Description, typage, modélisation et interrogation de données XML (XML, XML Schema, XDM, XQuery)

Jacques Le Maitre

UNIVERSITÉ du SUD
Toulon-Var

Objectif

- L'objectif de ce cours est double : apprendre à rédiger des documents XML et à les interroger en utilisant le langage XQuery.
- Pour pouvoir interroger un document XML, il faut :
  - typer les données qu'il contient ;
  - en donner une représentation abstraite.
- Les documents XML interrogés par XQuery :
  - doivent être typés conformément au langage XML Schema ;
  - sont représentés sous la forme d'arbres conformément au modèle XDM (XML Data Model).
- Après avoir étudié XML et avant d'étudier XQuery, nous étudierons XML Schema et XDM.

Plan

- XML
  - un langage de balisage extensible
- XML Schema
  - un langage de typage des données XML
- XDM
  - un modèle de données XML
- XQuery
  - un langage d'interrogation de données XML

XML
un langage de balisage extensible

http://www.w3.org/TR/xml11

Différents aspects d'un document

- Editorial
  - présentation du document.
- Signalétique
  - identification du document : ISBN, titre, auteurs, éditeur, année...
- **Structurel** (pris en compte par le langage XML)
  - organisation logique du document : découpage en chapitres et en paragraphes, figures, annotations...
- Sémantique
  - sujet traité par le document
- Multimédia
  - type des données véhiculées : textes, images, sons, animation...

De SGML à XML en passant par HTML

- SGML (1986)
  - a été inventé par Charles Goldfarb
  - son principe fondamental est la séparation totale entre la structure logique d'un document et sa mise en page
- HTML (1992)
  - a été proposé par Tim Berners-Lee et Dan Connolly comme langage de description des pages du Web
  - est basé sur SGML mais contrairement à celui-ci décrit la présentation d'une page web et non sa structure logique
- XML (1998)
  - a été proposé par un groupe de travail du W3C
  - "*Its goal is to enable generic SGML to be served, received, and processed on the web in the way that is now possible with HTML.*" (introduction à la recommandation du 18 février 1998)
  - comme SGML c'est un langage de balisage extensible, mais plus simple

Un exemple de document

- Le guide :

  Itinéraires skieurs dans la Vallée de la Clarée

  Jean-Gabriel Ravary

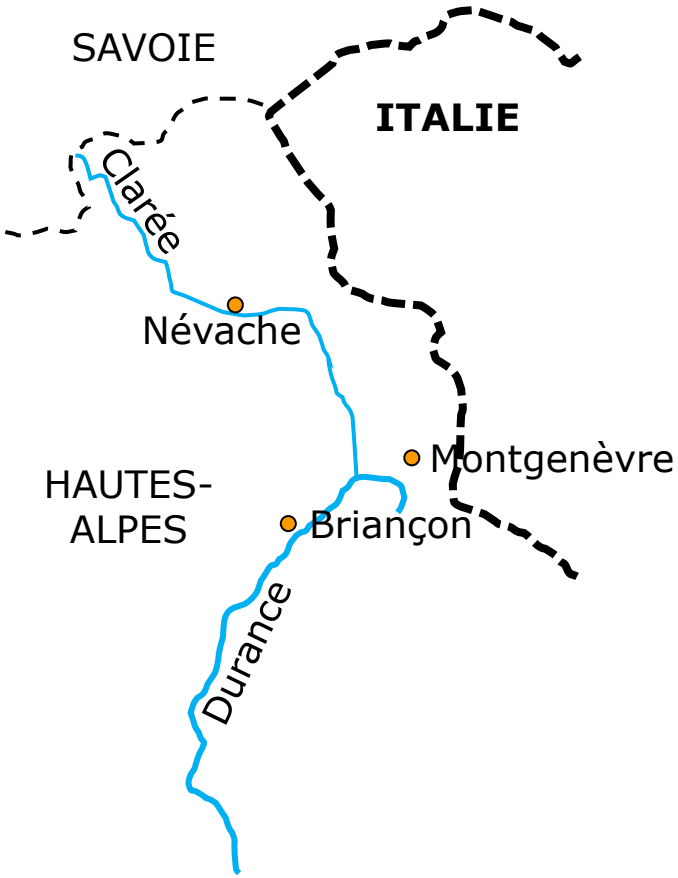
  Le Polygraphe (http://www.polygraphe.fr/)

  1991

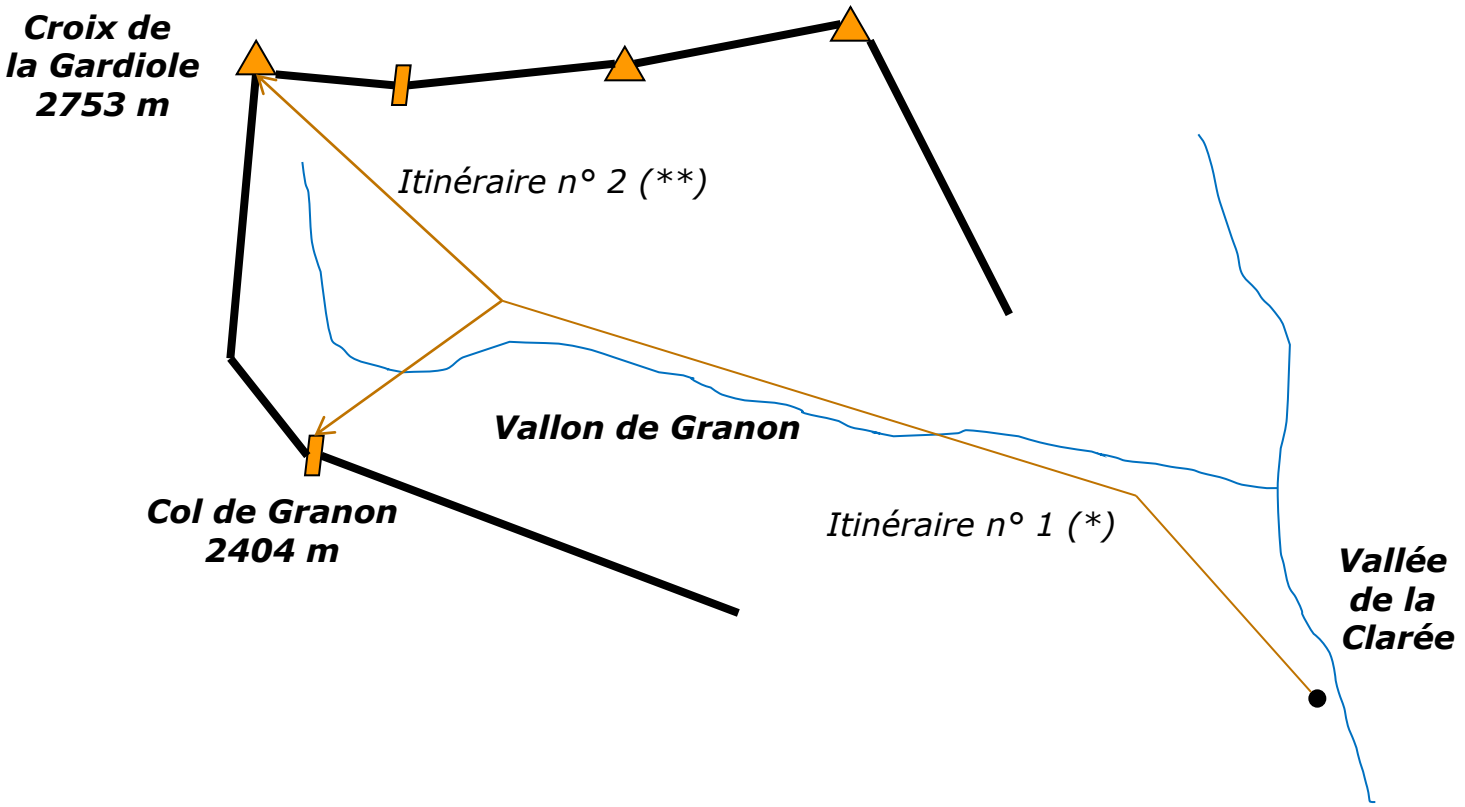
  (Je tiens à remercier Jean-Gabriel Ravary pour m'avoir autorisé à utiliser son livre pour illustrer ce cours.)
- La Vallée de la Clarée est située au nord de Briançon dans le département des Hautes-Alpes : c'est une vallée magnifique !

  (http://www.accueil-tourisme-nevache.com/fr/)

Situation géographique



Vallon et itinéraires



Extrait du guide

Itinéraires skieurs dans la Vallée de la Clarée

par Jean-Gabriel Ravary

Le Polygraphe, *éditeur*
1991

Vallon des Muandes

Vallon situé à l'est du refuge des Drayères.

Le vallon le plus utilisé pour la traversée sur la Vallée Etroite. Ce vallon est également accessible du refuge Laval.

Col de Névache (2 794 m) ** n° 1

S'élever au-dessus du refuge des Drayères en direction est. Suivre la rive droite du torrent de Brune puis s'engager sur le flanc droit du ravin des Muandes que l'on quitte vers 2500 m pour rejoindre le col situé au nord. Descente possible sur Valmeinier. Départ assez raide.

Pointe de Névache (2 892 m) * n° 2**

Du col de Névache (*itinéraire n° 1*), suivre la ligne de crête qui mène à la pointe de Névache. Attention : corniches possibles. Crampons utiles au printemps.

Le guide « Itinéraires skieurs » en XML (1)

```
<?xml version="1.0"?>
<guide>
<titre>Itinéraires skieurs dans la vallée de la Clarée</titre>
<auteur>Jean-Gabriel Ravary</auteur>
<editeur>Le Polygraphe</editeur>
<année>1991</année>
…
<vallon id="V15">
<nom>Vallon des Muandes</nom>
<intro>
<para>Vallon situé à l'est du refuge des Drayères.</para>
<para>Le vallon le plus utilisé pour la traversée sur la Vallée
Etroite. Ce vallon est également accessible du refuge
Laval.</para>
</intro>
...itinéraires...
</vallon>
</guide>
```

Le guide « Itinéraires skieurs » en XML (2)

```
<itinéraire id="I15.1">
<nom>Col de Névache</nom>
<alt>2794</alt>
<cotation>**</cotation>
<num>1</num>
<para>S'élever au-dessus du refuge des Drayères en direction est. Suivre la rive droite du torrent de Brune puis s'engager sur le flanc droit du ravin des Muandes que l'on quitte vers 2500 m pour rejoindre le col situé au nord. Descente possible sur Valmeinier. <note type="prudence">Départ assez raide.</note></para>
</itinéraire>
<itinéraire id="I15.2">
<nom>Pointe de Névache</nom>
<alt>2892</alt>
<cotation>***</cotation>
<num>2</num>
<para>Du col de Névache (<renvoi cible="I15.1"/>), suivre la ligne de crête qui mène à la pointe de Névache. <note type="prudence">Attention : corniches possibles.</note> <note type="matériel">Crampons utiles au printemps.</note></para>
</itinéraire>
...
```

Un document XML en quelques mots…

- Un **document** XML est découpé en **éléments** structurés hiérarchiquement.
- Un document a un élément racine appelé **élément du document**.
- Un élément est composé :
  - d'un **nom** ;
  - d'**attributs** ;
  - d'un **contenu** formé d'**éléments** ou de **textes**.
- Un texte est une chaîne de caractères.
- Un attribut a un **nom** et une **valeur** qui est une chaîne de caractères.
- Syntaxiquement, les éléments d'un document XML sont marqués dans le document lui-même par des paires de **balises** ouvrantes et fermantes.

Définition d'un type de document (DTD)

- Les éléments qui décrivent un document peuvent être définis dans une **DTD** (Déclaration de Type de Document) ou bien dans un schéma écrit en **XML Schema**.
- Un document XML est dit **bien formé** si sa description est syntaxiquement correcte.
- Un document XML bien formé est dit **valide** si sa description est conforme à une DTD ou à un schéma.

Traitement d'un document XML

- Un document XML est destiné à être traité par un **processeur XML** qui agit pour le compte d'une application
- Un processeur XML
  - lit le document ;
  - vérifie s'il est bien formé et s'il est valide (si sa DTD ou son schéma est fourni) ;
  - le traduit sous une forme interne adaptée à son traitement par l'application.

Caractères

- Le jeu de caractères utilisable pour décrire un document XML est celui d'**Unicode** ou de l'un de ses sous-ensembles.
- Un **caractère de nom** est soit une lettre, soit un chiffre, soit un point, soit un tiret, soit un espace souligné, soit un deux-points.
- Un **caractère blanc** est soit un espace, soit une tabulation, soit un retour chariot (CR), soit un saut de ligne (LF).

Noms et lexèmes nominaux

- Un **nom** est une suite de un ou plusieurs caractères de nom dont :
  - le premier est soit une lettre, soit un espace souligné, soit un deux-points ;
  - les suivants sont des caractères de nom.
- Le caractère deux-points est réservé à la séparation d'un nom et de son préfixe (voir *Espaces de noms*).
- Par exemple :
  - `xml:lan` `extrait_de` `titre` `poeme-79`
- Un **lexème nominal** est une suite de caractères de nom.

Document XML bien formé

- Un document XML bien formé est composé :
  - d'un **prologue** facultatif ;
  - de l'élément du document ;
  - d'une suite de commentaires, ou d'instructions de traitement.
- Un document est physiquement découpé en **entités** enregistrées dans un ou plusieurs fichiers.

Elément

- Un élément est composé :
  - d'une **balise de début** qui contient le nom de l'élément et éventuellement ses attributs ;
  - d'un **contenu** ;
  - d'une **balise de fin**.
- Par exemple :
  - `<note type="prudence">Départ assez raide.</note>`
    - balise de début : `<note type="prudence">`
    - nom : `note`
    - attribut : `type="prudence"`
    - contenu : `Départ assez raide.`
    - balise de fin : `</note>`

Contenu d'un élément (1)

- Le **contenu** d'un élément est constitué d'une chaîne de caractères dans laquelle peuvent être insérés :
  - des éléments ;
  - des instructions de traitement ;
  - des commentaires.
- Ces insertions découpent le contenu d'un élément en 4 types de constituants :
  - des **textes** : les plus longues chaînes d'au moins un caractère dans lesquelles ne sont pas insérés d'éléments, d'instructions de traitement ou de commentaires ;
  - des éléments ;
  - des instructions de traitement ;
  - des commentaires.
- Les constituants du contenu d'un élément sont les **enfants** de cet élément.

Contenu d'un élément (2)

- Par exemple, le contenu de l'élément :
  - `<para>Du col de Névache (<renvoi cible="I15.1"/>), suivre la ligne de crête qui mène à la pointe de Névache.</para>`

  est constitué :
  - du texte : `Du col de Névache (`
  - de l'élément : `<renvoi cible="I15.1"/>`
  - et du texte : `), suivre la ligne de crête qui mène à la pointe de Névache.`

Types de contenu d'un élément

- **vide**
  - `<renvoi cible="I15.1"></renvoi>` ou
    `<renvoi cible="I15.1"/>`
- **composé d'éléments**
  - `<intro>`
    `<para>Vallon situé à l'est du refuge …</para>`
    `<para>Le vallon le plus utilisé pour …</para>`
    `</intro>`
- **mixte** (mélange de textes, d'éléments, d'instructions de traitement et de commentaires)
  - `<nom>Col de Névache</nom>`
  - `<para>Du col de Névache (<renvoi cible="I15.1"/>), suivre la ligne de crête qui mène à la pointe de Névache.<note type = "prudence">Attention : corniches possibles. </note><note type="matériel">Crampons utiles au printemps.</note></para>`

Textes contenant les délimiteurs < et &

- Dans le contenu d'un élément, les caractères `<` et `&` servent de délimiteurs :
  - de début d'élément, d'instruction de traitement ou de commentaires (`<`) ;
  - de début d'appel d'entité (`&`).
- Lorsque le texte à placer dans le contenu d'un élément contient l'un de ces deux caractères, il est nécessaire de marquer ces caractères pour ne pas les confondre avec ces délimiteurs.
- Deux solutions sont possibles :
  - insérer le fragment de texte contenant ces caractères dans une section CDATA ;
  - remplacer ces caractères par les appels d'entités `<` et `&`.

Section CDATA

- Une **section CDATA** a la forme suivante :
  - `<![CDATA[`*texte contenant des délimiteurs*`]]>`
- Le texte inséré peut contenir n'importe quels caractères excepté la chaîne `]]>` :
  - une section CDATA ne peut donc pas en contenir une autre.
- Par exemple, la phrase :
  - « L'expression <alt>2794</alt> est un élément XML. »

  peut être représentée par l'élément :
  - `<phrase>L'expression <![CDATA[<alt>2794</alt>]]> est un élément XML.</phrase>`

Attributs

- Un attribut est une paire nom-valeur où :
  - le nom est un nom XML ;
  - la valeur est une suite de caractères entre guillemets ou entre apostrophes.
- Par exemple :
  - `type="prudence"`
- Si une valeur d'attribut est placée entre guillemets doubles, elle peut contenir des guillemets simples et si elle est placée entre guillemets simples, elle peut contenir des guillemets doubles.
- Par exemple :
  - `select="itinéraire[cotation='****']"`
  - `select='itinéraire[cotation="****"]'`

Instructions de traitement

- Une **instruction de traitement** est une information à l'usage de l'application qui traite le document.
- Elle a la forme suivante :
  - `<?` *nom texte de l'instruction* `?>`
- Une instruction de traitement peut apparaître :
  - dans le contenu d'un élément ;
  - à la fin du document (après la balise de fin de l'élément du document).

Commentaires

- Un **commentaire** a la forme suivante :
  - `<!--`*texte du commentaire*`-->`
- Un commentaire peut contenir n'importe quel suite de caractères excepté `--`.
- Par exemple :
  - `<!-- Liste des itinéraires -->`
- Un commentaire peut apparaître partout dans un document excepté :
  - dans la balise ouvrante ou fermante d'un élément ;
  - dans une instruction de traitement ;
  - dans un commentaire.
- Un commentaire peut aussi apparaître à certains emplacements d'une DTD.

DTD

- La déclaration d'un type de document (DTD) est composée d'une suite de déclarations :
  - déclarations d'élément ;
  - déclarations des attributs d'un élément ;
  - déclarations d'instruction de traitement ;
  - déclarations d'entité.

Exemple : DTD *Guide d'itinéraires à skis* (1)

- Un guide est composé d'un titre, d'un ou plusieurs auteurs, d'un éditeur, d'une année et d'un ou plusieurs vallons.
- Un titre, un auteur, un éditeur et une année sont des textes.
- Un vallon est composé d'un nom, d'une introduction et des itinéraires que l'on peut y réaliser (un ou plusieurs itinéraires).
- Un nom est un texte.
- Une introduction est composée d'un ou plusieurs paragraphes.
- Un paragraphe est un texte dans lequel sont insérés des renvois vers d'autres itinéraires et des notes.
- Une note est un texte donnant des consignes de prudence ou recommandant l'utilisation d'un matériel spécifique (crampons, piolet...)

Exemple : DTD *Guide d'itinéraires à skis* (2)

```
<!ELEMENT guide (titre, auteur+, éditeur, année, vallon+)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteur (#PCDATA)>
<!ELEMENT éditeur (#PCDATA)>
<!ELEMENT année (#PCDATA)>
<!ELEMENT vallon (nom, intro, itinéraire+)>
<!ATTLIST vallon id ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT intro (para+)>
<!ELEMENT para (#PCDATA | renvoi | note)*>
<!ELEMENT renvoi EMPTY>
<!ATTLIST renvoi cible IDREF #REQUIRED>
<!ELEMENT note (#PCDATA)>
<!ATTLIST note type (prudence | matériel) "prudence">
<!ELEMENT itinéraire (nom, alt, cotation, num, para+)>
<!ATTLIST itinéraire id ID #REQUIRED>
<!ELEMENT alt (#PCDATA)>
<!ELEMENT cotation (#PCDATA)>
<!ELEMENT num (#PCDATA)>
```

Déclaration d'un élément

- Un élément est défini par la déclaration :
  - `<!ELEMENT` *nom modèle de contenu*`>`
- Un élément mixte pouvant contenir dans un ordre quelconque des éléments n_1, …, n_k a pour modèle de contenu :
  - `(#PCDATA |` n_1 `|` … `|` n_k`)*`
- Un élément composé d'une suite d'éléments n_1, …, n_k a pour modèle de contenu l'expression régulière construite sur le vocabulaire $\{n_1, …, n_k\}$ à l'aide des opérateurs :
  - `,` (infixe) : concaténation
  - `* +` (suffixes) : 0 ou plusieurs répétitions et 1 ou plusieurs répétitions
  - `?` (suffixe) : optionalité
- Un élément vide a pour modèle de contenu `EMPTY`.
- Un élément de contenu quelconque a pour modèle de contenu `ANY`.

Déclaration des attributs d'un élément (1)

- A chaque type d'élément est attaché un ensemble d'attributs.
- Une définition d'attributs a la forme suivante :
  - `<!ATTLIST` *nom-élément*
    nom-attribut type déclaration-de-défaut
    …
    nom-attribut type déclaration-de-défaut `>`

  où :
  - le type est celui des valeurs de l'attribut ;
  - la déclaration de défaut spécifie si la valeur de l'attribut doit être ou non présente dans le document et fournit éventuellement une valeur par défaut.
- Les noms d'attributs sont locaux à chaque type d'élément :
  - deux éléments de types différents peuvent avoir des attributs de même nom.

Déclaration des attributs d'un élément (2)

- Le type de valeur peut être :
  - `CDATA` : texte
  - `ID` : lexème nominal identifiant l'élément dans le document
  - `IDREF` ou `IDREFS` : lexème nominal ou suite de lexèmes nominaux référençant des éléments du document
  - `NMTOKEN` ou `NMTOKENS` : lexème nominal ou suite de lexèmes nominaux
  - (*nom*$_1$ | … | *nom*$_n$) : un lexème nominal parmi *nom*$_1$, ..., *nom*$_n$
  - `ENTITY`, `ENTITIES` : nom ou suite de noms d'entités déclarées dans la DTD
  - `NOTATION` : nom d'une notation qui identifie une donnée non XML

Déclaration des attributs d'un élément (3)

- La déclaration de défaut peut être :
  - `#REQUIRED` : l'attribut doit être présent dans la balise de l'élément
  - `#IMPLIED` : l'attribut est facultatif
  - *valeur* : valeur à affecter à l'attribut s'il est absent de la balise de l'élément (valeur par défaut)
  - `#FIXED` *valeur* : valeur que doit avoir l'attribut s'il est présent dans la balise de l'élément ou qui lui sera affectée s'il est absent de cette balise
- Les déclarations de défaut sont prises en compte par le processeur XML afin de compléter le document analysé.

Liens internes

- Les attributs de type `ID` permettent d'identifier des éléments dans un document et les attributs de type `IDREF` permettent d'y faire référence depuis d'autres éléments.
- Par exemple :
  - les lexèmes nominaux `I15.1` et `I15.2` identifient les itinéraires n° 1 et n° 2 du Vallon des Muandes (le 15e) dans le document *Itinéraires skieurs dans la Vallée de la Clarée* ;
  - l'élément `<renvoi cible="I15.1"/>` renvoie à l'itinéraire n° 1 de ce même vallon.

Entités

- Un document XML est physiquement découpé en **entités**.
- Une entité est un fragment nommé de document.
- On distingue :
  - les **entités prédéfinies** : caractères réservés de XML ;
  - les **entités caractère** : caractères UNICODE désignés par leur code numérique ;
  - les **entités générales** : fragments nommés de l'élément du document ;
  - les **entités paramètres** : fragments nommés de DTD.
- Les entités générales et paramètres sont définies dans la DTD.

Réalisation physique du document
Itinéraires skieurs dans la Vallée de la Clarée

```
<?xml version="1.0" ?>
<!DOCTYPE guide SYSTEM "guide.dtd"
[
...
<!ENTITY vallon-muandes
                    SYSTEM "muandes.xml">
...
]>
<guide>
<titre>Itinéraires skieurs dans la
Vallée de la Clarée</titre>
<auteur>Jean-Gabriel Ravary</auteur>
<editeur>Le Polygraphe</editeur>
<année>1991</année>
...
&vallon-muandes;
...
</guide>
```

```
<?xml version="1.0" ?>
<!ELEMENT guide (titre,
auteur+, editeur, année,
vallon+)>
<!ELEMENT titre (#PCDATA)>
...
```

```
<?xml version="1.0" ?>
<vallon>
<nom>Vallon des Muandes</nom>
<intro>
<para>Vallon situé à l'est du
refuge des Drayères.</para>
<para>Le vallon le plus utilisé
pour la traversée sur la Vallée
Etroite. ...</para>
</intro>
...
</vallon>
```

Appel d'entité

- Un **appel d'entité** a l'une des 3 formes suivantes :
  - &#*code*; appel de caractère (celui de code *code*)
  - &*nom*; appel d'entité générale ou prédéfinie
  - %*nom*; appel d'entité paramètre
- Un appel d'entité prédéfinie, caractère ou générale qui ne peut apparaitre que dans la valeur d'un attribut ou dans le contenu d'un élément.
- Un appel d'entité paramètre qui ne peut apparaître que dans la DTD.
- Le traitement d'un appel d'entité par un processeur XML consiste à remplacer cet appel par le fragment de document désigné par l'entité appelée.

Entités prédéfinies

- Les caractères < > & ' " qui sont des délimiteurs XML peuvent être remplacés dans un texte par une référence à une entité prédéfinie. Ces entités sont les suivantes :
  - `<` référence au caractère <
  - `>` référence au caractère >
  - `&` référence au caractère &
  - `&apos;` référence au caractère '
  - `&quot;` référence au caractère "
- Par exemple, la phrase :
  - « L'expression <alt>2794</alt> est un élément XML. »

  peut être représentée par l'élément suivant :
  - `<phrase>L'expression <alt>2794</alt> est un élément XML.</phrase>`

Entités caractères

- Un caractère non disponible sur la station de travail peut être représenté par son code Unicode en décimal ou en hexadécimal, sous la forme d'un appel d'entité :
  - `&#`*code décimal*`;`
  - `&#x`*code hexadécimal*`;`
- Par exemple :
  - `&#38;` référence au caractère `&`
  - `&#x03A6;` référence à la lettre grecque `Φ`

Entités générales

- Déclaration d'une entité générale :
  - **interne** (enregistrée dans sa déclaration)
    - `<!ENTITY` *nom* `"`*entité*`">`
  - **externe** (enregistrée dans un fichier externe à celui de sa déclaration)
    - `<!ENTITY` *nom* `SYSTEM "`*nom du fichier contenant l'entité*`">`
  - Par exemple :
    - `<!ENTITY ref "refuge">`
    - `<!ENTITY vallon-muandes SYSTEM "mon_site/muandes.xml">`
- Appel d'entité générale :
  - `&`*nom de l'entité*`;`
  - Par exemple :
    - `<para>S'élever au-dessus du &ref; des Drayères …</para>`

    est équivalent à :
    - `<para>S'élever au-dessus du refuge des Drayères …</para>`

Entités paramètres

- Déclaration d'une entité paramètre :
  - interne (enregistrée dans sa déclaration)
    - `<!ENTITY % nom "entité">`
  - externe (enregistrée dans un fichier externe à celui de sa déclaration)
    - `<!ENTITY % nom SYSTEM "nom du fichier contenant l'entité">`
  - Par exemple :
    - `<!ENTITY % identificateur "ID #REQUIRED">`
- Appel d'entité paramètre :
  - `%nom;`
  - Par exemple :
    - `<!ATTLIST renvoi cible %identificateur;>`
  - au lieu de :
    - `<!ATTLIST renvoi cible ID #REQUIRED>`

Organisation d'un document XML valide

- Un document XML valide est composé d'une **entité document** (sans nom) et d'un ensemble d'entités externes.
- L'entité document est composé d'un **prologue** et de l'**élément du document**.
- Le prologue est composé d'une **déclaration XML** et d'une DTD.
- La déclaration XML indique : la version de XML, le jeu de caractères et l'éclatement ou non du document en plusieurs **entités externes**.
- La DTD est constituée d'une **partie interne** placée dans l'entité document et d'une **partie externe**, enregistrée dans un fichier à part dont le nom est déclaré dans l'entité document.
- La partie interne de la DTD, l'élément du document et les entités externes peuvent appeler des entités externes. Ces appels doivent être non récursifs et non circulaires.

Organisation d'un document XML bien formé

- L'organisation d'un document bien formé est similaire à celle d'un document à l'exception de la DTD qui est :
  - soit absente ;
  - soit présente mais ne contient que des déclarations d'entités générales.

Document XML monofichier

- `<?xml version="1.0" encoding="…" standalone="yes" ?>`
  `<!DOCTYPE` *nom* `[`*déclarations*`]>`
  `<`*nom*`>`
  …
  `</`*nom*`>`
  où :
  - l'attribut `standalone="yes"` indique que le document est contenu en entier dans le fichier ;
  - le nom de l'élément du document doit être identique à celui de la DTD.

Document XML multifichiers

- `<?xml version="1.0" encoding="…" standalone= "no" ?>`
  `<!DOCTYPE` *nom* `SYSTEM` *nom fichier* `[`*partie interne de la DTD*`]>`
  élément du document

  où :
  - *nom fichier* est le nom du fichier contenant la partie externe de la DTD ;
  - l'attribut `standalone="no"` indique qu'il est fait appel à des entités externes soit dans la partie interne de la DTD, soit dans l'élément du document ;
  - le nom de l'élément du document doit être celui de la DTD ;
  - une entité externe peut débuter (et c'est conseillé) par une déclaration XML sans attribut `standalone`.

Codage des caractères

Unicode et UCS

- La norme ISO 10646 en accord avec l'Unicode définit un jeu de caractères universel : l'UCS (Universal Character Set) qui permet de représenter les caractères de toutes les langues actuelles mais aussi des langues anciennes, graphiques...
- Chaque caractère UCS est identifié par un code qui est un nombre représenté sur 4 octets (2^{32} - 1 positions).
- Les 65 536 premières positions de l'UCS (c.-à-d. les deux octets de poids faible) forment le BMP (« Basic Multilingual Plane ») et codent les jeux de caractères les plus courants (latin, grec, arabe...). D'où deux codages :
  - UCS-4 : totalité de l'UCS ;
  - UCS-2 : BMP.

Codages de transformation

- Plusieurs codages de transformation ont été définis, notamment :
  - l'UTF-8 qui permet de coder les caractères de l'UCS en longueur variable en codant sur un octet les caractères ASCII qui sont les plus fréquents ;
  - l'UTF-16 qui permet d'inclure des caractères de l'UCS-4 dans une chaîne codée en UCS-2.

Déclaration du codage

- Toutes les applications XML doivent accepter les codages UTF-8 et UTF-16.
- D'autres codages peuvent être acceptés, tels que le codage ISO-8859-1 (l'ISO-Latin).
- Le codage des caractères d'une entité doit être déclaré dans la déclaration XML de cette entité, comme valeur de l'attribut `encoding`. S'il ne l'est pas, l'application considérera être en présence d'un codage UTF-8.

Espaces de noms

Espace de noms

- L'importation, dans un document, d'éléments ou d'attributs contenus dans des entités externes peut entraîner des conflits de noms.
- Ces conflits peuvent être évités en définissant des **espaces de noms** identifiés de façon unique et en associant à un nom l'espace de noms dont il provient.

Nom développé, nom local, nom qualifié

- Un espace de noms est identifié par une URI (Uniform Resource Identifier).
- Tout nom *n* d'un espace de nom identifié par une URI *u* est désigné de façon unique par son **nom développé**, qui est la paire (*u*, *n*) :
  - *u* est le nom de l'espace de noms ;
  - *n* est le **nom local**.
- A l'intérieur d'un élément XML :
  - un préfixe est associé à chaque espace de nom ;
  - un nom d'un espace de noms est écrit sous forme d'un **nom qualifié** : *préfixe*:*nom local*.

Déclaration d'un espace de noms

- Pour utiliser les noms d'un espace de noms à l'intérieur d'un élément, on doit déclarer :
  - l'URI de cet espace ;
  - un préfixe qui est un nom XML n'utilisant pas le caractère deux-points (`:`).
- Pour cela, on insère dans la balise ouvrante de l'élément l'attribut :
  - `xmlns:`*préfixe*`="`*URI de l'espace de noms*`"`
- On peut déclarer un espace de noms par défaut par l'attribut :
  - `xmlns="`*URI de l'espace de noms*`"`

  ou l'annihiler par la déclaration :
  - `xmlns=""`

Visibilité d'une déclaration d'espace de noms

- La déclaration d'un espace de noms muni d'un préfixe est visible dans l'élément la contenant (son nom et ses attributs y compris) et dans tous ses descendants excepté ceux pour lesquels un nouvel espace de noms de même préfixe est déclaré.
- La déclaration d'un espace de noms par défaut est visible dans l'élément la contenant (son nom et ses attributs y compris) et dans tous ses descendants excepté ceux pour lesquels cette déclaration est annihilée ou ceux pour lesquels un nouvel espace de noms par défaut est déclaré.

