ENST Bretagne
Département informatique

Formation continue : Maîtriser XML et XSLT

Yannis Haralambous

École Nationale Supérieure des Télécommunications de Bretagne
Technopôle Brest Iroise, CS 83818, 29238 Brest Cedex 3

Applications industrielles de XML

Qu'est-ce que XML ?

- Un document XML contient des données ayant une structure d'arbre ;
- cette structure est déterminée par des balises qui se trouvent dans les données ;
- les balises sont écrites selon une syntaxe très stricte ;
- on a des outils généralistes qui peuvent traiter de la même manière tous les documents XML (plus besoin d'outils spécifiques à chaque format de fichier) ;
- à terme, on voudrait que toutes les informations stockées ou échangées soient structurées en XML.

À quoi sert XML ?

- XML est une norme de structuration des données ;
- la structuration peut servir au stockage ou à la communication.

Exemples d'application :

- *Business Transaction Protocol* : gestion de transactions commerciales complexes sur Internet ;
- *Business-Centric Methodology* : interopérabilité de systèmes d'information e-Commerce ;
- *Universal Description, Discovery and Integration* : méthode standard pour découvrir et invoquer des services Web de manière dynamique ;
- *Universal Business Language* : une bibliothèque XML de documents commerciaux (bons de commande, factures, etc.) ;

Applications de XML

- *Application Vulnerability Description Language* et *Web Application Security* : normalisation de l'échange d'information sur les lacunes de sécurité des applications exposées en réseau ;
- *Digital Signature Services* : interface XML pour gérer les signatures numériques des services Web et autres applications ;
- *eXtensible Access Control Markup Language* : représentation et évaluation des stratégies de contrôle d'accès ;
- *Provisioning Services* : cadre XML pour gérer les informations identitaires et les ressources système à l'intérieur et entre les organisations ;
- *Public Key Infrastructure* : utilisation des certificats numériques pour la gestion d'accès aux ressources numériques et pour les transactions électroniques ;

Applications de XML

- *Security Services (SAML)* : définition d'un cadre XML pour la création et l'échange d'information sécuritaires entre parténaires en ligne ;
- *XML Common Biometric Format* : description normalisée de l'information biométrique (ADN, empreintes digitales, scans d'iris, géométrie de la main) ;
- *legalXML* : gestion de documents légaux, jurisprudence, casiers judiciaires, documents notariaux, impôts et taxes ;
- *Materials Markup Language* : normalisation des mesures et infos relatifs à la manifacture de matériaux ;
- *Production Planning and Scheduling* : modèles objets pour la planification collaborative et le planning en manifacture ;
- *Customer Information Quality* : spécifications ouvertes et globales, indépendantes d'application, pour la gestion de l'information et du profil des clients ;

Applications de XML

- *Human Markup* : balisage contextuel des intentions culturelles, sociales, kinésiques et psychologiques dans la communication ;
- *Translation Web Services* : automatisation de la traduction et de la localisation en tant que service Web ;
- *XML Localisation Interchange File Format* : standard de localisation de l'échange de données ;
- *Computer Graphics Markup*, *Scalable Vector Graphics* : standards graphiques ;
- *DocBook* : standard de structuration de document technique ;
- *Text Encoding Initiative* : idem pour textes littéraires ;
- *Open Document Format for Office Applications* : interopérabilité des suites bureautiques ;

Applications de XML

- *eXtensible Resource Identifier* : aller au-delà des URL et URI ;
- *User Interface Markup Language* et *eXtensible User-interface Language* : structuration des interfaces utilisateur ;
- *XRI Data Interchange* : créer un standard de partage, de lien et de synchronisation de données au-dessus d'Internet et d'autres réseaux en se servant des XRI ;
- *Implementation Interoperability and Conformance* : infrastructure pour l'interopérabilité des applications ;

  [Références : `http://www.oasis-open.org/committees`]
- *DARPA Agent Markup Language* : outils pour créer des ontologies et pour baliser l'information selon celles-ci ;
- *Artificial Intelligence Markup Language* et *Robotic Markup Language* : systèmes de balisage d'unités de connaissance pour robots ;

Applications de XML

- *Chemical Markup Language* : représentation des formules chimiques ;
- *Mathematics Markup Language* : représentation logique ou graphique des mathématiques ;
- *Geography Markup Language* : représentation de données géospatiales ;
- *Bioinformation Sequence Markup Language* : spécification et dépôt de données bio-informatiques ;
- *Architecture Description Markup Language* : description d'architecture matérielle (composantes, connecteurs, ports, rôles, propriétés, systèmes, etc.) ;
- *Virtual Reality Markup Language* : description de réalités virtuelles ;
- *Music Markup Language* : informatisation de la notation musicale ;

Applications de XML

- *Systems Biology Markup Language* : représentation des modèles de réseaux de réactions biochimiques (réseaux métaboliques, chemin de signalisation cellulaire, réseaux régulatoires, etc.) ;
- *Voice Extensible Markup Language* : description de dialogues basés sur une logique programmée (par exemple : répondeurs automatiques intelligents) ;
- *Wireless Markup Language* : le format du WAP ;
- *Remote Telescope Markup Language* : balisage de description de requêtes d'observation téléscopique ;
- *Log Markup Language* : description de fichiers log de serveurs Web ;
- *Turing Machine Markup Language* : description de machines de Turing ;

Applications de XML

- *Theological Markup Language* et *Liturgical Markup Language* : balisage de textes théologiques et liturgiques ;
- *Eatdrinkfeelgood Markup Language* : recettes structurées en menus ;
- *XML Kitchen Sink* : réparation à distance d'éviers de cuisine à l'aide de documents XML, norme publiée le 1er avril 2001.
- et, bien sûr :
- *XHTML* (eXtended HyperText Markup Language) ;
- et *RSS* (Really Simple Syndication).

Le document XML

XML version 1.0, du 10 février 1998

Document XML

Ceci n'est pas un document XML :

```
<?xml version="1.0" encoding="utf-8"?>
<affirmation id="x01" auteur="moi">
Essayons de comprendre <i>ce qui se
passe <b>dans ce texte</i></b>
<!-- encore faut-il en connaître la
syntaxe --></affirmation>
```

Ceci est un document XML :

```
<?xml version="1.0" encoding="utf-8"?>
<affirmation id="x01" auteur="moi">
Essayons de comprendre <i>ce qui se
passe <b>dans ce texte</b></i>
<!-- encore faut-il en connaître la
syntaxe --></affirmation>
```

Document bien formé

- Un objet de données est un document XML, s'il est *bien formé*. En plus, il peut être *valide*.
- Un document est *bien formé* si :

1. la syntaxe des balises est respectée ;
2. les éléments qu'il contient sont bien imbriqués les uns dans les autres (et forment donc un *arbre*).

Structure logique : déclaration XML

- Un document XML commence par la *déclaration XML* :
  ```
  <?xml version="1.0" standalone="no" encoding="utf-8"?>
  ```
- La version actuelle de XML est la 1.0 (10 février 1998).
- Un document XML est *autonome* (*standalone*) quand il ne contient pas de déclarations d'entités externes.
- Le codage par défaut est *Unicode*, codé en *UTF-8*.

Qu'est-ce qu'un codage ?

- Le texte est stocké dans la mémoire de l'ordinateur sous forme de bits groupés en octets.
- Un *codage de caractères* est une correspondance entre :
  - des valeurs d'octets (→ codage à 6, 7 ou 8 bits) ou de paires d'octets (→ codage à 16 bits) ou de groupes de quatre octets (→ codage à 17 … 32 bits), et
  - un certain nombre de symboles (lettres d'alphabets divers, chiffres, signes de ponctuation, idéogrammes, pictogrammes, etc.).

Exemples de codages

- *ASCII* est un codage à 7 bits.
- *ISO Latin-1* est un codage à 8 bits (dont les 128 premières positions forment le codage ASCII)
- *Unicode* est un codage à 21 bits, divisés en 17 plans de 65 536 positions. Ses 256 premières positions forment le codage ISO Latin-1.

Qu'est-ce que UTF-8 ?

UTF-8 est une forme du codage Unicode.

Au lieu de coder un texte systématiquement en 4 octets (32 bits), on combine les bits pour former des caractères de longueur variable : de 1 à 4 octets. Un caractère ASCII est codé par un seul octet et un texte entièrement codé en ASCII reste inchangé.

UCS-32BE	UTF-8			
	Octet 1	Octet 2	Octet 3	Octet 4
00000000 00000000 0xxxxxxx	0xxxxxxx			
00000000 00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
00000000 zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuzz zzzzyyyy yyxxxxxx	11110uuu	10zzzzzz	10yyyyyy	10xxxxxx

Catégories de caractères 1/2

Les caractères Unicode peuvent appartenir à plusieurs *catégories* et *sous-catégories* :

- *Lettres* (majuscules, minuscules, casse de titre) ;
- *Marques* (à chasse nulle, combinatoires avec chasse, englobantes) ;
- *Nombres* (chiffres décimaux, lettres, autres) ;
- *Ponctuation* (connecteurs, tirets, débuts, fins, guillemets, autres) ;
- ...

Catégories de caractères 2/2

- ...
- *Symboles* (mathématiques, devises monétaires, modificateur, autres);
- *Nombres* (chiffres décimaux, lettres, autres);
- *Séparateurs* (espaces, lignes, paragraphes);
- *Autres* (commandes, formatages, code d'indirection, usage privé, non affectés).

Qu'est-ce qu'un *nom XML* ?

Un *nom XML* est une unité lexicale

- commençant par une *Lettre* ou un '_' ;
- suivi de zéro, une ou plusieurs *Lettres*, *Nombres*, *Marques*, '.', '-', soulignés ou ':'.

Exemples : `livre`, `_php_4`, `enstb:info`, كِتاب, *βιβλίον*, `___`, `ẑâẑôû`.

Contre-exemples : `2fois`, `.bashrc`, `je-suis-☺`, `C++`, `--`.

Attention ! Le ':' est utilisé pour les *espaces de nommage*.

Déclaration de type de document

- Après la déclaration XML, un document XML devrait contenir une *Déclaration de type de document* (DTD). Celle-ci contient des *déclarations de balisage* et définit des contraintes sur la structure logique du document.
- Un document XML qui respecte les contraintes imposées par sa DTD est un document XML *valide*. Un processeur XML qui teste la validité d'un document est un *validateur* XML.

Structure globale de document XML

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nom SYSTEM "fichier.dtd ou URL" [
déclarations
]>

... corps du document ...
```

Structure logique : éléments

Chaque document XML contient un ou plusieurs *éléments* dont les limites sont marquées par des *balises ouvrantes* ou *fermantes* ou par des *balises d'élément vide*. Le nom de la balise est un *nom XML*. Entre les balises ouvrante et fermante se trouve le *contenu* de l'élément.

Exemples :

```
<titre>L'intellect dans ma vie</titre>
<auteur>Loana Lofteuse</auteur>
<logo/>
```

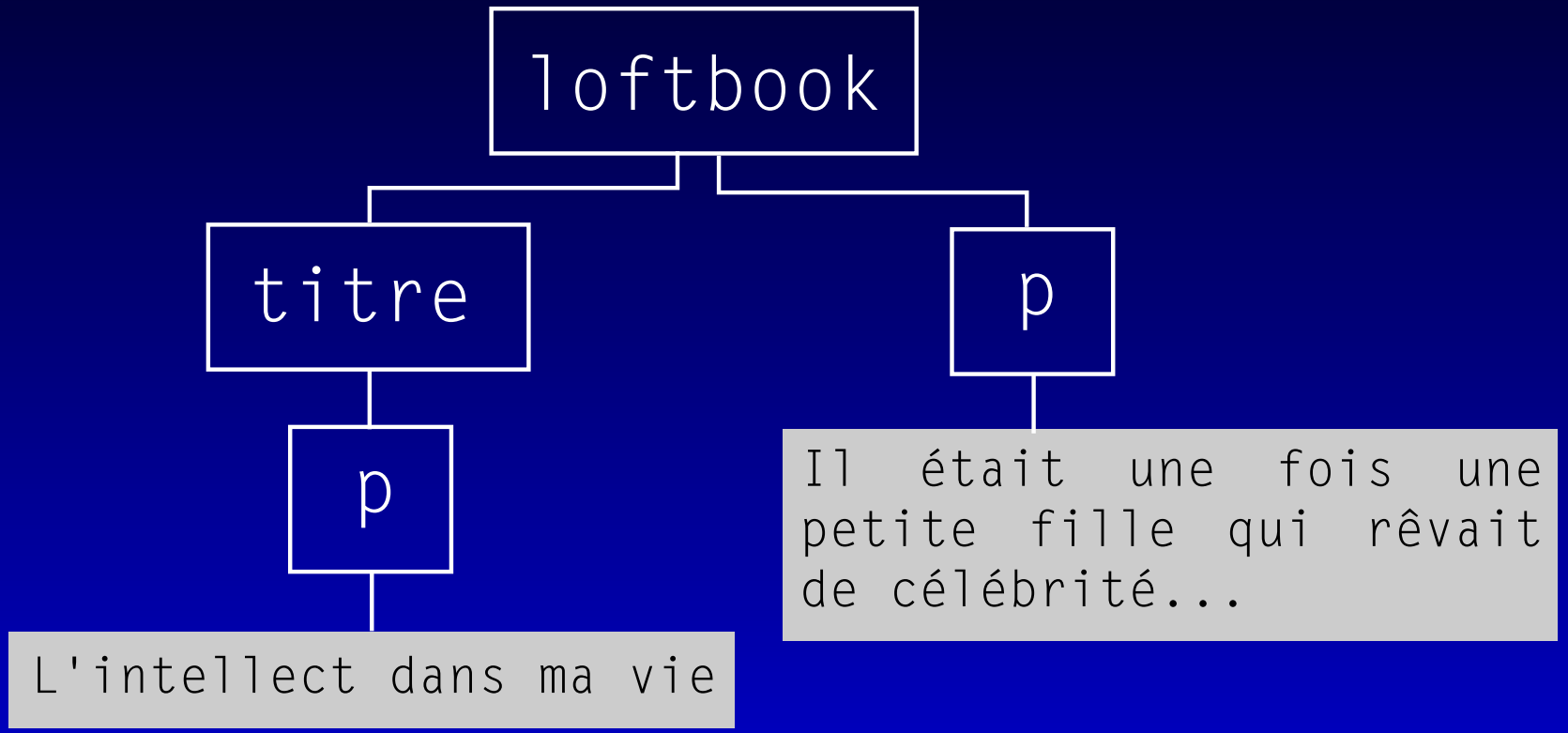
Structure logique : imbrication

Les éléments *s'imbriquent* et forment ainsi un *arbre*. Au sommet de l'arborescence on trouve un élément de même nom que la DTD. Exemple :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE loftbook SYSTEM "loftbook.dtd">
<loftbook><titre><p>L'intellect dans ma vie</p></titre>
<p>Il était une fois une petite fille qui rêvait de célébrité...</p>
</loftbook>
```

Structure logique : imbrication

L'exemple précédent sous forme d'arbre :



Structure logique : attributs

À l'intérieur des balises ouvrantes ou d'élément vide on peut placer des *attributs*. Un attribut a un *nom* et une *valeur* (données textuelles entre quotes ou double-quotes).

Exemples : `<titre niveau="1">`, `<logo src="logo.gif" alt="Le logo"/>`, `<section type="Chapitre" label="intro1">`

Attention ! Ne pas mettre de balise dans la valeur d'un attribut.

Un attribut prédéfini

La spécification XML nous invite à utiliser l'attribut `xml:lang` pour spécifier la langue d'un block de texte. Sa valeur doit être un *code de langue* (ISO 639 ou privé) suivi éventuellement d'un ou plusieurs *sous-codes* (ISO 3166).

Exemples :

```
<p xml:lang="fr-FR">, <p xml:lang="fr-CA">
<p xml:lang="br">, <p xml:lang="br-FIN">
<p xml:lang="x-br-FR-ENSTB-D3:114">
```

Commentaires

Les *commentaires* ne font pas partie des données textuelles du document.

Syntaxe :

```
<!-- ...commentaire quelconque... -->
```

Exemple :

```
<!-- Dernières corrections 4/11/01 -->
```

Attention ! Pas de `--` dans un commentaire.

Instructions de traitement

Les *instructions de traitement* (ou «IT») sont des commentaires *ciblés*.

Syntaxe :

```
<?cible ...code quelconque...?>
```

Exemple :

```
<?tex $x^2+y^2=\sqrt{2}$?>
```

(le code ci-dessus, lu par TEX, produira : $x^2 + y^2 = \sqrt{2}$.)

Attention ! Pas de `?>` dans le code. Les noms de cible `xml` et `XML` sont reservés.

Sections CDATA

Ce sont des blocs de données textuelles dans lesquels les balises ne sont plus reconnues comme telles. Syntaxe :

`<![CDATA[ ...`*`données quelconques`*`... ]]>`

Les données peuvent contenir des caractères parmi les suivants : `09`, `0A`, `0D`, `20` à `D7FF`, `E000` à `FFFD`, `10000` à `10FFFF`.

Attention ! Pas de `]]>` dans les données.

Structure physique : entités

Un document XML contient des *entités* :

- l'*entité document*, et éventuellement :
- des *appels de caractère* ;
- des *entités internes* ;
- des *entités externes*.

L'entité document est le fichier par lequel on débute la lecture du document XML. Tous les autres types d'entité ont des *appels* et des *textes de remplacement*.

Appels de caractère, entités prédéfinies

Un *appel de caractère* est une manière simple d'obtenir n'importe quel caractère Unicode. Syntaxe : `&#` suivi d'un nombre décimal (ou `&#x` suivi d'un nombre hexadécimal) et d'un point virgule.

Exemple : `&#x2639;` est le caractère ☹.
Contre-exemple : le caractère 🪟 ne peut pas être obtenu ainsi.

Entités prédéfinies : `<`, `>`, `&apos;`, `&quot;`, `&` pour les caractères <, >, ', ", &.

Entités internes

Une *entité interne* est définie dans la DTD : on y déclare son nom et on y donne explicitement son texte de remplacement. Son appel, dans le document, s'écrit :

```
&nom;
```

Le processeur XML remplace cet appel par son texte de remplacement.

Entités externes

Une *entité externe* est un fichier. Elle est définie dans la DTD : on y déclare son nom et on y donne son URL ou chemin d'accès. Son appel, dans le document, s'écrit :

`&nom;`

Le processeur XML remplace cet appel par le contenu du fichier.

Attention ! Un tel fichier doit commencer par une déclaration XML. Le codage peut être différent de celui du fichier contenant l'appel.

Arbre des entités

Les entités (aussi bien internes qu'externes) peuvent contenir d'autres entités. On obtient ainsi un *arbre* d'entités. Les nœuds de cet arbre sont les différents textes de remplacement qui sont situés dans la DTD, et les différents fichiers utilisés par le document XML.
Le processeur XML va d'abord réduire cet arbre en remplaçant tous les appels d'entité par les textes de remplacement et contenus de fichier respectifs.

La Définition de Type de Document

Déclaration d'élément

Pour déclarer un élément *toto* on écrit son nom et ses sous-éléments possibles :

`<!ELEMENT` *toto specs* `>`

où *specs* peut être :

- `EMPTY` si l'élément est vide ;
- `ANY` si tout sous-élément est permis ;
- *un contenu mixte* ;
- *une expr. rég. de sous-éléments*.

Contenu mixte

On déclare un élément comme ayant un *contenu mixte* s'il peut contenir des données textuelles ainsi qu'un certain nombre de sous-éléments.
On écrit alors :

```
<!ELEMENT toto (#PCDATA|titi|tata)*>
```

Dans cet exemple, `toto` peut contenir des éléments `titi`, `tata`, ainsi que des données textuelles.
Attention ! On n'a aucun moyen de spécifier l'*ordre* des sous-élements.

Expr. rég. de sous-éléments 1/2

Si un élément ne contient pas de données textuelles, on peut être plus précis sur l'ordre de ses sous-éléments. On utilise pour cela :

- les opérateur quantitatifs `?` `*` et `+` ;
- la virgule `,` (ordre obligatoire) ;
- le booléen `|` (ordre quelconque) ;
- les parenthèses.

Expr. rég. de sous-éléments 2/2

Exemples :

- `(emph|tt)` : `emph` ou `tt`, une seule fois ;
- `(emph,tt)` : `emph` suivi de `tt`, une seule fois ;
- `(emph*,tt)` : `emph` zéro, une ou plusieurs fois, suivi de `tt` une seule fois ;
- `(pre,p+,epi?)` : `pre` une fois, `p` une ou plusieurs fois, `epi` zéro ou une fois.

Entités paramètre

Pour faciliter l'écriture des contenus mixtes ou des expr. rég. de sous-éléments on dispose d'*entités paramètres*, utilisables uniquement dans des DTD externes.

Déclaration :

```
<!ENTITY % divisions "(h1|h2|h3|h4)*">
```

Utilisation :

```
<!ELEMENT corps (prologue,%divisions;,epilogue)>
```

Exemple de DTD

```
<!ENTITY % text "#PCDATA|b|i">

<!ELEMENT document (p+)>
<!ELEMENT p (%text;|table)*>
<!ELEMENT b (%text;)*>
<!ELEMENT i (%text;)*>
<!ELEMENT table (cellule+)>
<!ELEMENT cellule (%text;|table)*>
```

Déclaration d'attributs

```
<!ATTLIST élément att1 type1 valeur1
          att2 type2 valeur2
          ...
          attn typen valeurn
>
```

où *élément* est le nom de l'élément, *att* le nom de l'attribut, *type* son *type d'attribut*, *valeur* sa *valeur implicite*.

Types d'attributs 1/2

- `CDATA` : une chaîne de caractères sans balises ;
- `ID` : un identificateur *unique* de l'élément.
  Attention ! Sa valeur doit être un *nom* XML ;
- `IDREF` : l'`ID` d'un autre élément ;
- `IDREFS` : plusieurs `ID` séparés par des blancs ;
- `ENTITY` : un nom d'entité déclarée dans la DTD ;
- `ENTITIES` : plusieurs `ENTITY` séparés par des blancs ;
- `NMTOKEN` : un nom XML (commence par une lettre ou un souligné) ;
- `NMTOKENS` : plusieurs noms XML séparés par des blancs ;
- `NOTATION` : un type de *notation* ;
- une liste de valeurs fixes, entre parenthèses et séparées par des |.

Valeurs implicites

- `#REQUIRED` : attribut obligatoire ;
- `#IMPLIED` : attribut facultatif ;
- `#FIXED "`*`valeur`*`"` : attribut à valeur fixe.
- "*`valeur`*" : attribut facultatif avec valeur par défaut.

