Récursivité en XSL

Pourquoi et comment utiliser la récursivité dans les transformations XSL ?

Les limites de XSL

- La modification des variables est impossible
  - Pas de boucle de type pour i de 1 à n
  - Complexifie le codage de certains algorithmes pourtant simples à la base

Solution : la récursivité

- Utilisation de template récursif
  - C'est-à-dire un template qui se rappelle lui-même
  - Au lieu de modifier les variables, on modifie les paramètres du template

Exemple de problème n°1

- On cherche à déterminer le maximum dans une liste de nombres : listemax.xml

```
<ListeNombres>
    <nombre>111</nombre>
    <nombre>27</nombre>
    <nombre>9</nombre>
    <nombre>57</nombre>
    <nombre>2</nombre>
    <nombre>363</nombre>
</ListeNombres>
```

Algorithme simple

max ← premier élément nombre de listeNombres
Pour chaque élément nombre **de** listeNombres
  si nombre > max
  alors max ← nombre
  fsi
Finpour
Écrire max

Problème :

en XSL on ne peut pas modifier la valeur de max

Algorithme récursif

```
Fonction RechercheMax(max, indice){
    si indice = nbéléments(listeNombres)
        alors si listNombre[indice] > max
                    alors écrire listNombre(indice)
                    sinon écrire max
              fsi
    sinon si listNombre[indice] > max
              alors rechercheMax(listNombre[indice], indice + 1)
              sinon rechercheMax(max, indice +1)
              fsi
    fsi
}
```

Premier appel : RechercheMax(listNombres[1],2)

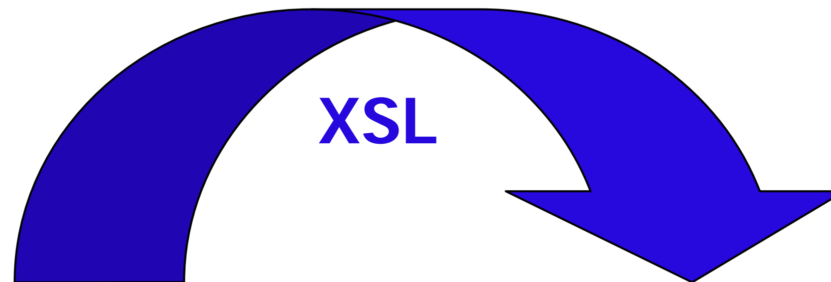
listemax.xml
rechercheMax.xsl

```
<xsl:template name="RechercheMax">
        <xsl:param name="max"></xsl:param>
       <xsl:param name="indice"></xsl:param>

        <xsl:choose>
             <!-- On teste si on a atteint le dernier élément nombre -->
           <xsl:when test="$indice= count(//nombre)">
                  <xsl:choose>
                  <!-- On teste si la valeur de l'élément courant est supérieure à max -->
                  <xsl:when test="//nombre[position() = $indice] &gt; $max">
                      <!-- On affiche la valeur de l'élément à l'indice -->
                          <xsl:value-of select="//nombre[position() = $indice]"/>…
                          <xsl:otherwise>
                      <!– Sinon on affiche max -->
                      <xsl:value-of select="$max"/>

           <!– Si on n'a pas atteint le dernier élément -->
           <xsl:choose>
                      <!-- On teste si la valeur de l'élément courant est supérieure à max -->
                  <xsl:when test="//nombre[position() = $indice] &gt; $max">
                      <!-- on rapelle le template avec la nouvelle valeur de max et on incrémente
                        l'indice -->
                      <xsl:call-template name="RechercheMax">
                          <xsl:with-param name="max"><xsl:value-of select="//nombre[position()
                            = $indice]"/></xsl:with-param>
                          <xsl:with-param name="indice"><xsl:value-of select="$indice +1"/>
                           </xsl:with-param>
                          </xsl:call-template>…
```

Exemple de problème n°2

- Énoncé
  - A partir d'une liste de nombres décimaux naturels, on souhaite récupérer un fichier XML contenant les mêmes nombres en binaire

Exemple de problème n°2




```
<ListeNombre>
   <nombre>0</nombre>
   <nombre>1</nombre>
   <nombre>2</nombre>
   <nombre>3</nombre>
   <nombre>4</nombre>
</ListeNombre>
```

```
<ListeNombreBinaire>
   <nombreBinaire VD="0">0</nombreBinaire>
   <nombreBinaire VD="1">1</nombreBinaire>
   <nombreBinaire VD="2">10</nombreBinaire>
   <nombreBinaire VD="3">11</nombreBinaire>
   <nombreBinaire VD="4">100</nombreBi naire>
</ListeNombreBinaire>
```

