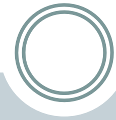
XSL : XPATH – XSLT



L3 DIST – 2015-2016
LIONEL MÉDINI

Université Claude Bernard Lyon 1
UFR FST – département Informatique

D'après le cours de Yannick Prié

XSL : Extensible Stylesheet Language

- Famille de langages pour définir des transformation et des présentations de documents XML
- Trois parties
  - **Xpath**
    - langage pour désigner des informations dans un arbre XML sous la forme de chemins (paths)
  - **XSLT**
    - langage de description de transformations à opérer sur un arbre XML
      - transcodage d'un document XML vers un autre document XML
  - XSL Formatting Objects (XSL-FO)
    - Langage de spécification de formatages (pour construire des formes physiques de présentation)

XPath

- Recommandation W3C
- Versions
  - Xpath 1.0 : 16/11/1999
  - Xpath2.0 : 14/12/2010
- Objectif :
  - localiser des documents / identifier des sous-structures dans ceux-ci
- Utilisé par d'autres spécifications XML
  - XPointer, XQuery, XSLT…

Contexte et éléments XML

- La signification d’un élément peut dépendre de son contexte
  - `<book><title>…</title></book>`
    `<person><title>…</title></person>`
- Supposons que l’on cherche le titre d’un livre, pas le titre d’une personne
- Idée
  - exploiter le contexte séquentiel et hiérarchique de XML pour spécifier des éléments par leur contexte (*i.e.* leur position dans la hiérarchie)
  - exemple : `book/title` ≠ `person/title`

Xpath : principe général

- Décrire un modèle de chemin dans un arbre XML
  → **expression**
- Récupérer les nœuds qui répondent à ce chemin en utilisant l'expression
  → résultat de l'**application de l'expression à l'arbre XML**
- Une expression sera utilisée et appliquée au sein de différentes syntaxes
  - URL : **http://abc.com/getQuery?/book/intro/title**
  - XSL : `<xsl:pattern match="chapter/title">…</xsl:pattern>`
  - Xpointer :

```
<link href="./doc.xml#xptr(book/intro/title)">
  Link to introductory title
</link>
<!-- la valeur de l'attribut href est celle de l'élément title
situé dans l'élément intro, situé dans l'élément book, éléments
qui se trouvent dans le fichier doc.xml, lui-même situé dans le
dossier où se trouve le fichier XML en cours -->
```

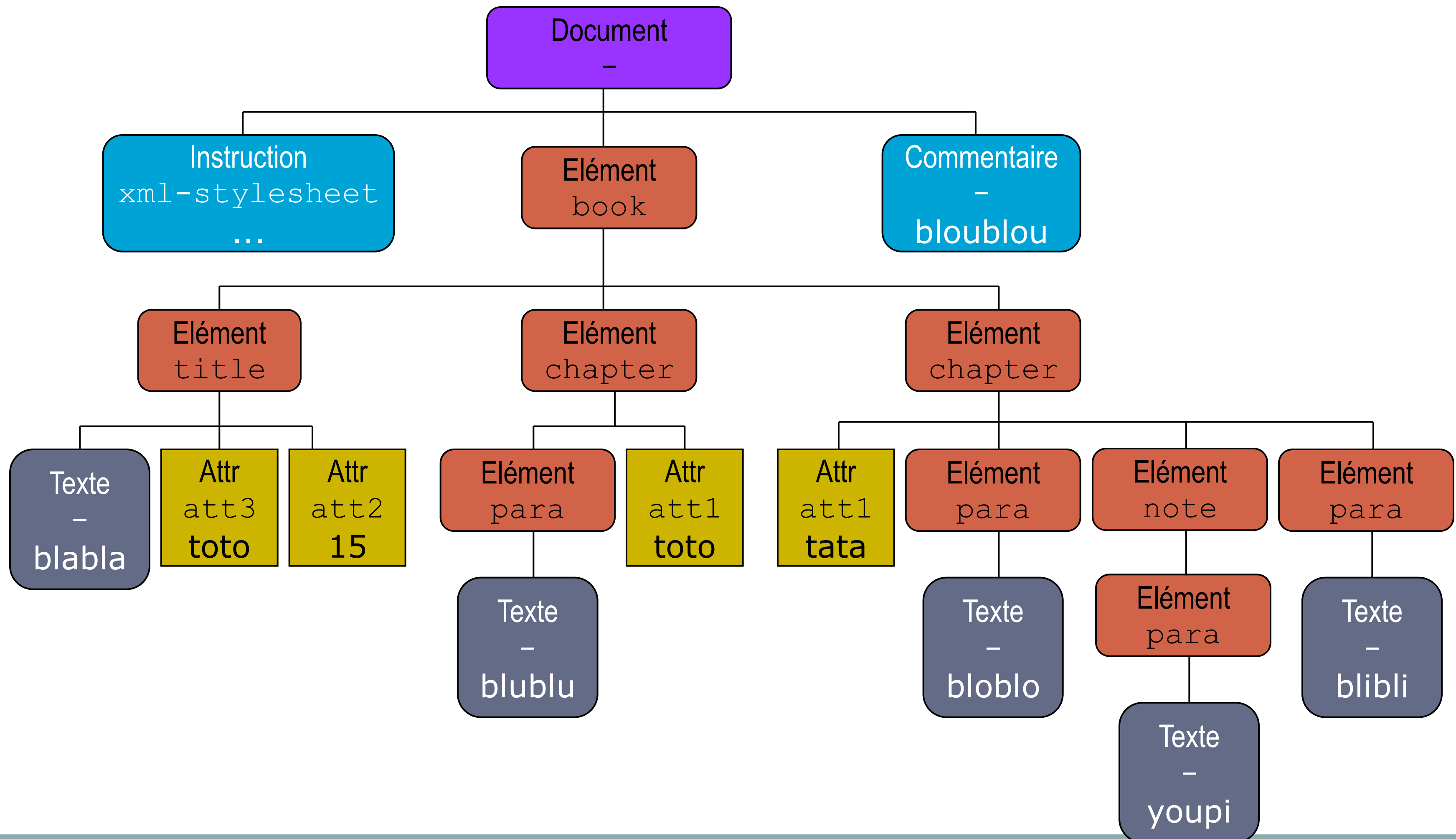
Document/arbre/nœuds Xpath

- Dans XML
  - arbre XML = élément XML
- Dans Xpath
  - arbre XPATH = arbre avec toutes les informations repérables dans un document XML:
    - noeuds éléments (= nœud XML)
    - noeud racine (représente tout le doc XML)
    - noeuds attributs
    - noeuds textes
    - noeuds instructions de traitement
    - noeuds commentaires
    - (noeuds espaces de nom)

Version XML


```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="fichier.xsl"
  type="text/xsl"?>
<book>
<title att3="toto" att2="15">blabla</title>
<chapter att1="toto">
<para>blublu</para>
</chapter>
<chapter att1="tata">
<para>bloblo</para>
<note>
<para>youpi</para>
</note>
<para>blibli</para>
</chapter>
</book>
<!-- bloublou -->
```

Exemple de référence



Chemins de localisation

- Les expressions identifient des noeuds par leur position dans la hiérarchie
- Permet de
  - monter/descendre dans la hiérarchie de l'arbre XML
  - aller voir les voisins (frères) d'un noeud
  - *en fait : suivre des axes*
- Un chemin peut être
  - relatif
    - à partir de l'endroit où l'on est
  - absolu
    - à partir de la racine

Chemins relatifs

- On se place dans le contexte d’un nœud
- A partir de là, on explore l’arbre XML, et on garde les nœuds qui vérifient l’expression
- Exemple
  - **`para`** (ou **`child::para`**) sélectionnera les fils du nœud courant qui ont le nom '**`para`**'

```
<chapter>                    <!– Noeud courant -->
  <para>…</para>             <!-- Sélectionné -->
  <note>
    <para>…</para>           <!– Non sélectionné -->
  </note>
  <para>…</para>             <!-- Sélectionné -->
</chapter>
```

Chemins absolus

- Expression identique aux chemins relatifs, mais
  - tout chemin absolu commence par '**/**'
  - signifie qu'on part de l'élément racine
- Exemple
  - Trouver tous les éléments '**para**' dans un '**chapter**'
    - **/book/chapter/para**

```
<book>                          <!-- racine -->
    <chapter>
      <para>...</para>          <!-- Sélectionné -->
      <note>
        <para>...</para>
      </note>
      <para>...</para>          <!-- Sélectionné -->
    </chapter>
    <chapter>
      <para>...</para>          <!-- Sélectionné -->
    </chapter>
 <book>
```

Chemins à plusieurs étapes

- Séparer les étapes par des '`/`'
- Exemple
  - **`book/title`** (version courte)
  - **`child::book/child::title`** (version longue)
  - depuis le noeud courant, on sélectionne d'abord `book`, qui devient le contexte courant, puis on sélectionne `title`

Notion d'étape Xpath

- Une étape contient trois composants

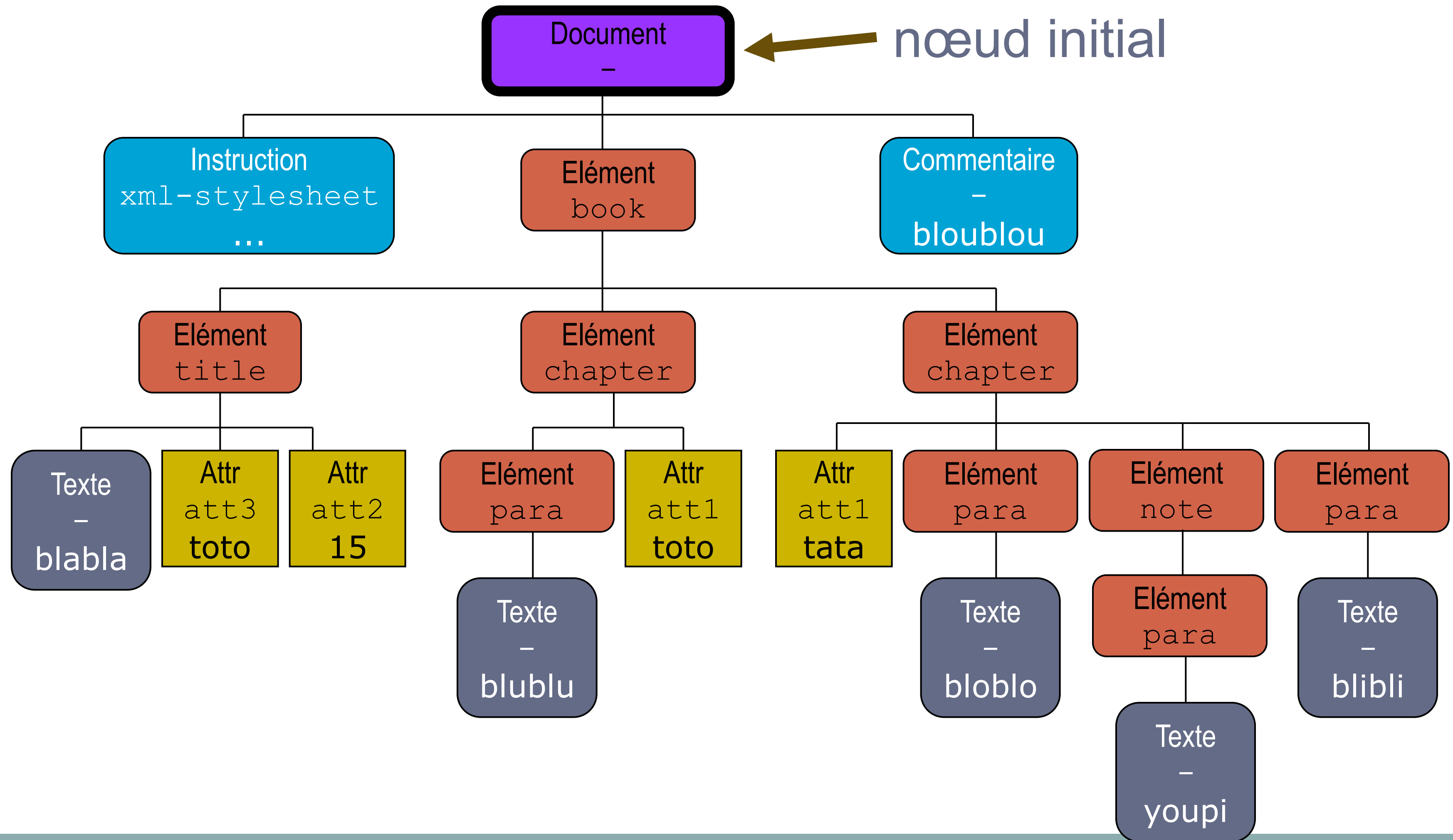
  `Axe :: Filtre [ Prédicat ]`

  - axe
    - sens de parcours des nœuds
  - filtre
    - type des nœuds retenus
  - prédicats
    - propriétés satisfaites par les nœuds retenus
      - on peut enchaîner les prédicats
- Exemple
  - **`child :: chapter [@att1 = "toto"]`**
- Remarques
  - il existe une syntaxe bavarde (verbose) et une syntaxe raccourcie, plus pratique
  - possibilité de multiples expressions séparées par '|'
    - équivalent d'un OU