Exemples d'attributs

```
<!ELEMENT p (#PCDATA|b|i)>
<!ATTLIST p id ID #REQUIRED
      indent (Opt|normal|double) "normal"
      couleur CDATA "noir"
      xml:lang CDATA "br">

<!ELEMENT crossref EMPTY>
<!ATTLIST crossref id IDREF #REQUIRED
      type (page|section) "page">
```

Exemples d'attributs

En utilisant l'exemple précédent :

```
<p id="monnom" indent="normal" couleur="rouge" xml:lang="fr">Mes
parents m'ont appelé <i>Loana</i> parce que...</p>
```

```
<p id="173" couleur="noir" xml:lang="fr">À la page <crossref
id="monnom"/> j'explique pourquoi mes parents...</p>
```

Déclaration d'entité

Entité interne :

```
<!ENTITY nom "texte de remplacement">
```

Entité externe :

```
<!ENTITY nom SYSTEM "fichier ou URL">
```

Exemples :

```
<!ENTITY sncf "S.N.C.F.">
<!ENTITY unesco "<sc>Unesco</sc>">
<!ENTITY chapitre1 SYSTEM "/home/yannis/these/chapitre1.xml">
<!ENTITY logo SYSTEM "http://www.w3c.org/logo.gif">
```

Sections conditionnelles

Pour donner une vague illusion de conditionnalité, on utilise la construction suivante :

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

Exemple récapitulatif : fichier XML

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE loftbook SYSTEM "loftbook.dtd">
<loftbook><chapter id="ch1">
<titre>&letitre;</titre>
<section num="1" id="sec1"><p>Il était une fois...</p> <img
src="../photo.gif"/></section>
</chapter></loftbook>
```

Exemple récapitulatif : fichier DTD

```
<!ELEMENT loftbook (chapter+)>
<!ELEMENT chapter (titre?,section+)>
<!ATTLIST chapter id ID #REQUIRED>
<!ELEMENT titre (#PCDATA)*>
<!ELEMENT section (p|img)*>
<!ATTLIST section id ID #REQUIRED
      num NUMTOKEN #IMPLIED>
<!ELEMENT img EMPTY>
<!ATTLIST img src CDATA #REQUIRED>
<!ELEMENT p (#PCDATA)*>
<!ENTITY letitre "Ma vie et l'intellect">
```

Conception de DTD

- Quels éléments définir ?
- Faut-il être strict dans les contraintes de sous-éléments ?
- Choix entre sous-éléments et attributs.
- Quelles entités définir ?
- Quand faut-il utiliser des instructions de traitement ?
- Danger de sur-structuration.

Les espaces de nommage

Recommandation W3C du 14 janvier 1999

Espace de nommage

Un *espace de nommage* est une manière de définir des éléments et attributs *uniques* dans le monde. On attache une information unique à un nom de balise.

Idée de génie : cette information est une *URL*.

Alias d'URL

Même si l'on prend des URL simples, comme `http://omega.enstb.org`, il serait très maladroit de les attacher telles quelles au noms des balises et des attributs :

```
<http://omega.enstb.org:page>
```

On se sert plutôt d'*alias*.

L'attribut *xmlns:toto*

En écrivant

```
<toto:page xmlns:toto="http://omega.enstb.org">
<toto:p>Quelques lignes</toto:p>
</toto:page>
```

on utilise l'espace de nommage d'alias `toto`, défini à l'intérieur de la balise `page`.

Espace de nommage par défaut

L'exemple précédent peut aussi s'écrire :

```
<page xmlns="http://omega.enstb.org">
<p>Quelques lignes</p>
</page>
```

N'ayant qu'un seul espace de nommage utilisé, on se passe d'*alias*.

Espaces de nommage multiples

Dans l'exemple :

```
<page xmlns="http://omega.enstb.org"
      xmlns:isbn="http://loc.gov/books">
<p>Quelques lignes et un numéro ISBN <isbn:number/></p>
</page>
```

on utilise deux espaces de nommage distincts : le premier est l'espace par défaut, le deuxième est l'espace `isbn`.

Bibliographie

- Spécification de XML : `http://www.w3.org/TR/2004/REC-xml-20040204/` ;
- livres intéressants : *Introduction à XML* par Erik T. Ray, O'Reilly 2001 `http://www.oreilly.fr/catalogue/intro-xml.html`, *XML in a Nutshell* (en français, malgré le titre) par W. Scott Means *ed al.*, O'Reilly, 2002 `http://www.oreilly.fr/catalogue/xml_nutshell_2.html` ;
- sites Web intéressants : `http://www.xmlhack.com/` (en anglais), `http://www.xmlfr.org/` (en français).

Le Web

Historique 1/2

- 1945 : V. Bush décrit dans l'article «As We May Think» sa vision de l'hypertexte et d'Internet ;
- 1957 : les Russes lancent le premier Spoutnik. Conséquences : le PCF atteint un record d'adhésions, les Américains crèent la DARPA (*Defense Advanced Research Projects Agency*) et la NASA ;
- 1962 : l'auteur de ce cours est né et DARPA invente l'*Internet* (appelé *Arpanet*) ;
- 1969 : l'Internet devient réalité, le premier réseau a quatre nœuds : Stanford, UCLA, Santa Barbara, Utah ;
- 1978 : ANSI forme un comité pour étudier GML et GenCode, en 1980 il pond SGML, qui devient une norme ISO en 1986 ;
- 1989 : Tim Berners-Lee développe HTML v1 ;

Historique 2/2

- 1990 : il cherche un nom pour le Web : *Mine Of Information* et *The Information Mine* étant considérés égocentriques, il trouve *WWW*. Le jour de noël de 1990, première connexion à un serveur Web (la machine du bureau voisin) ;
- 1992 : nombre record de sites Web : 50 !
- 1994 : le W3C est fondé, sortie de Netscape v1 ;
- 1995 : sortie d'Explorer v1 et de HTML v2 ;
- 1996 : W3C lance une initiative de simplification de SGML qui va déboucher en XML ;
- 1998 : 4 200 000 serveurs Web ;
- 2000 : sortie de XHTML ;
- 2002 : 617 M d'utilisateurs Internet dans le monde.

Protocoles et technologies

- Le Web est basé sur le *protocole de communication* HTTP ;
- à travers HTTP on envoie des ressources qui le plus souvent des données textuelles *balisées* en (X)HTML, mais elles peuvent aussi être des images, des sons, des animations ;
- il existe des *couches d'information* qui s'ajoutent à (X)HTML : les feuilles de style CSS, le langage JavaScript ;
- il existe des *langages de programmation* qui génèrent du (X)HTML, comme PHP, JSP, etc. ;
- enfin, il existe des moyens de produire du (X)HTML à partir de documents balisés en XML.

HTTP

Version 1.1, 1999 (RFC 2616)

Qu'est-ce que HTTP ?

- HTTP = *Hypertext Transfer Protocol* ;
- *The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems* ;
- *It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers* ;
- *A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred* ;
- pas de Web sans HTTP.

Comment ça marche ?

- le client envoie une requête : une méthode, une URI, une version de protocole, un message MIME ;
- le serveur répond par une ligne d'état (version de protocole, code de réussite ou d'échec) suivie d'un message MIME ;
- la connexion peut être directe ou alors un *proxy*, une *passerelle*, un *tunnel*.

La requête

- Une requête a une partie obligatoire et une partie optionnelle ;
- la partie obligatoire (*request-line*) est une ligne contenant une méthode, une URL, et la version de HTTP. Exemple :

```
GET http://omega.enstb.org/yannis.html HTTP/1.1
```

- suit une ligne blanche et un ensemble de données (appelé une *entité*) ;
- la partie optionnelle est une série d'entêtes, qui se placent entre la ligne de requête et la ligne blanche. Ces entêtes peuvent être de 3 types : entête de *message*, de *requête*, d'*entité*.

Les méthodes

Les méthodes les plus connues sont :

- `GET` : récupérer les données auxquelles on accède à travers l'URL ;
- `POST` : on demande au serveur d'accepter les informations contenues dans le message MIME et de les soumettre à l'URL ;
- `PUT` : on demande au serveur de remplacer les données de l'URL par celles que l'on envoie ;
- `DELETE` : on supprime les données de l'URL ;
- `TRACE` : on demande que la requête soit retournée à l'expéditeur, telle qu'elle a été reçue.

Les URL

- Les URL sont des chaînes de caractères qui identifient une ressource ;
- `http://`*`hôte`*`[:`*`port`*`][`*`chemin Unix`*`[?`*`requête(s)`*`]]`
- l'hôte peut être un nom de domaine ou un numéro IP ;
- le port par défaut est 80 ;
- les requêtes CGI sont des paires «clé = valeur» séparées par des & ;
- les caractères ISO Latin-1 sont obtenus par la syntaxe `%hh`.

Le type MIME

- MIME signifie *Multipurpose Internet Mail Extensions* ;
- le *type MIME* est une classification des données transmises sur le Web. Il consiste en un type principal et un sous-type, séparés par un slash ;
- il y a sept types principaux : `application` (quand des applications communiquent entre elles, par exemple par le biais de SOAP), `audio` (son), `image`, `message` (échange de messages), `model` (par exemple VRML), `multipart` (combinaison de plusieurs types), `text` (le plus fréquent : (X)HTML, XML, RTF, etc.), `video` ;
- les sous-types sont énumérés sur `http://www.iana.org/assignments/media-types/`. Sur cette page on peut également soumettre des nouveaux types MIME ;
- cf. aussi les RFC 2045-2046 (nov. 1996).

Serveur Apache et types MIME

- Le serveur Apache indique le type MIME dans l'entête HTTP selon l'extension de nom de fichier. Les types de base sont décrits dans le fichier `/etc/httpd/conf/mime.types` :

```
audio/x-aiff aif aiff aifc
image/gif gif
image/jpeg jpeg jpg jpe
text/html html htm
text/plain asc txt
text/rtf rtf
...
```

On peut ajouter de nouveaux types MIME (ou modifier la configuration de types existant) dans le fichier `httpd.conf` de la manière suivante :

```
AddType application/x-tar .tgz
```

Quelques entêtes de message

- `Connection` : est-ce qu'on garde la connexion (`Keep-Alive`) ou est-ce qu'elle est terminée (`close`) ?
- `Date` : date d'envoi de la requête ;
- `Pragma` : des infos spécifiques à l'implémentation, ou `no-cache` (ne pas stocker) ;
- `Cache-Control` : faut-il stocker dans le cache, jusqu'à quand, comment, etc. ;
- `Via` : on spécifie les passerelles et proxies par lesquels la requête va passer.

Quelques entêtes de requête

- `Accept` : les types de données que l'on peut accepter en retour, avec éventuellement des préférences. Exemple type :

  ```
  Accept: */*
  ```

  Exemple plus sophistiqué :

  ```
  Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.2
  ```

- `Accept-Language` : pareil, pour les langues (codées en deux caractères). Exemple :

  ```
  Accept-Language: fr, en; q=0.5, jp; q=0.1
  ```

- `Expires` : suivi d'une date, ne pas envoyer de réponse après cette date ;
- `From` : suivi de l'adresse mél de la personne qui envoie la requête ;
- `If-Modified-Since` : suivi d'une date, ne répondre que si les données ont été modifiées depuis cette date.

Quelques requêtes d'entité

L'«entité» est le message envoyé en même temps que la requête.

- `Content-Length` : la longueur du message ;
- `Content-Language` : la langue du message ;
- `Content-Type` : le type MIME (`text/html`, etc.) ;
- `Last-Modified` : dernière date de modification ;
- `Expires` : date d'expiration des données.

La réponse

- La réponse a la même syntaxe que la requête, à deux exceptions près : la première ligne donne l'état de la réponse, et mis à part les entêtes de message et d'entité on a des entêtes spéciales de réponse ;
- la *ligne d'état* consiste en la version de HTTP suivi d'un code d'état et d'une explication du code. Exemple :

  ```
  HTTP/1.1 200 OK
  ```

  ou :

  ```
  HTTP/1.1 404 Not Found
  ```

- les codes sont fixes, les explications peuvent être quelconques ;
- il y a cinq familles de codes : 1** (*information*), 2** (*réussite*), 3** (*redirection*), 4** (*échec de client*), 5** (*échec de serveur*).

Quelques codes de réponse 1/3

- 100 : «*Continue*» (tout va bien pour le moment, mais le serveur n'a pas fini de traiter, donc attendre ou alors envoyer la suite) ;
- 200 : «*OK*» (tout va bien, voici la réponse) ;
- 201 : «*Created*» (une nouvelle ressource a été créée) ;
- 202 : «*Accepted*» (la requête a été acceptée mais le traitement n'a pas fini) ;
- 204 : «*No Content*» (tout va bien sauf qu'il n'y a pas de contenu à retourner, nada) ;
- 301 : «*Moved Permanently*» (la ressource a déménagé, on fait suivre le courrier) ;
- 307 : «*Temporary Redirect*» (la ressource a temporairement changé de place, faire suivre) ;

Quelques codes de réponse 2/3

- 400 : «*Bad Request*» (mauvaise requête, du charabia) ;
- 401 : «*Unauthorized*» (pas le droit d'obtenir la ressource, à moins de se faire autoriser) ;
- 403 : «*Forbidden*» (pas le droit, un point c'est tout) ;
- 404 : «*Not Found*» (la ressource n'existe pas) ;
- 405 : «*Method Not Allowed*» (mauvaise méthode, accompagné d'un entête `Allow` qui donne les bonnes méthodes) ;
- 408 : «*Request Timeout*» (le serveur a attendu trop longtemps) ;

Quelques codes de réponse 3/3

- 411 : «*Length Required*» (il aurait fallu que dans la requête on précise la longueur de l'entité par un `Content-Length`);
- 414 : «*Request-URI Too Long*» (l'UR[IL] est trop longue);
- 500 : «*Internal Server Error*» (le serveur s'est planté);
- 503 : «*Service Unavailable*» (le serveur est temporairement débordé);
- 505 : «*HTTP Version Not Supported*» (mais vous parlez quoi là ?).

Quelques entêtes de réponse

- `Location` : en cas de redirection (codes 3xx), l'URL à suivre ;
- `Server` : le type de serveur.

Exemple de connexion

La requête :

```
GET /yannis HTTP/1.0
Connection: Keep-Alive
Pragma: no-cache
Accept: */*
Accept-Language: en
Host: omega.enstb.org
Referer: http://www.schroepl.net/cgi-bin/http_trace.pl
User-Agent: Mozilla/4.0 (compatible; MSIE 5.23; Mac_PowerPC)
EXTENSION: Security/Remote-Passphrase
UA-CPU: PPC
UA-OS: MacOS
```

Exemple de connexion

La réponse :

```
HTTP/1.0 200 OK
Connection: close
Date: Wed, 08 Sep 2004 10:44:33 GMT
Accept-Ranges: bytes
ETag: "5dc06-cf18-40daaceb"
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) PHP/4.1.2
Content-Length: 53016
Content-Type: text/html
Content-Type: text/html; charset=utf-8
Last-Modified: Thu, 24 Jun 2004 10:28:59 GMT
Client-Date: Wed, 08 Sep 2004 10:03:44 GMT
Client-Response-Num: 1
Title: Curriculum Vitae : Yannis Haralambous
```

suivie de 53 016 octets de données.

Autre exemple de connexion

```
HTTP/1.0 200 OK
Connection: close
Date: Wed, 08 Sep 2004 13:22:13 GMT
Accept-Ranges: bytes
ETag: "b8d1a-214c-40bdcd08"
Server: Apache/1.3.22 (Unix) PHP/4.0.4pl1 mod_fastcgi/2.2.10 PHP/3.0.16
FrontPage/4.0.4.3 mod_perl/1.27 tomcat/1.0
Content-Length: 8524
Content-Type: text/html
Last-Modified: Wed, 02 Jun 2004 12:50:16 GMT
Brest et Rennes : diplôme d'ingénieur, mastères, formation continue, recherche,
grands projets européens
Projets européens,Mastères,Master of Science, MSC, Grandes, Ingénieur Télécom,
Ingénieur en Télécommunications, Recherche Brest, Brest Recherche,
écoles,Bretagne,Rennes,Campus,Mer,Sports,Langues, Grandes écoles,
Technique,telecommunications,ecole,ingenieur,superieur,europeen,mastere
Client-Date: Wed, 08 Sep 2004 13:22:12 GMT
Client-Response-Num: 1
Title: Ecole Nationale Supérieure des Télécommunications de Bretagne
X-Meta-Description: Grande école française de télécommunications
X-Meta-Generator: CuteHTML
X-Meta-Keywords: Télécommunications,Ecole,Ingénieur,Formation,
Supérieur,Brest,Recherche,Entreprendre,Alternance,
```

suivie de 8 524 octets de données...

XHTML

Version 1.1, du 31 mai 2001

Qui hérite de qui

- *SGML* est un système de balisage générique. Tellement générique qu'on a du mal à développer des outils pour s'en servir ;
- *HTML* est une application de SGML. Les balises HTML servent à décrire des pages Web. HTML est rudimentaire et il a fallu d'autres technologies (CSS, DOM, JavaScript) pour le rendre utilisable professionnellement ;
- *XML* est un remake de SGML avec quelques simplifications judicieusement choisies. Bon compromis entre puissance et simplicité ;
- *XHTML* est la version XMLisée de HTML. Aussi rudimentaire que HTML mais écrit dans une syntaxe meilleure. Profite des *espaces de nommage*.

Exemple de document XHTML

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="fr">
<head><title>Bonjour le monde !</title></head>
<body><p>Bonjour le monde !</p></body>
</html>
```

Décorticage de l'exemple 1/2

- `<?xml version="1.0" encoding="utf-8"?>` : c'est la *déclaration XML* qui dit : Attention ! Ce qui suit est du XML, version 1.0, codage Unicode UTF-8 ;
- `<!DOCTYPE html ... >` : la DTD utilisée est `html`. Une DTD contient la déclaration de toutes les balises et la description de leur comportement mutuel ;
- `PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"` : une manière plus précise de décrire la DTD `html` ;
- `"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"` : l'endroit où se trouve la version officielle de cette DTD ;
- `<html ... > ... </html>` : l'élément principal du document ;

Décorticage de l'exemple 2/2

- `xmlns="http://www.w3.org/1999/xhtml"` : l'espace de nommage XHTML. Alors que la DTD n'indique que les noms et le comportement des balises, l'espace de nommage détermine leur sémantique ;
- `xml:lang="fr"` : jusqu'à indication du contraire, le document est en français ;
- `<head> ... </head>` : le préambule du document ;
- `<title> ... </title>` : le titre du document, tel qu'il apparaît sur la barre de la fenêtre du navigateur ;
- `<body> ... </body>` : le corps du document ;
- `<p> ... </p>` : un paragraphe de texte.

Différences entre HTML et XHTML

- Imbrication stricte ;
- on écrit *toutes* les balises d'ouverture et de fermeture ;
- l'élément vide s'écrit avec une barre à la fin : `<br/>` ;
- les valeurs d'attribut sont toujours protégées par des quotes ou des double-quotes ;
- certains éléments ont été limogés : `font`, `basefont`, `center`, `s`, `strike`, `u`, `dir`, `menu`, `isindex` ;
- XHTML est *modulaire*.

Module 1 : Structure de document

- `html` : balise principale, attributs : `version`, `xmlns` ;
- `head` : préambule du document ;
- `title` : titre du document ;
- `body` : corps du document, attributs : `alink`, `background`, `bgcolor`, `link`, `text`, `vlink`.

Module 2 : Texte 1/2

- `abbr` : abréviation : `mon <abbr title="psychanalyste">psy</abbr>` ;
- `acronym` : acronyme : `<acronym title="Patrick Poivre d'Arvor">PPDA</abbr>` ;
- `address` : adresse ;
- `blockquote` : citation longue ;
- `br` : saut de ligne ;
- `cite` : référence de la source d'une citation ;
- `code` : code informatique ;
- `dfn` : définition d'un terme ;
- `div` : bloc de texte générique ;
- `em` : mode emphatique ;

Module 2 : Texte 2/2

- `h1-h6` : subdivisions de niveau 1 à 6 ;
- `kbd` : code écrit sur la ligne de commande ;
- `p` : paragraphe de texte ;
- `pre` : texte préformatté ;
- `q` : citation courte ;
- `samp` : exemple ;
- `span` : élément générique porteur d'instructions CSS ;
- `strong` : mode «fort» ;
- `var` : variable de programme informatique.

Module 3 : Hypertexte

- `a` : lien hypertexte ou cible de lien hypertexte, attributs :
  - `href` : l'URL de la cible,
  - `name` (ou `id`) : l'identifiant du nœud courant,
  - `rel` : le rapport de la cible avec la page actuelle. Cet attribut doit utiliser des mots-clé parmi les suivants : `alternate`, `appendix`, `bookmark`, `chapter`, `contents`, `copyright`, `glossary`, `help`, `index`, `next`, `prev`, `section`, `start`, `stylesheet`, `subsection`,
  - `type` : le type MIME de la cible.

Module 4 : Listes

- `ol` : liste énumérative, attributs : `compact`, `start` (la valeur par laquelle on commence à compter) ;
- `ul` : liste non-énumérative, attribut : `compact` ;
- `li` : entrée de liste ;
- `dl` : liste de définitions, attribut : `compact` ;
- `dt`, `dd` : terme défini, définition du terme.

Module 5 : Présentation

- `b`, `i`, `tt` : gras, italique, machine à écrire ;
- `big`, `small` : grand corps, petit corps ;
- `sub`, `sup` : indice, exposant ;
- `hr` : filet horizontal, attribut : `noshade` (pas d'effet 3D).

Mod. 6, 7 : Édition, bidirectionnalité

- `del` : texte supprimé, attributs : `cite` (la raison pour laquelle le texte est supprimé), `datetime` (date et heure de la suppression, au format *yyyy*`-`*MM*`-`*dd*`T`*hh*`:`*mm*`:`*ss* suivi de l'indicateur de zone horaire (`Z` = «zoulou», ou `+`*hh*`:`*mm* ou `-`*hh*`:`*mm*) ;
- `ins` : texte inséré, mêmes attributs que `del` ;
- `bdo` : directionnalité explicite, attribut `dir` (valeurs `rtl` droite à gauche, ou `ltr` gauche à droite).

Module 8 : Formulaires 1/2

- `form` : un formulaire, attributs : `action` (l'URL à laquelle on envoie les données), `method` (`GET` ou `POST`) ;
- `fieldset`, `legend` : une partie de formulaire et sa légende ;
- `input` : un champ ou un bouton, attributs : `type` (le type de champ : `button`, `checkbox`, `file`, `hidden`, `image`, `password`, `radio`, `reset`, `submit`, `text`), `accept` (les types MIME de fichiers à envoyer), `alt` (description alternative), `checked` (coché ou non), `disabled` (désactivé ou non), `maxlength` (longueur maximale), `name` (nom interne), `readonly` (lecture seule ou non), `size` (taille visible), `src` (source de l'image, si `type` est `image`), `value` (valeur) ;
- `textarea` : un champ de saisie de texte long, attributs : `cols` (largeur en caractères), `disabled`, `name`, `readonly`, `rows` (nombre de lignes) ;

Module 8 : Formulaires 2/2

- `button` : un bouton, l'avantage étant que la légende du bouton est le contenu de `button` et peut donc contenir d'autres balises. Attributs : `name`, `type` (à choisir entre `button`, `reset`, `submit`), `value` ;
- `label` : une manière d'associer une légende à un champ (`<label>Nom <input type="text" name="nom"/></label>`) ;
- `select` : une liste d'options `option`. Attributs : `disabled`, `multiple` (liste à choix multiple ou non), `name`, `size` (nombre d'options visibles) ;
- `optgroup` : un groupe d'options `option`. Attributs : `disabled`, `label` (l'entrée de menu pour ce groupe) ;
- `option` : une option, attributs : `disabled`, `selected` (option sélectionnée ou non), `value` (valeur transmise au serveur).

Module 9 : Tableaux 1/2

- `table` : un tableau. Attributs : `align` (alignement du tableau par rapport au contexte), `bgcolor`, `border` (épaisseur du filet extérieur en pixels), `cellpadding` (blanc rajouté dans chaque cellule), `cellspacing` (épaisseur du filet intérieur), `frame` (indique le côté du tableau muni d'un filet : `above`, `below`, `border`, `hsides`, `vsides`, `lhs`, `rhs`, `void`), `width` (largeur du tableau, en pourcentage), `rules` (indique les filets internes : `all`, `cols`, `rows`, `groups`, `none`), `summary` (une description alternative) ;
- `caption` : la légende du tableau ;
- `thead`, `tbody`, `tfoot` : le préambule, corps et postambule du tableau. Attributs : `align` (alignement horizontal par défaut du contenu des cellules), `valign` (idem, mais vertical), `bgcolor` ;
- `tr` : une ligne du tableau. Attributs : `align` et `valign`, `bgcolor` ;

Module 9 : Tableaux 2/2

- `td` : une cellule du tableau. Attributs : `abbr` (une version abrégée du contenu de la cellule), `align` et `valign`, `axis` (un mot-clé définissant la catégorie de la cellule), `bgcolor`, `colspan` (nombre de colonnes pour une cellule multi-colonnes), `headers` (pointe vers l'attribut `id` de la cellule `th` correspondante), `width` et `height` (largeur et hauteur en pourcentage ou en pixels), `nowrap` (ne pas passer à la ligne), `rowspan` (nombre de lignes pour les cellules multi-lignes) ;
- `th` : une cellule privilégiée (généralement dans l'en-tête), mêmes attributs que `td` ;
- `col` : les spécifications d'une colonne du tableau, attributs : `align` et `valign`, `span` (nombre de colonnes pour lesquelles les spécifications s'appliquent), `width` ;
- `colgroup` : les spécifications d'un groupe de colonnes du tableau, mêmes attributs que `col`.

Module 10 : Objets externes

- `object` : chargement par le navigateur d'un «objet» (applet Java, plug-in, etc.). Attributs : `align`, `archive` (préchargement de données nécessaires à l'objet), `data` (l'URL de l'objet), `declare` (le préchargement de l'objet, sans instantiation), `height` et `width`, `hspace` et `vspace` (de l'espace autour de l'objet), `name`, `standby` (du texte à afficher pendant que l'objet se charge), `type` (le type MIME de l'objet) ;
- `param` : un paramètre de l'objet. Attributs : `name`, `type`, `value`, `valuetype` (`data` = données, `object` = le paramètre est un autre objet de la page, `ref` : le paramètre est une URL).

