Ulric GENIEVRE
Yannick LITAIZE
Wojciech MACHOCKI
Ludovic MAURILLON
Benoît ROGER
Sébastien VALLEE

Département de Génie Mathématique
5° Année

INSA de Rouen

Conversion TEX et Rendu MathML

Etude et Réalisation d'applications concernant la diffusion d'informations mathématiques

Sommaire

Remerciements

Merci à Catherine BARRY d'avoir accepter d'encadrer ce projet,

Merci à l'ensemble des développeurs de Xerces, Xalan et Crismon (en gros à l'ensemble des développeurs d'IBM et de Sun…) de nous avoir fourni de si beaux outils,

Merci à Jean Frédéric MESNIL pour l'ensemble de ses remarques, de ses suggestions, et de son talent.

Répartition des tâches

Organisation de l'Equipe

Wojciech :

- Rédaction de la feuille de style XSLT pour le passage de DOM TEX à DOM MathML,
- Rédaction de la théorie sur les transformations XML. (XSL, XSLT, Xpath).
- Explication du fichier XSL.

Sébastien :

- Recherche sur le mécanisme d'un moteur de rendu,
- Documentation sur le standard UNICODE,
- Recherche d'une font compatible avec les caractères unicode utilisés pour afficher les formules mathématiques,
- Recherche de méthodes permettant d'afficher des fonts en Java (positionnement, taille, etc.)
- Implémentation du moteur pour quelques exemples, à partir du parser MathML.
- Rédaction des chapitres sur UNICODE, fonts, ….

Benoît :

- Modélisation de la conversation TEX - MathML par XSLT, démonstration de la faisabilité,
- Parser MathML : aide au debuggage, migration de Xerces à Crimson.
- Moteur de rendu : création du modèle objet de visualisation, assistance technique sur l'affichage des glyphs et le viewer.

Ludovic :

- Détail des spécifications MathML,
- Recherche sur les parsers existants,
- implémentation de la DOM MathML, puis du parser par rapport à Xerces,

- Rédaction des chapitres sur XML, MathML et sur le parser MathML.

Yannick :

- Mise à niveau sur TEX,
- Modélisation du parser TEX,
- Rédaction du rapport TEX

Ulric :

- Modélisation et implémentation du parser TEX,

Introduction

Le développement des nouvelles technologies a permis l'apparition de nouveaux langages de modélisation de données dont XML et plus particulièrement MathML. Ce dernier permet de recentrer la publication sur le Web autour d'articles scientifiques, ce qui était la tâche première d'Internet lors de sa création. Cependant, bien que les applications tournant autour d'XML foisonnent de plus en plus, les outils MathML sont peu étudiés et développés actuellement. Il nous a donc semblé important de réfléchir à des solutions de traitement de données mathématiques. Pour ce faire nous avons lancé le projet **MathIT**, une initiative étudiante, avec pour philosophie les trois règles suivantes :

1) **MathIT** est une réalisation et une initiative des étudiants du département Génie Mathématique de l'Institut National des Sciences Appliquées de Rouen. Il se doit donc de reprendre dans chacune de ses études les deux composantes qui font la force de ses élèves, à savoir les mathématiques et l'informatique.

2) **MathIT** est une cellule de recherche. Il se doit donc d'aborder chaque sujet sous un angle nouveau : il ne s'agit pas de récupérer ce qui existe déjà pour arriver à nos fins, mais à chaque fois de tirer parti des nouveaux standards et langages, ou de faire une utilisation originale d'outils déjà maîtrisés. De plus, cela permettra aux concepteurs d'évaluer les nouveaux produits et d'en faire par la suite une synthèse objective.

3) **MathIT** est totalement opensource. Il est destiné à aider toute personne désireuse de communiquer des données mathématiques via le Web. Par conséquent, il n'est pas la propriété intellectuelle de ses initiateurs, toute proposition venant de l'extérieure est à prendre en compte.

Ces belles paroles étant lancées, ils nous a fallu définir de manière plus concrète des objectifs à atteindre :

Comme tout langage ou format, XML se doit de respecter une syntaxe et une grammaire. Celles-ci doivent évidemment être contrôlées. Cette opération est réalisée par ce que l'on appelle un « parser ». Il existe actuellement différents parsers, libres ou non, bâtis sur différents modèles (événementiels ou objet), notre application pourrait donc se contenter d'utiliser l'un d'entre eux. Cependant, bien que MathML respecte la grammaire et la syntaxe XML classique, ces concepteurs ont rajouté certaines règles, qu'un parser XML générique ne peut contrôler. Le développement d'un parser MathML spécifique était donc primordial.

A l'heure actuelle, il est extrêmement difficile de visualiser correctement des équations mathématiques sur le Web. En effet, dans la plupart des cas ses équations sont représentées par des images, ce qui procure un rendu peut appréciable et l'impossibilité de les récupérer. Par conséquent, afin de mieux transmettre l'information mathématique, le développement d'un moteur de rendu nous est apparu comme essentiel.

Enfin, pour favoriser la diffusion et de ses documents, ainsi que pour récupérer l'information déjà présente dans bons nombres de documents sous d'autres formats, il serait souhaitable de développer des « traducteurs » qui assureraient la conversion des données provenant de formats divers vers MathML, et réciproquement. Et le format le plus couramment utilisé pour l'édition d'équations mathématiques étant T_EX, il nous a semblé évident de commencer par celui-ci.

Ce rapport ressasse d'en un premier lieu les notions XML, XSL et MathML qu'il nous a fallu acquérir afin de développer correctement le parser et le moteur de rendu MathML. Par la suite, nous présenterons l'architecture de nos applications.

XML

Présentation du langage et des technologies associées

> *« Pensez cous que le soleil est ébloui par sa propre lumière ? »,*
> *James Hinton*

Pour bien comprendre MathML et les applications que nous avons développées, il est nécessaire d'être familier avec l'univers XML. En effet, il s'agit d'un monde vaste et parfois complexe, qui se cherche encore et qui ne cesse d'évoluer. Nous allons donc présenter ici XML et ses acolytes tel que nous les comprenons et que nous les entendons, afin que vous puissiez par la suite vous immerger plus facilement dans cet univers…

Le langage XML

XML est l'acronyme de **eXtensible Markup Language**. En français, cela correspond à langage de « balisage » extensible. XML représente une série de règles pour définir des balises ayant une valeur sémantique et pour structurer un document en différentes sections et identifier ces sections. C'est un langage de **méta-description**, car il propose une syntaxe qui lui permet de définir à son tour d'autres langages de description structurés plus spécifiques à certains domaines (au mathématiques, par exemple…). Ces langages de description « dérivés » sont appelés des applications XML. Nous allons essayer dans cette partie, de présenter de façon rapide et condensée ces applications. Mais tout d'abord, replaçons XML dans l'histoire des documents électroniques…

Historique

SGML (Standard Generalized Markup Language, ou langage normalisé de balisage généralisé, ce qui n'est pas beaucoup plus parlant), adopté comme standard en 1986, a été la première tentation de création de réels documents électroniques, c'est-à-dire des documents qui n'étaient plus des documents papiers sous forme électronique. La principale idée sous-jacente était de séparer le **contenu** (logique) d'un document de sa **forme** (matérielle/imprimée). Mais l'intention finale était toujours principalement de produire des documents imprimés, quoique plus économiquement — un unique document (logique) étant transformé automatiquement en différents formats imprimés (par ex. des documents de maintenance pour des avions produits en différents formats selon les normes de présentation de différentes compagnies aériennes ou armées de l'air). SGML a été une percée, mais il était si complexe que sa manipulation s'est trouvée restreinte aux spécialistes (rédacteurs techniques dans l'industrie ou spécialistes textuels dans les humanités).

A partir de 1992, le Web et HTML (Hypertext Markup Language, ou langage de balisage hypertexte, conçu vers 1990) sont devenus une réalité, et ont popularisé les documents électroniques hypertextuels sur une immense échelle. Depuis 1995, les moteurs de recherche ont démontré la stupéfiante capacité de recherche d'information rendue possible par le Web.

Ainsi, avant l'apparition de XML, existaient :

- Un langage de balisage normalisé, riche en sémantique mais relativement lourd à mettre en oeuvre et inadapté au Web : **SGML**

- Un langage parfaitement adapté au Web (puisque développé uniquement pour cette application) mais dont les applications sont limitées par une bibliothèque de balises figée et réduite : **HTML**

Il convenait donc de définir un langage qui ait la facilité de mise en oeuvre de HTML tout en offrant la richesse sémantique de SGML. C'est la raison d'être de XML. XML est un sous-ensemble au sens strict de SGML, dont il ne retient pas les aspects trop ciblés sur certains besoins. En cela il représente un "profil d'application" de la norme SGML.

Pourquoi XML et plus HTML ?

Vous devez d'abord prendre conscience que XML ne se place pas à la suite de HTML, mais bien avant. HTML définit un ensemble de balises figées une bonne fois pour toutes pour décrire les éléments. Si vous ne trouvez pas la balise dont vous avez besoin, vous ne pouvez que râler contre le créateur du langage et espérer que la prochaine version comblera votre attente.

C'est l'énorme différence avec XML. Etant un langage de description, c'est vous qui créez selon vos besoins les balises dont vous voulez vous servir. Par, exemple, dans notre cas, nous avons voulu regrouper toutes les informations traditionnelles que comporte un document mathématique. Le consortium mis en place pour définir MathML se contente donc de recenser ces informations et de les structurer. Il ne se perd pas dans les détails des paragraphes, des listes d'énumérations, des mises en gras et autres catégories non liées au sens des données, mais à leur présentation.

Cela nous conduit naturellement vers le second point essentiel concernant XML, à savoir que ce langage décrit **uniquement** la structure et la signification du contenu d'un document. Il ne s'intéresse nullement au formatage des éléments tels qu'ils peuvent être lus ensuite, ce formatage faisant l'objet d'une feuille de style. De façon diamétralement opposée, le langage HTML mélange formatage, structure et sémantique ; une balise <H1> signifie qu'il s'agit par exemple d'une police Helvetica gras en 20 points, d'un niveau de titre 1 et du titre de la page.

HTML est un format de document prévu pour être utilisé sur Internet et chargé par un navigateur Web. XML permet bien sûr cela aussi. Mais XML est bien plus vaste ; vous pouvez vous en servir comme format de stockage pour un traitement de texte, comme format d'échange entre programme, comme moyen de forcer la conformité avec des modèles intranet ou encore comme moyen de stocker des données sous un format restant lisible par les êtres humains.

Conformité de XML - DTD

XML comporte deux niveaux de conformité :le premier correspond à la bonne formulation et le second à la validité. La bonne formulation est un critère minimum obligatoire pour permettre à un processeur de lire un fichier XML. La spécification XML interdit d'ailleurs aux programmes d'analyse XML d'essayer de réparer ou de comprendre des documents mal formulés ; la seule chose qu'il est autorisé à faire est de rendre compte de l'erreur. La validité est une deuxième contrainte que peut s'imposer le programmeur XML s'il veut mettre son application en conformité avec celle définit par un groupe. Les règles à respecter sont alors définit à travers ce que l'on appelle une DTD, une Définition Type de Document.

Mais énonçant d'abord les huit commandements qu'un document XML doit respecter (en plus du respect de la casse des lettres) pour être bien formé :

- le document doit commencer par une déclaration XML,
- les éléments contenant des données doivent posséder une balise ouvrante et une balise fermante,
- les éléments vides ne comportant qu'une balise doivent la clore par />,
- le document doit contenir un et un seul élément racine qui est le parent de tout les autres,

- les éléments peuvent s'imbriquer, mais pas se chevaucher,
- les valeurs des attributs doivent être délimitées par des guillemets (ou des apostrophes),
- les méta-caractères < et & ne doivent être utilisés que pour démarrer respectivement une balise et une référence d'entité,
- les seules références d'entité qui doivent apparaître sont les cinq définies dans la norme (&, <, >, ', et ").

On se rend compte au vu de ses quelques règles que XML et beaucoup plus rigide que HTML. En effet, certaines balises ne se ferment jamais, la gestion des attributs est parfois bizarre… On rencontre souvent ces problèmes dans les pages Web :

- Absence de balises fermantes (éléments non appariés),
- Balises fermantes sans balises ouvrantes,
- Eléments en chevauchement,
- Attributs non délimités,
- Méta-caractères non encodés par références d'entités (comme les guillemets),
- Pas d'élément racine.

Ces négligences sont à l'origine de nombreux problèmes d'affichage sur le Web. Les créateurs d'XML ont donc imposé une syntaxe plus rigide afin de ne plus rencontrer ce genre de défauts.

Parlons maintenant de la DTD. Cette acronyme signifie littéralement Document Type Definition, soit définition type de document en bon français. Une telle définition fournit la liste des éléments, des attributs, des notations et des entités que contient le document ainsi que les règles des relations qui les régissent. Une DTD définit un ensemble de règles correspondant à la structure du document. Un document est dit valide s'il est conforme à la DTD associée.

L'intérêt d'une DTD est multiple. Tout d'abord, elle nous évite de réinventer la roue à chaque création de document XML. En effet, comme nous l'avons dit, XML est un méta-langage. Chacun est donc libre de créer sa propre version XML d'un document mathématique, musicale, etc. La DTD nous fournit donc des structures déjà validées par d'autres utilisateurs.

Mais le gros avantage d'une DTD, comme la notion de document valide, l'indique, c'est qu'elle permet en effet de valider un document !! En effet, un parser validera un document si :

- Ce document respecte bien la syntaxe du langage XML (il est correctement formé),
- Ce document respecte les règles de construction définies par sa DTD,
- Toutes les références à des entités peuvent être résolues.

Enfin, le troisième avantage des DTD se situe par rapport aux feuilles de style. En effet, en définissant une feuille de style par rapport à une DTD, nous pourront l'appliquer directement à tout les documents valides. En revanche, dans le cas contraire, il nous faudrait reconstruire une feuille de style spécifique à chaque nouveau document, ce qui peut s'avérer très complexe et très long. En définissant des feuilles de style par rapport à la DTD MathML, nous avons l'assurance que celles-ci s'appliqueront à tout les document XML valides, qu'ils

soient rédigés par notre main ou non. Cependant, sans rentrer pour l'instant dans les détails, il existe trois problèmes majeurs concernant les DTD :

- Sa difficulté d'écriture : en effet, n'étant pas une application XML, les DTD sont extrêmement mal construites, elles en deviennent illisibles lorsque elles sont complexes, et difficiles à mettre à jour,
- Le non héritage : une DTD ne peut pas hériter d'une autre,
- Le non typage des attributs : il est impossible de définir le type de la valeur d'un attribut.

Nous allons à présent introduire différentes technologies dérivées ou associées à XML, du moins celles qui sont approchés par notre application. Les plus importantes sont bien évidemment DOM et XSL mais nous reviendrons aussi sur les XML schemas, qui ne sont pas encore validés par le W3C mais qui constitue une alternative intéressante aux DTD.

DOM

Pour comprendre ce qu'est la DOM, il faut d'abord intégrer le principe de validation des documents. Pour traiter des données XML, toute application doit être interfacé avec une API. Actuellement, il en existe de deux types, qui correspondent chacune un modèle de traitement différent. Le premier modèle consiste à considérer le document XML comme un flot de données en entrée, et à reconnaître les balises au fur et à mesure qu'elles se présentent. Lorsqu'un nom connu est intercepté, l'élément est passé à l'interpréteur qui exécute alors l'action associée. Ce modèle est qualifié d'événementiel, et l'API standard qui implémente ce modèle et le rend disponible aux applications est appelé SAX,, *Simple API for XML*.

Le second modèle cherche quant à lui à compiler l'ensemble du document et à le représenter en interne sous forme d'arbre. L'API propose alors à l'application un ensemble de fonction lui permettant de parcourir cet arbre. L'API qui implémente ce modèle dit « objet » est appelé DOM, *Document Object Model*. Ce modèle est une norme du W3C, qui va spécifier des règles pour construire, supprimer, et accéder aux différents nœuds de cet arbre. La DOM est défini indépendamment de tout langage de programmation particulier.

Les différents types de nœuds, et donc d'interfaces, sont les suivants :

- L'interface **Node** : il s'agit de l'interface la plus générale, qui permet de représenter tout type de nœud. Elle définit un ensemble de méthodes génériques à l'ensemble des nœuds, telle que l'insertion d'un fils, la duplication, etc. Toutes les interfaces dérivent de `Node`.
- Le nœud **Document** : une instance de `Document` représente la racine d'un document complet. L'interface propose des méthodes pour créer un élément, du texte, un commentaire, une instruction de traitement, etc. un nœud `Document` ne peut avoir qu'un seul nœud de type `Elément`, l'élément racine du document.
- Le nœud **DocumentType** : tout document a un attribut nul ou contenant une référence à un objet instance de `DocumentType`. Cette interface propose des attributs permettant l'accès aux entités et aux notations déclarées dans le document.
- Le nœud **Notation** : il représente les déclarations de notation. L'attribut `NodeName`, hérité de `Node`, permet de récupérer le nom de la notation.

- Le nœud **EntityReference** : représente une référence à une entité.

- Le nœud **Entity** : représente une entité dans un document XML. L'attribut `NodeName` hérité de `Node` contient le nom de l'entité. La liste des fils est la même que pour un nœud `EntityReference`, ayant la même valeur pour l'attribut `NodeName` (à savoir des éléments, des instructions de traitements, des commentaires, du texte, etc…).

- Le nœud **Elément** : il propose des méthodes pour retrouver la valeur d'un attribut d'un élément, pour créer un nouvel attribut ou en supprimer un existant. Une méthode permet également de retrouver tous les descendants de l'élément ayant un certain nom de type.

- Le nœud **Attribute** : un objet `Attribute` n'est pas un fils d'un élément pour lequel il est défini. Il ne s'insère donc pas dans l'arbre des nœuds qui représentent la hiérarchie des éléments du document. La valeur d'un nœud `Attribute` est déterminée conformément aux règles du langage XML : si une valeur explicite est fournie, elle est utilisée. Si aucune valeur n'est présente, mais qu'une valeur par défaut est définie dans la DTD, la valeur par défaut est utilisée. Sinon, le nœud attribut est absent du Document, tant qu'il n'est pas créer avec une valeur. La valeur d'un attribut est représentée par un nœud de type `Text`, fils du nœud `Attribute`.

- Le nœud **Text** : il représente le contenu textuel des éléments et des attributs. Il propose la méthode `splitText` qui découpe le contenu textuel à partir d'une certaine position et crée un nouvel objet `Text`, frère de l'objet depuis lequel la méthode a été invoquée. Son interface dérive de l'interface `Data`, qui ne correspond à aucun objet de la DOM. ELle ne sert qu'à définir différentes méthodes qui sont héritées par des interfaces spécialisées représentant des nœuds à contenu textuel, à savoir les nœuds `Text` et `CDATASection`.

- Le nœud **CDATASection** : le contenu d'un objet instance de cette classe est représentée par un nœud fils de type `Text`.

De plus, la DOM définit deux interfaces très pratique pour parser le document et pour stocker les données :

- **NodeList** : une liste ordonnée de nœuds. Cette interface comprend deux méthodes :

    - La méthode `getLength()` qui renvoie le cardinal de la liste,

    - La méthode `item(int index)` qui renvoie le index-ième nœud de la liste.

- **DOMImplementation** : cette interface, qui n'a rien à voir avec la structure du document, ne sert qu'à vérifier si l'implémentation DOM utilisée accepte les modèles HTML, XML, MathML, etc.

XML Schemas

Même si nous ne reparlons pas par la suite des XML schemas (notre parseur ne les implémentera pas pour le moment), il nous semble important d'en présenter ici la philosophie et la pertinence, car ceux ci pourront sans doute résoudre quelques problèmes posés par la recommandation MathML.

Partant du principe que les DTD ne permettaient pas de contraindre suffisamment les données et les structures, partant également du principe qu'il n'y avait aucune raison pour que les modèles ne soient pas codés sous forme XML, les XML Schemas proposent une méthode de réalisation de modèles, alternative aux DTD.

L'objectif des Schemas est de définir des contraintes sur des classes de documents conformes à un même modèle. À la différence des DTD, qui ne définissaient *que* les relations entre les différents composants d'un document, les Schema peuvent typer les données et, par là-même, en documenter leur sens et, donc, leur utilisation. L'objectif n'est pas d'avoir un langage extensible permettant d'exprimer n'importe quelle contrainte : seules, les contraintes consensuelles sont exprimables. Celles-ci doivent pouvoir être validées par un parseur et être mises en jeu pour assister à la création d'information, surtout dans les éditeurs.

Enfin, les Schema ne s'intéressent pas à la validation d'un document XML (d'un fichier) en tant que tel. Un outil validant se basera sur la représentation en arbre d'objets typés et *valués*, *désérialisées* selon le standard Infoset[1]. Cette volonté permet de rendre le processus de validation indépendant d'une syntaxe de codage quelconque d'un document.

Les spécifications sur les Schema sont divisées en deux parties : une sur les *structures* et une sur les *types de données*.

D'un point de vue structurel, un Schema se définit comme étant l'assemblage, sous forme d'arbre XML, d'un ensemble de composants. Il en existe 12, dont les principaux sont :

- *Les composants de définition de types*, simples ou complexes : ils permettent de typer des éléments pour leur affecter un modèle de contenu. Il en va de même pour les attributs. Ainsi, une adresse de livraison et une adresse de facturation relèveront toutes deux d'un même type adresse. ET c'est en utilisant les types simples que seront définis, dans la partie "datatypes", des types de base comme les chaînes de caractères, les nombres, les URI ou encore toutes sortes de dates.

D'un point de vue programmation, les notions de types de données permettent de se reposer sur ces types de données pour effectuer des traitements partagés à différents éléments du même type. Ceci simplifie beaucoup les méthodes de programmation, tout en nécessitant d'avoir toujours accès au modèle, en même temps que l'on effectue les traitements.

- *Les composants de déclaration* : éléments, attributs et notations. Ces composants permettent de déclarer, comme avec les DTD, des éléments, avec leurs modèles de contenus, et ainsi que des attributs. Pour ce faire, et dans le but de factorisation précédemment expliqué, les déclarations peuvent se reposer sur des composants de définition de type.

- *Les composants groupes* : pour la factorisation d'informations de modèles de contenus ou d'attributs. Ces composants permettent de définir des modèles qui seront souvent réutilisés, soit pour définir des composants de déclaration, soit pour définir des composants de définition de types. Dans les DTD traditionnelles, ces groupes étaient formalisés par des entités paramètres, qui permettaient, par exemple, de définir tous les composants blocks (paragraphes, listes, remarques, etc.) nécessaires lors de la définition de modèles de contenus.

D'un point de vue typage de données, la spécification propose un ensemble important de types et une façon d'étendre ces types, pour une application donnée, à des choses plus complexes.

[1] Norme en cours de validation par le W3C. voir le glossaire pour plus d'information

Finalement, un Schema est un modèle d'information défini sous forme électronique (comme l'était une DTD), mais avec des outils de modélisation plus puissants. Dans la sphère XML, le Schema pourra être utilisé comme outil de validation interactif de données, lors de la création de celles-ci, dans un éditeur, voire, plus tard, dans des formulaires, sur Internet.

Il serait peut-être bon ici de montrer un petit exemple pour rendre les choses plus claires. Prenons l'exemple simple d'un carnet d'adresse, où l'on définit plusieurs individu en donnant leur nom, prénom, adresse, etc. Le ficher XML correspondant est le suivant :

Source du carnet d'adresse

```
<?xml version="1.0" encoding="UTF-8"?>
<CARNET-ADRESSE DATE-CREATION="2001-06-13" DATE-MAJ="2001-06-14">
   <ADRESSE>
      <IDENTITE>
         <NOM>Machocki<NOM><PRENOM>Wojciech<PRENOM>
      </IDENTITE>
      <COORDONNEES>
         <ADRESSE-POSTALE>
            <NUMERO>72</NUMERO><RUE>avenue du 8 Mai 1945 </RUE>
            <CODE-POSTAL>54400</CODE-POSTAL><VILLE>LONGWY</VILLE>
            <PAYS>FRANCE</PAYS>
         <ADRESSE-POSTALE>
      </COORDONNEES>
   </ADRESSE>
   <ADRESSE> … </ADRESSE> …
</CARNET-ADRESSE>
```

La DTD que l'on pourrait associer à cet exemple est :

Listing de la DTD associée

```
<!ELEMENT CARNET-ADRESSE (ADRESSE+)>
<!ATTLIST CARNET-ADRESSE
          DATE-CREATION  CDATA #REQUIRED
          DATE-MAJ       CDATA #REQUIRED >

<!ELEMENT ADRESSE (IDENTITE,COORDONNEES,COORDONNEES?,NOTE*)>

<!ELEMENT IDENTITE (NOM,PRENOM?,SURNOM?)>

<!ELEMENT NOM (#PCDATA)>
<!ELEMENT PRENOM (#PCDATA)>
<!ELEMENT SURNOM (#PCDATA)>

<!ELEMENT COORDONNEES (ADRESSE-POSTALE,TELEPHONE*,INTERNET?)>

<!ELEMENT ADRESSE-POSTALE (NUMERO,RUE,CODE-POSTAL,VILLE,PAYS)>

<!ELEMENT NUMERO (#PCDATA)>
<!ELEMENT RUE (#PCDATA)>
<!ELEMENT CODE-POSTAL (#PCDATA)>
<!ELEMENT VILLE (#PCDATA)>
<!ELEMENT PAYS (#PCDATA)>
```

Cette DTD n'est certes pas très complexe, et nous la comprenons aisément, mais nous voyons bien qu'elle deviendrait indécodable si l'on augmentait le nombre de types de donnée. Il est de plus impossible de définir le

contenu de l'élément numéro comme étant un entier, de spécifier la structure de l'attribut DATE-CREATION, etc. Parallèlement, le Schema correspondant serait le suivant :

listing du Schema correspondant

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2000/10/XMLSchema">
  <xs:element name="CARNET-ADRESSE">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ADRESSE" minOccurs="1"
maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="DATE-CREATION" use="required"
type="xs:date"/>
      <xs:attribute name="DATE-MAJ" use="required"
type="xs:date"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="ADRESSE">
    <xs:complexType >
      <xs:sequence>
        <xs:element ref="IDENTITE"/>
        <xs:element ref="COORDONNEES" minOccurs="1"
maxOccurs="2"/>
        <xs:element ref="NOTE" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="IDENTITE">
    <xs:complexType >
      <xs:sequence>
        <xs:element ref="NOM"/>
        <xs:element ref="PRENOM" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="SURNOM" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
.
.
.
…</xs:schema>
```

On reconnaît bien ici la structure classique d'un document XML, avec l'entête, l'élément racine, la syntaxe XML, les attributs,… On retrouve aussi la structure propre aux Schemas, à savoir la présence des composants de déclaration et des composants de déclaration de type complexe. De même, il est possible de spécifier le type de laveur prise par l'attribut DATE-CREATION. Cette structure est, on le voit bien, assez souple et plus facile à définir que les DTD.

MathML

Mathematical Markup Language Version 2.0

> *« Et c'est comme s'il disait : Tout commence »*
>
> *Nikos Kazantzaski, la Dernière Tentation Du Christ*

Maintenant que les notions XML sont acquises, nous pouvons nous intéresser d'un peu plus près à MathML, un standard dérivée d'XML, adopté comme recommandation par le W3C en 1998, et dont la version 2.0, dernière version en cours, a été adoptée le 26 février 2001. Nous allons présenter la philosophie MathML, et nous reviendrons sur la structure de certains éléments que nous jugeons primordiaux.

Généralités

MathML est une application de notation mathématique, dont le but est d'en récupérer à la fois la *structure* et le *sens*. MathML doit permettre aux mathématiques d'être desservies, reçues et traitées sur le Web, comme HTML l'a permis pour le texte. Parallèlement, il doit permettre à des logiciels de calculs formels comme Maple ou Matlab, ainsi qu'à des applications classiques développées dans des langages standards (C, Java, LISP, etc.) d'échanger et d'interpréter des objets mathématiques complexes. MathML doit :

- Encoder de la substance mathématique adaptée à la communication scientifique et mathématique de tout niveau,
- Encoder la notation et la sémantique,
- Faciliter la conversion de et vers d'autres formats de présentation et de sémantique mathématique. Les formats de sorties doivent comprendre :
    - Des représentations graphiques
    - Des synthétiseurs vocaux
    - Des données traitables par les systèmes algébriques
    - D'autres langages mathématiques (e.g. TeX)
    - Des représentations « plain text »
    - Des affichages divers, y compris le braille

RQ : *Il est reconnu que la conversion de ou vers d'autres systèmes de notation ou médias peut entraîner une perte d'information.*

- Supporter les longues expressions à la navigation
- Permettre le passage d'information voulue pour des « rendeurs » spécifiques et des applications,
- Permettre d'être étendu,Etre adapté aux templates et autres techniques d'édition,
- Etre humainement lisible,
- Etre facile à gérer et à traiter par les logiciels,De plus, le W3C Math Working

Group a identifié une brève liste d'objectifs supplémentaires d'implémentation. Cette liste a pour but de décrire rapidement les fonctionnalités élémentaires que les moteurs de rendu et les logiciels de traitement MathML devrait fournir :

- •Les équations dans les pages HTML doivent avoir un rendu correct dans les navigateurs, en accord avec les préférences d 'affichage des auteurs et des lecteurs, et avec une qualité optimale au vue des capacités de chaque plate-forme,
- •Les pages HTML contenant des équations MathML doivent être imprimer correctement et à la résolution optimale de l'imprimante (idem pour le braille, les synthétiseurs vocaux, etc.),
- •Les équations MathML dans une page Web doivent être capables de réagir au comportement de l'utilisateur (clic d'un souris par ex.), et de gérer la communication avec d'autres applications, à travers le navigateur,
- •Des éditeurs d'équation et des « convertisseurs » doivent être développés pour faciliter la création de pages Web contenant des équations MathML.

En outre, comme nous l'avons dit, l'un des objectifs de MathML est la diffusion d'information mathématique sur Internet. Par conséquent, il se doit d'être en relations avec les autres technologies Internet connues. Il faut plus particulièrement que :

- •Les langages de balises mathématiques existants puissent être convertis en MathML,
- •L'on puisse ajouter du MathML dans les pages HTML pour que les applications Web qui manipulent dés aujourd'hui du HTML puissent dans le futur manipuler facilement du MathML,
- •MathML puisse être rendu dans les navigateurs existants de quelque façon que ce soit (e.g. par images), même si cela est loin d'être idéal.

Syntaxe et grammaire

MathML est une application XML. Par conséquent sa syntaxe est en partie dictée par la syntaxe XML et sa grammaire est spécifiée par une DTD. Ainsi, nous retrouvons les éléments classiques de la syntaxe XML, à savoir :

- Les caractères Unicode (incluant les caractères ASCII ordinaires),
- Les entités,
- La structure de ses éléments,

- Les attributs,
- La casse des chaînes de caractères.

Cependant, le W3C, comme nous l'avons vu, dans sa volonté d'encourager la modularisation des applications construites avec XML, trouve que la forme actuelle d'une DTD ne convient pas exactement et un groupe fut créé afin de développer des spécifications pour les XML Schemas. MathML étant conçu pour que les mathématiques puissent tirer avantage des dernières évolutions technologiques en matière de Web, il doit donc y avoir un Schema pour MathML, ce qui est le cas.

De plus, MathML définit aussi certaines règles de syntaxe et de grammaire qui lui sont propres, lui permettant d'encoder plus d'information, sans pour autant introduire beaucoup plus d'éléments, sans utiliser une DTD ou un Schema plus complexe. Evidement, l'un des inconvénients de ces règles spécifiques à MathML, est qu'elles sont totalement transparentes pour un processeur XML classique. Ces règles sont de deux types :

- Des critères supplémentaires sur la valeur des attributs,
- Des restrictions supplémentaires sur les éléments fils (des règles d'ordre par exemple).

L'ordonnancement des éléments fils est très pratique (voir indispensable) dans bon nombre de cas, car il évite des problèmes de compréhension sans surcharger la syntaxe XML. Par exemple, une fraction est constituée de deux éléments principaux : le numérateur et le dénominateur. MathML fournit donc une balise `mfrac` possédant seulement deux fils pouvant être de tout type, le premier étant obligatoirement le numérateur, le deuxième étant obligatoirement le dénominateur. $\frac{a}{b}$ s'écrit donc :

Exemple a/b :