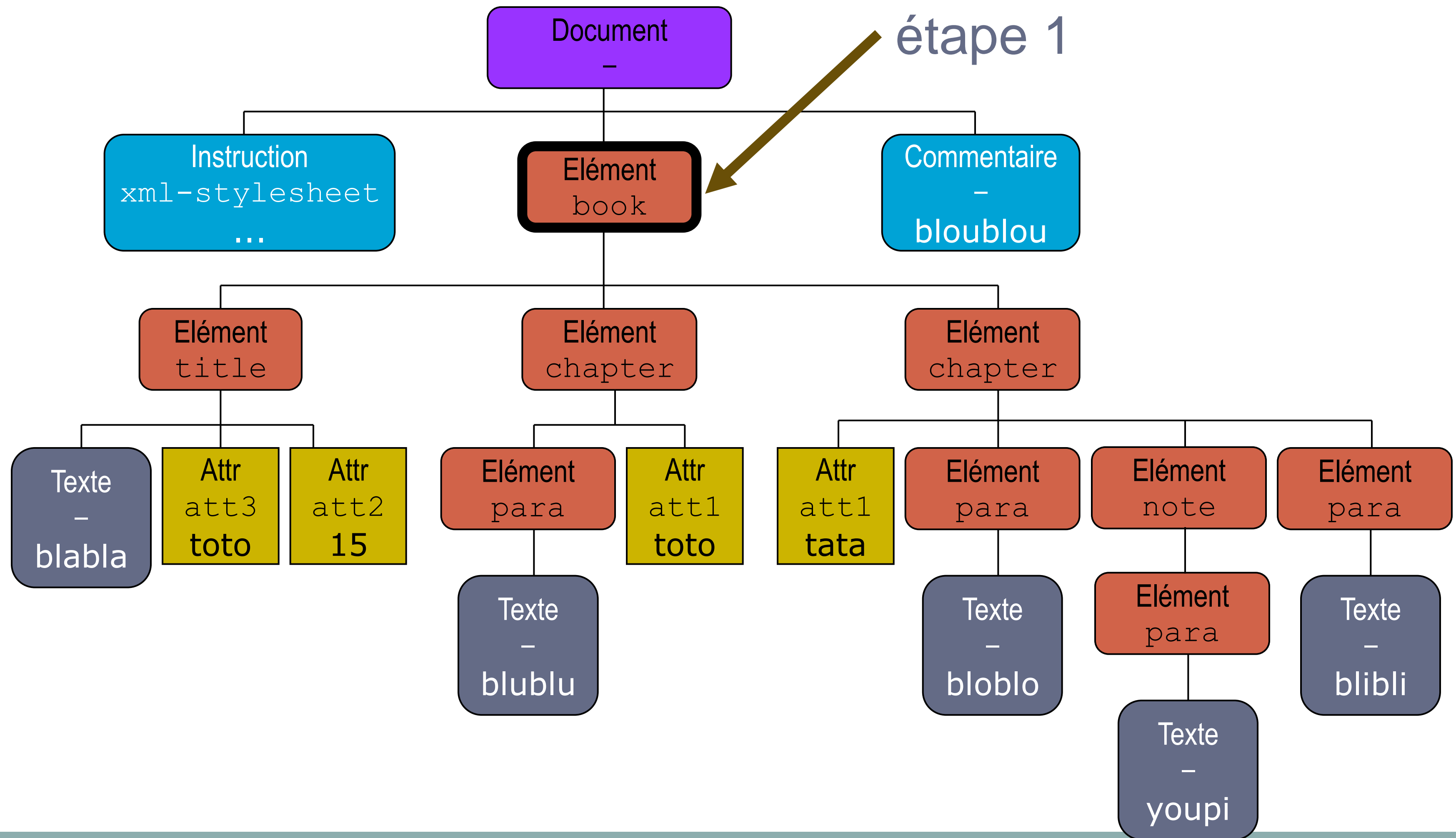
Evaluation d'une expression Xpath

- Expression = séquence d'étapes
- On part du nœud contexte (ou de la racine)
  - on évalue l'étape 1
  - on récupère un ensemble de nœuds
  - pour chacun de ces nœuds
    - il devient le nœud contexte
      - on évalue l'étape 2
      - on récupère un ensemble de nœuds
        - pour chacun de ces nœuds
        - …

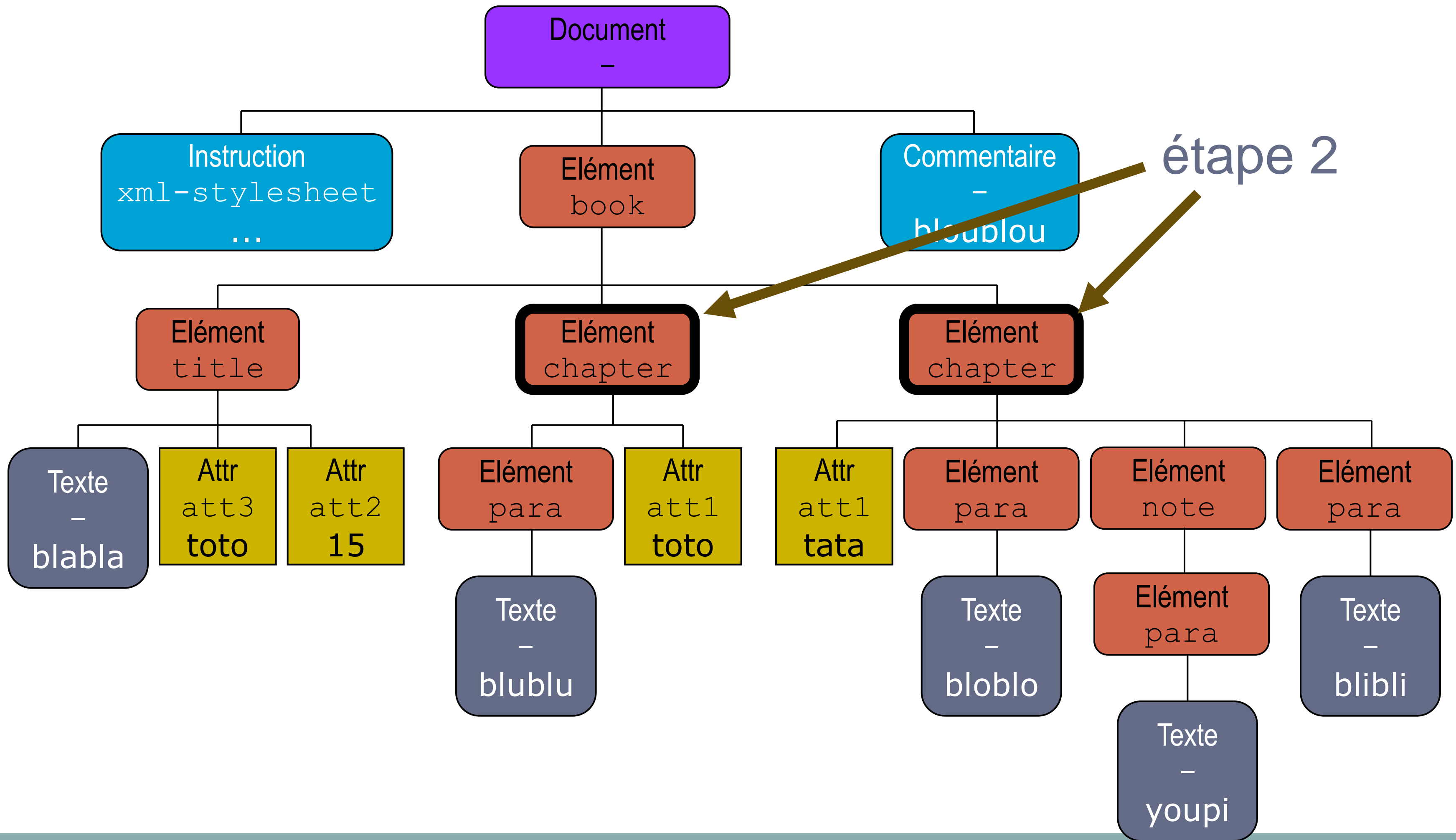
/child::book/child::chapter/attribute::att1
/book/chapter/@att1 (expression raccourcie)



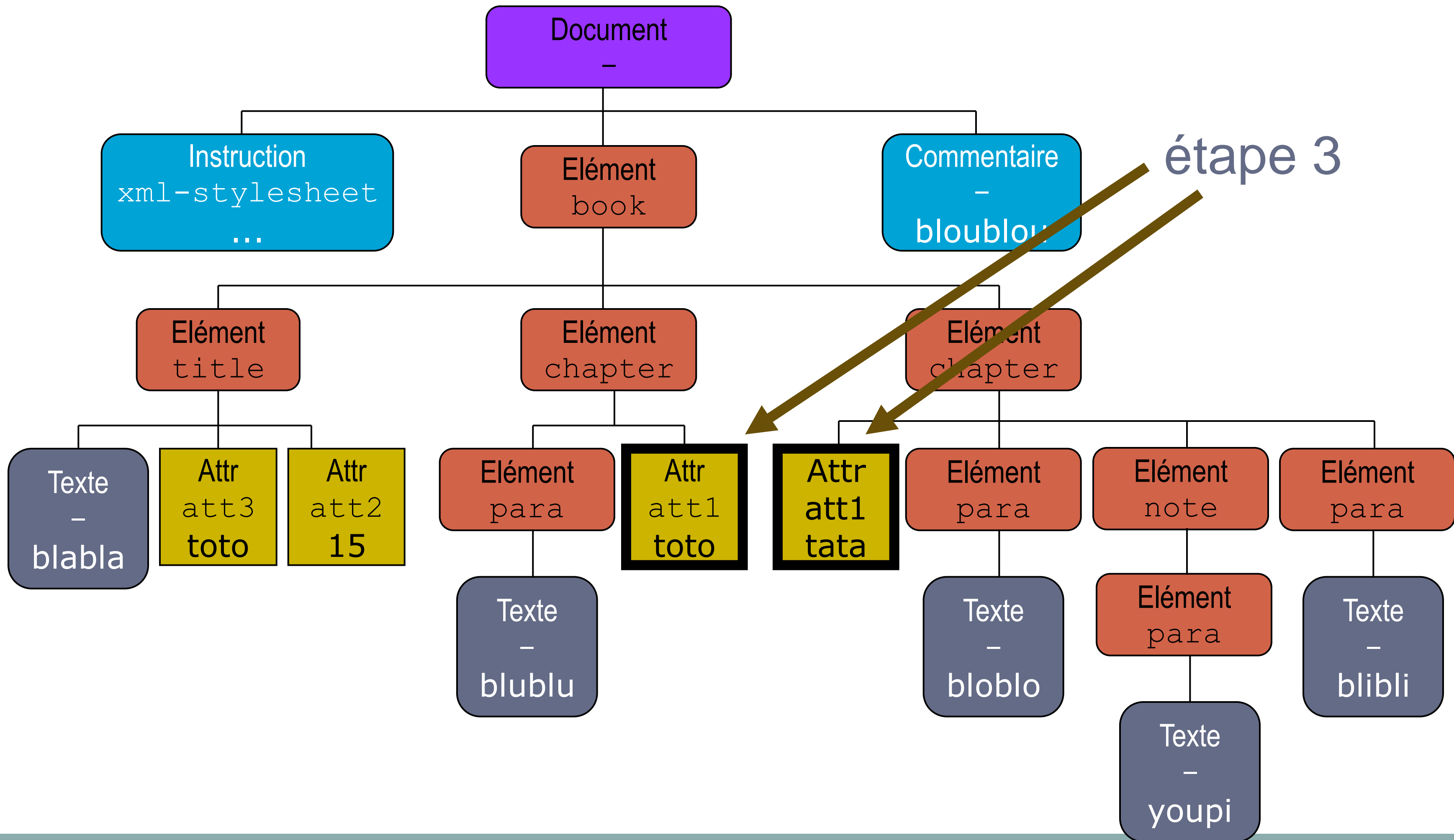
/child::book/child::chapter/attribute::att1
/book/chapter/@att1 (expression raccourcie)



/child::book/child::chapter/attribute::att1
/book/chapter/@att1 (expression raccourcie)



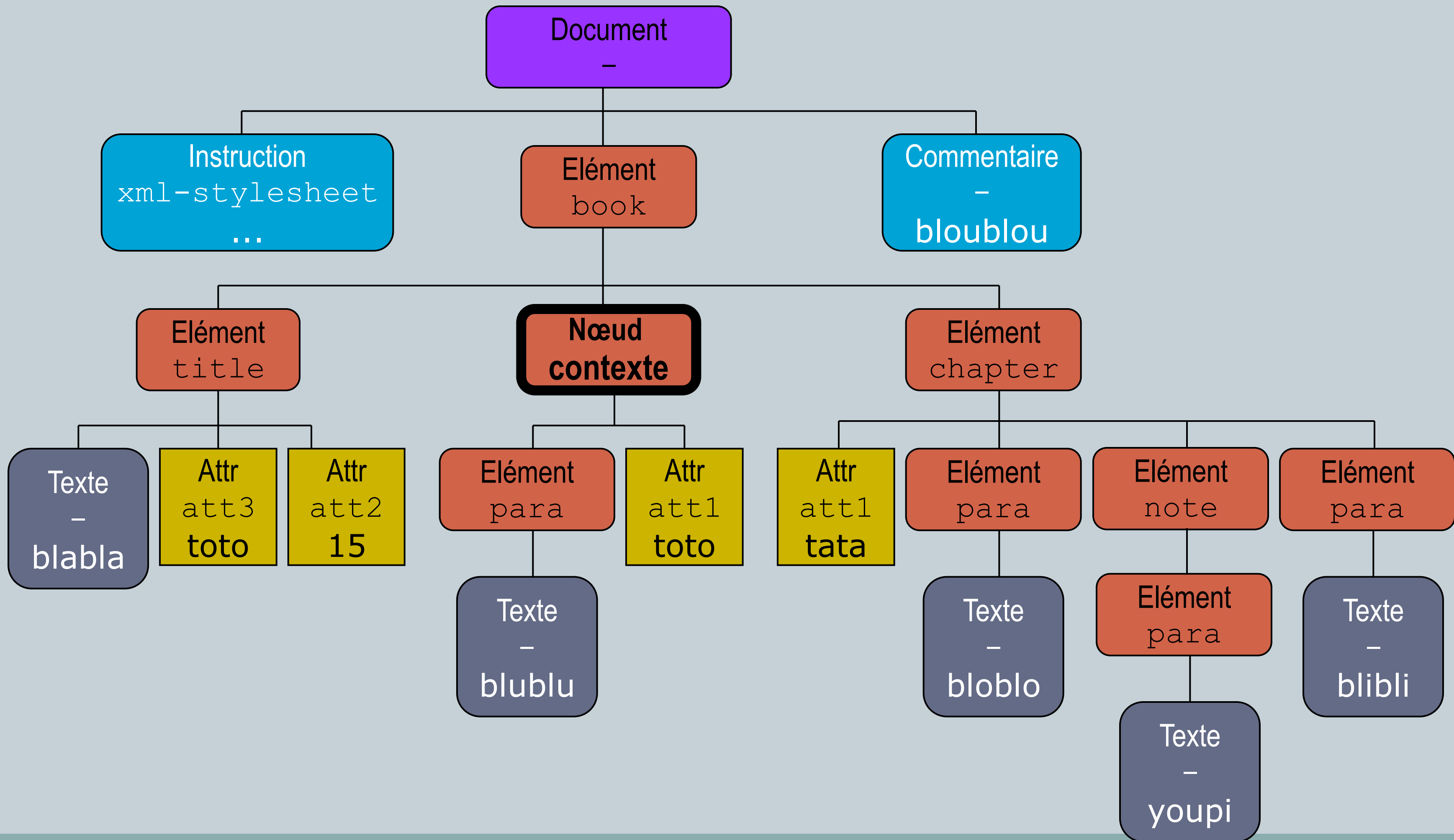
/child::book/child::chapter/attribute::att1
/book/chapter/@att1 (expression raccourcie)