Module 11 : Roubi

- Les *roubi* sont les annotations idéographiques (souvent des *kana* qui indiquent la prononciation des *kanji*) ;
- `ruby`, `rb`, `rt` : les roubi, le texte de base, l'annotation ;
- `rp` : du code à ajouter si le navigateur ne sait pas gérer les *roubi* (par exemple des parenthèses) ;
- `rbc`, `rtc` : un autre regroupement des *roubi*.


```
<ruby><rb>日</rb><rt>に</rt><rb>本</rb>
<rt>ほん</rt><rb>語</rb><rt>ご</rt></ruby>
```

Module 12 : Images

- `img` : une image (élément vide). Attributs : `align`, `alt` (description alternative), `border` (épaisseur du filet), `height` et `width`, `hspace` et `vspace`, `ismap` (si cet attribut existe, et si l'image est incluse dans un lien hypertexte, le navigateur envoie les coordonnées du clic de l'utilisateur à la suite de l'URL du lien), `longdesc` (l'URL d'une description plus longue de l'image), `name` ou `id`, `src` (l'URL du fichier qui contient l'image), `usemap` (un mot-clé (précédé d'un *hash* #) indiquant le nom de l'élément «carte» `map` qui contient des régions rectangulaires, circulaires ou polygonales et des liens hypertexte associés).

Module 13 : Cartes

- `map` : une «carte» associée à une image est une collection de zones rectangulaires (`area`), chacune associée à un lien hypertexte ;
- `area` : une zone de carte, attributs : `shape` (type de la zone : `circle`, `rect` ou `poly`), `coords` (les coordonnées séparées par des virgules : dans le cas du cercle, ce sont les coordonnées du centre et le rayon, dans le cas du rectangle les coordonnées des points inférieur gauche et supérieur droit, dans le cas du polygone, les coordonnées de tous les points), `href` (l'URL de la cible du lien hypertexte), `alt`.

Module 14 : Métadonnées

- `meta` : des méta-informations concernant le document tout entier et placées dans le préambule. Attributs : `name` et `content` (nom et valeur), `http-equiv` et `content` (un champ HTTP et sa valeur), `scheme` (un niveau organisationnel au-dessus).

Exemples :

```
<meta http-equiv="Refresh" content="1,http://quelquepart.com"/>
<meta http-equiv="content-type" content="text/html;
charset=windows-1252">
<meta http-equiv="Expires" content="Sun, 21 Apr 2002 12:00:00 CET">
```

Module 15 : Scripts

- `script` : du code dans un langage de scriptage. Attributs : `type` (type MIME du langage de scriptage, par exemple `text/javascript`), `language` (nom du langage de script), `src` (l'URL d'un fichier contenant le code), `charset` (le codage de ce fichier) ;
- `noscript` : texte à afficher si le navigateur ne reconnaît pas le langage de scriptage.

Attention :

```
<script type="text/javascript">
<!-- if (1 < 2) alert("C'est cuit !"); --></script>
<script type="text/javascript">
<![CDATA[ if (1 < 2) alert("C'est cuit aussi !"); ]]>
</script>
```

Modules 16-18 : Styles, Liens, Base

- `style` : du code dans un langage de feuille de style. Attributs : `type` (type MIME du langage de feuille de style, par exemple `text/css`), `media` (valeurs possibles : `all`, `aura`, `braille`, `handheld`, `print`, `projection`, `screen`, `tty`, `tv`) ;
- `link` : un lien placé dans le préambule. Attributs : `href` ou `src` [NN] (une URL), `charset` (son codage), `hreflang` (sa langue), `media`, `rel` (le rapport entre la page courante et le lien, aussi `stylesheet` [IE4+,NN4+], `fontdef` [NN4]), `type` (le type MIME du lien) ;
- `base` : l'URL du document courant, tous les chemins relatifs le seront par rapport à cette URL, attribut : `href`.

Cascading Style Sheets

CSS version 2, du 31 mai 2001

Où sont les cascades ?

- *CSS* est une syntaxe de règles de présentation de documents XHTML (et plus généralement XML) ;
- dans CSS on a des *règles* qui contiennent des *sélecteurs* et des *déclarations*. Selon les sélecteurs plusieurs règles peuvent s'appliquer au même éléments. CSS donne un ordre de priorité à ces règles. On assiste donc à une «cascade» de propriétés (orthogonales ou non) qui sont appliquées dans un certain ordre.

Exemple

```
<style type="text/css">
  em { color:pink; text-decoration: underline; }
  .important { color:green; text-transform: uppercase; }
  p em { color:blue; font-family: "Helvetica"; }
  p > em { color:red; font-size: 24pt; }
</style>
```

Que donnera alors le code suivant ?

```
<p><em class="important">Quelques mots, comment seront-ils
présentés ??</em></p>
```

Voici le résultat :

QUELQUES MOTS, COMMENT SERONT-ILS PRÉSENTÉS ??

Où écrit-on les règles ?

- On écrit les règles dans un élément `style` :

```
<style type="text/css">
  p { color:red; }
</style>
```

- ou dans l'instance de l'élément à présenter :

```
<p style="color:red;">Bla bla.</p>
```

- ou dans un fichier externe :

```
<style type="text/css">
  @import url("./toto.css");
</style>
```

Syntaxe générale

- Une *règle* est toujours de la forme :
  sélecteur { *déclarations* }
- Une *déclaration* est toujours de la forme :
  propriété: *valeur*;
- Il s'agit donc d'étudier les sélecteurs, les propriétés et les valeurs.

Sélecteurs 1/4

- `toto` : la règle s'applique à tous les éléments `toto` (= sélecteur par type);
- `toto, titi, tata` : la règle s'applique à tous les éléments `toto`, `titi` et `tata` (= groupage);
- `*` : la règle s'applique à tous les éléments (= sélecteur universel);
- `toto.qqch` : tous les `toto` dont l'attribut `class` prend la valeur `qqch` (= sélecteur par classe);
- `.qqch` : tous les éléments dont l'attribut `class` prend la valeur `qqch` (= sélecteur par classe universel);
- `toto#s31` : l'élément dont l'attribut `id` prend la valeur `s31` (= sélecteur par identificateur);

Sélecteurs 2/4

- `toto titi` : tout `titi` qui est descendant de `toto` (= sélecteur de descendant) ;
- `toto > titi` : tout `titi` qui est enfant direct de `toto` (= sélecteur d'enfant) ;
- `toto + titi` : tout `titi` qui est frère de `toto` et qui le suit directement (= sélecteur de frère adjacent) ;
- `toto[fifi]` : tout `toto` muni d'un attribut `fifi` (= sélecteur par existence d'attribut) ;
- `toto[fifi="oui"]` : tout `toto` dont l'attribut `fifi` prenne la valeur `oui` (= sélecteur par valeur d'attribut) ;

Sélecteurs 3/4

- `toto[fifi~="pomme"]` : tout `toto` dont l'attribut `fifi` contient un certain nombre de chaînes séparées par des blancs, dont aussi la valeur `pomme` (= sélecteur par liste de valeurs d'attribut) ;
- `toto[lang|="fr"]` : tout `toto` dont l'attribut `lang` prenne la valeur `fr` suivie éventuellement d'un trait d'union et d'autres caractères (= sélecteur par début de valeur d'attribut) ;
- `toto:titi`, où `:titi` peut être un «pseudo-élément» parmi les suivants : `:first-letter` (la première lettre de `toto`) [NN6+,IE5.5+], `:first-line` (la première ligne de `toto`) [NN6+,IE5.5+], `:before` (l'espace avant l'élément) [NN6+], `:after` (l'espace après l'élément) [NN6+] ;

Sélecteurs 4/4

- `toto:titi`, où `:titi` peut être une «pseudo-classe» parmi les suivantes :
  - `:active` : un élément `a` au moment ou l'on clique dessus [NN6+,IE4+],
  - `:first-child` : un élément qui est le premier enfant d'un autre élément [NN6+,IE/M5+],
  - `:focus` : un élément qui a le focus [NN6+,IE/M5+],
  - `:hover` : un élément traversé par le pointeur [NN6+,IE4+],
  - `:lang(`*`langue`*`)` : un élément qui a le code langue *`langue`* [IE/M5+],
  - `:link` : un élément `a` qui n'a pas encore été visité [NN6+,IE4+],
  - `:visited` : un élément `a` déjà visité [NN6+,IE4+].

Règles «arobase»

Mis à part les règles décrites dans les transparents précédents, dans une feuille de style on trouve également des *règles arobase* :

- `@import url(`*`toto.css`*`)` : importation du fichier `toto.css` qui contient une feuille de style [NN6+,IE4+] ;
- `@charset` *`codage`* : codage du fichier en question [NN6+,IE5+] ;
- `@media` *`médium`* `{` *`règles`* `}` : des règles à appliquer lorsque le médium est *`médium`* (par exemple `print`, `screen`, `braille`, etc.) [NN6+,IE/W5+] ;
- `@font-face { `*`règles`*` }` : si parmi les propriétés il y a une propriété `src` dont la valeur est une URL, et un propriété `font-family` dont la valeur est un nom de famille de fonte, alors la fonte pointée par l'URL devient associée au nom de famille de fonte [IE/W4+].

Propriétés : fond de bloc

- `background-attachment` : lorsque l'on a une image de fond, est-elle fixe ou pas ? Valeurs : `fixed` ou `scroll` ;
- `background-color` : couleur de fond ;
- `background-image` : image de fond (URI) ;
- `background-position-x` : coordonnée horizontale du coin supérieur gauche de l'image de fond. Valeurs : un pourcentage, ou une longueur, ou un mot-clé parmi `left`, `center`, `right` ;
- `background-position-y` : coordonnée verticale du coin supérieur gauche de l'image de fond. Valeurs : un pourcentage, ou une longueur, ou un mot-clé parmi `top`, `center`, `bottom` ;
- `background-repeat` : pavage ou non du fond par l'image. Valeurs : `no-repeat`, `repeat`, `repeat-x`, `repeat-y` ;

Propriétés : filets 1/2

- `border-*-color`, où `*` peut être `bottom`, `left`, `right`, `top` : couleur du filet d'un élément ;
- `border-color` : idem, mais suivi d'une à quatre valeurs (une valeur = tous les côtés prennent la même valeur ; deux valeurs = les filets horizontaux prennent la première valeur, les verticaux la deuxième ; trois valeurs = haut, droite + gauche, bas ; quatre valeurs : haut, droite, gauche, bas) ;
- `border-*-style` : style du filet d'un élément. Valeurs : `double`, `groove` (= rainure), `hidden`, `inset`, `none`, `outset`, `ridge` (= crête), `solid` ;
- `border-style` : idem, mais suivi d'une à quatre valeurs (selon le modèle de `border-color`) ;

Propriétés : filets 2/2

- `border-*-width`, où `*` peut être `bottom`, `left`, `right`, `top` : largeur du filet d'un élément ;
- `border-width` : idem, mais suivi d'une à quatre valeurs (selon le modèle de `border-color`) ;
- `border-collapse` : est-ce que les filets sont tracés séparement ou fusionnés ? Valeurs : `collapse` ou `separate` ;
- `border-spacing` : espace entre les cellules, suivi d'une ou de deux valeur(s) (une valeur = espace horizontal et vertical ; deux valeurs = horizontal, vertical) ;

Propriétés : flottement, *clipping*

- `clear` : est-ce que le bloc en question peut être aligné avec d'autres blocs ou non ? Valeurs : `both`, `left`, `right`, `none` ;
- `float` : indique de quelle côté le bloc est «flottant», c'est-à-dire peut s'aligner avec d'autres blocs. Valeurs : `left`, `right`, `none` ;
- `clip: rect(`*`t r b l`*`)` : du bloc en question on ne voit que la partie contenue dans le rectangle *`t r b l`* (haut, droite, bas, gauche) ;

Propriétés : affichage de bloc 1/4

- `content` : à utiliser avec les pseudo-éléments `:before` et `:after` [NN6+] pour introduire du contenu avant ou après un élément. Valeurs possibles : une chaîne, un URI, `attr(`*`attribut`*`)` (la valeur d'un attribut), `open-quote`, `close-quote`, `no-open-quote`, `no-close-quote` ;
- `cursor` : change le curseur quand il traverse le bloc en question. La valeur peut être un type (une bonne trentaine) ou un URI ;
- `display` : la manière d'afficher le bloc. Valeur la plus intéressante : `none` [NN6+,IE4+] ;
- `visibility` : visibilité du bloc. Valeurs possibles : `collapse`, `hidden`, `visible`, `inherit` ;
- `height` (resp. `width`) : la hauteur (resp. largeur) du bloc. Valeurs : une longueur, un pourcentage, ou `auto` ;

Propriétés : affichage de bloc 2/4

- `left` (resp. `right`, `top`, `bottom`) : position de la partie gauche (resp. droite, haute, basse) du bloc. Valeurs : une longueur, un pourcentage, ou `auto` ;
- `margin-*` : les quatre marges (`*` comme pour `border-*-color`). Valeurs : une longueur ou `auto` ;
- `padding-*` : blanc à l'intérieur du bloc (`*` comme pour `border-*-color`). Valeurs : une longueur ou un pourcentage ;
- `position` : est-ce que le bloc est positionnable, et comment ? Valeurs possibles : `absolute` (position absolue dans la fenêtre, défile avec le contenu) [NN4+,IE4+], `fixed` (position fixe par rapport au bord de la fenêtre, ne défile pas) [NN6+,IE/M5+], `relative` (pas de positionnement explicite, mais devient un répère pour ses descendants) [NN4+,IE4+], `static` (pas de positionnement explicite) [NN6+,IE4+] ;

Propriétés : affichage de bloc 3/4

- `max-height`, `min-height` : hauteur maximale et minimale pour un bloc donné. Lorsque le contenu dépasse la hauteur maximale, c'est la propriété `overflow` qui indique ce qu'il faut faire. Valeurs : longueur, pourcentage ou `none` ;
- `overflow` : que se passe-t-il lorsque le contenu excède la hauteur fixée de son contenant ? Valeurs possibles : `auto`, `hidden`, `scroll` (barres de défilement), `visible` ;
- `max-width`, `min-width` : largeur maximale et minimale pour un bloc donné. Valeurs : longueur, pourcentage ou `none` ;

Propriétés : affichage de bloc 4/4

- `vertical-align` : alignement vertical par rapport au contexte. Valeurs possibles : `bottom`, `top`, `baseline`, `middle`, `sub`, `super`, `text-bottom`, `text-top`, une longueur ou un pourcentage ;
- `z-index` : lorsqu'un bloc est positionné, est-ce qu'il est au-dessus ou au-dessous des autres éléments ? On définit une pile de blocs (dans le sens de l'écran vers l'humain), cette propriété prend une valeur entière (la hauteur dans la pile). Lorsque deux éléments ont la même valeur, c'est leur ordre dans le code XHTML qui définit leur position transversale.

Propriétés : fontes

- `font-family` : une suite de noms de familles de fontes. Ces noms peuvent être explicites ou génériques (`serif`, `sans-serif`, `cursive`, `fantasy`, `monospace`) ;
- `font-size` : le corps de la fonte. Valeurs : une dimension absolue (`xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`) ou une dimension relative (`larger`, `smaller`), ou une longueur, ou un pourcentage ;
- `font-style` : `normal`, `italic` ou `oblique` ;
- `font-variant` : `normal` ou `small-caps` ;
- `font-weight` : `bold`, `bolder`, `lighter`, `normal` ou $100 \cdot n$ avec $n \in \{1, \ldots, 9\}$;

Propriétés : texte 1/2

- `color` : couleur de texte ;
- `direction` : la direction du texte (`ltr` ou `rtl`) ;
- `letter-spacing` : interlettrage (à éviter !). Valeurs : `normal` ou une longueur positive ou négative ;
- `line-height` : interlignage. Valeurs : `normal`, un entier, une longueur ou un pourcentage ;
- `quotes` : quels sont les guillemets ? Valeurs : quatre chaînes (guillemet ouvrant de 1er ordre, guillemet fermant de 1er ordre, guillemet ouvrant d'ordre supérieur, guillemet fermant d'ordre supérieur) ;
- `decoration` : l'enrichissement de mauvais goût du texte : `blink`, `linethrough`, `overline`, `underline`, `none` ;

Propriétés : texte 2/2

- `text-align` : la justif du texte. Valeurs : `center`, `justify` [NN6+,IE5+], `left`, `right` ;
- `text-indent` : le retrait. Valeurs : une longueur ou un pourcentage ;
- `text-transform` : capitalisation. Valeurs : `uppercase`, `lowercase`, `capitalize` ou `none` ;
- `unicode-bidi` : comment mélanger des textes avec des directions d'écriture différentes. Valeurs : `bidi-override`, `embed`, `normal` ;
- `white-space` : comment gérer les blancs ? Valeurs : `normal` (blancs fusionnés, CR/LF = blanc), `nowrap` (blancs respectés, CR/LF = blanc), `pre` (blancs et CR/LF respectés) ;

Propriétés : listes

- `list-style-image` : l'image à utiliser comme marqueur de liste. Valeurs : `none` ou un URI ;
- `list-style-position` : est-ce que le marqueur se place dans la marge ou dans la justification du texte ? Valeurs : `inside`, `outside` ;
- `list-style-type` : type de marqueur. Valeurs : `circle`, `disc`, `square`, `decimal`, `decimal-leading-zero`, `lower-roman`, `upper-roman`, `lower-greek`, `lower-alpha`, `upper-alpha`, `hebrew`, `armenian`, `georgian`, `cjk-ideographic`, `hiragana`, `katakana` ;

Propriétés : roubi

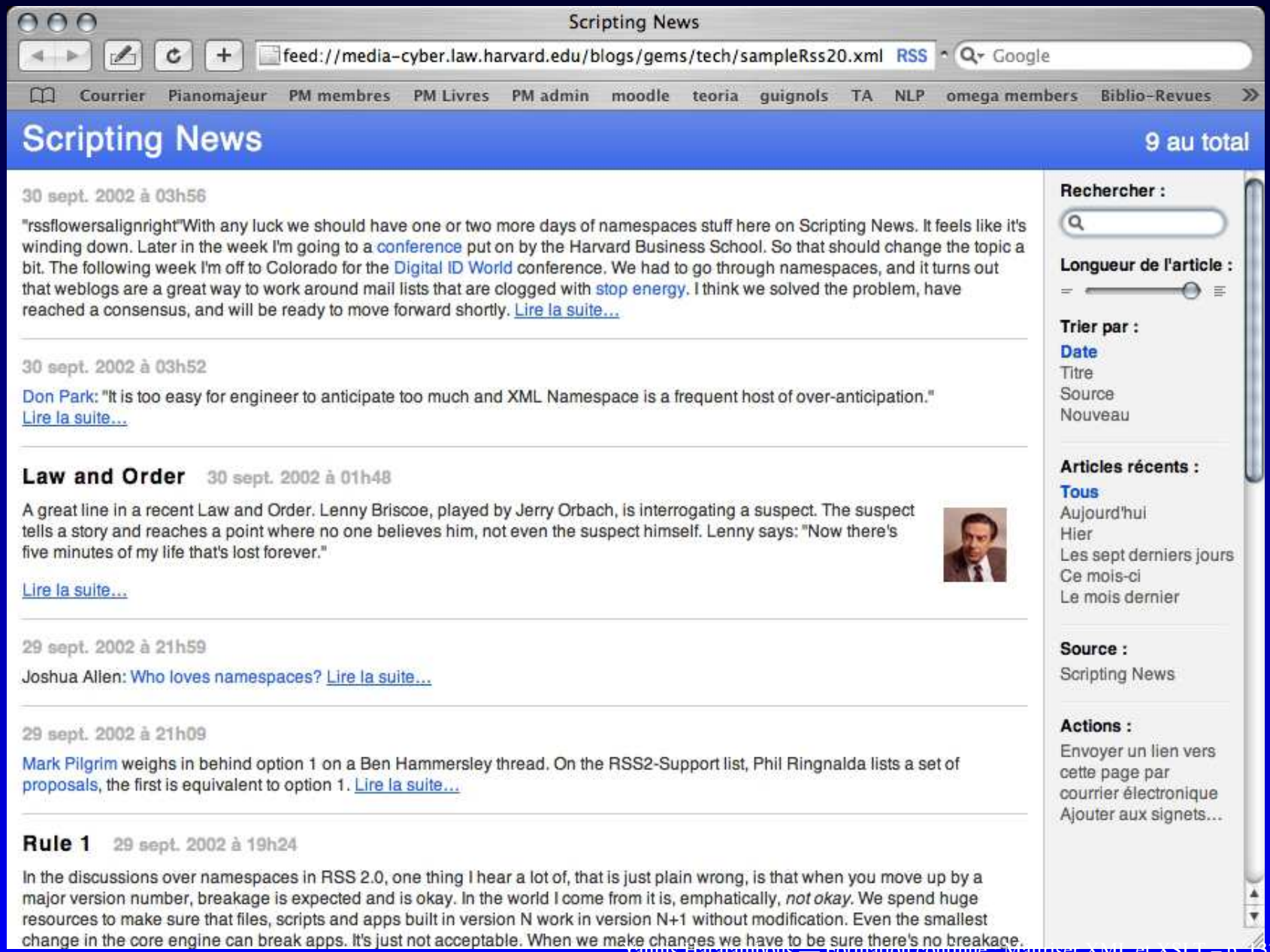
- `ruby-align` : alignement des roubi au-dessus des kanji. Valeurs : `auto`, `center`, `distribute-letter`, `distribute-space`, `left`, `line-edge`, `right` [IE5+] ;
- `ruby-overhang` : que faire lorsque les roubi dépassent le kanji ? Valeurs : `none`, `auto`, `whitespace` [IE5+] ;
- `ruby-position` : est-ce que l'on place les roubi au-dessus du kanji (courageux), ou à côté (lâche) ? Valeurs : `above`, `inline`.

RSS
La syndication des données

Qu'est-ce que RSS ?

- RSS = *Really Simple Syndication* ou *RDF Site Summary* ou ... ;
- RSS est un système de *syndication de données*, c'est-à-dire de mise à disposition de petits blocs de texte, que certains logiciels affichent de manière synthétique et à qui on peut, sous certaines conditions, s'abonner ;
- l'exemple typique est un service de dépêches d'une agence d'information comme *Associated Press*, l'annonce de mises à jour d'un logiciel, etc. ;
- les *annuaires* (en anglais : *registry*) sont des sites qui accumulent et tentent de classifier des milliers d'URLs de flux (en anglais : *feed*) ;
- les *agrégateurs* sont des logiciels qui, éventuellement, filtrent les flux, les combinent en méta-flux, etc.
- un des intérêts de RSS est que l'on peut créer une page Web en republiant les flux des autres...

Copie écran d'agrégateur RSS



Les différents formats

- Il y a au moins cinq formats de fichier, présentés comme «versions» de RSS :
- la version 0.91 (Netscape, juillet 1999). Il est *pull based* c'est-à-dire que l'utilisateur doit explicitement demander l'actualisation des données. 15 entrées maximum ;
- la version 0.92 (Userland Software, décembre 2000). Une évolution de 0.92. On peut s'abonner. Un nombre illimité d'entrées ;
- la version 1 (groupe de travail RSS-DEV, également décembre 2000) : utilise les espaces de nommage et RDF, permet l'abonnement ;
- la version 2 : (Userland Software, août 2002) dernière évolution de 0.92 ;
- «Atom» : (Internet Engineering Task Force).

