres obliques, qui traversent les éléments et les attributs dans les documents XML. Dans cet exemple, il y a trois étapes :

1. `doc("agenda.xml")` appelle une fonction XQuery nommée **doc**, en lui passant le nom du fichier à ouvrir ;
2. `personne` sélectionne l'élément de personne, l'élément le plus à l'extérieur du document ;
3. `nom` sélectionne tous les noms de l'élément `personne`

Le résultat de la requête sera les quatre noms des personnes, exactement tels qu'ils apparaissent (avec les mêmes attributs et contenus) dans le document d'entrée. L'exemple ci-dessous montre le résultat complet :

```
<?xml version="1.0" encoding="UTF-8"?>
<nom titre="M">sahnoun</nom>
<nom titre="Mlle">ayad</nom>
<nom titre="M">youb</nom>
<nom titre="Mlle">salah</nom>
```

Les expressions de chemin peuvent également renvoyer des attributs, en utilisant le symbole `@`. Par exemple, l'expression de chemin : `doc("agenda.xml")/*/nom[@titre]` renverra les quatre attributs `titre` dans le document d'entrée. L'astérisque (*) peut être utilisé comme caractère générique pour indiquer n'importe quel nom d'élément. Dans cet exemple, le chemin renverra tous les noms des personnes de l'élément le plus à l'extérieur, quel qu'il soit cet élément. Vous pouvez également utiliser une double barre oblique (//) pour renvoyer les éléments du `nom` qui apparaissent n'importe où dans le document `agenda`, comme dans : `doc("agenda.xml")//nom[@titre]`

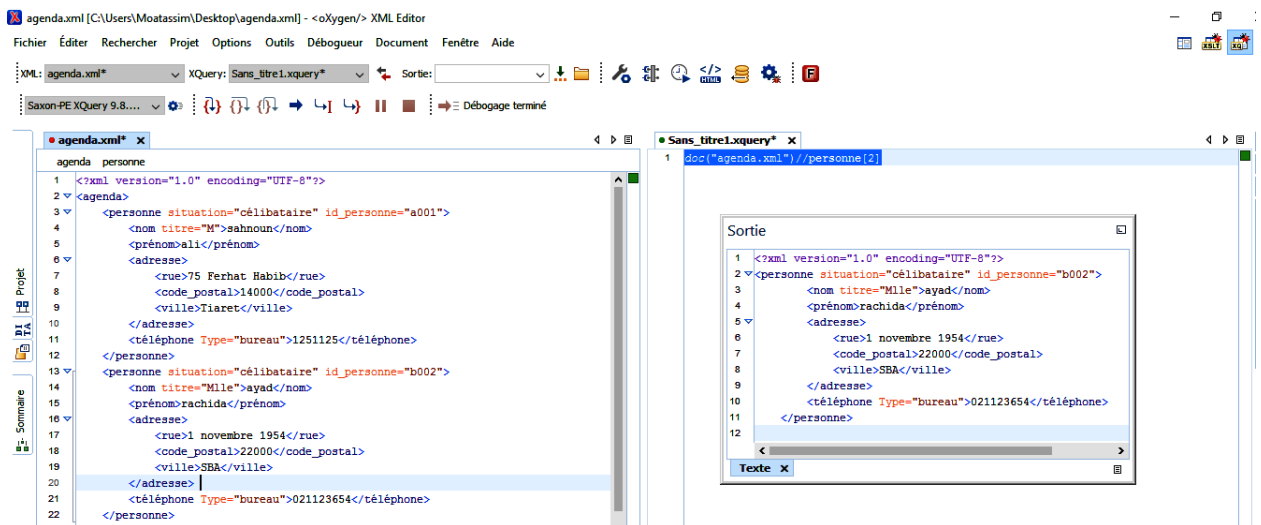
En plus de parcourir le document XML, une expression de chemin peut contenir des prédicats qui filtrent les éléments ou les attributs qui ne répondent pas à un critère particulier. Les prédicats sont indiqués par des crochets. Par exemple, l'expression de chemin : `doc("agenda.xml")/agenda/personne/nom[@titre="M"]` contient un