Association nom-espace de nom

- Tout nom d'élément ou tout nom d'attribut qui n'est pas une déclaration d'espace de noms est un nom qualifié ayant l'une des deux formes suivantes :
  - *préfixe*:*nom-local*
  - *nom-local*
- Un nom qualifié préfixé appartient à l'espace de noms associé à ce préfixe déclaré dans le plus imbriqué des éléments contenant ce nom.
- Un nom qualifié non préfixé :
  - appartient à l'espace de noms par défaut associé à ce préfixe déclaré dans le plus imbriqué des éléments contenant ce nom, s'il en existe un ;
  - n'appartient pas à un espace de noms s'il n'existe pas de déclaration d'espace de noms par défaut dans les éléments contenant ce nom.

Exemple de déclaration d'un espace de noms (1)

- On suppose que :
  - les noms des éléments décrivant le titre, les auteurs l'éditeur et l'année d'édition appartiennent à l'espace de noms :
    - `http://jlm.univ-tln.fr/livre`

    de préfixe `livre` ;
  - tous les autres noms d'éléments ou d'attributs n'appartiennent pas à un espace de noms.

Exemple de déclaration d'un espace de noms (2)

```
<guide xmlns:livre="http://jlm.univ-tln.fr/livre">
<livre:titre>Itinéraires skieurs...</livre:titre>
<livre:auteur>Jean-Gabriel Ravary</livre:auteur>
...
<vallon>
<nom>Vallon des Muandes</nom>
...
<itinéraire id="I15.1">
<nom>Col de Névache</nom>
<alt>2794</guide:alt>
<cotation>**</cotation>
<num>1</num>
<para>
...<note type="prudence">Départ assez raide.</note>...
</para>
</itinéraire>
...
</guide>
```

Exemple de déclaration d'un espace de noms par défaut (1)

- On suppose maintenant que tous les noms d'éléments ou d'attributs qui n'appartiennent pas à l'espace de noms :
  - `http://jlm.univ-tln.fr/livre`

  appartiennent à l'espace de noms :
  - `http://jlm.univ-tln.fr/clarée`

  qui est l'espace de noms par défaut.

Exemple de déclaration d'un espace de noms par défaut (2)

```
<guide xmlns:livre="http://jlm.univ-tln.fr/livre"
       xmlns="http://jlm.univ-tln.fr/clarée">
<livre:titre>Itinéraires skieurs...</livre:titre>
<livre:auteur>Jean-Gabriel Ravary</livre:auteur>
...
<vallon>
<nom>Vallon des Muandes</nom>
...
<itinéraire id="I15.1">
<nom>Col de Névache</nom>
<alt>2794</guide:alt>
<cotation>**</cotation>
<num>1</num>
<para>
...<note type="prudence">Départ assez raide.</note>...
</para>
</itinéraire>
...
</guide>
```

Préfixes réservés et préfixes usuels

- Le préfixe `xml` est lié à l'espace de noms `http://www.w3.org/XML/1998/namespace`. Il peut mais ne doit pas être déclaré et ne doit pas être lié à un autre espace de noms. Il ne doit pas y avoir d'autres préfixes liés à cet espace de noms et il ne doit pas être déclaré comme espace de noms par défaut.
- Le préfixe `xmlns` est lié à l'espace de noms `http://www.w3.org/2000/xmlns/`. Il ne doit pas être déclaré. Il ne doit pas y avoir d'autres préfixes liés à cet espace de noms et il ne doit pas être déclaré comme espace de noms par défaut. Les noms d'éléments ne doivent pas avoir le préfixe `xmlns`.
- Tous les préfixes commençant par les lettres `x`, `m` et `l` sont réservés.
- Chaque langage de définition ou de manipulation de données XML possède son propre espace de noms ainsi que son propre préfixe par défaut qu'il est recommandé d'utiliser pour des raisons de lisibilité.

XML Schema
un langage de typage de données XML

http://www.w3.org/TR/xmlschema-0/

http://www.w3.org/TR/xmlschema-1/

http://www.w3.org/TR/xmlschema-2/

Introduction

- **XML Schema** est un langage qui permet de définir des classes de documents XML de façon beaucoup plus fine qu'une DTD.
- Un **schéma** XML Schema est un document XML dont les éléments définissent des types ou déclarent des éléments ou des attributs.
- Le langage XML Schema est relativement complexe comme en témoigne le volume des deux recommandations qui lui sont consacrées :
  - nous n'en ferons donc qu'une présentation simplifiée et au travers d'exemples.
- L'espace de noms associé à XML Schema est :
  - `http://www.w3.org/2001/XMLSchema`

  dont le préfixe usuel est `xsd`.

Types

- Un type peut être :
  - **simple** ou **complexe**
  - **prédéfini** ou **défini** (par l'utilisateur)
  - **anonyme** ou **nommé**
  - **dérivé** par **restriction** ou **extension** de types existants.

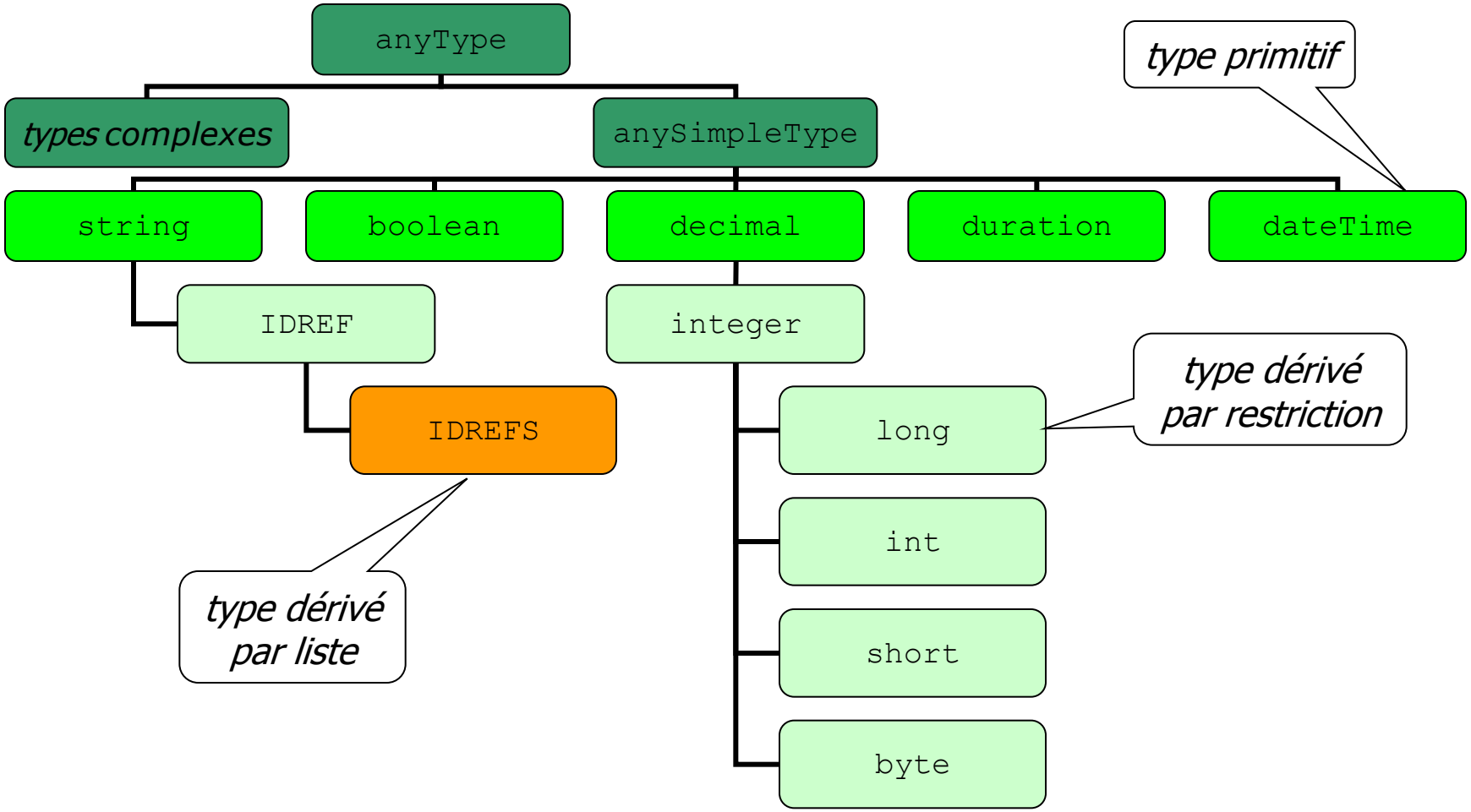
Types simples

- Un type simple est défini par :
  - un **ensemble de valeurs** ;
  - une **représentation lexicale** ;
  - un ensemble de **facettes** caractérisant l'ensemble de valeurs.
- Un type simple peut être :
  - un type **atomique** ;
  - un type **liste** ;
  - un type **union**.

Types atomiques, types liste et types union

- Un type atomique est un ensemble de valeurs indécomposables dites **valeurs atomiques**.
- Un type liste est un ensemble de séquences de longueur finie de valeurs atomiques.
- Un type union est un ensemble de valeurs atomiques appartenant à plusieurs types atomiques.

Hiérarchie des types prédéfinis : un extrait



Facettes des types primitifs

	string	boolean	decimal	float	duration	dateTime
length	x				x	x
minLength	x					
maxLength	x					
pattern	x	x	x	x	x	x
enumeration	x		x	x	x	x
whiteSpace	x	x	x	x	x	x
minInclusive			x	x	x	x
minExclusive			x	x	x	x
maxInclusive			x	x	x	x
maxExclusives			x	x	x	x
totalDigits			x			
fractionDigits			x			

Types prédéfinis temporels

`duration`	1 an, 2 mois, 3 jours, 10 heures, 30 minutes : `P1Y2M3DT10H30M`
`dateTime`	13h20, le 31 mai 1999 (UTC) : `1999-05-31T13:20` 13h20, le 31 mai 1999 (UTC + 5h) : `1999-05-31T13:20:00-05:00`
`time`	13h20 du jour courant (UTC) : `13:20`
`date`	31 mai 1999 : `1999-05-31`
`gYear`	1999 : `1999`
`gYearMonth`	mai 1999 : `1999-05`
`gMonth`	mois de mai de l'année courante : `--05--`
`gMonthDay`	31 mai de l'année courante : `--05-31`
`gDay`	31 du mois courant : `--31`

Types primitifs : `string`, `boolean`, `float` et `decimal`

`string`	Ensemble des séquences de caractères de longueur finie	
`boolean`	Ensemble de valeurs = {vrai, faux, 1, 0}	
`float`	Ensemble de valeurs = $m \times 2^e$ où $\|m\| \leq 2^{24}$ et $-149 \leq e \leq 104$ + zéro positif et négatif infinité positive et négative pas un nombre	-1E4 1267.43233E12 12.78e-2 12 0 -0 INF –INF NaN
`decimal`	Nombre décimaux de précision arbitraire. Ensemble de valeurs = $i \times 10^n$ où i et n sont des entiers et $n \geq 0$	-1.23 12678967.543233 +100000.00 210

Types prédéfinis dérivés du type `decimal` : un extrait

`integer`	Dérivé du type `decimal` par `fractionDigits` = 0
`long`	Dérivé du type `integer` par `minInclusive` = -9223372036854775808 et `maxInclusive` = 9223372036854775807
`int`	Dérivé du type `integer` par `minInclusive` = -2147483648 et `maxInclusive` = 2147483647
`short`	Dérivé du type `int` par `minInclusive` = -32768 et `maxInclusive` = 32767
`byte`	Dérivé du type `short` par `minInclusive` = -128 et `maxInclusive` = 127

Exemple de type simple dérivé par restriction

- Ensemble des altitudes des points de la terre (le plus haut sommet a une altitude un peu inférieure à 8850m).
  - ```
    <xsd:simpleType name="altitude">
    <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="8850"/>
    </xsd:restriction>
    </xsd:simpleType>
    ```
    ```

Exemple de type simple dérivé par restriction

- Liste des noms de pays :
  - ```
    <xsd:simpleType name="nom-pays">
    <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Algérie"/>
    <xsd:enumeration value="Australie"/>
    <xsd:enumeration value="Belgique"/>
    <xsd:enumeration value="France"/>
    ...
    </xsd:restriction>
    </xsd:simpleType>
    ```
    ```

Exemple de type simple dérivé par restriction

- Liste des codes des pays :
  - ```
    <xsd:simpleType name="code-pays">
    <xsd:restriction base="xsd:string">
    <xsd:enumeration value="DZ"/>
    <xsd:enumeration value="AUS"/>
    <xsd:enumeration value="B"/>
    <xsd:enumeration value="F"/>
    ...
    </xsd:restriction>
    </xsd:simpleType>
    ```
    ```

Exemple de type liste

- Liste de pays :
  - ```
    <xsd:simpleType name="liste-pays">
    <xsd:list itemType="nom-pays"/>
    </xsd:simpleType>
    ```
- Elément dont le contenu est conforme à ce type :
  - ```
    <pays-himalayen>Chine Inde Népal Pakistan</pays-himalayen>
    ```

Exemple de type union

- Ensemble des noms ou des codes des pays :
  - <xsd:simpleType name="nom-ou-code-pays">
    <xsd:union memberTypes="nom-pays code-pays"/>
    </xsd:simpleType>
- Eléments dont le contenu est conforme à ce type :
  - <pays>AUS</pays>
  - <pays>Belgique</pays>

Types complexes

- Un type complexe définit un ensemble de compositions d'éléments et/ou d'attributs
- Une composition est réalisée à l'aide de 3 constructeurs :
  - `sequence` : séquence d'éléments ;
  - `choice` : un élément parmi une liste d'éléments possibles ;
  - `all` : groupe d'éléments non ordonnés dans lequel il n'existe pas 2 élément de même nom.

Déclaration d'un élément ou d'un attribut

- Rappelons qu'un schéma est un document XML.
- Un élément ou un attribut est déclaré sous la forme d'un élément du schéma.
- Un élément ou un attribut peut être déclaré :
  - soit au niveau **global** comme enfant de l'élément du schéma ;
  - soit au niveau **local** dans le contenu de son élément parent.
- Un **type d'élément** définit comment sont composés les éléments de ce type : leurs attributs et leur contenu.
- Un **type d'attribut** définit l'ensemble des valeurs possibles pour les attributs de ce type.

Déclaration d'un élément

- Un élément est déclaré sous la forme d'un élément `xsd:element` :
  - dont les attributs peuvent être :
    - `name` : nom de l'élément
    - `type` : type de l'élément
    - `ref` : référence à un nom d'élément déclaré au niveau global
    - `minOccurs` : nombre minimum d'occurrences de l'élément (1 par défaut)
    - `maxOccurs` : nombre maximum d'occurrences de l'élément (1 par défaut)

    …
  - qui peut avoir un contenu qui est la définition du type de cet élément, si ce type n'a pas été spécifié par l'attribut `type` ou si cet élément n'est pas déclaré au niveau global et référencé par l'attribut `ref`.

Déclaration d'un attribut

- Un attribut est déclaré sous la forme d'un élément `xsd:attribute` :
  - dont les attributs peuvent être :
    - `name` : nom de l'attribut
    - `ref` : référence à un nom d'attribut déclaré au niveau global (réutilisation)
    - `type` : type des valeurs de l'attribut
    - `use` : required, optional (par défaut)...
    - `default` : valeur par défaut
    - `fixed` : valeur fixée

    ...
  - qui peut avoir un contenu qui est la définition du type de cet attribut, si ce type n'a pas été spécifié par l'attribut `type` ou si cet attribut n'est pas déclaré au niveau global et référencé par l'attribut `ref`.

Exemple de déclaration d'un élément de type simple

- L'élément :
  - `<pays>France</pays>`

  est conforme à la déclaration :
  - `<xsd:element name="pays" type="xsd:nom-pays"/>`

Exemple de déclaration d'un élément de type complexe (composé d'éléments)

- L'élément :
  - <livre>
    <titre>Programmer en XML</titre>
    <année>2004</année>
    </livre>

  est conforme à la déclaration :
  - <xsd:element name="livre">
    <xsd:complexType>
    <xsd:sequence>
    <xsd:element name="titre" type="xsd:string"/>
    <xsd:element name="année" type="xsd:gYear"/>
    </xsd:sequence>
    </xsd:complexType>
    </xsd:element>

Exemple de déclaration d'un élément composé d'éléments déclarés au niveau global

- Si les éléments `titre` et `année` ont été déclarés au niveau global par :
  - `<xsd:element name="titre" type="xsd:string"/>`
  - `<xsd:element name="année" type="xsd:gYear"/>`
- L'élément `livre` aurait pu être déclaré :
  -
```
<xsd:element name="livre">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="titre"/>
<xsd:element ref="année"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Exemple de déclaration d'un élément de type complexe (avec attributs et composé d'éléments)

- L'élément :
  - ```
    <livre édition="2">
    <titre>Programmer en XML</titre>
    <année>2004</année>
    </livre>
    ```

  est conforme à la déclaration :

  - ```
    <xsd:element name="livre">
    <xsd:complexType name="livre">
    <xsd:sequence>
    <xsd:element name="titre" type="xsd:string"/>
    <xsd:element name="année" type="xsd:gYear"/>
    </xsd:sequence>
    <xsd:attribute name="édition"
                   type="xsd:positiveInteger"/>
    </xsd:complexType>
    </xsd:element>
    ```

Exemple de déclaration d'un élément de type complexe (à contenu mixte)

- L'élément :
  - `<titre>Les <sigle>BD</sigle> relationnelles</titre>`

  est conforme à la déclaration :
  - ```
    <xsd:element name="titre">
    <xsd:complexType mixed="true">
    <xsd:sequence>
    <xsd:element name="sigle" type="xsd:string"/>
    </xsd:sequence>
    </xsd:complexType>
    </xsd:element>
    ```
    ```

Exemple de déclaration d'un élément de type complexe (avec attributs et à contenu vide)

- L'élément :
  - `<prix monnaie="euros" valeur="25.5">`

  est conforme à la déclaration :
  - ```
    <xsd:element name="prix">
    <xsd:complexType>
    <xsd:attribute name="monnaie" type="xsd:string"/>
    <xsd:attribute name="valeur" type="xsd:decimal"/>
    </xsd:complexType>
    </xsd:element>
    ```
    ```

Exemple de nommage d'un type complexe

- On peut nommer le type définissant un prix en euros ou en dollars :
  - ```
    <xsd:complexType name="prix-international">
    <xsd:attribute name="monnaie" type="xsd:string"/>
    <xsd:attribute name="valeur" type="xsd:decimal"/>
    </xsd:complexType>
    ```

  et y faire référence dans la déclaration de l'élément prix :
  - ```
    <xsd:element name="prix"
                 type="prix-international"/>
    ```

Exemple de déclaration d'un élément de type complexe (à type de contenu choisi parmi plusieurs)

- Les éléments :
  - `<prix><euros>25.5</euros></prix>`
  - `<prix><dollars>28.75</dollars></prix>`

  sont conformes à la déclaration :
  - ```
    <xsd:element name="prix">
    <xsd:complexType>
    <xsd:choice>
    <xsd:element name="euros" type="xsd:decimal"/>
    <xsd:element name="dollars" type="xsd:decimal"/>
    </xsd:choice>
    </xsd:complexType>
    </xsd:element>
    ```
    ```

Exemple de nommage de groupes d'éléments

- On peut nommer des groupes d'éléments ou d'attributs.
- La séquence d'éléments :
  - `<prénom>Jean</prénom><nom>Dupont</nom>`

  est conforme à la déclaration :
  - ```
    <xsd:group name="prénom-nom">
    <xsd:sequence>
    <xsd:element name="prénom" type="xsd:string"/>
    <xsd:element name="nom" type="xsd:string"/>
    </xsd:sequence>
    </xsd:group>
    ```
- On peut faire référence à ce groupe dans la définition d'un type personne :
  - ```
    <xsd:complexType name="personne">
    <xsd:group ref="prénom-nom"/>
    </xsd:complexType>
    ```

Exemple de type complexe dérivé par extension

- A partir du type `personne`, on peut définir le type `chercheur` en ajoutant au prénom et au nom, le laboratoire et le pays :
  - ```
    <xsd:complexType name="chercheur">
    <xsd:complexContent>
    <xsd:extension base="personne">
    <xsd:sequence>
    <xsd:element name="laboratoire" type="xsd:string"/>
    <xsd:element ref="pays">
    </xsd:sequence>
    </xsd:extension>
    </complexContent>
    </xsd:complexType>
    ```
    ```

Exemple de schéma XML (1)

- Construction d'un schéma XML pour des document tels que :
  - <livres>
    …
    <livre édition="1">
    <titre>Le langage XML</titre>
    <auteur>
    <prénom>Jean</prénom><nom>Dupont</nom>
    <laboratoire>SIS</laboratoire><pays>France</pays>
    </auteur>
    <auteur>
    <prénom>Pierre</prénom><nom>Durand</nom>
    <laboratoire>LSIS</laboratoire><pays>France</pays>
    </auteur>
    <année>2004</année>
    <prix monnaie="euros" valeur="25.5"/>
    </livre>
    …
    </livres>

Exemple de schéma XML (2)

conformes à la DTD :

```
<!ELEMENT livres (livre*)>
<!ELEMENT livre (titre, auteur+, année, prix)>
<!ATTLIST livre édition CDATA #REQUIRED>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteur (prénom, nom, laboratoire, pays)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prénom (#PCDATA)>
<!ELEMENT laboratoire (#PCDATA)>
<!ELEMENT année (#PCDATA)>
<!ELEMENT prix EMPTY>
<!ATTLIST prix monnaie CDATA #REQUIRED
               valeur CDATA #REQUIRED>
<!ELEMENT pays (#PCDATA)>
```

Exemple de schéma XML (3)

```
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:group name="prénom-nom">
<xsd:sequence>
<xsd:element name="prénom" type="xsd:string"/>
<xsd:element name="nom" type="xsd:string"/>
</xsd:sequence>
</xsd:group>
<xsd:complexType name="personne">
<xsd:group ref="prénom-nom"/>
</xsd:complexType>
...
```

Exemple de schéma XML (4)

```
…
<xsd:simpleType name="nom-pays">
<xsd:restriction base="xsd:string">
<xsd:enumeration value="Algérie"/>
<xsd:enumeration value="Australie"/>
<xsd:enumeration value="Belgique"/>
<xsd:enumeration value="France"/>
…
</xsd:restriction>
</xsd:simpleType>
<xsd:element name="pays" type="nom-pays"/>
…
```

Exemple de schéma XML (5)

```
...
<xsd:complexType name="chercheur">
<xsd:complexContent>
<xsd:extension base="personne">
<xsd:sequence>
<xsd:element name="laboratoire" type="xsd:string"/>
<xsd:element ref="pays"/>
</xsd:sequence>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="prix-international">
<xsd:attribute name="monnaie" type="xsd:string"/>
<xsd:attribute name="valeur" type="xsd:decimal"/>
</xsd:complexType>
...
```

Exemple de schéma XML (6)

```
...
<xsd:element name="livre">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="titre" type="xsd:string"/>
<xsd:element name="auteur"
             type="chercheur"
             maxOccurs="unbounded"/>
<xsd:element name="année" type="xsd:gYear"/>
<xsd:element name="prix" type="prix-international"/>
</xsd:sequence>
<xsd:attribute name="edition"
               type="xsd:positiveInteger"/>
</xsd:complexType>
</xsd:element>
...
```