Technicités

Le type MIME de RSS v2 est `application/rss+xml`

«Protocole» `feed` :

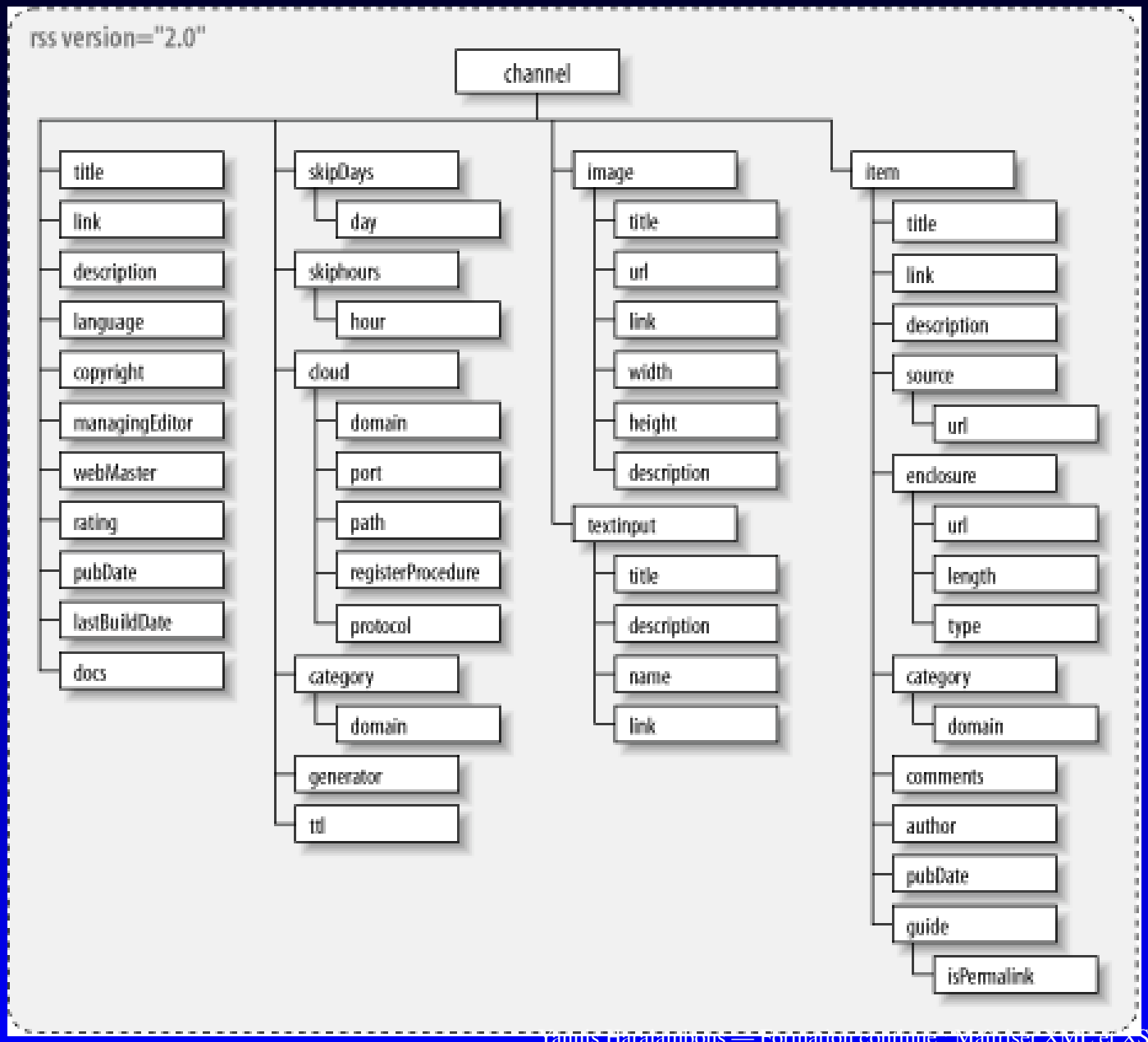
- pour afficher un flux on peut aussi utiliser `http` ;
- on est obligé d'utiliser `feed` pour les abonnements et désabonnements. Exemple :

  `feed://addfeed/title=Slash%20Dot/image=http%3a%2f%2f images.slashdot.org%2ftitle.gif/http://slashdot.org/index.rss`

Auto-découverte de flux RSS : mettre dans le code XHTML la ligne

```
<link rel="alternate" type="application/rss+xml" title="RSS"
href="http://moi.com/mon-flux-rss.xml">
```

Les éléments de RSS 2



Les éléments de RSS v2

- `rss` et `channel` : un flux de données ;
- `title`, `link`, `description` : l'intitulé du flux, son URL et une description ;
- `language` : langue du flux (code ISO) ;
- `copyright` : copyright ;
- `managingEditor`, `webMaster` : mél du responsable éditorial et du responsable technique ;
- `rating` : des métadonnées PICS, entre autres le fait si le flux est pour adultes uniquement, dans quelle mesure et pourquoi (du genre : «Harmless conflict; some damage to objects», «Creatures injured or killed; damage to objects», «Humans injured or killed with small amount of blood», «Humans injured or killed; blood and gore», «Wanton and gratuitous violence; torture; rape» ou, dans une autre échelle :«Romance; no sex», «Passionate kissing», «Clothed sexual touching», «Non-explicit sexual activity», «Explicit sexual activity; sex crimes») ;

Les éléments de RSS v2

- `pubDate` et `lastBuildDate` : date de publication et date de dernière modifications (en format RFC 822) ;
- `docs` : une URL qui pointe vers la description du format RSS utilisé (ne pas confondre avec l'espace de nommage) ;
- `skipHours` et `skipDays` : les heures et les jours que l'agrégateur peut ignorer ;
- `cloud` : possibilité de s'abonner à ce flux en utilisant SOAP ou XML-RPC ;
- `category` : quelques mots-clé ;
- `generator` : le logiciel qui a généré le flux (un peu de pub) ;
- `ttl` : «Time to live» d'après la chanson de Bob Dylan et des Byrds, la durée du vie du flux dans le cache avant d'être renouvellé ;
- `image` : une image qui peut accompagner le flux. Trois sous-éléments obligatoires : `url` (l'URL du flux en tant que RSS), `title`, `link` (l'URL du flux quand il est représenté en XHTML) ;

Les éléments de RSS v2

- `textInput` : zone de saisie de texte qui peut être incluse dans le flux. Dans la spec on lit : «Le but de l'élément `<textInput>` est presque un mystère. Vous pouvez l'utiliser pour spécifier une boite de recherche. Ou pour permettre à un lecteur de vous envoyer des commentaires. La plupart des agrégateurs l'ignorent.» ;
- `item` : l'entrée de flux.

Les sous-éléments de `item` 1/2

- `title`, `link`, `description` : idem que pour `channel` ;
- `source` : son contenu doit être identique à celui de `title` du flux ;
- `enclosure` : possibilité d'inclure une ressource avec l'entrée, par exemple un fichier MP3. Il faut donner l'URL, le type MIME et la taille de la ressource :

```
<enclosure url="http://qqpart.com/tralala.mp3" length="1221632"
type="audio/mpeg" />
```

- `category` : idem que pour `channel` ;
- `comments` : URL d'une page de commentaires ;
- `author` : mél de l'auteur de l'entrée ;
- `pubDate` : idem que pour `channel` ;

Les sous-éléments de `item` 2/2

- `guid` : un *identifiant global unique*, c'est-à-dire une chaîne quelconque qui identifie de manière unique le lien (→ espaces de nommage). Attention : il possède un attribut `isPermaLink` dont la valeur par défaut est `true`. Dans ce cas la valeur de `guid` doit être une URL, celle du texte original.

Spécification de RSS v2 : `http://www.stervinou.com/projets/rss/`

La DTD de RSS v2 : `http://www.silmaril.ie/software/rss2.dtd`

Exemple de flux RSS v2

```
<?xml version="1.0"?>
<rss version="2.0">
<channel>
<title>Exemple de flux</title>
<link>http://www.example.org/</link>
<description>Juste du blabla</description>

<item>
<title>Une entrée typique</title>
<link>http://www.example.org/something.html</link>
<guid>http://www.example.org/something.html</guid>
<pubDate>Tue, 08 Apr 2003 10:28:59 GMT</pubDate>
<description>Voici le texte descriptif.</description>
</item>

</channel>
</rss>
```

Ce qui donne...


Exemple de flux
http://www.example.org/something.h
RSS
Google
Courrier
Pianomajeur
PM membres
PM Livres
PM admin
moodle
teoria
Exemple de flux
1 au total
Une entrée typique
8 avr. 2003 à 12h28
Voici le texte descriptif. Lire la suite…
Rechercher :
Longueur de l'article :
Trier par :
Date
Titre
Source
Nouveau

Scalable Vector Graphics

v. 1.1, du 30 avril 2002

Introduction

L'acronyme SVG signifie *Scalable Vector Graphics*. Cette norme décrit les *images vectorielles* éventuellement dynamiques ou animées.

La DTD de SVG propose 81 éléments et une foule d'attributs sous l'espace de noms

`http://www.w3.org/2000/svg`

pour décrire les images. Pour mieux spécifier les propriétés graphiques des objets on peut se servir également de feuilles de style CSS2 ou XSL. SVG est compatible XLink, XPointer, SMIL. Il dispose d'un DOM qui lui est propre.

Types de données

- *entier* : un nombre entre $-2\,147\,483\,648$ et $2\,147\,483\,647$;
- *nombre* : un nombre rationnel entre $-3,4 \times 10^{38}$ et $3,4 \times 10^{38}$;
- *longueur* : un *nombre* éventuellement suivi d'une unité de longueur parmi `em`, `ex`, `px`, `pt`, `pc`, `cm`, `mm` ;
- *angle* : un *nombre* éventuellement suivi de `deg` (par défaut), `grad` ou `rad` ;
- *couleur* : une spécification de couleur CSS2 (des noms de couleur prédéfinis, `#abcdef`, `#fb0` = `#ffbb00`, `rgb(255,255,255)` = `rgb(100%,100%,100%)`) ;
- *fréquence* : un nombre positif suivi de `Hz` ou `kHz` ;
- *temps* : un nombre positif suivi de `ms` ou `s`.

Éléments de structuration : svg

L'élément global est `svg`. Les `svg` peuvent s'imbriquer. Parmi ses attributs on trouve :

- la version de SVG (`version`) ;
- en cas d'imbrication, les coordonnées relatives de l'origine de l'image (`x` et `y`) ;
- les dimensions absolues de l'image (`width` et `height`) ;
- on peut attacher à un élément `g` des attributs de présentation, de style, d'événement, de transformation, etc.

Exemple :
```
<svg:svg version="1.0" width="5cm" height="5cm"
 xmlns:svg="http://www.w3.org/2000/svg">
```

Éléments de structuration : g

- l'élément g permet de grouper des composantes graphiques. Les g peuvent s'imbriquer ;
- on peut attacher à un élément g des attributs de présentation, de style, d'événement, de transformation, etc. ;

Éléments de structuration : defs

- l'élément `defs` correspond à un `g` dont l'attribut `display` a la valeur `none` ;
- on placera dans `defs` des objets qui seront référencés. Exemple :

```
<defs>
 <linearGradient id="Gradient01">
  <stop offset="20%" stop-color="#39F" />
  <stop offset="90%" stop-color="#F3F" />
 </linearGradient>
</defs>
<rect x="1cm" y="1cm" width="6cm" height="1cm"
 fill="url(#Gradient01)"/>
```

Éléments de structuration : `desc`, `title`

- les éléments `desc` et `title` peuvent être placés dans n'importe quel élément SVG graphique ou conteneur ;
- ils permettent d'inclure des descriptions lisibles par un humain des différents ingrédients de l'image. Ceux-ci peuvent être utilisés en cas d'impossibilité d'afficher telle ou telle ressource ;
- ils peuvent contenir des éléments dans d'autres espaces de nom. Exemple :

```
<svg width="4in" height="3in" version="1.1"
 xmlns="http://www.w3.org/2000/svg">
 <desc xmlns:m="http://omega.enstb.org/mondoc">
  <m:title>Voici un fichier SVG</m:title>
  <m:para>avec des balises de l'espace
   de noms <m:emph>mondoc</m:emph>.</m:para>
 </desc>
</svg>
```

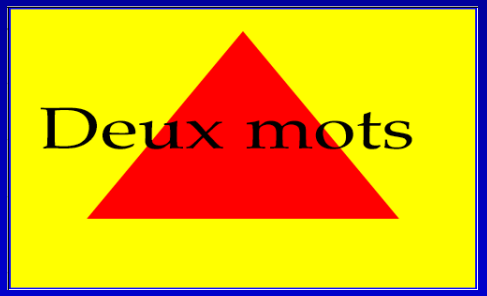
Éléments de structuration : `symbol`, `use`

- l'élément `symbol` permet de créer un élément «abstrait» qui ne peut être utilisé seul, mais uniquement instantié par `use`. Il peut avoir des attributs `viewbox` et `preserveAspectRatio` :
  - `viewbox` $(x_{\min}\ y_{\min}\ w\ h)$ établit une «fenêtre» de visualisation et un nouveau système de coordonnées, qui peut être utilisé dans la branche dont le sommet contient l'attribut ;
  - si `preserveAspectRatio` est `none`, alors la mise à l'échelle horizontale peut être différente de la verticale, pour remplir le *viewbox*.
- l'élément `use` peut «appeler» tout élément `svg`, `symbol`, `g`, tout élément graphique ou autre `use`.

Exemple de symbol, use

```
<svg width="500px" height="300px"
 xmlns="http://www.w3.org/2000/svg">
 <symbol viewBox="0 0 1500 1200" preserveAspectRatio="none" id="MonRect">
  <rect x="0" y="0" width="1500" height="1200"
   fill="yellow" stroke="blue" stroke-width="12"/>
  <path fill="red" d="M 750,100 L 250,900 L 1250,900 z"/>
  <text x="100" y="600" font-size="250" font-family="Palatino">
   Deux mots
  </text>
 </symbol>
 <use x="0" y="0" width="500px" height="300px" xlink:href="#MonRect"/>
</svg>
```

donnera :



Éléments de structuration : `symbol`, `use`

- le lien entre `use` et `symbol` se fait par des attributs `id` et `xlink:href`. Ce dernier provient de la norme XLink.
- ne pas oublier d'inclure la déclaration de l'espace de nom XLink à un moment ou un autre dans le document SVG :

```
xmlns:xlink="http://www.w3.org/1999/xlink"
```

- `use` peut également avoir un attribut `transform`, qui contient une liste de transformations parmi `matrix(`*`a b c d e f`*`)`, `translate(`*`tx`* `[`*`ty`*`])`, `scale(`*`sx`* `[`*`sy`*`])`, `rotate(`*`angle`* `[`*`cx cy`*`])`, `skewX(`*`angle`*`)`, `skewY(`*`angle`*`)`.

Éléments de structuration : `image`

- l'élément `image` va afficher sous forme d'image bitmap des fichiers SVG, PNG ou JPEG externes appelés par `xlink:href` ;
- il va également convertir en bitmap toute sa branche SVG.

Traitement conditionnel : `switch`

L'élément `switch` va choisir un élément parmi ses enfants directs suivant certains critères indiqués comme valeurs d'attribut :

- `requiredFeatures` (les fonctionnalités du lecteur SVG, prises dans une liste qui fait partie de la norme : `http://www.w3.org/TR/SVG11/feature#SVGDOM-animation`, etc.) ;
- `requiredExtensions` (les extensions au-délà de la norme requises par le lecteur SVG) ;
- `selectedLanguage` (la (ou les) langue(s) choisie(s) par l'utilisateur dans les préférences du lecteur).

Exemple : `<switch>`

```
<text systemLanguage="fr" x="100" y="100">Bonjour</text>
<text systemLanguage="en" x="100" y="100">Good morning</text>
</switch>
```

Feuilles de style

Il existe un certain nombre de *propriétés de style* qui peuvent être utilisées aussi bien sous forme d'attribut qu'à la CSS, dans le cadre d'un élément `style`, avec éventuellement des attributs `class`. Exemples :

```
<svg width="10cm" height="5cm" viewBox="0 0 1000 500"
 xmlns="http://www.w3.org/2000/svg" version="1.1">
 <rect x="200" y="100" width="600" height="300"
  fill="red" stroke="blue" stroke-width="3"/>
</svg>
```

Feuilles de style

On peut placer la feuille de style dans un fichier CSS :

```
rect {fill: red;stroke: blue;stroke-width: 3}
```

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet href="mystyle.css" type="text/css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
 "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="10cm" height="5cm" viewBox="0 0 1000 500"
 xmlns="http://www.w3.org/2000/svg" version="1.1">
 <rect x="200" y="100" width="600" height="300"/>
</svg>
```

Feuilles de style

Ou alors intégrer la feuille de style dans le document XML :

```
<svg width="10cm" height="5cm" viewBox="0 0 1000 500"
 xmlns="http://www.w3.org/2000/svg" version="1.1">
 <defs>
  <style type="text/css"><![CDATA[
  rect.rougebleu {fill: red;stroke: blue;stroke-width: 3}
  ]]></style>
 </defs>
 <rect class="rougebleu" x="200" y="100" width="600" height="300"/>
</svg>
```

Feuilles de style

On peut aussi écrire directement :

```
<svg width="10cm" height="5cm" viewBox="0 0 1000 500"
 xmlns="http://www.w3.org/2000/svg" version="1.1">
 <rect style="fill: red;stroke: blue;stroke-width: 3"
  x="200" y="100" width="600" height="300"/>
</svg>
```

L'élément `style` peut avoir un attribut `media` pour spécialiser la feuille de style à un certain type de médium.

Chemins

Pour décrire un chemin on utilise une syntaxe très proche de celle du PostScript (SVG a été développé par Adobe) : des lettres qui indiquent les actions (déplacement droit, courbe de Bézier de degré 2 ou 3, arc elliptique, fermeture de chemin) et des nombres qui indiquent les coordonnées nécessaires.

Un chemin est un élément `path`. Sa description se trouve dans la valeur de l'attribut `d`. On peut aussi lui affecter une longueur (attribut `pathLength`).

Chemins : instructions `MLHVZ/mlhvz`

- `M` (absolu) `m` (relatif) : *moveto* suivi de deux nombres, les coordonnées horizontale et verticale. Cette instruction établit le point courant ;
- `L` (absolu) `l` (relatif) : *lineto* suivi de deux nombres, les coordonnées horizontale et verticale. Cette instruction trace une ligne à partir du point courant ;
- `H` (absolu) `h` (relatif) : un *lineto* horizontal ;
- `V` (absolu) `v` (relatif) : un *lineto* vertical ;
- `Z` (absolu) `z` (relatif) : *closepath*. Cette instruction ferme le chemin courant ;

Chemins : instructions C/c, S/s

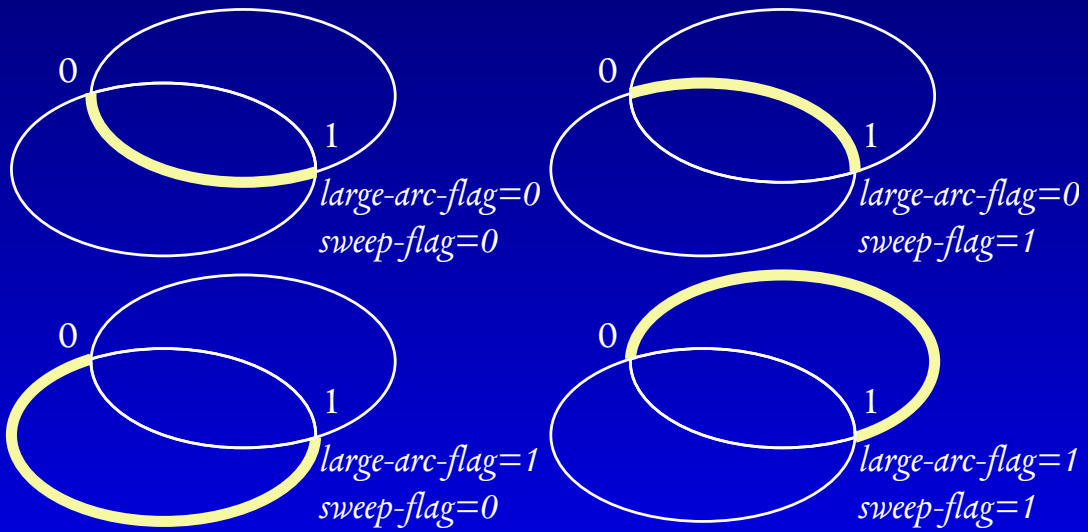
- C (absolu) c (relatif) : *curveto* suivi de six nombres x_1, y_1, x_2, y_2, x, y. Cette instruction trace une courbe de Bézier ayant le point courant comme point de départ, x_1, y_1 et x_2, y_2 comme points de contrôle, et x, y comme point d'arrivée ;
- S (absolu) s (relatif) : *smooth curveto* suivi de quatre nombres x_2, y_2, x, y. Cette instruction trace une courbe de Bézier ayant le point courant comme point de départ, le premier point de contrôle symétrique avec le deuxième de la courbe précédente, x_2, y_2 comme deuxième point de contrôle, et x, y comme point d'arrivée ;

Chemins : instructions `Q`/`q`, `T`/`t`

- `Q` (absolu) `q` (relatif) : *quadratic curveto* suivi de quatre nombres x_1, y_1, x, y. Cette instruction trace une courbe quadratique de Bézier ayant x_1, y_1 comme point de contrôle, et x, y comme point d'arrivée ;
- `T` (absolu) `t` (relatif) : *smooth quadratic curveto* suivi de deux nombres x, y. Cette instruction trace une courbe quadratique de Bézier ayant comme point de contrôle le symétrique de celui de la courbe précédente, et x, y comme point d'arrivée ;

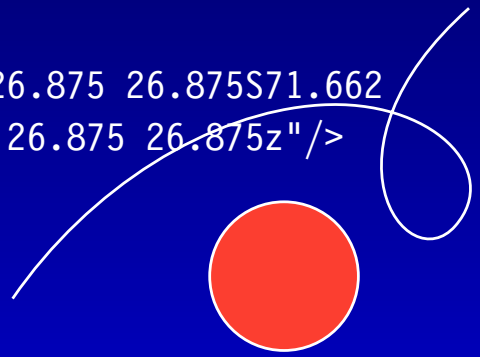
Chemins : instructions A/a

- A (absolu) a (relatif) : *elliptic curveto* suivi de sept nombres $r_x, r_y, \theta_x, l, s, x, y$. Cette instruction trace un arc d'ellipse (de rayons r_x, r_y et d'angle θ_x) passant par le point courant et par le point x, y. Les nombres $l, s \in \{0, 1\}$ (*large-arc-flag* et *sweep-flag*) indiquent s'il s'agit de l'arc le plus long ($l = 1$), de l'arc à droite ($s = 1$) ou à gauche ($s = 0$) de la droite reliant les deux points ;



Exemple de chemin

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000303 Stylable//EN"
"http://www.w3.org/TR/2000/03/WD-SVG-20000303/DTD/svg-20000303-stylable.dtd" [
<!ENTITY st0 "fill:#FA3F1E;"><!ENTITY st1 "fill:none;">
<!ENTITY st2
"fill-rule:nonzero;clip-rule:nonzero;stroke:#000000;stroke-miterlimit:4;">]>
<svg width="166.43pt" height="124.622pt" viewBox="0 0 166.43 124.622"
xml:space="preserve">
<g id="Calque_x0020_1" style="&st2;">
<path style="&st1;" d="M0.412 105.372c27.5-40 71.25-67.5 111.25-70s66.25 23.75 48.75
43.75s-52.5-27.5 5-78.75"/>
<path style="&st0;" d="M125.412 97.247c0 14.843-12.032 26.875-26.875 26.875S71.662
112.09 71.662 97.247s12.032-26.875 26.875-26.875s26.875 12.032 26.875 26.875z"/>
</g>
</svg>
```



Les formes de base : `rect`

L'élément `rect` produit un rectangle, avec les propriétés/attributs suivants :

- `x`, `y` : les coordonnées du point inférieur gauche ;
- `w`, `h` : la largeur et hauteur ;
- `rx`, `ry` : les arrondissements horizontal et vertical des sommets [sous réserve de faisabilité] ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Les formes de base : `circle`

L'élément `circle` produit un cercle, avec les propriétés/attributs suivants :

- `cx`, `cy` : les coordonnées du centre ;
- `r` : le rayon ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Les formes de base : `ellipse`

L'élément `ellipse` produit une ellipse, avec les propriétés/attributs suivants :

- `cx`, `cy` : les coordonnées du centre ;
- `rx`, `ry` : les rayons horizontal et vertical ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Les formes de base : `line`

L'élément `line` produit un segment de droite, avec les propriétés/attributs suivants :

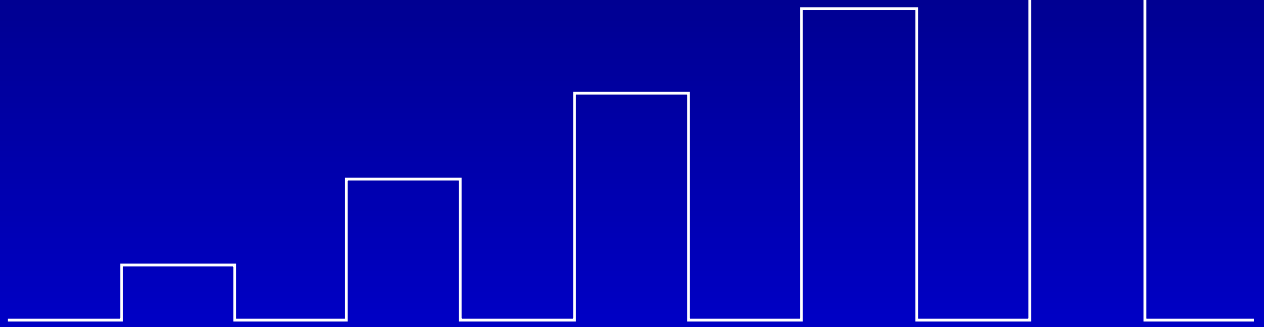
- `x1`, `y1` : les coordonnées du point de départ ;
- `x2`, `y2` : les coordonnées du point d'arrivée ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Les formes de base : `polyline`

L'élément `polyline` produit une ligne brisée, avec les propriétés/attributs suivants :

- `points` : un nombre quelconque de paires de coordonnées ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Exemple :
```
<polyline fill="none" stroke="blue" stroke-width="10"
points="50,375 150,375 150,325 250,325 250,375 350,375 350,250
450,250 450,375 550,375 550,175 650,175 650,375 750,375 750,100
850,100 850,375 950,375 950,25 1050,25 1050,375 1150,375" />
```

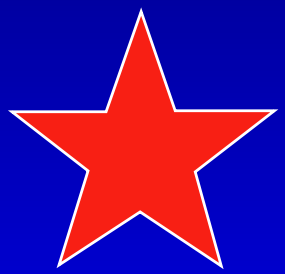


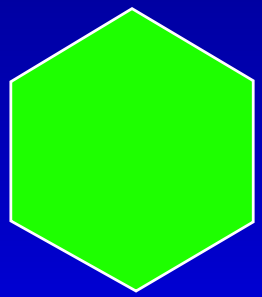
Les formes de base : `polygon`

L'élément `polygon` produit un polygone fermé, avec les propriétés/attributs suivants :

- `points` : un nombre quelconque de paires de coordonnées ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Exemple :
```
<polygon fill="red" stroke="blue" stroke-width="10"
points="350,75 379,161 469,161 397,215 423,301 350,250 277,301
303,215 231,161 321,161" />
<polygon fill="lime" stroke="blue" stroke-width="10" points="850,75
958,137.5 958,262.5 850,325 742,262.6 742,137.5" />
```





Les éléments de texte : `text`

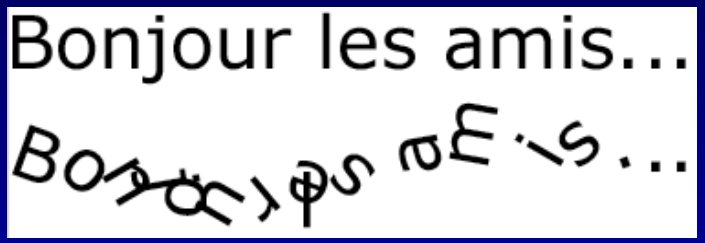
L'élément `text` produit une ligne de texte, avec les propriétés/attributs suivants :

- `x`, `y` : les coordonnées de l'origine du premier glyphe (ou des n premiers glyphes) ;
- `dx`, `dy` : décalage des coordonnées de l'origine du premier glyphe (ou des n premiers glyphes) ;
- `rotate` : l'angle du premier glyphe (ou des n premiers glyphes)
- `textLength` : calibrage de la largeur du texte (permet la justification précise !) ;
- `lengthAdjust` : prend les valeurs `spacing` ou `spacingAndGlyphs` et indique la manière de calibrer ;
- Tous les attributs de style, coloriage, opacité, etc. s'appliquent.