Axes : directions à suivre

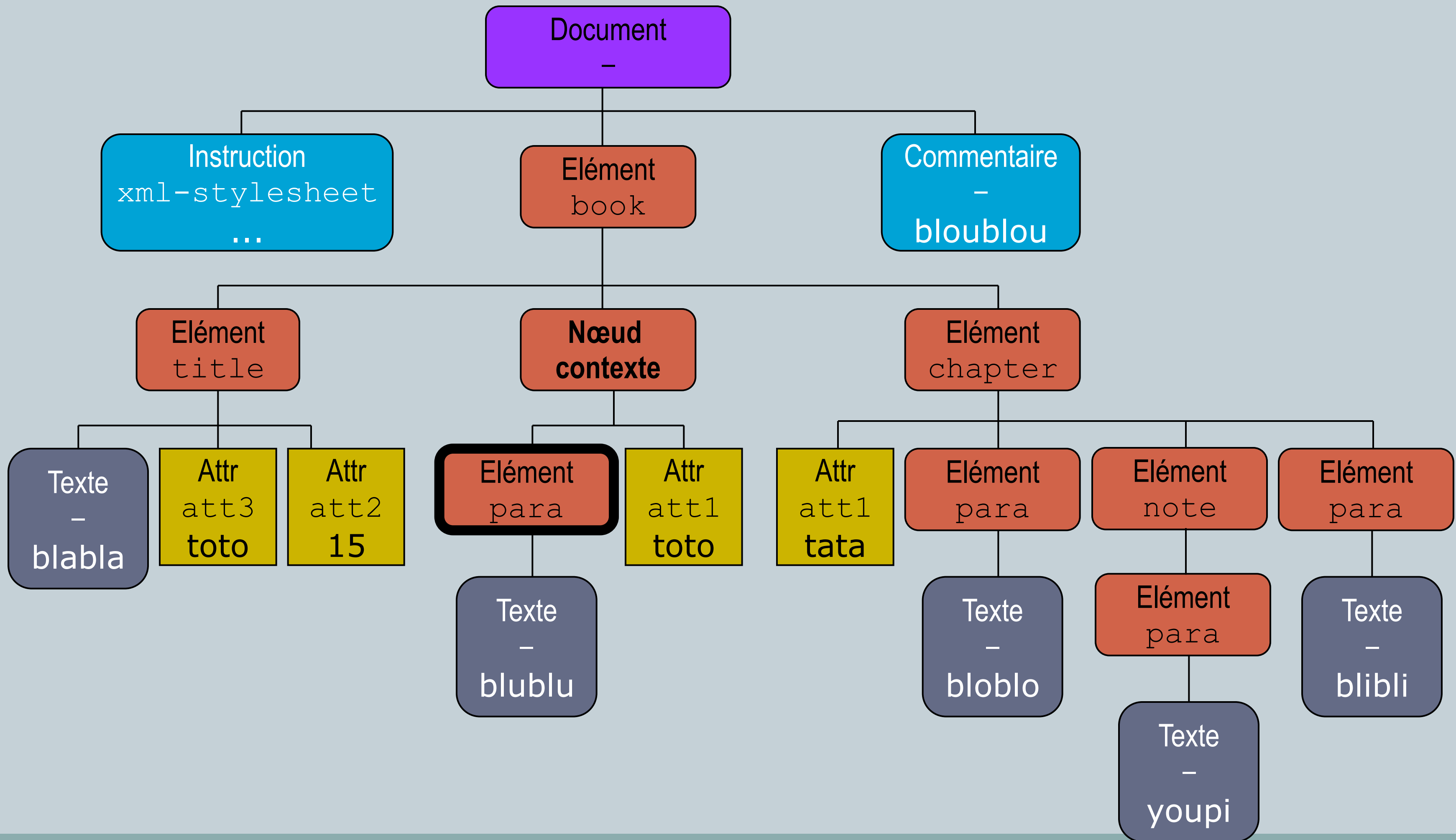
- self:: (abrégé : .)
- child:: (abrégé : *rien*)
- attribute:: (abrégé :@)
- parent:: (abrégé : ..)
- descendant::
- descendant-or-self:: (abrégé : //)
- ancestor::
- ancestor-or-self::
- following::
- following-sibling::
- preceding::
- preceding-sibling::

Axe self : `self::*`



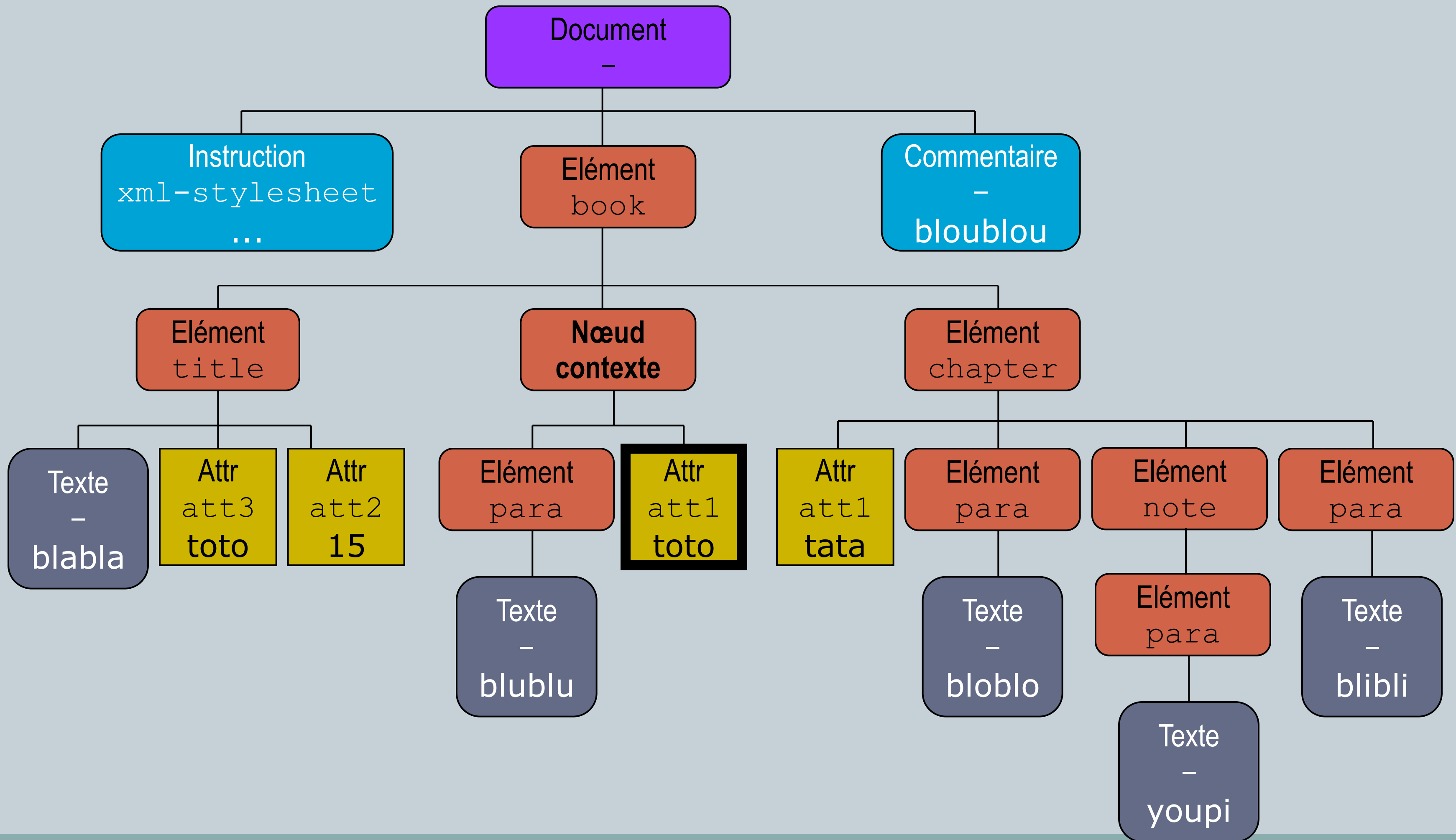
Axe :: Filtre [Prédicat]