prédicat. Il sélectionne uniquement les éléments de nom dont la valeur d'attribut titre est M. Le résultat obtenu est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<nom titre="M">sahnoun</nom>
<nom titre="M">youb</nom>
```

Lorsqu'un prédicat contient un nombre, il sert d'index.

Par exemple : `doc("agenda.xml")//personne[2]` renverra le deuxième élément personne dans agenda. Le résultat obtenu en utilisant oXygen est le suivant :



Les expressions de chemin (XPath) sont pratiques en raison de leur syntaxe compacte et facile à mémoriser. Cependant, ils ont une limitation, par exemple, ils ne peuvent renvoyer que des éléments et des attributs tels qu'ils apparaissent dans les documents d'entrée. Tous les éléments sélectionnés dans une expression de chemin (XPath) apparaissent dans les résultats avec les mêmes noms, les mêmes attributs et contenus, et dans le même ordre que dans le document d'entrée. Lorsque vous sélectionnez les éléments personne, vous les obtenez avec tous leurs enfants et avec leurs attributs situation. Les expressions de chemin (XPath) sont traitées en détail au chapitre 3.

Nous vous conseillons de vous référer au troisième chapitre car il complète le cinquième chapitre !!!...

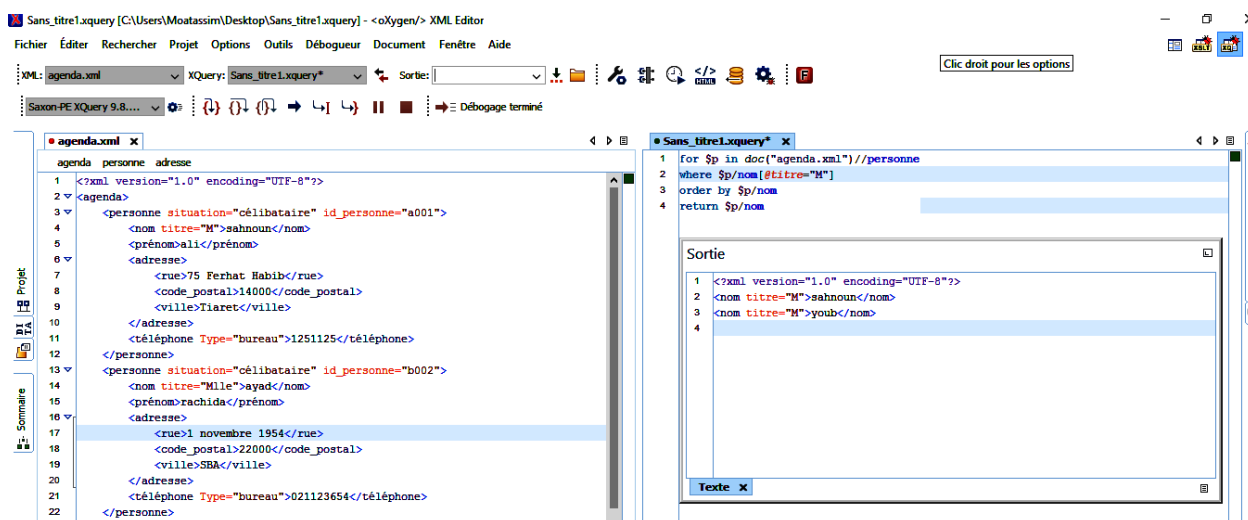
5.3.2 Expressions FLOWR

Cette partie décrit les fonctionnalités de XQuery pour sélectionner, filtrer et joindre des données à partir d'un ou plusieurs documents d'entrée. L'expression FLWOR (prononcé «*flower*») est l'une des caractéristiques les plus puissantes et les plus distinctives de XQuery. FLWOR est un acronyme : FOR, LET, WHERE, ORDER BY, RETURN. FLWOR est vaguement analogue à SELECT-FROM-WHERE de SQL et peut être utilisé

pour fournir une fonctionnalité de type jointure aux documents XML. Une expression FLWOR est une expression comme toutes les autres que nous avons déjà vues, telles que les expressions de chemin (XPath) que nous avons abordées dans le chapitre 3. De plus, une expression FLWOR doit toujours renvoyer une valeur, même si cette valeur est la séquence vide. FLWOR, contrairement aux expressions de chemin (XPath), vous permet de manipuler, transformer et trier vos résultats. L'exemple ci-dessous montre un FLWOR simple qui renvoie les noms de toutes les personnes de type masculin, c'est-à-dire (titre="M").

```
for $p in doc("agenda.xml")//personne
where $p/nom[@titre="M"]
order by $p/nom
return $p/nom
```

Le résultat obtenu en utilisant oXygen est le suivant :



Comme vous pouvez le voir, le FLWOR est composé de plusieurs parties :

1. La clause : for

Cette clause établit une itération à travers les nœuds `personne`, et le reste du FLWOR est évalué une fois pour chacune des occurrences de `nom`. A chaque fois, une variable nommée `$p` est liée à un élément `personne` différent. Les signes dollar sont utilisés pour indiquer les noms de variables dans XQuery.

Pour mieux comprendre la clause `for`, il faut prendre l'exemple suivant : si on veut fournir une séquence d'entiers dans la clause `for` afin de spécifier le nombre de fois à itérer. Cela peut être accompli grâce à une expression de plage, qui crée une séquence d'entiers consécutifs. Alors, l'expression de l'intervalle 1 à 3 correspond à une séquence d'entiers (1, 2, 3). Le FLWOR illustré dans l'exemple ci-dessous itère trois fois et renvoie trois éléments NBR.

Requête XQuery

```
for $i in 1 to 3
return <NBR>$i</NBR>
```

Résultat

```
<NBR>1</NBR>
<NBR>2</NBR>
<NBR>3</NBR>
```

Vous pouvez utiliser la fonction `reverse` si vous voulez descendre en valeur, comme dans `:for $i in reverse(1 to 3)`.

Vous pouvez également incrémenter d'une valeur autre que 1 en utilisant une expression telle que `:for $i in (1 to 100)[. mod 2 = 0]`, ce qui vous donne tous les autres nombres (2, 4, 6, etc.) jusqu'à 100.

Les clauses multiples : for

Vous pouvez utiliser plusieurs clauses `for` dans un `FLWOR`, ce qui est similaire aux boucles imbriquées dans un langage de programmation. Le résultat est que le reste du `FLWOR` est évalué pour chaque combinaison des valeurs des variables. L'exemple ci-dessous montre une requête avec deux clauses `for`, et montre l'ordre des résultats.

Requête XQuery

```
for $i in (1, 2) for $j in ("a", "b")
return <NBRCAR>i est $i et j est $j</NBRCAR>
```

Résultat

```
<NBRCAR> i est 1 et j est a </NBRCAR>
<NBRCAR> i est 1 et j est b </NBRCAR>
<NBRCAR> i est 2 et j est a </NBRCAR>
<NBRCAR> i est 2 et j est b </NBRCAR>
```

L'ordre est important ; il utilise la première valeur de la première variable (`$i`) et itère sur les valeurs de la deuxième variable (`$j`), puis prend la deuxième valeur de `$i` et itère sur les valeurs de `$j`.

En outre, plusieurs variables peuvent être liées dans une seule clause `for`, séparées par des virgules. Cela a le même effet que d'utiliser plusieurs clauses `for`. L'exemple ci-dessous renvoie les mêmes résultats que l'exemple précédent. Cette syntaxe est plus courte mais peut-être moins claire dans le cas d'expressions complexes.