```
<mfrac>
   <mi> a </mi>
   <mi> b </mi>
<mfrac>
```

Les informations supplémentaires sur la valeur des attributs résident essentiellement sur le typage de ces valeurs. Ce problème peut être géré en préférant les Schemas aux DTD. Il peut être aussi traiter lors du parsage en demandant à l'analyseur de vérifier les données passées en valeur des attributs. Le non-respect de ces règles entraîne une erreur MathML, bien qu'il ne gène par le parsage XML.

Les notations utilisées pour décrire les types de ces valeurs sont les suivantes :

number	Entier ou rationnel, pouvant commencer par l'opérateur binaire '-'
unsigned-number	Entier ou rationnel non signé
integer	Entier, pouvant commencer par l'opérateur binaire '-'
positive-integer	Entier non signé et non nulle
string	Chaîne de caractères quelconques (toujours la totalité de valeur de l'attribut)
character	Caractère seul, quelconque (excepté l'espace), ou une entité.
#rrggbb	Code RVB d'une couleur
h-unit	Unité de longueur (la liste des unités est donnée ci-dessous)

v-unit	Unité de hauteur (la liste des unités est donnée ci-dessous)
css-fontfamily	unité CSS
css-colorname	unité CSS
form+	Une ou plusieurs instances de « form »
form*	Zéro ou plusieurs instances de « form »
$f_1\ f_2 \ldots f_n$	Une instance de chaque f_i, séparés ou non par des espaces
$f_1 \mid f_2 \ldots \mid f_n$	N'importe lequel des f_i spécifiés
[form]	0 ou 1 instance de form
(form)	1 instance de form
mot « plain text »	Ce mot littéralement présent dans la valeur de l'attribut
quoted symbol (e.g. '+' ou "+")	Ce symbole, littéralement présent dans la valeur de l'attribut

L'ordre de priorité des opérateurs de cette syntaxe est la suivante (du plus fort au plus faible) :

- form+ ou form*
- $f_1\ f_2 \ldots f_n$ (séquence)
- $f_1 \mid f_2 \ldots \mid f_n$ (alternative)

Dans la valeur des attributs, les mots-clés et les nombres doivent être séparés par des espaces, excepté les « unités » (h-unit, v-unit) qui suivent les nombres. Pour les valeurs des attributs, l'espace n'est pas particulièrement requis, mais il est autorisé dans toutes les expressions exposées précédemment, excepté devant une unité (pour une comptabilité avec CSS1), entre le signe '-' et la valeur du nombre négatif, et entre # et rrggbb.

Le signe '+' n'est pas autorisé dans la syntaxe d'un nombre entier ou rationnel positif, sauf s'il est explicitement demandé.

Comme nous l'avons dit dans le tableau, quelques attributs acceptent comme valeur des hauteurs et des largeurs (`h-unit` et `v-unit`) qui sont généralement formés d'un nombre suivit d'une unité. `h-unit` et `v-unit` se réfèrent donc à une unité pour exprimer respectivement la hauteur et la largeur lors du rendu des balises. Les unités possibles sont :

em	Relatif à la largeur de la police courante
ex	Relatif à la hauteur de la police courante
px	Pixels (1 pixel = 2.54 cm)
in	Inches
cm	Centimètres
mm	Millimètres
pt	Points (1 point = 1/72 inch)

pc	Picas (1 pica = 12 pts)
%	Pourcentage de la valeur par défaut

Ces unités sont tirées des spécifications de CSS. Cependant, la syntaxe MathML reste différente sur ce point de la syntaxe CSS, puisque les nombres dans les feuilles de styles CSS ne peuvent être rationnels, et peuvent être précédés du signe '+'.

Lorsqu'il n'y a pas d'unité, le nombre donné sert de multiplicateur à la valeur par défaut.

Exemple :

```
<mo maxsize = "2"> … </mo>

   équivaut à

<mo maxsize = "200%"> … </mo>
```

Cependant, pour rester compatible avec CSS, il est rare que les unités de longueur soit optionnelles. Toujours par compatibilité avec CSS, 0 n'a pas besoin d'être suivie d'une unité, même si la syntaxe en spécifie une.

Pour faciliter la comptabilité avec CSS1, tous les éléments MathML acceptent les attributs `class`, `style` et `id`, en plus des attributs définis pour chaque élément. Les moteurs de rendu ne supportant pas CSS se doivent d'ignorer ces attributs. La valeur de ces attributs est définie comme étant une chaîne de caractère.

Tous les éléments MathML acceptent aussi l'attribut `other`, qui permet de passer des attributs non-standards sans violer la DTD MathML. Cet attribut peut s'avérer pratique, mais il est fortement déconseillé de l'utiliser, car dans la plupart des cas il est possible de trouver dans MathML d'autre façon de transmettre ces informations.

Nous allons maintenant présenter succinctement les principales caractéristiques des balises MathML, sans rentrer dans le détail, de telle sorte que si vous ne souhaitez pas en savoir beaucoup plus sur MathML, vous puissiez vous arrêter là. Par la suite, nous reviendrons bien sûr plus en détail sur les différents éléments, mais le but de la section qui suit est de donner une première approche des balises MathML.

Les balises MathML

Toutes les balises MathML appartiennent à l'une des catégories suivantes :

- Les balises de présentation,
- Les balises de contenu,
- Les balises d'interface.

Les balises de Présentation

Cette classe, composée d'environ 30 éléments acceptant un peu plus de 50 attributs, regroupe l'ensemble des éléments censés représenter des notations mathématiques. De manière générale, chaque élément de présentation correspond à plusieurs types de structure de notation, comme des intégrales, des exposants, des indices, des matrices, etc. Toutes les formules mathématiques seront mises en formes grâce à ces balises emboîtées les unes dans les autres, avec comme extrémités de l'arbre résultant les éléments simples tels que des symboles, des chiffres, des caractères… les balises de présentation sont donc de deux types :

- **Les token elements** : ce sont les symboles, les noms, les nombres, etc. En général, ils n'ont que des caractères et des éléments `mchar` comme fils. Tous les identificateurs, les nombres, les opérateurs, doivent être présentés à l'aide de ces éléments. Ils permettent aussi de représenter du texte ou des espaces qui ont plus un intérêt esthétique que réellement significatif, et pour représenter des chaînes de caractères afin de les rendre compatibles avec des systèmes algébriques.

- **Le Layout schemata** : permet de construire des expressions. Par conséquent vous comprendrez aisément que l'on nomme ces éléments des « constructeurs d'expression ». Ces expressions spécifient dans quel ordre les sous-expressions sont construites et intégrées (d'où l'importance du nombre et de la disposition de ses enfants). On appelle aussi un enfant de ces constructeurs un *argument*.

Les constructeurs peuvent eux-mêmes être regroupés en plusieurs classes :

- Les éléments génraux : `mrow`, `mstyle`…
- Les scripts : `msup`, `munder`, …
- Les tableaux : `mtable, mtr,` …
- `maction.`

Nous pouvons présenter un exemple simple pour voir la syntaxe de ces éléments. Prenons les racines d'un polynôme de second degré ; on obtient $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, ce qui en MathML donne[2] :

Balise de présentation

```
<mrow>
   <mi>x</mi>
   <mo>=</mo>
   <mfrac>
      <mrow>
         <mrow>
            <mo>-</mo>
            <mi>b</mi>
         </mrow>
         <mo><mchar name="PlusMinus"/></mo>
         <msqrt>
            <mrow>
```

[2] l'ensemble des exemples est tiré de la recommandation MathML 2.0 du W3C

```
                <msup>
                   <mi>b</mi>
                   <mn>2</mn>
                </msup>
                <mo>-</mo>
                <mrow>
                   <mn>4</mn>
                   <mo><mchar name="InvisibleTimes"/></mo>
                   <mi>a</mi>
                   <mo><mchar name="InvisibleTimes"/></mo>
                   <mi>c</mi>
                </mrow>
             </mrow>
          </msqrt>
       </mrow>
       <mrow>
          <mn>2</mn>
          <mo><mchar name="InvisibleTimes"/></mo>
          <mi>a</mi>
       </mrow>
    </mfrac>
</mrow>
```

La balise mrow

Cet élément est utilisé pour regrouper un nombre quelconque de sous-expressions, généralement conçus d'un ou plusieurs opérateurs (de type `mo`) agissant eux même sur différentes expressions. La plupart des éléments agissent par défaut comme si leurs arguments étaient contenus dans un élément `mrow`. Cet élément accepte seulement les attributs communs à l'ensemble des balises MathML, à savoir id, `xref`, `class` et `style`.

Les éléments `mrow` sont rendus, comme leur nom l'indique, comme étant une rangé horizontale dans laquelle sont insérées des expressions (de gauche à droite), et par les synthétiseurs vocaux comme une séquence d'expression. Mais cette balise n'est pas responsable du rendu des espaces entre les opérateurs et les sous-expressions. En effet, les opérateurs mo se chargent de fournir cette information. De même, cette balise n'indique pas directement où le rendeur peut « couper » les expressions et revenir à la ligne (par exemple lorsque l'expression est trop longue pour être visualisée sur une seule ligne), mais la présence des éléments `mrow` donne des points de repère sémantique à l'interpréteur pour qu'il puisse déterminer ou séparer l'expression.

Si les balises `mrow` ne contiennent qu'un seul argument, tout se passe comme si l'élément était seul, sans être encapsulé dans des balises `mrow`. Cette équivalence est présente pour simplifier la vie des développeurs d'outils MathML. De même, si la balise `mrow` redéfinit des attributs (l'attribut style par exemple), il n'est pas demandé, dans la spécification MathML, d'exercer des règles de rendu particulières, mais les nouvelles directives d'affichage induites devront se répercuter sur l'argument.

Il n'existe aucune règle grammaticale pour l'insertion et l'utilisation des balises `mrow`. En fait, tout est laissé à l'appréciation de l'auteur. Cependant, il existe quelques recommandations pour l'utilisation de cet élément. L'intérêt de ces règles est triple :

- Améliorer l'affichage en associant des espacements aux `mrow` lors du rendu,
- Favoriser les sauts de lignes lorsque cela est nécessaire au sein d'une même équation,
- Améliorer la compréhension sémantique lorsque l'on transmet le document MathML a un interpréteur algébrique, ou à des synthétiseurs vocaux.

Généralement, on regroupera un opérateur avec l'ensemble de ses arguments, de manière à rendre les choses plus lisibles et donc plus compréhensible. Mais voici les règles précises que l'on doit suivre :

> ***Deux opérateurs adjacents, séparés ou non par des expressions (i.e. tout ce qui n'est pas un opérateur), appartiennent au même élément `mrow` si et seulement si les deux opérateurs ont la même priorité, et si ces opérateurs appartiennent au dictionnaire des opérateurs fournit dans la recommandation MathML (appendice F). Dans tous les autres cas, les opérateurs seront dans des `mrow` distincts.***

On entend par 'opérateur' l'ensemble des opérateurs algébriques, mais aussi des parenthèses et des séparateurs.

En respectant cette règle, l'exemple 2x +y – z s'écrit :

2x +y – z

```
<mrow>
  <mrow>
    <mn> 2 </mn>
    <mo> &InvisibleTimes; </mo>
    <mi> x </mi>
  </mrow>
  <mo> + </mo>
  <mi> y </mi>
  <mo> - </mo>
  <mi> z </mi>
</mrow>
```

de même, (x, y) s'écrit :

(x, y)

```
<mrow>
  <mo> ( </mo>
  <mrow>
    <mi> x </mi>
    <mo> , </mo>
    <mi> y </mi>
  </mrow>
  <mo> ) </mo>
</mrow>
```

la balise maction

La balise `maction` permet d'associer une action particulière à une expression. Cela permet d'ajouter une certaine interactivité au document. L'élément `maction` peut avoir un nombre quelconque d'arguments, et il possède les deux attributs suivants :

- **`actiontype`** : cet attribut est indispensable, son absence génère une erreur lors de l'analyse du document. Il n'y a pas de valeur par défaut, nous allons lister ci-dessous les valeurs possibles.

- **`selection`** : de type entier, cet attribut définit le nombre de sous expressions concernées par l'action liée. Par défaut, cette valeur est 1.

La valeur de l'attribut `actiontype` n'est pas formalisée, elle est fortement dépendante du type d'application utilisée. Par conséquent, notre parser ne vérifiera que la présence de cet attribut, sans se soucier de la syntaxe de sa valeur. Pour indication, des valeurs possibles sont :

- **`toggle`** : cela permet d'afficher à tour de rôle les différentes sous expressions, en changeant l'affichage à chaque clic de l'utilisateur.

- **`hightlight`** : l'expression est surlignée lorsqu'on la sélectionne avec la souris,

- **`menu`** : fournit un pop-up menu des sous expressions.

les balises de Contenu

Cette classe regroupe une centaine d'éléments acceptant à peu prés une douzaine d'attributs. Les éléments de contenu sont soumis aux mêmes règles d'ordre que les éléments de présentation.

Ces éléments, qui correspondent à une large gamme d'opérateurs, de relations et de fonctions, véhicule essentiellement de la sémantique, des entités mathématiques au sens propre, censées être dégagé de toute notion de notation. Le but de ces éléments est de structurer une information afin de la retransmettre à des interpréteurs algébriques. Ainsi, l'ensemble des entités mathématiques étant conséquent, il existe un grand nombre de balises de contenu. De plus, il existe peu d'attributs contre un grand nombre d'élément, le W3C Math Working Group préférant lever le plus d'ambiguïté possible en introduisant de nouveaux éléments plutôt que de nouveaux attributs.

Un bon nombre de fonctions et d'opérations nécessite des « quantificateurs » pour être bien défini. Par exemple, une intégrale doit spécifier les bornes d'intégration ainsi que la variable sur laquelle nous intégrons. Il y a donc un certains nombre de « **qualifier schemata** » comme `bvar`, ou `lowlimit`, utilisés avec des opérateurs comme `diff` (différentiel) ou `int` (intégrale).

Présentons à présent deux des éléments les plus importants des balises de contenu : les balises `apply` et `declare`.

La balise declare

L'élément `declare` est particulièrement important pour les éléments de contenu qui peuvent être évalués par des systèmes algébriques. Cet élément fournit un simple mécanisme d'assignation où une variable peut être déclarée comme étant d'un certain type, avec une certaine valeur.

La balise apply

La balise `apply` est utilisée pour appliquer des opérations à des expressions ainsi que pour créer de nouveaux objets mathématiques à partir d'objets existants. Elle permet par exemple d'appliquer une fonction à un ensemble d'arguments. Une fois de plus, l'ordre des arguments à une importance capitale : le premier fils correspond à la fonction à appliquer, les fils restants correspondent aux arguments de la fonction. En ce qui concerne les opérations algébriques, il faut respecter, comme en LISP, la notation polonaise inversée ; l'opération *a – b* s'écrit :

a – b

```
<apply>
   <minus/>
   <ci> a </ci>
   <ci> b </ci>
</apply>
```

Reprenons l'exemple $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$:

balises de Contenu :

```
<apply>
   <eq/>
   <ci>x</ci>
   <apply>
      <divide/>
      <apply>
         <fn><mo><mchar name='PlusMinus'/></mo></fn>
         <apply>
            <minus/>
            <ci>b</ci>
         </apply>
         <apply>
            <root/>
            <apply>
               <minus/>
               <apply>
                  <power/>
                  <ci>b</ci>
                  <cn>2</cn>
               </apply>
               <apply>
                  <times/>
                  <cn>4</cn>
                  <ci>a</ci>
                  <ci>c</ci>
               </apply>
            </apply>
            <cn>2</cn>
         </apply>
      </apply>
      <apply>
         <times/>
         <cn>2</cn>
         <ci>a</ci>
      </apply>
   </apply>
</apply>
```

Comme vous pouvez le remarquer, ce code est beaucoup plus long que celui utilisé pour la notation, et la balise `apply` joue un rôle primordial.

Notons ici que les éléments de contenu et de présentation peuvent être mélangés. La seule règle à respecter dans ce cas, est que le contenu « panaché » n'est autorisé à apparaître que s'il veut vraiment dire quelque chose. Par exemple, pour l'équation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, le caractère "±" n'existe pas en balise de contenu. On utilise donc la balise `fn` pour déclarer que la balise de cet opérateur agit en tant qu'élément de contenu.

Une autre option est l'utilisation de la balise `semantics`. Cet élément est utilisé pour lier des expressions MathML avec d'autres types de notation. Une utilisation courante de la balise semantics est de joindre des éléments de contenu à des éléments de présentation, en tant qu'annotation sémantique. Ainsi, l'auteur peut spécifier qu'une notation non-standard est utilisée lors du rendu d'une expression utilisant seulement des éléments de contenu. Prenons l'exemple de l'intégrale de 0 à t de 1/x, codée en balises de contenu. Le rendu par défaut serait sans doute $\int_0^t (1/x)dx$. Nous allons donc fournir une expression de présentation qui servira d'annotation sémantique afin d'obtenir l'affichage $\int_0^t \frac{dx}{x}$

Il faut cependant savoir que l'utilisation présentée ci-dessus des annotations ne fait pas partie des spécifications MathML. L'utilisation qui en sera faite sera donc liée au moteur de rendu choisie.

L'interface MathML

Comme nous l'avons déjà vu, pour être efficace, MathML doit être associé à plusieurs moteurs de rendu, à des éditeurs, des processeurs, des traducteurs, etc. Et puisque que MathML a pour objectif premier de transmettre de l'information mathématique sur le Web, l'association la plus importante est sans doute celle de MathML avec HTML et XHTML.

Il y a trois problèmes principaux rencontrés lorsque l'on veut interfacer du MathML avec du XML :

- Les balises MathML doivent être reconnues comme un objet XML valide, et non être perçues comme une erreur par l'analyseur. Ce problème est avant tout un problème de gestion des espaces de noms, les fameux *Namespaces*.

- Dans le cas d'intégration dans du HTML ou du XHTML, le rendu des équations MathML doit être assuré par les navigateurs. Actuellement, un certain nombre de browsers gèrent nativement le rendu de MathML (Mozilla par exemple). D'autres ont développé des outils permettant de mettre à jour leurs produits afin qu'ils puissent réagir face à MathML.

- Les outils générant et traitant du code MathML doit être capable de réagir entre eux. En effet, bien qu'il existe déjà un grand nombre de produits concernés par MathML, les expressions MathML ont tendance à croître en taille assez rapidement, ce qui augmente tout autant la probabilité de commettre des erreurs. Par conséquent, ces outils doivent fournir des mécanismes permettant de construire rapidement des fichiers MathML, et d'en vérifier régulièrement la validité.

Nous n'allons parler que des problèmes que nous pouvons à peu près contrôler, à savoir l'introduction de MathML dans d'autres documents XML.

MathML et les Namespaces

La gestion de MathML, dans un document XML en général et dans un document XHTML en particulier, passe en grande partie par les namespaces.

Un « espace de nom » XML est une collection de noms réservés par une URI que l'on spécifie. L'URI pour les namespaces MathML est :

http://www.w3c.org/1998/Math/MathML

Pour déclarer des espaces de noms, nous pouvons soit utiliser l'attribut `xmlns`, soit un attribut avec `xmlns` comme préfixe. Lorsque l'on utilise `xmlns` seul, on fixe une bonne fois pour toutes le namespace associé à cet élément et à l'ensemble de ses fils, nous n'avons plus besoin d'y retoucher par la suite.

Namespaces sans préfixe

```
<math xmlns= "http://www.w3c.org/1998/Math/MathML" >
   <mrow> … </mrow>
</math>
```

Lorsque l'on préfère utiliser `xmlns` avec un préfixe, ce préfixe peut être utilisé pour associer explicitement des éléments au namespace déclaré n'importe tout dans la suite du document :

Namespaces avec préfixe

```
<body xmlns:m= "http://www.w3c.org/1998/Math/MathML" >
…
   <m:math><m:mrow> … </m:mrow></m:math>
…
</body>
```

Naturellement, ce type de déclaration ne modifie par la grammaire de MathML, c'est à dire que toute insertion d'équation MathML doit commencer par l'élément `math`.

Ces deux modes de déclaration de namespaces peuvent être utilisés conjointement. Par exemple, nous pouvons utiliser une déclaration explicite via un préfixe en début de document, puis utiliser la déclaration par défaut lorsque nous nous trouvons dans les éléments mathématiques :

Namespaces avec et sans préfixe

```
<body xmlns:m= "http://www.w3c.org/1998/Math/MathML" >
…
   <m:math xmlns= "http://www.w3c.org/1998/Math/MathML" >
      <mrow> … </mrow>
   </m:math>
…
</body>
```

Le problème à l'heure actuelle est que les DTD ne supportent pas encore très bien les namespaces. La plupart du temps, pour s'en sortir, il faut définir explicitement les namespaces dans la DTD (de XHTML par exemple). L'inconvénient de cette méthode est qu'elle n'est absolument pas modulable et maintenable, toute évolution de la DTD MathML devant induire une modification des espaces de noms dans la DTD XHTML.

De plus, les modes d'utilisation des namespaces présentés ci-dessus (avec ou sans préfixe), bien que faisant partie de la recommandation du W3C, ne sont jamais appliqués à la lettre par les interpréteurs XML, quels qu'ils soient. Il convient donc, pratiquement, de respecter les règles suivantes lors d'une déclaration d'espaces de noms :

- Utiliser le préfixe `m:` pour introduire des balises MathML, il s'agit souvent du seul préfixe reconnu par les interpréteurs. L'usage des préfixes explicites est généralement plus sur.

- Si vous choisissez les préfixes, n'en choisissez qu'un et utilisez le constamment dans votre document,

- Déclarer explicitement les namespaces MathML dans toutes les balises math.

Nous avons donc les exemples suivants :

Exemple privilégié des namespaces

```
<body>
…
   <m:math xmlns:m= "http://www.w3c.org/1998/Math/MathML" >
      <m:mrow> … </ m:mrow>
   </m:math>
…
</body>
```

ou

```
<body>
…
   <m:math xmlns = "http://www.w3c.org/1998/Math/MathML" >
      <mrow> … </ mrow>
   </math>
…
</body>
```

l'élément racine math

MathML ne définit qu'un seul élément racine pour toute ses équations : l'élément `math`, qui encapsule toutes les instances de MathML dans tout type de documents. L'élément `math` est toujours l'élément le plus haut d'une expression MathML, et un élément `math` ne peut en contenir un autre.

En conséquence, tout interpréteur ou processeur qui retourne une expression MathML (lors d'une interprétation algébrique ou simplement lors d'un copier-coller) doit l'encapsuler dans une balise math. Et logiquement, on peut tester la présence d'équations mathématiques dans un document en vérifiant simplement la présence de l'élément `math`.

L'élément `math` peut contenir un nombre quelconque de fils. Par défaut, le rendu du contenu de cet élément est le même que celui d'une balise `mrow`. Les attributs de cet élément, compatible avec DOM et les feuilles de styles (CSS ou XSL) sont :

- **macros** : permet de pointer sur des macros externes. Les macros ne font pas parties des spécifications MathML. Cependant, cet attribut est là en prévision d'un développement futur des macros sur MathML. La valeur de cet attribut est une séquence d'URL ou d'URI, séparés par des espaces.

- **mode** : la valeur par défaut est `inline`, mais les valeurs possibles sont `display` et `inline`. Il définit le type de rendu choisi.

- **display** : cet élément, compatible avec CSS, est censé remplacer l'élément `mode`. Les valeurs possibles sont `block` et `inline`.

Ces attributs sont très importants, puisqu'ils s'appliquent finalement à l'ensemble de l'expression contenue dans l'élément `math`. Cependant, afin de rendre proprement ces expressions dans un navigateur, et de les intégrer correctement dans les documents XHTML, un autre ensemble d'attributs assez utile est défini :

- **overflow** : si l'on ne peut pas jouer sur la taille de l'expression ou si cette taille se heurte aux préférences de l'utilisateur (par exemple, si l'expression est trop longue), cet attribut est censé proposer une alternative au moteur de rendu. Les valeurs possibles sont :
    - **Scroll** : on peut ajouter une barre de déroulement verticale ou horizontale,
    - **elide** : on supprime une partie des expressions de sorte qu'elles rentrent dans la fenêtre. Par exemple, on supprime une partie d'un long polynôme et on le remplace par + … +. Les futurs moteurs de rendu devraient savoir gérer le zoom sur des parties manquantes,
    - **truncate** : l'expression est simplement tronquée à droite et en bas de la fenêtre,
- **scale** : on remet les fonts à l'échelle de la fenêtre de telle sorte que l'expression complète rentre dans la fenêtre.
- **alting** : cet attribut, dont la valeur est une URL, fournit une issue de secours pour les browsers qui ne supportent pas les expressions MathML,
- **alttext** : cet attribut, dont la valeur est une chaîne de caractères, fournit une issue de secours pour les browsers qui ne supportent pas les expressions MathML,

La DOM MathML

MathML étant une application XML non modifiable(on ne peut pas définir de nouveaux éléments), une DOM plus spécifique peut lui être associée. Nous allons donc définir quelles sont les grandes règles de cette DOM particulière, ce qui nous permettra par la suite de présenter parallèlement les éléments MathML important et les modèles objets qui leur sont associés.

Cette DOM MathML spécifique a pour but :

- De spécialiser des fonctionnalités et d'en ajouter d'autres qui sont spécifiques aux éléments MathML,
- De fournir des mécanismes pratiques permettant d'accéder plus facilement et rapidement aux opérations MathML fréquentes. (accès à des attributs, des arguments, etc.)

Les principales spécifications de cette DOM, afin de réaliser ces objectifs et de rester conforme aux spécifications DOM générales, sont les suivantes :

- Définir une interface `MathMLElement`, dérivée de l'interface `Element`, qui spécifie l'ensemble des opérations communes à tous les éléments MathML. Cela concerne notamment l'accès aux attributs communs à tout les éléments (`class`, `id`, …).
- Définir des interfaces héritant de `MathMLElement` afin de représenter les éléments MathML possédant, en plus des attributs communs à tous, des attributs spécifiques. Dans, ce cas, la DOM doit fournir des méthodes explicites pour accéder à ces attributs, ou pour modifier leur valeurs.

- Définir des méthodes spéciales dans chacune de ces interfaces afin d'insérer ou de récupérer un ou plusieurs nœuds fils. Ces méthodes sont issues des méthodes classiques définies dans les interfaces `Node` et `Element`, elles fournissent simplement un mécanisme plus pratique pour accéder aux enfants, de manière à lever toute ambiguïté, et à simplifier le travail des programmeurs. Par exemple, l'interface `MathMLFractionElement` permet d'accéder directement au numérateur par l'appel de la fonction `getDenominator()`, fonction beaucoup explicite que `getChildNodes().item(2)`.

Lorsqu'une interface ne fournit pas de méthodes spécifiques pour modifier ou récupérer un attribut ou un argument, ce seront bien sûr les méthodes DOM génériques qui devront être utilisées.

MathML définit des règles spécifiques, qui sont transparentes pour les processeurs et parseurs XML classiques. Le fait que les objets de la DOM MathML soit requis pour vérifier le respecter de ces règles, ainsi que pour fournir l'exception adéquate lorsque ces règles sont violées, est une raison suffisante pour utiliser cette DOM spécifique). La DOM MathML est vraiment indispensable pour contrôler les règles d'ordonnancement des balises, et en l'absence des Schemas, elle sera bien la seule à pouvoir contrôler le typage de la valeur des attributs.

Le Parseur T_EX

« Le présent serait plein de tous les avenirs, si le passé n'y projetait déjà une histoire », André Gide

T_EX

C'est quoi donc ?

Le T_EX est un puissant langage de mise en forme de documents, écrit par Donald E. Knuth en 1978. Conçu à l'origine pour modéliser des équations mathématiques complexes, on le connaît aujourd'hui dans sa version 3.14.

L_AT_EX est en fait une version spéciale de T_EX: il s'agit d'une collection de macro-commandes T_EX et de catégories de documents (article, rapport, livre, etc…)

L_AT_EX est un système performant de préparations de documents, et ne se classe pas dans la catégorie des wysiwyg (what you see is what you get) comme Microsoft Word, WordPerfect ou encore StarOffice. L_AT_EX décrit un document et le résultat n'est visualisable qu'après compilation. Il n'est de ce fait pas soumis aux impératifs de temps réel qu'offrent les wysiwyg. Sa diffusion se fait dans le domaine public, il est donc en perpétuelle évolution et de nombreux utilisateurs écrivent des extensions qui seront ensuite réutilisables par tous.

Des concepts communs aux autres langages

En ce qui concerne la structure des documents, L_AT_EX possède un certain nombre de concepts communs avec des langages comme le XML, le HTML, etc.… Un fichier L_AT_EX est visualisable dans n'importe quel éditeur capable de lire du texte normal. Son utilisation se fait à partir du principe de balise délimitant les différents types de données et fonctionnant par couple début-fin.

On en arrive ainsi au concept fondamental de L_AT_EX qui veut que l'on s'intéresse plus au fond du document qu'à son apparence. Le choix de la mise en forme se fait par l'intermédiaire d'un fichier de classe de documents auquel peuvent être adjoints des fichiers d'extension. Ces derniers permettent de modifier la mise en forme très simplement : il suffit de changer de fichier de classe et de recompiler.

Le fichier T_EX se compile en utilisant le programme L_AT_EX. Il peut être ensuite converti, par exemple en Postscript.

Utilisé dans de nombreux domaines

Comme nous l'avons dit plus haut, L_AT_EX est depuis bien longtemps employé par les scientifiques du fait de sa capacité à mettre en forme les équations mathématiques complexes. Mais aujourd'hui, L_AT_EX est présent dans

bien d'autres domaines : partitions musicales, formules chimiques, circuits électriques, graphismes complexes, tableaux, langues orientales, etc.…

Un peu de pratique

Certains symboles comme %,#,$,&,{,},\,… sont interprétés comme des instructions ; pour les faire apparaître à l'écran, il faut les faire précéder du symbole \, qui est le plus important, puisqu'il marque le début d'une commande.

Un programme source L_AT_EX minimal se présente sous la forme suivante :

Exemple T_EX