Axe child : `child::*`

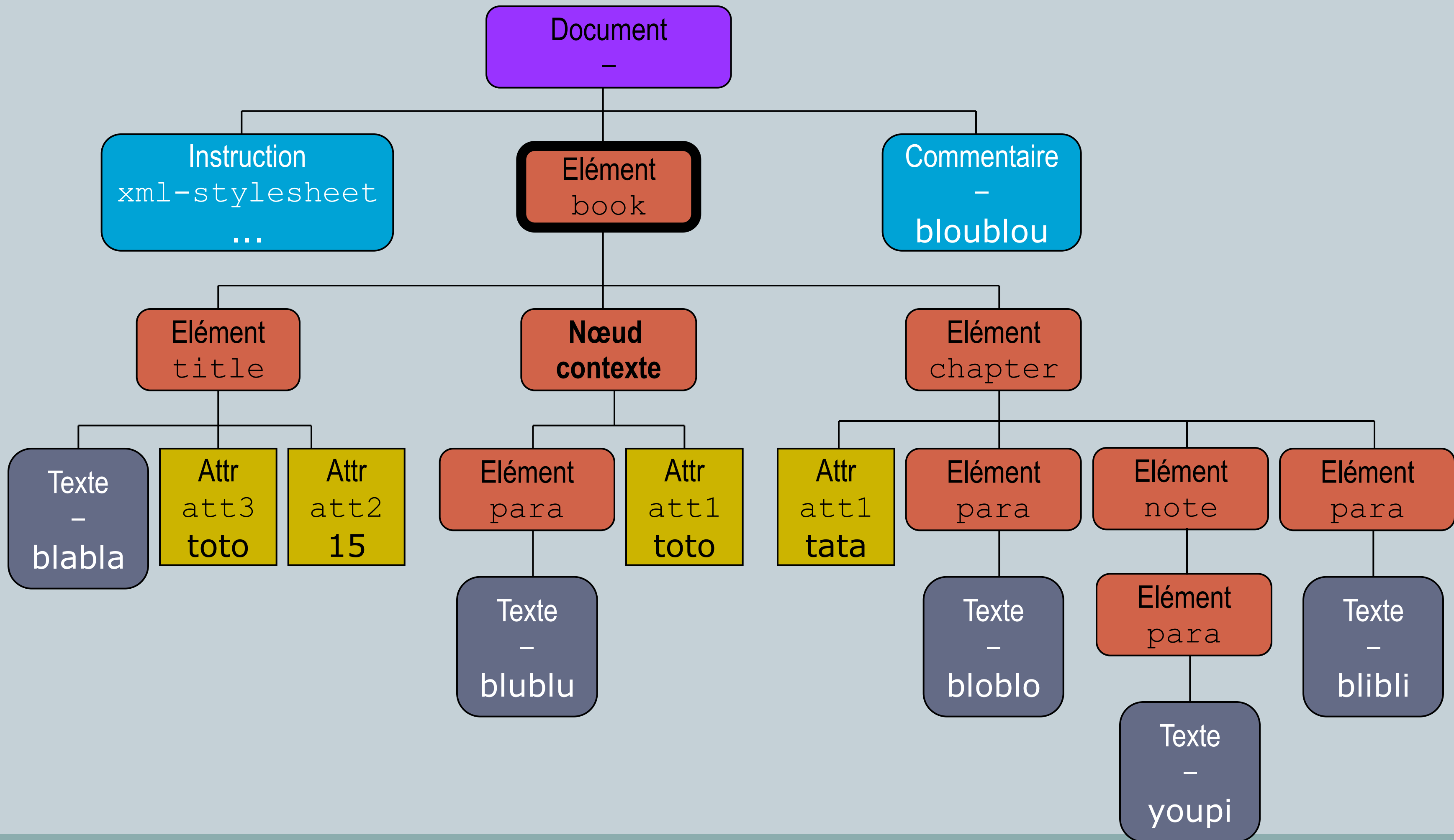


<u>**Axe**</u> **:: Filtre [Prédicat]**

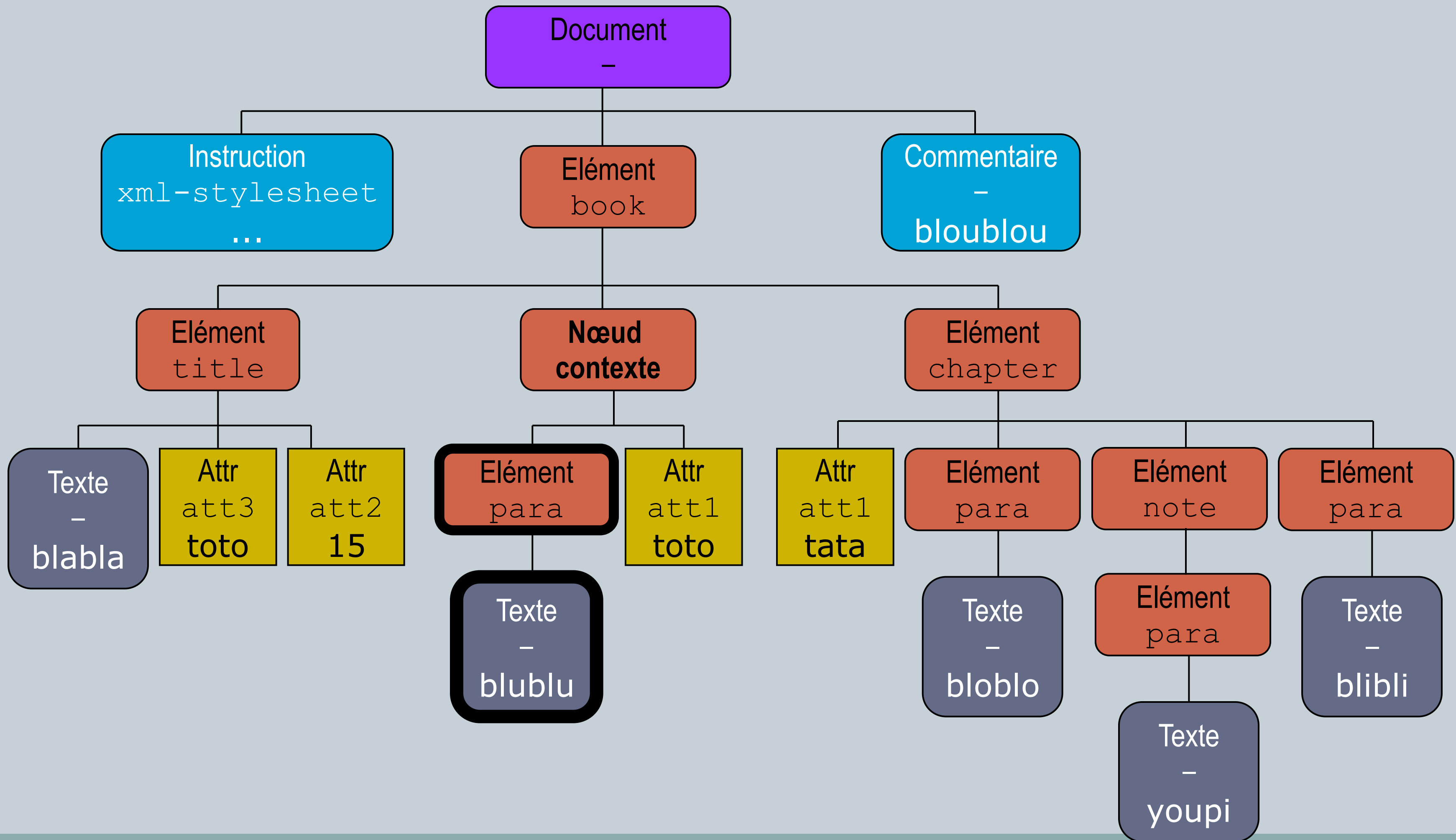
Axe attribute : `attribute::*`



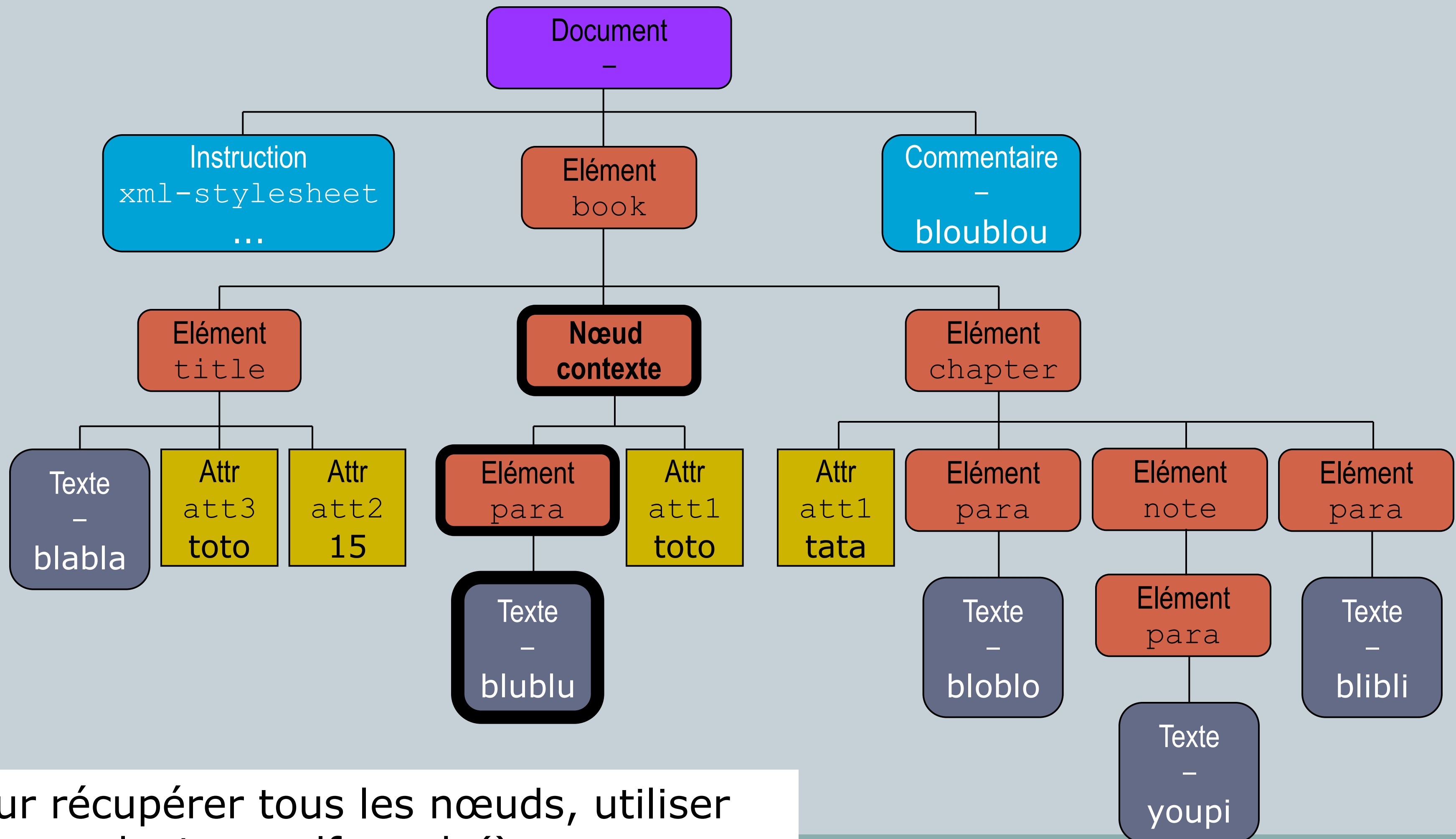
Axe parent : `parent::*`



Axe descendant : `descendant::*`



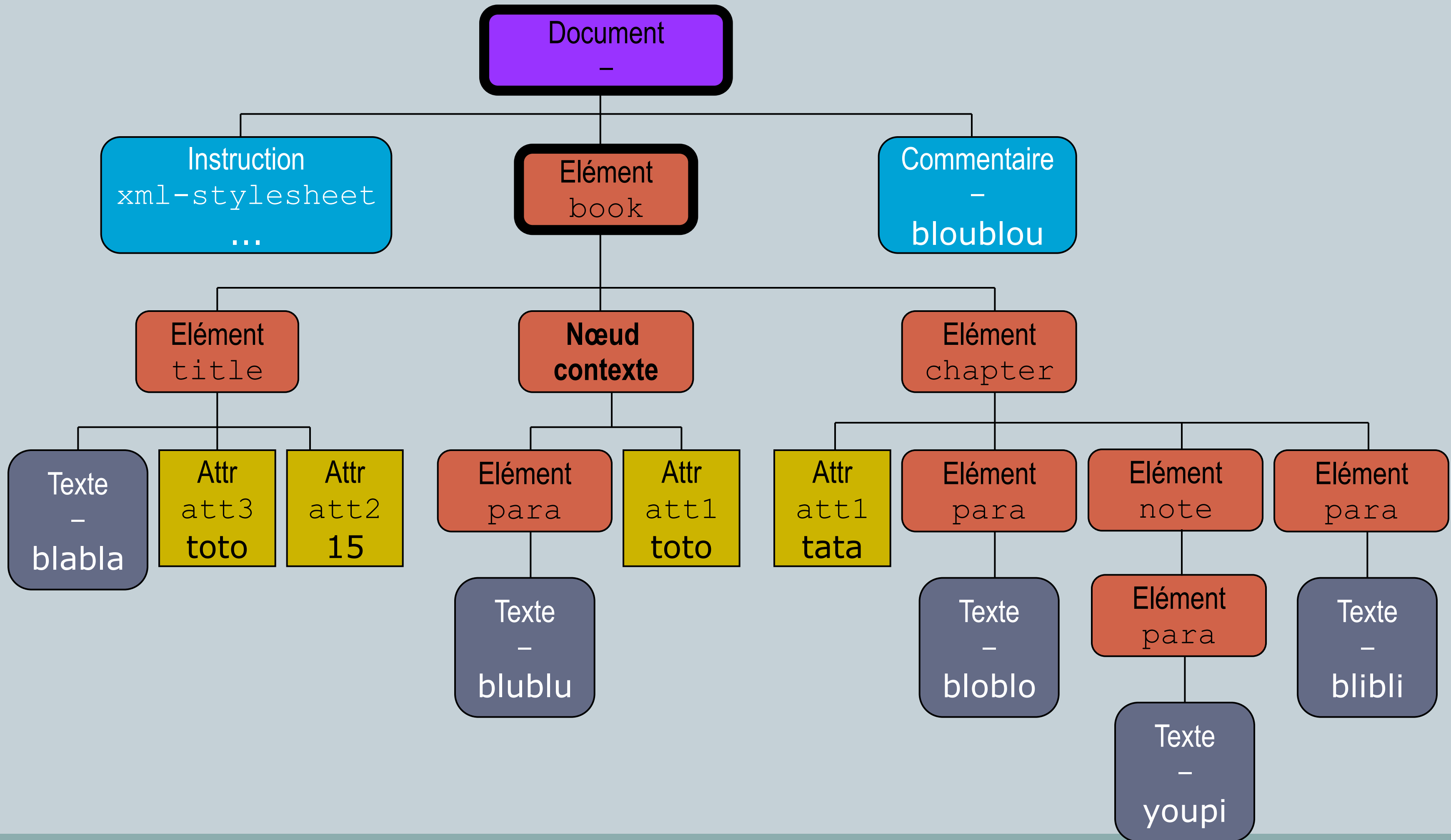
Axe descendant-or-self : `descendant-or-self::*`



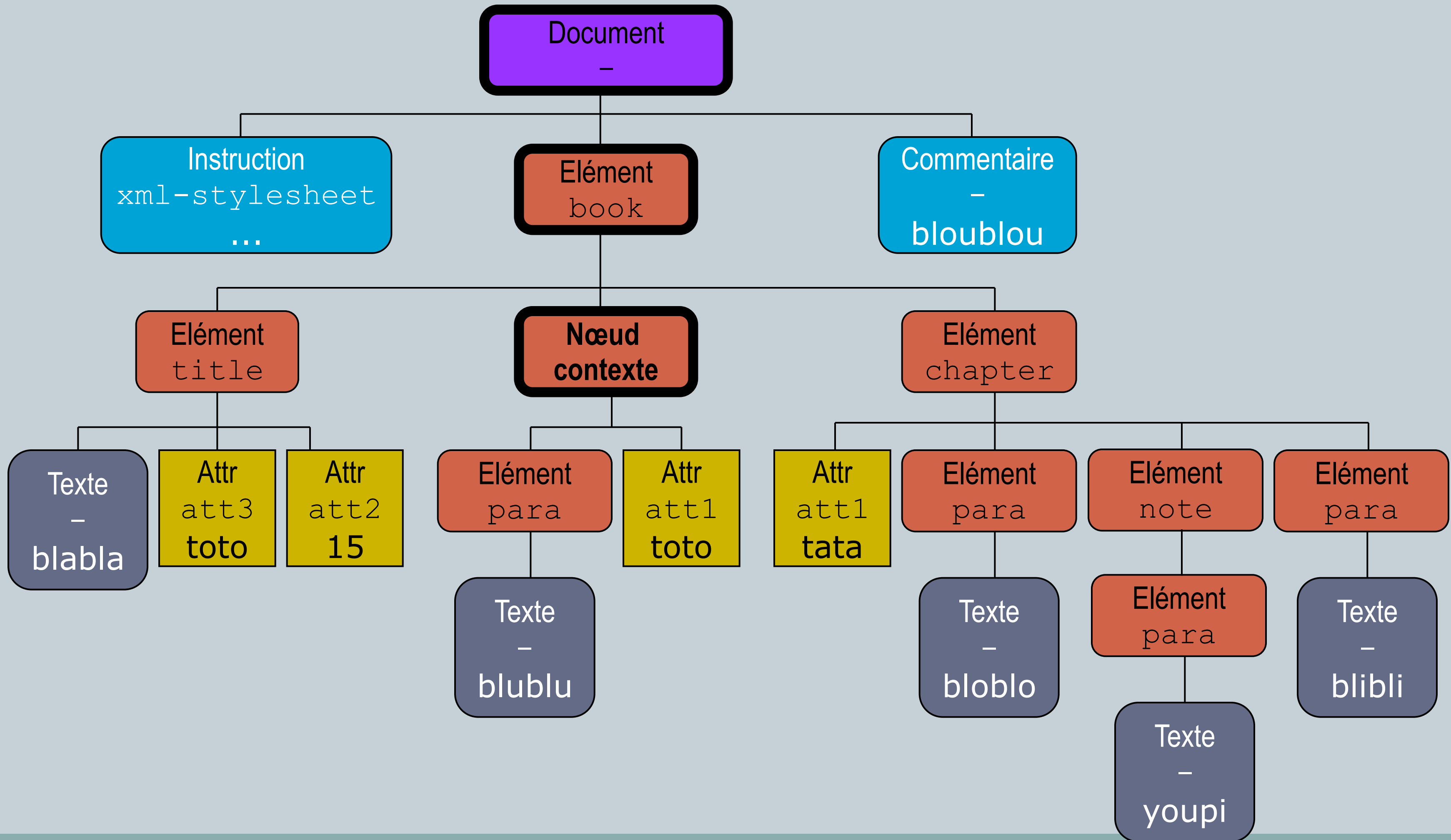
Pour récupérer tous les nœuds, utiliser /descendant-or-self::node()