```
for $i in (1, 2), $j in ("a", "b")
return <NBRCAR> i est $i et j est $j</NBRCAR>
```

2. La clause : let

La clause `let` (le L dans FLWOR) est utilisée pour définir la valeur d'une variable. Contrairement à une clause `for`, elle ne crée pas d'itération. L'exemple ci-dessous montre un FLWOR qui renvoie le même résultat que l'exemple précédent (ci-dessus). La deuxième ligne est une clause `let` qui affecte l'enfant de l'élément `personne` à une variable appelée `$nom`. La variable `$nom` est ensuite référencée ultérieurement dans le FLWOR, à la fois dans la clause `order by` et dans la clause `return`.

```
for $p in doc("agenda.xml")//personne
let $nom := $p/nom
where $p/nom[@titre="M"]
order by $p/nom
return $p/nom
```

- R** La clause `let` sert de commodité de programmation qui évite de répéter la même expression plusieurs fois. En utilisant certaines implémentations, cela peut également améliorer les performances, car l'expression n'est évaluée qu'une seule fois au lieu de chaque fois qu'elle est nécessaire.

Pour illustrer la différence entre les clauses `for` et `let`, comparez le premier exemple de la clause `for` (séquence d'entiers (1, 2, 3)) avec l'exemple ci-dessous.

Requête XQuery

```
let $i := (1 to 3)
return <NBR>$i</NBR>
```

Résultat

```
<NBR>1 2 3</NBR>
```

Le FLWOR avec la clause `let` ne renvoie qu'un seul élément `NBR`, car aucune itération n'a lieu et la clause `return` n'est évaluée qu'une seule fois.

Une ou plusieurs clauses `let` peuvent être entremêlées avec une ou plusieurs clauses `for`. Chacune des clauses `let` et `for` peut faire référence à une variable liée dans n'importe quelle clause précédente. La seule exigence est qu'elles apparaissent toutes avant les clauses `where`, `order by` ou `return` de ce FLWOR. L'exemple ci-dessous montre un tel FLWOR.

```
let $doc := doc("agenda.xml") for $p in $doc//personne
let $pd := $p//nom/@titre
let $pn := $p/nom
where $pd = "M" or $pd = "Mlle"
return $pn
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<nom titre="M">sahnoun</nom>
<nom titre="Mlle">ayad</nom>
<nom titre="M">youb</nom>
<nom titre="Mlle">salah</nom>
```

Comme pour les clauses `for`, les clauses `let` peuvent être représentées à l'aide d'une syntaxe légèrement raccourcie qui remplace le mot clé `let` par une virgule (,) comme dans l'exemple ci-dessous :

```
let $doc := doc("agenda.xml")
for $p in $doc//personne
let $pd := $p//nom/@titre, $pn := $p/nom
where $pd = "M" or $pd = "Mlle"
return $pn
```

Une autre utilisation pratique de la clause `let` consiste à exécuter plusieurs fonctions ou opérations dans l'ordre. Par exemple, supposons que nous voulions prendre une chaîne de caractères et remplacer toutes les instances de `at` par `@`, remplacer toutes les instances de point par un point (.) et supprimer tous les espaces restants. Nous pourrions écrire l'expression : `replace(replace(replace($MaChaine,'at','@'),'point','.'),' ','')`, mais c'est difficile à lire et à déboguer, d'autant plus que d'autres fonctions sont ajoutées. Une alternative est l'expression ci-dessous :

```
let $MaChaine2 := replace($MaChaine,'at','@')
let $MaChaine3 := replace($MaChaine2,'dot','.')
let $MaChaine4 := replace($MaChaine3,' ','')
return $MaChaine4
```

3. La clause : where

La clause `where` est utilisée pour spécifier les critères qui filtrent les résultats du FLWOR. Par exemple, cette clause sélectionne uniquement les noms des personnes de titre masculin "M". Cela a le même effet qu'un prédicat (`"[@titre = "M"]`) dans une expression de chemin (XPath).

De plus, la clause `where` peut faire référence à des variables liées par une clause `for` ou `let`. Par exemple : `where $pd = "M" or $pd = "Mlle"` fait référence à la variable `$pd`. En plus d'exprimer des filtres complexes, la clause `where` est également très utile pour les jointures. Une seule clause `where` peut être incluse par FLWOR, mais elle peut être composée de plusieurs expressions jointes par les mots clés `and` et `or`, comme le montre l'exemple ci-dessous sous l'éditeur `oXygen` :

```

agenda.xml x
agenda personne adresse
13 <personne situation="célibataire" id_personne="b002">
14 <nom titre="Mlle">ayad</nom>
15 <prénom>rachida</prénom>
16 <adresse>
17 <rue>1 novembre 1954</rue>
18 <code_postal>22000</code_postal>
19 <ville>SBA</ville>
20 </adresse>
21 <téléphone Type="bureau">021123654</téléphone>
22 </personne>
23 <personne situation="marié" id_personne="a003">
24 <nom titre="M">youb</nom>
25 <prénom>sidahmed</prénom>
26 <adresse>
27 <rue>10 Amir AEK</rue>
28 <code_postal>31000</code_postal>
29 <ville>oran</ville>
30 </adresse>
31 <téléphone Type="perso">001255</téléphone>
32 </personne>
33 <personne situation="célibataire" id_personne="a004">
34 <nom titre="Mlle">salah</nom>
35 <prénom>aicha</prénom>
36 <adresse>
37 <rue>05 juillet 1962</rue>
38 <code_postal>16000</code_postal>
39 <ville>Alger</ville>
40 </adresse>
41 <téléphone Type="portable">897422258</téléphone>
42 </personne>
43 </agenda>