```
\documentclass{article}
\begin{document}
% ceci est un commentaire
% il représente le corps du document
\end{document}
```

La première ligne renseigne sur la classe du document choisi, on aurait pu mettre à la place « report », « book » ou « letter ».

En ce qui concerne le mode mathématique, il peut être enclenché de plusieurs manières :

1. le mode mathématique en ligne : entrée par $ ou \(ou \begin{math} et sortie par $ ou \) ou \end{math} ;
2. le mode mathématique de mise en évidence : entrée par $$ ou \[ou \begin{displaymath} et sortie par $$ ou \] ou \end{displaymath}.

Le premier, comme son nom l'indique, permet d'inclure des éléments mathématiques dans une phrase :

Mode 1 :

```
%fichier exemple1.tex
\documentclass{article}
\begin{document}
Un demi ajout \'e \`a un tiers vaut $\frac{5}{6}$
\end{document}
```

La commande \frac donne une fraction avec, comme premier argument, le numérateur, et comme second, le dénominateur. Le symbole ^ sera utilisé pour les exposants et _ pour les indices.

Le second mode mathématique est beaucoup plus adapté aux présentations :

Mode 2 :

```
%fichier exemple2.tex
\documentclass{article}
\begin{document}
Une \'equation homog\'ene du second degr\'e :$$x_1^2+x_2^3+x_3^2=0
$$
\end{document}
```

Avantages et inconvénients

Les principaux avantages de L_AT_EX par rapport à un traitement de texte traditionnel sont :

- Mise en page professionnelle qui donne aux documents l'air de sortir de l'atelier d'un imprimeur.
- Composition des formules mathématiques de manière pratique.
- Il est seulement nécessaire de connaître quelques commandes de base pour décrire la structure logique du document. Il n'est pas nécessaire de se préoccuper de la mise en page.
- Des structures complexes telles que les notes de bas de page, des renvois, la table des matières ou les références bibliographiques sont produites facilement.
- Pour la plupart des tâches de la typographie qui ne sont pas directement gérées par L_AT_EX, il existe des extensions supplémentaires gratuites.
- L_AT_EX encourage les auteurs à écrire des documents bien structurés, car c'est ainsi qu'il fonctionne
- T_EX, l'outil de formatage de L_AT_EX, est réellement portable et gratuit. Ainsi il est disponible sur quasiment toutes les machines existantes.

L_AT_EX a également quelques inconvénients :

- Il consomme plus de ressources (mémoire, espace disque, puissance de calcul) qu'un traitement de texte normal.
- Bien que quelques paramètres des mises en page prédéfinies puissent être personnalisés, la mise au point d'une présentation entièrement nouvelle est difficile et demande beaucoup de temps.

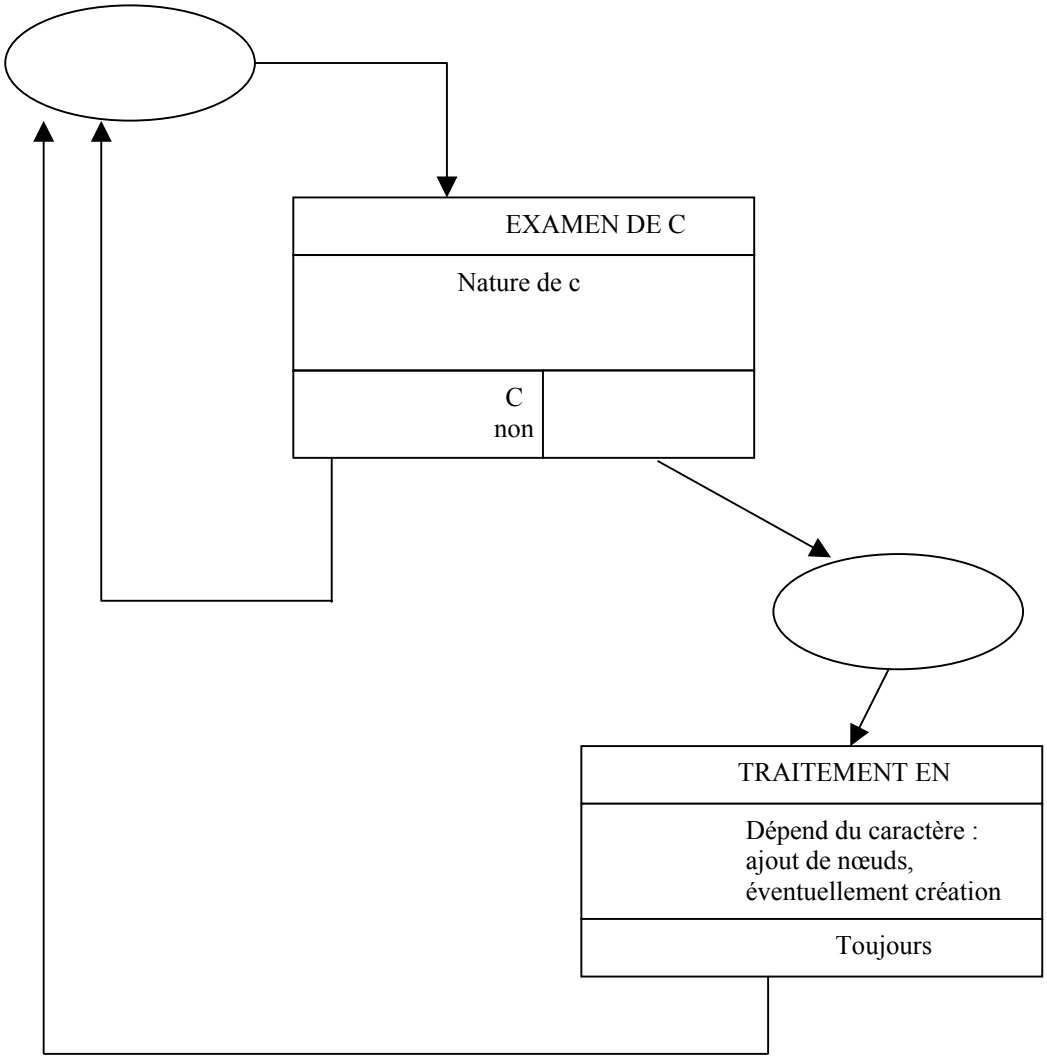
Lorsque nous mettons MathML et T_EX en parallèle, nous voyons de suite une différence fondamentale : MathML possède à la fois l'aspect mise en forme du texte et la signification réelle : c'est le sens et la forme. Au contraire, T_EX est uniquement destiné à gérer la mise en forme de document, et donc de formules mathématiques. Ce n'est pas un inconvénient, mais un élément de moins sur T_EX.

En conclusion, le point principal à retenir pour notre projet est que le langage T_EX permet de formuler simplement des équations mathématiques complexes. Nous allons maintenant voir comment traiter ces fichiers T_EX pour leur faire correspondre un arbre DOM le plus fidèle possible.

Conception (à partir de la version T_EX2_ε)

Définition : un parseur est une application permettant de lire un fichier et de faire des traitements en conséquence.

Etude des traitements



Le fichier T_EX est lu de manière séquentielle, caractère par caractère. Si le caractère lu est significatif (différent des caractères qui sont ignorés par le compilateur T_EX), un traitement spécifique est alors effectué : ajout d'un nœud, lecture jusqu'à un autre caractère « clé » défini.

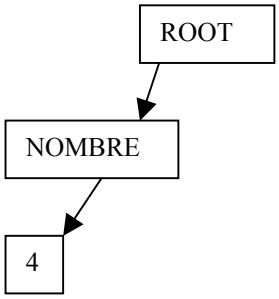
Ce schéma est valable pour un traitement général. Nous nous sommes cependant restreint au traitement des objets mathématiques du fait de l'étendue et de la richesse de T_EX.

Premier exemple

Voyons sur un exemple comment un fichier T_EX est parsé :

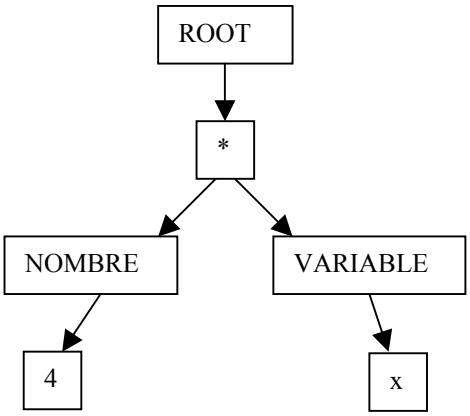
Soit un fichier contenant 4x+1.

La procédure de parsing commence : on lit d'abord le caractère « 4 » ; c'est un nombre, on crée alors un fils de root contenant l'entier « 4 » de la manière suivante :



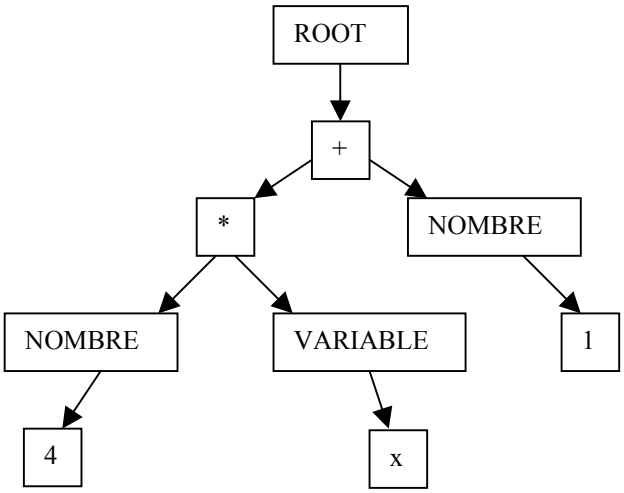
Le nœud courant est alors ROOT, le nœud NOMBRE ayant déjà son fils.

Puis on lit le caractère « x » : ce caractère se situe juste après un entier, on considère donc que c'est une variable. De plus, c'est une multiplication (implicite certes !) entre 4 et x : il faut prendre ceci en compte pour construire l'arbre suivant :

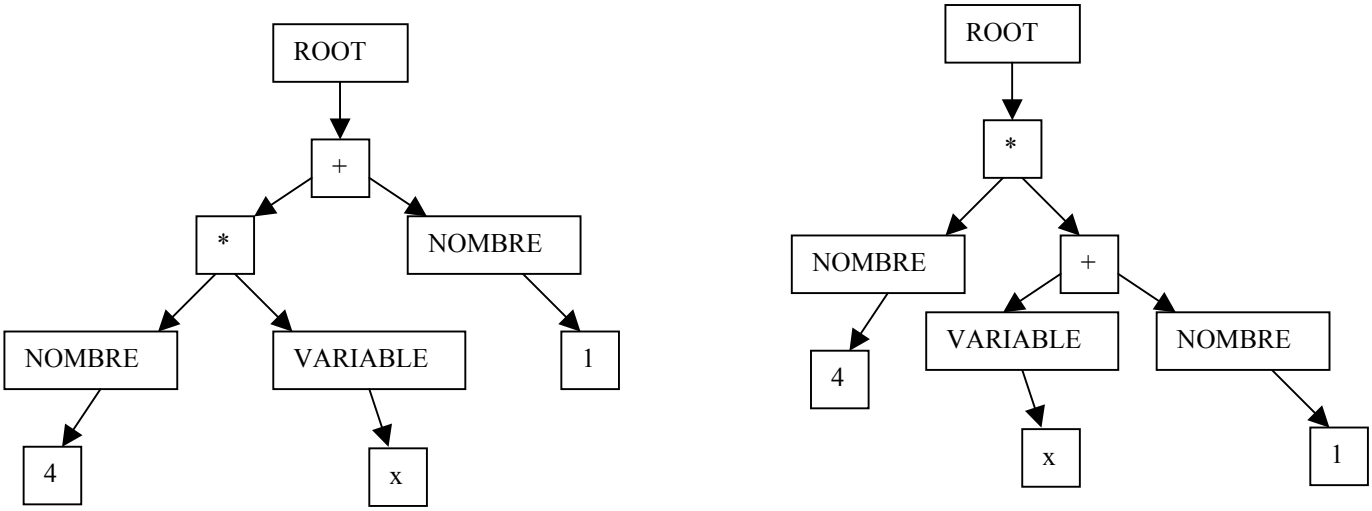


Le nœud courant reste « * ».

On continue le parsing du fichier, nous lisons successivement « + » et « 1 ». L'arbre devient alors :



Cet exemple simple nous donne néanmoins l'occasion de poser un problème qui est celui de la priorité des opérateurs : le « * » et le « / » ont bien sûr une priorité plus grande que celle de « + » et « - » . Il est nécessaire de prendre en compte cette notion de priorité pour éviter le problème suivant :



En effet, si l'on ne prend pas en compte la priorité des opérateurs, nous pouvons nous retrouver avec l'arbre de droite qui ne représente plus 4x+1 mais 4(x+1) !

L'algorithme est donc le suivant :

Algorithme :

```
REPETER
   SI l'on rencontre un opérateur lors de la lecture et que le nœud courant est un opérateur ALORS
      SI le nouvel opérateur a une priorité plus importante que celle du nœud courant ALORS
         SI on peut affecter un fils au nœud courant ALORS
            Alors le nouvel opérateur devient fils du nœud courant
         SINON
            Le dernier fils du nœud courant devient le fils du nouvel opérateur
            Le nouvel opérateur devient fils du nœud courant
         FINSI
      SINON
         le nœud courant désigne maintenant le père du nœud courant
      FINSI
   SINON
      Le dernier fils du nœud courant devient fils du nouvel opérateur.
      Le nouvel opérateur devient fils du nœud courant.
   FINSI
TANT QUE le nouvel opérateur n'est pas inséré dans l'arbre
```

En fait, ajouter cette notion de priorité des opérateurs revient à donner du sens à la formule mathématique ! Or, comme nous l'avons dit plus haut, la différence entre MathML et T_EX se situe dans la notion de sens présente dans MathML et pas dans T_EX. Si nous arrivions à insérer cette notion de sens lors du parsing du document T_EX, il y aurait un gain d'informations formidable !

Toutefois, donner un sens à l'ensemble des écritures mathématiques T_EX reviendrait à créer un interpréteur de l'écriture « naturelle » des mathématiques. En effet, comme nous l'avons précisé précédemment, T_EX ne permet que de décrire une écriture sans son sens. Par exemple, l'intégrale de la fonction sinus appliquée à x sur l'ensemble [0 ;1], s'écrie de manière naturelle par $\int_0^1 \sin(x)dx$. En T_EX, on dira qu'on écrie un signe « intégrale de 0 à 1 », suivi de la fonction sinus, elle-même suivie d'un « xdx ». Mais il ne sera précisé en aucune manière que l'intégrale s'applique sur le « sinx » et que le « dx » désigne la variable d'intégration. Par conséquent, pour donner un sens à l'arbre créé, il faudrait déceler le sens intuitif de l'ensemble des écritures mathématiques. Cela représente un travail trop important et nous nous sommes contentés de créer un arbre qui respecte la logique de présentation uniquement. Pour mieux comprendre ce problème, vous pouvez consulter le chapitre traitant des problèmes et améliorations.

Limitation d'interprétation

Le problème d'absence de sens dans l'écriture T_EX, nous a conduit à faire un choix dans l'interprétation des écritures. Par exemple « x(t+1) » peut signifier qu'on distribue la variable x sur la somme entre parenthèse ou que c'est la valeur de la composante en x d'une fonction paramétrique à l'instant « t+1 ». Fonction ou produit, c'est là que nous devons faire un choix.

Nous avons décidé de considérer qu'une variable suivi directement d'une parenthèse sera en fait une fonction appliquée à ce qui se trouve entre parenthèse. Pour obtenir le produit, on pourra rajouter un signe « ∗». Ce signe n'est par contre pas nécessaire lorsque c'est un nombre qui est devant la parenthèse comme dans « 4(x+1) ».

Représentation des objets mathématiques usuels :

Objets mathématiques	Représentation TEX	Représentation dans l'arbre
Nombre	4	`<NUMBER>4</NUMBER>`

Somme	4+3	`<PLUS>`
		`<NUMBER>4</NUMBER>`
		`<NUMBER>3</NUMBER>`
		`</PLUS>`

Multiplication	4*3	`<MULTIPLIED>`
		`<NUMBER>4</NUMBER>`
		`<NUMBER>3</NUMBER>`
		`</ MULTIPLIED >`

Soustraction	4-3	`<MINUS>`
		`<NUMBER>4</NUMBER>`
		`<NUMBER>3</NUMBER>`
		`</MINUS>`

Division	4/3	<DIVIDED> <NUMBER>4</NUMBER> <NUMBER>3</NUMBER> </DIVIDED>
Variables	x+4	<PLUS> <VAR>x</VAR> <NUMBER>4</NUMBER> </PLUS>
Les parenthèses	(x+1)	<EXPRESSION ENDMARKER=")" STARTMARKER="("> <PLUS> <VAR>x</VAR> <NUMBER>1</NUMBER> </PLUS> </ EXPRESSION >
Les accolades	{4}	< EXPRESSION ENDMARKER="}" STARTMARKER="{"> <NUMBER>4</NUMBER> </ EXPRESSION >
Fraction	\fract{4}{3} (équivaut à $\frac{4}{3}$)	<FRAC> <ARGUMENT ENDMARKER="}" STARTMARKER="{"> <NUMBER>4</NUMBER> </ARGUMENT> <ARGUMENT ENDMARKER="}" STARTMARKER="{"> <NUMBER>3</NUMBER> </ARGUMENT> </FRAC>
Egalité	x=4	<EQUAL> <VAR>x</VAR> <NUMBER>4</NUMBER> </EQUAL>
Puissance	x^2	<POWER> <VAR>x</VAR>

		<NUMBER>2</NUMBER> </POWER>
Le * « invisible »	4x+1	<PLUS> <INVISIBLETIMES> <NUMBER>4</NUMBER> <VAR>x</VAR> </ INVISIBLETIMES> <NUMBER>1</NUMBER> </PLUS>
Factoriel	5!	<FACTORIAL> <NUMBER>5</NUMBER> </FACTORIAL>
Somme	\sum_{i=0}^{N}(i+2) équivaut à $\sum_{i=0}^{N}(i+2)$	<SUM> <LOWERBOUND ENDMARKER="}" STARTMARKER="{"> <EQUAL> <VAR>i</VAR> <NUMBER>0</NUMBER> </EQUAL> </LOWERBOUND> <UPPERBOUND ENDMARKER="}" STARTMARKER="{"> <VAR>N</VAR> </UPPERBOUND> <ARGUMENT ENDMARKER=")" STARTMARKER="("> <PLUS> <VAR>i</VAR> <NUMBER>2</NUMBER> </PLUS> </ARGUMENT> </SUM>
Cos cosh	\sin {x} \cos{x}	<FUNCTION ENDMARKER="}" NAME="COS" STARTMARKER="{">

arccos sin sinh arcsin exp log logn	\tan {x} …	<VAR>x</VAR> </FUNCTION>

La notion d'opérateur

En examinant l'ensemble des opérateurs mathématiques, nous avons remarqué que certains d'entre eux possédaient des propriétés similaires et particulières. Ces opérateurs sont particulier pour plusieurs raisons :

- ils ont besoin de 2 arguments exactement ;
- ils ne peuvent s'enchaîner l'un après l'autres ;
- la notion de priorité leur est applicable.

C'est par exemple le cas pour un « + » ou un « / ». Ce n'est pas le cas pour les opérateurs intégrale ou somme par exemple qui se rapprochent assez de la notion d'argument ou plutôt de variable puisqu'ils peuvent être l'argument d'une addition, d'une autre somme, d'une fonction … Pour ceux-ci, la notion de priorité n'est pas appliquée puisque lorsqu'on construit un sous-arbre intégrale, il est hors de question de faire « remonter » un des nœuds en cours de création pour qu'il s'applique à l'intégrale.

De plus, en se basant uniquement sur la logique d'arbre, on peut ajouter dans l'ensemble « opérateur » tel que nous l'avons défini les virgules, qui servent à séparer deux expressions comme à l'intérieur d'une fonction à deux variables. Attention : il ne s'agit pas de la virgule séparatrice de partie décimale et entière. Pour cette fonctionnalité nous considérerons que c'est le point qui doit être utilisé.

Ainsi, nous aurons l'ensemble des opérateurs suivant : + − * / , = ^. Nous distinguerons aussi pour des raisons d'affichage les opérateurs multipliés sous-entendu dans les expressions comme « 4x ».

Algorithmes généraux des traitements mathématiques :

On distingue plusieurs « familles » de caractères, et ce découpage entraîne des traitements spécifiques selon la famille :

- lecture d'un opérateur : Comme nous l'avons dit précédemment, le placement de ce nouveau dépendra du niveau de priorité du nœud courant.
- lecture d'un chiffre : On lit tous les caractères suivants tant que l'on rencontre un chiffre (ou un « . », signifiant un nombre à partie décimale) le tout forment un nombre. Si le caractère qui suit est « ! », alors on rajoute un nœud FACTORIAL avec un fils : le nombre lu. Sinon ce nombre est directement ajouté en tant que fils du nœud courant
- lecture d'une lettre : Dans ce cas, on considère qu'on a affaire à une variable. Par conséquent, on lit tous les caractères de type lettre ou nombre qui suivent et ceux-

ci. Ils forment le nom de la variable (Xvar ou Y1 par exemple). Cette variable est ajoutée en tant que fils du nœud courant.

- lecture d'un antislash (« \ ») : On lit toutes les caractères de type lettre qui suivent. Ils forment un mot. Si ce mot est connu comme étant une fonction (cos, sin, exp par exemple) alors on crée un nœud fonction dont le nom est le mot lu. Sinon on crée un nœud dont le nom est le mot lu. Si le caractère qui suit est « _ », on ajoute un nœud limite inférieure. Ce caractère est forcément suivi d'une accolade ou d'un crochet ouvrant, dont le contenu sera parsé et ajouté comme fils du nœud « limite inférieure ». Puis, si le caractère qui suit est « ^ », on ajoute un nœud limite supérieure. Ce caractère est forcément suivi d'une accolade ou d'un crochet ouvrant, dont le contenu sera parsé et ajouté comme fils du nœud « limite supérieure ». Puis, on boucle avec en condition de sortie : le caractère suivant n'est pas (, [ou {, en ajoutant un sous-arbre dont la racine est un nœud argument et dont les fils sont le contenu parsé de l'accolade, crochet ou parenthèse.

- lecture d'un {, (ou [: On considère alors que nous avons affaire à une expression. On commence par regarder la nature du nœud courant :

Si c'est un opérateur *alors*
Si on peut lui ajouter des fils *alors*
On crée un noeud expression avec comme délimiteur ouvrant le caractère lu précédemment et ayant comme fils le contenu parsé de l'expression.
Ce nœud est ajouté au nœud courant

Sinon

On récupère le dernier fils du nœud opérateur
Si c'est une variable *alors*
on le transforme en nœud fonction qui aura comme argument le contenu parsé de l'expression.

Sinon
On crée un nœud opérateur « multiplié invisible » qu'on substitue au fils récupéré, qui lui, devient fils de ce nouveau nœud
Ensuite, on parse le contenu de l'expression et on l'ajoute en tant que fils de ce nouveau nœud.

Finsi

Finsi

Sinon
Si le nœud courant a un fils *alors*
On récupère ce fils
Si c'est une variable *alors*
on le transforme en nœud fonction qui aura comme argument le contenu parsé de l'expression.

Sinon
On crée un nœud opérateur « multiplié invisible » qu'on substitue au fils récupéré, qui lui, devient fils de ce nouveau nœud
Ensuite, on parse le contenu de l'expression et on l'ajoute en tant que fils de ce nouveau nœud.

Finsi

Sinon
On crée un noeud expression avec comme délimiteur ouvrant le caractère lu précédemment et ayant comme fils le contenu parsé de l'expression.
Ce nœud est ajouté au nœud courant

Finsi

Finsi

Remarques :

1. Lorsque dans l'algorithme, il est précisé que l'on parse une expression, c'est évidemment un parsing des objets mathématiques qui doit être effectué.

2. Les caractères espace, tabulation et retour à la ligne sont ignorés lors du traitement comme il est précisé dans les spécifications T_EX. Toutefois ils servent à marquer la fin d'une commande.

3. Si on rencontre durant le parsing un], un) ou une }, on ajoute ce caractère comme délimiteur fermant du nœud père et le parsing d'objets mathématiques est arrêté.

Développement

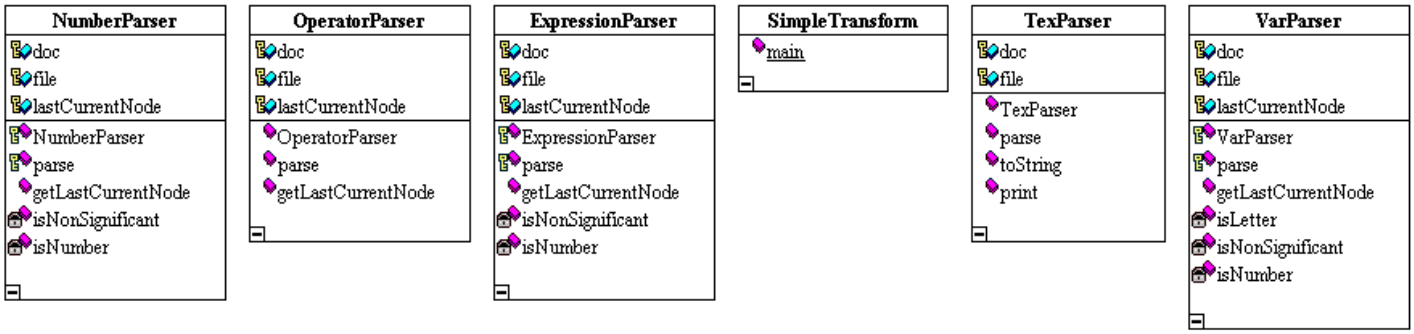
Présentation de Xerces

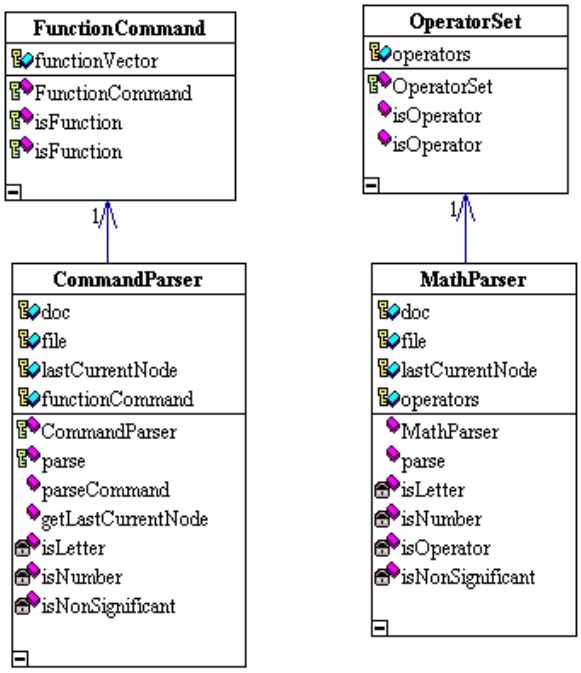
Xerces 1.3.1 est présenté comme étant un parseur Java. En fait, ce n'est pas en lui-même un parseur, mais un ensemble homogène constitué de plusieurs couches servant dans le processus de parsing :il supporte la recommandation XML 1.0 faite par le W3C, il contient les deux API SAX et DOM (les interfaces les plus communes pour la programmation XML) ainsi que les documentations API correspondantes, le tout pouvant alors parser des documents XML en respectant les spécifications et recommandations du W3C.

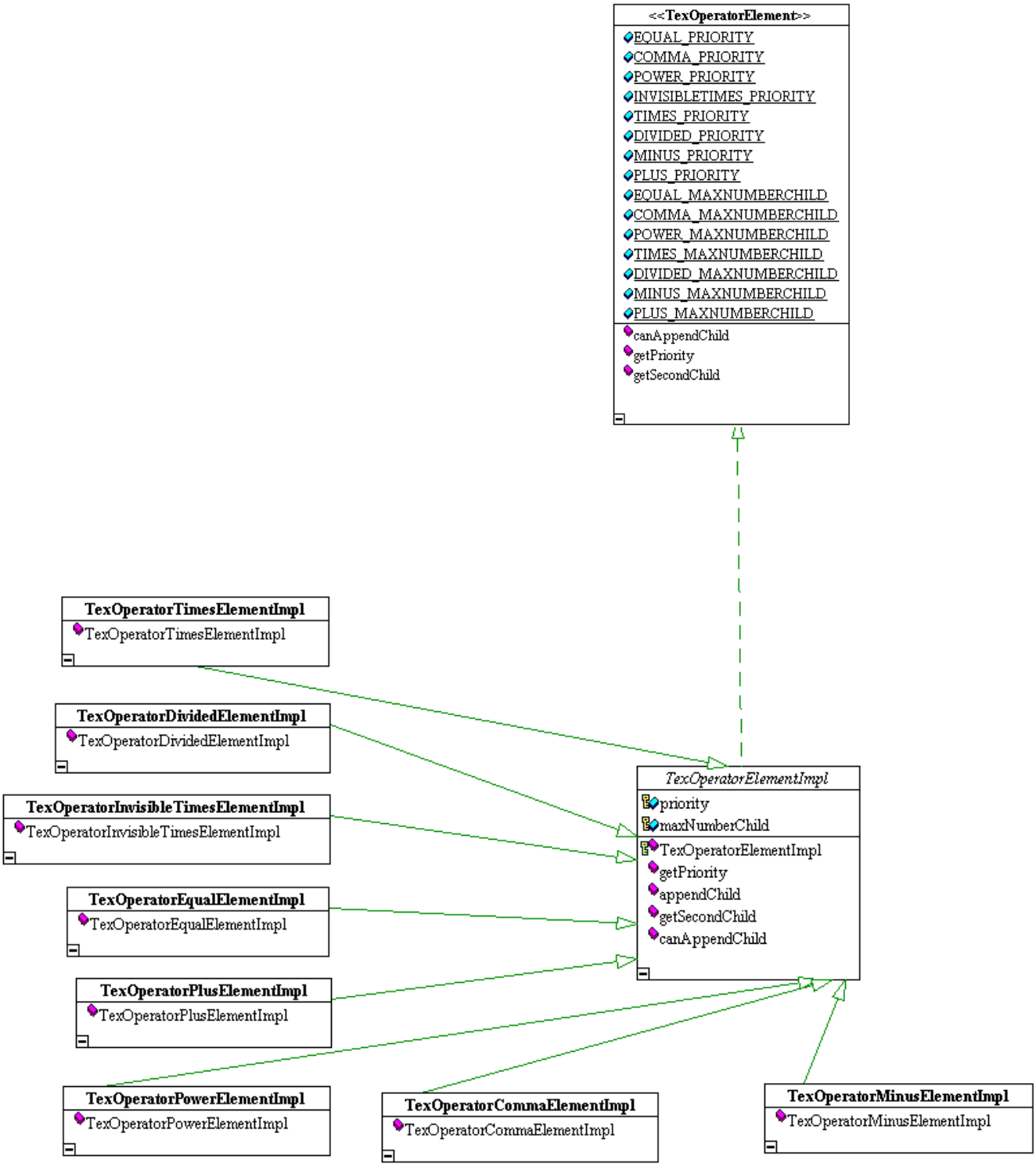
Modélisation objet basée sur les spécifications DOM de Xerces

Afin d'avoir un ensemble homogène et respectant le plus les normes définies par le W3C, nous nous sommes inspirés de la modélisation objet de Xerces pour celle de notre parseur T_EX. En fait, seul les objets qui sont des opérateurs tels que nous les avons définis sont définis dans des classes. Il n'est pas utile de créer un objet pour chaque fonction ou commande de T_EX.

Diagramme UML :






<<TexOperatorElement>>
EQUAL_PRIORITY
COMMA_PRIORITY
POWER_PRIORITY
INVISIBLETIMES_PRIORITY
TIMES_PRIORITY
DIVIDED_PRIORITY
MINUS_PRIORITY
PLUS_PRIORITY
EQUAL_MAXNUMBERCHILD
COMMA_MAXNUMBERCHILD
POWER_MAXNUMBERCHILD
TIMES_MAXNUMBERCHILD
DIVIDED_MAXNUMBERCHILD
MINUS_MAXNUMBERCHILD
PLUS_MAXNUMBERCHILD
canAppendChild
getPriority
getSecondChild
TexOperatorTimesElementImpl
TexOperatorTimesElementImpl
TexOperatorDividedElementImpl
TexOperatorDividedElementImpl
TexOperatorInvisibleTimesElementImpl
TexOperatorInvisibleTimesElementImpl
TexOperatorEqualElementImpl
TexOperatorEqualElementImpl
TexOperatorPlusElementImpl
TexOperatorPlusElementImpl
TexOperatorPowerElementImpl
TexOperatorPowerElementImpl
TexOperatorCommaElementImpl
TexOperatorCommaElementImpl
TexOperatorElementImpl
priority
maxNumberChild
TexOperatorElementImpl
getPriority
appendChild
getSecondChild
canAppendChild
TexOperatorMinusElementImpl
TexOperatorMinusElementImpl

Description des objets

Les opérateurs

- TexOperatorElement.java : Cette interface sert à représenter les opérateurs tels qu'ils ont été définis précédemment. Elle permet de définir l'ensemble des constantes représentant leur priorité et leur nombre maximal de fils. C'est une extension de l'interface Element définie dans Xerces. Quelques méthodes propres aux opérateurs ont été ajoutée afin de compléter l'interface Element pour une utilisation plus spécifique.

- TexOperatorElementImpl.java : Cette classe abstraite implémente l'interface vue ci-dessus. Elle permet de créer simplement des opérateurs particuliers, sans avoir à redéfinir l'ensemble des méthodes et attributs communs aux opérateurs. A priori, pour l'ensemble des opérateurs, seul le constructeur sera à faire.

- TexOperatorCommaElementImpl.java : Cette classe permet de créer des nœuds représentant des opérateurs « virgule ».

- TexOperatorDividedElementImpl.java : Cette méthode permet de représenter les opérateurs « divisé par ».

- TexOperatorEqualElementImpl.java : Cette méthode permet de représenter les opérateurs « égal à ».

- TexOperatorInvisibleTimesElementImpl.java : Cette méthode permet de représenter les opérateurs « multiplié par » sous-entendu, comme dans l'expression 4x par exemple.

- TexOperatorMinusElementImpl.java : Cette méthode permet de représenter les opérateurs « moins ».

- TexOperatorPlusElementImpl.java : Cette méthode permet de représenter les opérateurs « plus ».

- TexOperatorPowerElementImpl.java : Cette méthode permet de représenter les opérateurs « puissance ».

- TexOperatorTimesElementImpl.java : Cette méthode permet de représenter les opérateurs « multiplié par ».

- TexParser.java : Cet objet permet de parser un document T_EX en créant un arbre DOM.

- MathParser.java : Cet objet permet de parser les mathématiques sous T_EX. Il se servira des autres objets « parser » définis ci-dessous en fonction du caractère rencontré, comme cela a été défini plus haut.

- CommandParser.java : Cet objet permet de parser les commandes, c'est-à-dire l'ensemble des éléments débutant par un antislash.

- FunctionCommand.java : Cet objet fournit la liste de toutes les commandes qui ne sont en fait que des fonctions. Ce sont par exemple les commandes \sin ou \log. Cette liste est lu à partir d'un InputStream.

- ExpressionParser.java : Cet objet permet de parser une expression.

- NumberParser.java : Cet objet permet de parser les nombres.
- OperatorParser.java : Cet objet permet de parser l'ensemble des opérateurs tels que nous l'avons défini.
- OperatorSet.java : Cet objet fourni la liste des opérateurs traité par l'OperatorParser. Elle permet aux autres objets de repérer les nouveaux opérateurs implémentés sans avoir à les modifier.
- VarParser.java : Cet objet sert à parser les variables.
- SimpleTransform.java : Cet objet contient le main. Il sert d'illustration de l'utilisation d'un TexParser.

PROBLEMES ET AMELIORATIONS

La principale différence entre MathML et T_EX réside dans le fait que MathML considère à la fois la forme et le sens mathématique alors que T_EX prend en compte seulement la forme.

Cette différence montre bien que lorsque nous parserons un document T_EX, il y aura une sorte de « perte d'informations » du fait que l'on n'aura pas la notion de sens. C'est pourquoi nous avons voulu introduire à l'intérieur même du parseur cette notion afin que l'arbre DOM généré puisse transmettre cela via les feuilles XSLT.

Toutefois, ceci nous a obligé à répertorier tous les cas particuliers des expressions traitées afin de pouvoir les prendre en compte. Et il est clair que certains cas sont difficilement traitables et demandent de faire un choix de mise en forme en altérant la signification mathématique. C'est pourquoi, après plusieurs tentatives de modélisation, nous avons décidé de ne pas inclure la notion de sens lorsque cela s'avérait trop coûteux en terme de temps notamment.

Il est vrai que nous avons instauré la notion de priorité dans les opérateurs afin de respecter les caractéristiques de la multiplication face à l'addition par exemple.

Par ailleurs, quelques expressions restent très délicates à mettre en place, à cause des multiples cas particuliers que l'on peut rencontrer ; prenons le cas de l'intégrale : soit l'exemple suivant

$$\int_3^8 f(x)\mathrm{d}x = 6$$

Cette intégrale est codée en T_EX de la manière suivante :