Les éléments de texte : text

Exemple :

```
<text x="0" y="70" font-family="Verdana" font-size="70">
Bonjour les amis... </text>
<text x="0" y="170" rotate="20 40 60 80 100 120 140 160 180 200 220
240 260 280 300 320 340 360" font-family="Verdana" font-size="70">
Bonjour les amis... </text>
```



Les éléments de texte : t

À l'intérieur d'un élément `text` on peut changer les propriétés de fonte ou la position de texte courante à l'aide d'éléments `tspan`. Ils prennent les mêmes attributs que `text`. Exemple :

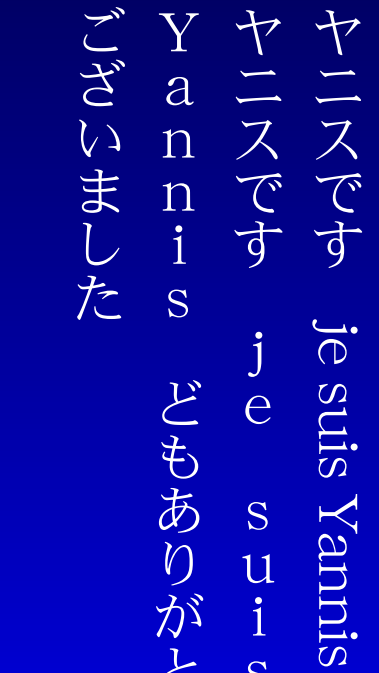
```
<text x="0" y="150" font-family="Times" font-size="120"> T<tspan
dy="20" dx="-10">E</tspan><tspan dy="-20" dx="-7">X</tspan> is the
<tspan fill="none" stroke="red">best</tspan>... </text>
```

Attributs de texte : writing-mode, glyph-orientation-*

writing-mode peut prendre les valeurs lr, rl et tb.

glyph-orientation-vertical peut prendre les valeurs auto (tout ce qui est «pleine largeur» est composé droit, le reste est composé à 90 degrés), 0, 90, 180, 270. Exemples :

ヤニスです je suis Yannis.
どもありがとございました



Autres attributs de texte

- `font-family` (une liste de noms de fonte), `font-style` (*`normal`*, *`italic`*, *`oblique`*), `font-variant` (*`small-caps`*), `font-weight` (*`normal`*, `bold`, etc.), `font-stretch` (*`normal`*, `wider`, etc.), `font-size`, `font-size-adjust` (la valeur d'aspect de la première fonte choisie de manière à garder la même hauteur de bas-de-casse);

Autres attributs de texte

- `dominant-baseline` (*alphabetic*, *mathematical*, *ideographic*, etc. = la ligne de base de base), `alignment-baseline` (alignement des enfants avec les parents), `baseline-shift` (*super*, *sub*, *30%*, *50pt*, etc. = décalage temporaire de toute la table) ;
- `kerning` (*auto*, *1pt*, etc.), `letter-spacing` (*normal*, *2pt*, etc.), `word-spacing` (*normal*, *5pt*, etc.) ;
- `text-decoration` (*none*, *underline*, *overline*, *line-through*, *blink*).

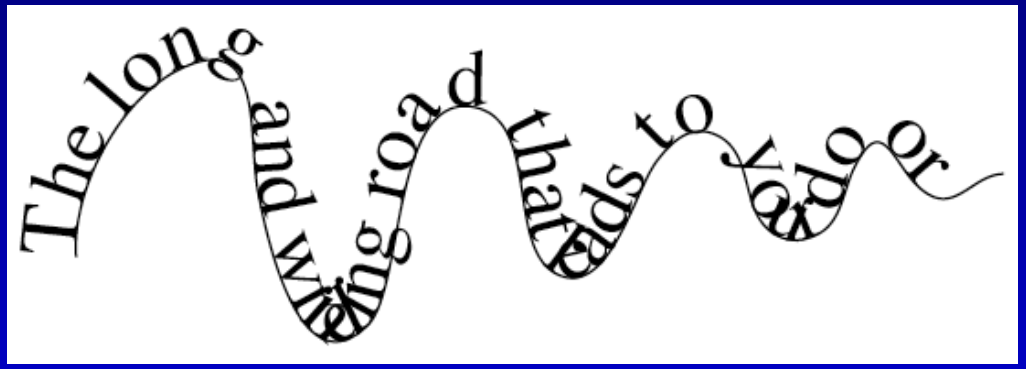
Texte sur un chemin : `textPath`

Cet élément permet de composer du texte sur un chemin. Attributs :

- `xlink:href` : la référence du chemin [à noter que les coordonnées du chemin sont interpretées selon le système de coordonnées du texte] ;
- `startOffset` (*longueur* ou *percentage*) : l'origine du texte sur le chemin ;
- `method` (*`align`* ou *`stretch`*) : savoir si on va torturer les glyphes ;
- `spacing` (*`auto`*, *`exact`*) : qui a la responsabilité du positionnement des glyphes. *`exact`* signifie : on prend une distance sur le chemin égale à la chasse du glyphe et on place le glyphe tangent au milieu de cette distance.

Texte sur un chemin : exemple

```
<defs><path id="tortueux" d="M36 130.719c-2.252-32.237 20.988-76.2
49.96-90.367c41.189-20.142 32.652 32.089 37.762 58.904c3.361 17.638 19.286 97.69
53.152 64.108c17.422-17.276 11.152-113.068 51.41-104.703c31.937 6.637 14.849 66.834
38.995 79.949 c25.358 13.774 38.103-39.809 48.66-53.799c7.203-9.545 16.475-18.659
29.053-12.743c15.57 7.323 12.777 27.813 20.636 40.303c6.265 9.957 17.53 13.224 28.624
7.151c10.92-5.978 11.629-20.152 16.719-30.892c13.604-28.699 18.934-3.246 31.415 8.33
c17.04 15.804 25.146-6.076 41.114-6.241"/></defs>
<use xlink:href="#tortueux" fill="none" stroke="black"/>
<text font-family="Times" font-size="40"><textPath xlink:href="#tortueux">The long and
winding road that leads to your door</textPath></text>
```



Fontes *WebFont*

CSS2 prévoit l'utilisation de fontes de différents types. On utilise la propriété `src` qui peut prendre 3 types de valeur : `local()`, `url()`, `format()`. Parmi les valeurs possibles pour `format` on a : `type-1`, `truetype`, `opentype`, `embedded-opentype` et `svg`. Exemple :

```
<svg width="400px" height="300px" version="1.1" xmlns =
'http://www.w3.org/2000/svg'> <defs> <style type="text/css">
<![CDATA[ @font-face  font-family: 'Belle fonte'; font-weight:
normal; font-style: italic; src:
url("http://omega.enstb.org/belle.otf") format(opentype)  ]]>
</style> </defs>
<text x="100" y="100" style="font-family: 'Belle fonte';
font-weight:normal; font-style: italic">Du texte composé en belle
fonte</text> </svg>
```

Fontes SVG

- La méthode *WebFont* présente encore certains désavantages. SVG propose un format de fonte «interne» qui donne des résultats médiocres mais est compatible avec tous les lecteurs SVG.
- Une fonte SVG consiste en un élément `font` qui va contenir : une description générale et une collection de glyphes (y compris le «glyphe manquant»).

Fontes SVG : description générale

- La description générale se fait dans un élément `font-face`, qui prend des attributs `font-family`, `font-family`, `font-style`, `font-variant`, `font-weight`, `font-stretch`, `font-size`, `unicode-range`, `units-per-em`, `panose-1`, `stemv`, `stemh`, `slope`, `cap-height`, `x-height`, `accent-height`, `ascent`, `descent`, `widths`, `bbox`, `ideographic`, `alphabetic`, `mathematical`, `hanging`, `v-ideographic`, `v-alphabetic`, `v-mathematical`, `v-hanging`, `underline-position`, `underline-thickness`, `strikethrough-position`, `strikethrough-thickness`, `overline-position`, `overline-thickness`.

Fontes SVG : les glyphes 1/2

Chaque description de contour de glyphes est contenue dans un élément `glyph`. Cet élément peut être vide (le contour étant spécifié par un attribut) ou peut contenir tout un arbre SVG. Voici quelques attributs de `glyph` :

- `unicode` : le (ou les) caractère(s) Unicode auquel(s) correspond le glyphe. Ainsi `unicode="ffi"` servira à la ligature 'ffi'. On peut utiliser des entités d'appel de caractère `&#x039a;` ;
- `glyph-name` : le (ou les) nom(s) du glyphe. Utile lorsqu'on a plusieurs glyphes pour le même caractère ;
- `arabic-form` : (*`initial`*, *`medial`*, *`terminal`*, *`isolated`*) ;

Fontes SVG : les glyphes 2/2

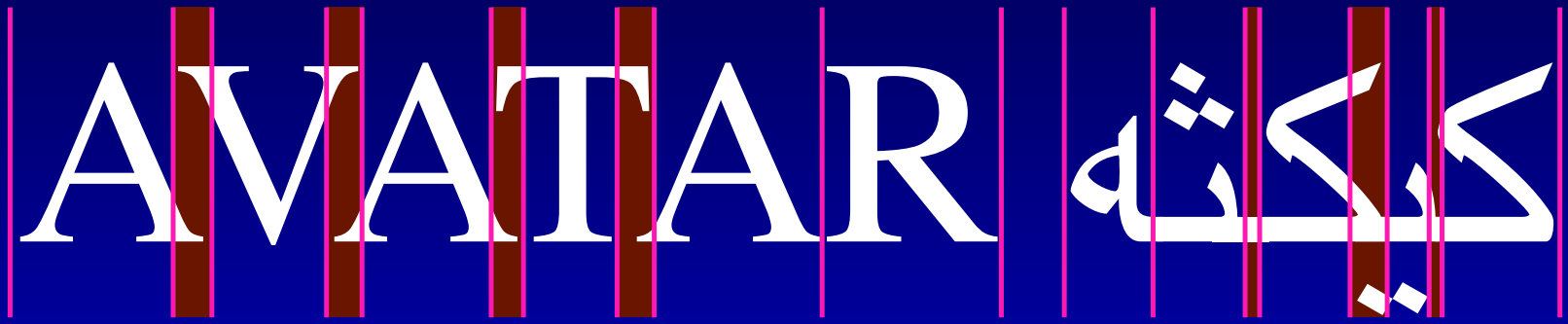
- `d` : le contour du glyphe, en utilisant la même syntaxe que pour les chemins (sauf que l'axe vertical pointe vers le haut et que l'attribut `units-per-em` (par défaut : 1000) de l'élément `font` donne la correspondance entre coordonnées internes et externes ;
- `horiz-adv-x` : la chasse [valeur par défaut : même attribut, appliqué à `font`] ;
- `orientation` : (*h* ou *v*) orientation forcée ;
- `lang` : une liste de codes de langue, séparés par des virgules. Le glyphe sera utilisé si l'élément de texte a un attribut `xml:lang` dans cette liste.

Fontes *SVG* : `missing-glyph`

L'élément (obligatoire) `missing-glyph` décrit un glyphe sans nom, ni position Unicode. Celui-ci est utilisé lorsqu'on fait appel à un glyphe inexistant dans la fonte. [Rêve d'un réseau de serveurs Web de glyphes...]

Fontes SVG : crénage 1/2

Le *crénage* est l'opération qui consiste à rapprocher ou éloigner des glyphes pour produire un meilleur résultat optique. Le crénage dépend de la fonte, mais aussi du contexte d'utilisation (texte courant, texte en petits caractères, titrage, etc.)



Ici : paires de crénage AV, VA, AT, TA, tha-kaf, kaf-ya, ya-kaf.

Fontes SVG : crénage 2/2

Le crénage (horizontal ou vertical) se fait à l'aide des éléments `hkern` et `vkern`. Ceux-ci sont vides et disposent des attributs suivants :

- `u1` : liste de caractères Unicode séparés par des virgules (pour la virgule, utiliser `g1`. Il s'agit du glyphe visuellement à gauche (ou en haut);
- `g1` : liste de noms de glyphe séparés par des virgules. Il s'agit du glyphe visuellement à gauche (ou en haut);
- `u2` et `g2` : idem mais ici il s'agit du glyphe visuellement à droite (ou en bas);
- `k` : valeur du crénage, dans les coordonnées de la fonte.

Gradients linéaires

L'élément `linearGradient` définit une zone donnée par ses coordonnées (attributs `x1`, `y1`, `x2`, `y2`) dans laquelle on a un gradient de couleurs dont le vecteur est défini par des éléments `stop` qui déterminent les couleurs (attribut `stop-color`) sur le vecteur (attribut `offset`). On peut remplir toute forme SVG par une référence à cette zone. Exemple :

```
<defs>
<linearGradient id="g1" gradientUnits="userSpaceOnUse" x1="50" y1="50"
x2="250" y2="250">
<stop offset="0" stop-color="red"/><stop offset="1" stop-color="blue"/>
</linearGradient> </defs>
<rect fill="url(#g1)" x="10" y="10" width="390" height="100"/>
```

Gradients linéaires adaptatifs

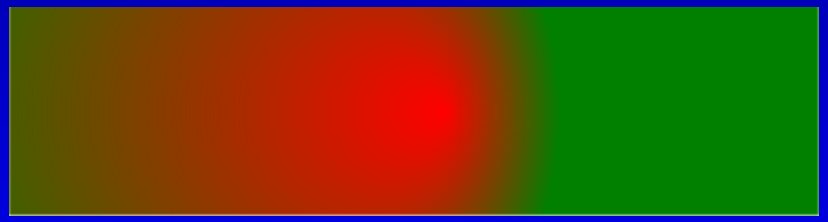
En donnant la valeur `objectBoundingBox` à l'attribut `gradientUnits`, le gradient linéaire s'adapte à la *bounding box* de la forme qui l'instantie. Exemple :

```
<defs>
<linearGradient id="g1" gradientUnits="objectBoundingBox">
<stop offset="0" stop-color="red"/><stop offset="1" stop-color="blue"/>
</linearGradient> </defs>
<rect fill="url(#g1)" x="10" y="10" width="390" height="100"/>
```

Gradients radiaux

L'élément `radialGradient` définit un disque donné par les coordonnées de son centre et son rayon (attributs `cx`, `cy`, `r`) dans lequel on a un g. de c. avec un certain centre focal (attributs `fx`, `fy`) et dont le vecteur est défini par des éléments `stop` qui déterminent les couleurs (attribut `stop-color`) sur le vecteur (attribut `offset`). On peut remplir toute forme SVG par une référence à ce disque. Ex. :

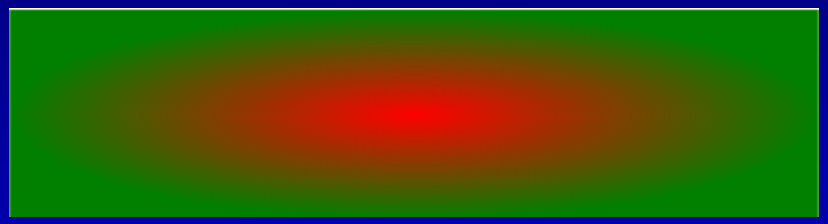
```
<defs>
<radialGradient id="g2" gradientUnits="userSpaceOnUse" cx="100" cy="170"
r="175" fx="220" fy="170">
<stop offset="0" stop-color="red"/><stop offset="1" stop-color="green"/>
</radialGradient> </defs>
<rect fill="url(#g2)" x="10" y="120" width="390" height="100"/>
```



Gradients radiaux adaptatifs

En donnant la valeur `objectBoundingBox` à l'attribut `gradientUnits`, le gradient radial s'adapte à la *bounding box* de la forme qui l'instantie. En particulier, il peut s'aplatir pour y tenir. Exemple :

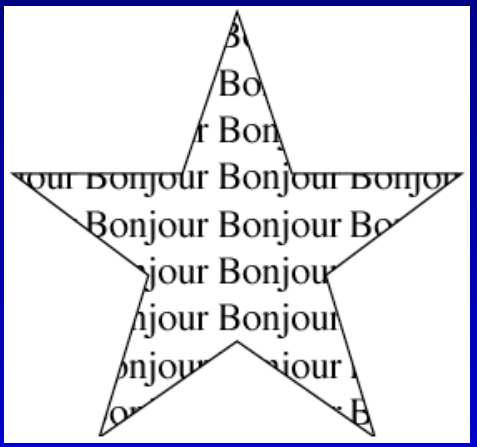
```
<defs>
<radialGradient id="g2" gradientUnits="objectBoundingBox">
<stop offset="0" stop-color="red"/><stop offset="1" stop-color="green"/>
</radialGradient> </defs>
<rect fill="url(#g2)" x="10" y="120" width="390" height="100"/>
```



Motifs

Un motif est une branche SVG qui se répète indéfiniment pour former un pavage. Au sommet de cette branche, l'élément `pattern` qui utilise des attributs `x`, `y`, `width`, `height` pour spécifier la taille du motif. Exemple :

```
<defs>
<pattern id="m1" patternUnits="userSpaceOnUse" x="0" y="0" width="70"
height="25">
<text x="0" y="20" font-family="Times" font-size="20"
fill="black">Bonjour</text>
</pattern>
</defs>
<polygon stroke="black" fill="url(#m1)"
points="150,25 179,111 269,111 197,165
223,251 150,200 77,251 103,165
31,111 121,111"/>
```



Superposition, opacité

Parfois une partie de l'image doit être cachée. On obtient cet effet à l'aide de l'élément `clipPath`.

L'attribut `opacity` appliqué à un groupe `g` permet de déterminer l'opacité du groupe : sa valeur est un nombre entre `0` (transparent) et `1` (solide). Exemple :

```
<text x="40" y="80" font-family="Times" font-size="70"
fill="red">N&#201;CESSIT&#201;</text>
<g opacity="0.5"><text x="15" y="95" font-family="Times" font-style="italic"
font-size="85" fill="black">morale</text></g>
```

Filtres

Un grand nombre de filtres permettent d'obtenir des effets de profondeur, d'éclairage spécial, de flou, de distortion, d'engraissement, de grain, etc.

Ils dont définis dans les définitions globales (`defs`) et utilisés à l'aide de l'attribut `filter` (dont la valeur est une référence à la définition).

Interactivité

Divers événements peuvent être interceptés : `focusin/out`, `activate`, `click`, `mousedown/up/over/move/out`.
D'autre part, la modification d'une image SVG dans un environnement de graphisme interactif revient à une transformation DOM. On a des événements pour plusieurs de ces transformations (ajout ou suppresion d'une branche ou d'un nœud, etc.).
On a également des événement pour les changements globaux : mise à l'échelle, zoom, etc. et pour les débuts, fins et répétitions d'animations.

Liens

SVG utilise XLink pour définir des liens simples (au sens de XLink). Cela se passe dans un élément `a` qui va donc prendre des attributs dans l'espace de noms de XLink. Exemple :

```
<a xmlns:xlink="http://www.w3.org/1999/xlink"
 xlink:href="http://moi.com"
 xlink:title="Un lien vers moi">
```

Les attributs XLink possibles sont : `xlink:role`, `xlink:arcrole`, `xlink:title`, `xlink:show`, `xlink:actuate` et `xlink:href` (obligatoire).

Scriptage et animation

On peut écrire des scripts dans différents langages (du moment qu'ils sont compatibles avec le lecteur SVG). Ces scripts peuvent accéder aux événements en passant par une API.

Les éléments d'animation sont inspirés du langage SMIL (*Synchronized Multimedia Integration Language*) : `animate` (changer la valeur d'attributs numériques au cours du temps), `set` (changement d'attributs non-numériques), `animateMotion` (déplacement d'un élément en suivant un chemin), `animateColor` (changement d'une valeur de couleur), `animateTransform` (changement d'attributs de transformation), `keypoints` (pour contrôler la vitesse de l'animation).