Exemple de schéma XML (7)

```
...
<xsd:element name="livres">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="livre" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

XDM
un modèle de données XML

http://www.w3.org/TR/xpath-datamodel/

Introduction

- Le modèle XDM (XML Data Model) est commun à plusieurs langages de manipulation de données XML :
  - XPath 2.0
  - XQuery 1.0
  - XSLT 2.0

  ...

Séquences

- Tout instance du modèle est une **séquence**.
- Une séquence est une collection ordonnée de 0, 1 ou plusieurs **items**.
- Un item est soit une **valeur atomique**, soit un **nœud**.
- Une séquence de 0 item est appelée **séquence vide**.
- Une séquence de 1 item est appelée **singleton**.
- Il y a équivalence entre un item et une singleton :
  - 12, peut être indifféremment traité comme l'item 12 ou comme la séquence constituée de l'item 12.
- Les séquences sont plates : une séquence ne peut pas être un élément d'une séquence.
  - 3, (5, 7), 9 ≡ 3, 5, 7, 9

Valeurs atomiques

- Une valeur atomique est une instance d'un type atomique XML Schema.

Nœuds

- Il y a 7 sortes de nœuds :
  - **document** ;
  - **élément** ;
  - **texte** ;
  - **attribut** ;
  - **espace de noms** ;
  - **commentaire** ;
  - **instruction de traitement**.

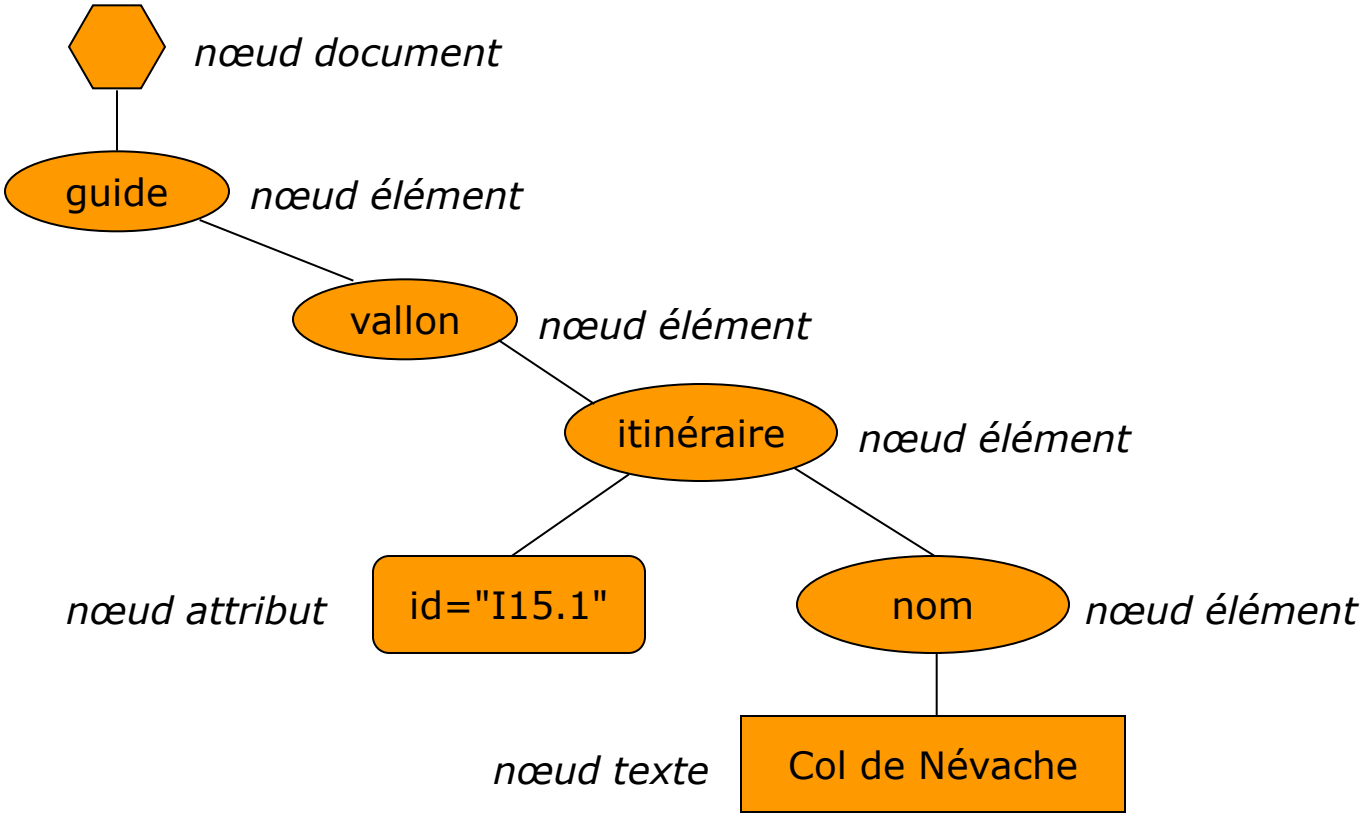
Identité d'un nœud

- Tout nœud a un **identificateur unique**.
- Chaque nœud dans une instance du modèle est unique : il est identique à lui-même et non identique à tout autre nœud.
- Il ne faut pas confondre l'identificateur d'un nœud élément et une valeur d'attribut de type ID associé à cet élément.

Arbres

- Les nœuds forment des **arbres**.
- Chaque nœud appartient à un et un seul arbre et chaque arbre a un et un seul nœud racine.
- Un arbre dont le nœud racine est un nœud document est appelé **document**.
- Un arbre dont le nœud racine n'est pas un nœud document est appelé **fragment**.

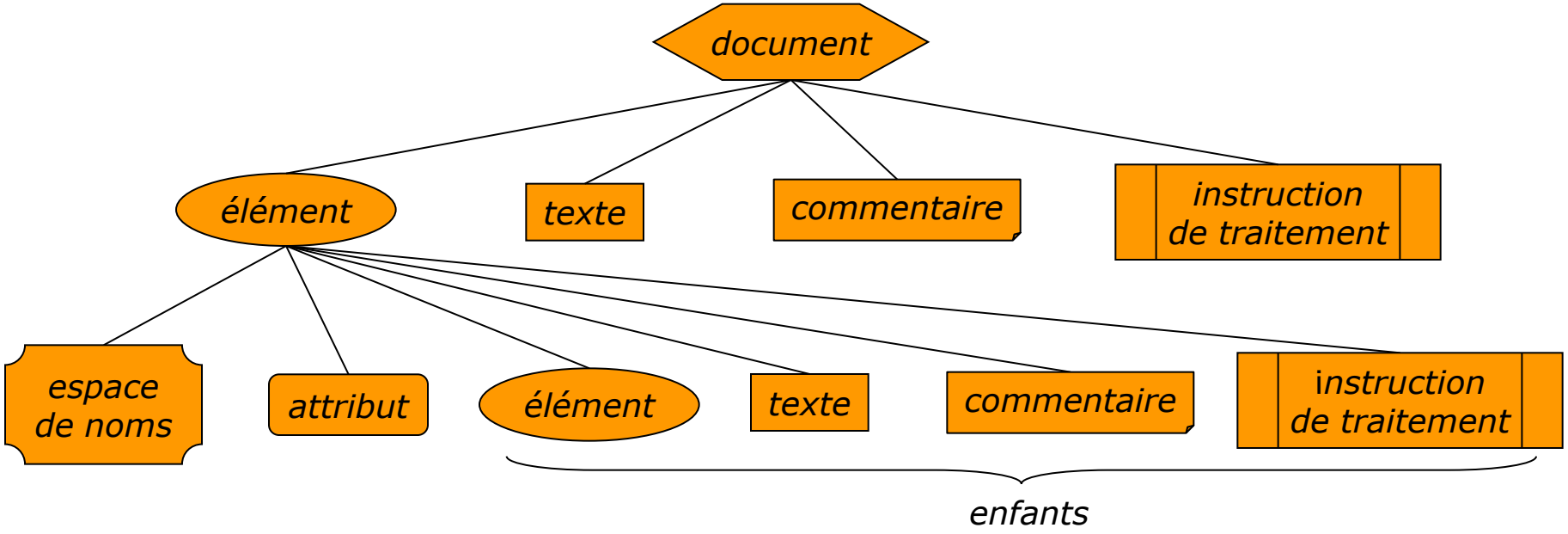
Extrait de l'arbre du document
Itinéraires skieurs dans la Vallée de la Clarée

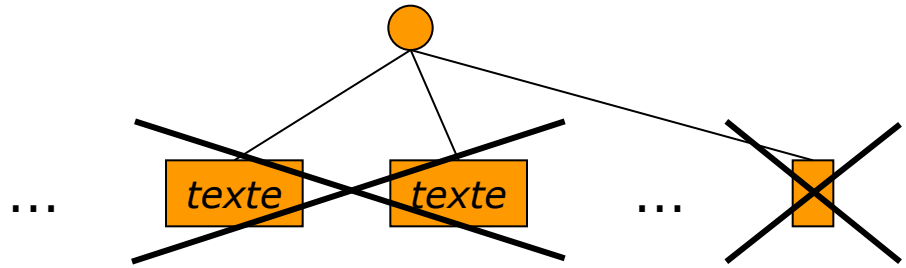


Contraintes sur les nœuds (1)

- Le nœud document peut avoir des nœuds fils qui peuvent être des nœuds commentaire, élément, instruction de traitement ou texte.
- Un nœud élément peut avoir des nœuds fils qui peuvent être des nœuds attribut, commentaire, élément, espace de noms, instruction de traitement ou texte.
- Les nœuds fils d'un nœud document ou élément qui sont des nœuds élément, texte, commentaire ou instruction de traitement sont appelés les **enfants** de ce nœud.
- Un nœud ne doit pas avoir deux enfants consécutifs qui sont des nœuds texte.
- Un nœud ne doit pas avoir des enfants qui sont des nœuds texte dont le contenu est vide.

Contraintes sur les nœuds (2)





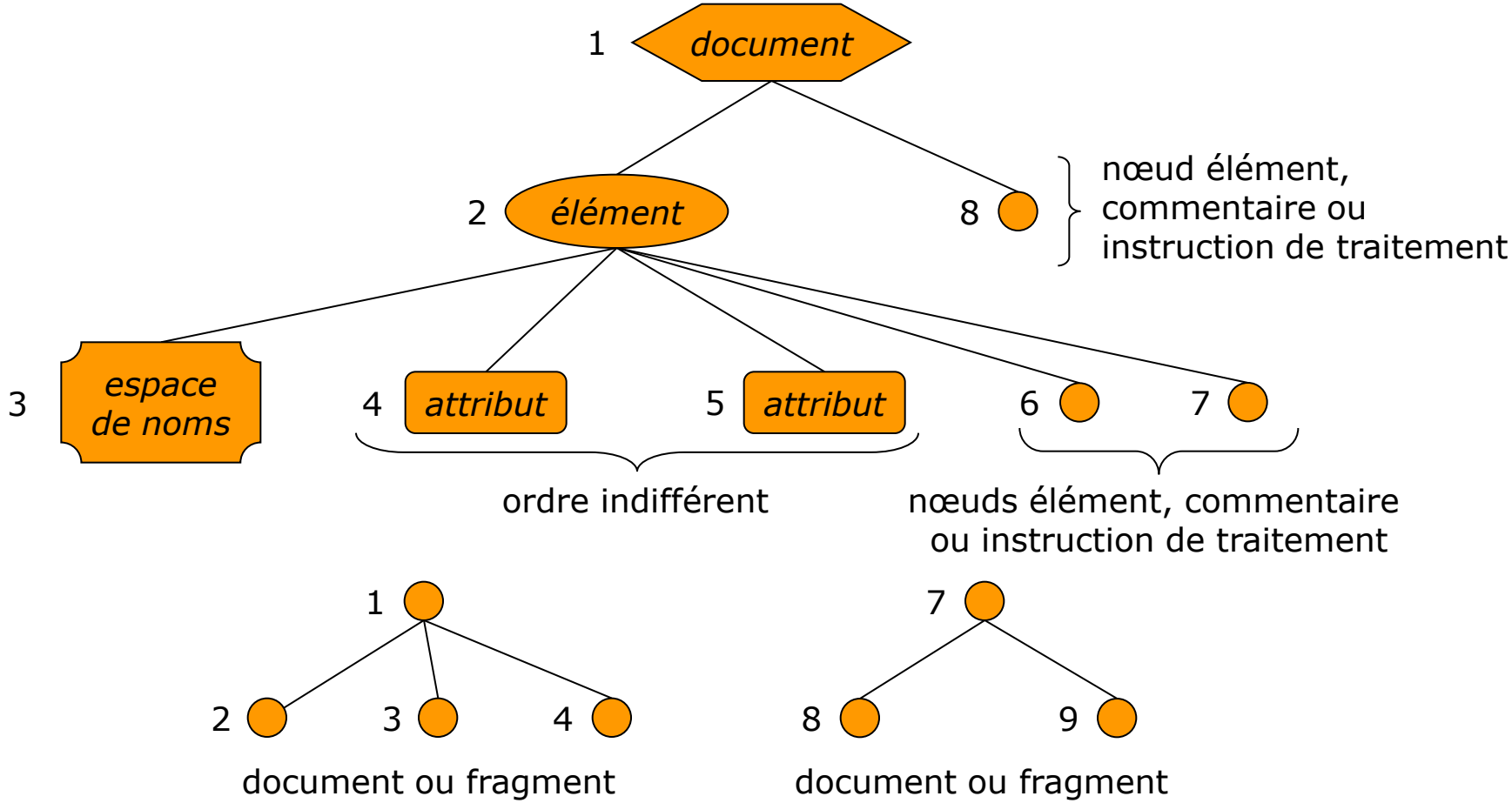
Ordre du document (1)

- L'ensemble des nœuds accessibles lors d'un traitement (requête ou transformation) est muni d'un ordre total : l'**ordre du document**
- Cet ordre doit être stable, c.-à-d. ne pas changer lors de ce traitement.
- L'ordre du document correspond à l'ordre de lecture, dans le document XML, des constituants représentés par chaque nœud.

Ordre du document (2)

- Ordre des nœuds d'un même arbre :
  - Le nœud racine est le premier nœud.
  - Chaque nœud apparaît avant tous ses descendants.
  - Tous les nœuds espace de noms fils d'un nœud élément *e* doivent suivre immédiatement *e*.
  - Tous les nœuds attribut fils d'un élément *e*, doivent suivre immédiatement :
    - tous les nœuds espace de noms fils de *e*, s'il en existe ;
    - *e*, sinon.
  - L'ordre des nœuds attribut fils d'un nœud élément est indifférent.
  - Les descendants d'un nœud apparaissent avant ses noeuds frères suivants.
- Ordre des nœuds d'arbres différents :
  - Si un nœud d'un arbre A_1 apparaît avant un nœud d'un arbre A_2, alors tous les nœuds de A_1 doivent apparaître avant tous les nœuds de A_2.

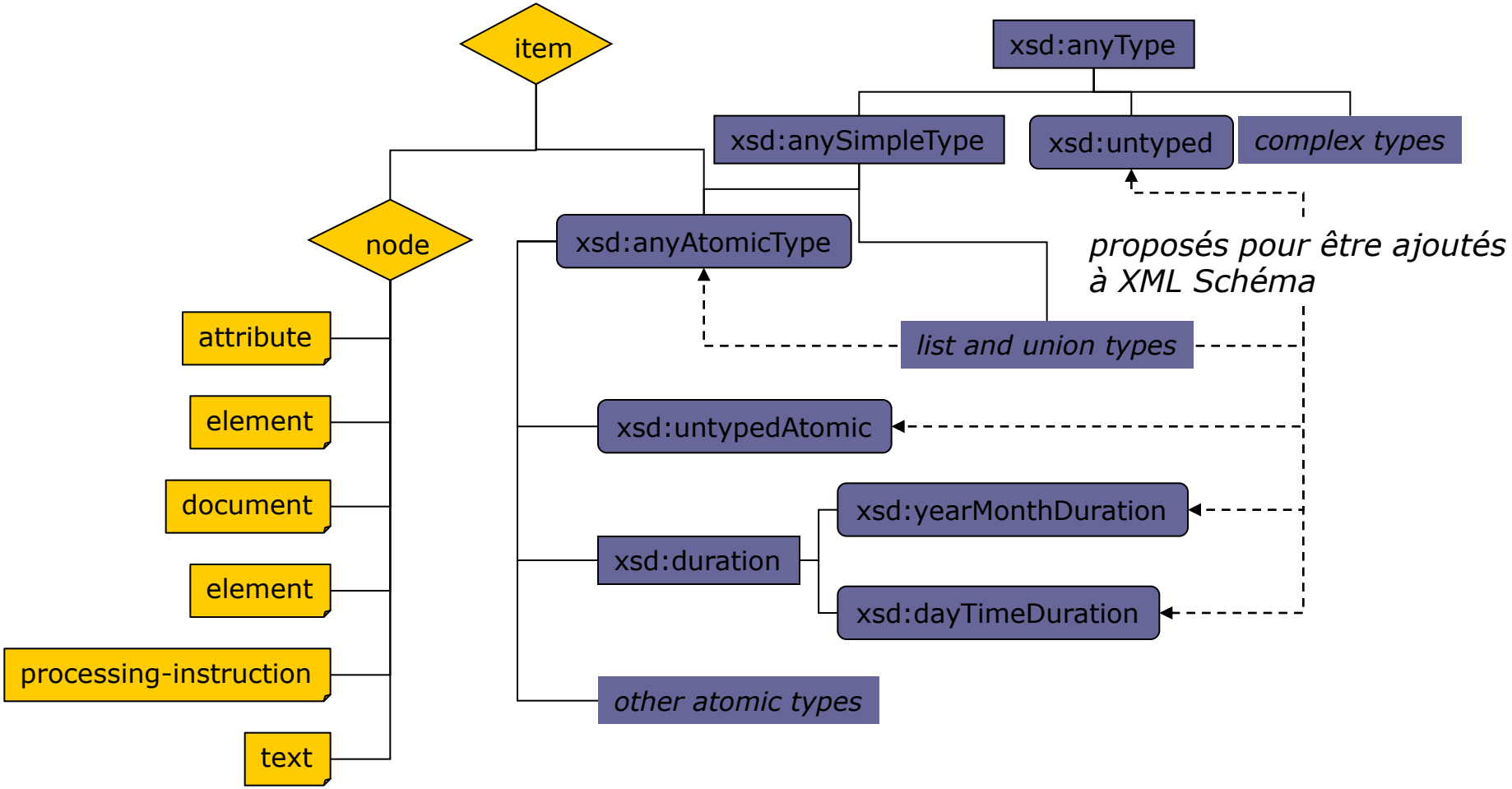
Ordre du document (3)



Types

- Chaque item d'une instance du modèle a une valeur et un type.
- Les types des valeurs atomiques sont ceux de XML Schéma auxquels se rajoutent les 5 types suivants:
  - `xsd:untyped` pour les éléments qui n'ont pas été validés ;
  - `xsd:untypedAtomic` pour les textes ou les valeurs d'attributs qui n'ont pas été validés ;
  - `xsd:anyAtomic` qui inclut toutes les valeurs atomiques ;
  - `xsd:dayTimeDuration` dérivé du type `xsd:duration` en restreignant la représentation lexicale à l'heure, la minute et la seconde ;
  - `xsd:yearMonthDuration` dérivé du type `xsd:duration` en restreignant la représentation lexicale à l'année et au mois.

Hiérarchie des types prédéfinis



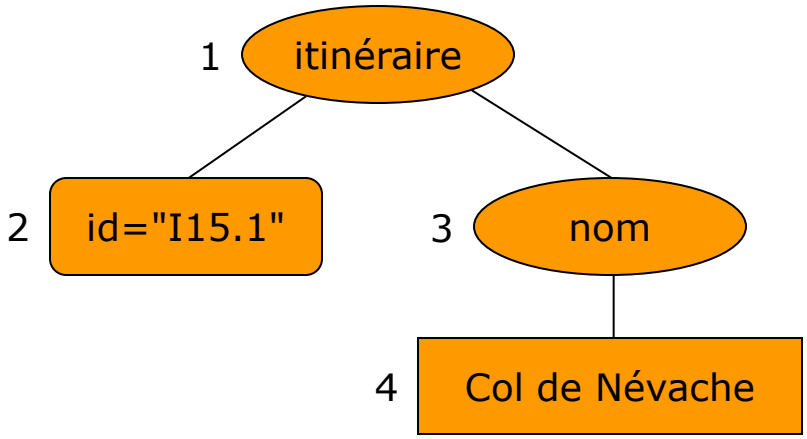
Valeur typée et valeur textuelle d'un nœud

- Chaque nœud d'un document a :
  - une **valeur textuelle** qui est une chaîne de caractères ;
  - une **valeur typée** qui est une séquence de valeurs atomiques.

Valeur textuelle d'un nœud (1)

- La valeur textuelle :
  - d'un nœud document ou élément est la concaténation dans l'ordre du document des valeurs textuelles de ses descendants ;
  - d'un nœud texte est la chaîne de caractères constituant le texte contenu dans ce nœud ;
  - d'un nœud attribut est la chaîne de caractères constituant la valeur de cet attribut.

Valeur textuelle d'un nœud (2)



valeur textuelle(1) = I15.1Col de Névache

valeur textuelle(2) = I15.1

valeur textuelle(3) = Col de Névache

valeur textuelle(4) = Col de Névache

Valeur typée d'un nœud

- La valeur typée :
  - d'un nœud document ou texte est la même que sa valeur textuelle et est une instance du type `xdt:untypedAtomic`.
  - d'un nœud commentaire ou instruction de traitement est la même que sa valeur textuelle et est une instance du type `xsd:string`.
  - d'un nœud attribut ou élément est calculée à partir de sa valeur textuelle et de son type.

Exemple de valeur typée d'un nœud attribut

- La valeur typée d'un nœud attribut ayant la valeur textuelle 3178 et le type `xsd:integer` est 3178 de type `xsd:integer`.
- La valeur typée d'un nœud attribut ayant :
  - le type `xsd:IDREFS`
  - la valeur textuelle V15.1 V15.2

  est la séquence (V15.1, V15.1) dont les items sont de type `xsd:IDREF`.

Exemple de valeur typée d'un nœud élément

- La valeur typée d'un élément ayant :
  - le type `xsd:untyped`
  - la valeur textuelle 3174

  est 123.45 de type `xsd:untypedAtomic`.
- La valeur typée d'un élément ayant :
  - le type `altitude`, un type complexe avec un contenu simple de type `xsd:decimal`
  - la valeur textuelle 3174.25

  est 3174.25 de type `xsd:decimal`.

Accesseurs (1)

- Le modèle XDM spécifie toutes les propriétés d'un nœud qu'une implantation du modèle doit rendre accessible aux applications, sous la forme d'un ensemble de 17 **accesseurs**.
- Chaque accesseur est défini comme une pseudo-fonction qui retourne la valeur d'une propriété d'un nœud.

Accesseurs (2)

- dm:attributes
- dm:base-uri
- dm:children
- dm:document-uri
- dm:is-id
- dm:is-idrefs
- dm:namespace-bindings
- dm:name-spacenodes
- dm:nilled
- dm:node-kind
- dm:node-name
- dm:parent
- dm:string-value
- dm:typed-name
- dm:typed-value
- dm:unparsed-entity-public-id
- dm:unparsed-entity-system-id

XQuery
un langage d'interrogation de données XML

http://www.w3.org/TR/xquery/

http://www.w3.org/TR/xpath-functions/

http://www.w3.org/TR/xquery-semantics/

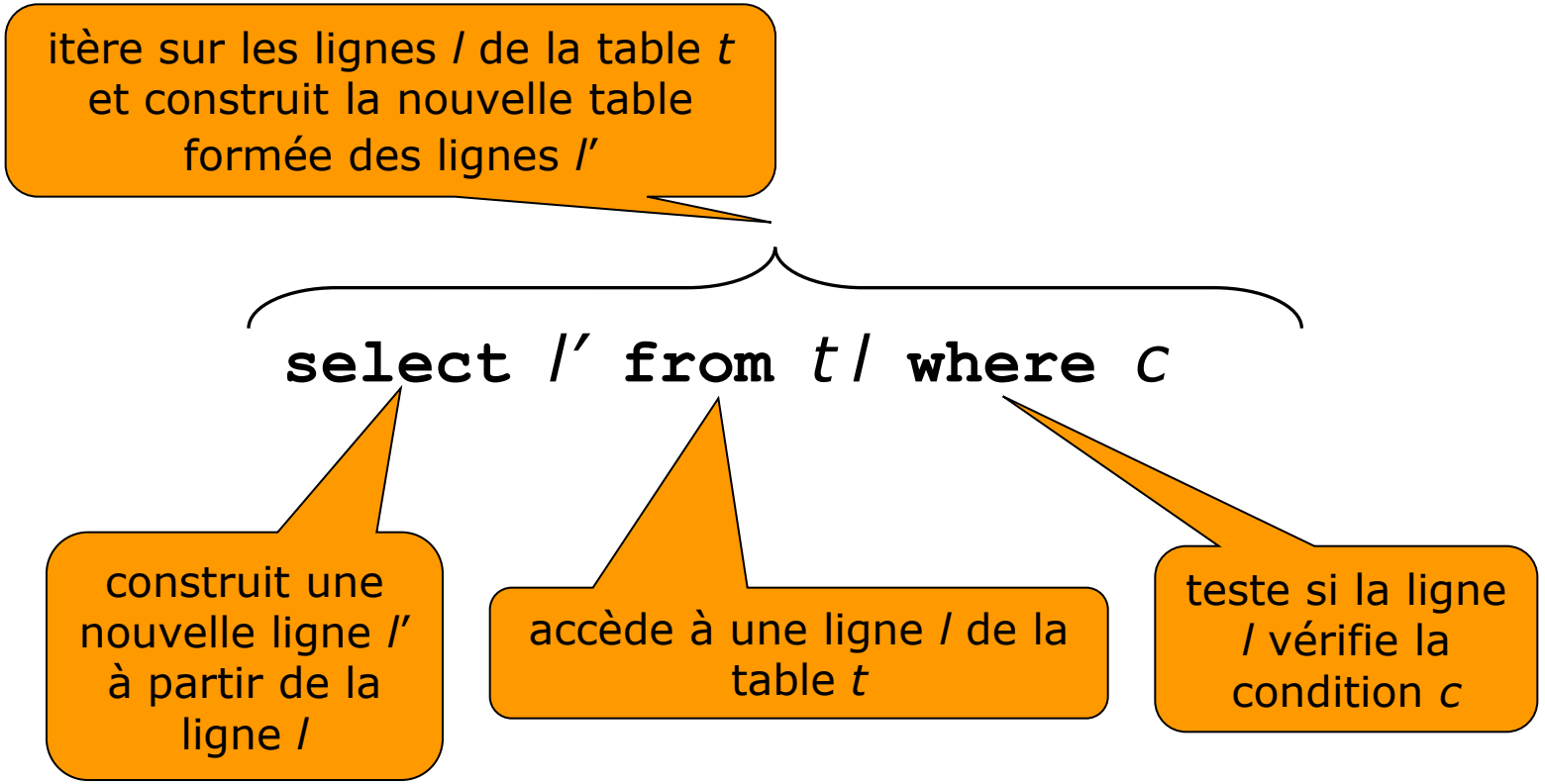
XQuery

- On peut caractériser XQuery comme étant le « SQL de XML ».
- Les premières propositions de langage de requêtes à la SQL pour SGML ou XML datent de 1994.
- XQuery 1.0 est une recommandation du W3C depuis janvier 2007.

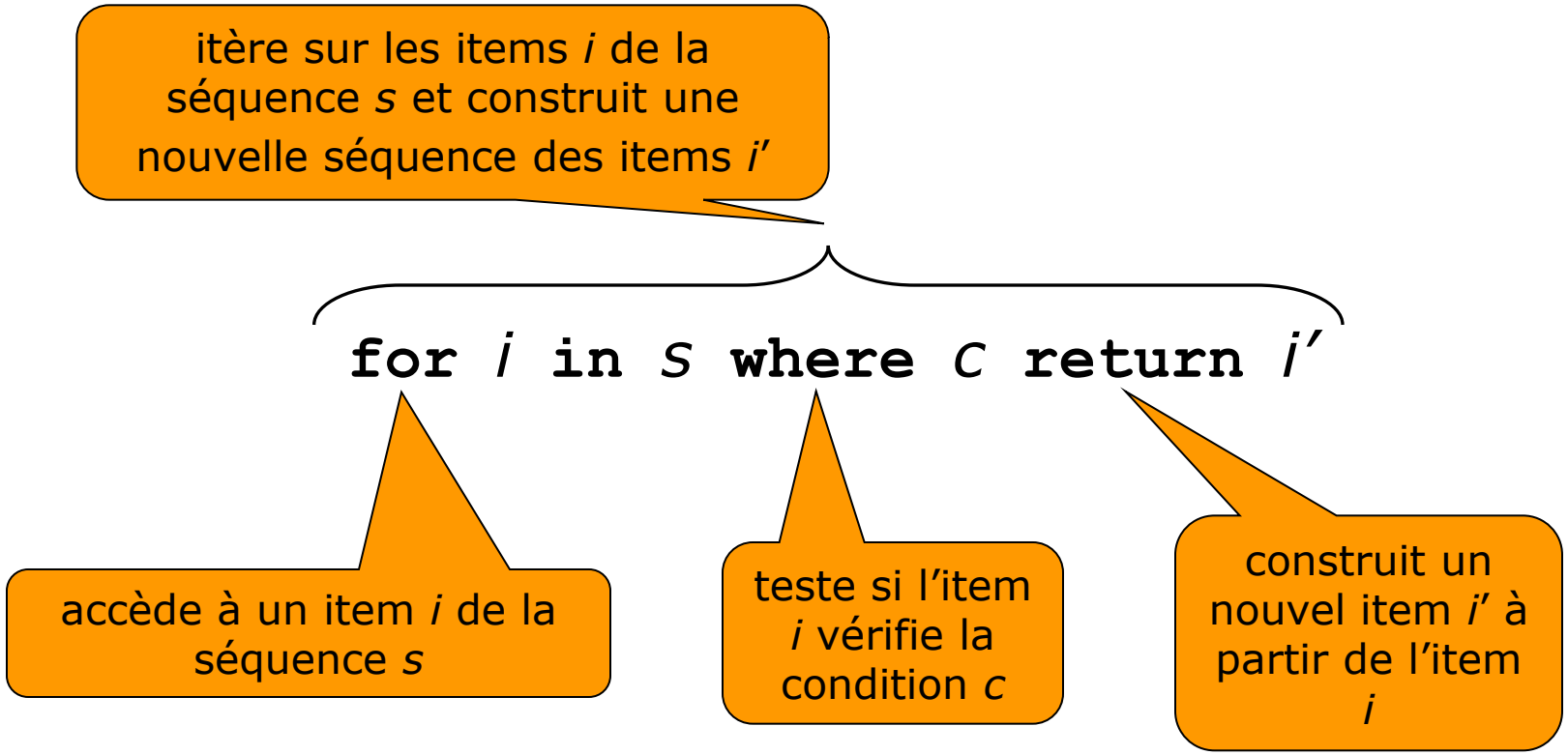
Le document exemple

- La plupart des exemples se rapporteront au document XML *Itinéraires skieurs dans la Vallée de la Clarée*.
- On supposera que ce document :
  - est enregistré dans le fichier `clarée.xml` du répertoire courant ;
  - est conforme à la DTD *Guide d'itinéraires à skis*.

Le bloc de base en SQL



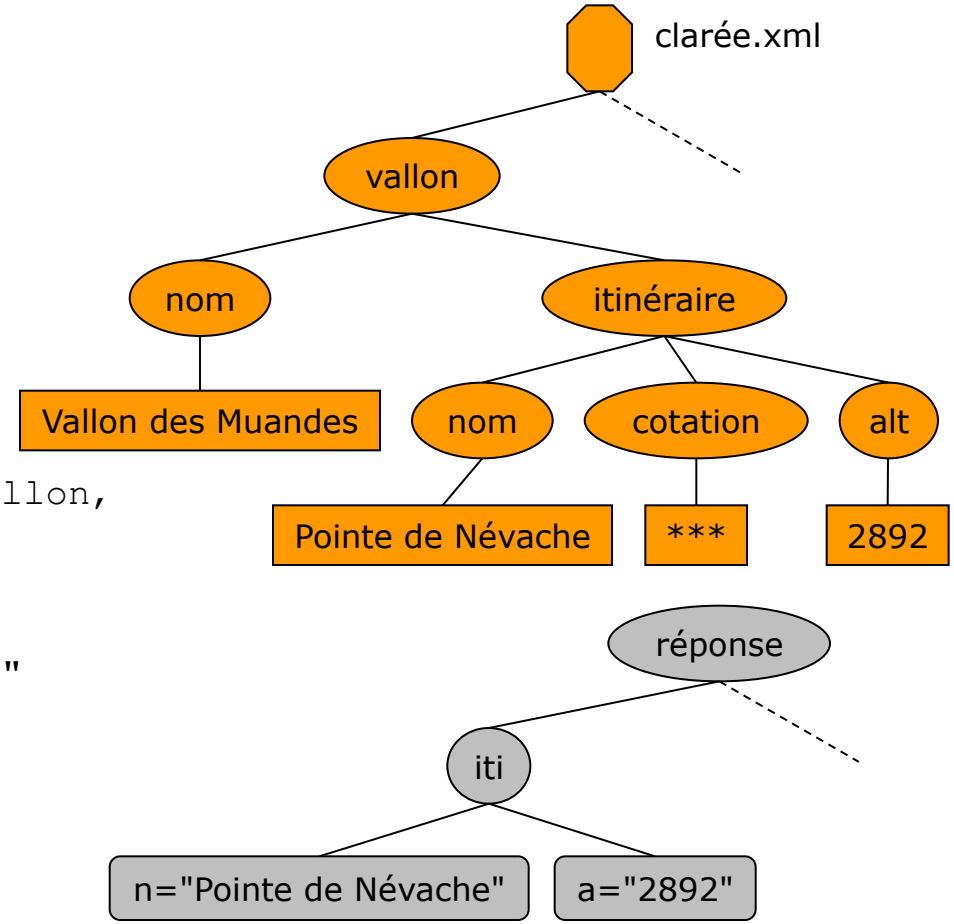
Le bloc de base en XQuery



Exemple d'expression XQuery (1)

*Nom et altitude des itinéraires ****
du Vallon des Muandes
triés par ordre alphabétique de nom ?

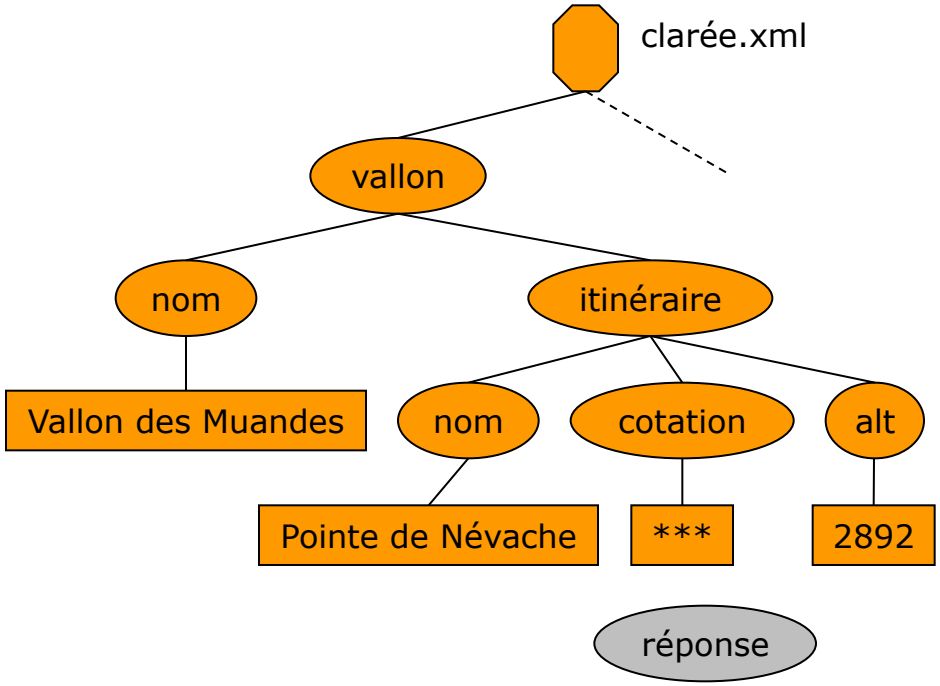
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 where $i/cotation = "***" and
       $v/nom = "Vallon des Muandes"
 order by $n
 return
  <iti n="{$n}" a="{$i/alt}"/>
 }
</réponse>
```



Exemple d'expression XQuery (2)

La réponse est un élément réponse ...

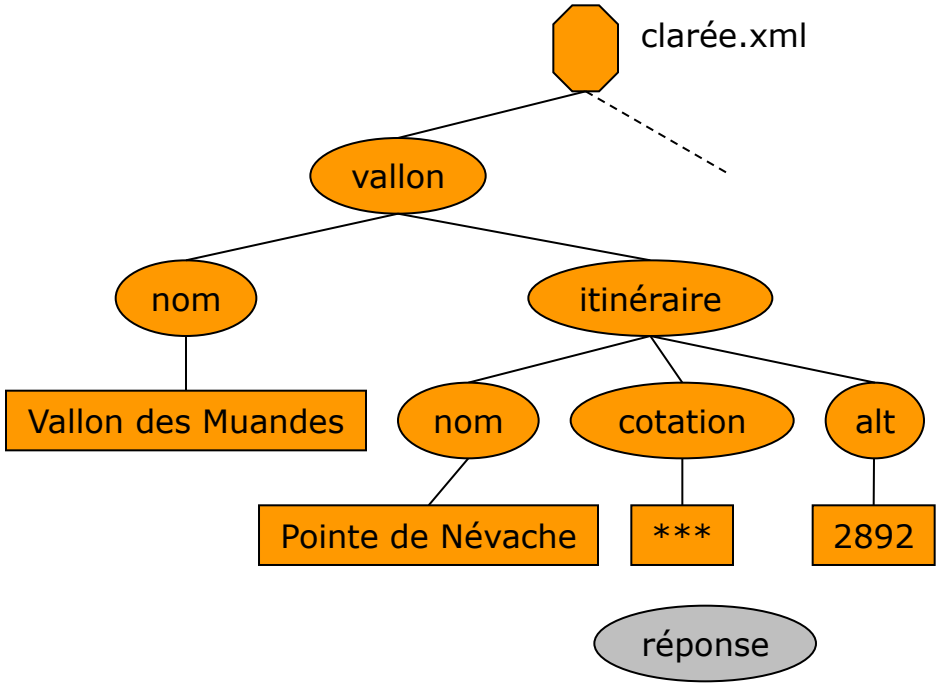
```
<réponse>
...
</réponse>
```



Exemple d'expression XQuery (3)

... dont le contenu est la valeur de l'expression entre accolades ...