```
\int_3^8 f(x)\ud x=6
```

Seulement, il se peut que l'on ne donne pas les bornes de l'intégrale, mais juste le domaine Φ d'application :

$$\int_{\Phi} f(x)\mathrm{d}x = 6$$

Donc cela change l'écriture de cette expression en T_EX, et surtout cela affecte le sens : en effet, le domaine Φ n'est en aucun cas une borne inférieure !

De plus, il se peut que l'on simplifie encore plus l'expression :

$$\int_{\Phi} f + \alpha = 10$$

A ce moment-là, on aura des difficultés à savoir quelle est la juste expression à insérer dans l'intégrale ; en effet, les deux expressions suivantes sont radicalement différentes :

$$\int_{\Phi} (f + \alpha) = 10 \quad \neq \quad \int_{\Phi} f + \alpha = 10$$

Mais à priori le code T_EX ne donne pas forcément les « ( ) ». Ainsi le code T_EX suivant :

```
\int_{0}^{10} g(x) + f(x)
```

est parfaitement valable. Il représente l'intégrale suivante :

$$\int_{0}^{10} f(x) + g(x)$$

Mais comment savoir si la fonction g(x) est à l'intérieur où à l'extérieur de l'intégrale ? A priori, nous ne pouvons pas déterminer ce caractère. Cela montre bien que nous pouvons afficher sous T_EX n'importe quelle formule mathématique, même si elle n'a pas de sens.

Dans un souci de donner un sens mathématique, nous avons cherché un moyen de délimiter l'expression concernée par l'intégrale. Mais, étant donné que l'insertion de la notion de sens est un luxe que l'on rajoute, nous avons décidé de ne pas en mettre dans ce cas-là.

Ce problème est aussi présent pour la balise \sum (somme), nous avons également inséré des <INVISIBLETIMES> lorsque les « ( ) » sont absents. Par exemple, $2\sum_{i=0}^{N} i$ est codé ainsi :

Codage avec INVISIBLETIMES :

```
<INVISIBLETIMES>
  <NUMBER>2</NUMBER>
  <SUM>
    <LOWERBOUND ENDMARKER="}" STARTMARKER="{">
      <EQUAL>
        <VAR>i</VAR>
        <NUMBER>0</NUMBER>
      </EQUAL>
    </LOWERBOUND>
    <UPPERBOUND ENDMARKER="}" STARTMARKER="{">
      <VAR>N</VAR>
    </UPPERBOUND>
    <ARGUMENT ENDMARKER=")" STARTMARKER="(">
      <VAR>i</VAR>
    </ARGUMENT>
  </SUM>
</INVISIBLETIMES>
```

Au niveau des améliorations possibles, il est clair que même si nous nous sommes limités rapidement aux expressions mathématiques, nous n'avons pas pu modéliser l'ensemble de ce domaine à cause de sa richesse (nombreux cas particuliers) et de la complexité de certains composants.

Cependant, nous avons modélisé un mode de traitement générique pour les commandes mathématiques codées par un « \ ». Il ne reste qu'à enrichir le traitement des cas particuliers !

En outre, il est clair qu'il faudrait dans un second temps se pencher sur le domaine de mise en forme de document de T_EX, domaine aussi vaste, voire plus grand car c'est la raison d'être de T_EX.

Pour ce qui est de la modélisation de ce domaine, il n'est pas du tout évident qu'il soit à mettre en rapport avec la modélisation du domaine mathématique.

Les transformations par XSL et XSLT

« Rien ne se perd, rien ne se crée, tout se transforme », Lavoisier

Introduction

De plus en plus de logiciels et de services sont basés sur les technologies XML et XSL (et maintenant XSLT).

Dans la première partie de ce chapitre, nous allons tâcher de donner les raisons pour lesquelles les transformations de fichiers XML sont aussi importantes. Puis, nous décrirons les différents composants et outils utiles voire nécessaires pour effectuer de telles transformations. Enfin, nous finirons par donner la syntaxe de base des transformations XSL, plus connues sous le nom de XSLT. Nous essayerons de montrer comment les processeurs Java XSLT utilisent des fichiers XML pour leur donner un aspect qui peut être totalement différent (transformation en un autre type de fichiers, i.e. passage d'un fichier XML à un fichier HTML ou texte par exemple) et comment la DOM, que nous avons déjà expliquée dans les chapitres précédents, est manipulée dans ce cas précis.

La spécification complète du langage XSL est énorme (quelques centaines de pages). Aussi, nous nous limiterons à montrer comment utiliser les transformations XSLT pour reformater un document XML. Notons également que la spécification XSL n'est pas encore figée, le dernier draft ayant été publié le 18 octobre 2000. Cependant, la norme concernant uniquement les transformations XSL que nous allons utiliser ici est stabilisée (la dernière modification datant de novembre 1999).

Il est donc évident que tous les aspects de XSL ne seront pas présentés dans ce chapitre. Nous vous indiquons par surcroît des références de choix:

- Le livre écrit par Robert Eckstein édité chez "O'Reilly & Associates" et intitulé "XML Pocket Reference".
- "Java and XML" de Brett McLaughlin chez O'Reilly dont cette présentation s'inspire partiellement (et qui est disponible à la bibliothèque de l'INSA!!).
- Le site Internet du W3C dédié aux transformations XML: http://www.w3.org/Style/XSL/.

Pourquoi une standardisation des transformations XML?

Tout d'abord, tout simplement par souci de présentations variées. Expliquons-nous un peu plus en détails sur ce sujet. Après avoir été formé, un fichier XML peut être nécessaire à plusieurs utilisateurs totalement différents. Un utilisateur peut être un être humain ou bien même un programme Java ou C++ par exemple. Suivant que

l'utilisateur est un être humain ou un programme, le fichier XML pourra contenir des informations différentes ou bien les mêmes informations mais présentées de manière différente : un être humain pourra s'intéresser à la partie purement informative du fichier alors que le programme nécessitera la lecture de certaines données totalement inutiles pour l'humain.

Enfin, il a été nécessaire de standardiser les processus de transformation pour pouvoir réutiliser les documents obtenus après leur changement. Un document A changé en un document B pourra ainsi être à nouveau transformé en un document C à son tour sans aucun problème.

Les outils nécessaires aux transformations XML

Aussi utiles que peuvent être les transformations XML, elles ne sont en général pas simples à réaliser. Trois recommandations distinctes du W3C ont vu le jour pour définir comment les transformations doivent s'effectuer: XSL, XSLT et XPath. Bien que l'une ait une utilisation plus poussée et qui va au-delà des feuilles de style (pour XPath), nous ne verrons ici que ce qui peut nous fournir les moyens de passer d'un format XML à un autre format.

En général, on ne fait pas une distinction claire et précise entre ces trois outils parce que leurs spécifications sont étroitement liées et qu'on ne les utilise toujours en même temps.

En fait, le terme XSLT, qui réfère spécifiquement aux transformations de forme, est souvent appliqué à XSL & XSLT. D'autre part, XSL regroupe souvent par abus de langage les trois termes réunis.

Ce chapitre va décrire ces trois recommandations séparément. Cependant, dans le reste du projet, nous regrouperons les termes XSL et XSLT pour parler des transformations XML dans un but de simplicité.

XSL

XSL, qui est en fait l'acronyme de "eXtensible Stylesheet Language", est un langage permettant de décrire des feuilles de style.

Il est subdivisé en deux parties:

- Un langage permettant d'appliquer des transformations à des documents XML.
- Un vocabulaire XML permettant de spécifier précisément la sémantique du formatage final.

Après lecture de ces deux points, d'aucun pourrait penser que ce sont deux définitions tout à fait identiques. Alors expliquons les plus précisément:

- La première définition indique que XSL prend en charge le passage d'une forme de document XML à une autre.
- La deuxième est basée sur la présentation du contenu au sein de n'importe quel document XML.

Pour comprendre comment XSL fonctionne, il est bon de savoir et de comprendre que toute l'information traitée lors d'une utilisation de XSL se fait sous forme d'arbre. Les "templates" sont utilisés pour récupérer l'élément de départ d'un document XML appelé élément racine. De même, les "leaf" sont appliquées aux éléments feuilles. Ainsi, on traite les documents XML et leur arborescence de la racine aux feuilles. Au cours du parcours de l'arbre et ce à n'importe quel moment, on peut executer des actions sur les éléments rencontrés dans l'arbre:

- Leur associer un style (font, couleur, …).

- Les recopier tels quels.
- Les ignorer.

Le principal avantage de cette structure est qu'elle permet de garder la notion de groupe ou plutôt de descendance. Par exemple, si on veut recopier un bout d'un fichier XML appelé A qui contiendrait les éléments B et C, lors de la copie de l'élément A, les éléments B et C seraient eux aussi recopiés.

La figure 1 donne un aperçu de telles transformations grâce à XSL.

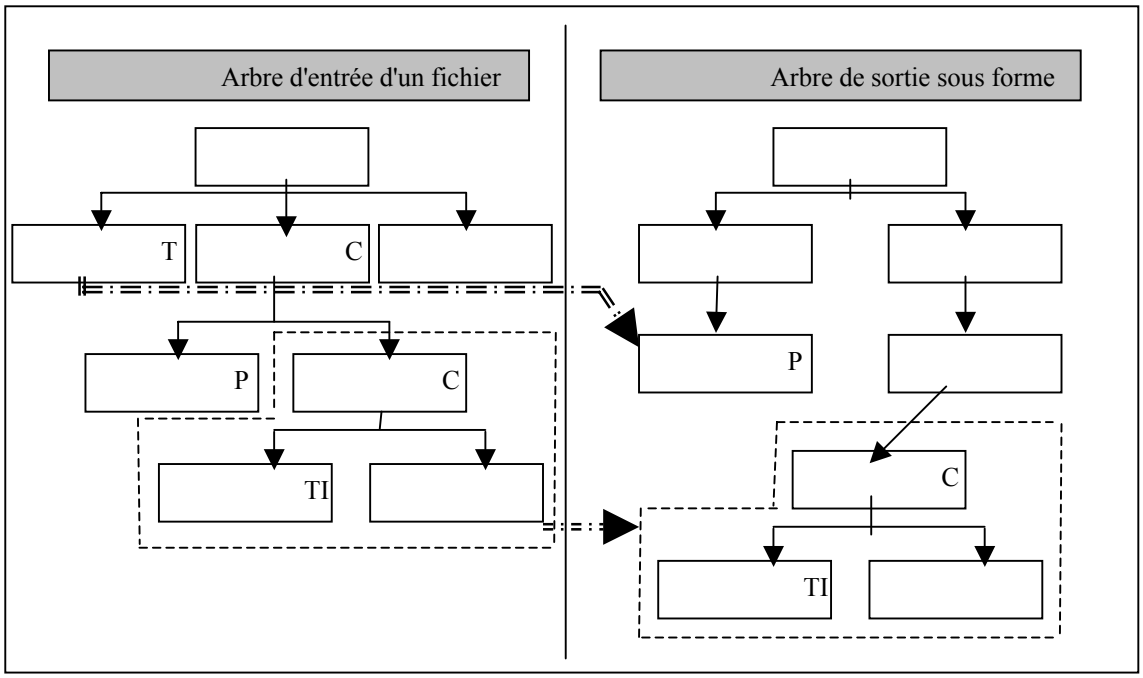


Figure 1

XSLT (XSL Transformations)

XSLT est le langage qui spécifie la conversion d'un document d'un format à un autre. La syntaxe utilisée dans XSLT concerne principalement les transformations textuelles pour exemple, XSLT sera généralement utilisé pour passer d'un fichier XML à un fichier HTML ou XHTML. Au contraire, nous n'utiliserons pas XSLT pour passer d'un fichier XML à un fichier ayant un format binaire tel qu'une image ou un fichier PDF.

Tout comme XSL, un document XSLT est toujours bien formé et valide. A XSL et XSLT, une DTD est associée qui se charge de vérifier les constructions autorisées.

Tout comme XSL encore, XSLT est basée sur une construction en forme d'arbre. Nous utiliserons là encore les mêmes termes que pour XSL, à savoir les termes de "racine" pour l'élément qui contient tous les autres, "parents" pour les éléments qui contiennent d'autres éléments, "enfants" pour les éléments qui sont contenus dans un autre élément, "feuilles" pour les éléments sans enfants.

XSLT donne un moyen de traverser les documents XML et d'appliquer diverses actions pour la transformation des données qui y sont contenues.

XML Path Language (XPath)

XPath est un langage qui permet d'adresser des nœuds dans une arborescence XML, ou plus précisément des éléments ou noms d'attributs ou valeurs d'attributs. Avec la structure de plus en plus complexe des documents XML, localiser un nœud spécifique peut être assez compliqué. L'impossibilité d'accéder à une DTD (ou à un ensemble de contraintes qui peuvent nous donner des informations sur la structure du fichier XML) rend cette localisation encore plus complexe. Pour faire cet adressage d'éléments, XPath définit une syntaxe en ligne qui allie la structure en forme d'arbre de XML et les processus XSLT.

Un peu de la même manière qu'une URL est une sorte de chemin qui étend un chemin UNIX standard, permettant de référencer n'importe quelle ressource sur Internet, XPath propose des méthodes permettant de référencer des nœuds dans une arborescence XML, à mi-chemin entre une URL et une expression régulière. Les expressions XPath sont en fait plus proches des expressions régulières dans la mesure où elles peuvent être considérées comme des règles utilisées pour définir un sous-ensemble des nœuds d'un document qui satisfont un certain nombre de propriétés.

Les syntaxes

XPath

Chaque expresion XPath est constituée de trois parties:

1. un axe
2. un test
3. un prédicat

Examinons de plus près chacune de ces trois composantes.

Les axes

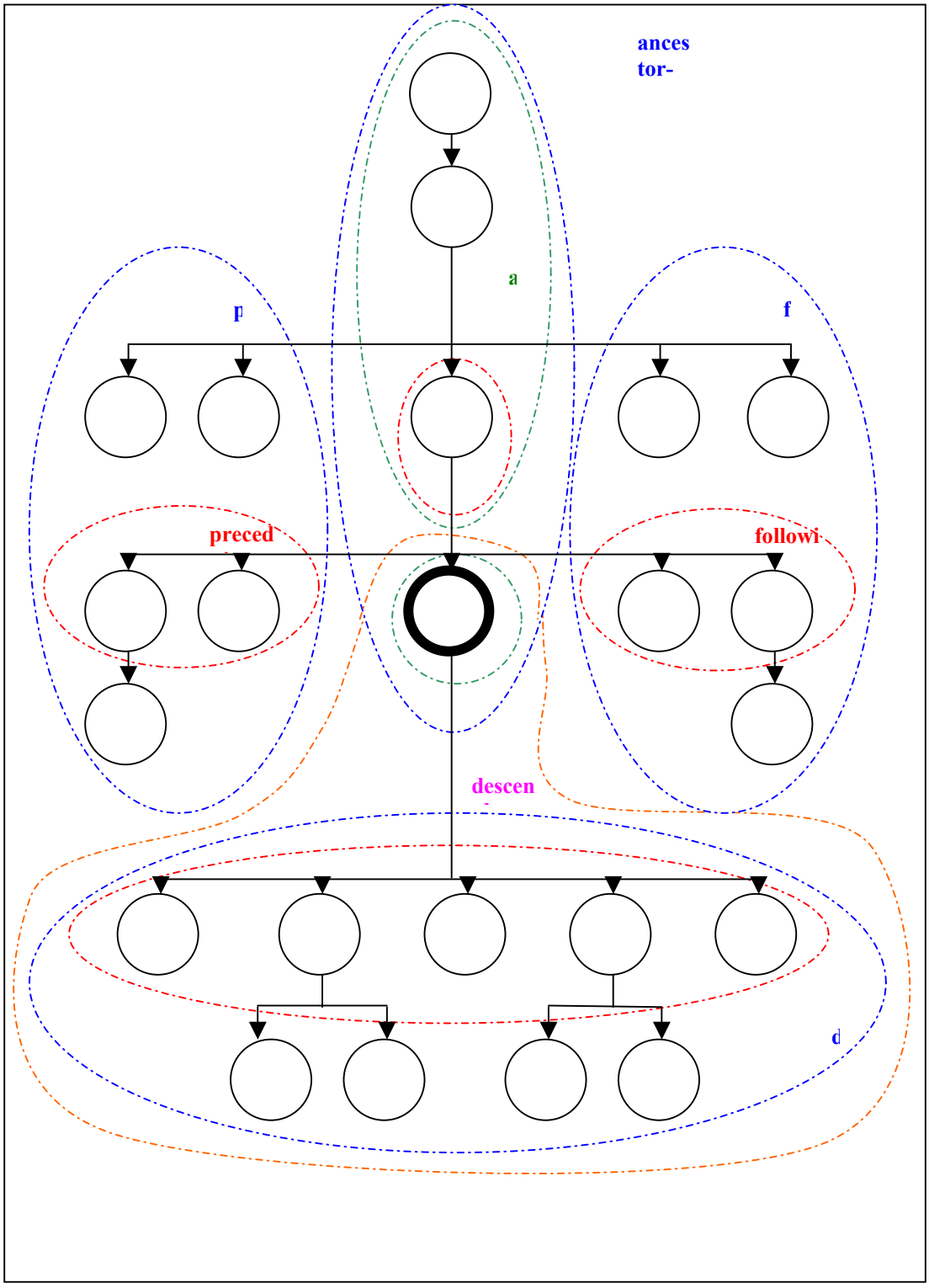
L'axe permet de définir la dimension (au sens mathématique du terme) dans laquelle l'adressage sera fait. En tout, la norme XPath définit 13 axes. Ceux-ci sont décrits dans le tableau ?.?

AXE	SIGNIFICATION
Child	Elément fils du nœud courant
Descendant	Un élément dont le nœud courant est l'ancêtre
Parent	Le parent direct du nœud courant
Ancestor	N'importe quel élément dont le nœud courant est un descendant
Attribute	Un des attributs de l'élément courant
Following	N'importe quel élément qui vient après l'élément courant dans l'ordre du document

Preceding	N'importe quel élément qui vient avant l'élément courant dans l'ordre du document
Following-sibling	N'importe quel élément qui vient après l'élément courant dans l'ordre du document et qui se trouve exactement au même niveau de l'arborescence
preceding-sibling	N'importe quel élément qui vient avant l'élément courant dans l'ordre du document et qui se trouve exactement au même niveau de l'arborescence
Namespace	N'importe quel élément qui possède un namespace donné
Self	L'élément courant
Descendant-or-self	L'élément courant ou n'importe lequel de ses descendants
ancestor-or-self	L'élément courant ou n'importe lequel de ses parents

Tableau 2

Comme le tableau précédent peut ne rien vouloir vous dire, voici un schéma qui montre de manière plus claire à quoi correspond chacun de ces axes dans une arborescence de nœuds :



Comme vous avez dû le noter, la plupart des définitions font référence au nœud courant. En effet, la plupart du temps, XPath est utilisé avec des chemins d'adressage relatif. Il est cependant possible de spécifier un chemin absolu de manière explicite en le préfixant par un slash (/) comme nous le verrons un peu plus loin.

Les tests

Après les axes, voici maintenant les tests disponibles.

TEST	SIGNIFICATION
*	N'importe quel nœud du même type que le nœud courant
Nom	Elément ou attribut (suivant le contexte) nommé nom
texte()	N'importe quel nœud texte
Processinginstruction()	N'importe quel nœud contenant une directive de traitement
node()	N'importe quel nœud
Comment()	N'importe quel nœud contenant un commentaire

Tableau 3

Si les axes et tests présentés précédemment ne vous *parlent* toujours pas, voici quelques exemples d'utilisation.

Supposons que l'on soit en train d'analyser le fichier XML défini ci-dessous:

livre.xml