Algorithme récursif

Algorithme

à appliquer pour chaque nombre de la listeNombre

```
fonction transformationBinaire(n:entier)
   début
       si n < 2
           alors écrire(n)
           sinon transformationBinaire(n ÷ 2)
                 écrire(n mod 2)
       fsi
   fin
```

Template XSL correspondant

```
<xsl:template name="transformationBinaire">
    <xsl:param name="nb"></xsl:param>
    <xsl:choose>
      <xsl:when test="$nb &lt; 2">
            <xsl:value-of select="$nb"/> //écrire nb
      </xsl:when>
      <xsl:otherwise>
            <xsl:call-template name="transformationBinaire">
                  <xsl:with-param name="nb" select="($nb - ($nb
                  mod 2)) div 2"/>
            </xsl:call-template>
            <xsl:value-of select="$nb mod 2"/> //écrire nb mod 2
      </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

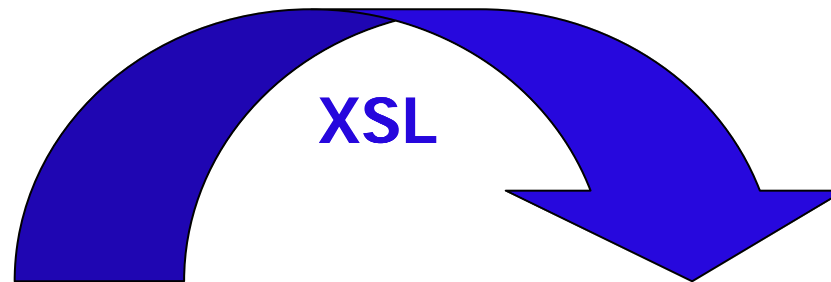
transformationBinaire.xsl

listeNombre.xml

Exemple de problème n°3

- Énoncé
  - A partir d'une liste de nombres décimaux naturels, on souhaite récupérer un fichier XML contenant les factoriels de chaque nombre
- Rappels :
  - 0! = 1
  - 1! = 1
  - 4! = 4x3x2x1
  - 5! = 5x4x3x2x1 = 5x4! (d'où la récursivité)

Exemple de problème n°3




```
<ListeNombre>
    <nombre>0</nombre>
    <nombre>1</nombre>
    <nombre>2</nombre>
    <nombre>3</nombre>
    <nombre>4</nombre>
</ListeNombre>
```

```
<ListeFactoriel>
    <factoriel nombreBase="0">1</factoriel>
    <factoriel nombreBase="1">1</factoriel>
    <factoriel nombreBase="2">2</factoriel>
    <factoriel nombreBase="3">6</factoriel>
    <factoriel nombreBase="4">24</factoriel>
</ListeFactoriel>
```

Algorithme non récursif

```
Fonction calculFactoriel (n:entier)
    début
        pour i de n à 1
            résultat= résultat * i
        finpour
        écrire résultat
    fin
```

Problèmes:

- Pas de boucles de ce type en XSL
- Impossible de revaloriser une variable en XSl
- Pas propre : ne suit pas la définition mathématique n! = n x (n-1)!

Algorithme récursif

- Il suffit de reprendre la définition mathématique :
  - n!= n x (n-1)!
  - 0! = 1

Fonction calculFactoriel(n:entier)
    début
      si n=0
            alors résultat ← 1
            sinon résultat ← n * calculFactoriel (n-1)
      fsi
    écrire résultat
    fin

Exercice

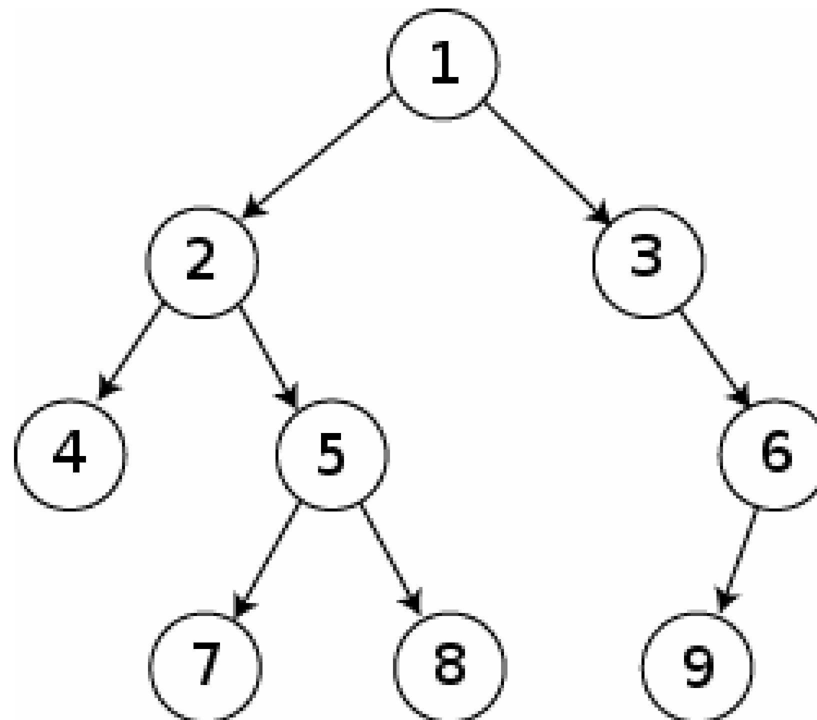
- Enoncé
  - Ecrire la XSL correspondante
- Solution
  - listeNombres.xml, calculFactoriel.xsl

Arbres B

- Représentation en XML
- Parcours
- Algorithmes divers

Représentation des arbres B

- Essayons de représenter l’arbre suivant en fichier XML



Représentation des arbres B

Représentation arborescente

<Arbre>
<nœud val=1>
<nœud val = 2>
<nœud val = 4/>
<nœud val = 5>
<nœud val = 7/>
<nœud val = 8/>
</nœud>
</nœud>
<nœud val = 3>
<nœud val = 6>
<nœud val = 9/>
</nœud>
</nœud>
</nœud>
<Arbre>

Représentation des arbres B

Représentation « à plat »