<u>**Axe**</u> **:: Filtre [Prédicat]**

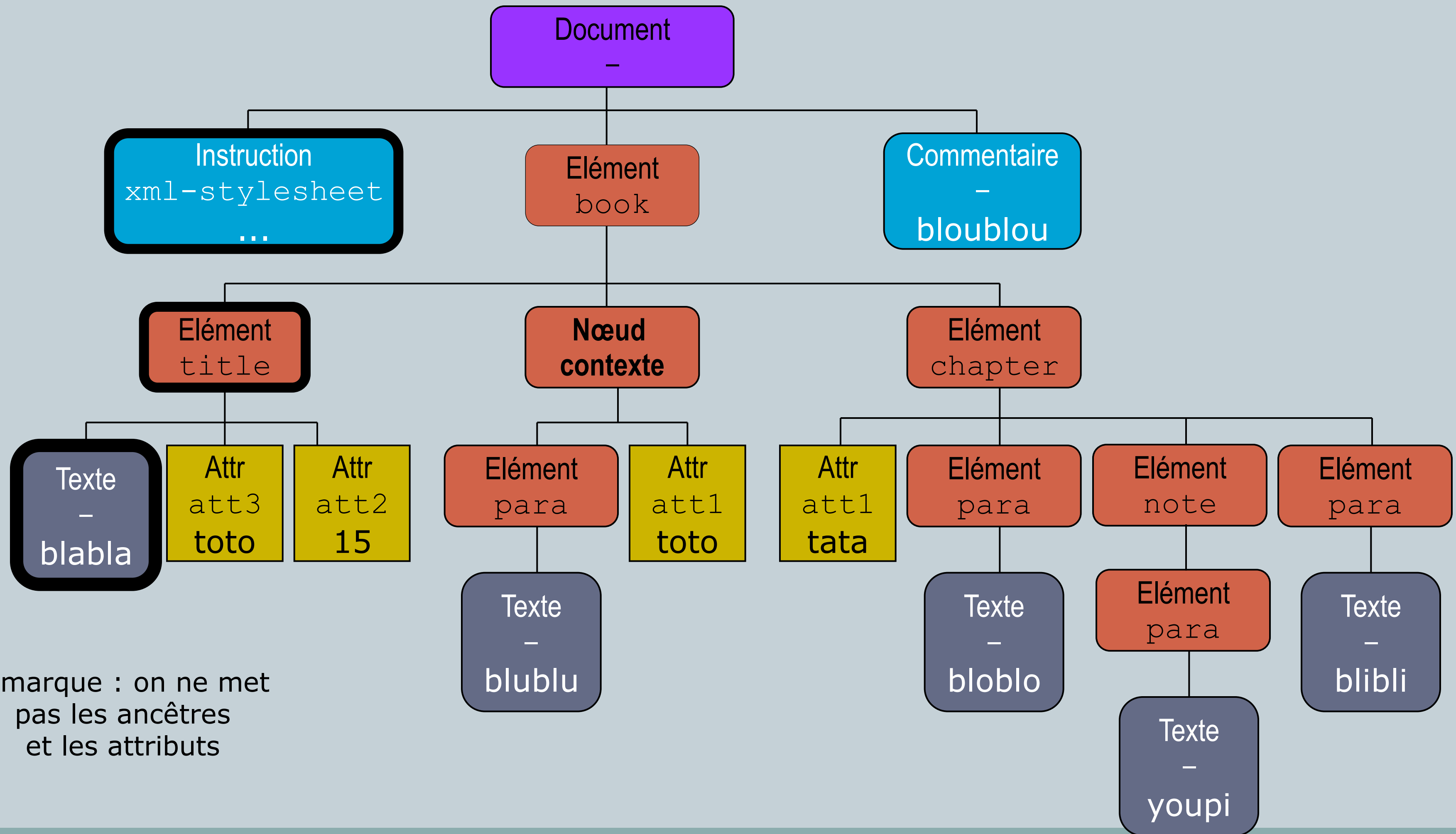
Axe ancestor : `ancestor::*`



Axe ancestor-or-self : `ancestor-or-self::*`



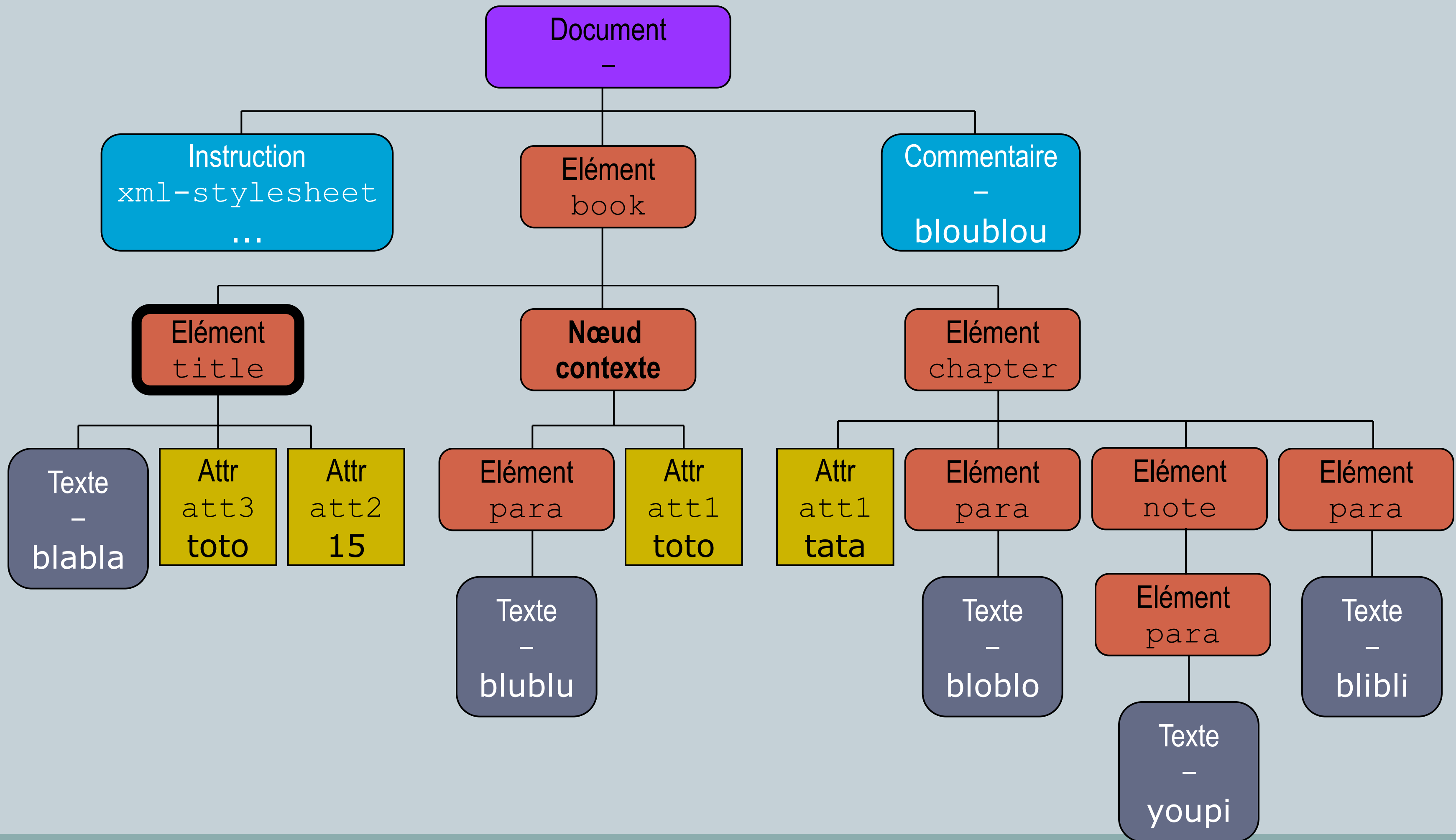
Axe preceding : `preceding::*`



Remarque : on ne met
pas les ancêtres
et les attributs

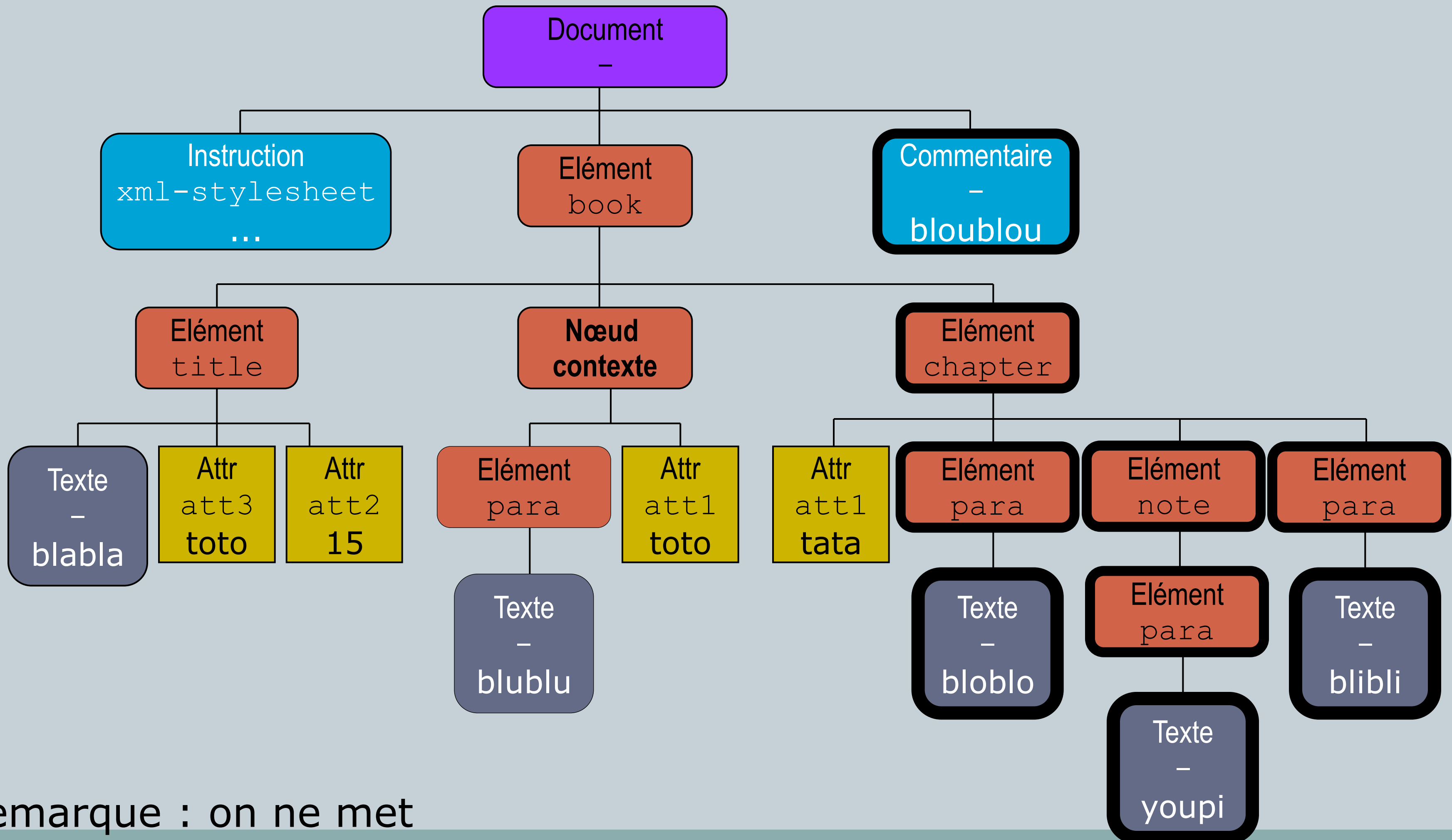
Axe :: Filtre [Prédicat]

Axe preceding-sibling : `preceding-sibling::*`



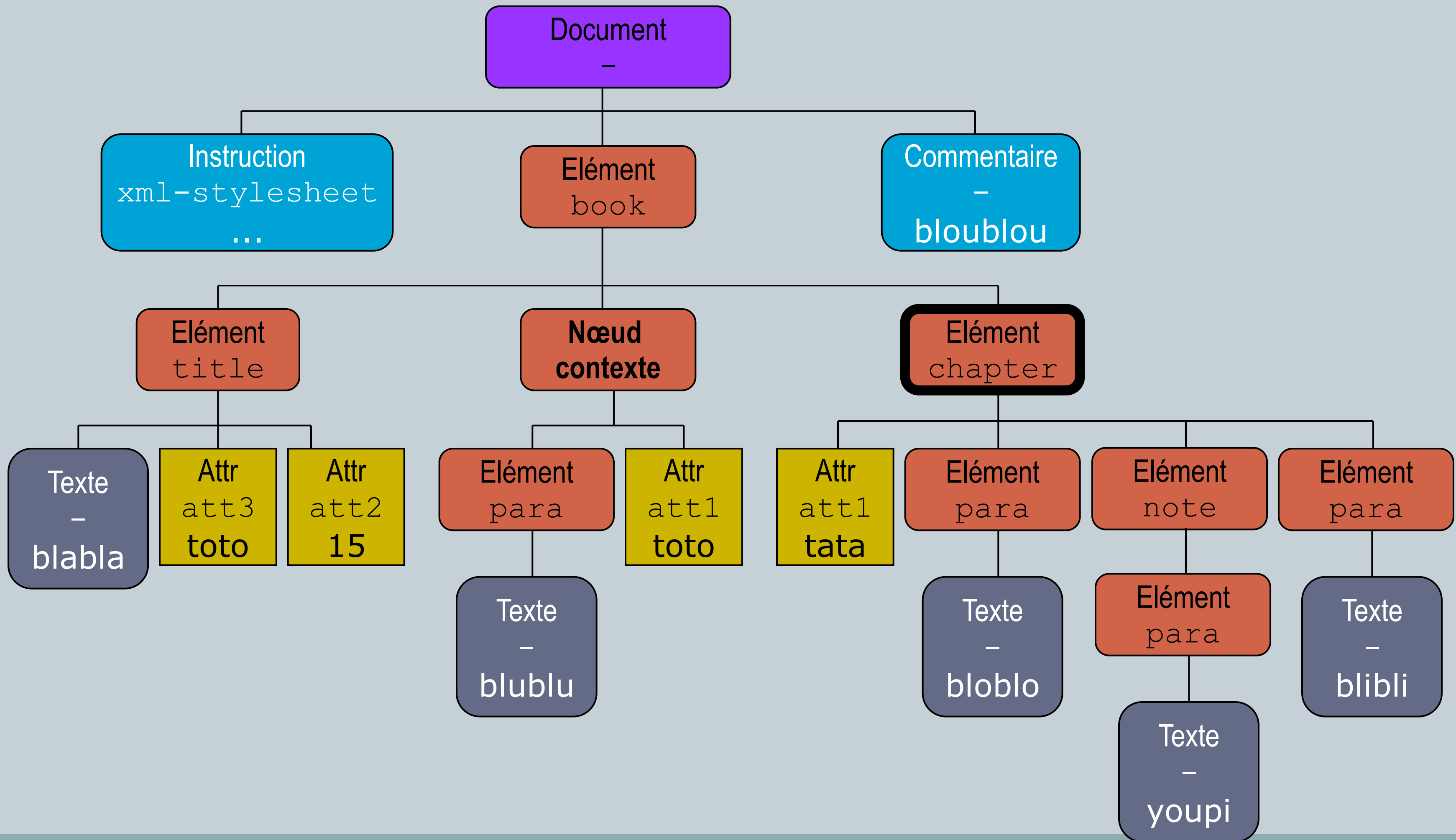
<u>Axe</u> :: Filtre [Prédicat]