```
<?xml version="1.0"?>
<livre>
   <titre> titre du livre </titre>
  <chapitre  id="these">
     <p>Premier chapitre</p>
     <t>Texte</t>
   </chapitre>
   <chapitre  id="antithese">
      <p>Deuxieme chapitre</p>
   </chapitre>
   <chapitre  id="synthese">
      <p>Toisieme chapitre</p>
   </chapitre>
   <chapitre  id="prothese">
      <p>Dernier chapitre</p>
   </chapitre>
</livre>
```

Dans ce contexte, les expressions XPath *child::p* et *descendant::p* correspondront à l'élément P qui contient le texte *Deuxieme chapitre*. Les expressions *parent::livre* et *ancestor::livre* correspondront à l'élément racine du document XML, i.e. *livre*. De même, *attribute::id* pointera vers l'attribut *id* dont la valeur est *antithese*.

Ces tests peuvent également être combinés au sein d'une seule expression XPath. Supposons que l'on veuille obtenir la liste des attributs *id* de tous les chapitres. Voici comment procéder logiquement et par étapes pour construire l'expression XPath nécessaire.

On commence tout d'abord par définir l'élément racine comme étant le contexte courant: / .

A partir de là, on recherche tous les nœuds fils du contexte courant de type chapitre: */child::chapitre*

Il ne reste plus qu'à ajouter l'expression nécessaire pour désigner les attributs correspondants (c'est-à-dire ayant pour nom *id*): */child::chapitre/*

Les prédicats

Les expressions XPath peuvent également être complétées avec un prédicat qui permet de filtrer un certain nombre de nœuds par rapport à un axe pour produire un sous-ensemble de l'expression de départ. Ce prédicat est évalué pour chacun des nœuds correspondant à l'expression et le nœud considéré n'est inclus que si et seulement si le prédicat est vrai pour le nœud en question. Chaque prédicat peut utiliser des fonctions de positionnement, des appels de fonctions, des opérateurs relationnels ou des prédicats groupés entre eux par les opérateurs *or* ou *and*.

Voici des exemples de prédicats assez souvent utilisés:

PREDICATS	SIGNIFICATION
[position() <= 3]	Est vrai pour les 3 premiers nœuds de l'ensemble
[position() mod 2 =0]	Est vrai pour les nœuds d'ordre pair dans l'ensemble
[position() = last()]	Est vrai pour le dernier nœud de l'ensemble
[substring(string(),2,5) = "MathML"]	Est vrai pour les nœuds dont la représentation sous forme de chaîne contient la sous chaîne MathML à partir du deuxième caractère
Auteur[normalize-space(prenom)="Ludo"	Est vrai pour les nœuds de type auteur qui possèdent un nœud fils prenom dont la valeur, sans tenir compte des espaces en début et fin de chaîne est Ludo

Tableau 4

Ainsi, en reprenant comme document de base notre exemple précédent, l'expression */child::chapitre[position() <= 2]/attribute::id* donne seulement la liste des attributs *id* pour les deux premiers chapitres (en fait, pour les deux premiers fils de l'élément courant qui est la racine, c'est-à-dire *livre)*.

La syntaxe XPath peut vous sembler plutôt fastidieuse et difficile à mettre en pratique. Il existe aussi une syntaxe abrégée qui permet de raccourcir la plupart des expressions.

Le tableau suivant donne un aperçu des principales d'entre elles.

FORME LONGUE	FORME ABREGEE	SIGNIFICATION
child::Fiat/child::Uno	Fiat/Uno	Sélectionne tous les nœuds de type Uno qui sont fils d'un nœud de type Fiat étant lui-même fils du nœud courant
child::personne[attribute::name ="Ludo"]	Personne[@nom="Ludo "]	Sélectionne les nœuds fils du nœud qui ont un attribut nom dont la valeur est Ludo
list/descendant-or-self ::node()/child::item	list//item	Sélectionne tous les éléments item qui sont descendants d'un élément list fils du nœud courant
parent::node()/child::Uno	../Uno	Sélectionne les nœuds Uno dont le père est également le père du nœud courant

child::item[position()=4] 'item'[attribute::type="fill" and substring(attribute::style,3)="alic"]	Item[4][@type="fill" and substring(@style,2)="alic"]	Sélectionne le quatrième nœud fils, de type du nœud courant à condition qu'il possède un attribut type ayant pour valeur fill et un attribut style qui contienne la sous chaîne alic à partir du deuxième caractère

Tableau 4

A ce stade, nous en savons assez pour nous repérer dans un document XML grâce à XPath.

XSLT

Nous pouvons dès à présent passer à l'écriture de la feuille de style XSL proprement dite.

Le langage de transformation XSL est basé sur la reconnaissance de certaines balises XML auxquelles on peut associer un texte de remplacement (appelé une *template*). Chaque fois que le parseur XSLT rencontrera un élément XML qui correspond (en anglais qui "*match*" une des *templates* de la feuille de style, alors l'élément sera remplacé en fonction des directives données dans la feuille de style. D'autre part, une feuille de style XSL doit impérativement être un document XML bien formé comme nous l'avons déjà dit. Toutes les directives utilisées pour les transformations XSL sont des éléments situés dans l'espace de nommage *xsl:* afin d'éviter d'éventuelles collisions.

Les principaux éléments utilisables dans une feuille de style sont les décrits dans les pages suivantes.

xsl:stylesheet

Description:

```
<xsl:stylesheet>
…
</xsl:stylesheet>
```

Utilisation:

Cet élément doit être absolument le nom racine du document. Il indique clairement que le document XML qui le contient est bien une feuille de style. Cette balise peut (et c'est recommandé) contenir des attributs qui sont des références à des espaces de nommage. Rappelons que les espaces de nommage sont des déclarations qui permettent de mélanger les marqueurs définis dans des DTD différentes dans un seul document XML. Par exemple, supposons que nous ayons déjà à notre disposition un ensemble de marqueurs XML pour décrire des dates et que nous désirions le réutiliser pour décrire les dates de notre fichier XML. Le fait de mélanger des marqueurs de différentes origines introduit un grave problème: le risque de conflit entre des noms de marqueurs provenant d'origines différentes. Les espaces de nommage permettent d'éviter ce genre de problèmes. Chaque déclaration d'espace de nommage introduit deux données: l'URL où il est possible de trouver la DTD correspondante et le préfixe qui doit être appliqué aux éléments qui en proviennent. Ainsi, en préfixant nos éléments avec l'espace de nommage *xsl*, on indique à notre parseur quelle URI il doit rechercher pour comprendre les balises *xsl*. L'espace de nommage pour *xsl* est situé sur le site du W3C et la version la plus récente se trouve à l'URI suivante: http://www.w3.org/1999/XSL/Transform .

Exemple:

```
<?xml version="1.0">
<!—un fichier xsl est par défaut un fichier xml, c'est donc normal
qu'il commence ainsi --!>

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

</xsl:stylesheet>
```

xsl:template

Description:

```
<xsl:template>
…
</xsl:template>
```

Utilisation:

C'est l'élément de base des feuilles de style dans la mesure où il permet de localiser un élément particulier ou un ensemble d'éléments à partir d'un document XML et il est à la base des transformations que l'on peut effectuer sur cet (ou ces) élément(s).

Il peut être utilisé avec un attribut *match* correspondant à une expression XPath. Dans ce cas, il permet de déterminer quels nœuds du document XML original donneront lieu à une substitution (cf xsl:apply-templates).

L'attribut *name* est également utilisable: il permet d'assigner directement un nom à une template, et de faire appel à elle explicitement plutôt que d'utiliser une liste de nœuds.

Exemple:

Supposons que l'on veuille appliquer une feuille de style(livre1.xsl) à livre.xml donné précédemment. Elle peut ressembler à l'exemple suivant même si, d'un point de vue pédagogique, cela ne nous apporte pas grand chose dans la mesure où on ne peut toujours pas récupérer d'information à partir de ce marqueur. Nous verrons un exemple plus utile dans le paragraphe dédié à la balise <*xsl:apply-templates*>.

livre1.xsl et livre1out.xml

```
<?xml version="1.0"?>
<!—-un fichier xsl est par défaut un fichier xml, c'est donc
normal qu'il commence ainsi -->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              version="1.0"
>
<xsl:output method="xml" indent="yes"/>
<!--pour avoir le fichier de sortie sous forme xml et indenté -->
```

```
<xsl:template match="livre">
   <html>
      <head>
         <title>titre du livre</title>
      </head>
      <body/>
   </html>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<html>
    <head>
        <title>titre du livre</title>
    </head>
    <body/>
</html>
```

---

xsl:apply-templates

Description:

```
<xsl:apply-templates [select="expression"]>
…
</xsl:apply-templates>

ou le plus souvent utilisé de la sorte:

<xsl:apply-templates [select="expression"]/>
```

Utilisation:

Permet de forcer l'application d'une template si l'expression passée en paramètre est évaluée à vrai.

Exemple 1 (utilisation de *xsl:apply-templates* sans attribut):

En associant la feuille de style livre2.xsl à livre.xml, on obtient le nouveau fichier xml livre2out.xml. Dans cet exemple, on désire récupérer tous les contenus des balises contenant des nœuds de type texte.

livre2.xsl et livre2out.xml

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

<xsl:template match="livre">
```

```
<book>
      <xsl:apply-templates />
</book>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<book>
    titre du livre

     Premier chapitre
     Texte


      Deuxieme chapitre


      Toisieme chapitre


      Dernier chapitre

</book>
```

Exemple 2 (utilisation de *apply-templates* avec l'attribut *select*):

En associant la feuille de style livre3.xsl à livre.xml, on obtient le nouveau fichier xml livre3out.xml. Ici, on ne veut que le contenu de la balise<t>.

livre3.xsl et livre3out.xml

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

<xsl:template match="livre">
<text>
      <xsl:apply-templates select="chapitre/t"/>
</text>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<text>Texte</text>
```

Exemple 3 (utilisation de *apply-templates* avec l'attribut *select* mixé avec *templates*):

En associant la feuille de style livre4.xsl à livre.xml, on obtient le nouveau fichier xml livre4out.xml. On veut placer les contenus des balises <p> entre des marqueurs <saut>.

Livre4.xsl et livre4out.xml

```
<?xml version="1.0"?>
```

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

<xsl:template match="livre">
   <paragraphe>
      <xsl:apply-templates select="chapitre/p"/>
   </paragraphe>
</xsl:template>

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

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<paragraphe>
    <saut>Premier chapitre</saut>
    <saut>Deuxieme chapitre</saut>
    <saut>Toisieme chapitre</saut>
    <saut>Dernier chapitre</saut>
</paragraphe>
```

xsl:value-of

Descripton:

```
<xsl:value-of select="expression_XPath">
</ xsl:value-of>
```

Utilisation:

Insère la valeur de l'expression XPath, qui peut aussi bien être le résultat d'une expression arithmétique que la représentation textuelle d'un nœud ou d'un attribut.

Exemple 1:

On ne veut récupérer que le texte entre les balises <titre> et </titre>. Pour cela, on applique *livre5.xsl* à *livre.xml*.

livre5.xsl et livre5out.xml

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <html>
      <head>
         <title><xsl:value-of select="titre" /></title>
      </head>
      <body>
```

```
         corps du livre
      </body>
   </html>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<html>
<head>
<title> titre du livre </title>
</head>
<body>
      corps du livre
</body>
</html>
```

Exemple 2:

On désire maintenant combiner les deux marqueurs *<xsl:value-of>* et *<xsl:apply-templates>*. De plus, on introduit une fonction *not()* qui prend en paramètre une expression XPath. livre6.xsl va nous permettre de mettre le contenu de la balise titre entre les marqueurs <title> puis de placer tous les autres contenus de marqueurs dans le corps, i.e. entre les balises <body>.

livre6.xsl et livre6out.xml

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <html>
      <head>
         <title><xsl:value-of select="titre" /></title>
      </head>
      <body>
         <xsl:apply-templates select="*[not(self::titre)]" />
      </body>
   </html>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<html>
    <head>
        <title> titre du livre </title>
    </head>
    <body>
     Premier chapitre
     Texte

      Deuxieme chapitre

      Toisieme chapitre

      Dernier chapitre
```

```
   </body>
</html>
```

xsl:if

Description:

```
<xsl:if test="expression_XPath">
{corps du test}
</xsl:if>
```

Utilisation:

L'expression XPath est évaluée puis passée en paramètre à la fonction boolean(). Si la valeur retournée est vraie, alors le corps du test est exécuté.

Notez qu'il n'existe pas d'équivalent au *else* du classique *if ... else ....* Si vous en avez besoin, utilisez plutôt la balise <xsl:choose> exposée plus bas.

La fonction boolean() fonctionne de la manière suivante selon le type de son paramètre:

- nombre en paramètre: retourne vrai si et seulement si le nombre est non nul ou s'il n'est pas considéré pour un nombre (NaN).
- ensemble de nœuds en paramètre: retourne vrai si et seulement si l'ensemble est non vide.
- chaîne en paramètre: retourne vrai si la longueur de la chaîne est non nulle.

Exemple :

Supposons que nous ne voulions écrire dans le corps de notre fichier de sortie qu'une petite phrase lorsque nous arrivons au chapitre *these*. La feuille de style *livre7.xsl* fournit le résultat suivant:

livre7.xsl et livre7out.xml

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <html>
      <head>
      </head>
      <body>
         <xsl:apply-templates select="*[not(self::titre)]" />
      </body>
   </html>
</xsl:template>
```

```
<xsl:template match="chapitre">
   <xsl:if test="@id='these'">
      c'est la these
   </xsl:if>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<html>
    <head/>
    <body>
      c'est la these
   </body>
</html>
```

xsl:for-each

Description:

```
<xsl:for-each select="expression_XPath">
[corps de la boucle]
</xsl:for-each>
```

Utilisation:

Comme tout langage de programmation qui se respecte (et XSL en est un à sa façon), il est possible d'utiliser des boucles. Aussi, le fonctionnement de *xsl:for-each* est sans surprise: il se contente de répéter le corps de la boucle pour chacun des nœuds correspondant à l'expression XPath qui lui est passée en paramètre. A l'intérieur de la boucle, chacun de ces nœuds devient l'élément courant.

Exemple:

Pour chaque chapitre, on veut récupérer le contenu des balises <p> qu'on placera entre des balises <chap>.

livre8.xsl et livre8out.xml

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <html>
      <head>
         <titre><xsl:value-of select="titre" /></titre>
      </head>
      <body>
         <xsl:for-each select="chapitre">
            <chap>
```

```
                <xsl:value-of select="p" />
            </chap>
         </xsl:for-each>
      </body>
   </html>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<html>
    <head>
        <titre> titre du livre </titre>
    </head>
    <body>
        <chap>Premier chapitre</chap>
        <chap>Deuxieme chapitre</chap>
        <chap>Toisieme chapitre</chap>
        <chap>Dernier chapitre</chap>
    </body>
</html>
```

---

xsl:choose

Description:

```
<xsl:choose>
<xsl:when test="expression">
{corps du when}
</xsl:when>
{possibilité de mettre autant de xsl:when à la suite que l'on
veut}
<xsl:otherwise>
{corps de otherwise}
</xsl:otherwise>
</xsl:choose>
```

Utilisation:

C'est en fait l'équivalent de l'instruction *switch* en XSL. Chacun des éléments <*xsl:when*> fonctionne comme un <xsl:if>, et si aucun d'entre eux n'a été *'passé'*,alors c'est l'élément <*xsl:otherwise*> qui sera utilisé, à condition qu'il y en ait un car il est facultatif.

Exemple:

Dans set exemple, on parcourt chacun des chapitres. Pour chaque chapitre, on regarde si le chapitre possède un attribut *id* de valeur *these* ou *antithese*. Si c'est le cas, on met un message l'indiquant respectivement entre les bornes <*these*> et <*antithese*>. Sinon, on met un autre message entre les marqueurs <*autre*>.

livre9.xsl et livre9out.xml

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <xsl:for-each select="chapitre">
      <xsl:choose>
         <xsl:when test="@id='these'">
            <these>voici la these</these>
         </xsl:when>
         <xsl:when test="@id='antithese'">
            <antithese>voici l'antithese</antithese>
         </xsl:when>
         <xsl:otherwise>
            <autre>ce n'est ni la these ni l'antithese</autre>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
< these >voici la these</these>
<antithese>voici l'antithese</ antithese >
<autre>ce n'est ni la these ni l'antithese</autre>
<autre>ce n'est ni la these ni l'antithese</autre>
```

xsl:element

Description:

```
<xsl:element name="expression">
{contenu}
</xsl:element>
```

Utilisation:

Cet élément permet d'insérer un élément dont le nom est calculé et non pas seulement écrit dans la feuille de style.

xsl:attribute

Description:

```
<xsl:attribute>
```

```
…
</xsl:attribute>
```

Utilisation:

Cet élément permet d'ajouter un attribut au dernier marqueur ouvert.

Exemple commun aux deux marqueurs:

Livre10.xsl et livre10out.xml

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                    version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <XML>
   <xsl:for-each select="chapitre">
      <xsl:element name="chapitre">
         <xsl:attribute name="nom">
            <xsl:value-of select="p" />
         </xsl:attribute>
         <xsl:value-of select="t" />
      </xsl:element>
   </xsl:for-each>
   </XML>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<XML>
    <chapitre nom="Premier chapitre">Texte these</chapitre>
    <chapitre nom="Deuxieme chapitre"/>
    <chapitre nom="Toisieme chapitre"/>
    <chapitre nom="Dernier chapitre"/>
</XML>
```

xsl:sort

Description:

```
<xsl:sort
select="expression"
data-type={"text" | "number"}
order={"ascending" | "descending"}
case-order={"upper-first" | "lower-first"}/>
```

Utilisation:

Lorsqu'il est utilisé à l'intérieur d'un marqueur *<xsl:apply-templates>* ou d'une boucle *<xsl:for-each>, <xsl:sort>* permet de trier la liste des éléments désignés par l'attribut *select*. Divers attributs sont disponibles pour spécifier l'ordre de tri: ordre croissant, décroissant, sensible ou non à la casse, tri de valeurs numériques ou alphabétiques.

Exemple:

Pour trier les chapitre dans l'ordre ascendant de la valeur de leur attribut *id*, on procédera de la sorte:

Livre11.xsl et livre11out.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                    version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <XML>
      <xsl:apply-templates select="chapitre">
         <xsl:sort select="@id" data-type="text" order="ascending"
/>
      </xsl:apply-templates>
   </XML>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<XML>
      Deuxieme chapitre

      Dernier chapitre

      Toisieme chapitre

     Premier chapitre
     Texte these
</XML>
```

xsl:text

Description:

```
<xsl:text>
…
</xsl:text>
```

Utilisation:

Le contenu des éléments *<xsl:text>* est recopié dans le document final sans tentative d'analyse par le parseur. Par défaut, les caractères d'espacement sont préservés.

Exemple:

A l'exemple précédent, on rajoute du texte.

Livre11.xsl et livre11out.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                    version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="livre">
   <XML>
      <xsl:apply-templates select="chapitre">
         <xsl:sort select="@id" data-type="text" order="ascending"
/>
      </xsl:apply-templates>
   </XML>
   <text><xsl:text>et un peu de texte pour finir</xsl:text></text>
</xsl:template>

</xsl:stylesheet>

<?xml version="1.0" encoding="UTF-8"?>
<XML>
      Deuxieme chapitre

      Dernier chapitre

      Toisieme chapitre

     Premier chapitre
     Texte these
</XML>
<text>et un peu de texte pour finir</text>
```

Xalan et Xerces

Maintenant que nous avons vu la théorie sur les feuilles de style XSL, nous pouvons vous montrer comment faire fonctionner les divers exemples donnés dans les pages précédentes sur votre ordinateur.

Pour ce faire, nous avons décidé d'utiliser les outils suivants:

- Xalan-Java version 1.2.2

- Xerces-java version 1.2.2

Cependant, de nouvelles versions de Xalan et Xerces ont vu le jour depuis (cf chapitre sur le parseur)

Nous avons aussi besoin de la dernière version du JDK existante (au moins la 1.1.8).

A quoi vont ils nous servir?

Xalan-Java version 1.2.2 est une API Java qui comprend un processeur XSLT, outil indispensable pour exécuter des transformations XML.

Xerces-java version 1.2.2 est le parseur XML qui est interfacé directement avec Xalan.

Pourquoi ces outils plutôt que d'autres?

Tout simplement parce qu'ils sont les plus aboutis sur le marché actuellement et qu'ils ont été cités en référence dans plusieurs magazines spécialisés et dans plusieurs livres.

Où les trouver?

Les deux Xalan-Java version 1.2.2 et Xerces-java version 1.2.2 sont réunis dans un fichier zippé à l'adresse suivante: http://xml.apache.org/dist/xalan-j/xalan-j_1_2_2.zip ou sous forme tar.gz à l'adresse: http://xml.apache.org/dist/xalan-j/ xalan-j_1_2_2.tar.gz.

Comment les installer?

Après s'être procuré le fichier compressé, il faut le décompresser et inclure au minimum les fichiers xalan.jar et xerces.jar dans votre chemin de classes (classpath) dans l'autoexec.bat (sous Windows 98). Toutefois, nous vous recommandons d'y ajouter le fichier xalansamples.jar qui va vous permettre de faire exécuter les exemples proposés dans Xalan. C'est tout.

Installation :

```
set
CLASSPATH=%CLASSPATH%;c:\xalan122\xerces.jar;c:\xalan122\xalan.jar
;c:\xalan122\samples\xalansamples.jar
```

Utilisation:

Voilà, maintenant vous pouvez essayez les différents exemples, tant ceux donnés dans les pages précédentes que ceux de Xalan.

Pour les exemples du chapitre XSLT, utilisez le programme fourni dans les exemples de Xalan qui s'intitule SimpleTransform.java en changeant uniquement les noms des fichiers utilisés au sein du programme même (voir exemple 1).

Exemple 1:

On se propose de faire fonctionner l'exemple donné dans le descriptif de xsl:templates, i.e. appliquer livre1.xsl à livre.xml pour donner le fichier livre1out.xml. les fichiers

Le contenu des fichiers est déjà exposé au chapitre XSLT. livre1.xsl et livre.xml sont à enregistrer dans le répertoire Samples/SimpleTransform.

Le programme `SimpleTransform.java` est donné à la page suivante.

Après compilation (javac SimpleTransform.java) et exécution (java SimpleTransform) de SimpleTransform.java dans le répertoire Samples/SimpleTransform, nous obtenons le fichier livre1out.xml.

SimpleTransform.java

```
import org.xml.sax.SAXException;
import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTResultTarget;
import org.apache.xalan.xslt.XSLTProcessor;

/**
 * Simple sample code to show how to run the XSL processor
 * from the API.
 */
public class SimpleTransform
{
public static void main(String[] args) throws java.io.IOException,

java.net.MalformedURLException,

org.xml.sax.SAXException
{
    // Have the XSLTProcessorFactory obtain a interface to a
    // new XSLTProcessor object.
   XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

    // Have the XSLTProcessor processor object transform
"livre.xml" to
      // System.out, using the XSLT instructions found in
"livre1.xsl".
    processor.process(new XSLTInputSource("livre.xml"),
                        new XSLTInputSource("livre1.xsl"),
                      new XSLTResultTarget("livre1out.xml"));
    System.out.println("resultat dans livre1out. xml ");
}
}
```

De l'arbre Tex à l'arbre MathML

Dorénavant, vous êtes au courant de tout ce dont on a besoin pour le passage de l'arbre Tex à l'arbre MathML. Nous allons ici décomposer notre feuille de style XSL de transformation afin de montrer de quelle manière nous agissons sur différents éléments d'un arbre Tex pour les transformer en éléments MathML.

Les nombres et les variables

Les nombres et variables sont simplement délimités en Tex par les marqueurs <NUMBER> et <VAR>. Ils sont retranscrits tels quels entre les balises <cn> pour les nombres et <ci> pour les variables.

100

arbre Tex

```
<?xml version='1.0'?>
<TEX>
   <NUMBER>100</NUMBER>
</TEX>
```

fichier de transformation XSL

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:output method="xml" indent="yes"/>

<xsl:template match="TEX">
<math>
   <mrow>
      <xsl:apply-templates select="*"/>
   </mrow>
</math>
</xsl:template>

<xsl:template match="NUMBER">
   <cn><xsl:apply-templates select="./text()"/></cn>
</xsl:template>

<xsl:template match="VAR">
   <ci><xsl:apply-templates select="./text()"/></ci>
</xsl:template>

</xsl:stylesheet>
```

arbre MathML

```
<?xml version="1.0" encoding="UTF-8"?>
<math>
    <mrow>
        <cn>100</cn>
    </mrow>
</math>
```

les opérateurs +, -, /,*, = et l'opérateur de multiplication non visible.

Les opérateurs =, +, -,* et / opèrent les mêmes transformations. L'arbre Tex définit l'écriture de '100+x' en Tex. Elle consiste en une balise <PLUS> possédant deux fils qui sont en fait les deux termes additionnés. L'arbre MathML s'écrit légèrement différemment puisque l'écriture se fait quasiment en ligne.

Voici l'exemple de l'addition. Donc, dès qu'on rencontre la balise <PLUS> dans le fichier Tex, on recherche son premier fils en y appliquant les autres règles du fichier XSL, on ajoute la balise MathML d'addition qui est <mo>+</mo>, puis on applique les autres règles de transformation au deuxième fils.

arbre Tex

```
<?xml version='1.0'?>

<TEX>
   <PLUS>
      <NUMBER>100</NUMBER>
      <VAR>x</VAR>
   </PLUS>
</TEX>
```

fichier de transformation XSL

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:output method="xml" indent="yes"/>

<xsl:template match="TEX">
<math>
   <mrow>
      <xsl:apply-templates select="*"/>
   </mrow>
</math>
</xsl:template>

<xsl:template match="NUMBER">
   <cn><xsl:apply-templates select="./text()"/></cn>
</xsl:template>

<xsl:template match="VAR">
   <ci><xsl:apply-templates select="./text()"/></ci>
</xsl:template>

<xsl:template match="PLUS">
   <xsl:apply-templates select="*[position()=1]"/>
   <mo>+</mo>
   <xsl:apply-templates select="*[position()=2]"/>
</xsl:template>
```

arbre MathML

```
<?xml version="1.0" encoding="UTF-8"?>
<math>
    <mrow>
```

```
        <cn>100</cn>
        <mo>+</mo>
        <ci>x</ci>
    </mrow>
</math>
```

l'opérateur exposant (^), les fractions

Ils sont différents des précédents car même s'ils s'écrivent de la même manière en Tex que + ou -, ils s'écrivent aussi de la même manière en MathML aux noms de marqueurs près.

X^100

arbre Tex

```
<?xml version='1.0'?>

<TEX>
   <POWER>
      <VAR>X</VAR>
      <NUMBER>100</NUMBER>
   </POWER>
</TEX>
```

fichier de transformation XSL

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:output method="xml" indent="yes"/>

<xsl:template match="TEX">
<math>
   <mrow>
      <xsl:apply-templates select="*"/>
   </mrow>
</math>
</xsl:template>

<xsl:template match="NUMBER">
   <cn><xsl:apply-templates select="./text()"/></cn>
</xsl:template>

<xsl:template match="VAR">
   <ci><xsl:apply-templates select="./text()"/></ci>
</xsl:template>

<xsl:template match="POWER">
   <msup>
      <xsl:apply-templates select="*[position()=1]"/>
      <xsl:apply-templates select="*[position()=2]"/>
   </msup>
</xsl:template>
```

arbre MathML

```
<?xml version="1.0" encoding="UTF-8"?>
<math>
    <mrow>
        <msup>
            <mrow>
                <ci>X</ci>
                <cn>100</cn>
            </mrow>
        </msup>
    </mrow>
</math>
```

$$\frac{100}{X}$$

arbre Tex

```
<?xml version='1.0'?>

<TEX>
   <FRAC>
      <ARGUMENT ENDMARKER=')' STARTMARKER='('>
      <NUMBER>100</NUMBER>
   </ARGUMENT>
   <ARGUMENT ENDMARKER=')' STARTMARKER='('>
         <VAR>X</VAR>
      </ARGUMENT>
   </FRAC>
</TEX>
```

fichier de transformation XSL

```
…
<xsl:template match="FRAC">
   <mfrac>
      <mrow>
         <xsl:apply-templates select="*[position()=1]"/>
      </mrow>
      <mrow>
         <xsl:apply-templates select="*[position()=2]"/>
      </mrow>
   </mfrac>
</xsl:template>
…
```

arbre MathML

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<math>
    <mrow>
        <mfrac>
            <mrow>
                <cn>100</cn>
            </mrow>
            <mrow>
                <ci>X</ci>
            </mrow>
        </mfrac>
    </mrow>
</math>
```

les sommes et les intégrales

En MathML, elles s'écrivent de la même façon:

Une borne <msubsup> pour les intégrales et < munderover> pour les sommes indiquent qu'on va placer des indices au-dessus et en dessous d'une forme donnée en premier fils de chacune de ces balises. Pour les sommes cette balise se caractérise par <mo><mchar name='sum' /></mo> et pour les intégrales, c'est<mo><mchar name='int' /></mo>, les deux autres fils étant les expressions qu'on va placer en bas et en haut de cette forme. Au même niveau que <msubsup> ou que < munderover>, on va mettre le terme qui est sommé.