```
<Arbre>
    <nœud ID= 'n1' fg= 'n2' fd= 'n3' >1<nœud>
    <nœud ID= 'n2' fg= 'n4' fd= 'n5' >2<nœud>
    <nœud ID= 'n3' fg= 'n6' >3<nœud>
    <nœud ID= 'n4' >4<nœud>
    <nœud ID= 'n5' fg= 'n7' fd= 'n8' >5<nœud>
    <nœud ID= 'n6' fg= 'n9' >6<nœud>
    <nœud ID= 'n7' >7<nœud>
    <nœud ID= 'n8' >8<nœud>
    <nœud ID= 'n9' >9<nœud>
</Arbre>
```

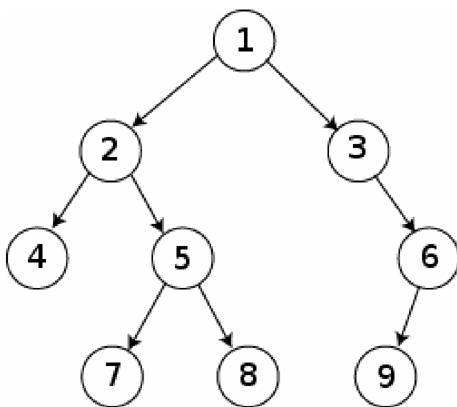
Avec fg: fils gauche et fd: fils droit

Les parcours d'arbres

- Parcours préfixé
- Parcours infixé
- Parcours postfixé

Parcours préfixé

- Principe
  - Il consiste à visiter le contenu du nœud dès qu'on est dessus, ensuite on parcourt son fils gauche puis son fils droit



Ainsi l'affichage de cet arbre en ordre préfixé donnerait :

1,2,4,5,7,8,3,6,9

Parcours préfixé

- Algorithme :

```
fonction imprimerPréfixé(arbre : ArbreBinaireChaîne, noeud : Noeud)
    début
        si non noeudvide(arbre,noeud) alors
            écrire(val(arbre, noeud))
            imprimerPréfixé(arbre, fg(arbre, noeud))
            imprimerPréfixé(arbre, fd(arbre, noeud))
        fsi
    fin
```

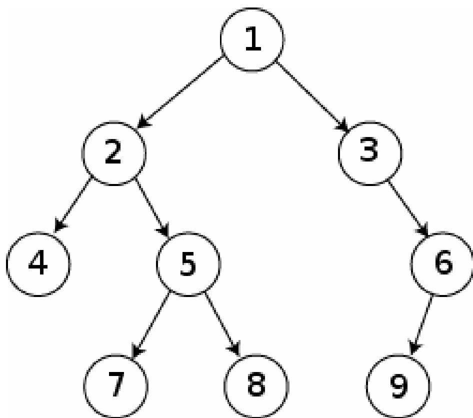
- Remarque :
  - le paramètre **noeud** est nécessaire pour pouvoir utiliser la récursivité, il est la racine du sous-arbre en cours de traitement
  - A l'appel de la fonction, noeud est la racine de arbre

Parcours préfixé en XSL: arbre.xml prefixe.xsl

```
<xsl:template match="/">
    <xsl:apply-templates select="//arbre"/>
</xsl:template>

<!– Premier appel sur la racine de l’arbre -->
<xsl:template match="arbre">
    <xsl:apply-templates select="child::noeud"/>
</xsl:template>