Métadonnées

Des métadonnées RDF peuvent être incluses dans un élément `metadata`. Exemple :

```
<metadata>
<rdf:RDF xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:dc =
"http://purl.org/dc/elements/1.1/" >
<rdf:Description about="http://example.org/myfoo" dc:title="MyFoo Financial
Report" dc:description="$three $bar $thousands $dollars $from 1998 $through
2000" dc:publisher="Example Organization" dc:date="2000-04-11"
dc:format="image/svg+xml" dc:language="en" >
<dc:creator>
<rdf:Bag> <rdf:li>Irving Bird</rdf:li> <rdf:li>Mary Lambert</rdf:li>
</rdf:Bag> </dc:creator>
</rdf:Description>
</rdf:RDF>
</metadata>
```

SVG

SAX, DOM, XSLT

SAX, DOM, XSLT

Trois technologies s'offrent à nous pour analyser, traiter et transformer les documents XML :

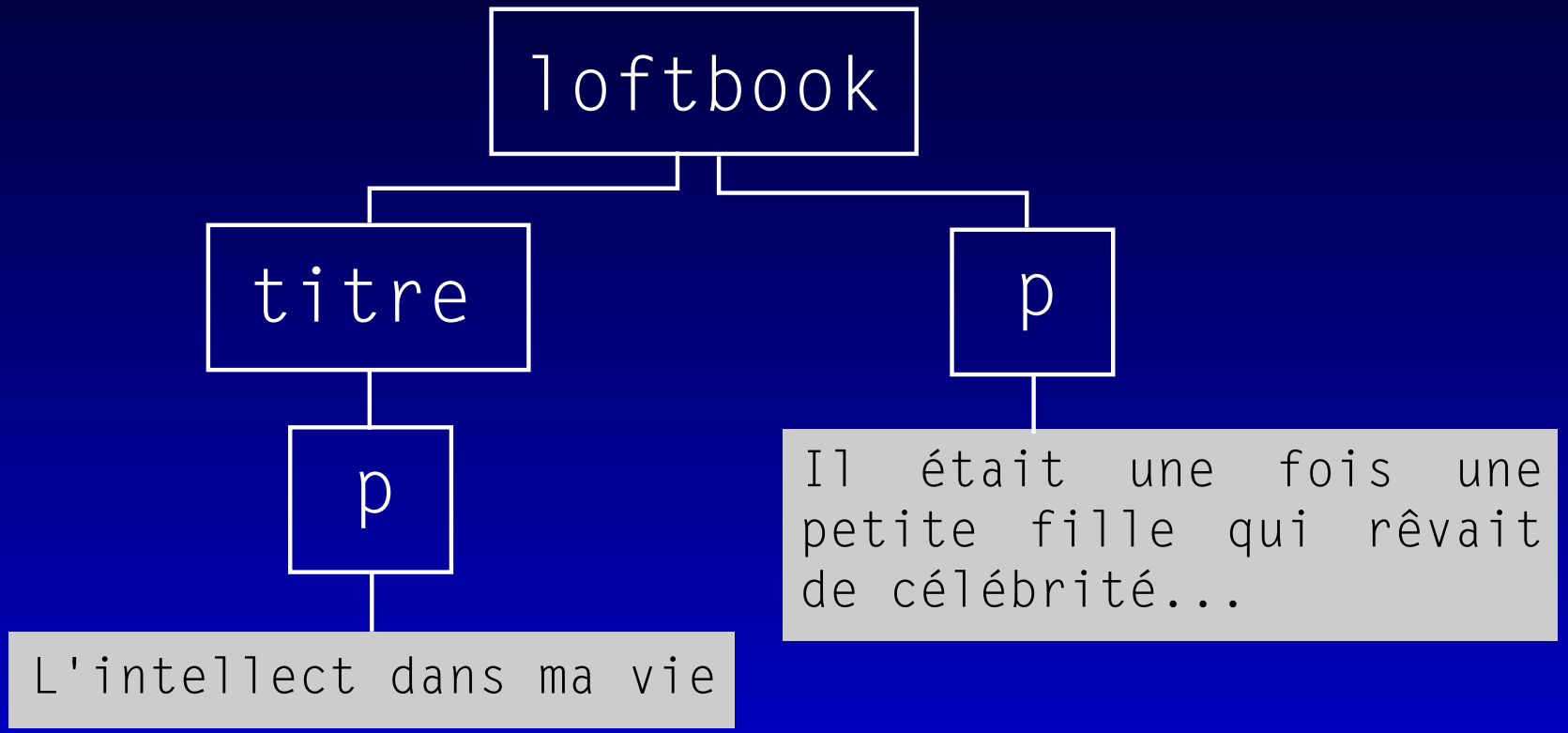
- *SAX*, une API événementielle qui lit linéairement (du début à la fin) un document XML et exécute des méthodes correspondant aux différents ingrédients du document ;
- *DOM*, une API basée sur une modélisation objet des documents XML avec des classes correspondant aux ingrédients et des méthodes correspondant à divers types de manipulations d'arbre ;
- *XSLT*, une norme W3C basée sur XML, décrivant une transformation par le biais de *règles* appliquées à des nœuds particuliers de l'arbre XML.

XPath

Version 1.0, du 16 novembre 1999

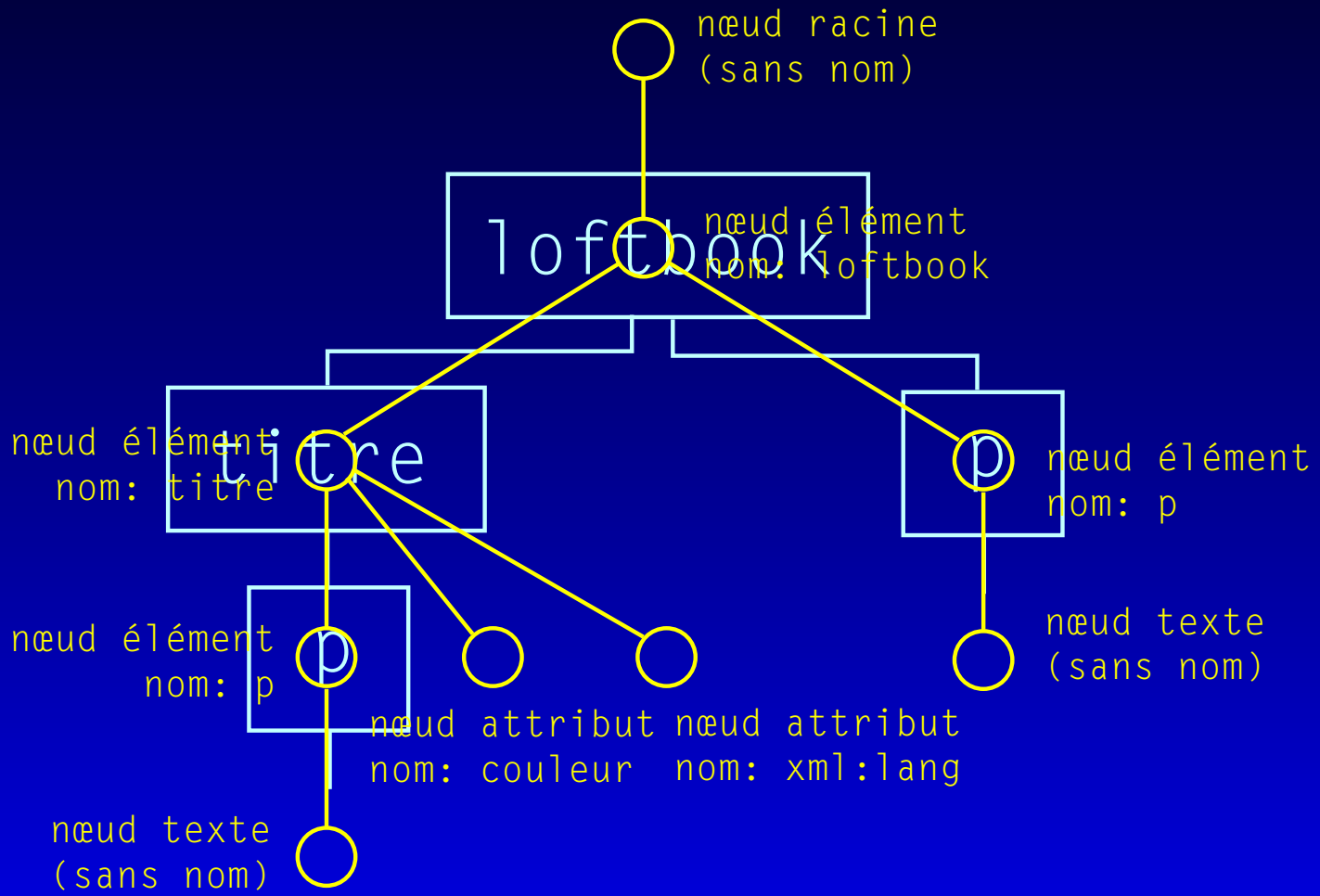
Arbre de nœuds

À la place de l'arbre d'éléments...



Arbre de nœuds

...on considère un *arbre de nœuds* :



Question : pourquoi a-t-on besoin du nœud racine ?

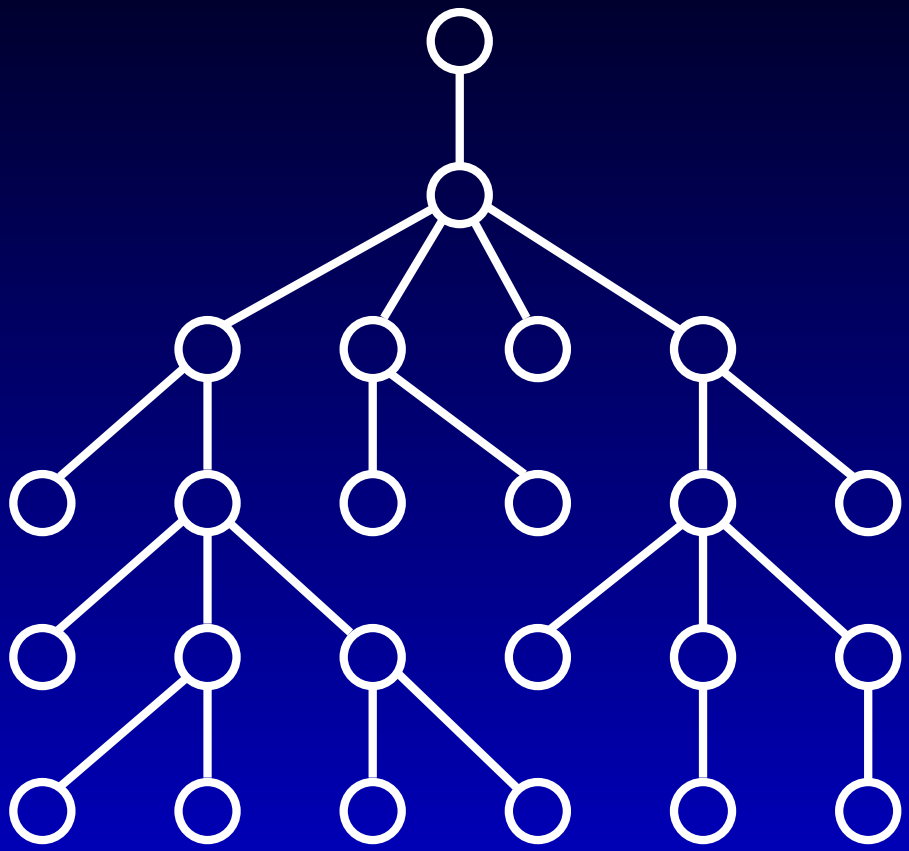
Types de nœuds

- *nœud racine* ou *nœud document* : valeur ;
- *nœud d'élément* : nom, valeur ;
- *nœud de texte* : valeur ;
- *nœud d'attribut* : nom, valeur ;
- *nœud d'espace de noms* : nom, valeur ;
- *nœud d'instruction de traitement* : nom, valeur ;
- *nœud de commentaire* : valeur.

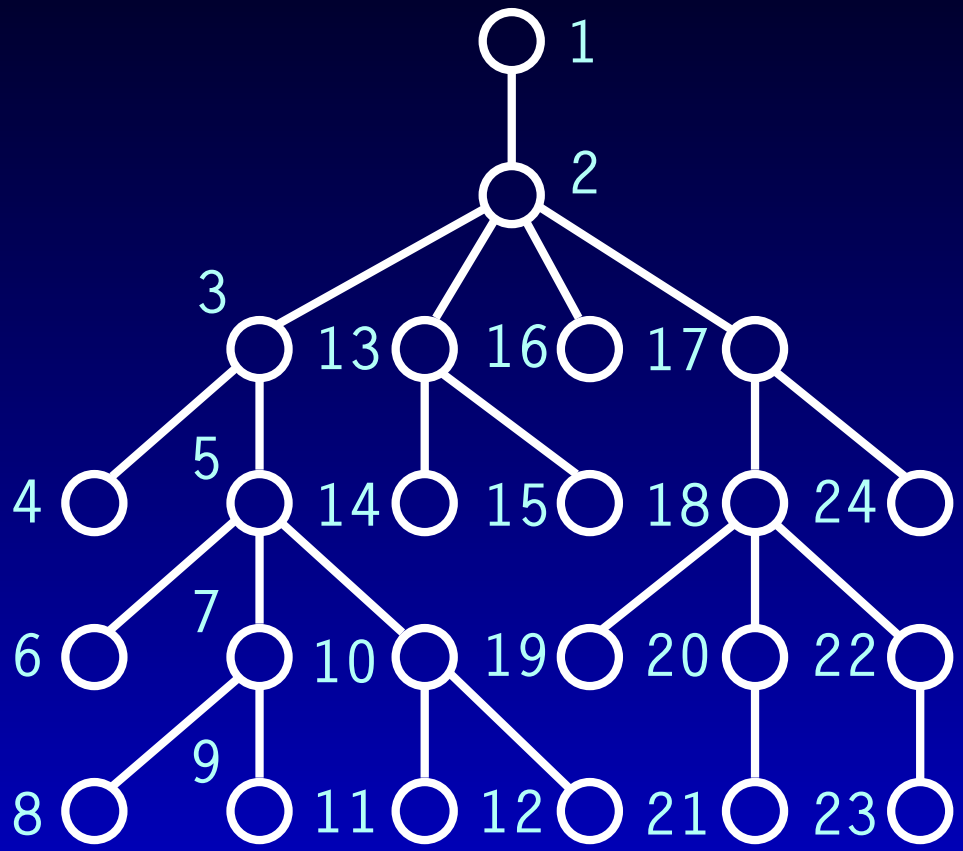
Valeurs des nœuds

- Un nœud de texte, de commentaire et d'instruction de traitement a comme valeur son contenu.
- Un nœud d'attribut a comme valeur, la valeur de l'attribut. (Un espace de noms est considéré aussi comme un attribut.)
- Un nœud d'élément, et en particulier le nœud racine, ont comme valeur la *concaténation des valeurs de leurs descendants qui sont des nœuds d'élément*.

Ordre des nœuds dans un arbre



Ordre des nœuds dans un arbre



On compte de haut en bas et de gauche à droite. On respecte l'ordre : élément, espace de noms, attributs.

Se déplacer dans un arbre de nœuds

On souhaite se déplacer à partir d'un nœud, le *nœud courant* et accéder, par exemple :

- à tous ses éléments descendants, de nom `para` ;
- à son dernier frère ;
- à l'attribut `id` de l'enfant `titre` de son grand-parent...

Pour ce faire on va se servir des *axes de déplacement XPath*.

Axes de déplacement XPath 1/2

- `child` : les enfants directs ;
- `descendant` : les descendants (pas d'attributs, ni d'espaces de noms) ;
- `descendant-or-self` : le nœud courant et ses descendants ;
- `parent` : le parent du nœud courant ;
- `ancestor` : les ancêtres ;
- `ancestor-or-self` : le nœud courant et ses ancêtres.
- `following-sibling` : tous les nœuds suivant le nœud courant et ayant le même parent ;
- `preceding-sibling` : tous les nœuds précédant le nœud courant et ayant le même parent ;

Axes de déplacement XPath 2/2

- `following` : tous les nœuds suivant le nœud courant et qui ne sont pas ses descendants ;
- `preceding` : tous les nœuds précédant le nœud courant et qui ne sont pas ses ancêtres ;
- `attribute` : tous les attributs du nœud courant ;
- `namespace` : tous les espaces de noms du nœud courant ;
- `self` : le nœud courant.

Note : `self` + `ancestor` + `descendant` + `preceding` + `following` = l'arbre tout entier.

Exemples de syntaxe :

`child::para` ou `./para` ou `para` (**important** !)
`child::*` ou `./*` ou `*`
`child::text()` ou `text()`

Exemples de syntaxe

`attribute::name` ou `@name`

`attribute::*` ou `@*`

`descendant::para` ou `//para`

`parent::node()` ou `..`

`ancestor::div`

`../@couleur`

`child::para/child::div` ou `para/div`

`/` (le nœud racine)

`fn:doc("http://moi.qqpart.com/toto.xml")`

Les tests de nœuds

On peut combiner la syntaxe précédente avec des tests. Exemples :

```
para[@couleur="rouge"]
para[@couleur="rouge"][5]
para[5][@couleur="rouge"]
chapter[titre="Introduction"]
chapter[titre]
```

Fonctions XPath 1/7

Les fonctions `fn:last()`, `fn:position()`, `fn:count()`, `fn:number()`, `fn:sum()`, `fn:round()` `fn:floor()` `fn:ceiling()`, `fn:abs()` retournent des nombres :

- `fn:last()` retourne l'ordinal du dernier frère ;
- `fn:position()` retourne l'ordinal du nœud courant parmi ses frères ;
- `fn:count(`*`nœuds`*`)` retourne le nombre de nœuds dans l'ensemble *nœuds*.

Fonctions XPath 2/7

- `fn:number(`*`objet`*`)` convertit *`objet`* en nombre ;
- `fn:sum(`*`nœuds`*`)` retourne la somme des valeurs des nœuds de *`nœuds`* converties en nombres ;
- `fn:round(`*`nombre`*`)`, `fn:floor(`*`nombre`*`)`, `fn:ceiling(`*`nombre`*`)` arrondissent les nombres ;
- `fn:abs(`*`nombre`*`)` retourne la valeur absolue.

Fonctions XPath 3/7

Les fonctions `fn:name()`, `fn:local-name()`, `fn:namespace-uri()`, `fn:string()`, `fn:concat()`, `fn:substring()`, `fn:substring-before()`, `fn:substring-after()` retournent des chaînes de caractères.

- `fn:name(`*nœud*`)` retourne le nom complet du nœud ;
- `fn:local-name(`*nœud*`)` retourne le nom local du nœud ;
- `fn:namespace-uri(`*nœud*`)` retourne l'espace de noms du nœud ;

Fonctions XPath 4/7

- `fn:string(`*`objet`*`)` convertit *objet* en chaîne de caractères ;
- `fn:concat(`*`chaîne`*`+)` concatène les chaînes de caractères ;
- `fn:substring-before(`*`chaîne`*`,`*`chaîne`*`)`, `fn:substring-after(`*`chaîne`*`,`*`chaîne`*`)`, retournent des sous-chaînes de caractères ;
- `fn:substring(`*`chaîne`*`,`*`nombre`*`,`*`nombre`*`?)`, retourne une sous-chaîne de caractères.

Fonctions XPath 5/7

Les fonctions `fn:starts-with()`, `fn:contains()`, `fn:boolean()`, `fn:not()`, `fn:true()`, `fn:false()`, `fn:lang()` retournent des booléennes.

- `fn:starts-with(`*chaîne*`,`*chaîne*`)` vérifie si la première chaîne commence par la seconde ;
- `fn:contains(`*chaîne*`,`*chaîne*`)` vérifie si la première chaîne contient la seconde ;

Fonctions XPath 6/7

- `fn:boolean(`*`objet`*`)` convertit *`objet`* en booléen : un nombre est vrai s'il est non-nul, un ensemble de nœuds ou une chaîne de caractères sont vrais s'il sont non-vides ;
- `fn:not(`*`booléen`*`)` inverse la valeur de *`booléen`* ;
- `fn:true()` (resp. `fn:false()`) retournent toujours vrai (resp. faux) ;
- `fn:lang(`*`langue`*`)` vérifie si la langue courante est *`langue`*.

Fonctions XPath 7/7

- La fonction `fn:id("toto")` retourne le nœud qui a comme identifiant unique *toto*.
- La fonction `fn:id(nœuds)` retourne les nœuds qui ont des identifiants uniques parmi les valeurs textuelles de tous les nœuds de *nœuds*.
- La fonction `fn:id(objet)` convertit d'abord *objet* en chaîne de caractères, ensuite la subdivise en unités lexicales séparées par des blancs, et finalement retourne tous les nœuds qui ont des identifiants uniques parmi celles-ci.

La syntaxe des expressions XPath (synth.)

Pour écrire des expressions XPath, on utilise :

- des *noms* XML : `livre`, etc. ;
- des nombres (avec ou sans partie décimale) : `21`, `0.5`, etc. ;
- des types de nœud : `fn:node()`, `fn:comment()`, `fn:text()`, `fn:processing-instruction()` ;
- des noms de fonction : `fn:last()`, `fn:position()`, etc. ;

La syntaxe des expressions XPath (synth.)

- des opérateurs : `and`, `or`, `mod`, `div`, `*`, `/`, `//`, `|`, `+`, `-`, `=`, `!=`, `<`, `<=`, `>`, `>=` ;
- des noms d'axe : `child`, `ancestor`, etc. ;
- des litéraux : `"Loana"`, `"123"`, `'t29'`, etc. ;
- des appels de variables : `$id`, `$taille`, etc.

XPath 2

Version 2.0, du 11 février 2005

Espaces de nommage utilisés

- (xs) les types de données provenant des schémas :
  http://www.w3.org/2001/XMLSchema ;
- (xdt) d'autres types de données :
  http://www.w3.org/2005/02/xpath-datatypes ;
- (fn) les fonctions XPath2 :
  http://www.w3.org/2005/02/xpath-functions ;
- (op) les opérateurs XPath2 :
  http://www.w3.org/2005/02/xpath-operators ;

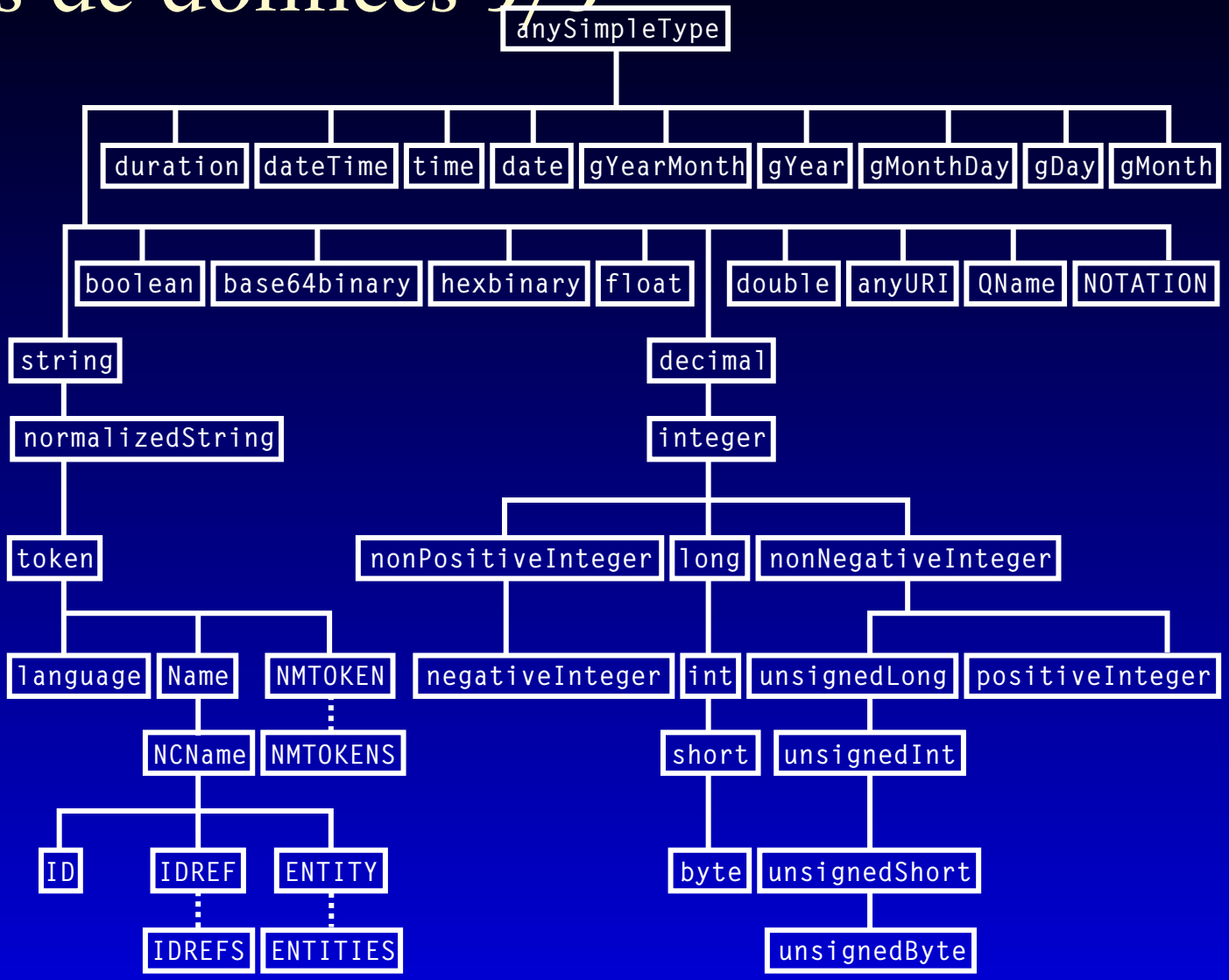
Types de données 1/3

- xdt:untypedAtomic, xs:string ;
- xs:boolean : true ou false ou 1 ou 0 ;
- xs:byte, xs:unsignedByte, xs:base64Binary, xs:hexBinary, xs:integer, xs:positiveInteger, xs:negativeInteger, xs:nonNegativeInteger, xs:nonPositiveInteger, xs:int, xs:unsignedInt, xs:long, xs:unsignedLong, xs:short, xs:unsignedShort, xs:decimal, xs:float, xs:double : des nombres ;
- xs:time, xs:dateTime, xs:date, xs:gMonth, xs:gYear, xs:gYearMonth, xs:gDay, xs:gMonthDay : des manières de préciser un instant temporel ;

Types de données 2/3

- `xs:duration` : une durée temporelle ;
- `xs:Name` : un nom XML ;
- `xs:anyURI` : une suite d'URLs séparées par des virgules ;
- `xs:language` : une spécification de langue (à la `xml:lang`) ;
- `xs:ID` : un identifiant unique (nom XML) ;
- `xs:IDREF` : une référence vers un identifiant unique ;
- `xs:ENTITY` : un nom d'entité déclarée ;
- `xs:NMTOKEN` : un pseudo-nom XML (un nom XML sans la restriction sur le premier caractère) ;

Types de données 3/3



Typage des données

- *valeur* `cast as xs:string`;
- *valeur* `cast as xdt:untypedAtomic`;
- *true* `cast as xs:integer` ou `xs:float` ou `xs:double` ou `xs:decimal` devient *1*, *1.0E0*, *1.0E0*, *1.0*;
- *déc* ou *int* `cast as xs:integer` ou `xs:float` ou `xs:double` ou `xs:decimal` (arrondi à l'entier le plus petit si > 0 ou plus grand si < 0);
- *déc* ou *int* ou *flt* ou *dbl* `cast as xs:boolean` (`false` si égal à zéro, `true` sinon);
- etc.