Dans l'arbre Tex, la somme est totalement séparée du corps de sommation par un marqueur <INVISIBLETIMES> qui indique clairement le lien entre les deux. La somme et l'intégrale sont représenté par les marqueurs <SUM> et <INTEG> qui contiennent au plus deux fils qui sont les délimiteurs de la somme (limites inférieure et limite supérieure respectivement symbolisés par les balises <LOWERBOUND> et <UPPERBOUND>. Au même niveau que <INVISIBLETIMES>, on trouve l'expression qui est sommée.

Ainsi, à la balise <INVISIBLETIMES>, on va récupérer les données du premier fils, puis du deuxième après avoir placé entre ces deux informations une balise <mo> <mchar name="InvisibleTimes" /> </mo> pour indiquer que le premier fils est lié au deuxième.

Le premier fils sera une balise <SUM> ou <INTEG> et contiendra au plus deux fils, les délimiteurs inférieurs et supérieurs symbolisés par les marqueurs. <LOWERBOUND> et <UPPERBOUND>. Arrivés à la balise <SUM> (<INTEG>), on placera dans notre fichier de sortie MathML la balise < munderover> (< msubsup>) à laquelle on associera les fils <mo> <mchar name="Sum" /> </mo> (<mo> <mchar name="int" /> </mo>) ainsi que les contenus des balises <LOWERBOUND> et <UPPERBOUND>. Ces dernières agissent commes des balises <EXPRESSION> que nous allons voir dans les pages suivantes.

Mais peut être que l'exemple suivant sera plus parlant que toutes ces phrases qui pourraient vous paraître obscures.

$$2+\int_{i=1}^{10} i$$

arbre Tex

```
<?xml version='1.0'?>

<TEX>
   <PLUS>
      <NUMBER>2</NUMBER>
      <INVISIBLETIMES>
         <INTEG>
```

```
            <LOWERBOUND>
               <EQUAL>
                  <VAR>i</VAR>
                  <NUMBER>1</NUMBER>
               </EQUAL>
            </LOWERBOUND>
            <UPPERBOUND>
               <NUMBER>10</NUMBER>
            </UPPERBOUND>
         </INTEG>
         <VAR>i</VAR>
      </INVISIBLETIMES>
   </PLUS>
</TEX>
```

fichier de transformation XSL

```
…
<xsl:template match="INTEG">
   <msubsup>
      <mrow>
         <mo><mchar name='int'/></mo>
         <xsl:apply-templates select="LOWERBOUND"/>
         <xsl:apply-templates select="UPPERBOUND"/>
      </mrow>
   </msubsup>
</xsl:template>

<xsl:template match="INVISIBLETIMES">
   <mrow>
      <xsl:apply-templates select="*[position()=1]" />
      <mo> <mchar name="InvisibleTimes" /> </mo>
      <xsl:apply-templates select="*[position()=2]" />
   </mrow>
</xsl:template>
…
```

arbre MathML

```
<?xml version="1.0" encoding="UTF-8"?>
<math>
    <mrow>
        <cn>2</cn>
        <mo>+</mo>
        <mrow>
            <msubsup>
                <mrow>
                    <mo>
                        <mchar name="int"/>
                    </mo>
                    <mrow>
                        <mrow>
                            <ci>i</ci>
                        </mrow>
                        <mo>=</mo>
                        <mrow>
```

```
                        <cn>1</cn>
                    </mrow>
                </mrow>
                <mrow>
                    <cn>10</cn>
                </mrow>
            </mrow>
        </msubsup>
        <mo>
            <mchar name="InvisibleTimes"/>
        </mo>
        <ci>i</ci>
    </mrow>
</mrow>
</math>
```

---

les fonctions

Dans la DOM TEX, les fonctions suivantes sont mises au même niveau: `sin, cos, tan, cot, sinh, cosh, tanh, coth, arccos, arcsin, arctan, log, logn, ln, sqrt, exp.` Elles sont désignées par le marqueur <FUNCTION> avec les attributs suivants:

- NAME pour le nom de la fonction (ex. NAME='cos' pour le cosinus)
- ENDMARKER pour le type de délimiteur de fin d'arguments (ex. ENDMARKER=']')
- STARTMARKER pour le type de délimiteur de début d'arguments (ex. STARTMARKER='{')

En MathML, cela se passe légèrement différemment dans la mesure où la fonction carrée (sqrt) et l'exponentielle ne sont pas considérés comme les autres fonctions. L'exponentielle est en fait une puissance et la fonction racine carrée s'écrit avec le symbole $\sqrt{}$ alors que toutes les autres fonctions (qu'on appellera ici fonctions classiques pour simplifier) s'écrivent telles quelles avec un (ou plusieurs) paramètre entre parenthèses (ou du moins entre des délimiteurs).

Les fonctions dites classiques s'écrivent donc en dans l'arbre MathML de la façon suivante:

- lorsqu'on trouve la balise <FUNCTION>, on récupère la valeur de l'attribut NAME qu'on place entre des marqueurs <mi>
- on annonce que cette dernière est une fonction en rajoutant la ligne <mo> <mchar name="ApplyFunction" /> </mo>
- on récupère la valeur de l'attribut STARTMARKER
- on récupère les données du fils de <FUNCTION>
- on récupère la valeur de l'attribut ENDMARKER

Pour la fonction exponentielle, on procède comme pour une puissance. Pour la fonction racine carrée, on procède comme pour la fonction exponentielle, à ceci près que la fonction racine carrée ne possède qu'un fils en MathML (pas forcément vrai pour les racines $n^{ièmes}$).

$$\sin(x) = x + \frac{x^3}{3!} + ...$$

arbre Tex

```
<?xml version='1.0'?>

<TEX>
   <EQUAL>
      <FUNCTION NAME='sin' STARTMARKER='(' ENDMARKER=')'>
         <VAR>x</VAR>
      </FUNCTION>
      <PLUS>
         <PLUS>
            <VAR>x</VAR>
            <FRAC>
               <ARGUMENT ENDMARKER=')' STARTMARKER='('>
                  <POWER>
                     <VAR>x</VAR>
                     <NUMBER>3</NUMBER>
                  </POWER>
               </ARGUMENT>
               <ARGUMENT ENDMARKER=')' STARTMARKER='('>
                  <FACTORIAL>
                     <NUMBER>3</NUMBER>
                  </FACTORIAL>
               </ARGUMENT>
            </FRAC>
         </PLUS>
         <CDOTS/>
      </PLUS>
   </EQUAL>
</TEX>
```

fichier de transformation XSL

```
<xsl:template match="FUNCTION">
   <mrow>
      <xsl:choose>
                              <xsl:when test="@NAME='sin' or
@NAME='cos' or @NAME='tan' or NAME='cot' or @NAME='sinh' or
@NAME='cosh' or @NAME='tanh' or @NAME='coth' or NAME='arccos' or
@NAME='arcsin' or @NAME='arctan' or @NAME='log' or @NAME='logn' or
@NAME='ln'" >
            <mi><xsl:apply-templates select="@NAME" /></mi>
            <mo> <mchar name="ApplyFunction" /> </mo>
               <mrow>
                  <mo><xsl:value-of select="@STARTMARKER" /></mo>
                     <mrow>
                        <xsl:apply-templates select="child::*"/>
                     </mrow>
                  <mo><xsl:value-of select="@ENDMARKER" /></mo>
```

```
            </mrow>
         </xsl:when>
         <xsl:when test="@NAME='sqrt'">
            <msqrt>
               <mrow>
                  <xsl:apply-templates select="child::*"/>
               </mrow>
            </msqrt>
         </xsl:when>
         <xsl:when test="@NAME='exp' or @NAME='e'">
            <msup>
               <mrow>
                  <mi><xsl:value-of select="@NAME" /></mi>
                  <xsl:apply-templates select="child::*"/>
               </mrow>
            </msup>
         </xsl:when>
      </xsl:choose>
   </mrow>
</xsl:template>
```

arbre MathML

```
<?xml version="1.0" encoding="UTF-8"?>
<math>
    <mrow>
        <mrow>
            <mrow>
                <mi>sin</mi>
                <mo>
                    <mchar name="ApplyFunction"/>
                </mo>
                <mrow>
                    <mo>(</mo>
                    <mrow>
                        <ci>x</ci>
                    </mrow>
                    <mo>)</mo>
                </mrow>
            </mrow>
        </mrow>
        <mo>=</mo>
        <mrow>
            <ci>x</ci>
            <mo>+</mo>
            <mfrac>
                <mrow>
                    <msup>
                        <mrow>
                            <ci>x</ci>
                            <cn>3</cn>
                        </mrow>
                    </msup>
                </mrow>
                <mrow>
                    <cn>3</cn>
                    <mo>!</mo>
                </mrow>
            </mfrac>
```

```
            <mo>+</mo>
            <mi>...</mi>
        </mrow>
    </mrow>
</math>
```

De l'arbre MathML à l'arbre Tex

Jusque là, vous devez vous dire, 'c'est limpide comme de l'eau de roche'. Seulement, il fallait bien qu'il y ait un problème tôt ou tard. Or, il intervient ici. En effet, pour passer de l'arbre TEX à l'arbre MathML, on aurait pu se dire: "il suffit de raisonner dans le sens inverse et ça va se faire tout seul". Mais malheureusement ce n'est pas le cas et c'est à ce moment que l'on peut se demander si nous n'aurions pas du construire notre DOM TEX d'une autre manière, plus similaire que la DOM MathML de présentation.

Je vous propose de voir la difficulté au travers d'un exemple. On veut passer de la représentation MathML de 1+2+3 à sa représentation TEX. Voici donc l'arbre de départ et celui que l'on devrait obtenir.

mathML.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<math>
    <mrow>
      <cn>1</cn>
      <mo>+</mo>
      <cn>2</cn>
      <mo>+</mo>
      <cn>3</cn>
    </mrow>
</math>
```

mathML.out

```
<?xml version="1.0" encoding="UTF-8"?>
<TEX>
    <PLUS>
        <PLUS>
            <NUMBER>1</NUMBER>
            <NUMBER>2</NUMBER>
        </PLUS>
        <NUMBER>3</NUMBER>
    </PLUS>
</TEX>
```

On constate bien qu'il faut alors passer d'une écriture en ligne à une écriture de regroupement de deux fois deux termes, un premier regroupement pour l'ensemble {1,2} et 3 et un autre regroupement pour 1 et 2. De plus, on voit bien que la première balise <PLUS> prend en fils un groupe (qui peut à priori être de n'importe quel type <EXPRESSION>, <FUNCTION>,…, ce qui complique encore plus notre problème).

Cela veut dire que chaque fois qu'on rencontre autre chose qu'un terme +, il faudrait tester si le terme d'après est encore ce symbole et ainsi de suite jusqu'au dernier. Mais ça ne devient plus vrai si on a à représenter: 1+2=x !!

En fait, ce devient vite vraiment très compliqué voire irréalisable à gérer d'autant plus que XSLT n'autorise pas la création de balises fermantes autre part que symétriquement dans le code.(ex.: …<PLUS><xsl:apply-templates …/></xsl:if></PLUS>…).

Le programme mathML.xsl (donné en annexes) propose une approche du problème qui fonctionne pour 1+2 ou pour x=1 mais qui est totalement fausse pour 1+2+3.

Le parser MathML

« J'aime qu'à mes desseins la fortune s'oppose: Car la peine de vaincre en accroît le plaisir »

JeanBertaut

Afin de pouvoir traiter efficacement les documents MathML, nous nous devons dans un premier temps de produire un parser spécifique. En effet, comme nous l'avons dit, MathML possède des règles de construction qui sont transparentes aux processeurs XML génériques. Et nous savons aussi que l'utilisation de la DOM spécifique à MathML nous permet de contrôler le respect de ces règles. Par conséquent, l'implémentation de cette DOM est indispensable.

Néanmoins, MathML reste une application XML. Il doit donc en respecter la syntaxe et la grammaire, ce que peut vérifier un parser XML quelconque. Nous pouvons donc aisément utiliser l'un d'entre eux pour implémenter notre parser MathML.

Nous avons recensé les parsers XML existants, et nous nous sommes attardés plus longuement sur deux d'entre eux pour notre application : Xerces et Crismons.

Xerces

Origines et Caractéristiques

Xerces est un parser XML développé par le groupe *XML Apache Project.* Le but de ce groupe, fondée en 1999 est :

- De fournir des solutions basées sur XML à des problèmes classiques, en open-source,
- De fournir des informations sur les standards XML (notamment ceux du W3C), pour aider les développeurs désireux de les implémenter,
- De faire le lien entre les applications XML et le serveur Apache.

Ce groupe travail donc actuellement sur plusieurs projets différents, mais ayant tous un rapport étroit avec XML :

- Le parser Xerces, développé parallèlement en Java, C++, et Perl,
- Le « transformeur » Xalan, un processeur basé sur les feuilles de styles XSLT, développé en Java et C++
- Cocoon, développé en Java, un outil de publication sur Internet basé sur XML,
- FOP, un éditeur d'objets XSL, développé en Java,

- Un serveur Web dynamique en JavaScript, Xang,
- SOAP, « Simple Object Access Protocol »,
- Batik, un éditeur SVG (Scalable Vector Graphics),
- Crimson, un autre Java-parser XML, basé sur le parser *Sun project X*

Comme nous l'avons vu, l'un des objectifs du groupe XML Apache et de respecter le plus possible les standards et les recommandations. Par conséquent, les développeurs de Xerces se sont imposés comme règles l'acceptation de l'ensembles des standards relatifs à XML, à savoir XML 1.0, les espaces de noms, les DTD, et depuis peu les XML schemas. Il est en de même pour les « modèles » de programmation : Xerces supporte la DOM 1 et la DOM 2, mais également SAX 1.0 et 2.0. Le dernier objectif des développeurs de Xerces, et non des moindres, et de rendre l'application la plus performante possible, ce qui comme nous le verrons nous a posé quelques problèmes… Cette caractéristique est très importante, car elle guide totalement la logique des choix d'implémentation pris par les développeurs.

Lorsque nous avons débuté ce projet, Xerces en était à la version 1.3.0. Au moment de la rédaction de ce rapport, la version 1.4.0 était disponible. De même, le XML apache Project planchait sur une nouvelle version de Xerces, basé sur les mêmes objectifs mais avec d'autre choix de développement.

Notre application se basant sur Xerces-j (la version Java de Xerces), nous nous devons d'en comprendre le fonctionnement.

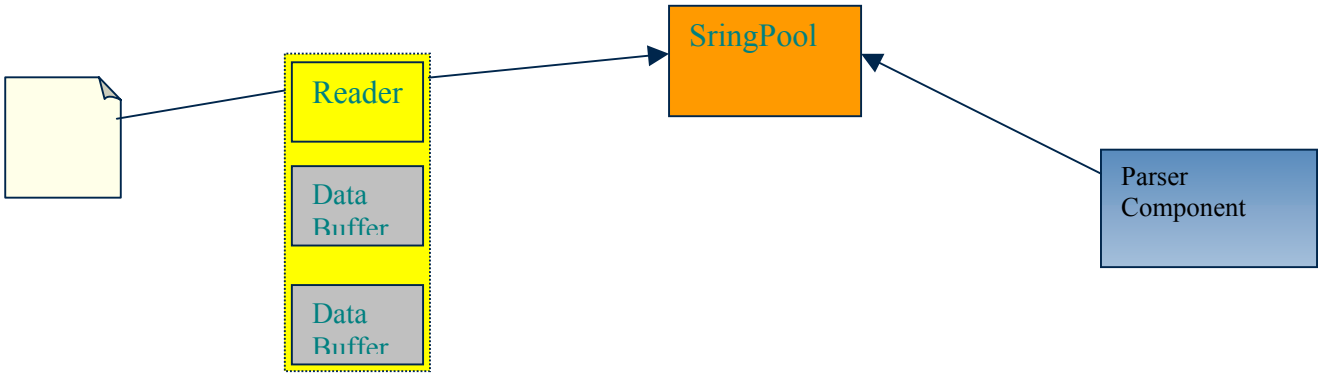
Le Modèle de fonctionnement

Comme nous venons de le dire, Xerces-j est avant tout conçu pour être performant. Par conséquent, tout le modèle de l'analyse du document repose sur cette obligation. Les développeurs ont donc conçu un certain nombre de classe pratique leur permettant de réaliser cet objectif. La plus importante d'entre elle est la classe `StringPool` ; cette classe :

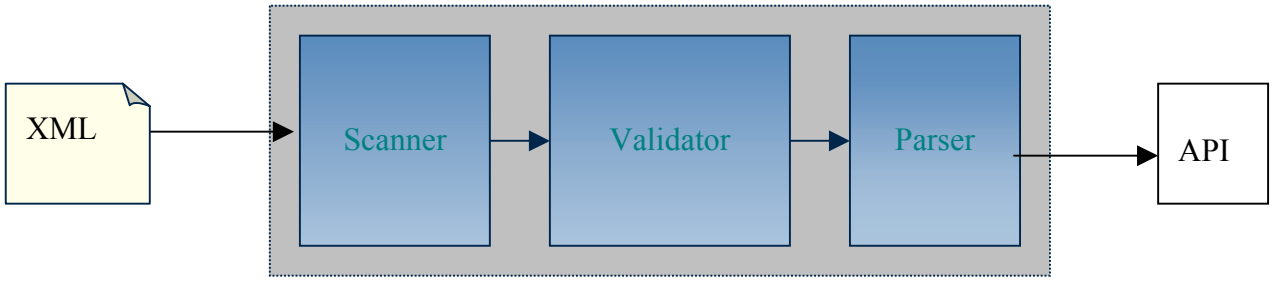
- Possède une table de Hachage contenant les chaînes souvent présentes dans le document,
- Transpose le code en byte dans un tampon (Data Buffer) avant de le transmettre aux composants du parseur (transmission du code différée).

A la suite de ce codage en byte particulier, le parseur définit donc également des méthodes spéciales pour accéder aux différents objets du document, pour les retourner ou pour en modifier le contenu.

Globalement, tout échange en entre les composants du parseur et le document XML passe obligatoirement par la classe `StringPool`. :



De plus le groupe de développement a choisit un modèle linéaire de traitement des données, une sorte de « pipeline ». En effet, afin d'être traiter correctement, un document est en premier lieu lu - on dit que le document est *scanné*, on teste en suite sa *validité*, c'est à dire que l'on vérifie s'il est bien formé, puis le parsage est exécuté, suivant le modèle DOM ou SAX. Le schéma est donc le suivant :



Xerces respecte totalement les spécifications DOM 1 et 2. Par conséquent, l'ensemble des interfaces définies par le W3C sont implémentées, non seulement pour XML mais également pour HTML et WML.

Les problèmes

A partir des précisions données ci-dessus, nous voyons bien les problèmes que nous pouvons rencontrer pour implémenter notre parser MathML.

Le premier réside dans ce choix de reporter le passage de données par tampon. En effet, l'ensemble des composants de Xerces font référence à la classe `StringPool`. Il en résulte que :

- Les méthodes ne peuvent pas accéder directement au données,
- Nous devons passer par `StringPool` à chaque fois que nous désirons récupérer une chaîne de caractère quelconque,
- La gestion de la mémoire s'avère compliquée : ce choix d'implémentation est plus performant mais plus coûteux en ressources, l'utilisateur doit donc se charger lui-même de libérer des ressources…

Par conséquent, lorsque nous avons développé notre parser MathML à partir de Xerces, nous nous sommes rendus compte que nous dépendions trop de ce parser. Il nous fallait dans un premier temps comprendre le bon fonctionnement du parser de base et de son `StringPool`, puis imposer à nos classes d'utiliser elles aussi `StringPool` pour accéder correctement aux données, auquel cas notre application analyserait efficacement les documents, mais ne pourrait plus en modifier la structure ni en obtenir d'autre information, ce qui n'a aucun intérêt.

De plus, le schéma de traitement exposé précédemment (le pipeline) implique une trop grande dépendance entre les éléments. En effet, le *Validator* reçoit ses informations directement du *Scanner*, et le *Parser* reçoit les siennes directement du *Validator*. D'où l'importance de connaître la forme des données transmises par les uns et les autres. Dans notre cas, nous devenons totalement dépendant des données présentes dans *Validator,* ce qui implique une fois de plus une bonne connaissance de sa structure et de son fonctionnement. Pour peu qu'une interface ou une classe essentielle soit modifiée d'une version à l'autre, notre application ne serait plus compatible avec la nouvelle version, ce qui nous interdirait de profiter des nouvelles fonctionnalités. Une fois notre parser finalisé, nous ne pouvons quasiment plus le faire évoluer…

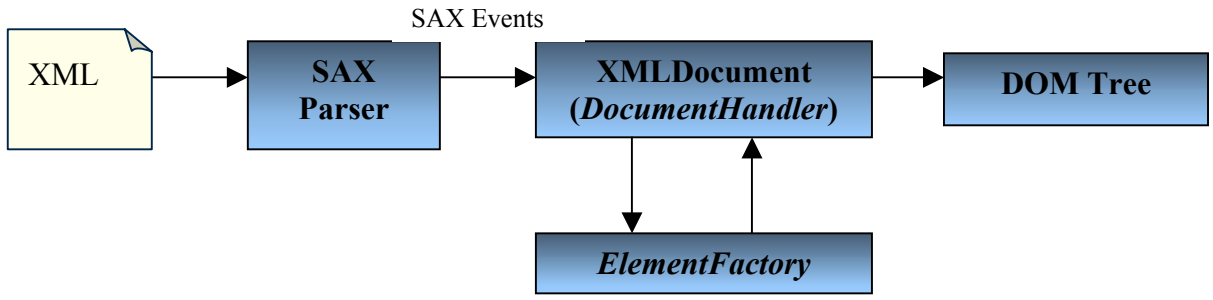
Xerces est véritablement un parseur performant et complet, mais malheureusement trop complexe. De jeunes développeurs comme nous, ne maîtrisant pas encore assez ces technologies, ne peuvent pas développer de manière efficiente une nouvelle application à partir de Xerces. Au sein du groupe XML Apache Project, beaucoup remettent en cause actuellement la complexité du Xerces actuel et travaille sur la conception et le développement de Xerces2. Xerces2 se fixe les mêmes objectifs que Xerces, mais ne met plus la performance au centre des débats.

Crimson

Crimson est un parser XML Java pour XML 1.0, basé sur les API suivantes :

- Java API for XML Processing (JAXP) 1.1: JAXP est une API permettant aux applications d'accéder aux données et aux documents XML indépendamment du parser choisi.
- SAX 2.0
- SAX2 Extensions version 1.0
- DOM level 2.

Le schéma de fonctionnement pour le parsage DOM est le suivant :



Ce schéma montre bien que SAX et DOM ne sont pas deux modes de traitement contradictoires, puisque l'on utilise la gestion événementielle pour construire le `Document`. Lorsque le `DocumentHandler` intercepte un évènement, il crée le nœud correspondant, soit directement, soit à travers `ElementFactory` pour les éléments. Le nœud ainsi créé est alors rajouté à l'arbre DOM.

Par conséquent, pour créer une DOM MathML, il suffit d'avoir un `DocumentHandler` (dérivant de XMLDocument) spécifique qui utilise une `ElementFactory` créant les éléments MathML.

Nous avons rencontré un problème pour la création des éléments avec Namespaces. En effet, dans `XMLDocument`, la création se fait sans tenir de `l'ElementFactory`. Il ne faudra donc pas oublier de surcharger la méthode `createElementNS()` pour que `l'ElementFactory` soit utilisé lors de la construction d'éléments MathML avec Namespaces.

La DOM MathML

Nous allons dans cette section présenter plus en détail certains éléments de la DOM MathML, et quelques choix d'implémentation.

L'interface MathMLDocument

Cette interface dérive de l'interface `Document` classique de la DOM.

Il est à noter que cette interface ne représente pas forcément un document MathML, mais le document contenant les éléments MathML, l' « arbre » MathML. En effet, à terme, le but du W3C et de permettre à XHTML de contenir tout type de balises, notamment des balises MathML. Il ne faut donc jamais perdre de vue qu'un `MathMLDocument` peut être inclus dans un `Document`.

`MathMLDocument` hérite de `Document` la méthode `getDocumentElement`(). La méthode héritée est censée retournée l'élément racine du document. Dans le cas de `MathMLDocument`, elle retournera le nœud racine MathML le plus haut dans l'arbre, à savoir l'élément `math`.

L'interface MathMLElement

Toutes les interfaces des éléments dérivent de cette interface, qui hérite elle-même de l'interface `Element`.

Elle fournit des méthodes pratiques permettant d'accéder plus rapidement aux attributs communs à tout les éléments MathML, à savoir :

- `class`,
- `style`,
- `id`,
- `xref`,
- `href`.

Ce genre de méthode est très courant dans l'ensemble de la DOM, elle permet d'accéder de façon plus lisible aux attributs, et d'éviter toute ambiguïté. Par exemple, il est plus commode de récupérer l'attribut `class` par `getClassName()` que par `getAttribute("class")`.

Cette interface définit également la méthode `getOwnerMathElement()`, qui retourne l'ancêtre math le plus proche de cet élément. Cela peut-être le nœud racine si nous nous trouvons dans un document MathML. Cette méthode renvoie `null` si nous sommes à la racine.

L'interface MathMLContainer

Cette interface abstraite définit des méthodes utiles à tous les éléments qui peuvent en contenir d'autres, à savoir les constructeurs de notation, (le layout schemata) et la majeure partie des balises de contenu. Cette interface ne peut être implémentée directement. Seules les interfaces `MathMLPresentationContainer`, `MathMLContentContainer` et `MathMLMathElement` peuvent être implémentées.

Cette interface définit deux types d'éléments fils :

- Les arguments,
- Les déclarations.

En effet, MathML oppose les fils arguments aux éléments declare et aux quantifieurs. (par exemple, les limites, les exposants, etc.). Concrètement, bien que cela soit définit dans cette interface, cela ne s'applique pas aux constructeurs de notation.

La limite entre les arguments et les déclarations est très subjective. Pour ma part, je considère que les bornes d'une intégrales sont aussi importantes que la fonction que l'on intègre, mais cela n'a pas l'air d'être le cas du Mathematical Working Group du W3C qui les cite souvent en exemple pour définir ce qu'est une déclaration. Pour l'instant, les seules balises qui rentrent sous la catégorie « déclaration » sont les balises `bvar`, `condition`, `sep`, `degree`, `lowlimit` et `uplimit`). Elles sont citées régulièrement dans toute la spécification MathML, nous nous contenterons donc de nous limiter à celles-ci.

En ce qui concerne les arguments, l'interface décrit les méthodes suivantes :

- `getNArguments()` : renvoie le nombre d'arguments de l'élément,
- `getArguments()` : renvoie la liste des arguments,
- `getArgument(int index)` : renvoie le index-ième argument,
- `SetArgument(MathMLElement newArgument, int index)` : remplace le indx-ième argument par `newArgument`,
- `insertArgument(MathMLElement newArgument, int index)` : insert `newArgument` avant le indx-ième argument,
- `deleteArgument(int index)` : supprime le index-ième argument sans valeur en retour,
- `remove(int index)` : supprime le index-ième argument et le retourne.

L'interface fournit exactement les mêmes méthodes pour les déclarations.

Nous présentons ces méthodes car l'ensemble des interfaces fournissent le même type de fonctions pour l'ensemble de ses éléments fils. Elles permettent toujours ces opérations : accéder à tous les fils, accéder à un fils en particulier, le modifier, le supprimer en le retournant ou non, insérer un autre objet avant. Dans la plupart des cas, nous trouverons aussi la méthode « `append` », qui permet d'ajouter un fils en fin de liste. D'ailleurs, si vous jetez un œil à la DOM classique, vous remarquerez que ces méthodes y sont déjà présentes.

MathMLMathElement

Cette interface hérite de `MathMLElement` et de `MathMLContainer`. Elle possède donc l'ensemble des méthodes exposées ci-dessus.

Cette interface peut être très importante puisqu'elle peut servir de relais entre les objets de la DOM consacrés aux documents (`Document`, `MathMLDocument`, etc.), et les Objets DOM éléments classiques qui sont ces fils. L'implémentation de cette interface devrait contenir un ensemble de méthodes qui lui permettront de réagir spécifiquement lorsque nos balises MathML appartiendront à un document XHTML. Pour le moment, ceci n'est pas d'actualité, puisque nous nous intéressons seulement au documents MathML complets.

A priori, il n'y a aucune raison pour qu'un élément `math` contienne directement des fils déclarations, cela semble tomber sous le sens. D'ailleurs, il n'accepte aucun des éléments `bvar`, `condition`, `sep`, `degree`, `lowlimit` et `uplimit` comme fils. Par conséquent, les méthodes en relation avec les déclarations renvoient toutes `null`, et les méthodes relatives aux arguments traitent tous les fils du nœud `math`.

MathMLPresentationContainer

L'interface `MathMLPresentationContainer` dérive de `MathMLContainer` et de `MathMLPresentationElement` que nous verrons plus tard. Cette interface ne définit pas de méthodes supplémentaires.

Comme nous l'avons dit précédemment, les constructeurs de notation ne possèdent pas de fils de type « déclaration ». par conséquent ils ne traitent que les arguments.

MathMLContentContainer

L'interface `MathMLContentContainer` dérive de `MathMLContainer` et de `MathMLContentElement` que nous verrons plus tard. Cette interface ne définit pas de méthodes supplémentaires.

L'implémentation de cette interface est là seule à ne pas retourner nécessairement une liste nulle lorsque l'on appelle la méthode `getDeclarations()`. D'ailleurs, cette distinction entre arguments et déclarations ne vaut (pour l'instant, les spécifications pouvant toujours évoluées) que pour les éléments de contenu. Comme nous l'avons dit, les seules balises rentrant sous la catégorie déclarations sont les balises `bvar, condition, sep, degree, lowlimit` et `uplimit.` Par conséquent, pour rendre les choses plus faciles à traiter, nous définissons dans la classe `MathMLContentContainerImpl` un vecteur `DeclarationName` contenant essentiellement ces six termes. Pour chaque nœud fils, nous regardons si son nom appartient à ce vecteur : si cela est le cas il s'agit d'un fils de type déclaration, sinon il s'agit d'un nœud de type argument.

L'interface MathMLPresentationElement

Comme son nom l'indique, cette interface, qui dérive logiquement de `MathMLElement`, sert de base à toutes les interfaces d'éléments de présentation. Cette interface ne définit pour l'instant aucune méthode. Cependant, il est bon qu'une interface, même vide, existe pour regrouper l'ensemble des éléments de présentation, car si des fonctionnalités différentes entre les balises de présentation et les balises de contenu venaient à apparaître, il serait plus commode de les introduire par cette interface.

On distingue, dans la DOM MathML, quatre types d'éléments de présentation :

- **Les feuilles.** Ces éléments n'ont aucun fils, y compris de fils de type `Text`. Il s'agit des balises `mglyph` et `mspace`. Ces éléments ne présentent aucune caractéristique particulière. Leur interface réciproque ne fournit que des méthodes permettant d'accéder directement à leurs attributs.

- **Les symboles.** Il s'agit des opérateurs (`mo`), des nombres (`mn`), des identificateurs (`mi`) et du texte (`mtext`).

- **Les « containers ».** Ces éléments peuvent contenir un nombre arbitraire de fils. Il s'agit des éléments `mrow`, `merror`, `mphantom`, `mstyle`, `mpadded` et `mfenced`.

- **Les constructeurs de notation.** Ce sont bien évidemment les plus importants et les plus nombreux. On y retrouve `mfrac, msqrt, msub, msup`…

L'interface MathMLPresentationToken

Cette interface dérive de `MathMLPresentationElement`. Elle fournit des accès aux attributs spécifiques de ces symboles. L'accès au corps de ces éléments se fait via la méthode `getNodeValue` héritée de l'interface `Node`.

Les attributs auxquels nous pouvons accéder directement via cette interface sont :

- `mathvariant` : par défaut, la valeur de cette attribut est `normal`, mais l'ensembles des valeurs possibles est `normal, bold, italic, bold-italic, double-struck, bold-fraktur, bold-script, fraktur, sans-serif, bold-sans-serif-italic, sans-serif-bold-italic`, et `monospace`.

- `mathsize`: par défaut, la valeur de cet attribut est héritée du nœud père, mais l'ensembles des valeurs possibles est `small, normal` ou `big` ou de la forme `nb v-unit`

- `mathfamily`: une chaîne de caractère de la forme `css-fontfamily`. Par défaut, la valeur de cet attribut est héritée du nœud père. Cet attribut n'est pas spécialement requis.

- `mathcolor`: de la forme #rrggbb. Par défaut, la valeur de cet attribut est héritée du nœud père.

- `mathbackground`: de la forme #rrggbb. Par défaut, la valeur de cet attribut est héritée du nœud père.

Ces attributs définissent clairement une représentation visuelle des symboles. Lorsque les applications supportent également les feuilles de styles CSS, ces attributs peuvent prendre la valeur des styles CSS prédéfinis soit par l'auteur du document soit par le lecteur. En ce qui nous concerne, nous ne nous soucions nullement des feuilles de styles CSS, l'interpréteur ne renvoie pas d'erreur lors du parsage lorsqu'il rencontre un élément CSS (i.e. cela ne remet pas en cause la validité du document), mais ces éléments ne pourront pas intervenir lors du rendu du document.

Cette interface fournit aussi également une méthode `getContents()`, qui renvoie l'ensemble des nœuds fils de l'élément. Généralement, il s'agit seulement du texte contenu entre les balises, sur lequel on appliquera les règles de présentation définies par les attributs. Mais l'élément peut également contenir des fils `mglyph` ou `mspace`.

Les éléments `mi`, `mo`, et `mtext` supportent directement cette interface. Les éléments `mo` et `ms` possédants des attributs supplémentaires à ceux exposés ci dessus, n'implémentent pas directement cette interface. Ils implémentent respectivement les classes `MathMLOperatorElement` et MathML`StringElement`.

Les interfaces « Presentation Schemata »

Ces interfaces regroupent l'ensemble des constructeurs de notation.

Comme nous l'avons déjà vu, MathML rajoute, par rapport à XML, des règles d'ordre au niveau des nœuds fils. Ceci prend enfin tout son sens ici. En effet, les interfaces associées à ces constructeurs fournissent généralement un certain nombre de fonctions pratiques permettant d'accéder directement au n-ième fils, censé représenter un argument particulier. Si l'on reprend l'exemple de la fraction, la méthode `setNumerator(newNumerator)` de l'interface `MathMLFractionElement` n'est rien d'autre que `setChild(newNumerator, getChild().item(2))`, mais en plus lisible. Par conséquent, si ces règles d'ordre ne sont pas respecter, l'accès aux éléments ne renverra pas d'erreur, mais toute la sémantique aura été perdu.

Ces interfaces dérivent toutes directement de `MathMLPresentationElement`, excepté `MathMLTableCellElement` qui dérive de `MathMLPresentationContainer`. En effet, tous ces constructeurs acceptent un nombre fini d'arguments, alors que les containers sont censés posséder un nombre arbitraire de fils. Les interfaces qui sont reprises dans cette catégorie sont (avec entre parenthèses la balise correspondante) :

- `MathMLFractionElement (maction)`

- `MathMLRadicalElement (msqrt)`

- MathMLScriptElement (msub, msup, msubsup)
- MathMLUnderOverElement (munder, mover, munderover)
- MathMLTableElement (mtable)
- MathMLTableRowElement (mtr)
    - MathMLLabeledRowElement (mlabeledtr)
- MathMLTableCellElement (mtd)
- MathMLAlignGroupElement (maligngroup)
- MathMLAlignMarkElement (malignmark)

L'interface MathMLScriptElement

Cette interface dérive de `MathMLPresentationElement` et représente les éléments `msub, msup,` et `msubsup`. Elle fournit des méthodes permettant d'accéder et de retourner les indices, les exposant, et la base des scripts.

Le fait que cette interface regroupe ces trois balises peut poser un problème, car qu'il s'agisse d'un indice, d'un exposant ou des deux à la fois, le nombre de fils de l'élément n'est pas le même, et ils n'ont plus la même signification. Les méthodes deviennent donc légèrement plus compliquées que pour les interfaces `MathMLFractionElement` ou `MathMLRadicalElement`. Néanmoins, cela n'est pas beaucoup plus compliqué, car la syntaxe de chaque élément est très figée ; il nous suffit donc de contrôler le nom de la balise afin de choisir le fils à renvoyer ou à modifier.

Les syntaxes de ces balises sont les suivantes :

Syntaxe des balises `msup`, `msub` et `msubsup`