<!-- fonction récursive -->
<xsl:template match="noeud">
    <!-- Impression de la valeur du noeud -->
    <xsl:value-of select="concat(attribute::val,', ')"/>
    <!-- Rappel de la fonction pour le fils gauche -->
    <xsl:apply-templates select="child::noeud[position()='1']"/>
    <!-- Rappel de la fonction pour le fils droit -->
    <xsl:apply-templates select="child::noeud[position()='2']"/>
</xsl:template>
```

Parcours postfixé

- Principe
  - Il consiste à afficher le contenu du nœud après avoir parcouru et affiché ses deux fils



Ainsi l'affichage de cet arbre en ordre postfixé donnerait :

4,7,8,5,2,9,6,3,1

Parcours postfixé

- Algorithme
  - Voici l'algorithme qui imprime un arbre dans l'ordre postfixé :

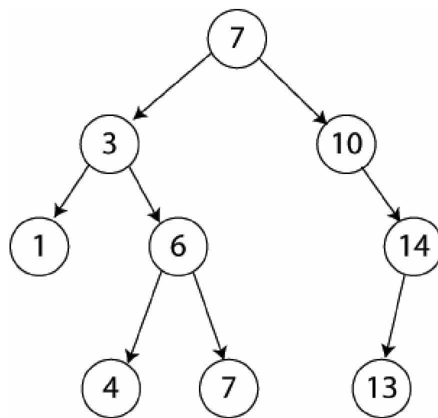
```
fonction imprimerPostfixé(arbre : ArbreBinaireChaîne, noeud :
   Noeud)
   début
       si non noeudvide(arbre, noeud) alors
          imprimerPostfixé(arbre, fg(arbre, noeud))
          imprimerPostfixé(arbre, fd(arbre, noeud))
          écrire(val(arbre, noeud))
       fsi
   fin
```

Parcours postfixé XSL: arbre.xml, postfixe.xsl

```
<xsl:template match="/">
    <xsl:apply-templates select="//arbre"/>
</xsl:template>

<!– Premier appel sur la racine de l’arbre -->
<xsl:template match="arbre">
    <xsl:apply-templates select="child::noeud"/>
</xsl:template>

<!-- fonction récursive -->
<xsl:template match="noeud">
    <!-- Rappel de la fonction pour le fils gauche -->
    <xsl:apply-templates select="child::noeud[position()='1']"/>
    <!-- Rappel de la fonction pour le fils droit -->
    <xsl:apply-templates select="child::noeud[position()='2']"/>
    <!-- Impression de la valeur du noeud -->
    <xsl:value-of select="concat(attribute::val,', ')"/>
</xsl:template>
```

Parcours infixé

- Principe
  - Il consiste à visiter le contenu du noeud après avoir visité le contenu de son fils gauche; une fois que c'est fait, on visite son fils droit



Ainsi l'affichage de cet arbre en ordre infixé donnerait :

1,3,4,6,7,7,10,14,13

On remarque que cela affiche les valeurs dans l'ordre croissant lorsqu'on a affaire avec un arbre de recherche

Parcours infixé

- Algorithme :

fonction imprimerPostfixé(arbre : ArbreBinaireChaîne, noeud : Noeud)
début
si non noeudvide(arbre, noeud) alors
imprimerPostfixé(arbre, fg(arbre, noeud))
écrire(val(arbre, noeud))
imprimerPostfixé(arbre, fd(arbre, noeud))

fsi
fin

Parcours infixé XSL: arbre.xml, infixe.xsl

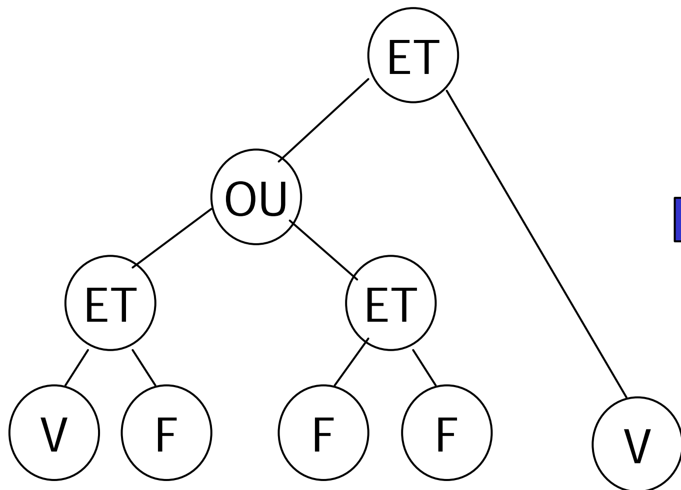
```
<xsl:template match="/">
    <xsl:apply-templates select="//arbre"/>
</xsl:template>