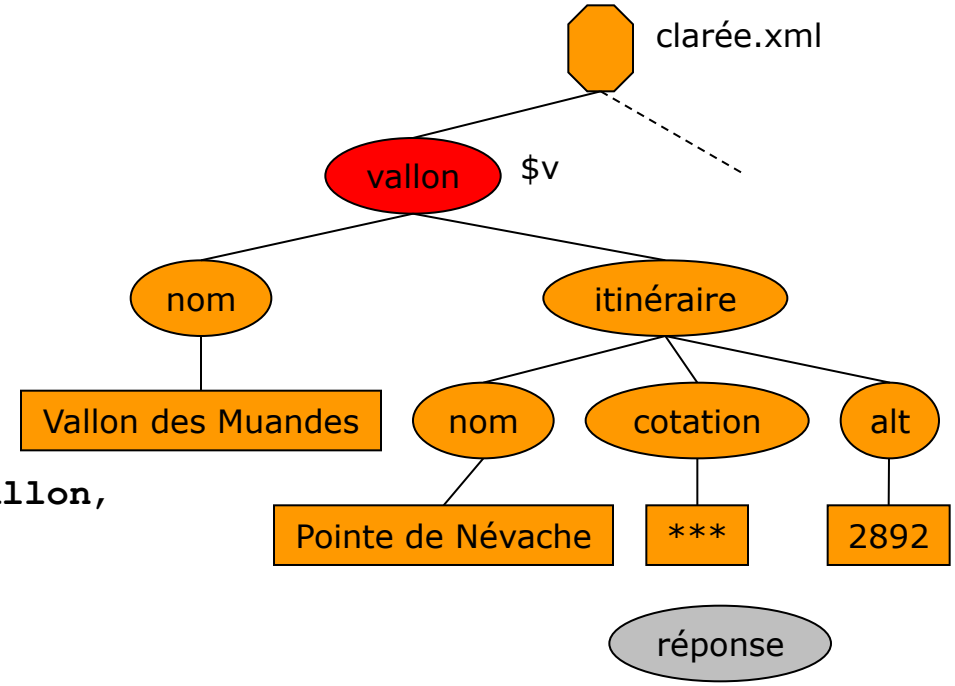
```
<réponse>
  {
  ...
  }
</réponse>
```



Exemple d'expression XQuery (4)

Pour chaque vallon v
du document `clarée.xml`

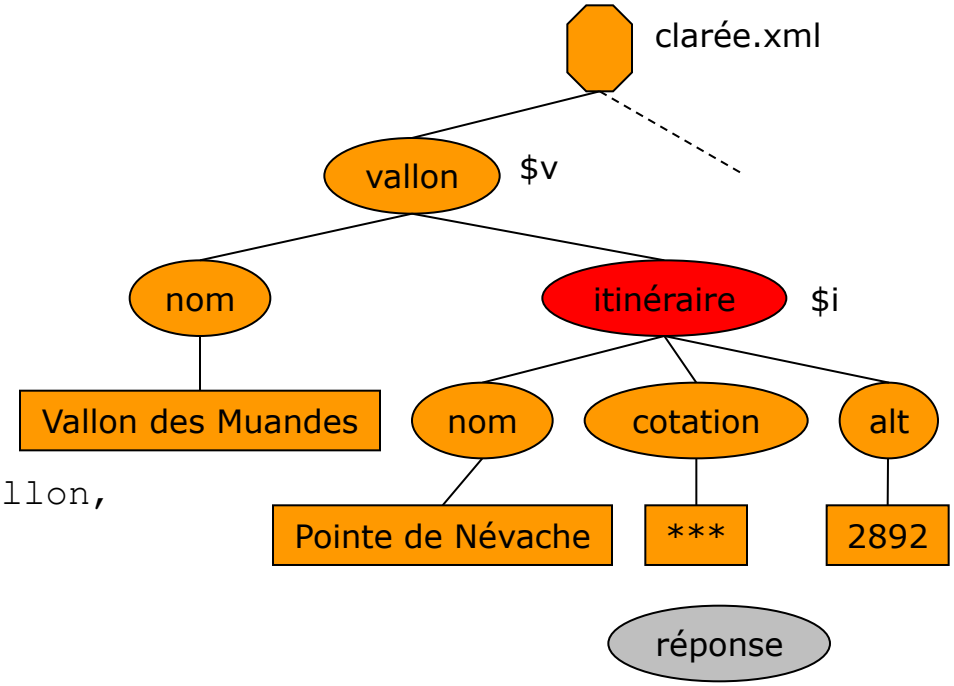
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
 ...
 }
</réponse>
```



Exemple d'expression XQuery (5)

... et pour chaque itinéraire i de v ...

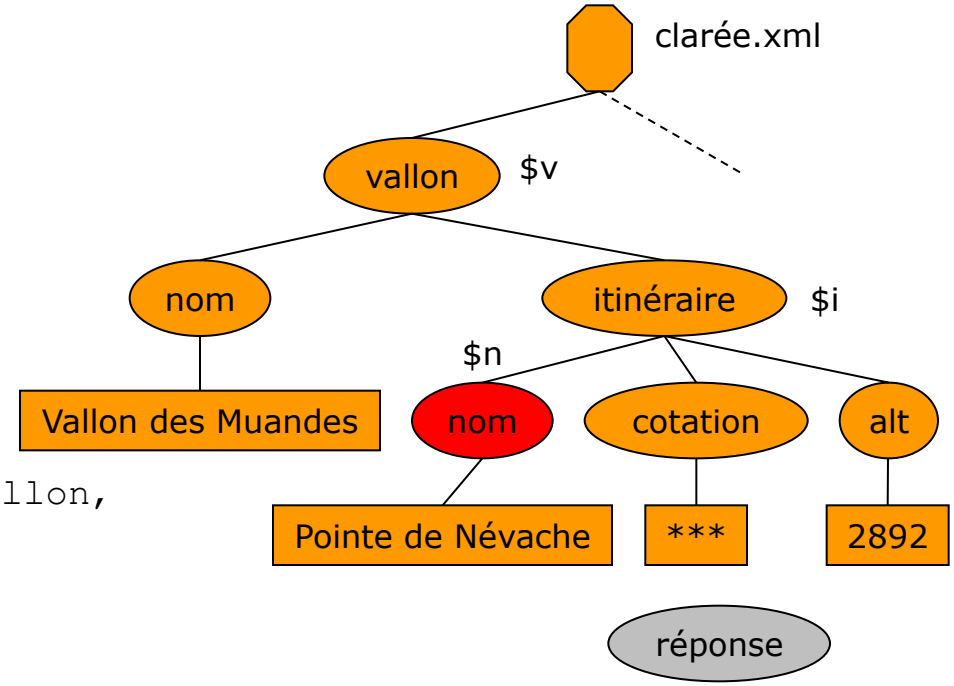
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 ...
 }
</réponse>
```



Exemple d'expression XQuery (6)

... soit n le nom de l'itinéraire i ...

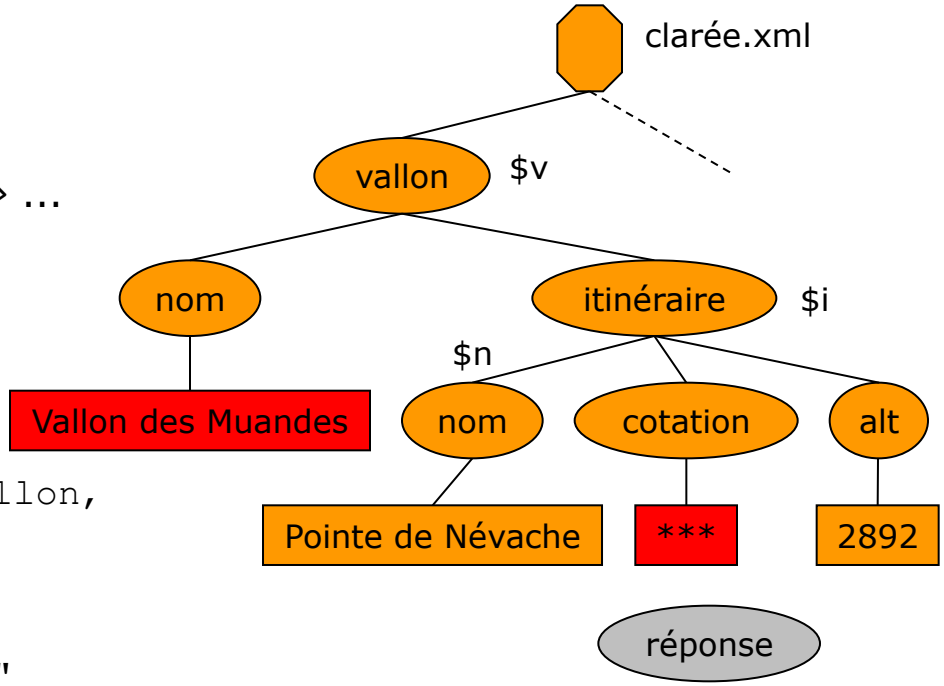
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 ...
 }
</réponse>
```



Exemple d'expression XQuery (7)

*... si la cotation de i est ****
et le nom de v est « Vallon des Muandes » ...

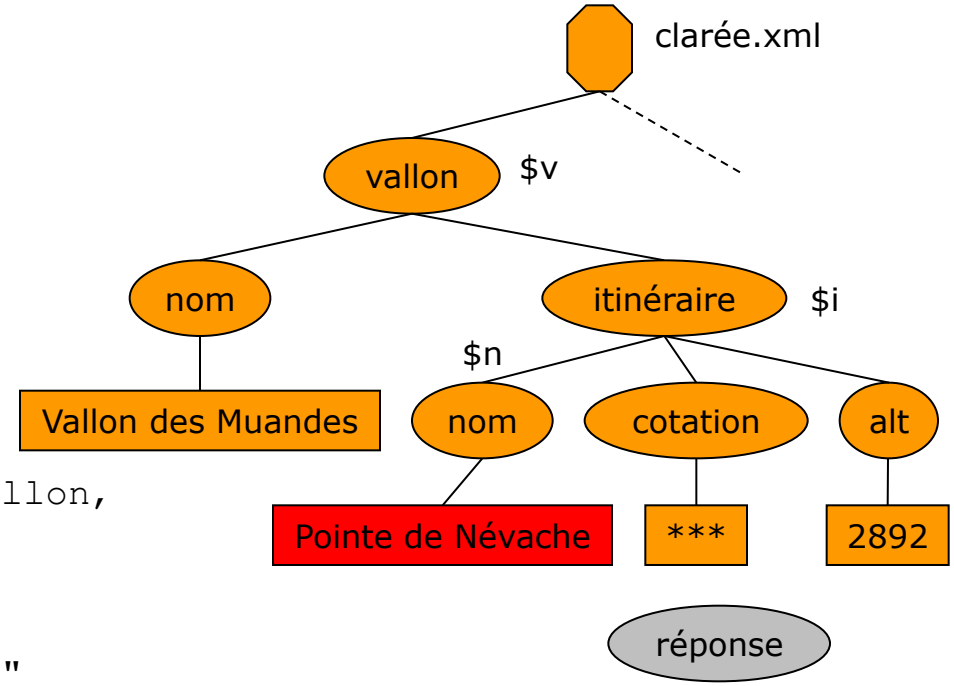
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 where $i/cotation = "***" and
       $v/nom = "Vallon des Muandes"
 ...
 }
</réponse>
```



Exemple d'expression XQuery (8)

... on retourne par ordre alphabétique de nom d'itinéraire ...

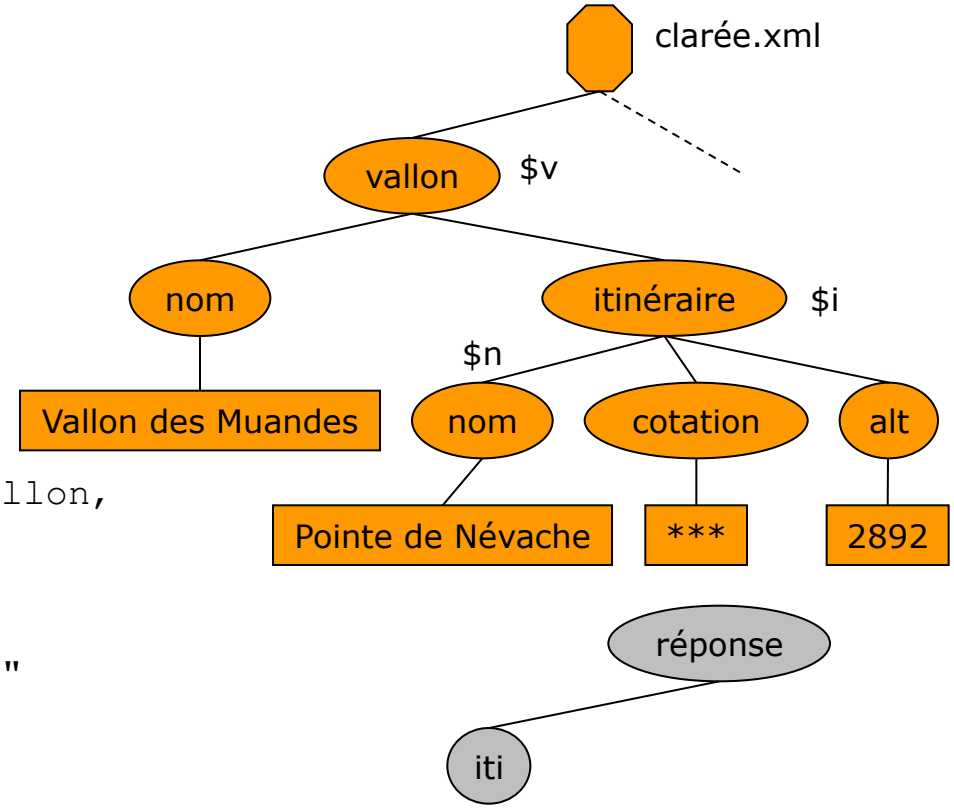
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 where $i/cotation = "***" and
       $v/nom = "Vallon des Muandes"
 order by $n
 return
 ...
 }
</réponse>
```



Exemple d'expression XQuery (9)

... un nouvel élément iti ...

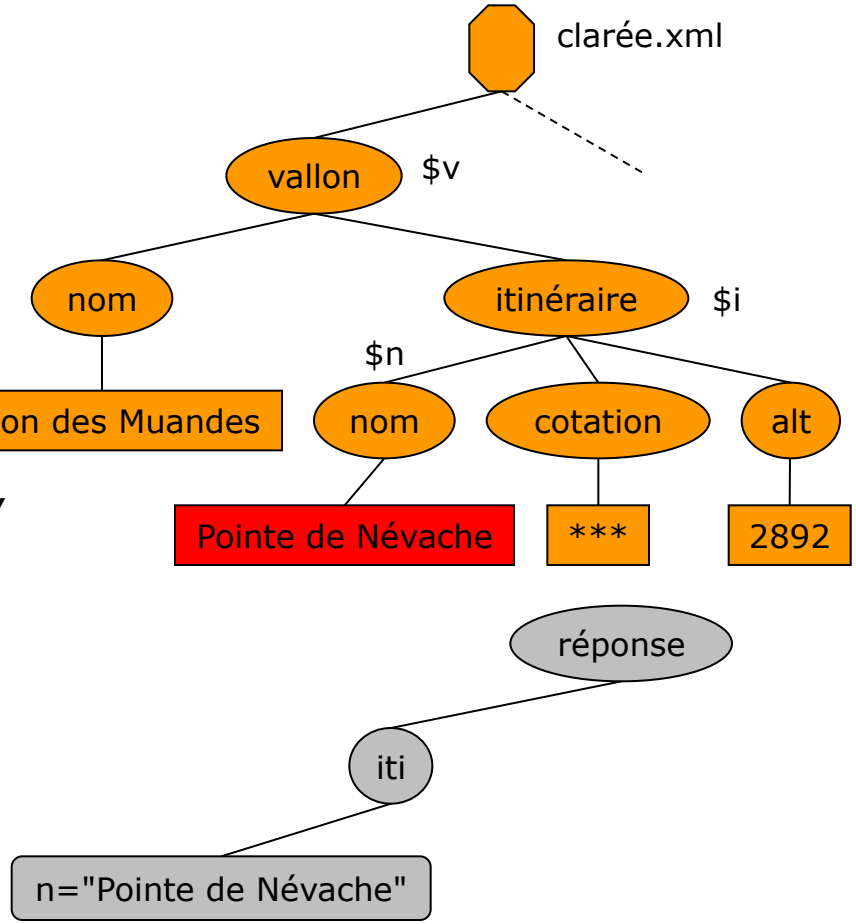
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 where $i/cotation = "***" and
       $v/nom = "Vallon des Muandes"
 order by $n
 return
  <iti ... />
 }
</réponse>
```



Exemple d'expression XQuery (10)

... ayant un attribut n dont la valeur est le nom de l'itinéraire i ...

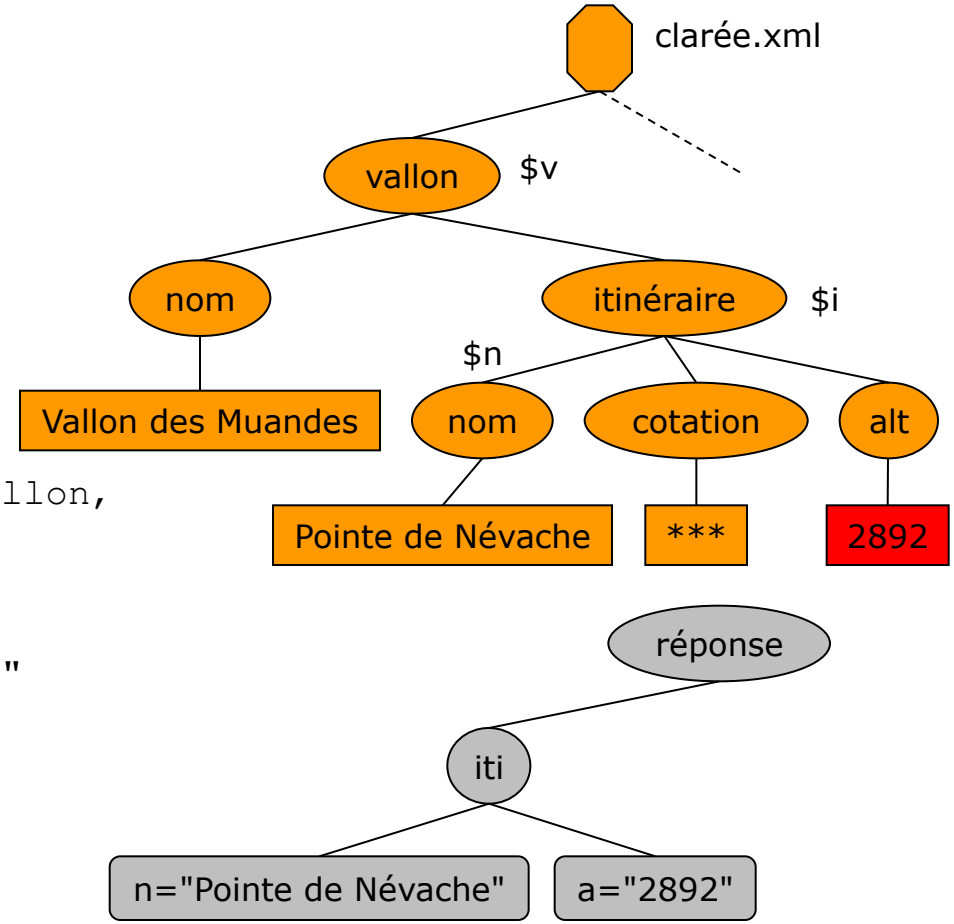
```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 where $i/cotation = "***" and
       $v/nom = "Vallon des Muandes"
 order by $n
 return
  <iti n="{$n}" ... />
 }
</réponse>
```



Exemple d'expression XQuery (11)

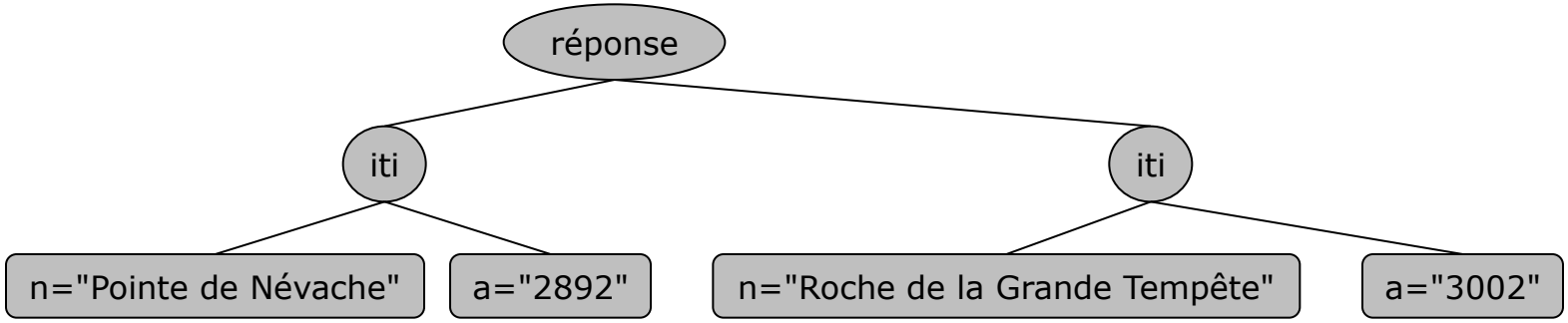
... et un attribut a dont la valeur est l'altitude de l'itinéraire i.

```
<réponse>
 {
 for $v in fn:doc("clarée.xml")//vallon,
     $i in $v/itinéraire
 let $n := $i/nom
 where $i/cotation = "***" and
       $v/nom = "Vallon des Muandes"
 order by $n
 return
  <iti n="{$n}" a="{$i/alt}"/>
 }
</réponse>
```

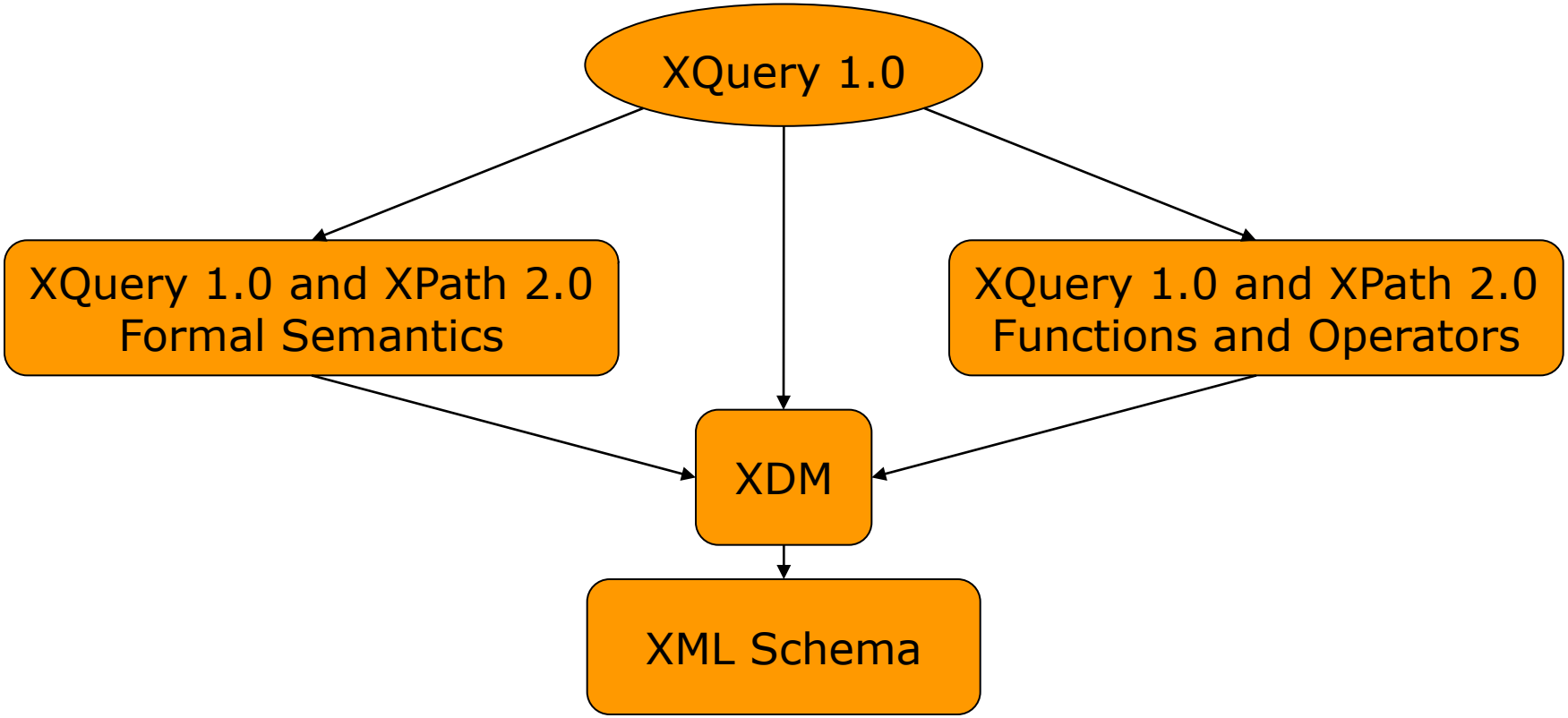


Exemple d'expression XQuery (12)

```
<réponse>
   <iti n="Pointe de Névache" a="2892"/>
   <iti n="Rocher de la Grande-Tempête"  a="3002"/>
</réponse>
```



Spécifications associées



Préfixes d'espace de noms prédéclarés

- Les préfixes suivants sont prédéclarés :
  - `xml` = http://www.w3.org/XML/1998/namespace
    espace de noms du langage XML
  - `xs` = http://www.w3.org/2001/XMLSchema
    espace de noms du langage XML Schema
  - `fn` = http://www.w3.org/2005/xpath-functions
    espace de noms des fonctions prédéfinies
  - `local` = http://www.w3.org/2005/xquery-local-functions
    espace de noms des fonctions locales à un module

Typage des données

- Une caractéristique importante du langage XQuery est sa grande souplesse vis-à-vis du typage des données qu'il manipule :
  - Les documents XML interrogés peuvent être ou non conformes à un schéma XML Schema.
  - Lorsque les opérandes d'un opérateur ou les arguments d'une fonction ne sont pas conformes aux types attendus, un mécanisme de conversion automatique est déclenché afin d'essayer d'obtenir cette conformité.
  - Le type attribué à la valeur d'une expression lors de son analyse, son **type statique**, peut être affiné à l'issue de l'évaluation de cette expression, c.-à-d. lorsque cette valeur est effectivement connue : ce type affiné est le **type dynamique** de cette valeur.

Modélisation des données

- Les données XML manipulés par XQuery doivent être conformes au modèle XDM :
  - **cette modélisation n'est pas du ressort de XQuery**.
- Chaque nœud d'une instance XDM a une annotation de type qui est une expression de type XML Schema.
- Si cette instance provient d'un document XML qui a été **validé** conformément à un schéma XML Shema, cette annotation est celle qui lui a été attribuée lors de cette validation.
- Sinon:
  - Un nœud attribut ou un nœud texte qui n'a pas été validé est annoté par `xs:untypedAtomic`.
  - Un nœud élément qui n'a pas été validé est annoté par `xs:untyped`. Ceux de ses descendants qui sont des nœuds élément sont annotés par `xs:untyped` et ceux qui sont des nœuds texte sont annotés par `xs:untypedAtomic`.
  - Un nœud élément qui a n'été que partiellement validé est annoté par `xs:anyType`. Ceux de ses descendants qui ont été validés sont annotés par un type plus spécifique.

Expression

- Une expression est construite à partir de littéraux, d'opérateurs, d'appels de fonctions, d'itérateurs…
- Toute expression a un type et une valeur.
- La valeur d'une expression est une instance du modèle XDM : une séquence d'items.
- Une expression est évaluée dans un contexte défini par :
  - les informations disponibles lors de l'analyse de l'expression : le **contexte statique** ;
  - les informations présentes lors de l'évaluation de l'expression : le **contexte dynamique**.

Contexte statique

- Le contexte statique est constitué des informations disponibles lors de l'analyse d'une expression :
  - espaces de noms et espace de nom par défaut ;
  - définition de types, déclarations d'éléments et d'attributs,
  - nom et type des variables ;
  - signature des fonctions ;
  - collations (une collation spécifie comment deux chaînes de caractères doivent être comparés) et collation par défaut ;
  - mode de construction (`preserve` ou `strip`) d'un nœud ;
  - mode d'ordonnancement (`ordered` ou `unordered`) des séquences de nœuds retournées par certains opérateurs ;
  - mode de traitement (`preserve` ou `strip`) des espaces frontières dans un constructeur direct de nœud élément ;

  …

Contexte dynamique

- Le contexte dynamique est constitué du contexte statique augmenté :
  - du **focus** qui est l'environnement ajouté pendant l'évaluation d'expressions de parcours ou de filtrage d'une séquence ;
  - de la valeur des variables ;
  - du code des fonctions ;
  - de la date et de l'heure courantes ;
  - des documents ou des collections de documents accédés par les fonctions `fn:doc` et `fn:collection` ;

  ...

Type statique et type dynamique

- A l'issue de l'analyse (lexicale, syntaxique et sémantique) d'une expression, un **type statique** est attribué à cette expression.
- La valeur d'une expression est conforme à son type statique.
- A l'issue de l'évaluation d'une expression, un **type dynamique** est attribué à la valeur de cette expression. Ce type dynamique peut être plus spécifique que le type statique de l'expression (mais pas moins).
- Par exemple, le type statique d'une expression peut être : `xs:integer*` (séquence d'entiers) et son type dynamique `xs:integer` (entier).

Notations

- Dans la suite on notera :
  - *exp* une expression XQuery ;
  - *valeur*(*exp*) la valeur de l'expression *exp* dans le contexte dynamique ;
  - *exp* $\Rightarrow$ *v* le fait que l'évaluation de l'expression dans le contexte dynamique produit la valeur *v*.

Atomisation

- L'**atomisation** est appliquée à une valeur quand cette valeur est utilisée dans un contexte dans lequel une séquence de valeurs atomiques est attendue.
- On a :
  - *atomisation*(i_1, …, i_n) =
    (*atomisation*(i_1), …, *atomisation*(i_n))
  - *atomisation*(*i*) = *i*, si *i* est une valeur atomique
  - *atomisation*(*i*) = valeur typée de *i*, si *i* est un nœud
- Par exemple :
  - *atomisation*(*valeur*(`78.5, <longueur>62.45</longueur>`))
    = (78.5, 62.45)
- L'atomisation d'une valeur est obtenue en lui appliquant la fonction prédéfinie `fn:data`.

Valeur booléenne effective

- La **valeur booléenne effective** d'une valeur est calculée quand cette valeur est utilisée dans un contexte dans lequel une valeur booléenne est attendue.
- La valeur booléenne effective :
  - d'un singleton de type `xs:boolean` ou dérivé de ce type est elle-même ;
  - d'une séquence vide est faux ;
  - d'une séquence dont le premier item est un nœud est vrai ;
  - d'un singleton *v* de type `xs:string`, `xs:anyURI` ou `xs:untypedAtomic` est :
    - faux si la longueur de *v* est nulle,
    - vrai sinon ;
  - d'un singleton *v* de type numérique ou dérivé d'un type numérique est :
    - faux si *v* est `NaN` ou 0,
    - vrai sinon.
- La valeur booléenne effective d'une valeur est obtenue en lui appliquant la fonction prédéfinie `fn:boolean`.

Types de séquence

- Une expression peut faire appel à des opérateurs qui testent le type des valeurs qu'ils manipulent ou bien les convertissent en un autre type.
- Pour exprimer ces types on utilise une notation appelée **type de séquence**, puisque toute instance du modèle XDM est une séquence.

Exemples de types de séquence

- `xs:date` spécifie un item de type prédéfini `xs:date`
- `attribute()?` spécifie une séquence de 0 ou 1 attribut
- `element()` spécifie un nœud élément quelconque
- `element(titre, xs:string)` spécifie un nœud élément de nom `titre` et de type `xs:string`
- `element(description, *)` spécifie un nœud élément de nom `description` et de type quelconque
- `element(*, xs:string)` spécifie un nœud élément de nom quelconque et de type `xs:string`
- `node()*` spécifie une séquence de longueur quelconque de nœuds de type quelconque
- `item()+` spécifie une séquence d'au moins un item

Fonctions prédéfinies

- Une bibliothèque de fonctions prédéfinies est associée aux langages XPath 2.0 et XQuery 1.0.
- Ces fonctions effectuent toutes les opérations classiques sur les nombres, les chaînes de caractères, les dates, les séquences...
- Elles sont définies dans le document :
  - XQuery 1.0 and XPath 2.0 Functions and Operators

  disponible à l'URL http://www.w3.org/TR/xpath-functions/.
- L'espace de noms associé à ces fonctions est http://www.w3.org/TR/xpath-functions/.
- Il est pré-déclaré et a pour préfixe `fn`.
- Par exemple, la fonction `fn:empty` appliquée à une séquence retourne vrai si cette séquence est vide et faux sinon.

Expressions

Catégories d'expressions

- Littéraux
- Référence à une variable
- Appel de fonction
- Constructeurs de séquence
- Item contexte
- Filtrage d'une séquence
- Expression de chemin
- Combinaisons de séquences de nœuds
- Expressions arithmétiques
- Expressions de comparaison
- Expressions logiques
- Expression conditionnelle
- Constructeurs de nœuds
- Expression FLWOR
- Expressions quantifiées
- Expressions sur les types de séquences

Littéraux

- Littéraux chaînes de caractères
  - Ils s'écrivent entre guillemets simples ou entre guillemets doubles :
    - `"Vallon des Muandes"` `'Vallon des Muandes'`
  - La valeur d'un littéral chaîne de caractères est de type `xs:string`.
- Littéraux numériques
  - Ils s'écrivent classiquement :
    - `2005` `3.14` `1.602e-19` `1.602E-19`
  - La valeur d'un littéral numérique sans `.` et sans `e` ou `E` est de type `xs:integer`.
  - La valeur d'un littéral numérique avec `.` et sans `e` ou `E` est de type `xs:decimal`.
  - La valeur d'un littéral numérique avec . et avec `e` ou `E` est de type `xs:double`.
- Il n'y a pas de littéraux booléens. Les valeurs vrai et faux sont obtenues par l'appel des fonctions prédéfinies `fn:true()` et `fn:false()` qui retournent des valeurs de type `xs:boolean`.