```
<msub> base subscript </msub>

<msup> base superscript </msup>

<msubsup> base superscript </msubsup>
```

nous pouvons prendre l'exemple $\int_0^1 e^x dx$ pour voir la syntaxe correcte d'un élément `mubsup` :

exemple d'une intégrale :

```
<mrow>
   <msubsup>
      <mo> &int; </mo>
      <mn> 0 </mn>
      <mn> 1 </mn>
   </msubsup>
   <mrow>
      <msup>
```

```
            <mi> &ExponentialE; </mi>
            <mi> x </mi>
         </msup>
         <mo> &InvisibleTimes; </mo>
         <mrow>
            <mo> &DifferentialD; </mo>
            <mi> x </mi>
         </mrow>
      </mrow>
</mrow>
```

L'interface MathMLUnderOverElement.

Cette interface dérive directement de `MathMLPresentationElement` et représente les balises `mover`, `munder`, et `munderover`. Ces balises correspondent aux scripts que l'on peut trouver au-dessus et au-dessous de chaque élément, comme une limite, les bornes d'une somme, etc. Elles fonctionnent exactement de la même manière que les balises `msub`, `msup`, et `msubsup` :

structure des balises `munder`, `mover`, et `munderover`

<munder> base underscript </munder>

<mover> base overscript </mover>

<munderover> base underoverscript </munderover>

Par conséquent, leur traitement est identique à celui des balises de script, et l'interface `MathMLUnderOverElement` possède quasiment les mêmes fonctionnalités que l'interface `MathMLScriptsElement`.

L'exemple le plus simple d'utilisation de la balise `munderover` est une somme de la forme $\sum_{i=1}^{n} x^i$. Cet exemple s'écrit en MathML :

Exemple de la balise `munderover` :

```
<mrow>
   <munderover>
      <mo> <mchar name="sum" /> </mo>
      <mrow>
         <mi> i </mi>
         <mo> = </mo>
         <mn> 1 </mn>
      <mrow>
      <mi> n </mo>
   </munderover>
   <mrow>
      <msup>
         <mi> x </mi>
         <mn> i </mn>
      </msup>
   </mrow>
</mrow>
```

L'interface MathMLTableElement

Cette interface dérive de `MathMLPresentationElement` et représente l'élément `mtable`. Cet élément définit les tables et les matrices, pour tout ce qui est notation. Il peut avoir un nombre arbitraire de fils, touts ces fils étant des éléments `mtr` ou `mlabeledtr`.

L'élément `mtable` possède avant tout un ensemble impressionnant d'attributs, concernant l'alignement, l'espacement, etc.. Par conséquent, l'interface correspondante introduit toutes les méthodes permettant d'accéder à cet attribut et de les modifier. De plus, l'interface définit des méthodes permettant :

- De retourner l'ensemble des lignes de la matrice ou de la table,
- D'insérer une ligne vide,
- De manipuler une ligne précise (modifier, supprimer, récupérer, etc.)

Pour une matrice identité 3x3 $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, on obtient le code suivant :

Exemple pour une matrice identité

```
<mrow>
  <mo> ( </mo>
  <mtable>
    <mtr>
      <mtd> <mn>1</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
    </mtr>
    <mtr>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>1</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
    </mtr>
    <mtr>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>1</mn> </mtd>
    </mtr>
  </mtable>
  <mo> ) </mo>
</mrow>
```

On remarquera que les parenthèses sont à l'extérieur de la matrice. Cela donne plus de flexibilité pour le rendu, puisque l'on peut définir ainsi quel type de séparateur (crochets, parenthèses, …) l'on souhaite pour encadrer notre matrice.

Pour faire le tour de la question, les balises `mtr` et `mlabeledtr` sont définies par les interfaces `MathMLTableRowElement` et `MathMLLabeledRowElement`, qui dérive de `MathMLTableRow`. Ces éléments contiennent des éléments `mtd`, supportés par l'interface `MathMLTableCellElement`. Comme `MathMLTableElement`, `MathMLTableRowElement` possède des méthodes permettant d'agir sur les cellules du tableau.

L'interface MathMLContentToken

Cette interface dérive de `MathMLContentElement` et fournit des méthodes communes à tous les éléments extrêmes de l'arbre MathML pour les balises de contenu, les feuilles de l'arbre de contenu en quelque sorte. Cette interface englobe donc les éléments ci, `csymbol` et `cn`. A l'inverse des éléments feuilles des balises de présentation (`mo`, `mi`, `mn`, `mtext`) qui ne pouvait contenir que du texte, ou à la limite une balise `mchar`, les token de contenu peuvent contenir tout type d'élément.. Par conséquent, l'interface correspondante fournit les méthodes `getArgument()` et `setArgument()` permettant d'accéder à la valeur de ces tokens.

Des exemples de balises `ci`, `cn` et `csymbol` sont :

`<cn type="real"> 12345.7 </cn>`
 	
`<cn type="integer"> 12345 </cn>`
 	
`<cn type="integer" base="16"> AB3 </cn>`
 	
`<cn type="rational"> 12342 <sep/> 2342342 </cn>`
 	
`<cn type="complex-cartesian"> 12.3 <sep/> 5 </cn>`
 	
`<cn type="complex-polar"> 2 <sep/> 3.1415 </cn>`
 	
`<cn type="constant">  &pi; </cn>`
	• **12345.7**
	• **12345**
	• **$AB3_{16}$**
	• **12342 / 2342342**
	• **12.3 + 5 i**
	• **Polar(2 , 3.1415)**
	• π

`<ci> x </ci>`
 	
`<ci type="vector"> V </ci>`
 	
`<ci>`
`<msub>`
`<mi>x</mi>`
`<mi>a</mi>`
`</msub>`
`</ci>`
	• x
	• V
	• x_i

`<apply>`
`<csymbol encoding="OpenMath" definitionURL="http://www.openmath.org/cd/BesselFunctions.ocd">`
`<msub><mi>J</mi><mn>0</mn></msub>`
`</csymbol>`
`<ci>y</ci>`
`</apply>`
 	
`<csymbol encoding="text" definitionURL="www.example.org/universalconstants/Boltzmann.htm">`
`k`
`</csymbol>`
	• $J_0(y)$
	• k

L'interface MathMLContentContainer

Cette interface, qui dérive à la fois de `MathMLContentElement`, et de `MathMLContainer`, supporte tous les éléments de contenu qui peuvent contenir un nombre quelconque de fils. Elle est supportée directement par les balises `reln`, `lowlimit`, `degree`, `domainofapplication` et `momentabout`. Les éléments possédant une interface qui dérive de `MathMLContentContainer` sont les éléments `apply`, `fn`, `interval`, condition, `declare`, `bvar`, `set`, `list`, `vector`, `matrix`, et `matrixrow`.

Cette interface définit les « attributs » suivants, c'est à dire qu'elle permet d'accéder à et de modifier :

- L'attribut `condition`, de type `MathMLConditionElement`. Il représente une condition éventuelle sur l'une ou plusieurs des variables de l'élément. Seuls les éléments `apply`, set et `list` acceptent cet élément comme fils. Tout autre élément doit renvoyer une exception si l'on tente d'accéder à cet attribut.

- L'attribut `domainOfApplication`, de type `MathMLElement`. Il représente le domaine d'application de l'élément. Cela peut être par exemple le domaine d'intégration d'une balise `apply` dont le premier fils est l'opérateur d'intégration int. `DomainOfApplication` est un fils de l'élément de contenu, il est donc de type `MathMLElement`.

- L'attribut `opDegree`, de type `MathMLElement`. Il représente le degré de l'expression, par exemple l'ordre de dérivation ou de différentiation si l'on se trouve dans une balise `apply` dont le premier élément est une balise `diff` ou `partialdiff`. Ce fils n'est supporté que par les balises `bvar` et `apply`. Tout autre élément doit renvoyer une exception si l'on tente d'accéder à cet attribut. `OpDegree` est un fils de l'élément de contenu, il est donc de type `MathMLElement`.

- L'attribut `momentAbout`, de type `MathMLElement`. Il représente le moment statistique (d'ordre p par exemple) associé à un moment général. Ce fils n'est supporté que par les éléments `apply` ayant pour premier fils l'élément moment. Tout autre élément doit renvoyer une exception si l'on tente d'accéder à cet attribut. D'autre part, bien que cet attribut soit toujours représenté par une balise `momentabout`, la valeur en retour est de type `MathMLElement`..

Voici quelques exemples d'utilisation de ces éléments :