<!– Premier appel sur la racine de l’arbre -->
<xsl:template match="arbre">
    <xsl:apply-templates select="child::noeud"/>
</xsl:template>

<!-- fonction récursive -->
<xsl:template match="noeud">
    <!-- Rappel de la fonction pour le fils gauche -->
    <xsl:apply-templates select="child::noeud[position()='1']"/>
    <!-- Impression de la valeur du noeud -->
    <xsl:value-of select="concat(attribute::val,', ')"/>
    <!-- Rappel de la fonction pour le fils droit -->
    <xsl:apply-templates select="child::noeud[position()='2']"/>
</xsl:template>
```

Évaluation d'une expression booléenne

- Création de l'arbre d'une expression booléenne
- Évaluation de l'expression

Évaluation d'une expression booléenne

- Représentation
  - L'opérateur dans la racine et les valeurs dans les fils

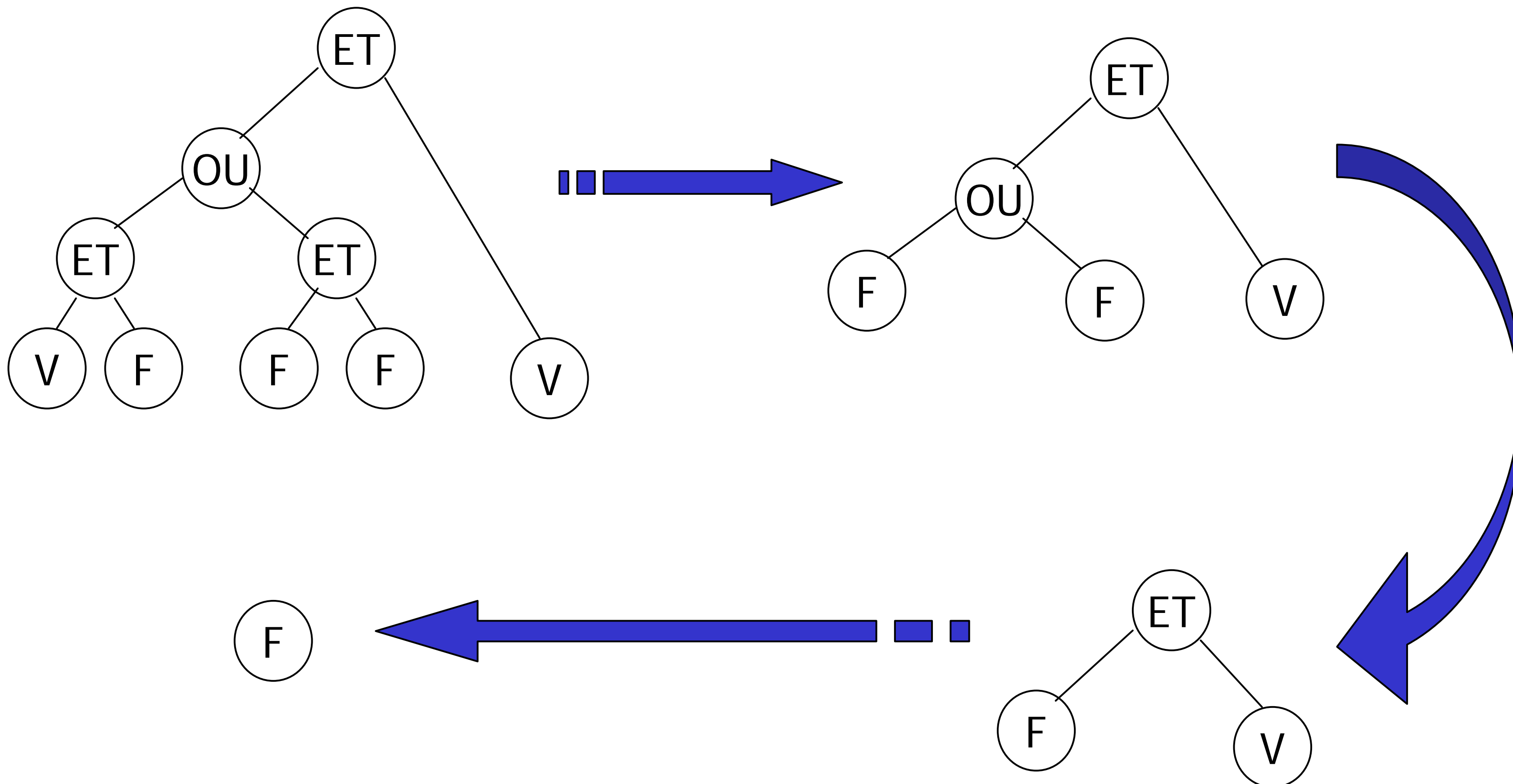