Axe following : `following::*`



Remarque : on ne met
ni les descendants ni les attributs

Axe following-sibling : `following-sibling::*`



Filtres

- Filtrage par le type
  - Éléments
  - Attributs
  - Instructions de traitement
- Filtrage par le nom

Filtrage par le type

- node()
  - garde tout nœud
- text()
  - noeud gardé si textuel
- comment()
  - noeud gardé si commentaire
- processing-instruction()
  - noeud gardé si instruction de traitement

Filtrage par le nom

- Nom connu
  - /book/chapter/note
  - (= /child::book/child::chapter/child::note)
- Nom inconnu
  - Utiliser le joker '`*`' pour tout élément simple
    - **`A/*/B`** permet de trouver **`A/C/B`** et **`A/D/B`**
    - version longue : **`child::*`**
  - Utilisation de plusieurs astérisques, plusieurs niveaux de correspondance
    - attention à contrôler ce qui se passe
      - nombre de niveaux
      - éléments trouvés
- Pour un attribut
  - @nom-attribut

Quelques exemples

- **`chapter//para`** (noeud contexte = book)
  `child::chapter/descendant-or-self::node()/child::para`
- **`.//para`** (noeud contexte = book)
  `self::node()/descendant-or-self::node()/child::para`
- **`../title`** (noeud contexte = chapter)
  `parent::node()/child::title`
- **`note | /book/title`** (noeud contexte = 2ème chapter)
- **`./*`** (noeud contexte = book)
- **`/comment()`**
- **`.//para/text()`** (noeud contexte = book)
- **`/descendant::node()/@att2`**
- **`//para/@*`**

Filtres avec prédicats

- Les chemins de localisation ne sont pas forcément assez discriminants
  - peuvent fournir une liste de noeuds
- Qu'on peut filtrer à nouveau avec des prédicats
  - prédicat indiqué entre crochets '`[ ]`'
  - **si ce qui est dans le prédicat est *Vrai* : on garde**
- Le prédicat le plus simple utilise la fonction **`position()`**
  - `para[position() = 1]      //1er para`
  - `chapter[2]                //2eme chapter`
- Possibilité de combiner les tests avec '**`and`**' et '**`or`**'
  - `//*[ self::chapter and @att1="tata" ]`

Premier test vrai ou faux ?

Deuxième test vrai ou faux ?

Tests sur les positions / texte

- **`last()`**
  - Récupère le dernier noeud dans la liste
- **`count()`**
  - Evalue le nombre d'items dans la liste

    ```
    child::chapter [count(child::para) = 2]
    ```
- **`string(…)`**
  - Récupére le texte d'un élément en enlevant toutes balises

Exemples

- `/book/chapter[@att1]`
  - les nœuds chapter qui ont un attribut att1
- `/book/chapter[@att1="tata"]`
  - les nœuds chapter qui ont un attribut att1 valant 'tata'
- `/book/chapter/descendant::para()[position()=1]`
  - Le(s) premier(s) nœud(s) de type Text descendants d'un /book/chapter
  - s'abrège en /book/chapter/descendant::text()[1]
- `/book/chapter[count(para)=2]`
  - Les nœuds chapter qui ont deux enfants de type para
- `//chapter[child::note]`
  - Les nœuds chapter qui ont des enfants note

Prédicat : divers

- Pour les booléens
  - not(), and, or
- Pour les numériques
  - <, >, != (différent)
  - +, -, *, div (division entière), mod (reste div entière)
  - number() pour essayer de convertir
  - autres opérateurs : **`round(), floor(), ceiling()`**
- Exemples
  - **`para [not(position() = 1)]`**
  - **`para [position() = 1 or last()]`**
  - **`//node()[number(@att2) mod 2 = 1]`**
    - les nœuds avec un attribut att2 impair

Tests sur les chaînes

- Possibilité de tester si les chaînes contiennent des sous-chaînes
  - `<note>hello there</note>`
    - **`note [contains(text(), "hello")]`**
  - `<note><b>hello</b> there</note>`
    - l'expression précédente ne fonctionne pas **`(note/text()`** donne **`"there")`**
    - utiliser plutôt **`note[contains(., "hello")]`**
      - `'.'` est le nœud courant, et on parcourra tous les enfants

Tests sur les chaînes (2)

- **starts-with*(chaine, motif)***
  - note[starts-with(., "hello")]
- **string(*chaine*)**
  - note[contains(.,string("12"))]
- **string-after(*chaine, terminateur*)**
- **string-before(*chaine, terminateur*)**
- **substring*(chaine, offset, longueur*)**

Tests sur les chaînes (3)

- **`normalize(`*`chaine`*`)`**
  - enlève les espaces en trop
- **`translate(`*`chaine, source, replace`*`)`**
  - **`translate(., "+", "plus")`**
- **`concat(`*`strings`*`)`**
- **`string-length(`*`string`*`)`**

Encore des exemples

- **`/book/chapter/child::para[child::note or text()]`**
  - Tout élément para fils de chapter ayant au moins un fils note ou un fils text
- **`/descendant::chapter[attribute::att1 or @att2]`**
  - Tout élément chapter ayant un attribut att1 ou att2
- **`//*[note]`**
  - Tout élément ayant un fils note
- **`* [self::note or self::para]`** (dans le contexte de chapter)
  - Tout élément note ou para fils du nœud contexte

Quelques fonctions

- S'appliquent sur un ensemble de noeuds
  - id(liste identificateurs) : récupère les éléments ayant ces identificateurs
    - nécessité d'avoir DTD / schéma
    - Ex. id('id54' '678')
  - count()
    - compte le nombre de nœuds. Ex. count(//para)
  - max()
    - rend la valeur maximale
  - sum()
    - rend la somme (les nœuds doivent correspondre à des valeurs numériques, traductibles par number())
  - distinct-values() (Xpath 2.0)
    - élimine les doublons

Conclusion sur XPath

- Xpath permet de retrouver toutes sortes d'information dans les documents XML
  - requêtes
  - transformation : lire une information sous une forme, l'écrire sous une autre forme → XSLT
- Nous avons vu les grands principes
  - pour la description systématique de la syntaxe
    - sites de références
  - pour plus d'exemples
    - sites avec tutoriaux
- Ce cours : présentation de XPATH 1.0
  - des améliorations dans XPATH 2.0

Retour sur la notion de feuille de style

- Ensemble d’instructions qui contrôlent une mise en page d’un document
  - passage de la partie logique à la partie physique
  - possibilité d’utiliser différentes feuilles de style pour des rendus différents à partir d’une même source
    - papier, Web, téléphone…

Spécifications de feuilles de style

- DSSSL - Document Style and Semantics Specification Language
  - Standard lié à SGML pour la présentation et la conversion de documents
- CSS - Cascading Style Sheet
  - Syntaxe simple pour assigner des styles à des éléments HTML ou XML (géré par les navigateurs web)
- XSL - Extensible Stylesheet Language
  - Combinaison des possibilité de DSSSL et CSS avec une syntaxe XML
    - Une feuille de style XSL est un fichier XML
  - Transformation de documents XML vers un autre format
    - XML, HTML, texte…
    - Pour présenter les informations
    - Pour transformer les informations d’un format à un autre

XSL : transformation *vs* mise en forme

- Pendant le développement de XSL, on s'est aperçu que XSL faisait deux choses différentes
  - définir une syntaxe pour transformer des éléments XML et des structures de documents
  - définir des éléments pour présenter du contenu
- XSL a donc été divisé entre
  - XSLT – XSL Transformations
  - XSL – XML Stylesheet Language (XSL-FO)
- Dans tous les cas, Xpath est utilisé pour
  - se déplacer dans l'arbre XML source
  - accéder aux informations

➔Dans ce cours, on verra essentiellement les mécanismes de transformation

Possibilités de XSLT

- Rajouter du texte à du contenu
- Effacer, créer, réordonner et trier des éléments
- Réutiliser des éléments ailleurs dans le document
- Transformer des données entre deux formats XML différents
- Utiliser un mécanisme récursif pour explorer le document
- …

Spécification XSLT

- Disponible à http://www.w3.org/TR/xslt
  - définit 34 éléments et leurs attributs
  - mais on peut se débrouiller en n'en utilisant que quelques uns (moins de 10)

- À connaître pour utiliser XSL
  - XML
    - Syntaxe
    - Espaces de noms (*namespaces*)
  - XPath
  - Rudiments de programmation
    - Tests
    - Boucles
    - Appels récursifs

Principes de XSLT

- Langage de programmation déclaratif
  - Dans une feuille de style XSLT, on déclare et on décrit des **transformations**
    - d'un document XML (arbre) d'entrée
    - vers un document (XML, HTML, texte) de sortie
  - On associe un document XML source contenant des données à cette feuille de style
  - Un outil (processeur XSLT) interprète ces transformations et produit le document de sortie

  ➔ On décrit ce qu'on veut obtenir et non pas la façon dont on l'obtient

- Aspects techniques
  - Une feuille de style XSLT est un document XML
  - Elle contient des instructions spécifiques destinées au processeur XSLT
  - Ces instructions sont préfixées par cet espace de noms :