Sans_titre1.xquery x
1 for $p in doc("agenda.xml")//personne
2 let $pd := $p//adresse
3 where $p//ville="Alger"
4 and starts-with($p//nom, "s")
5 or starts-with($p//nom, "y")
6 return $p

Sortie
1 <?xml version="1.0" encoding="UTF-8"?>
2 <personne situation="marié" id_personne="a003">
3 <nom titre="M">youb</nom>
4 <prénom>sidahmed</prénom>
5 <adresse>
6 <rue>10 Amir AEK</rue>
7 <code_postal>31000</code_postal>
8 <ville>oran</ville>
9 </adresse>
10 <téléphone Type="perso">001255</téléphone>
11 </personne>
12 <personne situation="célibataire" id_personne="a004">
13 <nom titre="Mlle">salah</nom>
14 <prénom>aicha</prénom>
15 <adresse>
16 <rue>05 juillet 1962</rue>
17 <code_postal>16000</code_postal>
18 <ville>Alger</ville>
19 </adresse>
20 <téléphone Type="portable">897422258</téléphone>
21 </personne>
22

```

4. La clause : order by

La clause `order by` dans une expression FLWOR spécifie l'ordre dans lequel les valeurs sont traitées par la clause `return`. Cette clause trie, par exemple, les résultats des noms de personnes par ordre alphabétique, ce qui n'est pas possible avec les expressions de chemin. Si aucune clause `order by` n'est présente, les résultats d'une expression FLWOR sont renvoyés dans un ordre non déterministe. De plus, la clause `order by` contient une ou plusieurs spécifications de classement. Les spécifications de classement sont utilisées pour réorganiser les tuples des liaisons de variables qui sont conservées après avoir été filtrées par la clause `where`. L'ordre résultant détermine l'ordre dans lequel la clause `return` est évaluée. L'exemple ci-dessous montre une clause `order by` dans un FLWOR.

```

agenda.xml x
agenda personne adresse
1 <?xml version="1.0" encoding="UTF-8"?>
2 <agenda>
3 <personne situation="célibataire" id_personne="a001">
4 <nom titre="M">sahnoun</nom>
5 <prénom>ali</prénom>
6 <adresse>
7 <rue>75 Ferhat Habib</rue>
8 <code_postal>14000</code_postal>
9 <ville>Tiaret</ville>
10 </adresse>
11 <téléphone Type="bureau">1251125</téléphone>
12 </personne>
13 <personne situation="célibataire" id_personne="b002">
14 <nom titre="Mlle">ayad</nom>
15 <prénom>rachida</prénom>
16 <adresse>
17 <rue>1 novembre 1954</rue>
18 <code_postal>22000</code_postal>
19 <ville>SBA</ville>
20 </adresse>
21 <téléphone Type="bureau">021123654</téléphone>
22 </personne>
23 <personne situation="marié" id_personne="a003">
24 <nom titre="M">youb</nom>
25 <prénom>sidahmed</prénom>
26 <adresse>
27 <rue>10 Amir AEK</rue>
28 <code_postal>31000</code_postal>
29 <ville>oran</ville>
30 </adresse>
31 <téléphone Type="perso">001255</téléphone>

Sans_titre1.xquery x
1 for $p in doc("agenda.xml")//nom
2 order by $p/@titre
3 return $p

Sortie
1 <?xml version="1.0" encoding="UTF-8"?>
2 <nom titre="M">sahnoun</nom>
3 <nom titre="M">youb</nom>
4 <nom titre="Mlle">ayad</nom>
5 <nom titre="Mlle">salah</nom>
6

```

Nous pouvons éventuellement spécifier les éléments suivants :

- Croissant et décroissant spécifient le sens du tri. La valeur par défaut est ascendante.
- Vide le plus grand et vide le moins spécifient comment trier la séquence vide.
- Collation, suivi d'un URI de collation entre guillemets, spécifie une collation utilisée pour déterminer l'ordre de tri des chaînes de caractères.

Par exemple, si vous spécifiez : `order by $p/@titre, $p/@situation descending`, cette requête s'applique uniquement à `$p/@situation`, pas à `$p/@titre`. Si vous voulez que les deux soient triés par ordre décroissant, vous devez spécifier : `order by $p/@titre descending, $p/@situation descending`.

5. La clause : return

La clause de retour se compose du mot-clé `return` suivi de l'expression unique à renvoyer. Elle est évaluée une fois pour chaque itération, en supposant que l'expression (des entiers) est évaluée comme vraie. La valeur de résultat (des entiers) du FLWOR est une séquence d'éléments renvoyés par chaque évaluation de la clause `return`. Par exemple, la valeur (des entiers) du FLWOR est :

```
for $i in (1 to 2)
return <NBR>$i</NBR>
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<NBR>1</NBR>
<NBR>2</NBR>
```

Alors, le résultat est une séquence de deux éléments `NBR`, un pour chaque fois que la clause `return` a été évaluée.

Si plusieurs expressions doivent être incluses dans la clause `return`, elles peuvent être combinées dans une séquence. Par exemple, la requête XQuery ci-dessous (FLWOR) :