Il voit des séquences partout

- Sous XPath2 tout est *séquence* ;
- les séquences sont *plates* ;
- elles sont *ordonnées* ;
- les nœuds peuvent se répéter.

Fonctions sur les chaînes

- `fn:concat(`*`ch1`*`,`*`ch2`*`)` concatène 2 ou plusieurs `xdt:anyAtomicType` en un `xs:string` ;
- `fn:string-join(`*`ch1`*`,`*`ch2`*`)`, `fn:substring(`*`ch1`*`,`*`déb`*`,`*`long`*`?)`, `fn:string-length(`*`ch`*`)` ;
- `fn:normalize-space(`*`ch`*`)` : enlève les blancs qui entourent la chaîne et remplace les blancs multiples par des `0020 SPACE` seuls ;
- `fn:normalize-unicode(`*`ch`*`,`*`méthode`*`?)` : normalise en NFC (oblig.) ou en NFD, NFKC, NFKD (opt.).
- `fn:upper-case(`*`ch`*`)`, `fn:lower-case(`*`ch`*`)` ;
- `fn:translate(`*`ch`*`,`*`orig`*`,`*`rés`*`)` :

Expressions régulières

- `fn:matches(`*`entrée`*`,`*`motif`*`,`*`flags`*`?)` retourne `true` si l'entrée contient une sous-chaîne conforme au motif ;
- `fn:replace(`*`entrée`*`,`*`motif`*`,`*`remplacement`*`,`*`flags`*`?)` transforme l'entrée en remplaçant le motif par le remplacement et retourne le résultat ;
- `fn:tokenize(`*`entrée`*`,`*`motif`*`,`*`flags`*`?)` sépare l'entrée en sous-chaînes séparées par le séparateur, et retourne le résultat sous forme de séquence de chaînes.

Rappel : les *expressions régulières*

Une *expression régulière* est une chaîne de caractères, délimiteurs, quantificateurs et jokers qui permet de spécifier avec précision un ensemble cible de chaînes de caractères.
Exemples :

- `[A-Z][a-z]+;`
- `(XML|xml);`
- `[+-]?[0-9]\.[0-9]{3,};`
- `[\p{IsGreek}]{7}.`

Ingrédients des expressions régulières

- les *quantificateurs* : `?` (zéro ou un), `*` (zéro, un ou plusieurs), `+` (un ou plusieurs), `{n,m}` (entre *n* et *m* fois), `{n,}` (au moins *n* fois) ;
- les *quantificateurs étendus* : `??` (zéro ou un), `*?` (zéro, un ou plusieurs), `+?` (un ou plusieurs), `{n,m}?` (entre *n* et *m* fois), `{n,}?` (au moins *n* fois) ;
- les les *parenthèses* permettent de grouper les expressions (par exemple : `(a[bc])?` représente la chaîne vide et les chaînes `ab` et `ac`) ;
- les *crochets* indiquent les ensembles ou intervalles de caractères : `[abc]` représente une lettre parmi `a`, `b` et `c` ; `[a-z]` représente une lettre entre `a` et `z` ; `[^a-z]` représente tout caractère Unicode qui ne soit pas compris entre `a` et `z` ;

Ingrédients des expressions régulières

- le *point* indique tout caractère Unicode, en «mode `s`» ou tout caractère mis à part NL, en mode par défaut ;
- le *tiret* pour indiquer des chaînes de caractères à soustraire de l'ensemble des chaînes trouvées, par exemple : `Le ([A-Z][a-z]+) -(Pen)` ;
- les *caractères d'échappement* indiquent les caractères qui sinon sont utilisés dans les expressions : `\\`, `\|`, `\.`, `\?`, `\*`, `\+`, `\(`, `\)`, `\{`, `\}`, `\[`, `\]`, `\^`, `\-` ;

Ingrédients des expressions régulières

- les *classes de caractères* et leurs *complémentaires* : par exemple, `\p{L}` représente tous les caractères Unicode qui sont des lettres, `\P{Nd}` représente tous les caractères Unicode qui ne sont pas des chiffres décimaux, `\p{IsGurmukhi}` toutes les lettres Gurmukhi, `\P{IsArabic}` tous les caractères Unicode qui ne sont pas des lettres arabes, etc. ;
- les *classes abrégées* : `\s` (un blanc ou une tabulation ou un CR ou un LF), `\i` (un caractère de début de nom XML), `\c` (un caractère de nom XML), `\d` (un chiffre), `\w` (tout caractère qui n'est pas *ponctuation* ou *séparateur* ou *autre*) et leurs complémentaires : `\S`, `\I`, `\C`, `\D`, `\W`.

Ingrédients des expressions régulières

- les *flags* : `s` (le «`.`» capte tout le monde, y compris NL), `m` (`^` est le début et `$` la fin de ligne), `i` (majuscules et minuscules confondus), `x` pour ignorer les blancs dans un motif ;
- les *caractères de début et de fin de chaîne* : `^` et `$` (sauf en mode `m`) ;
- les sous-expressions parenthésées sont obtenues dans la chaîne de remplacement par n ;
- dans un motif on dispose de références en arrière \n.

Fonctions sur les séquences

- `fn:boolean(`*séq* calcule la «valeur booléenne effective» de la séquence (`false` si la séquence est `(false)`, `("")`, `(0)` ou `(NaN)` ;
- `fn:empty(`*séq*`)` : retourne `true` si la séquence est vide ;
- `fn:exists(`*séq*`)` : retourne `true` si la séquence est non vide ;
- `fn:index-of(`*séq*`,`*obj*`)` retourne la ou les position(s) de `obj` dans `séq` ;
- `fn:distinct-values(`*séq* : supprime les doublons (mais on ne sait pas lesquels) ;
- `fn:insert-before(`*séq*`,`*n*`,`*séq2*`)` insère séq2 au début de séq (si n=0), à la fin de séq (si n=card(séq)), devant le n-ième élèment sinon ;
- `fn:remove(`*séq*`,`*n*`)` : supprime le n-ième élément ;
- `fn:reverse(`*séq*`)` : inverse l'ordre des éléments ;

Fonctions sur les séquences

- `fn:subsequence(séq,n,l)` : retourne la sous-séquence de taille l à partir du n-ième élément ;
- `fn:unordered(séq)` : retourne la séquence dans un ordre dépendant de l'implémentation.
- `fn:union(séq,séq2)` : retourne l'union des deux séquences ;
- `fn:intersection(séq,séq2)` : retourne l'intersection des deux séquences ;
- `fn:except(séq,séq2)` : retourne le complémentaire de l'intersection de deux séquences ;
- `fn:deep-equal(séq,séq2,tri?)` : retourne `true` si les nœuds des deux séquences sont égaux (s'ils sont atomiques leurs valeurs doivent être égales, sinon ils doivent avoir les mêmes noms et des enfants égaux) ;

Fonctions sur les séquences

- `fn:count(`*séq*`)`, `fn:avg(`*séq*`)`, `fn:max(`*séq*`)`, `fn:min(`*séq*`)`, `fn:sum(`*séq*`)` : le nombre, la moyenne, le max, le min, la somme des termes d'une séquence.
- `fn:id(`*ch*`)`, `fn:idref(`*ch*`)` : retourne la séquence de nœuds possédant des attributs de type `ID` (resp. `IDREF`);
- `fn:doc(`*uri*`)` : retourne le nœud document du document XML référencé par cette URI ;
- `fn:doc-available(`*uri*`)` : retourne `true` si le document XML référencé par cette URI est accessible ;

Boucles, tests, séquences quantifiées

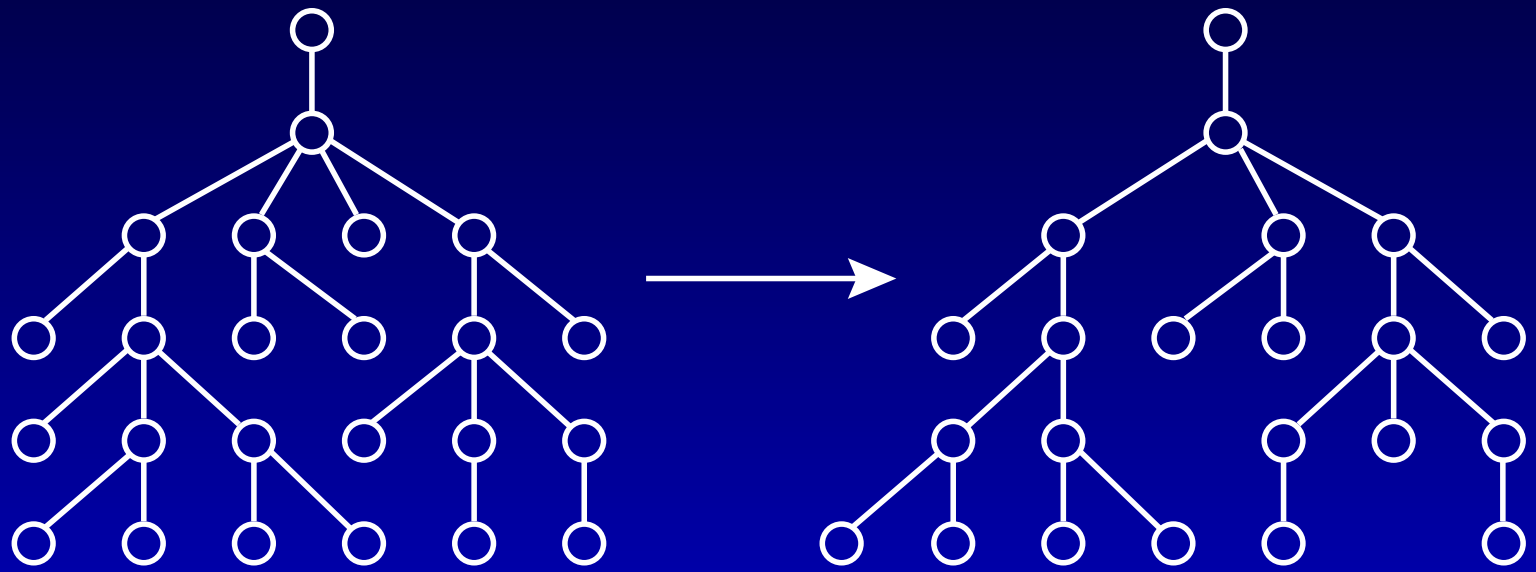
- `for $a in (séq), $b in (séq2)`
  `return (séq3)`
- `if (test) then expr1 else expr2`
- `if ($a castable as xs:boolean) then expr1 else expr2`
- `some $a in (séq) satisfies expr`
- `every $a in (séq) satisfies expr`

XSLT

Version 1.0, du 16 novembre 1999

Transformation XSLT

Une *transformation XSLT* va transformer un arbre source en un arbre résultat.



Feuille de style XSLT

Une *feuille de style* est représentée par l'élément `xsl:stylesheet` dans l'espace de noms `xsl` :

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

Un élément fils de `xsl:stylesheet` est appelé *élément de haut niveau*.

Éléments de haut niveau

- `xsl:template`;
- `xsl:output`;
- `xsl:variable`, `xsl:param`;
- `xsl:import`, `xsl:include`;
- `xsl:strip-space`, `xsl:preserve-space`;
- `xsl:key`, `xsl:decimal-format`, `xsl:namespace-alias`, `xsl:attribute-set`.

Les règles modèle

On spécifie une *règle modèle* (*template*) en utilisant l'élément `xsl:template`. L'attribut `match` prend comme valeur une expression XPath d'axe `child`. Exemple :

```
<xsl:template match="section/titre">
...texte de remplacement du nœud... </xsl:template>
```

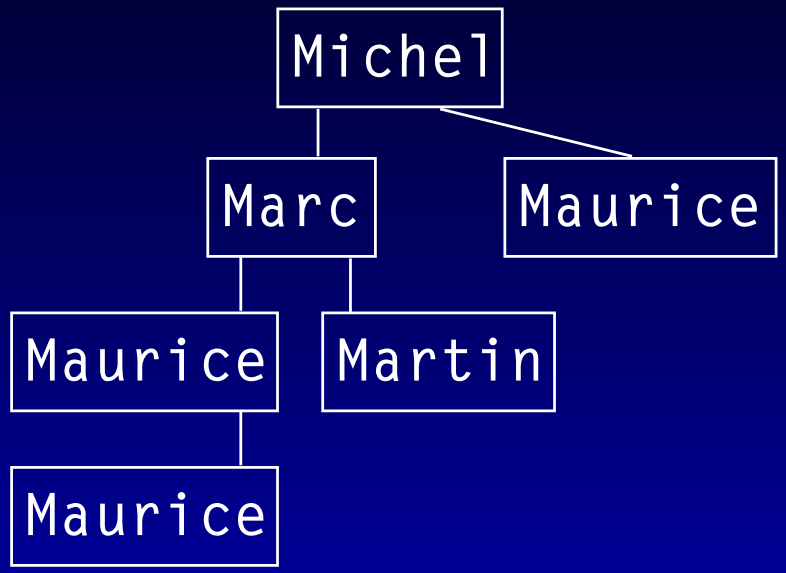
Les règles modèle

- On parcourt l'arbre en commençant par le nœud racine.
- Si le nœud courant ne correspond pas à l'expression XPath de l'attribut `match`, on examine ses enfants. Si le nœud n'a pas d'enfant, on passe au nœud suivant.
- Un nœud qui correspond à l'expression XPath est remplacé par le *texte de remplacement* (qui peut éventuellement contenir des balises XML ou XHTML). On n'examine pas ses enfants.

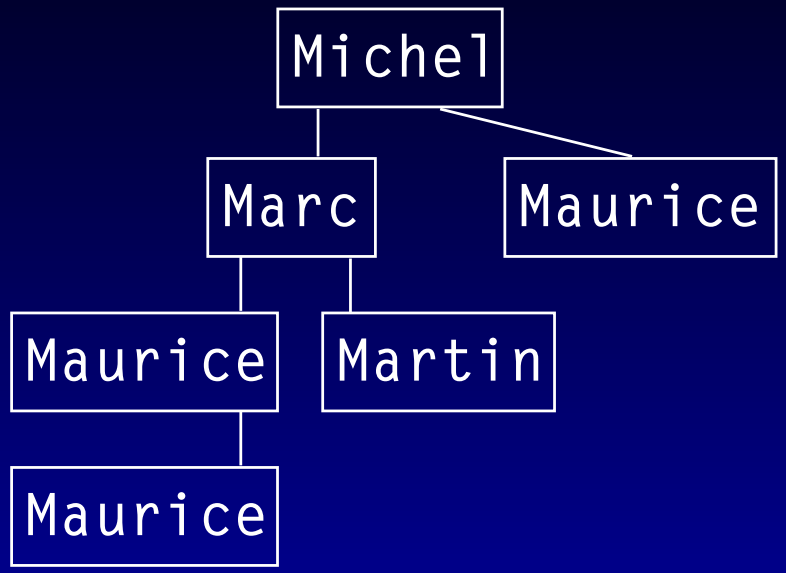
Mieux comprendre

Exemple :

```
<Michel><Marc>
<Maurice><Maurice>
</Maurice></Maurice>
<Martin></Martin>
</Marc>
<Maurice></Maurice>
</Michel>
```



Mieux comprendre



Quel va être l'effet de :

```
<xsl:template match="*"> MORT
</xsl:template>
```

?

Et l'effet de `match="Maurice"` ?
Comment tuer tous les `Maurice` ?

Traitement récursif

L'élément `<xsl:apply-templates/>` traite de manière récursive les *enfants* du nœud courant. Exemple :

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

On peut avoir plusieurs règles `xsl:template`. Le processeur utilisera la plus appropriée pour chaque nœud.

Traitement récursif filtré 1/2

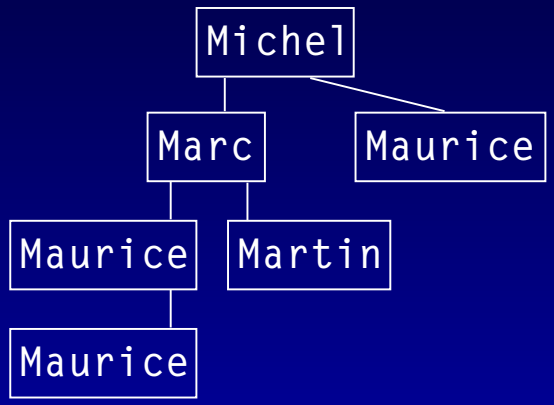
L'attribut `select` de `<xsl:apply-templates/>` permet d'appliquer la récursion à d'autres nœuds que les enfants. Exemple :

```
<xsl:template match="Marc"> MORT
<xsl:apply-templates select="Maurice"/>
</xsl:template>
```

La valeur par défaut de cet attribut est * (tous les enfants du nœud courant).

Exemples

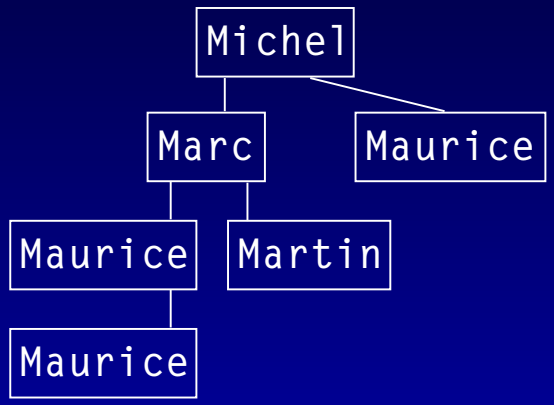
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="Maurice"/></xsl:template>
```



Résultat :

Exemples

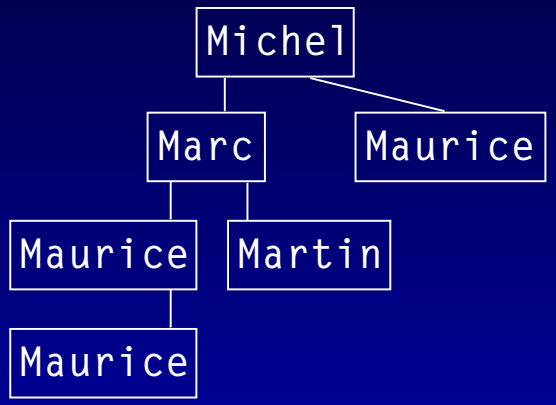
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="Maurice"/></xsl:template>
```



Résultat : Bonsoir Bonjour

Exemples

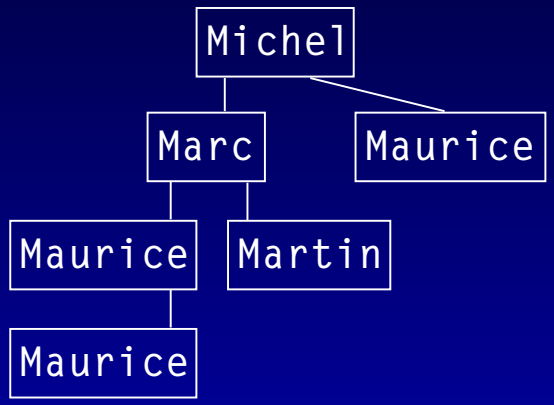
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="Marc/Maurice"/></xsl:template>
```



Résultat :

Exemples

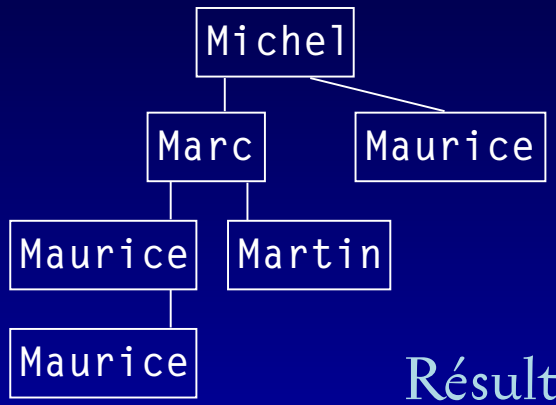
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="Marc/Maurice"/></xsl:template>
```



Résultat : Bonsoir Bonjour

Exemples

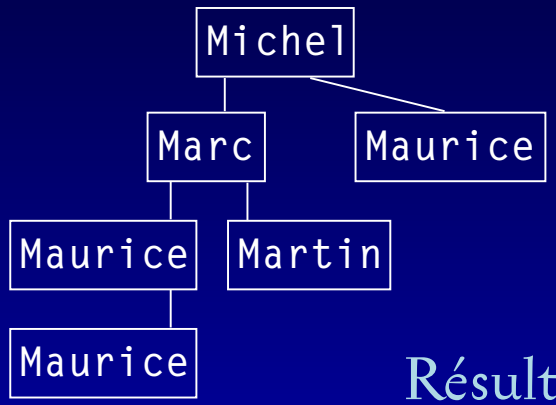
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="Marc//Maurice"/></xsl:template>
```



Résultat :

Exemples

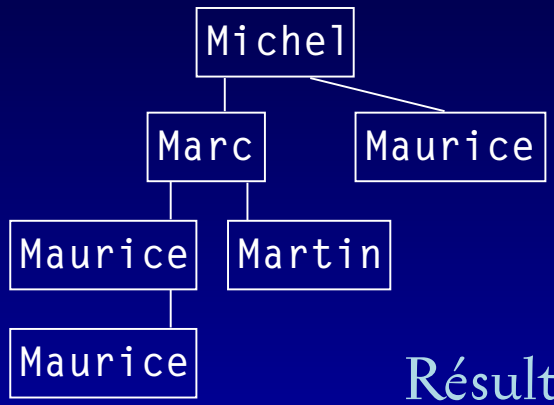
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="Marc//Maurice"/></xsl:template>
```



Résultat : Bonsoir Bonjour Bonjour

À éviter

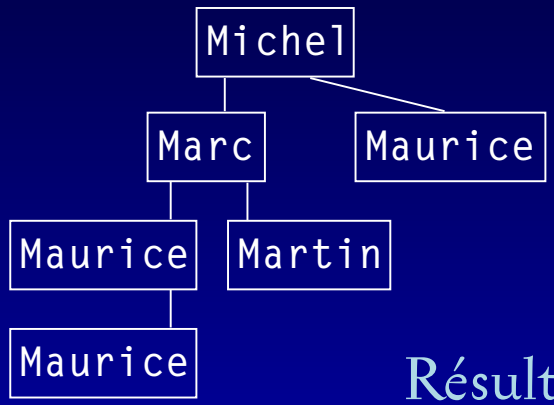
```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="."/></xsl:template>
```



Résultat :

À éviter

```
<xsl:template match="Maurice">Bonjour</xsl:template>
<xsl:template match="Michel">Bonsoir
<xsl:apply-templates select="."/></xsl:template>
```



Résultat : Segmentation fault (core dumped)...

Traitement récursif filtré 2/2

La valeur de l'attribut `select` de `<xsl:apply-templates/>` peut être une expression XPath quelconque, y compris une expression utilisant l'axe `ancestor` et des tests. Exemple :

```
<xsl:apply-templates select="ancestor::livre//titre[@id='s5']"/>
```

va appliquer les règles modèle à l'élément `titre` qui se trouve quelque part sous `livre` (un ancêtre du nœud courant) et qui a un attribut `id` de valeur `s5`.

Les valeurs des nœuds

Pour obtenir la *valeur* d'un nœud on utilise `<xsl:value-of select="."/>`. L'attribut `select` permet de limiter l'application de la «commande» à un certain type de nœud.