Référence à une variable

$*v*

- *v* est un nom de variable : un nom qualifié.
- Par exemple :
  - `$titre` `$guide:alt`
- $*v* ⇒ valeur de la variable *v* dans le contexte courant

Appel de fonction

$$f(exp_1, \ldots, exp_n)$$

- *f* est le nom de la fonction : un nom qualifié.
- $exp_1, \ldots, exp_n$ sont les arguments de la fonction.
- La fonction *f* peut être une fonction prédéfinie, en ce cas son nom appartient à l'espace de noms `http://www.w3.org/2004/10/xpath-functions` de préfixe `fn`.
- Par exemple :

```
plus-difficile-que($i, "**")
fn:deep-equal($v1, $v2)
```

où `fn:deep-equal` est une fonction prédéfinie qui teste l'égalité structurelle de deux arbres XML.

Evaluation d'un appel de fonction

- Les arguments effectifs sont évalués dans un ordre qui dépend de l'implantation.
- Les règles de conversion d'une valeur d'argument ou d'une valeur de retour sont appliquées.
- Si la fonction est prédéfinie, l'appel est évalué. La valeur retournée a le type de retour déclaré pour la fonction.
- Si la fonction est définie par l'utilisateur, le corps de la fonction est évalué. La valeur retournée par cette évaluation est convertie dans le type de retour déclaré pour la fonction en appliquant les règles de conversion d'une valeur d'argument ou d'une valeur de retour.

Conversion d'une valeur d'argument ou d'une valeur de retour

- Si le type attendu est une séquence d'un type atomique t_a :
  - la valeur à convertir est atomisée produisant une séquence *S* de valeurs atomiques ;
  - chaque valeur de *S* qui est de type `untypedAtomic` :
    - est convertie en `xs:double` si la fonction appelée est prédéfinie et si t_a est un type numérique,
    - est convertie en t_a sinon ;
  - si t_a = `xs:double`, chaque valeur de *S* qui est de type `xs:decimal` ou `xs:float` est convertie en `xs:double` ;
  - si t_a = `xs:float`, chaque valeur de *S* qui est de type `xs:decimal` est convertie en `xs:float` ;
  - si t_a est `xs:string`, chaque valeur de *S* qui est de type `xs:anyURI` est convertie en `xs:string`.
- Si la valeur résultante n'est pas conforme au type attendu une erreur est signalée.

Construction d'une valeur atomique

- Les valeurs de type atomique peuvent être construites en utilisant les fonctions prédéfinies de construction.
- A chaque type atomique *t* de XML Schema, il correspond une fonction de construction *t* qui appliquée à la chaîne de caractères représentant une valeur atomique retourne cette valeur atomique.
- Par exemple :
  - `xs:integer("12")` ⇒ valeur atomique 12 de type `xs:integer`
  - `xs:date("1789-07-14")` ⇒ valeur atomique de type `xs:date` représentant le 14 juillet 1789

Accès aux sources de données XML

- La fonction `fn:doc` traduit le document XML dont l'URI lui est fournie en argument en une instance XDM et retourne le nœud racine de cette instance.
- Par exemple :
  - `fn:doc("clarée.xml")`

  retourne le nœud racine de l'arbre du document *Itinéraires skieurs dans la Vallée de la Clarée* si celui-ci est enregistré dans le fichier `clarée.xml` du répertoire courant.

  Attention ! Nous supposerons que ce document n'a pas été validé et donc que les nœuds élément de cet arbre sont de type `xs:untyped` et les nœuds attribut ou texte sont de type `xs:untypedAtomic`.
- La fonction `fn:collection` retourne la séquence de nœuds contenue dans la ressource dont l'URI lui est fournie en argument.

Constructeurs de séquences

- Séquence vide
  - `()`
- Concaténation de séquences (opérateur `,`)
  - *exp*$_1$`,` *exp*$_2$ $\Rightarrow$ séquence constituée des items de la séquence *valeur*(*exp*$_1$) suivie des items de la séquence *valeur*(*exp*$_2$)
- Séquences d'entiers
  - *exp*$_1$ `to` *exp*$_2$ $\Rightarrow$
    - `()`, si *valeur*(*exp*$_1$) = `()` ou *valeur*(*exp*$_2$) = `()`
    - si *valeur*(*exp*$_1$) et *valeur*(*exp*$_2$) sont convertibles en deux entiers n_1 et n_2 :
      - `()`, si $n_1 > n_2$
      - séquence des entiers de n_1 à n_2, si $n_1 \geq n_2$

Exemples de constructeurs de séquences

- `1 + 5, 4 - 2, 7 * 3` ⇒ 6, 2, 21
- `((9, 1), 5, (8, 2))` ⇒ 9, 1, 5, 8, 2
- `2 to 5` ⇒ 2, 3, 4, 5
- `5 to 2` ⇒ ()

Parcours et filtrage des items d'une séquence

- Les opérateurs `/` et `[]` permettent de parcourir ou de filtrer les items d'une séquence.
- L'expression e_1/e_2 a pour valeur la séquence des items obtenus en évaluant l'expression e_2 pour chacun des nœuds de la séquence e_1.
- L'expression $e_1[e_2]$ a pour valeur la séquence des items de la séquence e_1 qui vérifient la condition e_2.

Focus

- L'évaluation d'une expression exp_1/exp_2 ou $exp_1[exp_2]$ consiste tout d'abord à évaluer l'expression exp_1 qui retourne une séquence d'items, puis à évaluer l'expression exp_2 pour chacun des items *i* de la séquence *S* dans un contexte appelé **focus** qui contient :
  - l'item *i* appelé **item contexte** ;
  - le rang de l'item *i* dans la séquence *S* appelé **position contexte** ;
  - la longueur de la séquence *S* appelée **taille contexte**.
- Ce focus n'existe que durant l'évaluation de exp_2 :
  - **Il disparaît à la fin de chaque itération**.

Accès au focus

- L'**item contexte** est obtenu comme valeur de l'expression `.` (point).
- La position contexte est obtenue en appelant la fonction sans argument `fn:position` :
  - l'expression `fn:position()` a pour valeur la position contexte dans le focus courant.
- La taille contexte est obtenue en appelant la fonction sans argument `fn:last` :
  - L'expression `fn:last()` a pour valeur la taille contexte dans le focus courant.

Parcours d'une séquence de nœuds

- La valeur d'une expression exp_1/exp_2 est la séquence construite de la façon suivante :
  - S := *valeur*(exp_1)
  - Si S n'est pas une séquence de nœuds, une erreur est signalée.
  - L'expression exp_2 est évaluée dans le focus pour chaque nœud n de la séquence S produisant des séquences qui doivent être homogènes (c.-à-d. ne contenir que des nœuds ou que des valeurs atomiques).
  - Les séquences ainsi obtenues sont combinées en une séquence unique de la façon suivante :
    - si elles sont toutes des séquences de nœuds, elles sont concaténées, puis les doublons sont éliminés et les nœuds restants sont triés selon l'ordre du document ;
    - si elles sont toutes des séquences de valeurs atomiques elles sont concaténées.

Filtrage d'une séquence

- Le filtrage d'une séquence *S* par un prédicat `[`*exp*`]` produit la séquence *S'* construite de la façon suivante :
  - *S'* := séquence vide
  - Pour chaque item *i* de *S* :
    - Soit *v* la valeur de l'expression *exp* dans le focus.
    - La valeur de vérité du prédicat [*exp*] est calculée de la façon suivante :
      - si *v* est une valeur numérique, alors elle vraie si *v* est égale (par application de l'opérateur de comparaison de valeur `eq`) à la position contexte, sinon elle fausse ;
      - si *v* n'est pas une valeur numérique, la valeur de vérité du prédicat est égale à la valeur booléenne effective de *v*.
    - Si la valeur de vérité du prédicat est vraie alors *S'* := *S'*, *i*.
- La valeur d'une expression exp_1`[`exp_2`]` est la séquence résultant du filtrage de la séquence *valeur*(exp_1) par le prédicat `[`exp_2`]` .

Exemples de filtrage d'une séquence

- `(1 to 9)[5]` ⇒ 5
- `(1 to 9)[. gt 5]` ⇒ 6, 7, 8, 9

Expression de chemin

- La combinaison des opérateurs `/` et `[]` permet de construire des **expressions de chemin**.
- Un expression de chemin sélectionne dans un arbre la séquence de nœuds atteints par un parcours spécifié par l'expression de chemin.
- Par exemple, l'expression de chemin :
  - `fn:doc("clarée.xml")/guide/vallon/itinéraire`

  a pour valeur la séquence des nœuds itinéraires de l'arbre du document `clarée.xml`.

Accès à la racine d'un arbre

- Le nœud racine de l'arbre auquel appartient un nœud peut être obtenu en appelant la fonction prédéfinie `fn:root` avec ce nœud pour argument.

Pas le long d'un axe

- Un **pas le long d'un axe** (*axis step*) est une expression composée :
  - d'un **axe** ;
  - d'un **test de nœud** ;
  - d'une suite éventuellement vide de **prédicats**.
- Ella a pour valeur la séquence des nœuds que l'on peut atteindre à partir du nœud contexte et qui :
  - appartiennent à l'axe ;
  - vérifient le test de nœud et chacun des prédicats.
- Syntaxe :

  axe::*test de nœud* *prédicat*$_1$ … *prédicat*$_n$

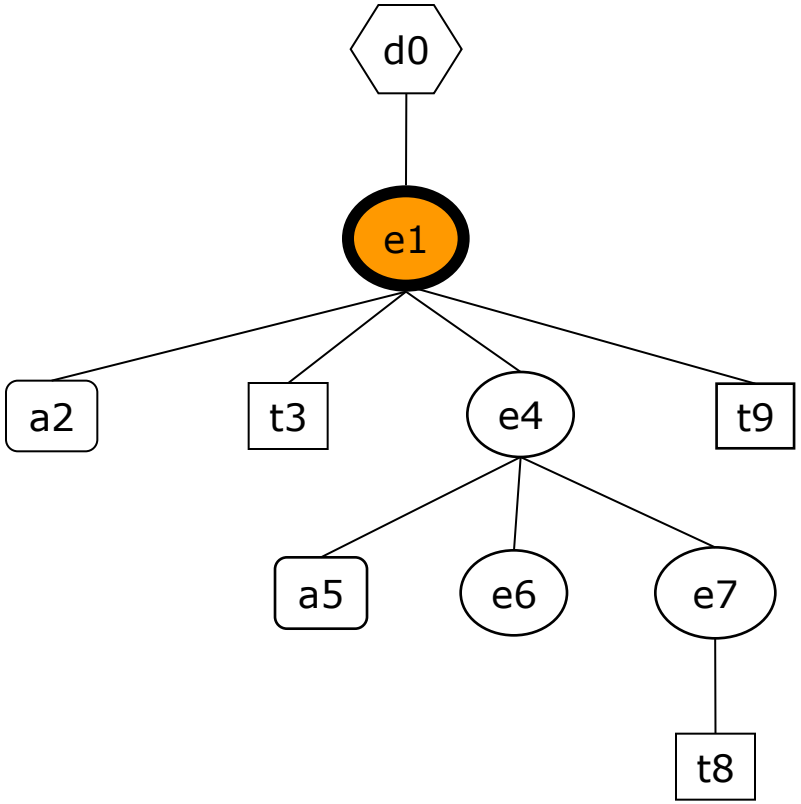
  (les prédicats sont facultatifs)

Axes

- Un axe sélectionne, dans un arbre et à partir du nœud contexte, l'ensemble des nœuds qui peuvent être atteints en suivant une certaine direction.
- On distingue 12 axes :
  - `self`, `child`, `descendant`, `descendant-or-self`, `parent`, `ancestor`, `ancestor-or-self`, `following-sibling`, `preceding-sibling`, `following`, `preceding`, `attribute`.
- Un axe a un **sens** : **avant** ou **arrière.**
  - Un axe dont les nœuds sont soit le nœud contexte, soit des nœuds qui suivent le nœud contexte dans l'ordre du document est un axe avant.
  - Un axe dont les nœuds sont soit le nœud contexte, soit des nœuds qui précèdent le nœud contexte dans l'ordre du document est un axe arrière.
- Un axe a une **sorte de nœud principal**.

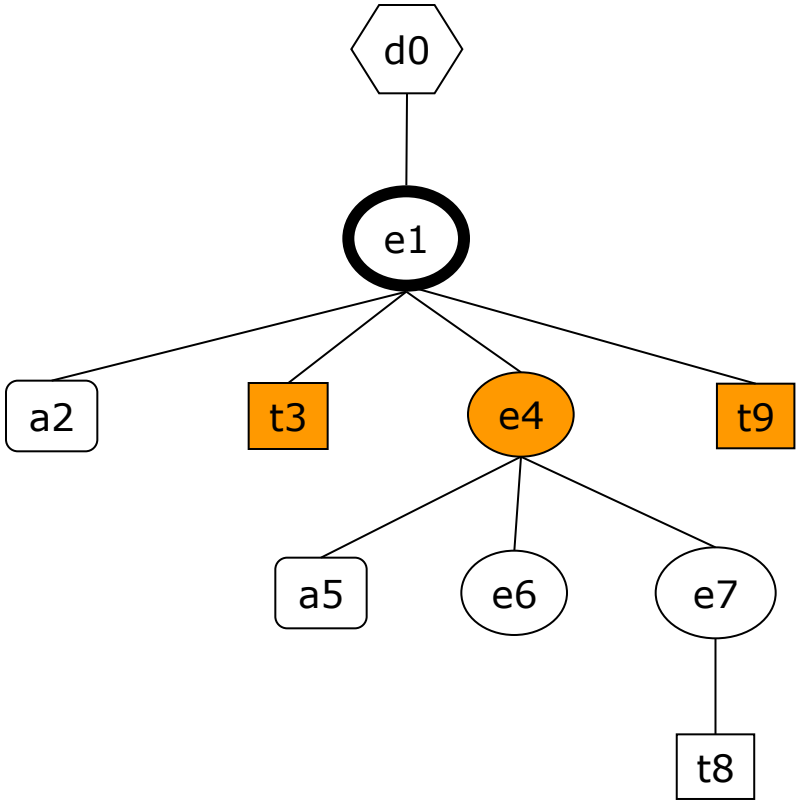
Axe `self`

Sens = avant ou arrière
Sorte de nœud principal = élément
Noeud contexte = e1
Nœuds sélectionnés (dans l'ordre) =
e1



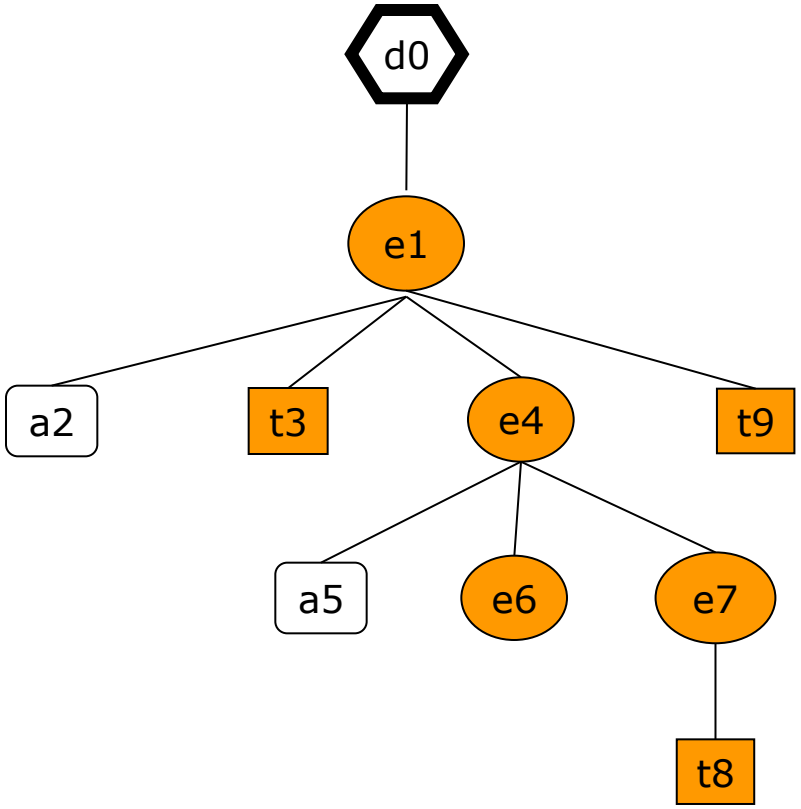
Axe child

Sens = avant
Sorte de nœud principal = élément
Noeud contexte = e1
Nœuds sélectionnés (dans l'ordre) =
t3, e4, t9



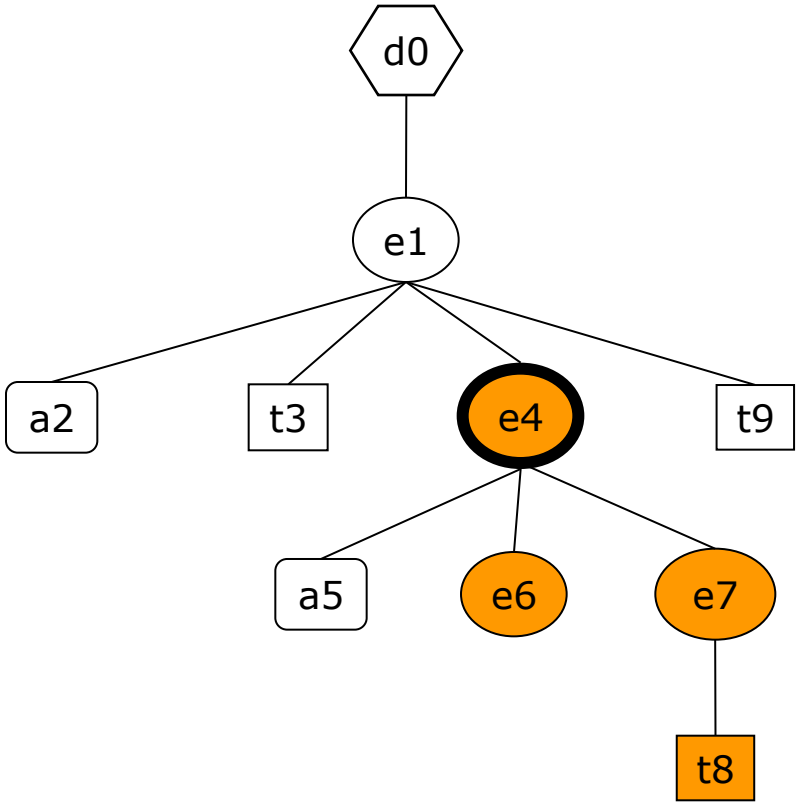
Axe descendant

Sens = avant
Sorte de nœud principal = élément
Noeud contexte = d0
Nœuds sélectionnés (dans l'ordre) =
e1, t3, e4, e6, e7, t8, t9



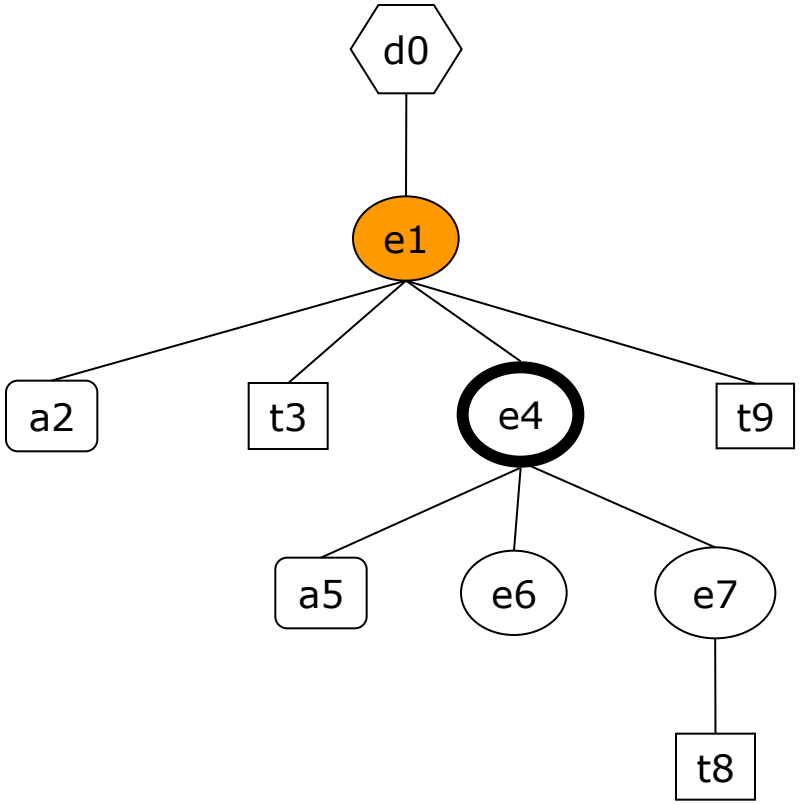
Axe descendant-or-self

Sens = avant
Sorte de nœud principal = élément
Noeud contexte = e4
Nœuds sélectionnés (dans l'ordre) =
e4, e6, e7, t8



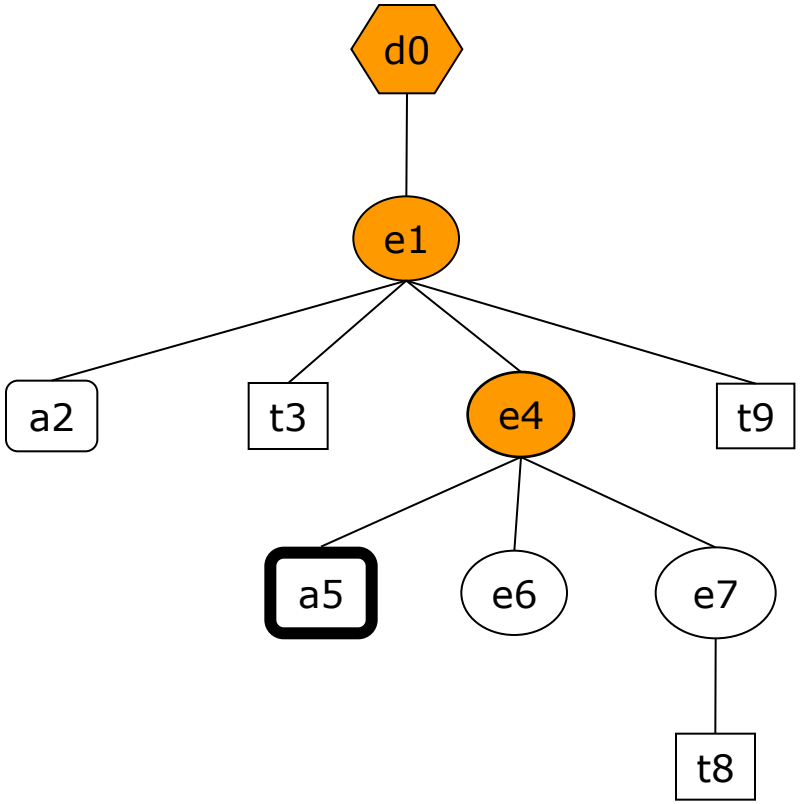
Axe parent

Sens = arrière
Sorte de nœud principal = élément
Noeud contexte = e4
Nœuds sélectionnés (dans l'ordre) =
e1



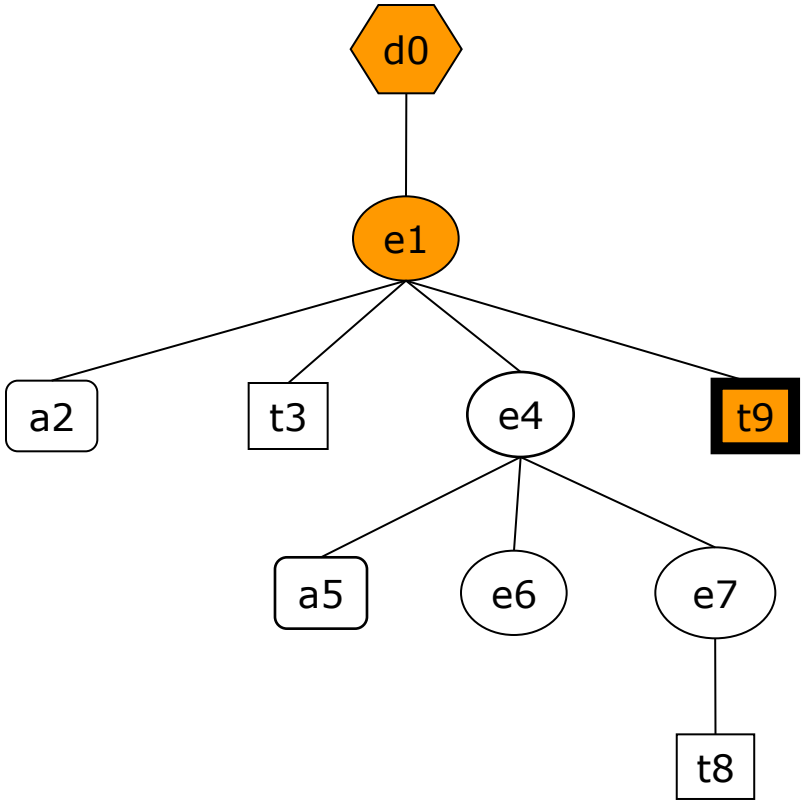
Axe ancestor

Sens = arrière
Sorte de nœud principal = élément
Noeud contexte = a5
Nœuds sélectionnés (dans l'ordre) =
e4, e1, d0



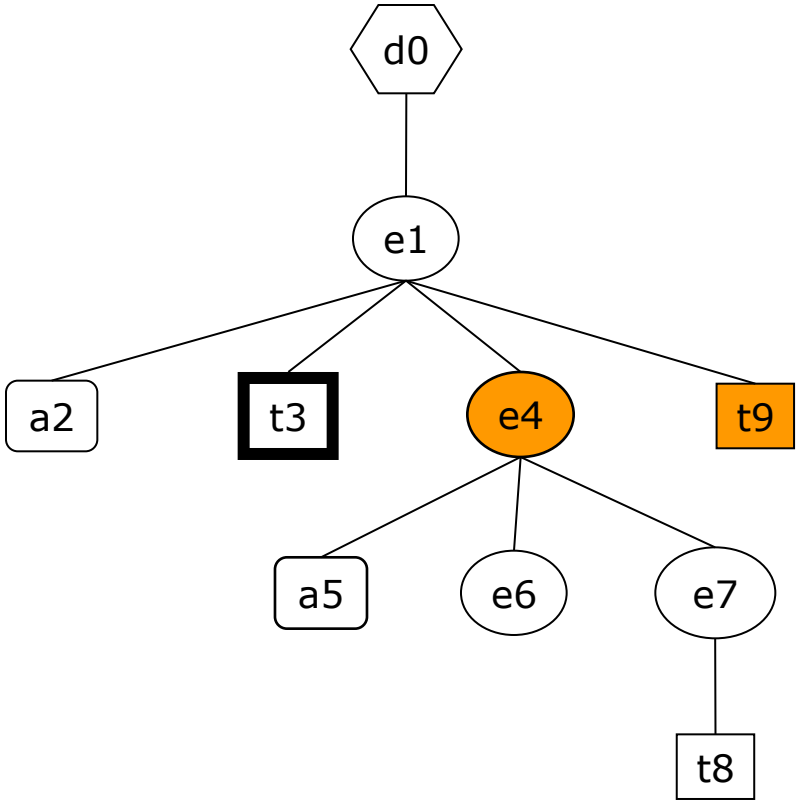
Axe ancestor-or-self

Sens = arrière
Sorte de nœud principal = élément
Noeud contexte = t9
Nœuds sélectionnés (dans l'ordre) =
t9, e1, d0



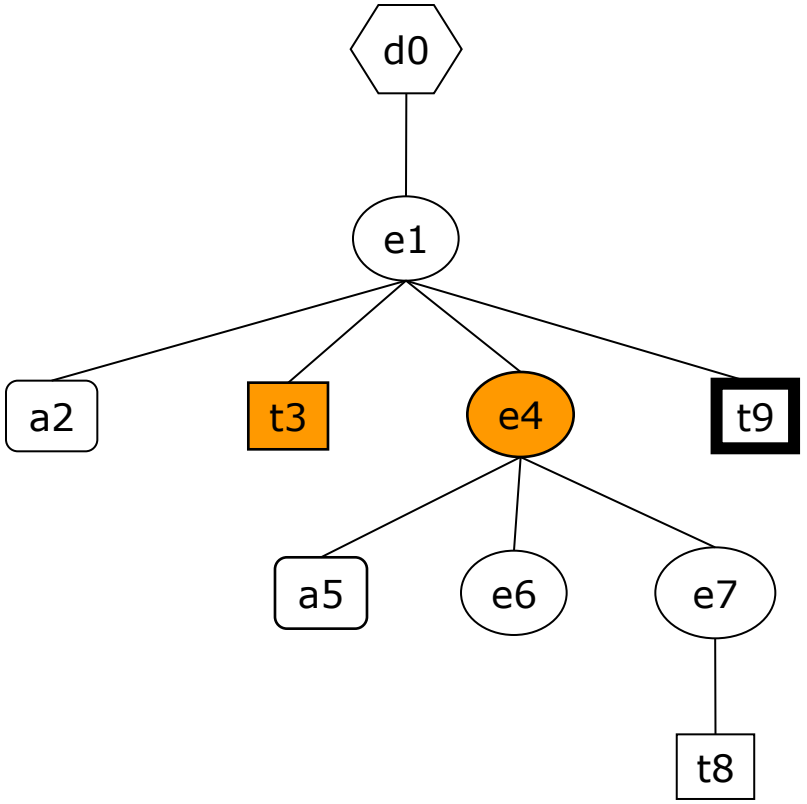
Axe following-sibling

Sens = avant
Sorte de nœud principal = élément
Noeud contexte = t3
Nœuds sélectionnés (dans l'ordre) =
e4, t9



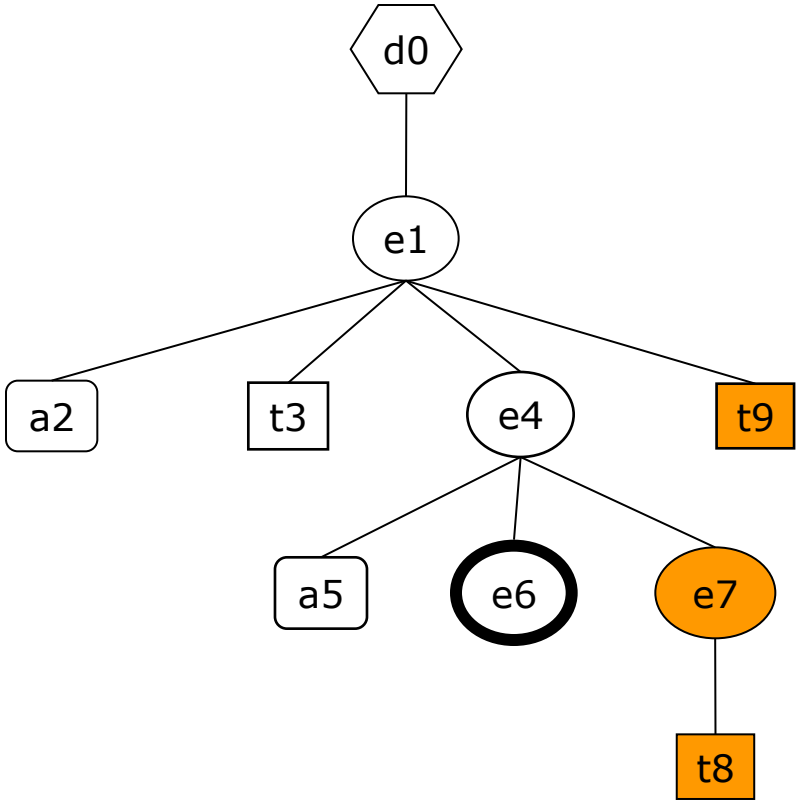
Axe preceding-sibling

Sens = arrière
Sorte de nœud principal = élément
Noeud contexte = t9
Nœuds sélectionnés (dans l'ordre) =
e4, t3