```
for $i in (1 to 2)
return (<A>$i</B>, <A>$i</B>)
```

Dans ce cas, le résultat de la requête ci-dessus renvoie une séquence de quatre éléments, deux pour chaque fois que la clause `return` est évaluée. Les parenthèses et la virgule sont utilisées dans la clause `return` pour indiquer qu'une séquence des deux éléments doit être renvoyée. Si aucune parenthèse ou virgule n'était utilisée, les deux constructeurs d'éléments ne seraient pas considérés comme faisant partie du FLWOR.

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<A>1</A>
<B>1</B>
<A>2</A>
<B>2</B>
```


5.4 XQuery avancé

Pour aller loin dans la syntaxe des requêtes XQuery, examinons quelques-unes des requêtes les plus avancées. Cette section décrit la syntaxe et les méthodes de certaines fonctionnalités de requête fréquemment demandées. Vous pouvez avoir ces mêmes exigences pour vos requêtes, mais même si ce n'est pas le cas, cette section vous montrera des façons créatives d'implémenter la syntaxe XQuery [Bec22][Cli20].

5.4.1 Fonctions personnalisées sur les éléments et les attributs

Parfois, vous souhaitez supprimer ou ajouter des attributs, ou modifier leurs noms. Cependant, le langage XQuery n'a pas de capacité de mise à jour spécifique, ni de fonctions ou d'opérateurs spéciaux qui effectuent ces modifications mineures. Par exemple, il n'y a pas de syntaxe directe qui signifie "sélectionnez tous les éléments nom, mais omettez leurs attributs titre".

La bonne nouvelle est que vous pouvez accomplir ces modifications en **reconstruisant** les éléments. Par exemple, vous pouvez écrire une requête pour construire un nouvel élément nom et inclure tous les enfants **sauf** les attributs. Encore mieux, vous pouvez écrire une fonction définie par l'utilisateur qui utilise des constructeurs calculés pour gérer ces modifications dans le cas général. Cette sub-section décrit certaines modifications courantes et fournit des fonctions utiles pour gérer ces cas. Notez que ces fonctions sont destinées à modifier la manière dont les éléments et les attributs apparaissent dans les résultats d'une requête, et non à les mettre à jour dans une base de données XML.

Comment ajouter des attributs à un élément ?

Pour ajouter un attribut à un élément, vous pouvez utiliser une fonction personnalisée comme celle illustrée dans l'exemple ci-dessous. Il prend comme arguments un élément, ainsi qu'un nom et une valeur d'attribut, et renvoie un élément nouvellement construit avec cet attribut ajouté.

Dans ce cas, la fonction utilise des constructeurs calculés pour créer dynamiquement un nouvel élément portant le même nom que l'élément d'origine passé à la fonction. Il utilise également un constructeur calculé pour créer l'attribut, en spécifiant son nom et sa valeur sous forme d'expressions.

Il copie ensuite l'attribut et les nœuds enfants de l'élément d'origine. L'expression `node()` est utilisée plutôt que `*` parce que `node()` renverra du texte, des instructions de traitement et des nœuds de commentaires en plus des éléments enfants. Cependant, `node()` ne renvoie pas d'attributs, donc une expression séparée `@*` est utilisée pour les copier.

L'expression :

```
doc("agenda.xml")//nom/functx:add-attribute(., "xml:sexe", "M|Mlle")
```

On utilise cette fonction pour renvoyer tous les éléments du nom de la personne, avec un attribut `xml:sexe="M|Mlle"` supplémentaire.

```

● Sans_titre1.xquery x
1 declare namespace functx = "http://www.functx.com";
2 declare function functx:add-attribute
3 ($element as element(), $name as xs:string,
4 $value as xs:anyAtomicType?) as element() {
5   element { node-name($element)}
6     { attribute {$name} {$value},
7       $element/!*,
8       $element/node() }
9 };
10 doc("agenda.xml")//nom/functx:add-attribute(., "xml:sexe", "M|Mlle")
11
12
Sortie
1 <?xml version="1.0" encoding="UTF-8"?>
2 <nom xml:sexe="M|Mlle" titre="M">sahnoun</nom>
3 <nom xml:sexe="M|Mlle" titre="Mlle">ayad</nom>
4 <nom xml:sexe="M|Mlle" titre="M">youb</nom>
5 <nom xml:sexe="M|Mlle" titre="Mlle">salah</nom>
6
Texte x

```

Comment supprimer des attributs d'un élément ?

La suppression d'attributs nécessite également la reconstruction de l'élément d'origine. L'exemple ci-dessous montre une fonction qui supprime les attributs d'un seul élément. Il le fait en reconstruisant l'élément, en copiant le contenu et tous les attributs sauf ceux dont les noms sont spécifiés dans le deuxième argument.

```

● Sans_titre1.xquery* x
1 declare namespace functx = "http://www.functx.com";
2 declare function functx:remove-attributes
3 ($element as element(), $names as xs:string*) as element() {
4   element { node-name($element)}
5     { $element/!*[not(name() = $names)],
6       $element/node() }
7 };
8 doc("agenda.xml")//nom/functx:remove-attributes(., ("titre", ""))
9
10
11
Sortie
1 <?xml version="1.0" encoding="UTF-8"?>
2 <nom>sahnoun</nom>
3 <nom>ayad</nom>
4 <nom>youb</nom>
5 <nom>salah</nom>
6
Texte x

```

La fonction acceptera une séquence de chaînes représentant les noms d'attributs à supprimer. Par exemple, l'expression suivante :

```
doc("agenda.xml")//nom/functx:remove-attributes(., ("titre", ""))
```

Cette requête renvoie tous les éléments nom du document, sans les attributs titre.