```
<arbre>
    <noeud val="et">
        <noeud val="ou">
            <noeud val="et">
                <noeud val="vrai"/>
                <noeud val="faux"/>
            </noeud>
            <noeud val="et">
                <noeud val="faux"/>
                <noeud val="faux"/>
            </noeud>
        </noeud>
        <noeud val="vrai"/>
    </noeud>
</arbre>
```

FAUX

Évaluation d'une expression booléenne représentée sous forme d'un arbre binaire

Les étapes de résolution



Algorithme

```
fonction evaluerExpression(arbre:arbreBin, nœud:Noeuds){
    si val(arbre, nœud) = vrai
      alors retourne vrai
    fsi
    si val(arbre, nœud) = faux
      alors retourne faux
    fsi
    si val (arbre, nœud) = 'et'
      alors valFg ← evaluerExpression(arbre, fg(arbre, nœud)
              valFd ← evaluerExpression(arbre, fd(arbre, nœud)
              retrourne ( valFg et valFd)
    fsi
    si val (arbre, nœud) = 'ou'
      alors valFg ← evaluerExpression(arbre, fg(arbre, nœud)
              valFd ← evaluerExpression(arbre, fd(arbre, nœud)
              retrourne ( valFg ou valFd)
    fsi
}
valFg, valFd : booléen
```

XSL

arbreBooleen.xml

evalueExprBooleenne.xsl

```
<xsl:template match="noeud">
    <xsl:choose>
        <xsl:when test= "./@val='et' ">
            <xsl:variable name="fg">
                <xsl:apply-templates select="child::noeud[position()='1']"/>
            </xsl:variable>
            <xsl:variable name="fd">
                <xsl:apply-templates select="child::noeud[position()='2']"/>
            </xsl:variable>
            <xsl:value-of select="$fg= 'vrai'  and $fd='vrai'"/> //résultat du template
        </xsl:when>
        <xsl:when test="./@val ='ou' ">
            <xsl:variable name="fg">
                <xsl:apply-templates select="child::noeud[position()='1']"/>
            </xsl:variable>
            <xsl:variable name="fd">
                <xsl:apply-templates select="child::noeud[position()='2']"/>
            </xsl:variable>
            <xsl:value-of select="$fg= 'vrai' or $fd='vrai'"/> //résultat du template
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="./@val"/> //résultat du template
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Combinaison de deux templates récursifs

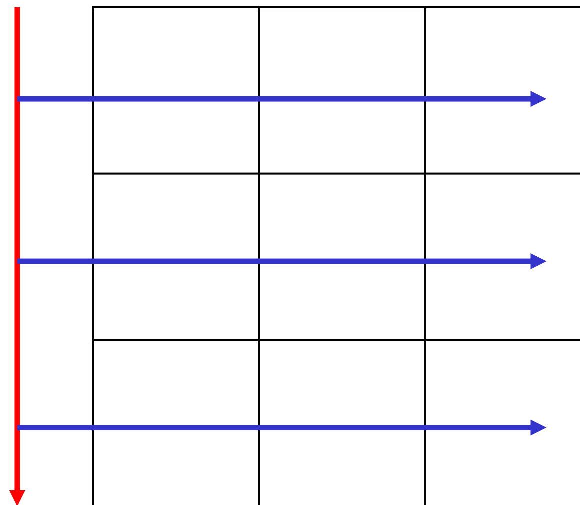
- Principe
  - Cette technique est très utile pour traiter la plupart des problèmes à plusieurs dimensions
  - Elle consiste à utiliser plusieurs templates récursifs, dont certains font appel à d'autres

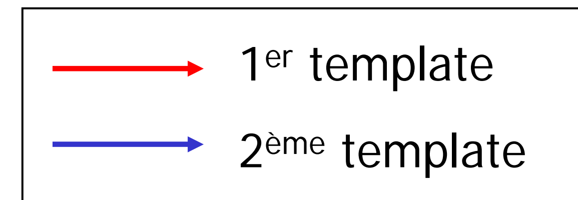
Exemple

- Création d’une table de multiplication
  - Une table de multiplication est représentée par un tableau à deux dimensions
  - C’est pourquoi, il est judicieux de combiner deux templates récursifs