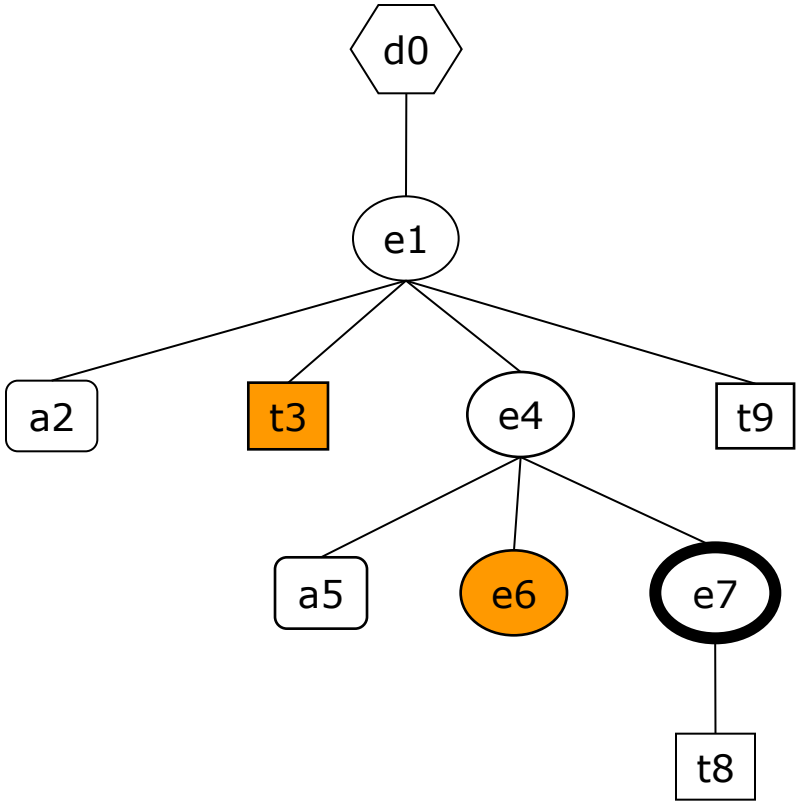
Axe following

Sens = avant
Sorte de nœud principal = élément
Noeud contexte = e6
Nœuds sélectionnés (dans l'ordre) =
e7, t8, t9
(les nœuds de la branche du nœud contexte sont exclus)



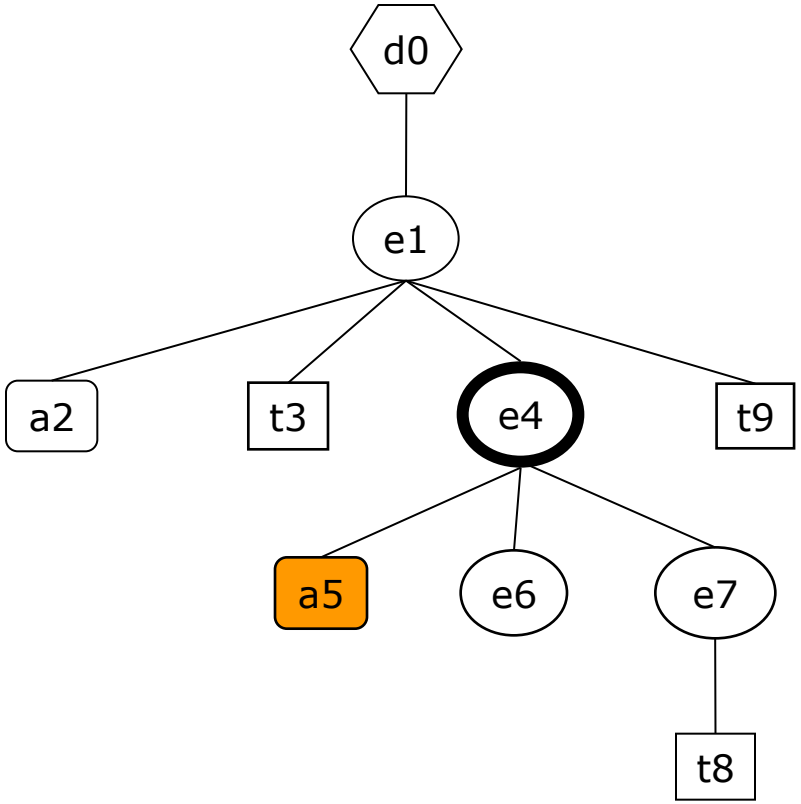
Axe preceding

Sens = arrière
Sorte de nœud principal = élément
Noeud contexte = e7
Nœuds sélectionnés (dans l'ordre) =
e6, t3
(les nœuds de la branche du nœud contexte sont exclus)



Axe attribute

Sens = avant
Sorte de nœud principal = attribut
Noeud contexte = e4
Nœuds sélectionnés (dans l'ordre) =
a5



Tests de nœud

- Un **test de nœud** sélectionne parmi les nœuds de l'axe, ceux dont le nom ou la sorte vérifie une certaine condition.
- Les principales formes de tests de nœud sont les suivantes :
  - *n* où *n* est un nom : sélectionne les nœuds de l'axe de même sorte que la sorte principale de l'axe et de nom *n* ;
  - `*` : sélectionne tout nœud de l'axe de même sorte que la sorte principale de l'axe ;
  - `node()` : sélectionne tout nœud de l'axe ;
  - `attribute()` : sélectionne tout nœud attribut de l'axe ;
  - `attribute(`*n*`)` : sélectionne tout nœud attribut de nom *n* de l'axe ;
  - `attribute(`*n*`, `*t*`)` : sélectionne tout nœud attribut de nom *n* et de type *t* de l'axe ;
  - `document()` : sélectionne tout nœud document de l'axe ;
  - `element()` : sélectionne tout nœud élément de l'axe ;
  - `element(`*n*`)` : sélectionne tout nœud élément de nom *n* de l'axe ;
  - `element(`*n*`, `*t*`)` : sélectionne tout nœud élément de nom *n* et de type *t* de l'axe;
  - `processing-instruction()` : sélectionne tout nœud instruction de traitement de l'axe ;
  - `text()` : sélectionne tout nœud texte de l'axe ;

  …

  (de nom *n* signifie de nom développé égal au nom développé de *n*)

Prédicats

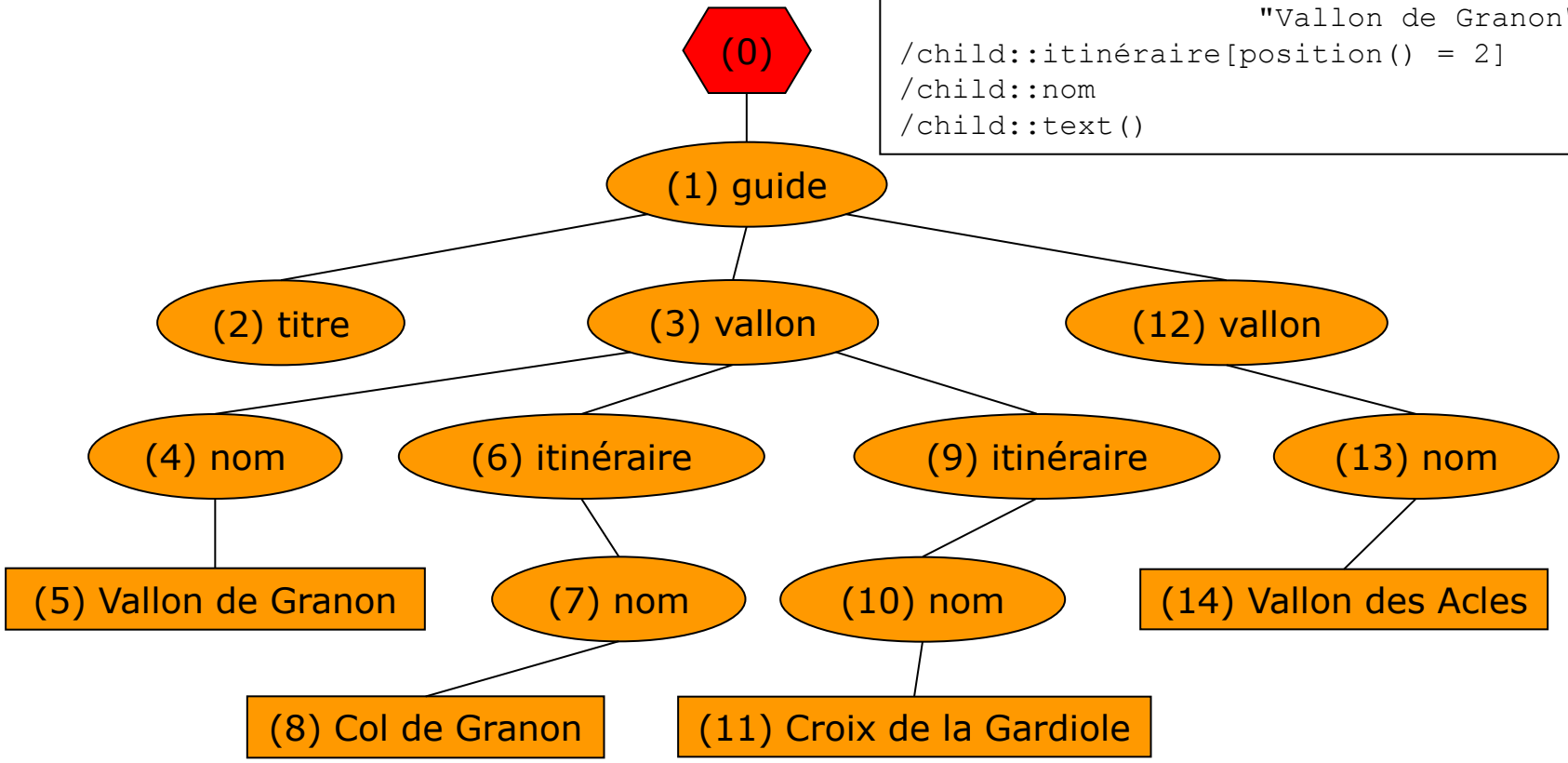
- Un prédicat est destiné à filtrer une séquence de nœuds.
- Un prédicat est une expression placée entre crochets. évaluée dans le focus.
- Par exemple :
  - `[child::nom = "Vallon des Muandes"]` qui est vrai si le nœud contexte a un enfant de nom `nom` dont la valeur textuelle est `"Vallon des Muandes"` ;
  - `[2]` qui est vrai si la position contexte est égale à 2.

Evaluation d'un pas le long d'un axe

- La valeur de l'expression $a::t[exp_1]\ldots[exp_n]$ est la séquence S_{n+2} produite de la façon suivante :
  - S_1 = séquence des nœuds contenus dans l'axe a
  - S_2 = sous-séquence des nœuds de S_1 qui vérifient le test de nœud t
  - Pour i = 1 à n :
    - S_{i+2} = filtrage de S_{i+1} par le prédicat $[exp_i]$

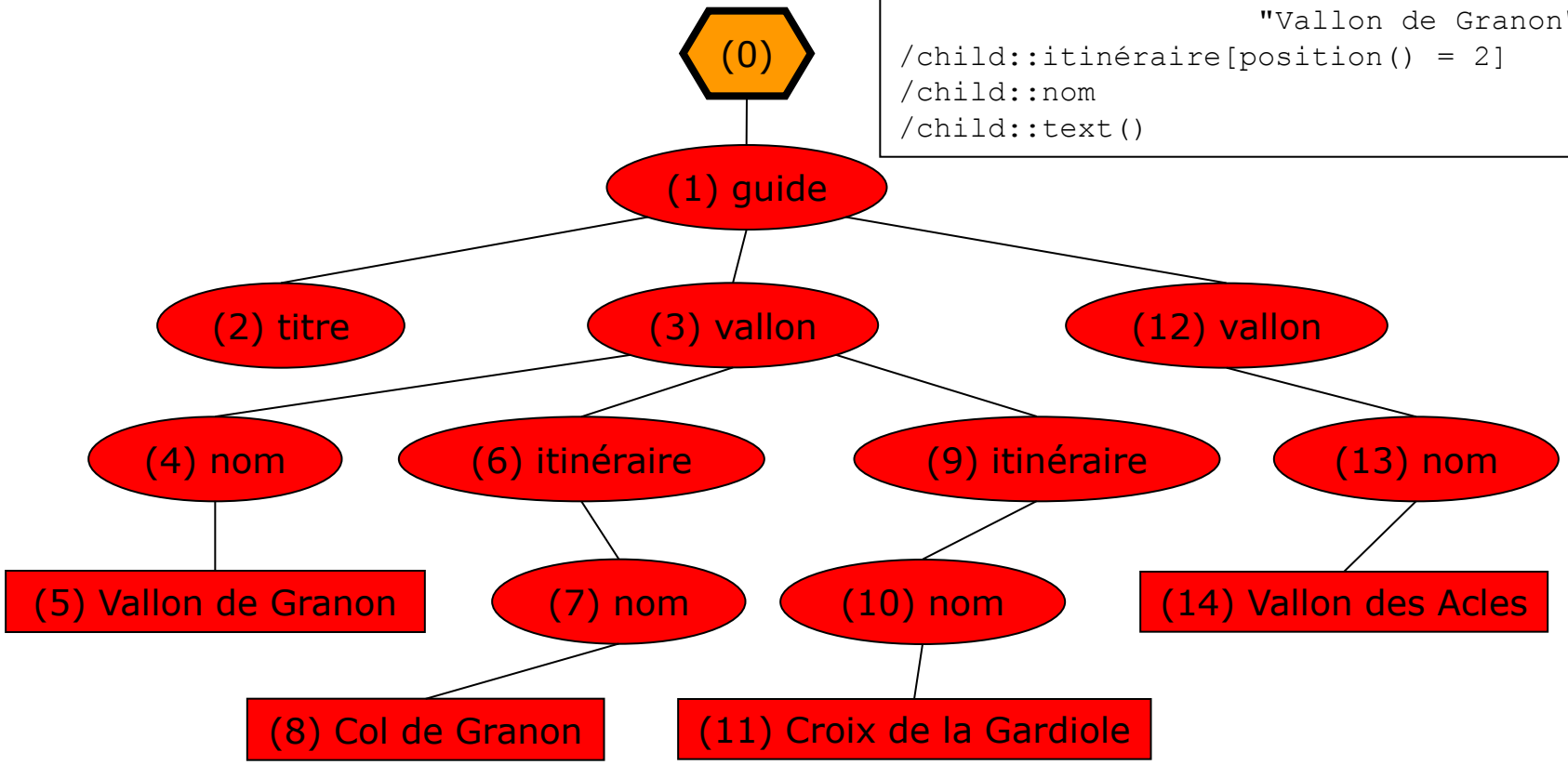
Evaluation d'une expression de chemin (1)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



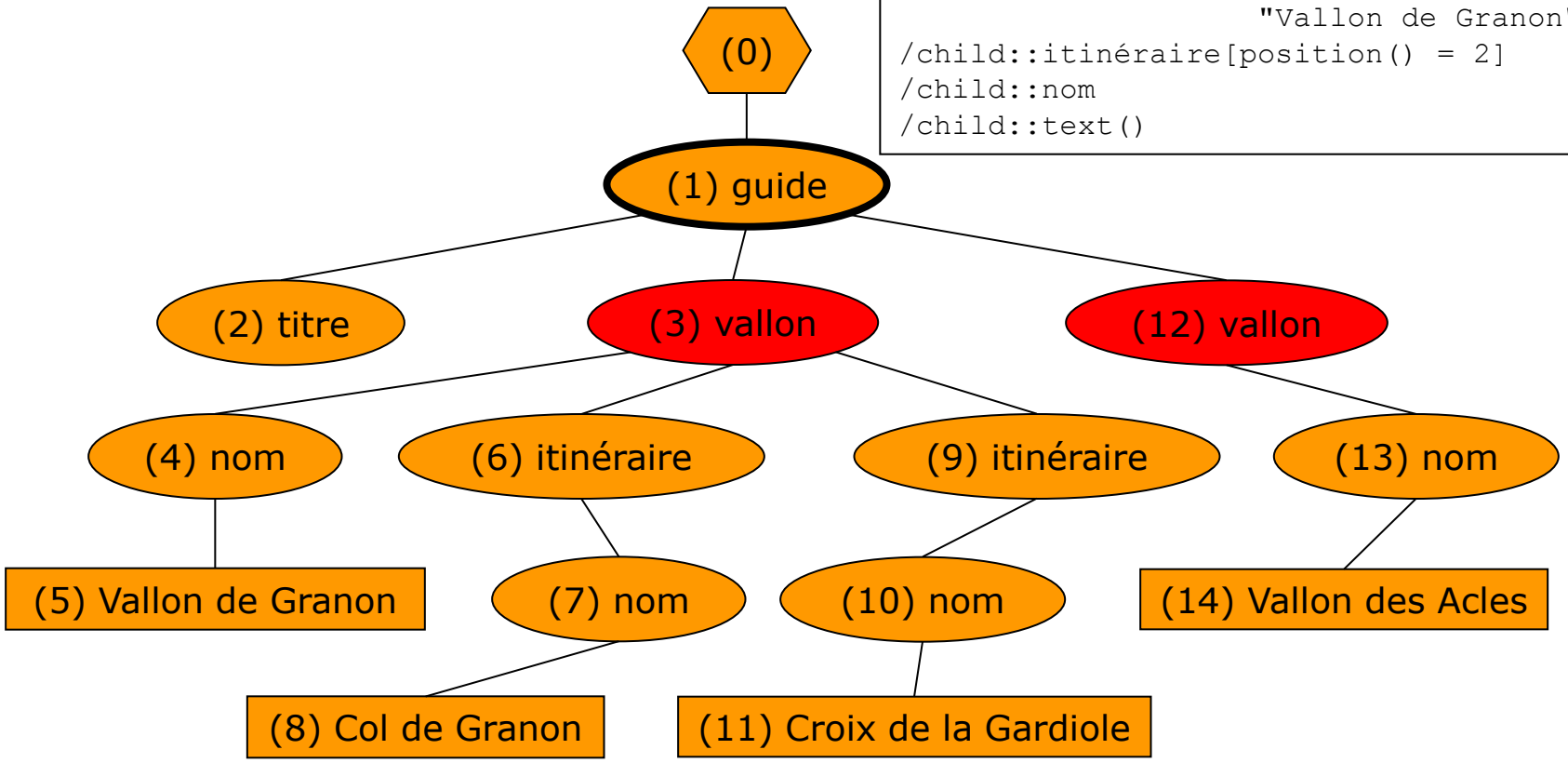
Evaluation d'une expression de chemin (2)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



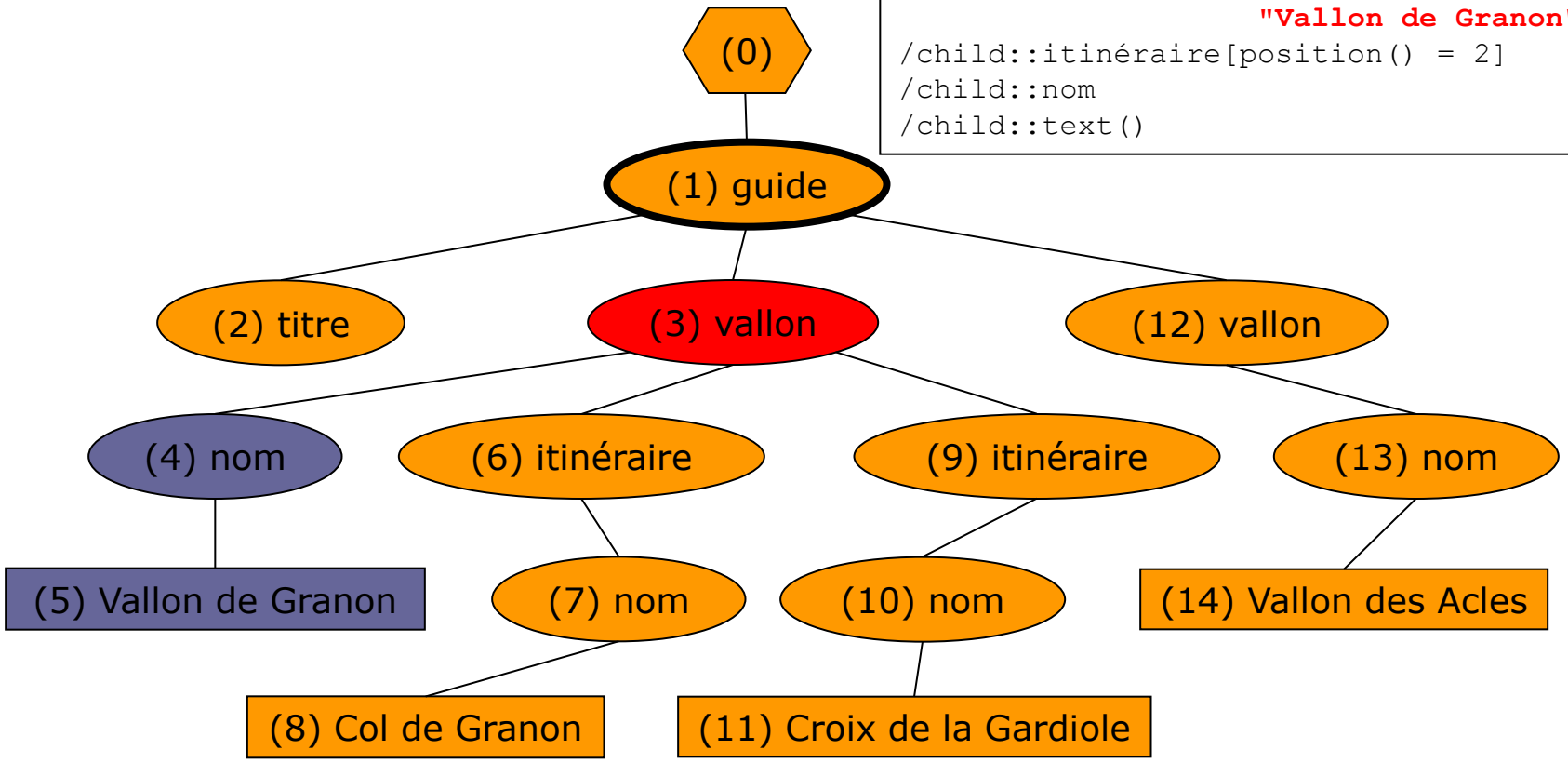
Evaluation d'une expression de chemin (3)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



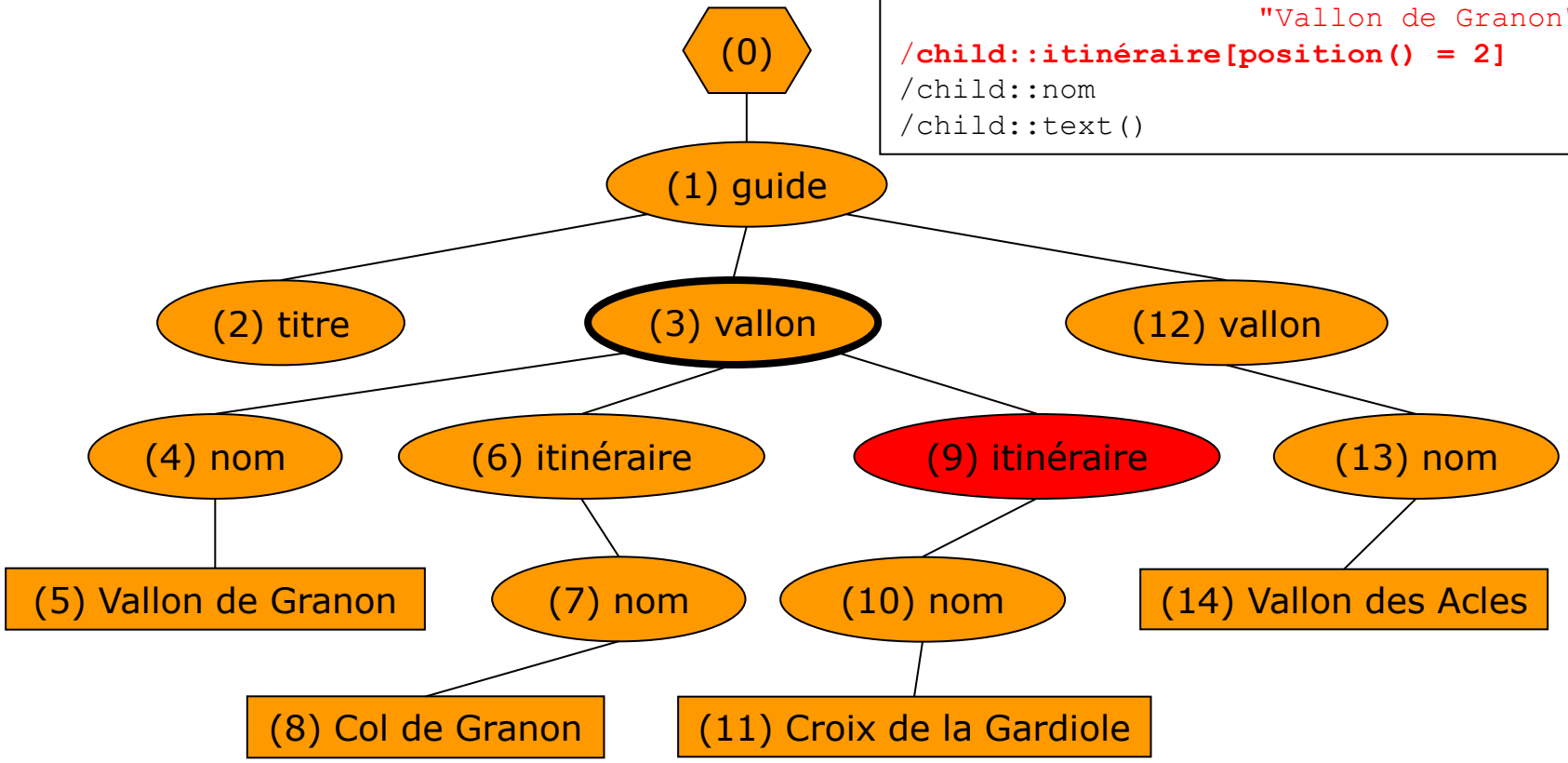
Evaluation d'une expression de chemin (4)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



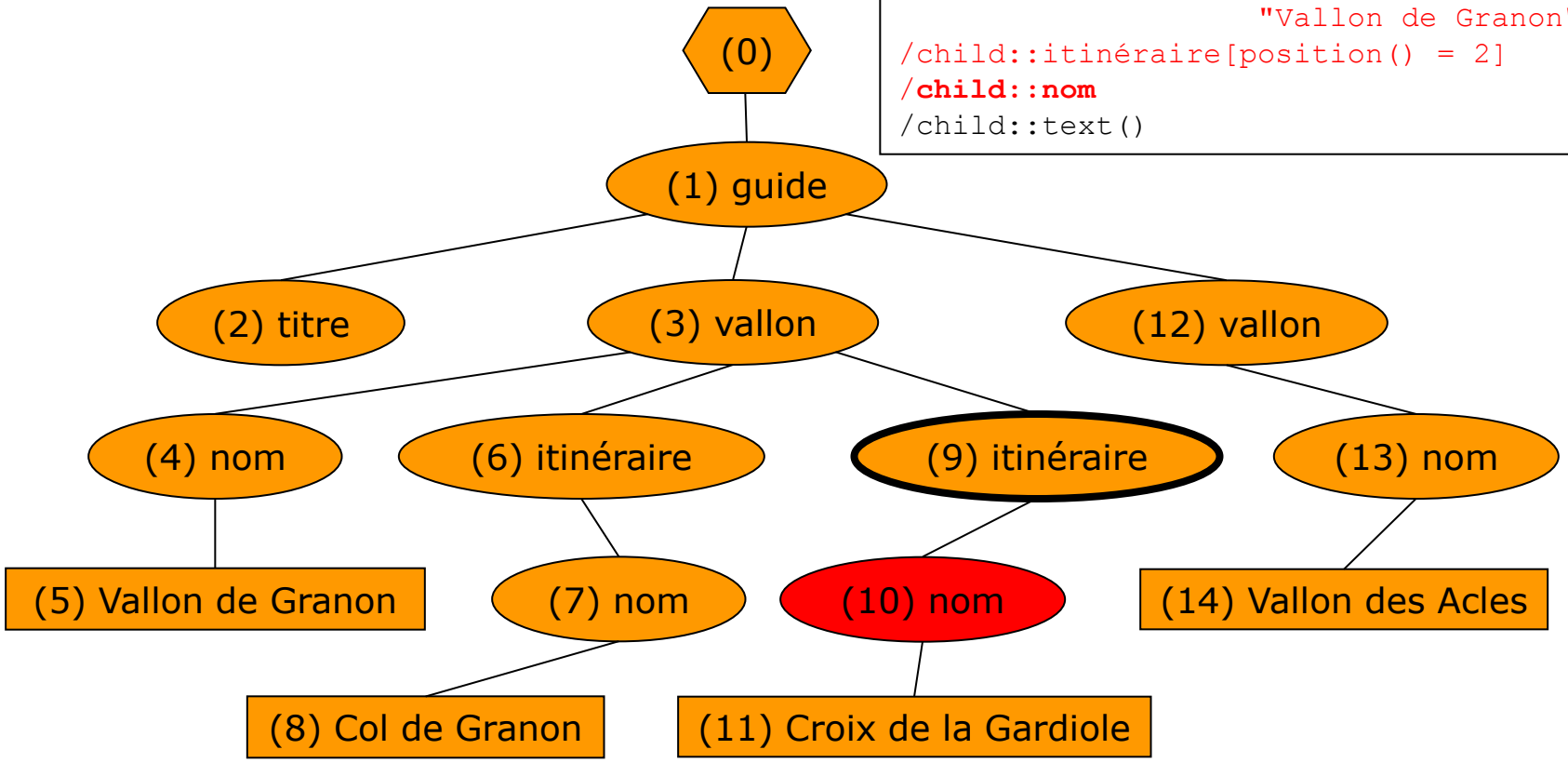
Evaluation d'une expression de chemin (5)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



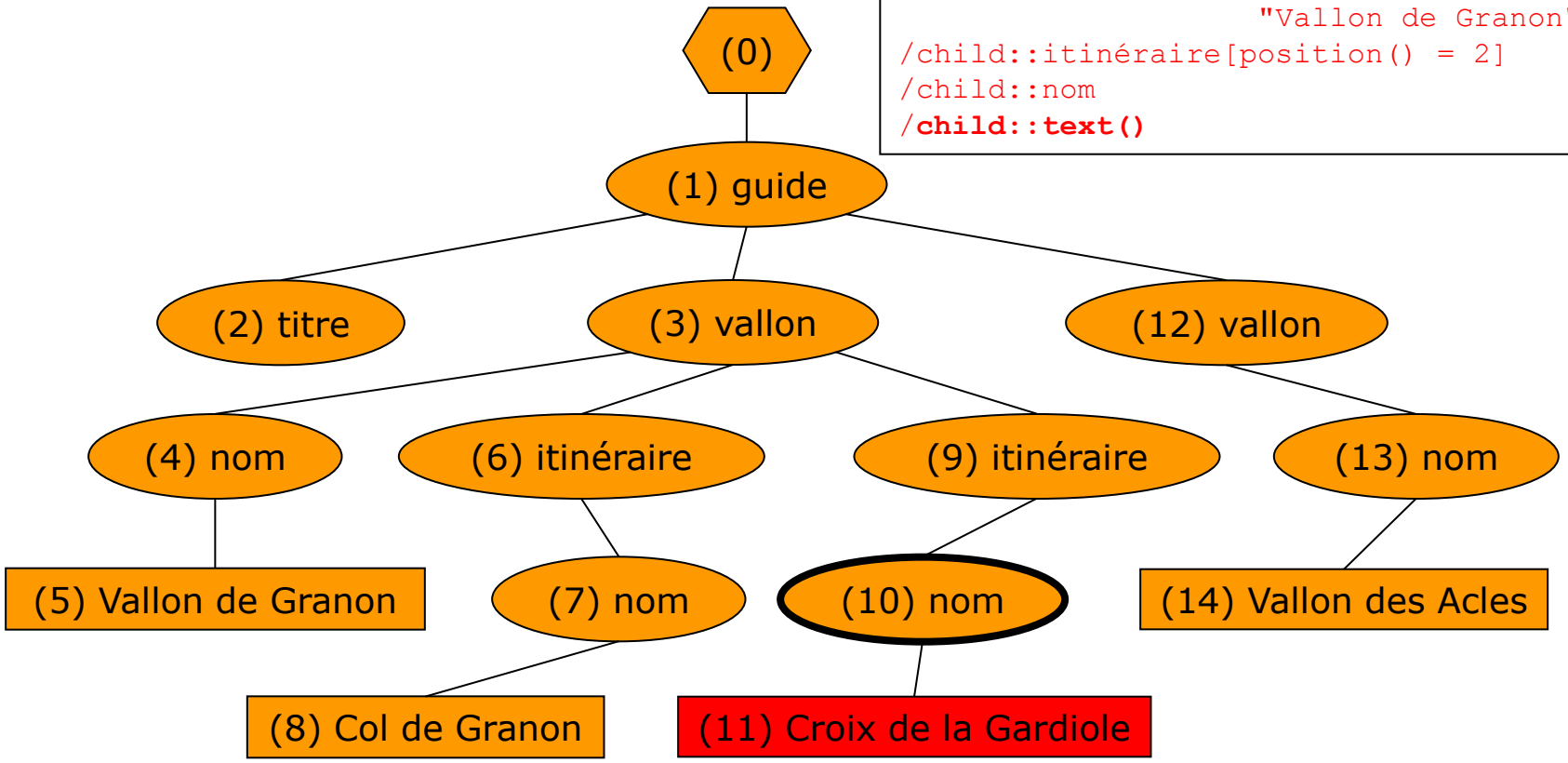
Evaluation d'une expression de chemin (6)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



Evaluation d'une expression de chemin (7)

```
fn:doc(clarée.xml)
/descendant::vallon[child::nom =
                    "Vallon de Granon"]
/child::itinéraire[position() = 2]
/child::nom
/child::text()
```



Expressions de chemins abrégées (1)

- /pas_2/…/pas_n
  - le 1er pas est `fn:root(self::node())` qui retourne le nœud racine de l'arbre auquel appartient le nœud contexte.
- //pas_2/…/pas_n
  - le 1er pas est :
    - `fn:root(self::node())/descendant-or-self::node()`

    qui retourne la séquence des nœuds de l'arbre auquel appartient le nœud contexte.
- pas_1/…//…/pas_n
  - le pas omis est `descendant-or-self::node()` qui retourne la séquence des nœuds du sous-arbre dont le nœud contexte est la racine.