Comment convertir des attributs en éléments ?

Vous pouvez aller plus loin et convertir les attributs en éléments. La fonction récursive `local:att_to_elt`, montrée dans l'exemple ci-dessous, accomplit cela.

```

1 declare function local:att_to_elt($tags as element()*)
2 {for $elem in $tags
3  return element personne {
4    for $child in $elem/(@*)
5    return element { if ($child instance of attribute())
6      then name($child)
7      else 'personne'}
8      { string($child)}}
9 };
10 local:att_to_elt(doc("agenda.xml")//personne)
11
12
13

```

Sortie

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <personne>
3   <situation>célibataire</situation>
4   <id_personne>a001</id_personne>
5 </personne>
6 <personne>
7   <situation>célibataire</situation>
8   <id_personne>b002</id_personne>
9 </personne>
10 <personne>
11   <situation>marié</situation>
12   <id_personne>a003</id_personne>
13 </personne>
14 <personne>
15   <situation>célibataire</situation>
16   <id_personne>a004</id_personne>
17 </personne>
18

```

Activer Windows
Accédez aux paramètres pour activer Windows.

Comment combiner les résultats (séquences) en XQuery ?

Les résultats de votre requête peuvent être constitués de plusieurs FLWOR ou d'autres expressions renvoyant chacune une séquence de résultats. De plus, les séquences que vous utilisez dans vos clauses `for` et `let` peuvent être composées de plusieurs séquences. Il existe plusieurs façons de combiner deux ou plusieurs séquences pour former une troisième séquence. Ils diffèrent par les éléments sélectionnés, si leur ordre est affecté, si les doublons sont éliminés et si les valeurs atomiques sont autorisées dans les séquences.

La façon la plus connue de fusionner deux séquences est simplement de créer une troisième séquence qui est la concaténation des deux premières. C'est ce qu'on appelle un constructeur de séquence, et il utilise des parenthèses et des virgules pour concaténer deux séquences ensemble. Par exemple :

```

let $a := doc("agenda.xml")//personne
let $u := doc("Univ.xml")//site_web
return ($a, $u)

```

Alors, le résultat de cette requête renvoie une séquence qui est la concaténation de deux autres séquences, \$p et \$u. Les éléments dans \$p sont *les premiers dans la séquence*, puis les éléments dans \$u, dans l'ordre dans lequel ils apparaissent dans cette séquence. Aucune tentative n'est faite pour éliminer les doublons ou trier les éléments de quelque manière que ce soit.

R Notez que la concaténation est le seul moyen de combiner des séquences contenant des **valeurs atomiques** (revenir au chapitre 2 : les types simples); les **unions**, les **intersections** et les **exceptions** fonctionnent sur des séquences contenant uniquement des nœuds.

1. **Les expressions Union** : Une autre approche pour combiner des séquences de nœuds consiste à utiliser une expression d'union, qui est indiquée par le mot-clé union ou le caractère de barre verticale (|). Les deux opérateurs ont exactement la même signification. Les nœuds résultants sont réorganisés dans l'ordre des documents. Nous utilisons l'opérateur barre verticale pour sélectionner l'union nom et prénom de la personne.

```

1 doc("agenda.xml")//personne/(nom | prénom)
2
3
4

```

Sortie

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <nom titre="M">sahnoun</nom>
3 <prénom>ali</prénom>
4 <nom titre="Mlle">ayad</nom>
5 <prénom>rachida</prénom>
6 <nom titre="M">youb</nom>
7 <prénom>sidahmed</prénom>
8 <nom titre="Mlle">salah</nom>
9 <prénom>aicha</prénom>
10

```

Texte x

Une alternative équivalente consiste à utiliser l'opérateur de barre verticale pour séparer deux expressions entières de chemin à plusieurs étapes, comme dans : `doc("agenda.xml")//nom | doc("agenda.xml")//prénom`

Une union élimine les nœuds en double non seulement entre les séquences, mais également dans l'une ou l'autre des séquences d'origine.

2. **Les expressions Intersection** : Une expression d'intersection donne une séquence qui contient uniquement les nœuds qui se trouvent dans les deux séquences d'origine. Comme pour les expressions d'union, les nœuds en double (basés sur l'identité du nœud) sont éliminés et les nœuds résultants sont réorganisés dans l'ordre du document. Par exemple, l'expression :


```

let $p := doc("agenda.xml")//nom
return $p[@titre="M"] intersect $p[//prénom = "ali"]

```

Le resultat obtenu :

```

<nom titre="M">sahnoun</nom>
<nom titre="M">youb</nom>

```

3. **Les expressions Exception** : Une expression `except` résulte en une séquence qui contient uniquement les nœuds qui se trouvent dans la première séquence, mais pas dans la seconde. Comme pour les expressions d'union, les valeurs en double (basées sur l'identité du nœud) sont éliminées et les nœuds résultants sont réorganisés dans l'ordre du document. Par exemple, l'expression : `doc("agenda.xml")//adresse/(* except ville)` renvoie tous les éléments enfants de l'adresse à l'exception des éléments ville.