Exemple

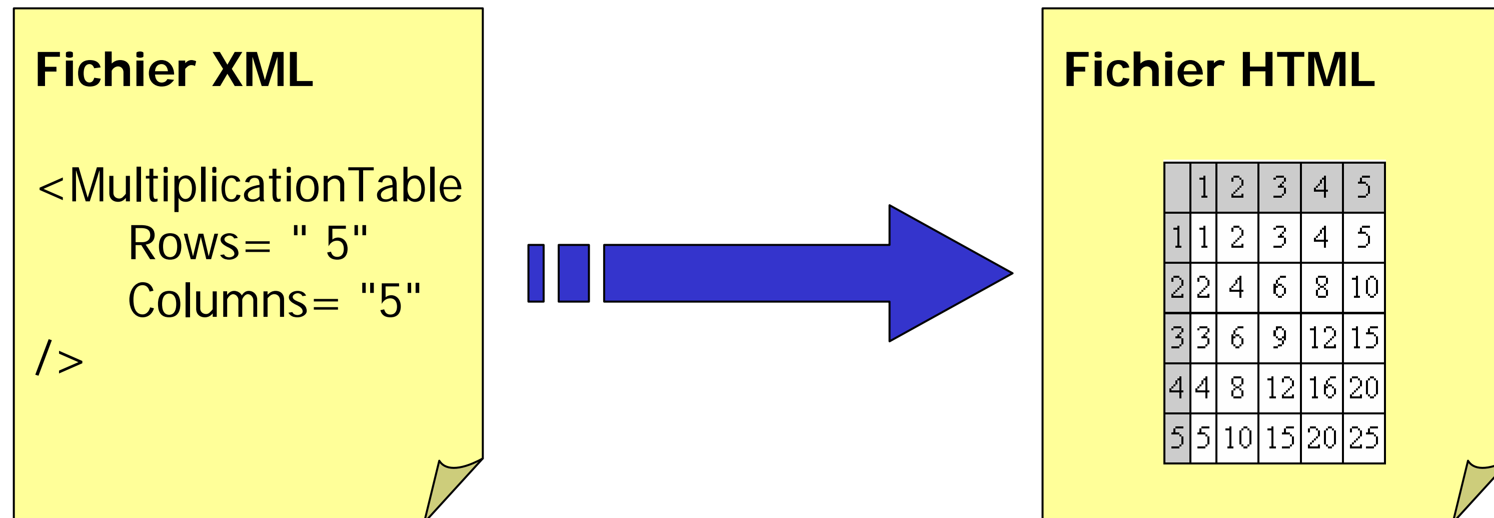
- Les templates
  - Le premier dessinera les lignes du tableau et utilisera le deuxième afin de dessiner les cellules pour chaque colonne de la ligne





Exemple :
Création d'une table de multiplication

- Objectif :
  - A partir d'un fichier XML contenant des informations sur le nombre de lignes et de colonnes, il s'agit de créer une table de multiplication au format HTML



	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Template drawRow

```
<xsl:template name="drawRow">
    <xsl:param name="currentRow"/>
    <xsl:param name="totalRows"/>
    <xsl:param name="totalCols"/>
      <xsl:if test="$currentRow &lt;= $totalRows">
          <tr> //création de la ligne courante
             <xsl:call-template name="drawCell">
               <xsl:with-param name="currentRow" select="$currentRow"/>
               <xsl:with-param name="currentCol" select="0"/>
               <xsl:with-param name="totalCols" select="$totalCols"/>
             </xsl:call-template>
          </tr>
          <xsl:call-template name="drawRow"> // ligne suivante
                <xsl:with-param name="currentRow" select="$currentRow  + 1"/>
                <xsl:with-param name="totalRows" select="$totalRows"/>
                <xsl:with-param name="totalCols" select="$totalCols"/>
          </xsl:call-template>
      </xsl:if>
  </xsl:template>
```

Template drawCell

mult_table.xml
mult_table.xsl

```
<xsl:template name="drawCell">
    <xsl:param name="currentRow"/>
    <xsl:param name="currentCol"/>
    <xsl:param name="totalCols"/>
    <xsl:if test="$currentCol &lt;= $totalCols"> //test de la position
        <xsl:variable name="bgColor">
            <xsl:choose>
              <xsl:when test="$currentRow = 0 or $currentCol = 0">#CCCCCC</xsl:when>
              <xsl:otherwise>white</xsl:otherwise>
            </xsl:choose>
        </xsl:variable>
        <xsl:variable name="value">
            <xsl:choose>
              <xsl:when test="$currentRow = 0 and $currentCol = 0"></xsl:when>
              <xsl:when test="$currentRow = 0"><xsl:value-of select="$currentCol"/></xsl:when>
              <xsl:when test="$currentCol = 0"><xsl:value-of select="$currentRow"/></xsl:when>
              <xsl:otherwise><xsl:value-of select="$currentRow * $currentCol"/></xsl:otherwise>
            </xsl:choose>
        </xsl:variable>
        <td bgcolor="{$bgColor}" align="center" valign="top"><xsl:value-of select="$value"/></td>
            <xsl:call-template name="drawCell">
                        <xsl:with-param name="currentRow" select="$currentRow"/>
                        <xsl:with-param name="currentCol" select="$currentCol + 1"/>
                        <xsl:with-param name="totalCols" select="$totalCols"/>
            </xsl:call-template>
    </xsl:if></xsl:template>
```