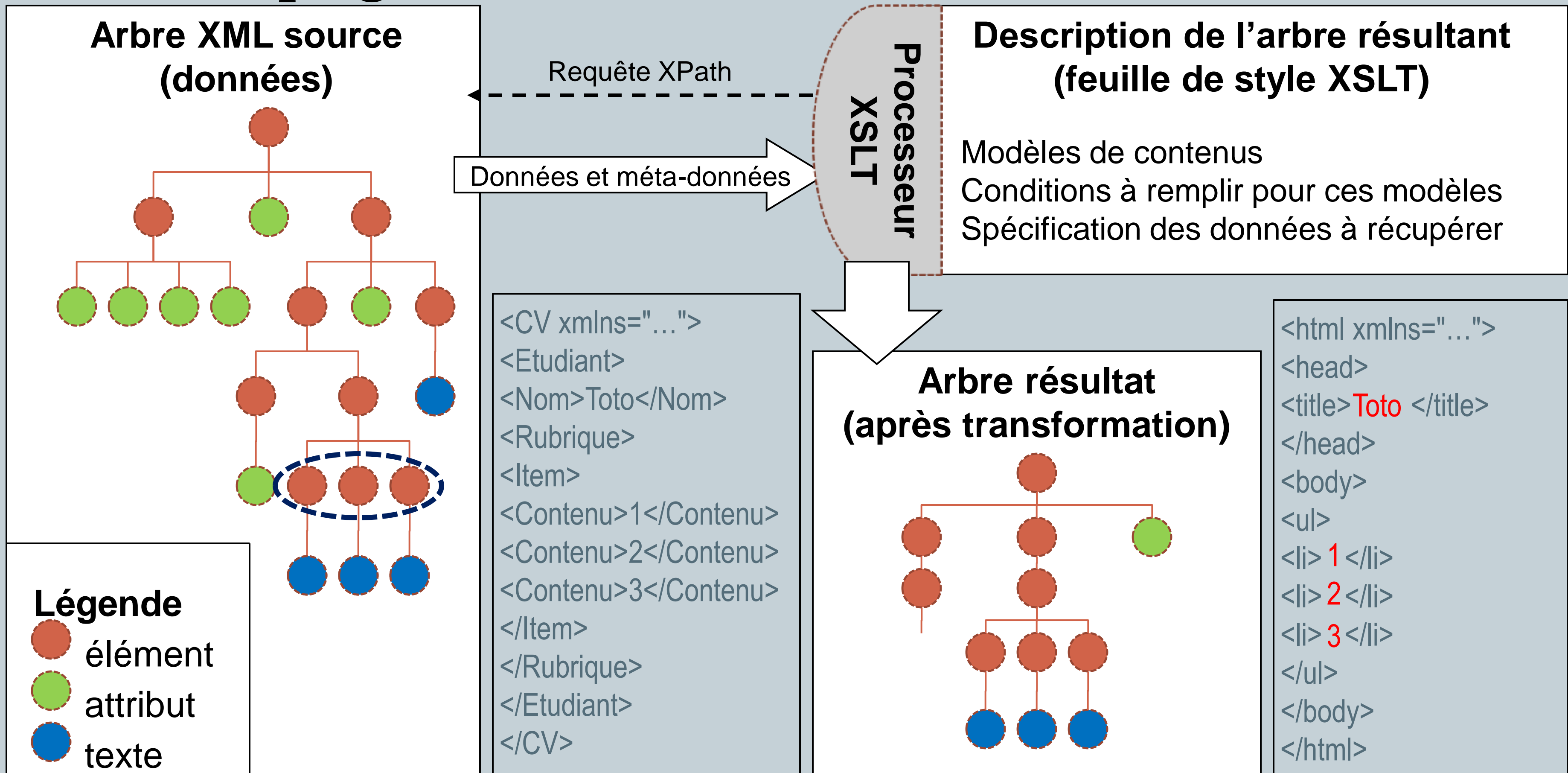
```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

Principe du traitement XSLT

- Description des transformations
  - modèles **(templates)** de transformation appliqués à un nœud
    ➔ Écrire une feuille de style = écrire des templates
  - chaque modèle correspond à un **motif (pattern)** qui décrit des éléments auxquels il s'applique en utilisant XPath
- Effectué récursivement sur une liste de nœuds
  - la liste initiale contient uniquement le nœud racine du document XML à traiter (= nœud contextuel)
- Pour chaque nœud de la liste
  - recherche d'un template (règle) qui lui corresponde
    - le pattern permet de trouver le nœud
  - exécution du template
    - écriture du contenu du template sur la sortie
    - exécution des instructions présentes dans le template
      - réécriture
      - appel de nouveaux templates avec nouveau nœud contexte, *etc.*

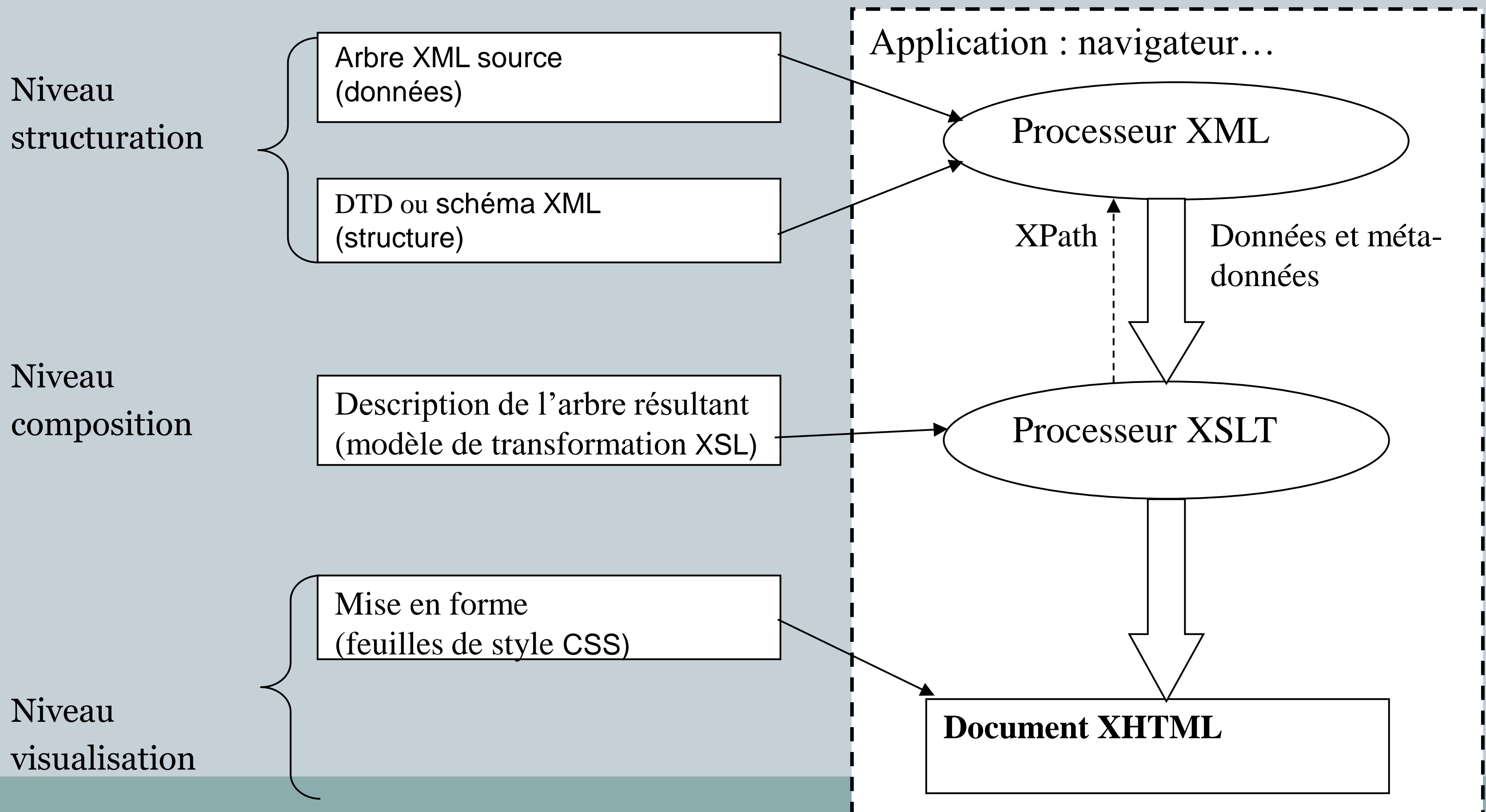
Transformation d'arbres XML : XSL

- Principe général de la transformation d'arbres



Transformation d'arbres XML : XSL

- Utilisation la plus courante de XSL



Spécifier une feuille de style

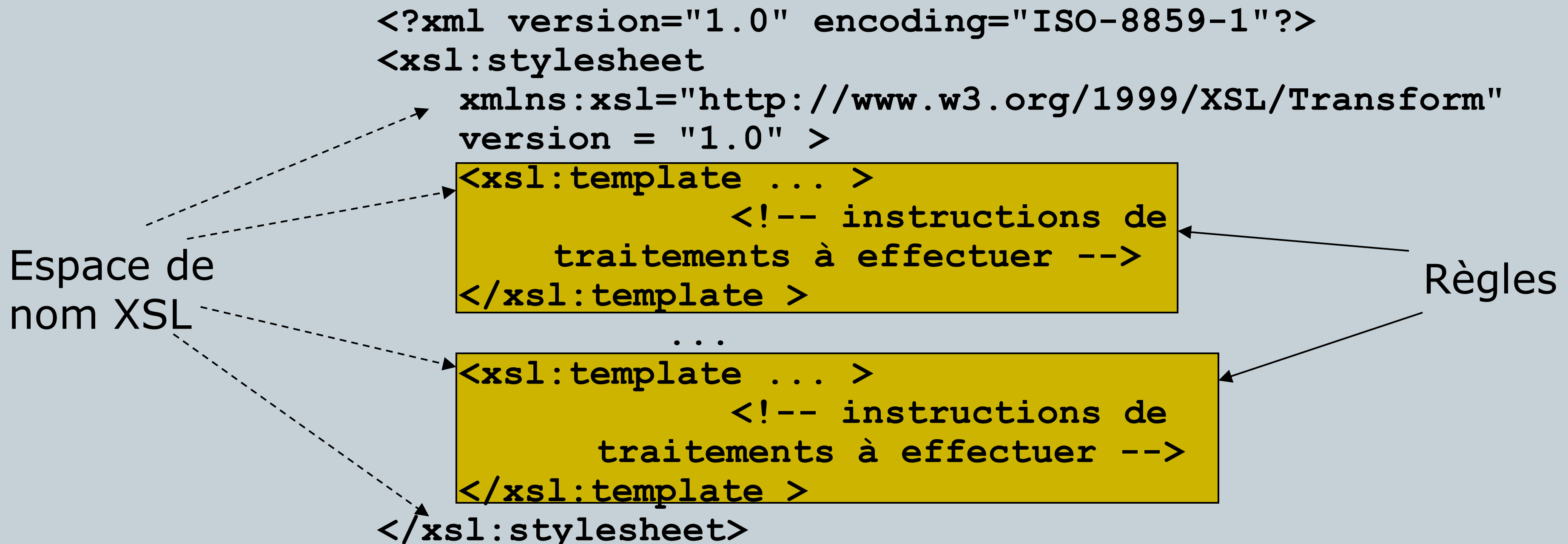
- Utiliser une instruction de traitement dans le prologue du document XML qui doit être transformé

```
<?xml-stylesheet
   href="le-fichier-style.xsl"
   type="application/xml+xsl" ?>
```

- Possibilité de mettre plusieurs choix
  - le processeur XSL choisira la feuille de style la plus adéquate

Élément de feuille de style

- L’élément racine est **`stylesheet`**
- La feuille de style est un ensemble de règles de transformation (**`template`**)



Un premier exemple

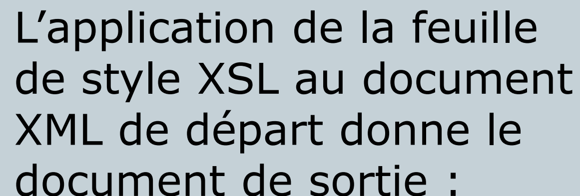
Document XML (données) :

```
<?xml version="1.0"?>
<doc>Hello</doc>
```

+

Feuille de style XSLT (transformations) :

```
<?xml version="1.0"?>
<xsl:stylesheet
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       version="1.0">
<xsl:template match="doc">
    <out>Résultat : <xsl:value-of select="."/></out>
</xsl:template>
</xsl:stylesheet>
```



L'application de la feuille de style XSL au document XML de départ donne le document de sortie :

```
<out>Résultat : Hello</out>
```

12 éléments de premier niveau


```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="..."/>
  <xsl:include href="..."/>
  <xsl:strip-space elements="..."/>
  <xsl:preserve-space elements="..."/>
→ <xsl:output method="..."/>
  <xsl:key name="..." match="..." use="..."/>
  <xsl:decimal-format name="..."/>
  <xsl:namespace-alias stylesheet-prefix="..." result-prefix="..."/>
  <xsl:attribute-set name="..."> ... </xsl:attribute-set>
  <xsl:variable name="...">...</xsl:variable>
  <xsl:param name="...">...</xsl:param>
→ <xsl:template match="..."> ... </xsl:template> ou
→ <xsl:template name="..."> ... </xsl:template>
</xsl:stylesheet>
```

Élément output

- Pour spécifier le format de sortie

```
<xsl:output  method="xml"
             indent="yes"
             encoding="iso-8859-1" />
```

- Attributs de **`output`**
  - **`method`** : xml, html, text
  - **`indent`** : yes, no
  - **`encoding`**
  - **`standalone`** (si on génère du XML)
  - …

Élément template

- Pour spécifier une règle de transformation

```
<xsl:template match="expression">
  ...
</xsl:template>
```

- L'attribut `match` a pour valeur une expression Xpath
  - limitée aux axes child, attribute, descendant-or-self
  - c'est le pattern associé au template
- Si le résultat de cette expression correspond à un nœud de la liste courante
  - la règle est sélectionnée
  - le nœud en question devient nœud contextuel dans le template
- Remarque
  - si on a plusieurs templates candidats, il faut utiliser des règles de priorité pour déterminer lequel utiliser

Élément template (suite)

- Contenu de l'élément xsl:template
  - du **texte**, qui peut contenir des balises
    - ce texte est inséré dans l'arbre destination
      - ex. : "<out>Résultat : </out>"
  - des **instructions** qui décrivent des traitements à effectuer
    - le résultat de leur exécution sera inséré à leur place dans l'arbre destination
      - ex. : <xsl:value-of select="." />
- Exemple

```
<xsl:template match="doc">
   <out>Résultat : <xsl:value-of select="."/></out>
</xsl:template>
```

Traduction : *à chaque fois qu'il y a un élément doc permettant de sélectionner le template, il faut écrire* "<out>Résultat : <xsl:value-of select="."/></out>"*, puis exécuter l'instruction xsl:value-of, et remplacer cette instruction par le résultat de son exécution*

Quelques « éléments instructions » à mettre ***dans*** un élément template (2ème niveau)

- **`xsl:value-of`**
  - Affiche dans le document de sortie la valeur textuelle du nœud courant et de tous ses fils
- **`xsl:apply-templates`**
  - Appelle les modèles disponibles pour les enfants du nœud courant. L'attribut **`select`** permet de spécifier le ou les nœuds sur lesquels continuer d'appliquer les templates
- **`xsl:if`**
  - Effectue un test conditionnel et n'applique la partie du modèle indiqué que si la condition est vraie
- **`xsl:choose`**
  - Structure conditionnelle de type "case" (utilisé en combinaison avec xsl:when et/ou xsl:otherwise)
- **`xsl:for-each`**
  - Permet d'appliquer un canevas à chaque noeud correspondant au modèle
- **`xsl:copy`**
  - Copie le noeud courant dans l'arbre résultat (mais pas les attributs et enfants)
- **`xsl:copy-of`**
  - Copie le noeud sélectionné et ses enfants et attributs
- **`xsl:element`**
  - Crée un élément avec un nom spécifié
- **`xsl:attribute`**
  - Crée un élément avec un nom et une valeurs spécifiés
- **`xsl:comment`**
  - Crée un commentaire dans l'arbre résultat

Élément xsl:value-of

- Pour convertir l'objet spécifié par un attribut '**`select`**' en une chaîne de caractères
  - cf. fonction string() de xpath
- Exemple

```
<p>Hello world</p>
----------------------------
<xsl:template match="p">
  <e><xsl:value-of select="."/></e>
</xsl:template>
```

donnera

```
<e>Hello world.<e>
```

Élément xsl:value-of

- Spécifier une valeur pour l'attribut **`select`**

```
<full-name first="John" second="Smith"/>
----------------------------------------
<xsl:template match="full-name">
  <person1>
    <xsl:value-of select="@first"> +
    <xsl:value-of select="@second">
  </person1>
  <person2 name="{@first} {@second}" />
      <!-- SYNTAXE SPECIALE -->
</xsl:template>
```

```
=> <person1>John + Smith</person1>
   <person2 name="John Smith"/>
```

- Remarque : plusieurs éléments possèdent cet attribut select
  - **`apply-templates`**, **`value-of`**, **`copy-of`**, **`param`**, **`sort`**, **`variable`**, **`with-param`**

Élément apply-templates

- Indique au processeur XSL de traiter les éléments enfants directs du nœud courant en leur appliquant les règles définies dans d’autres modèles
  - bref, « continuer le traitement sur les enfants »
  - Pour cela, on peut spécifier le nom des enfants (attribut **`select`**). Ou pas…
- Traitement récursif
  - Au début de l’analyse, le processeur applique le template correspondant au nœud de plus haut niveau.
  - Dans ce template, on peut utiliser l’élément **`apply-templates`** pour appliquer d’autres modèles
  - Dans ces autres modèles, on peut utiliser l’élément **`apply-templates`** pour appliquer d’autres modèles…

Élément apply-templates : avec attribut **select**

- L'attribut select permet de spécifier certains éléments enfants auxquels la transformation doit être appliquée
  - plus spécifique que <xsl:apply-templates />
- Utilisation de patterns Xpath pour sélectionner les enfants