Questions de cours

Répondez aux questions suivantes pour revoir votre compréhension de la première partie du module XML avancé & Web 2.0 ou pour en discuter avec vos collègues de Master II-Génie Logiciel :

1. Comment XQuery se compare-t-il à XSLT ?
2. Pourquoi apprendre XQuery pourrait-il être plus facile pour les débutants complets que pour ceux qui connaissent d'autres langages de programmation ?
3. Qu'en est-il de XQuery qui en fait un langage fonctionnel et le distingue des langages impératifs ?
4. En quoi l'utilisation de la programmation fonctionnelle par XQuery lui confère-t-elle des avantages par rapport aux langages de programmation procéduraux ?
5. A quoi servent les accolades ? A quoi sert une expression délimitée ? Dans quels types d'éléments peut-elle être insérée ?
6. Qu'est-ce qu'une expression FLWOR ? Quelles sont les principales clauses qui composent l'acronyme ?
7. Quelles clauses chaque expression FLWOR doit-elle contenir ?
8. Que signifie l'expression « lier une variable à une valeur » ?
9. Quelle est la différence entre une clause `let` et une clause `for` ?
10. Une expression FLWOR peut-elle se passer d'une clause `let` ou `for` ? Peut-elle se passer d'une clause de retour ?
11. Que fait une clause `where` ?
12. Comment trions-nous les résultats par ordre alphabétique ou par ordre alphabétique inverse ?
13. Quelle est la différence entre les deux clauses `order by` et `group by` ?
14. Pourquoi XQuery utilise-t-il la notion de valeurs booléennes effectives ?
15. Une expression conditionnelle peut-elle se passer d'une clause `else` ?

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<Cours>
  <titre > XML Avancé et Web 2.0 </titre>
  <corps >
    <p >
      Les DTD, <important>c'est facile</important>, il suffit de lire le
      cours de XML Avancé <cite ref="siteXML" />.
    </p>
    <p >
      Il y a aussi de bons livres <cite ref="Web 2.0" />.
    </p>
  </corps>
  <références >
    <référence code="siteXML">
      <nom>w3schools</nom>
      <url> https://www.w3schools.com/xml </url>
    </référence>
    <référence code="Web 2.0">
      <intitulé>Ajax</intitulé>
      <auteur>Laouni DJAFRI</auteur>
      <auteur>Esseddik El Moatassim Billah</auteur>
      <date>2022</date>
      <editor>ONT-Algerie</editor>
    </référence>
  </références>
</Cours>

```

FIGURE 5.1 – Document XML bien formé.

Exemple pratique 5.4.1

À l'aide de l'éditeur oXygen, et à partir du document XML bien formé ci-dessus, en considérant que le nœud contextuel est « Cours », donnez les requêtes (XQuery) suivantes :

1. Le nœud racine ;
2. Tous les descendants de la racine ;
3. Tous les attributs ;
4. Tous les nœuds de type texte ;
5. Les instructions de traitement présentes ;
6. Les contenus de tous les paragraphes ;
7. Les paragraphes contenant des aspects importants ;
8. La référence portant le code siteXML ;
9. Tous les deuxièmes auteurs (avec et sans la fonction position, puis sans aucune fonction).

Solution de l'exemple pratique 5.4.1 :

1. for \$x in /Cours
return \$x
2. for \$x in /Cours
return /Cours/child : :*
3. for \$x in doc("document XML bien formé.xml")//*[@*]
return \$x
4. for \$x in /Cours
return //*/text()
5. for \$x in doc("document XML bien formé.xml")/processing-instruction()
return \$x
6. for \$b in //p/text()
return /\$b
7. for \$b in //p
return \$b[child : :important]
8. for \$b in /Cours/références
return \$b/référence[@code="siteXML"]
9. **1-Avec Position**
for \$x in /Cours/références/référence
return \$x/auteur[position()=2]

2-Sans Position
for \$x in //*
return \$x/auteur[count(preceding : :auteur)=1]

3-Sans aucune fonction
for \$x in /Cours return \$x//auteur[(preceding : :auteur)]

5.5 Exercices ouverts

Exercice 5.1

A partir du document XML (XML_2022.xml) ci-dessous, donnez les résultats des requêtes XQuery suivantes : ■

<pre><?xml version="1.0" encoding="UTF-8"?> <a> <a> <c><d/></c> <d><a/><c/></d> <a><c/> <c><d/><a/></c> <c/><d/> <a><c/> <a> <c><d/><c/></c> <d><a/></d> <a><c/> <d> <a><c/> <a/><d/> <c/><d/> </d> </pre>	<ol style="list-style-type: none"> 1- for \$q in let \$a:= doc("XML_2022.xml")//following-sibling::* return \$a//a/b[last()<2] return \$q 2- for \$q in doc("XML_2022.xml")//following-sibling::*/child::* return \$q//ancestor-or-self::*/child::c/b 3- for \$q in doc("XML_2022.xml")//descendant-or-self::* return \$q//a/d 4- for \$q in doc("XML_2022.xml")// following-sibling::* return \$q//b/c 5- for \$q in doc("XML_2022.xml")//self::*/ child::* return \$q//b/c 6- for \$q in doc("XML_2022.xml")//self::* return \$q//b/c <p style="text-align: right; font-size: small;">Activer Windows</p>
--	--

Exercice 5.2 Soient les deux documents XML, **doc1.xml** et **doc2.xml** ci-dessous :

```
<?xml version="1.0" encoding="UTF-8"?>
<A>
  <B/>
  <B>
    <C>2MGL</C>
    <C>1MGL</C>
  </B>
  <D>
    <E>2MGI</E>
  </D>
</A>
<!-- doc1.xml -->
```

```
<?xml version="1.0" encoding="UTF-8"?>
<A>
  <B>
    <C>1MGI</C>
  </B>
  <D>
    <E>1MRT</E>
    <E>2MRT</E>
  </D>
  <D/>
  <E>1LMD</E>
</A>
<!-- doc2.xml -->
```

Donnez les résultats des requêtes XQuery suivantes :

— **Requête_1 :**