Exercice

- Énoncé
  - Inspirez-vous de l'exemple précédent pour construire le triangle de Pascal. On peut griser les 1

    1
    1 1
    1 2 1
    1 3 3 1
    1 4 6 4 1

Exercice

- Algorithme récursif

```
function Pascal(x, y: integer): integer;
begin
  if (x = 1) or (y = x) then Pascal := 1
  else Pascal := Pascal(x, y - 1) + Pascal(x -
  1, y - 1);
end;
```

Exercice

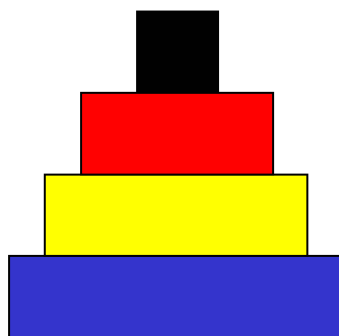
- Solution :
  - Pascal.xml Pascal.xsl

Les tours de Hanoi

- Résolution du problème des tours de Hanoi

Les tours de Hanoi

- Explications
  - Soient trois tours. Sur l'une d'entre sont empilés des disques par taille décroissante :



T1 T2 T3

  - Le but du jeu consiste à déplacer les n disques de la tour de départ vers une des deux autres tours

- Règles de déplacement :
  - on ne peut déplacer qu'un seul disque à la fois
  - un disque ne peut pas se trouver sur un disque de diamètre inférieur

Les tours de Hanoi

Algorithme de résolution du problème

fonction hanoï(nombredisques: entier, départ: chaîne, arrivée: chaîne,
intermédiaire: chaîne)
début
si nombredisques 0 **alors**
hanoï(nombredisques-1, départ, intermédiaire, arrivée)
écrire(" déplacer le disque ", nombredisques, " de ", départ, " vers ", arrivée)
hanoï(nombredisques-1, intermédiaire, arrivée, départ)
fsi
fin

Les tours de Hanoi

hanoi.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="hanoi.xsl"?>
<Test>
    <Hanoi Size="3"/>
</Test>
```

Hanoi.xsl

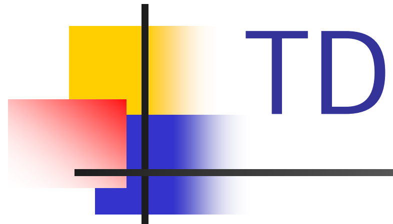
```
// Appel

<xsl:template match="/">
 <xsl:variable name="size" select="number(/*/Hanoi/@Size)"/>
 <HanoiSolution Size="{$size}">
        <xsl:call-template name="hanoi">
                <xsl:with-param name="height"    select="$size"/>
                <xsl:with-param name="fromTower" select="'A'"/>
                <xsl:with-param name="toTower"   select="'B'"/>
                <xsl:with-param name="withTower" select="'C'"/>
        </xsl:call-template>
 </HanoiSolution>
</xsl:template>
```

Hanoi.xsl

```
<xsl:template name="hanoi">
    <xsl:param name="height"/>
    <xsl:param name="fromTower"/>
    <xsl:param name="toTower"/>
    <xsl:param name="withTower"/>
    <xsl:if test="$height &gt;= 1">
        <xsl:call-template name="hanoi">
            <xsl:with-param name="height" select="$height - 1"/>
            <xsl:with-param name="fromTower" select="$fromTower"/>
            <xsl:with-param name="toTower"   select="$withTower"/>
            <xsl:with-param name="withTower" select="$toTower"/>
        </xsl:call-template>
        <Move FromTower="{$fromTower}" ToTower="{$toTower}"/>
        <xsl:call-template name="hanoi">
            <xsl:with-param name="height"    select="$height - 1"/>
            <xsl:with-param name="fromTower" select="$withTower"/>
            <xsl:with-param name="toTower"   select="$toTower"/>
            <xsl:with-param name="withTower" select="$fromTower"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>
```

Résultat

```
<?xml version="1.0" encoding="UTF-16"?>
<HanoiSolution Size="3" xmlns:xalan="http://xml.apache.org/xslt">
   <Move FromTower="A" ToTower="B" />
   <Move FromTower="A" ToTower="C" />
   <Move FromTower="B" ToTower="C" />
   <Move FromTower="A" ToTower="B" />
   <Move FromTower="C" ToTower="A" />
   <Move FromTower="C" ToTower="B" />
   <Move FromTower="A" ToTower="B" />
</HanoiSolution>
```



TD

- Énoncé
  - TD-XSL-recursif