Expressions de chemins abrégées (2)

- L'axe `child` est l'axe par défaut :
  - *t* est l'écriture abrégée de `child::`*t*
- `@` est l'écriture abrégée de `attribute::`
- `..` est l'écriture abrégée du pas de localisation `parent::node()`

Exemples d'expressions de chemin (1)

- *Nom des itinéraires du vallon des Muandes cotés ** et ne comportant pas de note de prudence.*
  - ```
    fn:doc(clarée.xml)
    //vallon[nom = "Vallon des Muandes"]
    /itinéraire[cotation = "**" and
                not (para/note[@type = "prudence"])]
    /nom
    /text()
    ```
- Nom des vallons possédant au moins deux itinéraires cotés **** ?
  - ```
    fn:doc(clarée.xml)
    //vallon[count(itinéraire[cotation = "****"]) > 1]
    /nom/text()
    ```

Exemples d'expressions de chemin (2)

- *Nom du 2e itinéraire ** du Vallon des Muandes ?*
  - ```
    fn:doc(clarée.xml)
    //vallon[nom = "Vallon des Muandes"]
    /itinéraire[cotation = "**"][2]
    /nom/text()
    ```

  Cet exemple montre l'intérêt des prédicats successifs lors de l'extraction d'un nœud par rapport à sa position de proximité. Ici on cherche le 2e des itinéraires **.
- *Noms des itinéraires du vallon des Muandes décrits après celui de la Pointe de Névache ?*
  - ```
    fn:doc(clarée.xml)
    //vallon[nom = "Vallon des Muandes"]
    /itinéraire[nom = "Pointe de Névache"]
    /following-sibling::itinéraire
    ```

Exemples d'expressions de chemin (3)

- *Nom du dernier vallon du guide ?*
  - `fn:doc(clarée.xml)//vallon[last()]/nom`
- *Textes de toutes les notes de type « prudence » ?*
  - `fn:doc(clarée.xml)//note[@type = "prudence"]/text()`
- *Nom du vallon dans lequel se trouve l'itinéraire I15.1 ?*
  - ```
    fn:doc(clarée.xml)
    //itinéraire[@id = "I15.1"]/parent::vallon/nom
    ```

  ou bien :
  - `fn:doc(clarée.xml)//itinéraire[@id = "I15.1"]/../nom`
- *Noms des itinéraires cités dans la description de l'itinéraire de la Pointe de Névache ?*
  - ```
    fn:doc(clarée.xml)
    //itinéraire[@id = fn:doc(clarée.xml)
                       //itinéraire[nom = "Pointe de Névache"]
                       //renvoi/@cible)]
    /nom
    ```

Combinaisons de séquences de nœuds

exp_1 `union` exp_2

exp_1 `|` exp_2

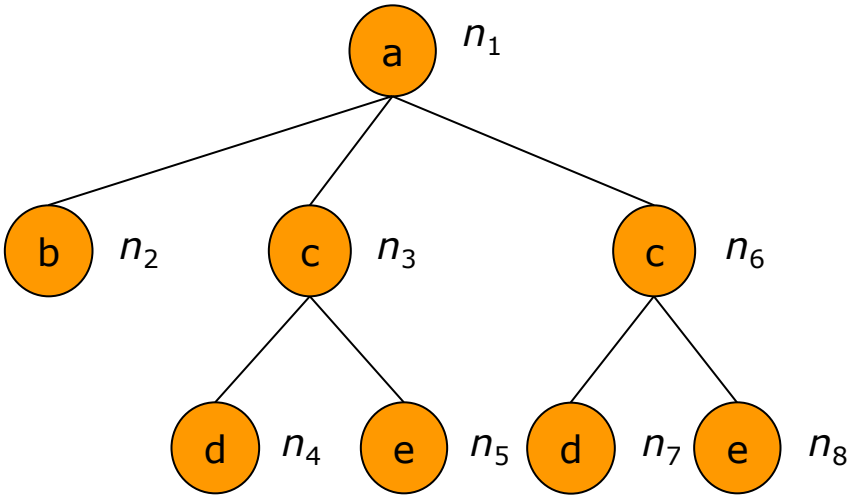
exp_1 `intersect` exp_2

exp_1 `except` exp_2

- Les valeurs des opérandes doivent être des séquences de nœuds, sinon une erreur est signalée.
- Dans la séquence retournée :
  - les doublons sont éliminés ;
  - les nœuds sont rangés dans l'ordre du document si le mode d'ordonnancement est `ordered`.

Exemples de combinaisons de séquences de nœuds(1)

- Si la variable `v` est liée à la racine du fragment :



- `$v/b union $v/c/*` $\Rightarrow n_2,\ n_4,\ n_5,\ n_7,\ n_8$
- `$v/c/d intersect $v/c/*` $\Rightarrow n_4,\ n_7$
- `$v/c/* except $v/c/d` $\Rightarrow n_5,\ n_8$

Exemples de combinaisons de séquences de nœuds (2)

- `fn:doc("clarée.xml")/(vallon | itinéraire)`

  ⇒ séquence des nœuds élément de type `vallon` ou `itinéraire` du document `clarée.xml`.

Expressions arithmétiques (1)

exp_1 + exp_2
exp_1 - exp_2
exp_1 * exp_2
exp_1 `div` exp_2
exp_1 `idiv` exp_2
exp_1 `mod` exp_2
+*exp*
-*exp*

- `idiv` et `mod` calculent le quotient et le reste d'une division entière.
- **Attention !** L'opérateur de division est `div` et non `/` qui est l'opérateur de parcours d'une séquence.

Expressions arithmétiques (2)

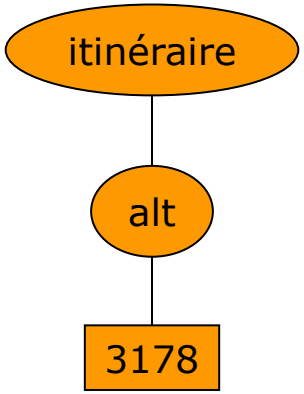
- Les opérateurs arithmétiques sont des opérateurs polymorphes. L'opération effectivement réalisée (sur les entiers ou sur les flottants) dépend du type des opérandes
- Chaque opérande est traité de la façon suivante :
  - Il est évalué puis atomisé.
  - Si la valeur produite est une valeur de type `xs:unTypedAtomic`, elle est convertie en `xs:double`.
- Si l'un des deux opérandes est une séquence vide, une séquence vide est retournée.
- Si les types des opérandes sont une combinaison valide pour l'opérateur considéré alors la valeur de l'opération est retournée.
- Sinon, une erreur est signalée.

Exemples d'expressions arithmétiques

- `3 + 4` $\Rightarrow$ 7
- `12.5 - 7` $\Rightarrow$ 5.5

Exemples d'expression arithmétique

- Si la variable `i` est liée à la racine du fragment non validé :



alors :

- `$i/alt * 3.28` ⇒ 10423.84 (altitude en pieds)

car :

- l'atomisation de *valeur*(`$i/alt`) produit la chaîne 3178 de type `untypedAtomic` qui est donc convertie en `xs:double`
- *valeur*(`3.28`) est de type `xs:decimal`. Elle est convertie en `xs:double`.
- l'opérateur `*` est applicable puisque ses deux opérandes sont de même type `xs:double`.

Expressions de comparaison

- Comparaisons de valeurs
- Comparaisons générales
- Comparaisons de nœuds

Comparaisons de valeurs (1)

exp_1 `eq` exp_2
exp_1 `ne` exp_2
exp_1 `lt` exp_2
exp_1 `le` exp_2
exp_1 `gt` exp_2
exp_1 `ge` exp_2

Comparaisons de valeurs (2)

- Comme les opérateurs arithmétiques, les opérateurs de comparaison de valeurs sont des opérateurs polymorphes.
- Chaque opérande est traité de la façon suivante :
  - Il est évalué puis atomisé.
  - Si la valeur produite est une valeur de type `xs:unTypedAtomic`, elle est convertie en `xs:string`.
- Si l'un des opérandes est une séquence vide, une séquence vide est retournée.
- Les opérandes sont convertis, si c'est possible, en leur plus petit type commun et si les types des opérandes sont une combinaison valide pour l'opérateur considéré alors la valeur de l'opération est retournée.
- Sinon, une erreur est signalée.

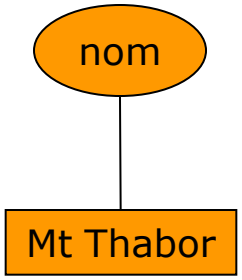
Exemples de comparaisons de valeurs

- `() eq 5` ⇒ ()
- `5 gt 7.3` ⇒ faux

  car :
  - *valeur*(`5`) = 5 de type `xs:integer`
  - *valeur*(`7.3`) = 7.3 de type `xs:decimal`
  - 5 est converti en `xs:decimal` qui est le plus petit type commun de `xs:integer` et `xs:decimal`
  - la comparaison peut alors être effectuée puisque les valeurs de opérandes sont de même type `xs:decimal`

Exemple de comparaison de valeurs

- Si la variable `n` est liée à la racine du fragment non validé :



alors :

- `$n eq "Mt Thabor"` ⇒ vrai

Comparaisons générales (1)

$exp_1 = exp_2$

$exp_1 != exp_2$

$exp_1 < exp_2$

$exp_1 <= exp_2$

$exp_1 > exp_2$

$exp_1 >= exp_2$

Comparaisons générales (2)

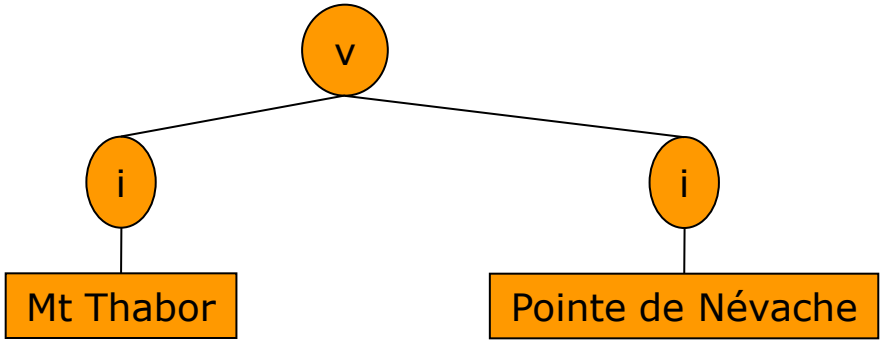
- Les opérandes sont évalués puis atomisés produisant les séquences S_1 et S_2.
- La comparaison retourne vrai, si et seulement si il existe $a_1 \in S_1$, $a_2 \in S_2$ pour lesquelles l'application des règles suivantes retourne vrai :
  - Si l'une des valeurs atomiques a_i est de type `xs:untypedAtomic` et l'autre a_j est de type numérique alors a_i est convertie en `xs:double`.
  - Si l'une des valeurs atomiques a_i est de type `xs:untypedAtomic` et l'autre a_j est de type `xs:untypedAtomic` ou `xs:string`, alors a_i est convertie en `xs:string` et a_j l'est aussi si elle est de type `xs:untypedAtomic`.
  - Si l'une des valeurs atomiques a_i est de type `xs:untypedAtomic` et l'autre a_j n'est ni de type `xs:untypedAtomic`, ni de type `xs:string`, ni d'un type numérique, alors a_i est convertie dans le type dynamique de a_j.
  - Les valeurs a_1 et a_2 sont comparées en appliquant l'un des opérateurs `eq`, `ne`, `lt`, `le`, `gt` ou `ge` selon que l'opérateur de comparaison générale est respectivement `=`, `!=`, `<`, `<=`, `>` ou `>=`.
- Sinon, elle retourne faux ou bien signale une erreur.

Exemples de comparaisons générales

- `5 != 7` ⇒ vrai
- `2 > (5, 9, 3)` ⇒ faux
- `(1, 2) = (7, 2, 9, 12)` ⇒ vrai
- `2 = ()` ⇒ faux

Exemple de comparaison générale

- Si la variable `v` est liée à la racine du fragment :



alors :

- `$v/i = "Mt Thabor"` ⇒ vrai

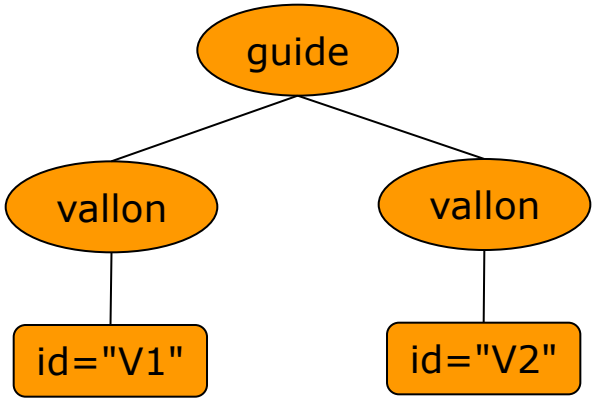
Comparaisons de nœuds

exp_1 `is` exp_2
exp_1 `<<` exp_2
exp_1 `>>` exp_2

- Les deux opérandes doivent avoir pour valeur soit une séquence vide, soit un nœud, sinon une erreur est signalée.
- Si l'un des opérandes a pour valeur une séquence vide, la valeur de la comparaison est la séquence vide.
- Si la valeur de exp_1 est le nœud n_1 et la valeur de exp_2 est le nœud n_2 :
  - exp_1 `is` exp_2 $\Rightarrow$ vrai, si les nœuds n_1 et n_2 sont les mêmes, faux sinon
  - exp_1 `<<` exp_2 $\Rightarrow$ vrai, si le nœud n_1 précède le nœud n_2 dans l'ordre du document, faux sinon
  - exp_1 `>>` exp_2 $\Rightarrow$ vrai, si le nœud n_1 suit le nœud n_2 dans l'ordre du document, faux sinon

Exemple de comparaison de nœuds

- Si la variable `g` est liée à la racine du fragment :



alors :

- `$g/vallon[@id = "V1"] << $g/vallon[@id= "V2"]`
  ⇒ vrai

Expressions logiques

exp_1 `and` exp_2

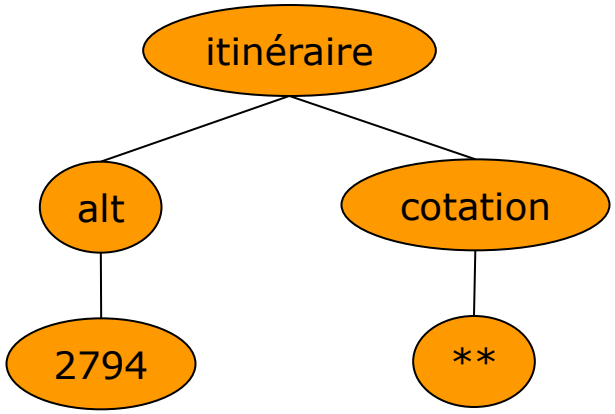
exp_1 `or` exp_2

`fn:not(`*exp*`)`

- exp_1 `and` exp_2 ⇒ conjonction des valeurs booléennes effectives de exp_1 et exp_2
- exp_1 `or` exp_2 ⇒ disjonction des valeurs booléennes effectives de exp_1 et exp_2
- `fn:not(`*exp*`)` ⇒ négation de la valeur booléenne effective de *exp*
- **Attention !** L'ordre d'évaluation des opérandes est dépendant de l'implantation. Si une erreur existe dans l'un des deux opérandes, elle ne sera signalée que si cet opérande est évalué en premier ou bien s'il est nécessaire de l'évaluer pour obtenir le résultat de l'opération.

Exemples d'expressions logiques

- `fn:not(1 and 1)` ⇒ faux
- Si la variable `i` est liée à la racine du fragment :



alors :

- `($i/cotation) = "**" and ($i/alt > 2700)` ⇒ vrai

Expression conditionnelle

`if` exp_1 `then` exp_2 `else` exp_3

- `if` exp_1 `then` exp_2 `else` exp_3 $\Rightarrow$
  - *valeur*(exp_2), si *valeur booléenne effective*(exp_1) = vrai (exp_3 n'est pas évaluée)
  - *valeur*(exp_3), si *valeur booléenne effective*(exp_1) = faux (exp_2 n'est pas évaluée)

Constructeurs de nœuds

- Un constructeur de nœud produit un nouveau nœud qui n'a pas de nœud parent.
- Il existe des constructeurs pour chaque sorte de nœud.
- On distingue :
  - les **constructeurs directs** ;
  - les **constructeurs calculés**.
- Un constructeur direct permet d'exprimer un nœud élément à construire selon une syntaxe calquée sur celle de l'élément correspondant écrit en XML.
- Un constructeur calculé permet de calculer tous les composants du nœud à construire : notamment son nom et les noms de ses attributs dans le cas d'un constructeur calculé de nœud élément.

Constructeur direct de nœud élément

$$<n_0\ \ n_1="v_1"\ \ \ldots\ \ n_k="v_k">c</n>$$

- n_0, n_1, ..., n_k sont des noms qualifiés.
- $n_1="v_1"$ … $n_k="v_k"$ sont soit des déclarations d'espace de noms soit des attributs.
- v_1, …, v_k peuvent contenir des caractères, des appels d'entités (prédéfinies ou caractères) ou bien des expressions XQuery placées entre accolades, appelées **expressions incluses**.
- *c* est le **contenu** du constructeur qui peut contenir des caractères, des appels d'entités (prédéfinies ou caractères), des constructeurs directs d'élément et des expressions incluses.
- Le nœud construit est un nouveau nœud élément dont :
  - le nom est obtenu en développant le nom qualifié n_0 à partir des espaces de noms déclarés dans l'environnement statique ou dans les nœuds de l'élément en cours de construction ;
  - les nœuds attributs sont ceux produits par l'évaluation des attributs de la balise ouvrante du constructeur et ceux produits par l'évaluation du contenu ;
  - les enfants sont les nœuds éléments ou texte produits par l'évaluation du contenu ;
  - le type est `xs:untyped` si le mode de construction est `strip`, `xs:anyType` si ce mode est `preserve`.

Évaluation des attributs

- Chaque attribut *n* = "*v*"de la balise ouvrante d'un constructeur direct d'élément produit un nouveau nœud attribut dont :
  - Le nom est obtenu en développant le nom qualifié *n* à partir des espaces de noms déclarés dans l'environnement statique ou dans l'élément en cours de construction.
  - La valeur est obtenue en traitant *v* de la façon suivante :
    - Les appels d'entités sont remplacés par l'entité désignée.
    - Chaque suite consécutive de caractères est remplacée par une chaîne de caractères formée par cette suite de caractères.
    - Chaque expression incluse est convertie en une chaîne de caractères de la façon suivante :
      - l'atomisation est appliquée produisant une séquence de valeurs atomiques *S* ;
      - chaque valeur atomique de *S* est convertie en une chaîne de caractères ;
      - les chaînes ainsi produites sont concaténées en une chaîne unique.
    - Les chaînes de caractères ainsi produites sont concaténées en une chaîne unique qui constitue la valeur de l'attribut.
  - Le type est `untypedAtomic`.

Espaces frontières

- On appelle **espace frontière** dans le contenu d'un élément les suites de caractères blancs (espace, tabulation, retour chariot ou saut de ligne) dont chaque extrémité est marquée :
  - soit par le début ou la fin de ce contenu ;
  - soit par le début ou la fin d'un constructeur direct d'élément ;
  - soit par une expression incluse.
- Par exemple, les espaces frontières des éléments suivants sont soulignés :
  - `<a>___<b/>___<c/>___</a>`
  - `<phrase>1 + 1 = {1 + 1}___</phrase>`

Évaluation du contenu (1)

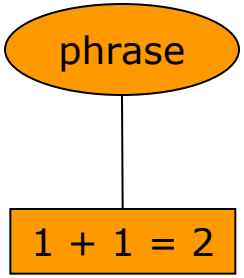
- Le contenu *c* est évalué en une séquence de nœuds produite de la façon suivante :
  - Si le mode de traitement des espaces frontières est `strip`, les caractères blancs qui les composent sont supprimés.
  - Les appels d'entités sont remplacés par l'entité désignée.
  - Chaque suite consécutive de caractères est remplacée par un nouveau nœud texte dont le contenu est la chaîne de caractères formée par cette suite de caractères.
  - Chaque constructeur direct de nœud est évalué produisant un nouveau nœud.
  - Chaque expression incluse est évaluée produisant une séquence qui est traitée de la façon suivante :
    - chaque valeur atomique est convertie en une chaîne de caractères ;
    - chaque suite consécutive de chaînes de caractères est remplacée par un nouveau nœud texte dont le contenu est la concaténation de ces chaînes de caractères séparées par un espace ;
    - chaque nœud est remplacé par un nouveau nœud qui est sa copie profonde.

Évaluation du contenu (2)

- Dans la séquence ainsi produite :
  - Chaque nœud document est remplacé par ses enfants.
  - Les nœuds texte consécutifs sont remplacés par un nœud texte unique contenant la concaténation des contenus de ces nœuds texte. Si le nœud produit a un contenu qui est une chaîne vide, il est supprimé de la séquence.
  - Si un nœud attribut suit un nœud qui n'est pas un attribut alors une erreur est signalée.
  - Chaque nœud élément reçoit le type `xs:untyped` si le mode de construction est `strip` ou le type du nœud copié, si ce mode est `preserve`.
  - Chaque nœud attribut reçoit le type `xs:untypedAtomic`.

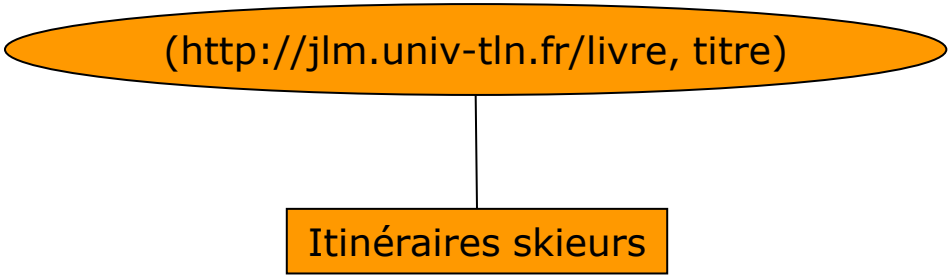
Exemple de constructeur direct de nœud élément

- `<phrase>1 + 1 = {1 + 1}</phrase>` ⇒



Exemple de constructeur direct de nœud élément

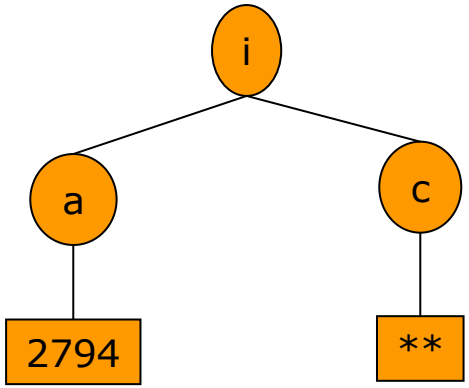
- <titre xmlns:livre="http://jlm.univ-tln.fr/livre">
  {"Itinéraires", "skieurs"}
  </titre> ⇒



- On remarquera :
  - qu'il n'y a pas de nœud espace de nom de créé ;
  - qu'un espace est inséré entre Itinéraires et skieurs !

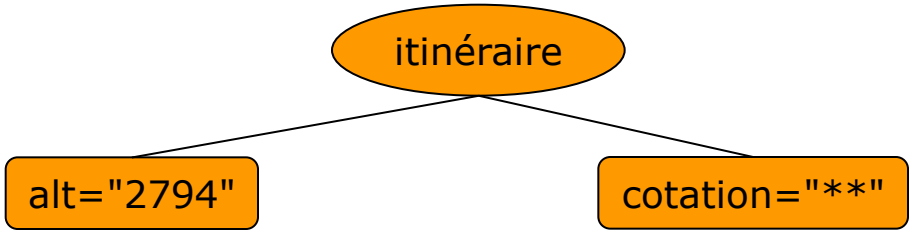
Exemple de constructeur direct de nœud élément

- Si la variable `i` est liée à la racine du fragment :



alors :

- `<itinéraire alt="{$i/a}" cotation="{$i/c}"/>` ⇒



Constructeurs calculés de nœud élément

`element` *n* `{`exp_c`}`

`element {`exp_{nom}`} {`exp_c`}`

- *n* est un nom qualifié.
- Le nœud construit est un nouveau nœud élément dont :
  - Le nom est obtenu :
    - soit en développant le nom qualifié *n* à partir des espaces de noms déclarés dans l'environnement statique ou dans l'élément en cours de construction ;
    - soit en évaluant l'expression exp_{nom}, en l'atomisant, puis en la convertissant si possible en un nom qualifié qui est ensuite développé.
  - La séquence des nœuds fils est produite en traitant l'expression exp_c de la même façon qu'une expression incluse dans le contenu d'un constructeur direct d'élément.
  - Le type est `xs:untyped` si le mode de construction est `strip`, `xs:anyType` si ce mode est `preserve`.

Constructeurs calculés de nœud attribut

`attribute` *n* {exp_v}

`attribute` {exp_{nom}} {exp_v}

- *n* est un nom qualifié.
- Le nœud construit est un nouveau nœud attribut dont :
  - Le nom est obtenu :
    - soit en développant le nom qualifié *n* à partir des espaces de noms déclarés dans l'environnement statique ou dans l'élément en cours de construction ;
    - soit en évaluant l'expression exp_{nom}, en l'atomisant, puis en la convertissant si possible en un nom qualifié qui est ensuite résolu.
  - La valeur de l'attribut est produite en traitant l'expression exp_v de la même façon qu'une expression incluse dans la valeur d'un attribut d'un constructeur direct d'élément.
  - Le type est `untypedAtomic`.

Constructeurs calculés de nœud document

`document {`*exp*`}`

- Le nœud construit est un nouveau nœud document dont la séquence des nœuds fils est produite en traitant l'expression *exp* de la même façon qu'une expression incluse dans un constructeur direct de nœud élément, excepté le fait que si un nœud attribut est produit une erreur est signalée.

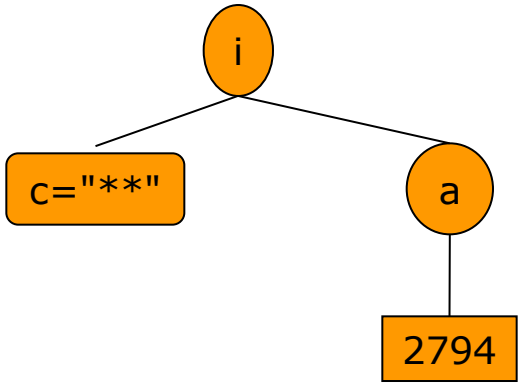
Constructeur calculé de nœud texte

`text {`*exp*`}`

- Le nœud construit est un nouveau nœud texte dont le contenu est obtenu de la façon suivante :
  - l'expression *exp* est évaluée puis atomisée,
  - si la séquence produite est vide, le nœud texte n'est pas construit, sinon les valeurs atomiques de cette séquence sont converties en chaînes de caractères qui sont concaténées en les séparant les unes des autres par un espace.

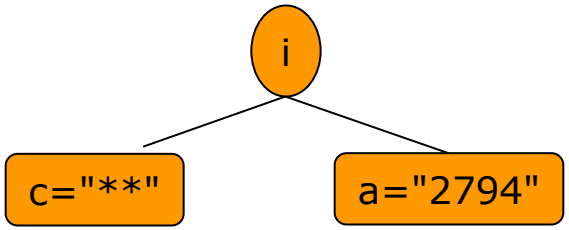
Exemple de constructeurs calculés de nœuds

- Si la variable `i` est liée à la racine du fragment :



alors :

- `element {fn:name($i)}`
  `{$i/@c, attribute {fn:name($i/a)} {$i/a}}` ⇒



Expression FLWOR (1)

- L'expression FLWOR est à XQuery ce que `select...from...where...order by...` est à SQL.
- Elle permet :
  - d'itérer sur des séquences ;
  - de créer des liaisons temporaires variable-valeur ;
  - de filtrer des séquences ;
  - de trier des séquences ;
  - et plus généralement, de restructurer un ensemble de fragments XML.

Expression FLWOR (2)

(clause `for` | clause `let`)+
(clause `where`)?
(clause `order by`)?
clause `return`

- Une expression FLWOR doit comporter au moins une clause `for` ou une clause `let` et une clause `return`.
- La clause `for` permet d'itérer sur une séquence.
- La clause `let` permet de créer une liaison temporaire.
- La clause `where` permet le filtrage.
- La clause `order by` spécife un ordre pour les items retournés.
- La clause `return` fournit l'expression qui permet de calculer les items retournés.