```
for $x in let $a:= doc("doc1.xml")//A
return $a//D
return $x
```

— **Requête_2 :**

```
for $p in doc("doc1.xml")//A
return $p/*/*
```

— **Requête_3 :**

```
for $x in let $a:= doc("doc2.xml")//A/*
return $a/E
return $x
```

— **Requête_4 :**

```
for $p in doc("doc2.xml")//A
return $p/*
```


5.6 Conclusion

Comme nous l'avons déjà indiqué dans ce chapitre, il existe une relation forte et intrinsèque entre les bases de données semi structurées (bases de données XML) et le langage de requête XQuery. L'une des principales raisons pour lesquelles XQuery a émergé aux côtés de XML était de permettre des requêtes sur des corpus de documents XML. XQuery fournit une syntaxe simple que vous pouvez utiliser pour apporter des modifications simples ou complexes à vos documents XML. Si vous travaillez avec des ensembles de données semi-structurés, XQuery sera un outil indispensable. La plupart des processeurs XQuery (mais pas tous) sont intégrés dans des bases de données XML, nous aurions donc aimé savoir comment stocker, interroger, récupérer et modifier des documents dans des bases de données XML. De plus, au fur et à mesure que vous développez des expressions de mise à jour plus complexes, vous devrez probablement également revoir les règles de mise en scène et de validation de plusieurs mises à jour. Il est également très probable que votre processeur XQuery offrira un ensemble étendu d'outils pour la mutation des données. Vous voudrez également apprendre toutes les fonctions et tous les outils spécifiques au processeur. N'oubliez pas de tester soigneusement toutes les expressions de mise à jour avant de les déployer. C'est la partie de XQuery où vous voulez éviter de faire des erreurs !

Bibliographie

Livres

- [All02] Andrew Watt ALLEN WYKE. *XML Schema Essentials*. 1^{re} édition. 9780471423201. Wiley, 2002 (cf. pages 45, 54).
- [Bec22] Margit BECHER. *XML : DTD, XML-Schema, XPath, XQuery, XSL-FO, SAX, DOM*. 1^{re} édition. 3658354348. Springer Vieweg, 2022 (cf. pages 73, 113).
- [Cli20] Joseph C. Wicentowski CLIFFORD B. ANDERSON. *XQuery for Humanists*. 1^{re} édition. 1623498295. Texas AM University Press, 2020 (cf. pages 105, 113).
- [GN07] Chagnon GILLES et Florent NOLOT. *XML : Synthèse de cours & exercices corrigés*. 2^e édition. France : Pearson Education, 2007 (cf. page 24).
- [Har04] Elliott Rusty HAROLD. *XML 1.1 Bible*. 3^e édition. 9780764569302. Microsoft Press, 2004 (cf. pages 10, 31, 54).
- [Hos13] Dorothy J. HOSKINS. *XML and InDesign*. 1^{re} édition. 9781449344160. O'Reilly Media, 2013 (cf. page 30).
- [Joe12] Liam R. E. Quin JOE FAWCETT Danny Ayers. *Beginning XML*. 5^e édition. 1118162137. Wrox, 2012 (cf. page 14).
- [Kah09] Atul KAHATE. *XML and Related Technologies*. 2^e édition. 9788131718650. Pearson Education, 2009 (cf. pages 30, 40).
- [Kay08] Michael KAY. *XSLT 2.0 and XPath 2.0 : programmer's reference*. 4^e édition. 9780470337523. Wiley Pub, 2008 (cf. page 89).
- [Ste03] STEVEHOLZNER. *Real World XML*. 1^{re} édition. Nw Ridrs Publications, 2003 (cf. pages 40, 45, 54).

- [Tid09] Doug TIDWELL. *XSLT*. 1^{re} édition. 9780596159528. O'Reilly Media, 2009 (cf. page 86).
- [Wal12] Priscilla WALMSLEY. *Definitive XML Schema*. 2^e édition. 9782147483649. Pearson Education Limited (US titles); Prentice Hall, 2012 (cf. page 38).
- [You02] Michael J. YOUNG. *XML Step By Step*. 2^e édition. Redmond, Washington 98052-6399 : Microsoft Press, 2002, pages 3-21 (cf. page 10).

Articles

- [Ama09] AMANN. « XSL and XSLT ». In : (2009), https://doi.org/10.1007/978-0-387-39940-9_773 (cf. page 86).
- [Ari08] L. Ahmedi ; M. ARIFAJ. « Processing XPath/XQuery to be Aware of XLink Hyperlinks ». In : ISBN : 978-960-474-002-4 (sept. 2008), pages 217-221 (cf. page 105).
- [Bou21] Zouhaier Brahmia ; Fabio Grandi ; Rafik BOUAZIZ. « Managing changes to XML schema design styles in a temporal and multi-version XML environment ». In : 15.4 (2021), DOI : 10.1504/IJWET.2020.114028 (cf. page 38).
- [Hon21] Rongxin Chen ; Zongyue Wang ; Yuling HONG. « Pipelined XPath Query Based on Cost Optimization, Scientific Programming ». In : 5559941 (2021), <https://doi.org/10.1155/2021/5559941> (cf. page 73).
- [Pet12] D PETKOVIĆ. « XPath and XQuery Full Text Standard and Its Support in RDBMSs ». In : 12. <https://doi.org/10.1007/s13222-012-0097-3> (2012), pages 205-213 (cf. page 105).