Règles modèle implicites

En l'absence de règles explicites le processeur XSLT utilisera les règles implicites suivantes :

```
<xsl:template match="*|/">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="text()|@*">
<xsl:value-of select="string(.)"/>
</xsl:template>
<xsl:template match="processing-instruction()|comment()"/>
```

Règles modèle implicites : exemple

```
<chapter id="1"><p>Un paragraphe.</p>
<p couleur="rouge">Un autre, mais rouge.</p>
</chapter>
```

auquel on applique une feuille de style vide, donnera :

Règles modèle implicites : exemple

```
<chapter id="1"><p>Un paragraphe.</p>
<p couleur="rouge">Un autre, mais rouge.</p>
</chapter>
```

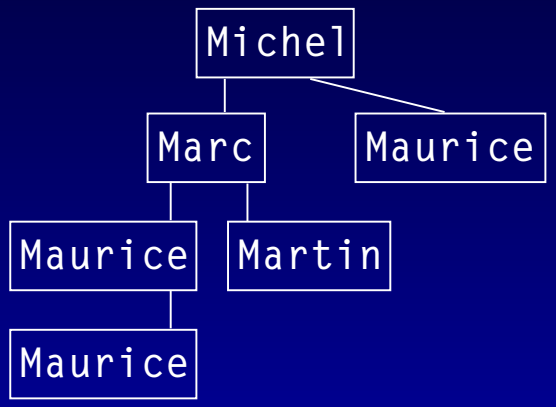
auquel on applique une feuille de style vide, donnera :

```
<?xml version="1.0" encoding="utf-8"?>Un paragraphe. Un autre, mais
rouge.
```

Application de la règle implicite

Re-examinons l'exemple précédent :

```
<xsl:template match="Maurice">Bonjour</xsl:template>
```

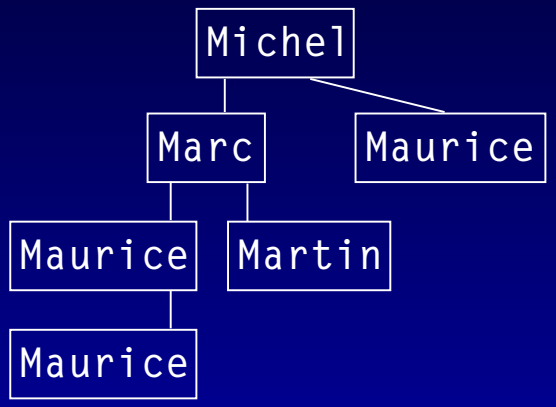


Résultat :

Application de la règle implicite

Re-examinons l'exemple précédent :

```
<xsl:template match="Maurice">Bonjour</xsl:template>
```



Résultat : Bonjour Bonjour

Méthodes de sortie

L'élément `<xsl:output method="xml"/>` nous permet de spécifier le type de la sortie du processeur XSLT : `xml`, `html` ou `text`. D'autres attributs nous permettent de spécifier la déclaration XML et la DTD. Exemple :

```
<xsl:output method="xml" version="1.0"
      standalone="no" encoding="utf-8"
      doctype-system="../dtds/toto.dtd"
      indent="yes">
```

Créer des nœuds

Les éléments suivants nous permettent de créer des nœuds :

- `<xsl:element name="nom">`
- `<xsl:attribute name="nom">`
- `<xsl:processing-instruction name="nom">`
- `<xsl:text>`
- `<xsl:comment>`

Grouper les attributs

Lorsqu'il y a des attributs (noms et valeurs) qui se répètent, on peut les grouper dans un `attribute-set` :

```
<xsl:attribute-set name="nom">
<xsl:attribute name="att1">a</xsl:attribute>
<xsl:attribute name="att2">b</xsl:attribute>
<xsl:attribute name="att3">c</xsl:attribute>
</xsl:attribute-set>
```

On l'utilise, par exemple, dans `<xsl:element attribute-set="nom">`.

Copie de nœud

On utilise `xsl:copy` pour copier des nœuds (de type élément, attribut, IT, commentaire ou texte). Cette opération peut être récursive. La *transformation identité* s'écrit par exemple :

```
<xsl:template match="@*|node()">
<xsl:copy>
<xsl:apply-templates select="@*|node()"/>
</xsl:copy>
</xsl:template>
```

Tris

On peut trier un ensemble de nœuds produit par une «commande» `<xsl:apply-templates>` ou `<xsl:for-each>`. On se sert de l'élément (vide) :

```
<xsl:sort select="critère"
      lang="langue"
      data-type="type de données"
      order="ordre"/>
```

Les valeurs possibles de `data-type` sont `text` et `number`. Celles de `order` sont `ascending` ou `descending`.

Exemple de tri

Soit un document XML contenant un grand nombre d'éléments `<personne>`, ayant comme sous-éléments `<nom>` et `<prénom>`. Pour trier par rapport aux noms, on écrira :

```
<xsl:template match="/">
<xsl:apply-templates select="personne">
<xsl:sort select="nom"/>
</xsl:apply-templates></xsl:template>
```

Attention ! Dans ce cas, `<xsl:apply-templates>` n'est plus un élément vide !

Tris multicritère

En écrivant plusieurs `<xsl:sort>` de suite, on obtient un tri multicritère. Par exemple, pour trier par rapport aux noms, et ensuite par rapport aux prénoms, on écrira :

```
<xsl:template match="/">
<xsl:apply-templates select="personne">
<xsl:sort select="nom"/>
<xsl:sort select="prénom"/>
</xsl:apply-templates></xsl:template>
```

Modes

On peut entrer dans un *mode* et n'exécuter des règles modèle spécifiques à lui :

- `<xsl:apply-templates select="qqch" mode="début"/>`
- `<xsl:template match="itou" mode="début">`
- le nom de mode est un nom XML, à deux exceptions près :
- `#default` : le mode par défaut ;
- `#all` : tous les modes.

Variables

On peut définir des *variables* et les utiliser par la suite dans la feuille de style XSL. On distingue deux types de variables :

- les *variables expression* : leurs valeurs sont des expressions XPath.
- les *variables fragment d'arbre* : leurs valeurs sont des fragments d'arbre de nœud, c'est-à-dire des nœuds de tout type, ayant un ancêtre commun qui est la racine du fragment d'arbre.

Variables expression

On utilise l'élément `<xsl:variable/>`. Dans ce cas, cet élément est vide et a deux attributs :

- `name` : le nom de la variable ;
- `select` : sa valeur : une expression XPath.

Variables fragment d'arbre

On utilise l'élément `<xsl:variable>`. Dans ce cas, l'élément n'est pas vide et n'a qu'un seul attribut :

- `name` : le nom de la variable ;

Le contenu de `<xsl:variable>` est la valeur de la variable.

Portée des variables

La portée d'une variable est celle de l'élément XSL dans lequel elle est définie. En particulier, une variable définie dans un `<xsl:template>` ne sera visible que dans cet élément. L'appel d'une variable non définie ou hors portée engendre une erreur de compilation.

Les variables de haut niveau sont visibles partout (à partir de l'endroit où elles ont été définies).

Utilisation des variables

On utilise les variables dans les valeurs d'attribut de certains éléments XSL. On écrit le nom de la variable précédé d'un dollar `$`.

On peut aussi les utiliser dans les attributs d'éléments non-XSL. Dans ce cas on inclut leur nom entre accolades `{}`.

Exemple :

```
<xsl:variable name="n" select="20"/>
<xsl:value-of select="item[position()=$n]"/>
<img width="{$n}" height="{$n}" src="toto.gif"/>
```

Produire des suites de nombres formatés

La «commande» `<xsl:number/>` nous permet de numéroter automatiquement des nœuds. L'attribut `format` indique le type de numérotation :

- `format="1"` : chiffres arabes 1, 2, ... ;
- `format="i"` : chiffres romains i, ii, ... ;
- `format="a"` : lettres minuscules a, b, ... aa, ab, ... ;
- `format="A"` : lettres majuscules A, B, ... AA, AB, ... ;
- d'autres alphabets et systèmes de numérotation peuvent être utilisés.

Produire des suites de nombres formatés

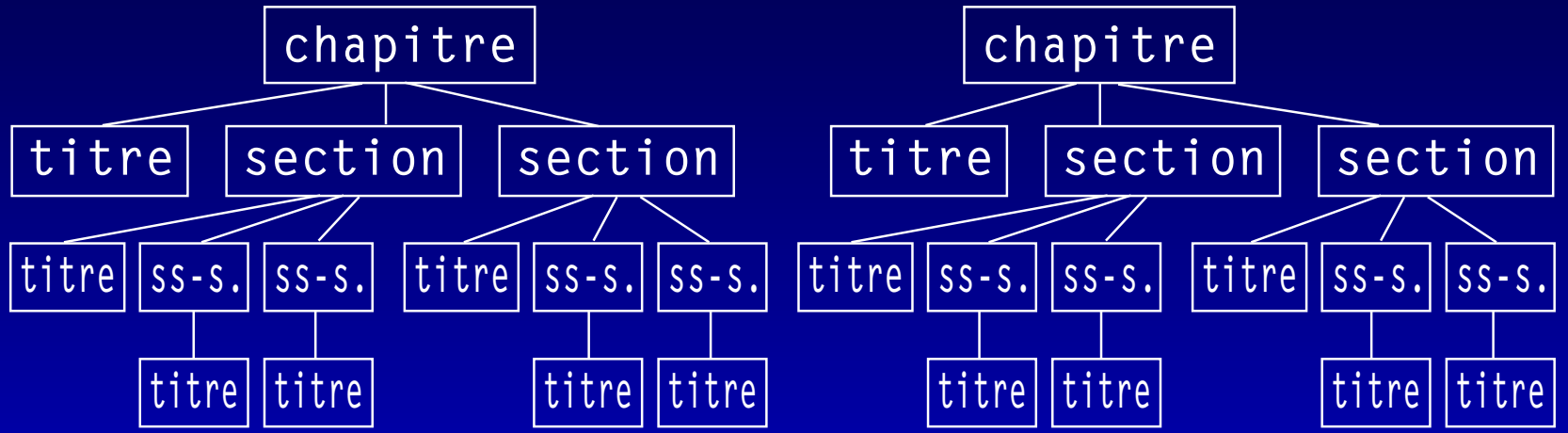
L'attribut `value` peut prendre une expression XPath comme valeur. Exemple :

```
<xsl:template match="item">
<xsl:number value="position()" format="1."/>
<xsl:apply-templates select="*"/>
</xsl:template>
```

va numéroter les éléments `<item>` successifs d'une liste.

Et si l'on compliquait un peu ?

Soit un document XML avec des éléments `<chapitre>` qui contiennent des éléments `<section>`, qui contiennent des éléments `<sous-section>`. Chacun de ces éléments possède un sous-élément `<titre>`.



Et si l'on compliquait un peu ?

```
<xsl:template match="titre">
Titre <xsl:number level="single"
count="chapitre|section|sous-section" format="1."/>
<xsl:apply-templates select="*"/>
</xsl:template>
```
nous fournit :

Titre (de chapitre) 1.
Titre (de section) 1.
Titre (de sous-section) 1., Titre (de sous-section) 2., etc.

Et si l'on compliquait davantage ?

Autrement dit :

- le *premier* ancêtre de l'élément courant, et qui satisfait à la condition `count` est choisi ;
- son numéro d'ordre correspond à sa position par rapport à ses frères (satisfaisant également à la condition).

Et si l'on compliquait davantage ?

```
<xsl:template match="titre">
Titre <xsl:number level="multiple"
count="chapitre|section|sous-section" format="1."/>
<xsl:apply-templates select="*"/>
</xsl:template>
```
nous fournit :

Titre (de chapitre) 1.
Titre (de section) 1.1.
Titre (de sous-section) 1.1.1., Titre (de sous-section) 1.1.2., etc.

Et si l'on compliquait davantage ?

Autrement dit :

- *tous* les ancêtres de l'élément courant, et qui satisfont à la condition `count` sont choisis ;
- chacun provoque la création d'une liste composée de lui-même et de ses frères (satisfaisant également à la condition) ;
- de chaque liste on obtient un numéro d'ordre ;
- on concatène ces numéros d'ordre, dans le bon ordre.

Et si l'on compliquait davantage ?

```
<xsl:template match="titre">
Titre <xsl:number level="multiple"
count="chapitre|section|sous-section" format="1.A.a."/>
<xsl:apply-templates select="*"/>
</xsl:template>
```
nous fournit :

Titre (de chapitre) 1.
Titre (de section) 1.A.
Titre (de sous-section) 1.A.a., Titre (de sous-section) 1.A.b., etc.

Une autre approche du comptage

Pour compter, par exemple, les *notes* d'un document, on se servira d'un comptage de type `any` :

```
<xsl:template match="note">
<xsl:number level="any" count="note" format="1."/>
<xsl:apply-templates/>
</xsl:template>
```

Autrement dit : pour chaque élément `note` on compte les éléments de même nom qui se trouvent avant lui, dans l'ordre du document.

Se limiter à une branche donnée

À l'aide de l'attribut `from="`*`expression`*`"` on peut limiter le comptage à une branche de l'arbre qui commence par le premier ancêtre du nœud courant qui satisfasse l'expression *`expression`*. Exemple :

```
<xsl:template match="note">
<xsl:number level="any" count="note" from="chapitre" format="1."/>
<xsl:apply-templates/></xsl:template>
```

va numéroter les notes en recommençant à 1 dans chaque chapitre.

Boucles

L'élément `<xsl:for-each select="elem">` répète son contenu pour chaque nœud *elem*. Exemple (basé sur l'exemple précédent) :

```
<xsl:template match="/">
<xsl:for-each select="chapter">
<xsl:apply-templates select="section"/>
<xsl:apply-templates select="titre"/>
</xsl:for-each>
</xsl:template>
```

Tests

L'élément `<xsl:if test="test">` permet de tester une expression XPath booléenne. Exemple :

```
<xsl:template match="sous-section/titre">
<xsl:apply-templates/>
<xsl:if test="not(position()=last())">
<xsl:text>,</xsl:text>
</xsl:if>
</xsl:template>
```

Attention ! Il n'y a malheureusement pas de `<xsl:else/>`...

Sélection

Les éléments `<xsl:choose>`, `<xsl:when test="test">` et `<xsl:otherwise>` permettent un traitement cas par cas d'une situation. Exemple :

```
<xsl:choose>
<xsl:when test="position()=1">premier</xsl:when>
<xsl:when test="position()=last()">dernier</xsl:when>
<xsl:otherwise>autres</xsl:otherwise>
</xsl:choose>
```

Lien entre doc. XML et feuille de style XSL

La feuille de style est écrite dans un fichier `.xsl` séparé. On peut lier le document XML de base et sa feuille de style par l'instruction de traitement suivante, insérée au début du document XML :

```
<?xml-stylesheet href="nom-fichier" type="text/xsl"?>
```

Particularité : les attributs `href` et `type` ne sont que des *pseudo-attributs* puisqu'une IT n'est pas censée avoir des attributs...

XSLT 2

Version 2.0, WD du 11 février 2005

Nouvelles fonctionnalités

Quoi 2 9 ?

- l'espace de nom de XLST 2 est le même, mais sa version est désormais 2 :

```
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

- XSLT 2 utilise les *séquences* et les *expressions régulières* de XPath 2 ;
- on peut écrire dans des documents de sortie multiples ;
- on peut classer les nœuds en groupes ;
- on peut définir des fonctions.

Séquences

L'élément `<xsl:sequence>` permet la création de séquences de nœuds.
Exemple :

```
<xsl:variable name="values" as="xs:integer*">
<xsl:sequence select="(1,2,3,4)"/>
<xsl:sequence select="(8,9,10)"/>
</xsl:variable>
<xsl:value-of select="sum($values)"/>
```

retourne 37.

Séquences

Autre exemple :

```
<xsl:variable name="prices" as="xs:decimal*">
<xsl:for-each select="//product">
<xsl:choose>
<xsl:when test="@price">
<xsl:sequence select="@price"/>
</xsl:when>
<xsl:otherwise>
<xsl:sequence select="@cost * 1.5"/>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:variable>
<xsl:value-of select="avg($prices)"/>
```

Autre version :

```
<xsl:value-of select="avg(//product/(+@price, @cost*1.5)[1])"/>
```

(pourquoi le `[1]` ? pourquoi le `+` ?)

Expressions régulières 1/3

Les éléments `<xsl:analyze-string>`, `<xsl:matching-substring>` et `<xsl:non-matching-substring>` permettent l'application d'expression régulières à des contenus textuels de séquences de nœuds. Exemple :

```
<xsl:analyze-string select="abstract" regex="\n">
<xsl:non-matching-substring>
<xsl:value-of select="."/>
</xsl:non-matching-substring>
<xsl:matching-substring>
<br/>
</xsl:matching-substring>
</xsl:analyze-string>
```

Ségmentation de la chaîne en sous-chaînes trouvées et intercalées.

Expressions régulières 2/3

```
<xsl:analyze-string select="body" regex="\[(.*?)\]">
<xsl:matching-substring>
<cite><xsl:value-of select="regex-group(1)"/></cite>
</xsl:matching-substring>
<xsl:non-matching-substring>
<xsl:value-of select="."/>
</xsl:non-matching-substring>
</xsl:analyze-string>
```

Expressions régulières 3/3

```
<xsl:variable name="months" select="'janvier', 'février', 'mars',
..."/>

<xsl:analyze-string select="normalize-space($input)"
regex="([0-9]{{1,2}})\s([a-zéû]+)\s([0-9]{{4}})">
<xsl:matching-substring>
<xsl:number value="regex-group(3)" format="0001"/>
<xsl:text>-</xsl:text>
<xsl:number value="index-of($months, regex-group(2))" format="01"/>
<xsl:text>-</xsl:text>
<xsl:number value="regex-group(1)" format="01"/>
</xsl:matching-substring>
</xsl:analyze-string>
```

(Attention : on double le accolades !)

Documents de sortie

```
<xsl:result-document
  format? = qname
  href? = { uri-reference }
  method? = { "xml" | "html" | "xhtml" | "text" }
  byte-order-mark? = { "yes" | "no" }
  doctype-public? = { string }
  doctype-system? = { string }
  encoding? = { string }
  indent? = { "yes" | "no" }
  normalization-form? = { "NFC" | "NFD" | "NKFC" | "NKFD" | "none" |
nmtoken }
  omit-xml-declaration? = { "yes" | "no" }
  standalone? = { "yes" | "no" | "omit" }
  output-version? = { nmtoken }>
...
</xsl:result-document>
```

Groupes

L'élément `<xsl:for-each-group>` permet la répartition d'une séquence de nœuds en sous-séquences. Il prend l'un des quatre attributs :

- `group-by` ;
- `group-adjacent` ;
- `group-starting-with` ;
- `group-ending-with`.

Groupes avec group-by

À partir de :

```
<cities>
<city name="Milan" country="Italie" pop="5"/>
<city name="Paris" country="France" pop="7"/>
<city name="Venise" country="Italie" pop="1"/>
<city name="Lyon" country="France" pop="2"/>
</cities>
```

obtenir :

```
<table>
<tr> <th>Numéro</th> <th>Pays</th> <th>Liste de villes</th>
<th>Population</th> </tr>
<tr> <td>1</td> <td>Italie</td> <td>Milan, Venise</td> <td>6</td>
</tr>
<tr> <td>2</td> <td>France</td> <td>Lyon, Paris</td> <td>9</td> </tr>
```

Groupes avec group-by

On utilisera :

```
<xsl:template match="cities">
<table><tr> <th>Numéro</th> <th>Pays</th> <th>Liste de villes</th>
<th>Population</th> </tr>
<xsl:for-each-group select="city" group-by="@country">
<tr>
<td><xsl:value-of select="position()"/></td>
<td><xsl:value-of select="@country"/></td>
<td>
<xsl:value-of select="current-group()/@name" separator=", "/>
</td>
<td><xsl:value-of select="sum(current-group()/@pop)"/></td>
</tr>
</xsl:for-each-group>
</table>
</xsl:template>
```

Groupes avec group-starting-with

À partir de :

```
<body>
<h2>Objectif du cours</h2>
<p>Vous apprendre quelque chose</p>
<h2>Objectif des TP</h2>
<p>Mettre la connaissance en pratique</p>
</body>
```

obtenir :

```
<body>
<section title="Objectif du cours">
<para>Vous apprendre quelque chose</para>
</section>
<section title="Objectif des TP">
<para>Mettre la connaissance en pratique</para>
</section>
```

Groupes avec group-starting-with

On utilisera :

```
<xsl:template match="body">
<xsl:for-each-group select="*" group-starting-with="h2" >
<section title="self::h2">
<xsl:for-each select="current-group()[self::p]">
<para><xsl:value-of select="."/></para>
</xsl:for-each>
</section>
</xsl:for-each-group>
</xsl:template>
```

Groupes avec `group-ending-with`

À partir de :

```
<doc>
<page cont="oui">Du texte</page>
<page cont="oui">Et encore du texte</page>
<page>Et ainsi de suite</page>
<page cont="oui">Et vogue le navire</page>
<page>de plus en plus loin...</page>
</doc>
```

obtenir :

```
<doc><pageset>
<page>Du texte</page>
<page>Et encore du texte</page>
<page>Et ainsi de suite</page>
</pageset><pageset>
<page>Et vogue le navire</page>
<page>de plus en plus loin...</page>
</pageset></doc>
```

Groupes avec group-ending-with

On utilisera :

```
<xsl:template match="doc">
<doc>
<xsl:for-each-group select="*"
  group-ending-with="page[not(@continued='yes')]">
<pageset>
<xsl:for-each select="current-group()">
<page><xsl:value-of select="."/></page>
</xsl:for-each>
</pageset>
</xsl:for-each-group>
</doc>
</xsl:template>
```

Groupes avec group-adjacent

À partir de :

```
<p>Dans vos <i>rapports</i>, soyez :
<ul>
<li>précis(e),</li>
<li>concis(e),</li>
<li>honnête</li>
</ul>
en toutes circonstances.</p>
```

obtenir :

```
<p>Dans vos <i>rapports</i>, soyez :</p>
<ul>
<li>précis(e),</li>
<li>concis(e),</li>
<li>honnête</li>
</ul>
<p> en toutes circonstances.</p>
```

Groupes avec group-adjacent

On utilisera :

```
<xsl:template match="p">
<xsl:for-each-group select="node()"
group-adjacent="self::ul or self::ol">
<xsl:choose>
<xsl:when test="current-grouping-key()">
<xsl:copy-of select="current-group()"/>
</xsl:when>
<xsl:otherwise>
<p><xsl:copy-of select="current-group()"/></p>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each-group>
</xsl:template>
```

Fonctions

Les éléments `<xsl:function>` et `<xsl:param>` permettent la définition de fonctions XSLT. Exemple : définir une fonction `str:reverse` qui inverse l'ordre des mots.

```
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:str="http://omega.enstb.org/yannis/examples">

<xsl:function name="str:reverse" as="xs:string">
<xsl:param name="sentence" as="xs:string"/>
<xsl:sequence select="if (contains($sentence, ' ')) then
concat(str:reverse(substring-after($sentence, ' ')), ' ',
substring-before($sentence, ' ')) else $sentence"/>
</xsl:function>

<xsl:template match="/">
<xsl:value-of select="str:reverse('YVES MORD YVONNE')"/>
</xsl:template></xsl:stylesheet>
```

Fonctions

Autre définition de la même fonction :

```
<xsl:function name="str:reverse" as="xs:string">
<xsl:param name="sentence" as="xs:string"/>
<xsl:choose>
<xsl:when test="contains($sentence, ' ')">
<xsl:sequence select="concat(str:reverse(substring-after($sentence, '
')),
' ',
substring-before($sentence, ' '))"/>
</xsl:when>
<xsl:otherwise>
<xsl:sequence select="$sentence"/>
</xsl:otherwise>
</xsl:choose>
</xsl:function>
```

Génération d'identifiants uniques

Le mécanisme des attributs `ID` de XML est quelque peu restrictif. XSL propose la fonction `generate-id(`*`nœud`*`)` qui va générer un identifiant unique pour le nœud *nœud*.

Si l'argument fourni à `generate-id` est un ensemble de nœuds, on en prend le premier.

Si l'argument est vide, on prend le nœud courant.

Les clés XSL : élément *xsl:key*

À l'aide de l'élément de haut niveau `xsl:key` on peut déclarer une famille d'identifiants uniques (appelés *clés*) qui a un nom, qui s'applique à un certain type de nœuds et dont les mambres prennent automatiquement certaines valeurs :

```
<xsl:key name="nom"
         match="nœuds"
         use="valeur"/>
```

Les clés XSL : fonction *key*

La fonction :

`key('`*nom*`',`*expression*`)`

retourne l'ensemble de nœuds qui ont une clé XSL de nom *nom* et de valeur, la valeur de l'expression XPath *expression*.

Les clés XSL : exemple

```
Le kig ha farz est un plat breton typique <bibref id="GuiRou"/>...
...
<bibliographie>
<bib abbr="GuiRou">Le Guide du routard</bib>
```

On souhaite obtenir :

```
Le kig ha farz est un plat breton typique [<a
href="#qqch">GuiRou</a>]...
<a name="qqch">Le Guide du routard</a>
```

Les clés XSL : exemple

```
<xsl:key name="biblio" match="bib" use="@abbr"/>

<xsl:template match="bib">
<a name="{generate-id()}">
<xsl:apply-templates/>
</a></xsl:template>

<xsl:template match="bibref">
<a href="#{generate-id(key('biblio',@id))}">
<xsl:value-of select="@id"/>
</a></xsl:template>
```

Index