```
<Biblio auteur="Sartre">On trouve notamment <Livre>La
  nausée</Livre> et <Livre>Les mains
  sales</Livre>.</Bilio>
-------------------------------------------------------
<xsl:template match="Biblio">
    Ouvrages : <xsl:apply-templates select="Livre" />
  </xsl:template>
  <xsl:template match="Livre">
    <xsl:value-of select="." />
  </xsl:template>
```

```
=> Ouvrages :
La nausée Les
mains sales
```

Élément apply-templates : sans attribut

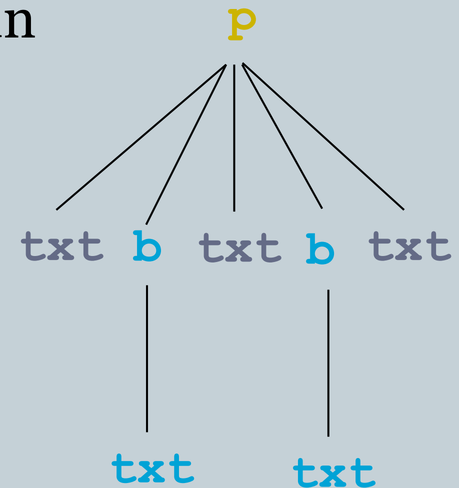
- L'élément apply-templates sans attribut applique un template pour **tous** les enfants du nœud courant

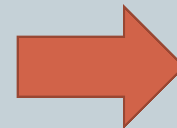
```
<p>C'est <b>très</b> important cette
<b>chose</b>.</p>
------------------------------

<xsl:template match="p">
  Para <xsl:apply-templates/>
</xsl:template>

<xsl:template match="b">
  Bold <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()">
  Text
</xsl:template>
```





```
Para
   Text
   Bold
      Text
   Text
   Bold
      Text
   Text
```

Élément apply-templates : sans attribut

- Autre exemple

```
<xsl:template match="book">
  <p>
    Un livre : <xsl:apply-templates/>
  </p>
</xsl:template>
```

- Remarque
  - On ne peut pas ré-arranger la structure hiérachique d'un document XML source (le document XSL serait mal formé)

```
<xsl:template match="firstname">
  <p><xsl:apply-templates/>
</xsl:template>                                <= mauvais
<xsl:template match="lastname">
  <xsl:apply-templates/></p>
</xsl:template>
```

Règles par défaut : racine/éléments

- Quand aucune règle n'est sélectionnée, le processeur XSLT applique des règles par défaut
- Première règle par défaut
  - pour les éléments et la racine du document.

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

  - on demande l'application de règles pour les fils du nœud courant (éléments ou textuels)
  - conséquence
    - pas obligatoire de faire une règle pour la racine du document à transformer

Règles par défaut : texte et attributs

- Deuxième règle par défaut
  - On insère dans le document résultat la valeur du noeud Text, ou de l'attribut

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

  - Conséquence
    - si on se contente des règles par défaut, on obtient la concaténation de noeuds de type text()
  - Remarque
    - par défaut, les nœuds attributs ne sont pas atteints (il faut des règles pour les atteindre)

Règles par défaut : autres nœuds

- Troisième règle par défaut
  - Pour les instructions de traitement et les commentaires : on ne fait rien

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

  - Conséquence
    - Si on ne les sélectionne pas explicitement, en définissant une règle pour les traiter, il ne se passe rien.

- Feuille de style XSLT minimale

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

Créer un élément

- Utiliser l’élément **`xsl:element`**

```
<part><title>Le titre</title></part>
--------------------------------------------
<xsl:template select= "part" >
    <xsl:element name="partie">
      <xsl:value-of select= "./title"/ >
    </xsl:element>
  </xsl:template>
```

```
<partie>Le titre</partie>
```

- Ou bien simplement écrire l’élément XML dans le modèle

```
<xsl:template select= "part" >
  <partie>
    <xsl:value-of select= "./title"/ >
  </partie>
</xsl:template>
```

Créer des éléments

- Utilisation avec des variables

```
<xsl:template name="CreateHeader">
  <xsl:param name="level">3</xsl:param>
  <xsl:element namespace="html" name="h{$level}">
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

<xsl:template match="title">
  <xsl:call-template name="CreateHeader">
    <xsl:with-param name="level">1</xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

Créer un attribut

- Utiliser l'élément **`xsl:attribute`**
  - Utilité
    - Permet de demander au processeur XSLT de calculer lui-même le nom ou la valeur de l'attribut

```
<personne><nom>Outang</nom><prenom>Laurent</prenom></personne>
--------------------------------------------
<xsl:template select= "personne" >
    <xsl:attribute name="nom">
        <xsl:value-of select= "./prenom"/ >
        <xsl:value-of select= "./nom"/ >
    </xsl:attribute>
</xsl:template>
```

```
<personne nom="Laurent Outang" />
```

Copier des éléments

- Élément '**copy**'
  - copie le nœud courant (mais pas les fils et les attributs)
  - ```
    <xsl:template match="h1|h2|h3|h4|h5|h6|h7">
     <xsl:copy>
      Header: <xsl:apply-templates/>
     </xsl:copy>
    </xsl:template>
    ```
- Pour créer de nouveaux attributs : xsl:attribute
  - ```
    <xsl:template match="h1|h2|h3|h4|h5|h6|h7">
     <xsl:copy>
      <xsl:attribute name="style">purple</xsl:attribute>
      Header: </xsl:apply-templates />
     </xsl:copy>
    </xsl:template>
    ```

Crée des éléments copiés avec
un attribut style qui vaut purple
Ex. <h3 style="purple">

Stocker et ajouter des groupes d’attributs

- Élément attribute-set

```
<xsl:attribute-set name="class-and-color">
 <xsl:attribute name="class">standard</xsl:attribute>
 <xsl:attribute name="color">red</xsl:attribute>
</xsl:attribute-set>

<xsl:template match="h1|h2|h3|h4|h5|h6|h7">
 <xsl:copy>
 <xsl:use-attribute-sets name="class-and-color” />
  Header: <xsl:apply-templates/>
 </xsl:copy>
</xsl:template>
```

Recopie de fragments d'arbres

- L'élément **`copy-of`** permet de copier des fragments du fichier d'entrée sans perdre les attributs

```
<xsl:template match="body">
  <body>
    <xsl:copy-of select="//h1 | //h2" />
    <xsl:apply-templates/>
  </body>
</xsl:template>
```

Conditions

- On peut faire un test '**`if`**' pendant le traitement

```
<xsl:template match="para">
 <html:p>
  <xsl:if test="position() = 1">
   <xsl:attribute name="style">color: red</xsl:attribute>
  </xsl:if>
  <xsl:if test="position() > 1">
   <xsl:attribute name="style">color: blue</xsl:attribute>
  </xsl:if>
  <xsl:apply-templates/>
 </html:p>
</xsl:template>
```

Conditions

- Les éléments '**choose**', '**when**', '**otherwise**'

```
<xsl:template match="para">
 <html:p>
  <xsl:choose>
   <xsl:when test="position() = 1">
    <xsl:attribute name="style">color: red</xsl:attribute>
   </xsl:when>
   <xsl:otherwise>
    <xsl:attribute name="style">color: blue</xsl:attribute>
   </xsl:otherwise>
   <xsl:apply-templates/>
  </xsl:choose>
 </html:p>
</xsl:template>
```

Boucles

- L'élément **`for-each`** permet de répéter une opération sur des éléments

```
<liste>
   <item><title>Titre1</title><year>2000</year></item>
   <item><title>Titre2</title><year>1998</year></item>
</liste>
----------------------------------------
<xsl:template match="liste">
   <xsl:for-each select="./item">
       <!-- traitement pour chaque item -->
       <p><xsl:value-of select="./title"/></p>
   </xsl:for-each>
</xsl:template>
```

```
<p>Titre2</p><p>Titre1</p>
```

Élément sort

- Permet de spécifier que les éléments sont triés suivant une certaine propriété

```
<list>
  <item>ZZZ</item>
  <item>AAA</item>
  <item>MMM</item>
</list>
```

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

AAAMMMZZZ

bbbcccaaa

```
<list>
  <item code="Z">aaa</item>
  <item code="A">bbb</item>
  <item code="M">ccc</item>
</list>
```

```
<xsl:template match="list">
  <xsl:apply-templates>
    <xsl:sort select="@code"/>
  </xsl:apply-templates>
</xsl:template>
```

Attributs de l'élément sort

- Attribut '**`order`**'
  - pour classer croissant ou décroissant
    - '**`ascending`**' ou '**`descending`**'
- Attribut '**`data-type`**'
  - pour indiquer si les données à prendre en compte sont une simple chaîne ou doivent être interprétées comme des nombres
    - '**`text`**' (par défaut) ou '**`number`**'
- Attribut '**`case-order`**'
  - ordre majuscules / minuscules
    - '**`lower-first`**' ou '**`upper-first`**'

Élément number

- Pour la numérotation automatique
  - ```
    <xsl:template match="item">
      <xsl:number/><xsl:apply-templates/>
    </xsl:template>
    ```
- Attributs
  - `level` = `'single'` ou `'any'` ou `'multiple'`
  - `count` = `"list-1|list-2"`
  - `format` = `"1.A"` (également `"I"` et `"i"`)
  - `from = "3"`
  - `grouping-separator = ","`
  - `grouping-size = "3"`
  - `value = "position()"`
    ```

Attribut mode

- Attribut de l'élément template
- Permet de spécifier quelle règle utiliser en fonction de l'élément retrouvé

```
<xsl:template match="chapter/title">
  <html:h1><xsl:apply-templates/></html:h1>
</xsl:template>

<xsl:template match="chapter/title" mode="h3">
  <html:h3><xsl:apply-templates/></html:h3>
</xsl:template>

<xsl:template match="intro">
  <xsl:apply-templates
          select="//chapter/title" mode="h3"/>
</xsl:template>
```

Élément variable

- On peut déclarer et utiliser des variables en XSLT
  - `<xsl:variable name="colour">red</xsl:variable>`
  - définition de la variable colour avec valeur red
- Une variable est référencée avec la notation **$**
  - `<xsl:value-of select="$colour"/>`
- On peut aussi l'utiliser dans les éléments de sortie
  - `<ajr:glyph colour="{$colour}"/>`

Appel explicite de templates

- Si on a besoin plusieurs fois du même formatage
  - on nomme le template pour pouvoir l'appeler

```
<xsl:template name="CreateHeader">
  <html:hr/>
  <html:h2>***<xsl:apply-templates/>***</html:h2>
  <html:hr/>
</xsl:template>
...

<xsl:template match="title">
  <xsl:call-template name="CreateHeader" />
</xsl:template>

<xsl:template match="head">
  <xsl:call-template name="CreateHeader" />
</xsl:template>
```

Passer des paramètres à un template

- L'élément **`param`**, une variable spéciale
  - `<xsl:param name="nom">`*`valeur par defaut`*`</xsl:param>`
  - `<xsl:with-param name="nom">`*`nouvelle valeur`*`</xsl:with-param>`
- L'élément **`call-template`** peut passer une nouvelles valeur de **`param`** à un **`template`**

```
<xsl:template match="name">
 <xsl:call-template name="salutation">
  <xsl:with-param name="greet">Hello </xsl:with-param>
 </xsl:call-template>
</xsl:template>

<xsl:template name="salutation">
 <xsl:param name="greet">Dear </xsl:param>
 <xsl:value-of select="$greet"/>
 <xsl:apply-templates/>
</xsl:template>
```

remplacera la valeur par défaut

valeur par défaut

Éléments import / include

- Pour composer une feuille de style à partir de plusieurs fichiers XSL

```
<xsl:stylesheet … >
  <xsl:import href="tables.xsl" />
  <xsl:import href="features.xsl" />
  <!-- ordre important, seul cas pour
   les éléments de premier niveau -->
 <xsl:template … > … </xsl:template>
  …
</xsl:stylesheet>
```

- Inclure des fichiers XML : **`xsl:include`**
  - comportement équivalent à **`xsl:import`**
  - mais pas de possibilité d'écraser une définition importée par une définition de plus haut-niveau → erreur si deux définitions similaires

XSL – Formatting objects

- Spécification des objets de formatage à associer à des éléments XML, pour sorties papier, audio, écran, téléphone portable, etc.
- Ensemble de « zones » (area) qui se suivent ou se contiennent les unes les autres

Un exemple XSL-FO


```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns="http://www.w3.org/1999/XSL/Format"
font-size="16pt">
<layout-master-set>
<simple-page-master
   margin-right="15mm" margin-left="15mm"
   margin-bottom="15mm" margin-top="15mm"
   page-width="210mm" page-height="297mm"
   master-name="bookpage">
<region-body region-name="bookpage-body"
   margin-bottom="5mm" margin-top="5mm" />
</simple-page-master>
</layout-master-set>
<page-sequence master-reference="bookpage">
<title>Hello world example</title>
<flow flow-name="bookpage-body">
<block>Hello XSLFO!</block>
</flow>
</page-sequence>
</root>
```

Conclusion sur XSL

- XSL n'est pas un langage de feuilles de styles
  - XPath : parcours d'arbre / localisation de nœuds
  - XSLT : transformation d'arbres
- XSL fait appel à des principes non présentés ici
  - Xpath : bases de données
  - XSLT : un début de programmation ?
- Remerciements
  - Ce cours s'appuie largement sur celui d'Alan Robinson *http://industry.ebi.ac.uk/~alan/XMLWorkshop/*
  - Cours Bernd Ammann programmation XSLT

Exercice (suite en TD)

Ecrire une feuille de style XSLT permettant de passer du document carte1.xml à card1.xml

carte1.xml

```
<carte>
  <titre>Dr.</titre>
  <nom>Paul Durand</nom>
  <telephone inter="33">4 78 34 25
   12</telephone>
  <telephone inter="33">6 12 45 25
   12</telephone>
  <adresse>
   <rue>Impasse des Fleurs</rue>
   <code>69001</code>
   <ville>Lyon</ville>
   <pays>France</pays>
  </adresse>
  <courriel>
paul.durand@provider.com</courriel>
</carte>
```

card1.xml

```
<card>
  <name title="Dr.">
    Paul Durand</name>
  <address>
    <street>Impasse des
   Fleurs</street>
    <zipcode>69001 Lyon</zipcode>
    <country>France</country>
  </address>
  <phones>
   <phone>(33)4 78 34 25 12</phone>
   <phone>(33)6 12 45 25 12</phone>
   </phones>
</card>
```