Exemple d'expression FLWOR

- *Nombres entiers impairs compris entre 1 et 10 triés par ordre décroissant ?*
  - ```
    let $n1 := 1, $n2 := 10
    for $i in $n1 to $n2
    where ($i mod 2) = 1
    order by $i descending
    return $i
    ⇒ 9, 7, 5, 3, 1
    ```
    ```

Exemple d'expression FLWOR

- *Noms et altitude de chaque itinéraire* **** *?*
  - `for $i in fn:doc("clarée.xml")//itinéraire`
    `where $i/cotation = "****"`
    `return <itinéraire nom="{$i/nom}"`
    `alt="{$i/alt}"/>`
- Évaluation (la séquence *R* est retournée) :
  - *R* := ()
  - Pour chaque item *i* de la séquence *valeur*(`fn:doc(clarée.xml)//itinéraire`) :
    - Lier *i* à la variable `i`
    - Si *valeur booléenne effective*(`$i/cotation = "****"`) = vrai alors :
      - *R* := *R*, *valeur*(`<itinéraire nom="{$i/nom}" alt="{$i/alt}"/>`)

Exemple d'expression FLWOR

- *Nom et altitude des itinéraires triés par altitude décroissante ?*
  - `for $i in fn:doc("clarée.xml")//itinéraire`
    `let $a := $i/alt`
    `order by $a descending`
    `return <itinéraire nom="{$i/nom}" alt="{$a}"/>`
- Évaluation (la séquence R est retournée) :
  - *R* := ()
  - *T* := liste vide
  - Pour chaque item *i* de la séquence *valeur*`(fn:doc(clarée.xml)//itinéraire)` :
    - Lier *i* à la variable `i`
    - Lier *valeur*(`$i/alt`) à la variable `a`
    - Ajouter la paire (*valeur*(`$a`), *valeur*(`<itinéraire nom="{$i/nom}" alt="{$a}"/>`)) à *T*
  - Trier les paires (*c*, *v*) de *T* par valeur décroissante de *c*.
  - Pour chaque paire (*c*, *v*) de *T* :
    - *R* := *R*, *v*

Exemple d'expression FLWOR

- *Itinéraires, triés par difficulté décroissante puis par ordre alphabétique de nom ?*
  - `for $i in fn:doc("clarée.xml")//itinéraire order by $i/cotation descending, $i/nom ascending return $i`
- Évaluation (la séquence R est retournée) :
  - *R* := ()
  - *T* := liste vide
  - Pour chaque item *i* de la séquence *valeur*(`fn:doc(clarée.xml)//itinéraire`) :
    - Lier *i* à la variable `i`
    - Ajouter le triplet (*valeur*(`$i/cotation`), *valeur*(`$i/nom`), *valeur*(`$i`)) à *T*
  - Trier les triplets (c_1, c_2, *v*) de *T* par valeur décroissante de c_1, puis par valeur croissante de c_2
  - Pour chaque triplet (c_1, c_2, *v*) de *T* :
    - *R* := *R*, *v*

Exemple d'expression FLWOR

- *Liste des paires nom de vallon-nom d'itinéraire ?*
  - `for $v in fn:doc("clarée.xml")//vallon`
    `for $i in $v/itinéraire`
    `return <vi vallon="{$v/nom}" itinéraire="{$i/nom}"/>`
- Évaluation (la séquence R est retournée) :
  - *R* := ()
  - Pour chaque item *v* de la séquence *valeur*(`fn:doc("clarée.xml")//vallon`) :
    - Lier *v* à la variable `v`
    - Pour chaque item *i* de la séquence valeur(`$v/itinéraire`) :
      - Lier *i* à la variable `i`
      - *R* := *R*, *valeur*(`<vi vallon="{$v/nom}" itinéraire="{$i/nom}"/>`)

Exemple d'expression FLWOR

- *Rang et le nom de chaque vallon ?*
  - `for $v at $r in fn:doc("clarée.xml")//vallon return <vallon rang="{$r}" nom="{$v/nom}"/>`
- Évaluation (la séquence R est retournée) :
  - *R* := ()
  - Pour chaque item *v* de rang *r* de la séquence *valeur*(`fn:doc("clarée.xml")//vallon`) :
    - Lier *v* à la variable `v`
    - Lier *r* à la variable `r`
    - *R* := *R*, *valeur*(`<vallon rang="{$r}" nom="{$v/nom}"/>`)

Clause `for`

$$\texttt{for } \$v_1 \texttt{ as } t_1 \texttt{ at } \$rang_1 \texttt{ in } exp_1,$$
$$\ldots,$$
$$\$v_n \texttt{ as } t_n \texttt{ at } \$rang_n \texttt{ in } exp_n$$
$$\equiv$$
$$\texttt{for } \$v_1 \texttt{ as } t_1 \texttt{ at } \$rang_1 \texttt{ in } exp_1$$
$$\ldots$$
$$\texttt{for } \$v_n \texttt{ as } t_n \texttt{ at } \$rang_n \texttt{ in } exp_n$$

- v_1, $rang_1$, ..., v_n, $rang_n$ sont des noms de variables.
- t_1, ..., t_n sont des types de séquence.
- Les clauses `as` et `at` sont facultatives.
- Lors du balayage d'une séquence exp_i, l'item courant est lié à la variable v_i et le rang de cet item à la variable $rang_i$, si la clause `at` est présente.
- L'item lié à une variable v_i doit être conforme au type t_i, si la clause `as` est présente.

Clause `let`

$$\texttt{let } \$v_1 \texttt{ as } t_1 \texttt{ := } exp_1, \ldots, \texttt{ } \$v_n \texttt{ as } t_n \texttt{ := } exp_n$$

$$\equiv$$

$$\texttt{let } \$v_1 \texttt{ as } t_1 \texttt{ := } exp_1$$

$$\ldots$$

$$\texttt{let } \$v_n \texttt{ as } t_n \texttt{ := } exp_n$$

- v_1, ..., v_n sont des noms de variables.
- t_1, ..., t_n sont des types de séquence.
- La clause `as` est facultative.
- La valeur liée à une variable v_i doit être conforme au type t_i, si la clause `as` est présente.

Clause `order by`

`order by` (`stable`)? *clé*$_1$, ..., *clé*$_n$

- *clé*$_1$, ..., *clé*$_n$ sont les clés de tri.
- `stable` précise l'ordre dans lequel placer deux paires de clés égales.
- Chaque clé de tri a la forme suivante :
  - *exp*
    (`ascending` | `descending`)?
    (`empty greatest` | `empty least`)?
    (`collation` *URI*)?

  où :
  - *exp* est l'expression qui calcule la valeur de cette clé ;
  - `ascending` ou `descending` indique le sens du tri pour cette clé ;
  - `empty least` ou `empty greatest` indique comment comparer une séquence vide avec une autre valeur ;
  - une collation spécifie la manière dont deux valeurs de cette clé doivent être comparées.

Sémantique de l'expression FLWOR

- La valeur d'une expression FLWOR *exp* est la séquence affectée à *R* après exécution des actions suivantes :
  - *R* := ()
  - *T* := *liste vide*
  - *évaluer-flwor*(*exp*) :
    - Si la clause `order by` est présente, *T* contient une liste d'éléments (c_1,..., c_n, *valeur*).
  - Si la clause `order by` est absente, *R* contient la séquence recherchée.
  - Si la clause `order by` est présente alors :
    - *T'* := *trier*(*T*)
    - Pour chaque élément (c_1, ..., c_n, *v*) de *T'* :
      - *R* := *R*, *v*

évaluer-flwor (1)

- On note :
  - (*var* = *val*)::*Env* l'ajout d'une liaison variable-valeur au contexte dynamique *Env*,
  - *L* @ *e* l'ajout d'un élément *e* en fin d'une liste.

évaluer-flwor (2)

- *évaluer-flwor*(`for $`*v* `at $`*rang* `in` *exp Suite*, *Env*) →
  - pour chaque item *i* de rang *r* dans *valeur*(*exp*, *Env*)
    - *évaluer-flwor*(*Suite*, (*v* = *i*)::(*rang* = *r*)::*Env*)
- *évaluer-flwor*(`let $`*v* `:=` *exp Suite*, *Env*) →
  - *évaluer*(*Suite*, (*v* = *valeur*(*exp*, *Env*))::*Env*)
- *évaluer-flwor*(`where` *exp Suite*, *Env*) →
  - si *valeur-booléenne-effective*(*exp*, *Env*) = vrai alors
    - *évaluer-flwor*(*Suite*, *Env*)
- *évaluer-flwor*(`order by` *clé*$_1$, ..., *clé*$_n$ `return` *exp*, *Env*) →
  - *T* := *T* @ (*valeur-clé*(*clé*$_1$, *Env*), ..., *valeur*(*clé*$_n$, *Env*), *valeur*(*exp*, *Env*))
- *évaluer-flwor*(`return` *exp*, *Env*) →
  - *R* := *R*, *valeur*(*exp*, *Env*)

Évaluation des clés de tri

- La valeur d'une clé de tri d'expression *exp*, est calculée de la façon suivante :
  - *exp* est évaluée ;
  - la valeur produite est atomisée ;
  - si la valeur produite est de type `xs:untypedAtomic`, elle est convertie en `xs:string`.
- Toutes les valeurs d'une clé de tri, doivent pouvoir être converties en un plus petit type commun sur lequel l'opérateur `gt` est défini.

Relation d'ordre entre clés de tri (1)

- Soit c_{i1} et c_{i2} deux valeurs d'une même clé de tri.
- On a $c_{i1} > c_{i2}$ si :
  - Lorsque empty least est spécifié :
    - $c_{i1} \neq ()$ et $c_{i2} = ()$ ou
    - $c_{i1} \neq$ NaN et $ci1 \neq ()$ et $c_{i2} =$ NaN
  - Lorsque empty greatest est spécifié :
    - $c_{i1} = ()$ et $c_{i2} \neq ()$ ou
    - $c_{i1} =$ NaN et $ci2 \neq ()$ et $c_{i2} \neq$ NaN
  - aucune collation n'est spécifiée et c_{i1} gt c_{i2} ou
  - une collation est spécifiée et c_{i1} est plus grande que c_{i2} selon cette collation.

(NaN, Not a Number, est le résultat d'une opération arithmétique inapplicable, par exemple, une division par 0)

Relation d'ordre entre clés de tri (2)

- On note :
  - $c_{i1} = c_{i2}$, le fait que ni $c_{i1} > c_{i2}$, ni $c_{i2} > c_{i1}$
  - $c_{i1} \neq c_{i2}$, le fait que $c_{i1} > c_{i2}$ ou $c_{i2} > c_{i1}$
- Soit :
  - $e_1 = (c_{11}, \ldots, c_{1k}, \ldots, c_{1n}, v_1)$
  - $e_2 = (c_{21}, \ldots, c_{2k}, \ldots, c_{2n}, v_2)$

  deux éléments de la liste à trier :
  - si $c_{11} = c_{21} \wedge \ldots \wedge c_{1(k\text{-}1)} = c_{2(k\text{-}1)} \wedge c_{1k} \neq c_{2k}$ $(k \leq n)$ alors e_1 sera placée avant e_2 dans la liste triée, si :
    - soit `ascending` est spécifiée pour la k^e clé et $c_{2k} > c_{1k}$
    - soit `descending` est spécifiée pour la k^e clé et $c_{1k} > c_{2k}$
  - Si $c_{11} = c_{21} \wedge \ldots \wedge c_{1n} = c_{2n}$ alors l'ordre de e_1 et e_2 dans la liste à trier sera préservé si `stable` est spécifié sinon cette préservation dépendra de l'implantation.

Exemple d'expression FLWOR complexe : *Cols de la Vallée de la Clarée* (1)

- Il s'agit d'extraire la séquence des cols de la vallée de la Clarée, avec pour chacun :
  - son nom ;
  - son altitude ;
  - le nombre d'itinéraires permettant de l'atteindre ;
  - la liste de ces itinéraires, avec pour chacun :
    - l'identifiant,
    - le nom du vallon de départ,
    - la difficulté ;

    triée par difficulté croissante ;

  triée par ordre alphabétique de nom.
- On suppose qu'un itinéraire aboutit à un col si son nom commence par « Col ». Pour le tester, on utilisera la fonction `fn:starts-with`.
- On utilise la fonction prédéfinie `fn:distinct-values` qui appliquée à une séquence de valeurs atomiques retourne la séquence de ces valeurs dans le même ordre, mais sans doublons.

Exemple d'expression FLOWR complexe : *Cols de la Vallée de la Clarée* (2)

```
let $itis := fn:doc("clarée.xml")//itinéraire
for $n in fn:distinct-values($itis/nom[fn:starts-with(., "Col ")])
order by $n
return
 let $itis-n := $itis[nom = $n]
 return
  <col>
   {$itis-n[1]/nom}
   {$itis-n[1]/alt}
   <nb-itinéraires>{fn:count($itis-n)}</nb-itinéraires>
   {
   for $i in $itis-n
   let $c := $i/cotation
   order by $c descending
   return
    <itinéraire id="{$i/@id}"
                vallon="{$i/parent::vallon/nom}"
                cotation="{$c}"/>
   }
  </col>
```

Exemple d'expression FLOWR complexe : *Cols de la Vallée de la Clarée* (3)

⇒

…

```
<col>
<nom>Col de Granon</nom>
<alt>2404</alt>
<nb-itinéraires>1</nb-itinéraires>
<itinéraire id="I1.1" vallon="Vallon de Granon" cotation="*"/>
</col>,
<col>
<nom>Col de Lenlon</nom>
<alt>2500</alt>
<nb-itinéraires>2</nb-itinéraires>
<itinéraire id="I1.6" vallon="Vallon de Granon" cotation="**"/>
<itinéraire id="I3.1" vallon="Vallon du Creuset"
cotation="***"/>
</col>,
```

…

Expressions quantifiées

some $$v_1$$ as $$t_1$$ in $$exp_1$$, …, $$v_n$$ as $$t_n$$
in $$exp_n$$ satisfies $$exp$$

every $$v_1$$ as $$t_1$$ in $$exp_1$$, …, $$v_n$$ as $$t_n$$
in $$exp_n$$ satisfies $$exp$$

- Les expressions quantifiées permettent de tester si un prédicat est vrai pour un (`some`) ou pour chaque (`every`) n-uplet d'une séquence de *n*-uplets d'items.
- v_1, ..., v_n sont des noms de variables.
- t_1, ..., t_n sont des types de séquence.
- La valeur liée à une variable v_i doit être conforme au type t_i si ce type est spécifié.

Sémantique des expressions quantifiées

- *valeur*(`some $`*v* `in` *exp Suite*, *Env*) =
  - *valeur-some* (`$`*v* `in` *exp Suite*, *Env*)
- *valeur-some*(`$`*v* `in` *exp Suite*, *Env*) =
  - ∃*i* ∈ *valeur*(*exp*, *Env*) (*valeur-some*(*Suite*, (*v* = *i*)::*Env*) = vrai)
- *valeur-some*(`satisfies` *exp*, *Env*) =
  - *valeur booléenne effective*(*exp*, *Env*)

- *valeur*(`every $`*v* `in` *exp Suite*, *Env*) =
  - *valeur-every* (`$`*v* `in` *exp Suite*, *Env*)
- *valeur-every*(`$`*v* `in` *exp Suite*, *Env*) =
  - ∀*i* ∈ *valeur*(*exp*, *Env*) (*valeur-every*(*Suite*, (*v* = *i*)::*Env*) = vrai)
- *valeur-every*(`satisfies` *exp*, *Env*) =
  - *valeur booléenne effective*(*exp*, *Env*)

Exemples d'expressions quantifiées

- *Est-ce que tous les itinéraires ont une altitude supérieure à 2000m ?*
  - `every $i in fn:doc("clarée.xml")//itinéraire satisfies $i/alt > 2000`
- *Existe-t-il des itinéraires de même altitude ?*
  -
```
let $itis := fn:doc("clarée.xml")//itinéraire
return
 some $i1 in $itis, $i2 in $itis
 satisfies fn:not($i1 is $i2) and
           $i1/alt = $i2/alt
```

Expressions sur les types de séquence

- Test d'appartenance à un type
- Conversion de type
- Test de convertibilité
- Contrainte sur le type dynamique d'une expression
- Choix

Test d'appartenance à un type

exp `instance of` *t*

- L'opérateur `instance of` teste si une valeur est une instance d'un type donné.
- *exp* `instance of` *t* ⇒
  - vrai, si *valeur*(*exp)* est une instance du type *t*
  - faux, sinon
- Par exemple :
  - `<jlm/> instance of element(jlm)` ⇒ vrai
  - `3.14 instance of xs:decimal` ⇒ vrai
  - `(1, 2, 3) instance of xs:integer*` ⇒ vrai
  - `() instance of xs:integer?` ⇒ vrai
  - `(1, 2, 3) instance of xs:integer+` ⇒ vrai

Conversion de type

exp `cast as` t_a

exp `cast as` t_a`?`

- L'opérateur `cast as` convertit une valeur atomique en une valeur d'un type atomique donné.
- t_a est un type atomique.
- Soit *v* la valeur atomisée de *exp* et $t = t_a$ ou $t = t_a$`?`
  - *exp* `cast as` $t \Rightarrow$
    - (), si $v = ()$ et $t = t_a$`?`
    - *v* convertie en t_a si *v* est une valeur atomique et si cette conversion est possible
    - signalement d'une erreur, sinon

Test de convertibilité

exp `castable as` *t*

- L'opérateur `castable` teste si une valeur est convertible en un type atomique donné.
- *exp* `castable as` *t* $\Rightarrow$
  - vrai, si l'expression *exp* `cast as` *t* est évaluable,
  - faux, sinon

Contrainte sur le type dynamique d'une expression

exp `treat as` *t*

- L'opérateur `treat as` est utilisé pour assurer qu'une expression aura un type dynamique donné. Par exemple, lorsqu'une fonction n'est applicable qu'à un sous-ensemble des instances du type statique de l'un de ses arguments.
- Lors de l'analyse statique l'expression *exp* `treat as` *t* est considérée comme une expression de type *t*.
- *exp* `treat as` *t* $\Rightarrow$
  - *valeur*(*exp*), si cette valeur est de type *t*,
  - signalement d'une erreur, sinon

Choix

```
typeswitch (exp)
    case ($var1 as)? t1 return exp1
    …
    case ($varn as)? tn return expn
    default ($vard)? return expd
```

- Cette expression permet de réaliser un calcul sur une valeur, en fonction de son type dynamique.
- La valeur retournée est celle calculée de la façon suivante :
  - L'expression *exp* est évaluée produisant une valeur *v*.
  - On recherche le premier type t_i pour *i* variant de 1 à *n* tel que le type dynamique de la valeur *v* est de type t_i.
  - S'il existe alors :
    - on lie la valeur *v* à la variable var_i et on retourne la valeur de l'expression exp_i.

    sinon :
    - on lie la valeur *v* à la variable var_d et on retourne la valeur de l'expression exp_d.

Exemple d'expression de choix : *Désannotation d'un paragraphe* (1)

- Dans la description du document `clarée.xml`, nous avons annoté :
  - les références à un autre itinéraire du même vallon, par un élément `<renvoi cible="I`*i*`.`*j*`"/>` où `I`*i*`.`*j* est l'identificateur de cet itinéraire, avec *i* n° du vallon et *j* n° de l'itinéraire dans le vallon.
  - les conseils de prudence ou d'utilisation d'un matériel spécifique par un élément `note` muni d'un attribut type de valeur `prudence` ou `matériel`, encadrant ce conseil.
- Par exemple, le paragraphe :
  - Du col de Névache (*itinéraire n° 1*), suivre la ligne de crête qui mène à la pointe de Névache. Crampons utiles au printemps.

  a été décrit de la façon suivante en XML :
  - `<para>Du col de Névache (<renvoi cible="I15.1"/>), suivre la ligne de crête qui mène à la pointe de Névache. <note type="prudence">Attention : corniches possibles.</note><note type="matériel"> Crampons utiles au printemps.</note></para>`
- Il s'agit de retrouver le texte initial de ce paragraphe.

Exemple d'expression de choix : *Désannotation d'un paragraphe* (2)

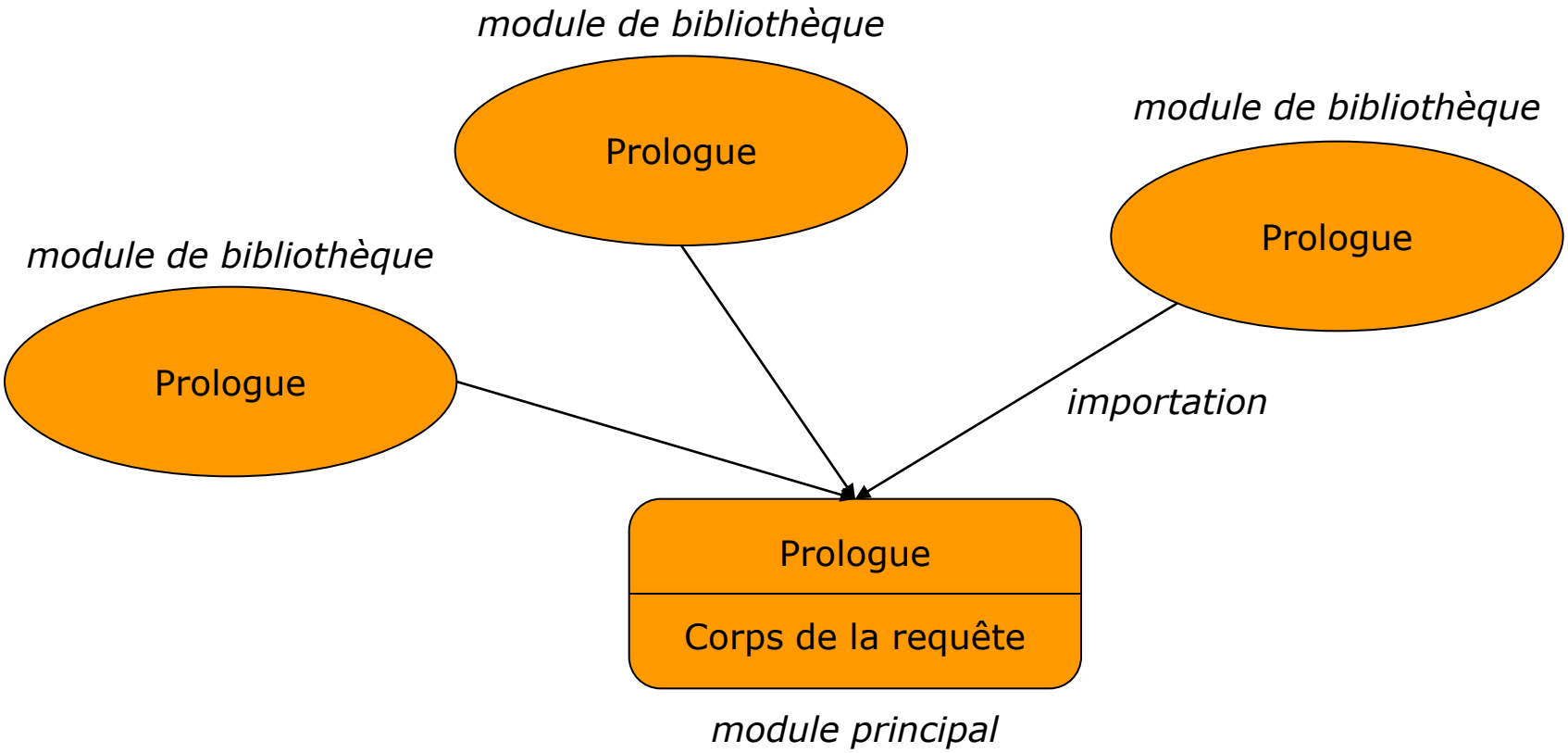
```
let $p := <para>Du col de Névache (<renvoi cible="I15.1"/>), suivre la
ligne de crête qui mène à la pointe de Névache. <note
type="prudence">Attention : corniches possibles.</note><note
type="matériel"> Crampons utiles au printemps.</note></para>
return
 <para>
  {
  for $e in $p/node()
  return
   typeswitch($e)
    case element(renvoi) return
     ("itinéraire n° ",
      $clarée//itinéraire[@id = $e/@cible]/num/text())
    case element(note) return $e/text()
    default return $e
  }
 </para>
⇒ <para>Du col de Névache (itinéraire n° 1), suivre la ligne de crête
qui mène à la pointe de Névache. Attention : corniches possibles.
Crampons utiles au printemps.</para>
```

Modules

Modules

- Une requête XQuery peut être construite à partir de un ou plusieurs **modules**.
- Un module est une unité compilable séparément qui contient un ensemble de déclarations :
  - le prologue ;
  - et éventuellement une expression à évaluer : le corps d'une requête.
- On distingue deux types de modules :
  - les **modules principaux** comportant un prologue et le corps d'une requête ;
  - les **modules de bibliothèque** réduits à un prologue.

Structure d'une requête XQuery



Prologue

- Un prologue peut comporter les déclarations :
  - de modes de traitement des espaces frontières, de construction directe d'élément et d'ordonnancement des nœuds ;
  - de collation par défaut ;
  - d'importation de schémas et de modules ;
  - d'espaces de nom ;
  - d'espaces de noms par défaut pour les éléments, les types et les fonctions ;
  - de variables globales ;
  - de fonctions ;

  ...

Espaces de noms par défaut

- Les noms d'éléments ou de types non préfixés appartiennent :
  - à l'espace de noms d'éléments ou de types par défaut si celui-ci est déclaré et n'est caché par une déclaration d'espace de noms par défaut dans un constructeur direct d'élément.
  - à aucun espace de noms, sinon.
- Les noms de fonctions non préfixés appartiennent à l'espace de noms de fonctions par défaut si celui-ci est déclaré, sinon ils appartiennent à l'espace de noms `http://www.w3.org/2005/xpath-functions` de préfixe `fn` et sont donc considérés comme étant des noms de fonctions prédéfinies.
- Afin de pouvoir définir des fonctions dans un module principal sans créer un nouvel espace de noms, l'espace de noms `http://www.w3.org/2005/xquery-local-functions` de préfixe `local` est prédéfini par XQuery :
  - il est recommandé de l'utiliser.

Déclaration d'une variable

```
declare variable $v as t := exp;
```

- *v* est un nom de variable.
- *t* est un type de séquence.
- La clause `as` est facultative.
- Si la clause `as` est présente, la valeur de *v* doit être une instance du type *t*, sinon le type de la variable sera inféré lors de l'analyse de *exp.*
- L'expression *exp* est évaluée dans un contexte qui inclut toutes les fonctions déclarées ou importées dans le prologue, mais uniquement les variables déclarées ou importées avant cette fonction dans le prologue.

Exemple de déclaration d'une variable

- Déclaration d'une variable `vallons` ayant pour valeur la séquence des éléments `vallon` du document `clarée.xml`.
  - ```
    declare variable $vallons as element(vallon)*
    := fn:doc("clarée.xml")//itinéraire;
    ```
    ```

Déclaration d'une fonction

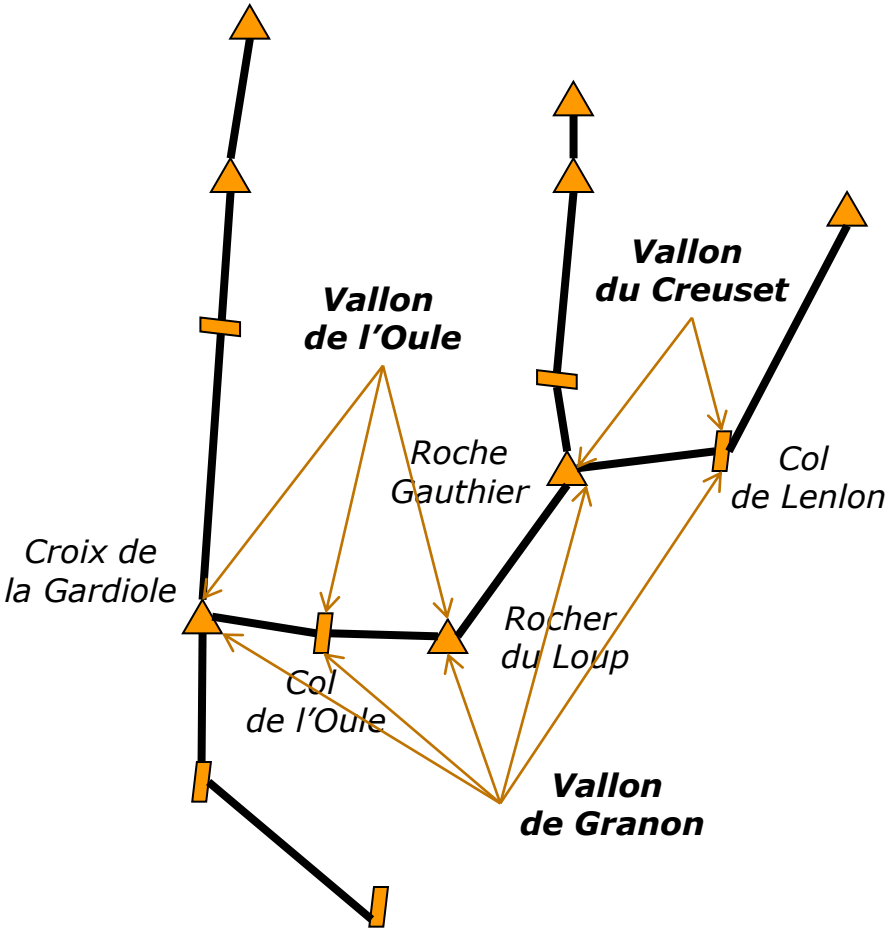
`declare function` *f*`($`n_1 `as` t_1`, …, $`n_k `as` t_k`) as` *t* `{`*exp*`}`

- *f* est le nom de la fonction : un nom qualifié.
- n_1, …, n_k sont les noms des arguments formels : des noms qualifiés. Une fonction peut ne pas avoir d'arguments.
- t_1, …, t_k et *t* sont des types de séquence, respectivement, les types des arguments et celui de la valeur retournée par l'appel de la fonction.
- *exp* est une expression : le **corps** de la fonction,
- Si le type d'un argument ou celui de la valeur retournée n'est pas déclaré, il est considéré comme étant par défaut `item()*`.
- L'expression *exp* est évaluée dans un contexte qui inclut toutes les fonctions déclarées ou importées dans le prologue, mais uniquement les variables déclarées ou importées avant cette fonction dans le prologue.

Exemple de déclaration d'une fonction : *vallon-adjacent-à* (1)

- Il s'agit de déclarer une fonction qui retourne les vallons adjacents à un vallon donné *v.*
- On suppose que deux vallons v_1 et v_2 ($v_1 \neq v_2$) sont adjacents s'il existe dans v_1 un itinéraire de même nom que celui d'un itinéraire de v_2.
- On notera que ce critère suppose que les noms des itinéraires sont les noms des points géographiques (col ou sommet) atteints. C'est le cas pour les itinéraires du guide des itinéraires skieurs de la Vallée de la Clarée, excepté ceux qui sont des circuits.
- Ce critère est évidemment très approximatif car deux vallons qui ont une portion de crête commune peuvent ne pas apparaître dans la réponse car :
  - il peut ne pas exister dans v_1 ou v_2 d'itinéraires qui atteignent un point de cette crête ;
  - il peut ne pas exister dans v_1 un itinéraire qui atteint un point de cette crête atteint par un itinéraire de v_2 ou inversement.

Exemple de déclaration d'une fonction : *vallon-adjacent-à* (2)



Bien que
le **Vallon du Creuset**
soit géographiquement
adjacent au
Vallon de l'Oule,
ceci n'apparaîtra pas
dans la réponse
car ces deux vallons
n'ont pas d'itinéraires
de même nom
(atteignant le même point)

Exemple de déclaration d'une fonction : *vallon-adjacent-à* (3)

```
declare function vallon-adjacent-à
  ($v as element(vallon)) as element(vallon)*
  {
  for $va in $vallons
  where fn:not($va is $v) and
        $va//itinéraire/nom = $v//itinéraire/nom
  return $va
  }
```

Exemple de requête :
Vallons adjacents à chaque vallon (4)

```
declare variable $vallons :=
 fn:doc("exemples/clarée/doc.xml")//vallon;
declare function local:vallon-adjacent-à($v)
 {
 for $va in $vallons
 where fn:not($va is $v) and
       $va//itinéraire/nom = $v//itinéraire/nom
 return $va
 };
for $v in $vallons
return
 <vallon nom="{$v/nom}">
  {
  for $va in local:vallon-adjacent-à($v)
  return
   <vallon-adjacent nom="{$va/nom}"/>
  }
 </vallon>;;
```

Exemple de requête :
Vallons adjacents à chaque vallon (5)

```
⇒
<vallon nom="Vallon de Granon">
<vallon-adjacent nom="Vallon du Creuset"/>
<vallon-adjacent nom="Vallon de l'Oule"/>
<vallon-adjacent nom="Vallon de Cristol"/>
</vallon>,
...
<vallon nom="Vallon du Creuset">
<vallon-adjacent nom="Vallon de Granon"/>
</vallon>,
<vallon nom="Vallon de l'Oule">
<vallon-adjacent nom="Vallon de Granon"/>
<vallon-adjacent nom="Vallon de l'Oule"/>
 <vallon-adjacent nom="Vallon de Cristol"/>
</vallon>,
...
```

Exemple de requête : *Vallons pour bons skieurs* (1)

- Il s'agit de rechercher les noms des vallons qui ne contiennent que des itinéraires dont la difficulté est supérieure ou égale à 3 étoiles.

Exemple de requête : *Vallons pour bons skieurs (2)*

```
declare function local:est-vallon-pour-bons-skieurs($v)
 {
 let $itis := $v//itinéraire
 return
  every $i in $v//itinéraire
  satisfies $i/cotation ge "***"
 };
<vallons-pour-bons-skieurs>
 {
 for $v in fn:doc("clarée.xml")//vallon
 where local:est-vallon-pour-bons-skieurs($v)
 return
  <vallon nom="{$v/nom}"/>
 }
</vallons-pour-bons-skieurs>
```

Exemple de requête : *Vallons pour bons skieurs* (3)

⇒

```
<vallons-pour-bons-skieurs>
   <vallon nom="Vallon du Creuset"/>
   <vallon nom="Vallon de l'Oule"/>
   <vallon nom="Vallon de Cristol"/>
   <vallon nom="Le vallon de Ricou"/>
   <vallon nom="Vallon de la Cula"/>
   <vallon nom="Vallon des Béraudes"/>
</vallons-pour-bons-skieurs>
```

Exemple de module de bibliothèque (1)

- Dans le cas d'une collection de guides d'itinéraires à skis ayant la même structure que celui de la Clarée, il peut être intéressant de créer un module de bibliothèque réunissant les fonctions usuelles pour interroger ces guides.
- Par exemple :
  - la fonction `vallon-adjacent-à`,
  - la fonction `est-vallon-pour-bons-skieurs`,

  ...

Exemple de module de bibliothèque (2)

```
module namespace guide = "http://..."
declare function guide:vallon-adjacent-à($v)
 {
 for $va in $vallons
 where fn:not($va is $v) and
       $va//itinéraire/nom = $v//itinéraire/nom
 return $va
 };
declare function guide:est-vallon-pour-bons-skieurs($v)
 {
 let $itis := $v//itinéraire
 return
  every $i in $v//itinéraire
  satisfies $i/cotation ge "***"
 };
...
```