```
<apply>
  <max/>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply> <and/>
      <apply><gt/><ci> x </ci><cn> 0 </cn></apply>
      <apply><lt/><ci> x </ci><cn> 1 </cn></apply>
    </apply>
  </condition>
  <apply>
    <minus/>
    <ci> x </ci>
    <apply>
      <sin/>
      <ci> x </ci>
    </apply>
  </apply>
</apply>
```

$$\max_x \{x - \sin x \mid 0 < x < 1\}$$

<apply> <int/> <bvar> <ci> x </ci> </bvar> <interval> <ci> a </ci> <ci> b </ci> </interval> <apply><cos/> <ci> x </ci> </apply> </apply>	$\int_a^b \cos x$
<apply><partialdiff/> <bvar><ci> x </ci><degree><ci> m </ci></degree></bvar> <bvar><ci> y </ci><degree><ci> n </ci></degree></bvar> <degree><ci> k </ci></degree> <apply><ci type="fn"> f </ci> <ci> x </ci> <ci> y </ci> </apply> </apply>	$\left(\frac{\partial^k}{\partial x^m \partial y^n}\right) f(x,y)$
<apply> <moment/> <degree> <cn> 3 </cn> </degree> <momentabout> <ci> p </ci> </momentabout> <ci> X </ci> </apply>	$\langle X^3 \rangle$

Comme vous pouvez le constater, ces éléments sont assez particuliers et non génériques à l'ensemble des classes. De plus, la plupart de ces balises peuvent être associées : nous avons décidé dans l'exemple 2 de définir notre domaine d'application comme étant un intervalle, mais nous aurions pu définir une limite supérieure et inférieure, ou encore une condition du type $x \in [a,b]$.

Par conséquent, différents modèles de conception s'opposent : est-il bon d'implémenter directement cette interface pour les elements qui la supporte ? Doit-on choisir de réaliser une implémentation différente de cette interface pour chaque élément ? Doit-on implémenter `MathMLContentContainer` pour regrouper un certain nombre de méthodes facilement assimilables par tous ces éléments ?

Pour notre part, nous avons choisi d'implémenter `MathMLContentContainer` et par conséquent de pouvoir instancier `MathMLContentContainerImpl`. En effet, certaines méthodes n'auront pas besoin d'être redéfinis par la suite comme la méthode `getCondition()` par exemple. De plus, cela peut permettre, lors d'un développement futur, d'ajouter plus facilement des fonctionnalités propres à ce type d'éléments, si le besoin s'en fait sentir. En outre, cette classe implémente les fonctions héritées de `MathMLContainer`, qui seront communes à l'ensemble des éléments et qui n'auront pas besoin d'être redéfinies par la suite. Enfin, l'interface `MathMLContentContainer` permet des accès privilégiés aux éléments fils `bvar`, ce qui est très pratique et

commun à l'ensemble de ces éléments. Nous avons donc implémenté la plus part des fonctions ci-dessus, afin que les balises supportant directement cette interface puisse accéder à ces fonctionnalités.

Mais nous nous rendons compte à présent qu'un ciblage plus poussé aurait été préférable. En effet, plus nous avançons dans la connaissance de MathML, plus nous découvrons ses subtilités et ses particularités, qui nous font penser que la plupart des éléments de contenu nécessiteraient leur propre classe. D'ailleurs, s'il existe un si grand nombre d'éléments de contenu (plus d'une centaine), ce n'est pas pour rien. Cela est bien la preuve que chaque élément possède ses attributs et ses fonctionnalités propres. Par conséquent, il serait bon par la suite de redéfinir des classes `MathMLRelnElementImpl`, `MathMLLowlimitElementImpl`, `MathMLDegreeElementImpl`, et `MathMLDomainofapplicationElementElementImpl`. Il en est de même pour les éléments dont l'interface dérive de `MathMLContentContainer` (comme la balise `apply`) : leur implémentation devrait redéfinir l'ensemble de ces méthodes.

L'interface MathMLDOMImplementation

Cette interface, qui dérive de l'interface `DOMImplementation`, en reprend toutes les fonctionnalités, et y ajoute la méthode `createDocument()`. Cette méthode permet de construire un `MathMLDocument` le plus simple possible, c'est à dire ne contenant qu'un objet `MathMLMathDocument`, sans enfant et sans attributs.

L'interface MathMLDocument

Cette interface hérite directement de l'interface `Document`. Elle ne redéfinit rien par rapport à son ancêtre, si ce n'est la présence de trois nouveaux attributs :

- `Referrer`,
- `Domain`,
- `URI`.

Ces éléments sont seulement utilisés pour la navigation au sein de plusieurs documents XHTML ou MathML liés.

Le Parser MathIT

Choix du langage

Nous n'avons que l'embarras du choix au niveau des langages pour traiter nos fichiers XML. En effet, plusieurs applications existent déjà en Java et C++, mais aussi en Perl ou encore en Python. Cependant, vu la connivence existante entre les créateurs respectifs de Java et de XML, cela est bien dans ce langage que les applications sont les plus pertinentes. De plus, le couplage d'un langage portable et d'un support de données portable met en avant la force de chacun d'entre eux. Nous implémenterons donc notre modèle en Java.

Structure des packages

Le parser MathIT, bâti sur Crimson, comprend quatre packages :

- Le package `mathit.dom.mathml` : ce package comprend l'ensemble des interfaces de la DOM MathML.

- Le package `mathit.parser.mathml` : ce package comprend l'ensemble des implémentations des interfaces DOM MathML.

- Le package `mathit.parser` : ce package ne contient que la classe `MathITElementFactory.` Cette classe contient une table de hashage associant à chaque élément la classe (et donc le constructeur) correspondante, et la méthode `createElementEx()` qui créer un nouvel objet à partir de cette table et du non de l'élément. Cette classe implémente l'interface `org.apache.crimson.tree.ElementFactory`

- Le package `mathit.jaxp.mathml` : ce package comprend :

  - La classe `MathMLDocumentBuilderImpl` qui dérive de la classe `javax.xml.parsers.DocumentBuilder`. Cette classe fournit les mécanismes pour parser un document MathML au sein d'un arbre DOM MathML, représenté par un objet `mathit.dom.mathml.MathMLDocument`. Une instance de cette classe est obtenue par appel de la méthode `newDocumentBuilder()` de la classe `MathMLDocumentBuilderFactoryImpl`. Cette classe gère donc la plus grande partie du parsage, grâce aux méthodes `parse()` avec différentes signatures, et à la méthode `newDocument()` qui crée un nouvel objet `Document` vide.

  - La classe `MathMLDocumentBuilderFactoryImpl` qui dérive de la classe `org.apache.crismon.jaxp.DocumentBuilderFactoryImpl.` Cette classe gère la configuration et la construction du parser. Sa fonction principale est de créer un `DocumentBuilder` qui parsera directement le `Document`.

La classe `MathMLDocumentBuilderImpl` aurait du dérivé directement de `DocumentBuilderImpl`, afin de profiter au maximum de la réutilisabilité de la programmation objet. Cependant, toutes les variables d'initialisation de `DocumentBuilderImpl` étant privées, et aucune interface n'étant définit pour y accéder, la classe dérivée n'aurait pas pu envoyer un `MathMLDocument` avec `l'ElementFactory`. Il nous a donc fallu reconstruire entièrement la classe `MathMLDocumentBuilderImpl`.

Diagramme de classes simplifié

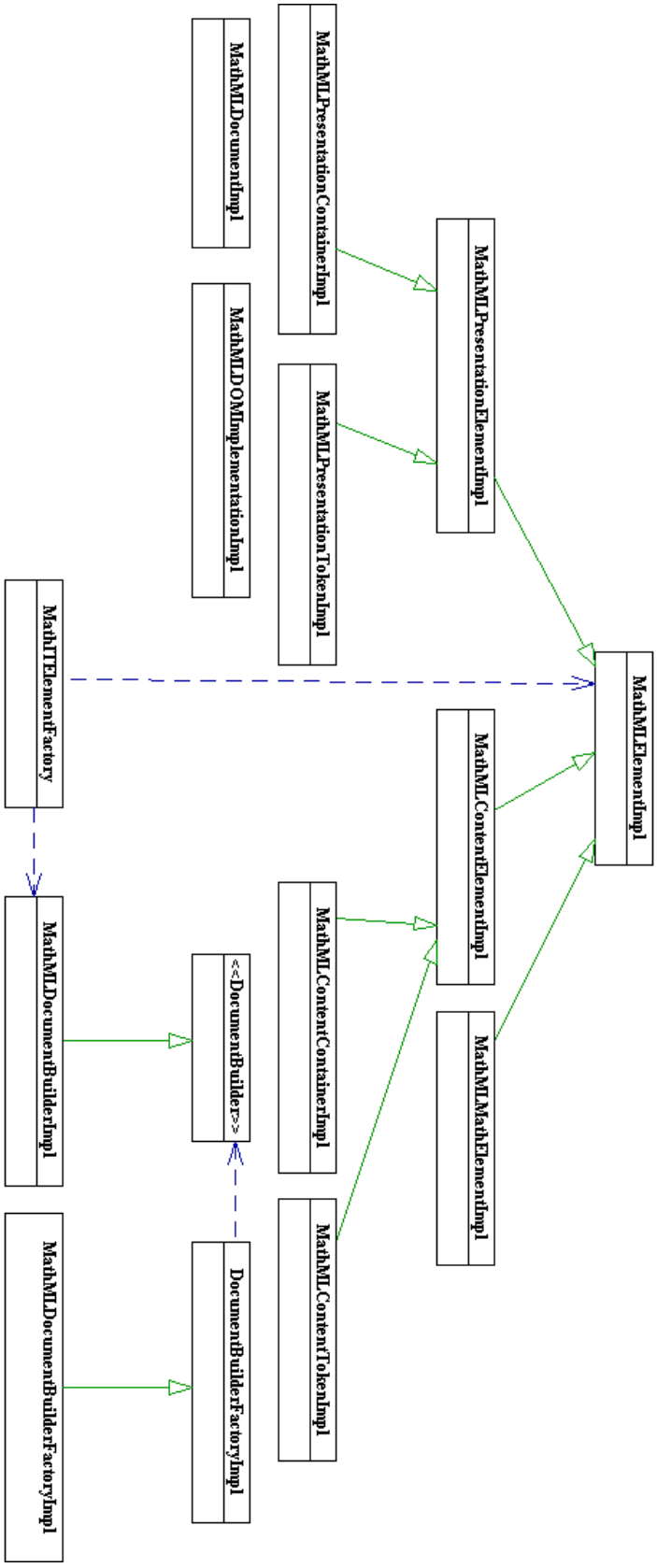
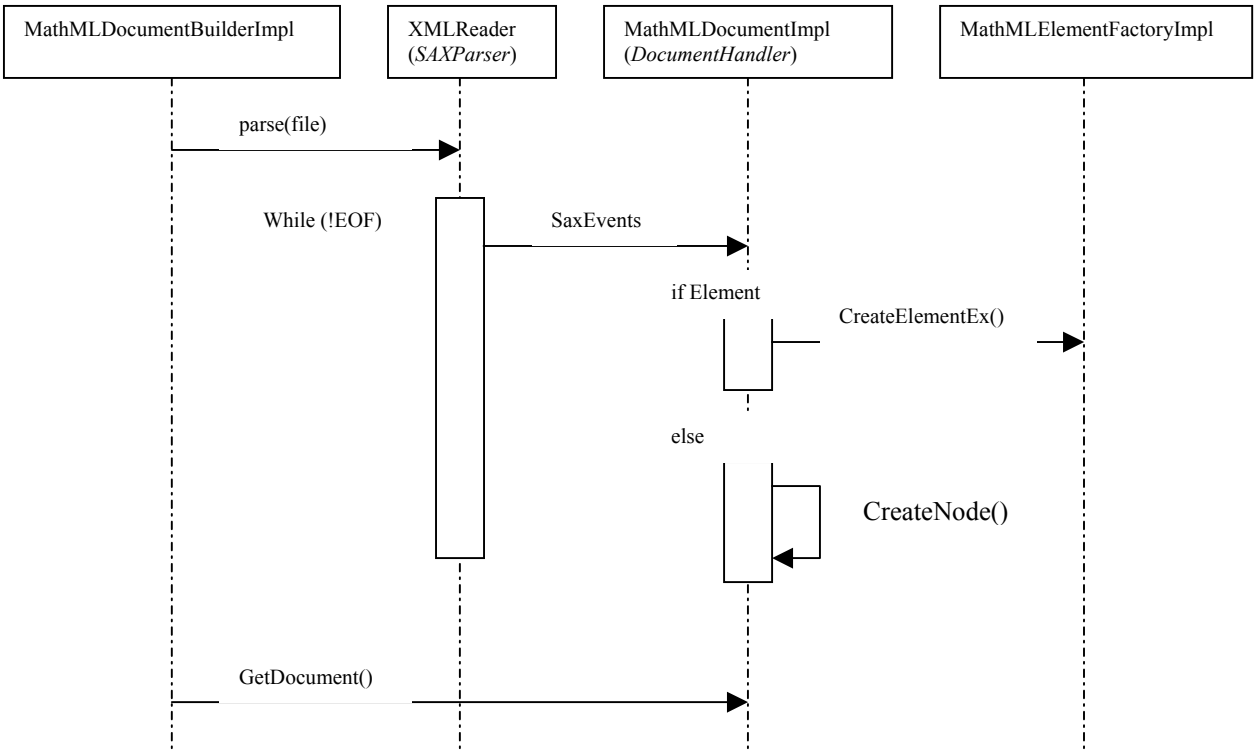


Diagramme de séquence

Le schéma de fonctionnement du parser est globalement le suivant :



Le Moteur de Rendu

« No ties bind so strongly than the links of inheritance », Stephen Jay Gould

Rôle du moteur de rendu

Le projet que nous avons élaboré se compose donc de plusieurs parties. La partie finale concerne l'affichage sur l'écran de l'expression mathématique. A partir de notre arbre MathML, nous allons donc lire les informations contenues dans celui ci afin de traduire de manière ces données sous forme d'affichage représentant clairement le sens de l'expression mathématique. Pour cela, il faut bien évidemment respecter certaines normes graphiques et mathématiques comme le positionnement des différents éléments entre eux. Par exemple, dans le cadre d'une intégrale, tous les éléments n'ont pas le même rôle et on devra donc marquer ces différences en affichant les différents composants de l'expression comme on le ferait à la main sur une feuille de papier. Donc pour cet exemple, il faudra mettre les bornes de l'intégrale à leurs places respectives et avec une taille cohérente avec l'ensemble de l'expression.

Le moteur de rendu devra donc permettre de retranscrire le plus fidèlement possible une expression mathématique telle qu'on peut la voir sur un document LaTex ou bien sur l'éditeur d'équation mathématique de Word. Ce moteur de rendu est donc la partie finale du projet qui permet de constater le résultat.

Ce moteur de rendu a donc été développé en Java dans la continuité des autres parties du projet. Cela permet non seulement de garantir une portabilité plus facile mais aussi une intégration plus aisée. Il consistera en une applet lancée dans le navigateur courant de l'utilisateur qui ne servira que lorsque l'on rencontrera des balises MathML. Les autres éléments déjà reconnus parfaitement par le navigateur (tel que le texte) seront laissés à la charge de ce dernier.

Il devra être relativement rapide à charger afin de ne pas ralentir la navigation et être accessible pour les personnes possédant des connexion Internet lentes.

Dans un premier temps nous avons cherché sur les différents groupes de travail sur MathML pour voir si certains proposaient déjà un moteur de rendu. Cela existait bien mais ils étaient développés par IBM ou d'autres grands groupes qui en proposaient mais de manière payante et ils semblaient assez volumineux. IE et Netscape devaient aussi inclure un plug-in MathML dans leur dernières versions mais il semble qu'ils aient abandonné cette idée (cela prendrait trop de place sur le navigateur). Il nous fallait donc développer notre propre moteur de rendu.

Nous allons donc expliquer les différentes phases de notre conception après avoir introduit les différents standards à respecter.

La norme Unicode

La naissance et l'utilité du standard Unicode

Fondamentalement, les ordinateurs ne comprennent que les nombres. Ils codent les lettres et autres caractères sous formes de nombres. Avant l'invention d'Unicode, des centaines de systèmes de codage de caractères ont été créés. Pas un seul d'entre eux n'était satisfaisant : par exemple, l'Union Européenne a besoin de plusieurs systèmes de codage pour couvrir toutes ses langues d'usage. Même pour une seule langue comme le français, aucun système de codage ne couvrait toutes les lettres, les signes de ponctuation et les symboles techniques en usage courant.

Ces systèmes de codage sont souvent incompatibles entre eux. Ainsi, deux systèmes peuvent utiliser le même nombre pour deux caractères différents ou utiliser différents nombres pour le même caractère. Les ordinateurs, et plus particulièrement les serveurs, doivent supporter plusieurs systèmes de codage de caractères, ce qui crée un risque de corruption des données à chaque transition.

La norme Unicode est donc une solution à tous ces problèmes.

Unicode spécifie un numéro unique pour chaque caractère, quelle que soit la plate-forme, quel que soit le logiciel, quelle que soit la langue.

Le standard Unicode a été choisi par des pionniers technologiques tels que Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys et bien d'autres. Unicode est exigé par de nombreux standards récents tels que XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, etc. Le développement d'Unicode est synchronisé avec celui de la norme ISO/CÉI 10646, la version 3.0 d'Unicode est identique code pour code avec l'ISO/CÉI 10646:2000 mais contient de nombreux éléments supplémentaires d'implantation. Unicode est utilisé dans de nombreux systèmes d'exploitation, dans tous les navigateurs récents, et dans de nombreux autres produits. L'apparition du standard Unicode, ainsi que la disponibilité d'outils le gérant, sont parmi les faits les plus marquants de la globalisation récente du développement logiciel.

L'incorporation d'Unicode dans les applications client-serveur, les applications distribuées et les sites Internet permet une simplification de l'architecture et une réduction des coûts par rapport à l'utilisation des systèmes de codage traditionnels. Grâce à Unicode, un seul logiciel ou site Internet peut satisfaire simultanément et sans modification les demandes de plusieurs plate-formes, langues et pays. Unicode permet aussi à des logiciels de provenance variée d'échanger des caractères sans pertes de données.

Le consortium Unicode

Le Consortium Unicode est une organisation sans but lucratif créée en 1991, ayant pour mission de développer, d'étendre et de promouvoir Unicode. Les membres du consortium sont issus d'un large éventail d'entreprises et d'organisations dans le domaine de l'informatique et des technologies de l'information. Le consortium est financé uniquement par les cotisations de ses membres. L'adhésion au consortium Unicode est ouverte à toutes les organisations et individus qui soutiennent le standard Unicode et souhaitent favoriser son extension et son utilisation.

Présentation plus complète d'Unicode

Unicode propose un moyen performant d'encoder du texte en plusieurs langues et apporte une facilité d'échange de fichiers textes écrits dans des langues différentes, et compréhensibles. Cela simplifie grandement le travail des personnes travaillant dans des domaines ou l'échange d'informations doit se faire de manière efficaces et sans pertes de données. Les mathématiciens, ingénieurs et techniciens qui utilisent régulièrement des symboles mathématiques et d'autres caractères techniques vont aussi apprécier ce standard.

La conception d'Unicode est basée sur la simplicité et la logique de l'ASCII, mais va bien au delà de la capacité limitée de ASCII qui n'encode que l'alphabet Latin. LE standard Unicode fournit la capacité d'encoder tous les caractères utilisés dans les langages écrits du monde entier. Unicode assigne à chaque caractère une valeur numérique unique et un nom.

Le but premier etait d'utiliser un encodage sur 16-bit qui fournit des codes pour plus de 65 000 caractères. Ce chiffre de 65 000est suffisant pour encoder la plupart des milliers de caractères utilisés dabs les langues principales mais Unicode supporte maintenant 3 formes d'encodage qui permet de coder plus d'un million de caractères, ce qui est suffisant pour décrire tous les caractères de langages passés ou présents, ainsi que les systèmes de notations communs (telles que les symboles mathématiques, les flèches…)

Ensemble des caractères inclus dans Unicode

Unicode définit les codes utilisés dans les langages écrits tels que les alphabets européens, les alphabets d'Asie (tels que les katakana japonais, le thaïlandais), d'Europe de l'Est (russe)…

Il fournit aussi les symboles de ponctuation, les diacritiques, les symboles mathématiques, les symboles techniques, les flèches… Le diacritiques sont des marques modifiant un caractère (tel que le ~) qui sont utilisés avec les caractères de base pour encoder les lettres accentuées ou vocalisées (comme le ñ). Dans la version 3.0 du standard, on dénombre plus de 49 000caractères pour les alphabets du mondes entiers (incluant les idéogrammes) regroupés dans l'ensemble appelé Basic Multilingual Plane (BMP).

Il y a environ 8000 codes inusités pour les extensions futures du BMP (dans le cas de l'apparition ou de la découverte d'un nouveau langage) ; de plus on a 900 000 codes disponibles. On compte ajouter 46 000 caractères dans les versions futures.

Il existe aussi des codes réservés à l'usage privé afin que les utilisateurs et vendeurs puissent assigner ces codes à leur propres caractères ou symboles pour usage interne ou bien les utiliser avec des fontes différentes. On en dénombre 6 400 mais on peut on rajouter 130 000 (même si l'utilité n'est pas prouvée).

Les formes d'encodage

Le standard d'encodage ne fournit pas seulement une identité à chaque caractère et sa valeur numérique, mais aussi comment cette valeur est représentée en bits. Unicode définit 3 formes d'encodage qui permettent de transmettre une même donnée en un bit, un format de mot ou de double mot (c.a.d avec 8, 16 ou 32 bits par code) avec la possibilité de passer de l'un à l'autre facilement.

- UTF-8 est assez utilisée pour le HTML et les protocoles similaires. Il transforme tous les caractères unicodes en un code de longueur variable de bits. Il présente l'avantage de donner la même valeur en bits que pour le code ASCII et que les caractères unicodes transformes en UTF-8 peuvent être utilisés avec une bonne part des programmes existant sans nécessiter de réécriture excessive.

- UTF-16 est utilisée dans des environnements qui nécessitent d'équilibrer un accès au caractère efficace avec une économie en terme de taille. Il est relativement compact et tous les caractères principaux tiennent dans un code 16 bits et les autres peuvent être accessible en utilisant des paires de code sur 16 bits.

- UTF-32 est utilisé quand l'espace mémoire n'est pas un souci mais ou un accès en taille fixe, sur un code simple est désiré. Chaque caractère est encodé sur un simple code 32 bits.

Bases du concept :

Afin de coder, traiter et interpréter efficacement du texte, un ensemble de caractères doit être universel, efficace, uniforme et sans ambiguïté.

- Définition des éléments d'un texte : les langages écrits sont constitués d'éléments textuel utilisées pour créer des mots et des phrases. Ces éléments peuvent être des lettre classiques ou bien des caractères utilisés en Japonais qui représentent des syllabes ou des idéogrammes tels que ceux utilisés en Chinois qui représente des mots entiers ou des concepts. La définition des éléments de texte varie souvent en fonction de la procédure de lecture d'un texte. Ainsi, dans l'espagnol ancien, « ll »compte comme un seul élément alors que dans la langue actuelle cela représente deux éléments différents. Pour éviter de tels problèmes Unicode définit des éléments de codes (appelés caractères). Un élément de code correspond généralement à l'élément le plus couramment utilisé. Donc dans l'exemple ; cet élément de code sera « l » et non « ll ». L'association de ces deux « l » sera laissée à la charge du lecteur de texte. On peut aussi citer les majuscules et minuscules qui sont deux éléments codes différents.

- Interpréter les caractères et l'affichage de Glyphes : La différence entre identifier un code et l'afficher sur l'écran ou sur papier est cruciale pour comprendre le rôle d'Unicode dans le traitement du texte. Unicode sert juste à donner un nom, un identifiant à un code. Il ne s'occupe pas de son affichage. Cette tache est laissée au Fontes et Glyphes qui seront définies par la suite. Unicode ne définit pas les images glyphes pouvant représenter ces codes. Unicode définit de quelle manière les caractères sont interprétés, et non comment les Glyphes sont affichés. C'est la machine ou le software qui seront responsable de l'apparence du caractère sur l'écran. Unicode ne définit pas la taille, la forme ni l'orientation des caractères sur l'écran.

- Création de caractères composites : les éléments de textes peuvent être composés d'une séquence ce caractères mais une fois représentés, ils sont affichés ensemble. Par exemple le « â » est un caractère composite créé en affichant « a » et « ^ » ensemble. Unicode définit donc un ordre dans les caractères à afficher en commençant par le caractère de base et ensuite ajoute le deuxième par dessus. On peut aussi utiliser des caractères pre-composés qui seront reconnus par un seul code. Par exemple le caractère « ü » peut être encode soit par le simple code U+00FC « ü », soit par le caractère de base U+0075 « u » suivi du caractère U+0308 « ~ ». Cela permet d'avoir des appels moins lourds quand le caractère composé est assez courant dans un langage. On peut aussi décomposer un caractère composite pour traiter le texte (par exemple quand un mail est reçu avec des accents, certaines machines ne sachant pas afficher ces caractères vont décomposer le caractère et le mettre en 2 parties).

Principes du standard Unicode

Unicode a été crée par une équipe d'informaticiens professionnels, de linguistes, et de chercheurs afin qu'il devienne un standard de caractère mondial, facilement utilisable. Il doit donc avoir les caractéristiques suivantes :

- Universel
- Avoir un ordre logique (regrouper tous les caractères particulier à une langue, et par région ensuite)
- Efficacité
- Unification
- Des caractères, et non des Glyphes
- Une composition dynamique
- Une sémantique

- Des séquences d'équivalence
- Plain Texte
- Convertibilité

Dans le cas de caractères communs à plusieurs langues, on ne garde qu'un seul code et non un pour chaque langage. Unicode sait aussi gérer le sens d'écriture de manière logique.

Assigner des codes caractères

Un nombre unique est donc associé à chaque élément de code. Chacun de ces nombres est appelé un code point et, quand on y fait référence dans un texte, il est listé en hexadécimal précédé de U. Par exemple le code point U+0041 est le nombre décimal 0041 (65 en décimal, soit le caractère A).

Les codes sont classée de manière logique (par ordre alphabétique souvent) et si possible de la même manière lorsque des alphabets sont similaires.

Les éléments de code sont groupés logiquement en partant de U+0000 avec les caractères standard ASCII et continue avec le Grec, le Cyriliic, l'Arabe, l'Indien puis les symboles, la ponctuation. Par la suite on trouve les Hiragana, Katakana et Bopomofo… Apres ces blocs, on trouve les code réservé pour l'usage privé et spécifique.

Conformité avec le Standard Unicode

Un implémentation doit respecter au minimum ce qui suit :

- les caractères sont dans le répertoire commun
- les caractères sont encodés en accord avec une des formes d'encodage
- les caractères sont interprétés par les sémantiques Unicode
- les codes non assignes ne sont pas utilisés, et
- les caractères inconnus ne sont pas corrompus

Pour avoir plus de renseignements sur ce standard et obtenir une mise à jour, vous pouvez consulter le site www.unicode.org

Utilité pour notre projet et tables de références utiles :

Dans le cadre de notre projet, il nous fallait trouver un moyen de représenter des symboles mathématiques. Pour cela, il fallait d'abord avoir une représentation logique. Cela est donc possible en utilisant le standard Unicode largement répandu et géré par Java entre autres. Notre expression mathématique peut donc être représentée comme une suite de codes unicodes. Toutefois cela ne suffit pas car ensuite, il nous faut afficher cette suite, ce qui sera fourni par le prochain chapitre qui présente les fontes et les glyphes. A ce stade, l'expression peut être représente de manière logique, sans ambiguïté (grâce à l'unicité des codes).

Avant de poursuivre l'étude du moteur de rendu, nous allons juste montrer les tables de caractères qui peuvent nous intéresser (avec fourni en plus une représentation possible sous forme de glyphe). Cette sélection pourra être étendue à l'avenir.

- Les codes de U+0000 à U+024F permettent l'affichage de lettre simples et plus complexes de l'alphabet Latin

- Les codes de U+0370 à U+03FF permettent l'affichage de lettre de l'alphabet Grec.

- Les codes de U+2100 à U+214F permettent l'affichage de lettre comme les ensembles Entiers, Réels…

- Les codes de U+2190 à U+21FF permettent l'affichage de flèches.

- Les codes de U+2200 à U+22FF permettent l'affichage de divers symboles mathématiques comme les intégrales.

Les Fontes et les Glyphes.

Nous venons donc de voir la représentation logique d'un chaîne de caractères. Il nous faut maintenant nous intéresser à sa représentation physique et donc au moyen de l'afficher sur notre écran. Pour cela il faut nous intéresser au Fontes et aux glyphes qui les composent.

Les fontes (ou polices)

Une fonte est un système de représentation utilisé pour afficher des éléments auxquels on associe à chacun une image : un Glyphe. Chaque fonte est une représentation possible d'un élément. Ainsi, on peut représenter, sous Word par exemple, un caractère « a » de manières très différentes. En effet, chaque fonte est différente et par exemple, avec la police *Times New Roman*, on obtient « a » alors qu'avec une autre police telle que *Trebuchet MS* on obtient « a ». Une fonte est caractérisée par de nombreuses choses dont la plus importante est le nombre de caractères Unicode qu'elle sait gérer. En effet, certaines polices ne gèrent qu'une certaine quantité de symboles car si une fonte supportait tous les symboles existants recensés dans le standard Unicode, celle si serait très volumineuse et lourde à utiliser. Pour cela, on a des fontes spécifiques à certains langages qui ne sont utilisées que quand la nécessité s'en ressent (par exemple ajouter les caractères d'Asie).

Les fontes regroupent donc un ensemble d'images qui sont utilisées pour l'affichage. On assigne à chaque image un index pour cette fonte, ce qui permettra d'avoir le symbole sur l'écran. On peut aussi spécifier la taille de la fonte lors de l'affichage, et le type de la fonte (Gras, Italique…).

Chaque jour, on voit l'apparition de nouvelles fontes crées soit par des groupes de travail soit par des utilisateurs particuliers qui veulent avoir leur propre police. Recenser toutes les polices existantes est très difficile et il faut mieux se limiter à celle proposées par les grands groupes, ce qui assure un maintien et une rigueur dans le respect des standards. Nous verrons plus tard la fonte que nous avons gardé pour notre travail. Avant cela, nous allons parler des composantes des Fontes : les Glyphes.

Les Glyphes

Comme nous l'avons précisé plus tôt, les Glyphes sont les moyens de représenter nos caractères. Une fonte est donc composée de plusieurs milliers de ces petites images ce qui peut faire grossir rapidement la taille de la fonte. Nous allons maintenant expliquer le fonctionnement et les caractéristiques des Glyphes.

Introduction

Pour chaque glyphe, on possédera les informations suivantes qui seront détaillées par la suite:

- la représentation vectorielle du glyphe appelée « outline ».
- diverses mesures telles que la zone limite où le glyphes se trouve, sa position…

On aura aussi des informations sur la manière de grouper les glyphes pour former du texte (par exemple, gérer l'espacement entre deux glyphes, la justification, les ligatures…)

Pour ceratins termes techniques, nous garderont le terme anglais car parfois, la traduction en francais n'est pas très claire

Le tracé (outline) des Glyphes

Il est possible t'afficher des glyphes à n'importe quelle échelle, et sous n'importe quelle transformation affine (donc des rotations, translations…). Toutefois, il ne faut pas oublier que si on choisi une échelle trop faible, le rendu sera peu lisible. Il faut de même éviter de faire des associations disproportionnées entre glyphes ou même de complètement déformer un glyphe en rendant le rapport hauteur/largeur trop disproportionné.

Pour éviter cela, le format de fonte fournit aussi un langage de programmation complet utilisé pour associer à chaque glyphe de petits programmes. Leur rôle est d'aligner les points après la mise à l'échelle. Cette opération est appelée grid-fitting (adaptation à la grille) ou hinting

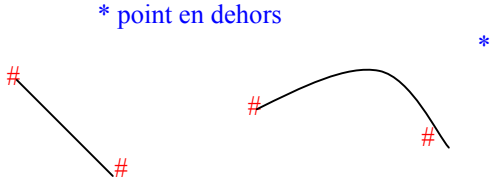
représentation vectorielle

le format source des outlines (tracé) est un ensemble de circuits fermés appelés contours. Chaque contour délimite une région interne ou externe au glyphe et peut être composée de segments ou bien de courbes de bézier du second ordre.

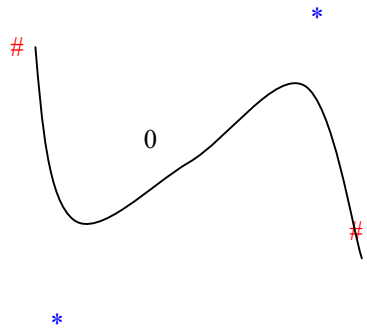
Cela est traduit par une série de points successifs possédant un indicateur pour savoir si il est sur ou en dehors de la courbe. Voici les règles appliquées pour décomposer un contour :

- deux points successifs sur la courbe indique qu'un segment de droite les joint
- un point en dehors entre deux points sur la courbe indique une courbe de bezier. Le point en dehors est alors le point de contrôle et les deux autres points les extrémités de la courbe.
- Deux points successifs en dehors oblige à créer un point virtuel sur la courbe au milieu. Cela permet de faciliter la définition d'arc de beziers successifs.

#point sur la courbe
* point en dehors



2 points sur la courbe Cas avec un point en dehors



Deux points sur la courbe et deux en dehors entre ceux ci. Le point 0 est au milieu des deux points en dehors et est un point virtuel sur la courbe à l'endroit ou la courbe va passer. Il n'apparaît pas dans la liste de points.

Les points sont mémorisés dans le fichier de la fonte comme des entier de 16-bits de coordonnes de grille.

Hinting et affichage en BMP.

Pour éviter les des erreurs d'affichage comme des tailles différentes pour les lettres telles que « E » et « H », il faut aligner les points sur la grille. De même il faut respecter les largeurs et hauteurs importantes dans toute la fonte. Par exemple, les lettres « I » et « T » doivent avoir leur ligne centrale verticale de la même largeur.

Les différents indices permettent au gestionnaire de fontes de respecter ces petites règles et d'assurer une qualité de lecture. Quand la tialle d'affichage est trop faible, il est possible de perdre quelques détails mais la famille de polices True Type minimises ces pertes.

Un des principaux souci est donc de créer des glyphes de bonne qualité, bien conçus et lisibles. Une fois que le code du glyphe à afficher a été exécuté, celui ci est converti en bitmpa (ou en pixmap avec le lissage de fonte).

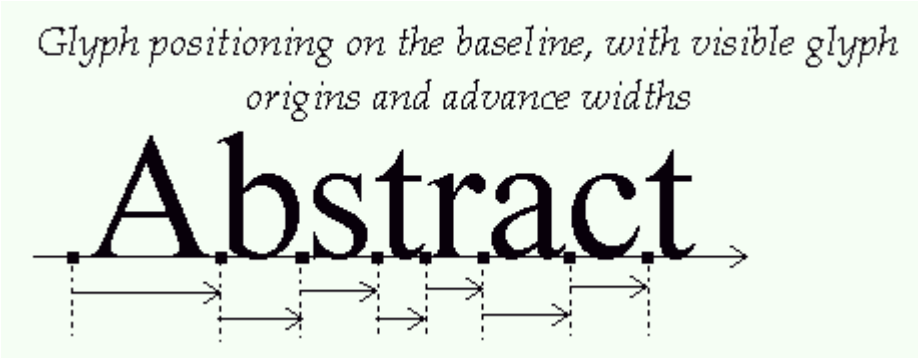
Les mesures des Glyphes (Glyph metrics)

Baseline (ligne de fuite), Pens et Layouts

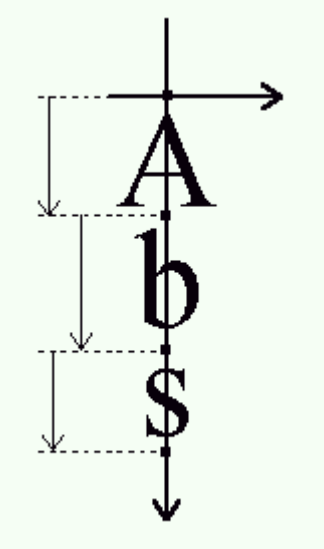
La baseline est une ligne imaginaire qui est utilisée pour « guider » les glyphes lors de l'affichage du texte. Elle peut être horizontale (comme pour le Roman, Cyrillic…), ou verticale (Chinois, Japonais…). De plus, pour afficher du texte, un point virtuel appelé la position du pen (stylo) est utilisée pour localiser les glyphes.

Chaque layout (mise en page) utilise des conventions différentes pour le placement des glyphes :

- avec une disposition horizontale, les glyphes sont « posés » sur la baseline. Le texte est affiche en incrémentant la position du pen soit vers la droite, soit vers la gauche. La distance entre 2 positions successives du pen est spécifique au glyphe et est appelée la advance width (largeur d'avancement).



- Avec une disposition verticale, les glyphes sont centrés autour de la baseline :



Mesures typographiques, et boites limites(bounding boxes)

Une grande quantité de mesures sont définies pour tous les glyphes d'une fonte donnée. Toutefois, trois d'entre elles sont seulement applicables dans le cas d'une disposition horizontale :

- l'Ascent (l'ascension) : c'est la distance de la baseline jusqu'à la coordonnée la plus haute utilisée pour placer un point du tracé. C'est une valeur positive (l'axe de y étant orienté vers le haut).

- Le Descent(la descente) : c'est la distance de la baseline jusqu'à la coordonnée la plus basse utilisée pour placer un point du tracé. C'est une valeur négative.

- Le linegap (espacement de ligne) : la distance qui doit être placée entre deux lignes de texte. La distance baseline à baseline doit être calculée de la manière

  ascent - descent + linegap

Les autres mesures les plus simples sont :

- La boite limite du glyphe (appelé bbox) : c'est une boite imaginaire qui contient le glyphe (généralement le plus petit possible). Il est composé de 4 champs xMin, yMin, xMax, et yMax.

- L'Internal leading (conduite intérieure) : ce concept vient directement du monde de la typographie traditionnelle. Cela représente la quantité d'espace dans le « leading » qui est réservé pour le glyphes qui s'affichent en dehors de la boite contenant la lettre M (le carre EM). Cela concerne par exemple les accents. On le calcule par :

  internal_leading = ascent - descent - EM_size
- L'External Leading (conduite extérieure) : c'est un autre nom pour le linegap

Bearings (support) et Advances(avancement)

Chaque glyphe possède ces deux mesures. Leur définition est constante mais leur valeur dépend du layout, un glyphe pouvant être utilisé pour afficher du texte horizontalement ou verticalement.

- 1.le left side bearing: nommé bearingX, c'est la distance horizontale de la position courante du pen au bord gauche de la bbox du glyphe.

- 2.le top side bearing: nommé bearingY, c'est la distance verticale de la baseline au haut de la bbox du glyphe.

- 3.le advance width (avancement en largeur): nommé advanceX. C'est la distance horizontale dont la position du pen doit être incrémentée ou décrémentée après que chaque glyphe soit rendu quand on traite du texte. C'est toujours positif pour la disposition horizontale, et nulle pour la verticale.

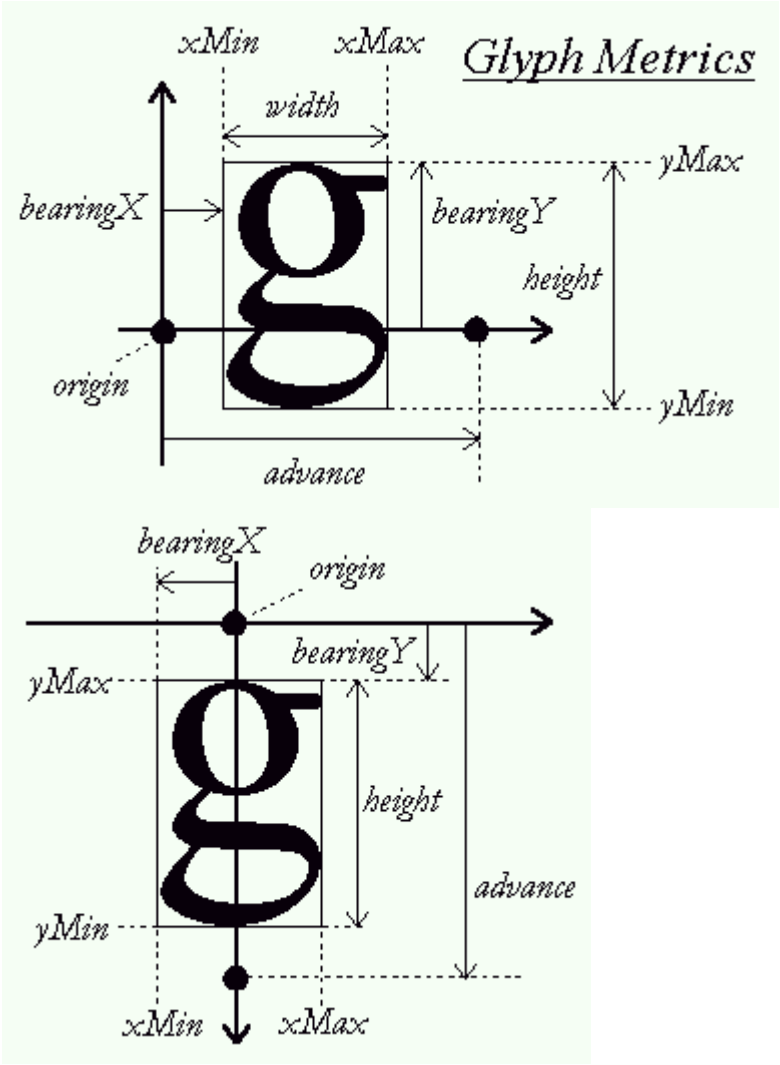
- 4.le advance height (avancement en hauteur): nommé advanceY. C'est la distance verticale dont la position du pen doit être décrémentée après que chaque glyphe soit rendu quand on traite du texte. C'est toujours positif pour la disposition verticale, et nulle pour la horizontale.

- 5.le glyph width (largeur du glyphe) : cela se calcule par *(bbox.xMax - bbox.xMin)* pour les coordonnes de fontes sans échelle. Pour les autres, son calcul nécessite plus d'attention.

- 6.le glyph height (hauteur de glyphe) : se calcule par *(bbox.yMax - bbox.yMin)* pour les coordonnes de fontes sans échelle.

- 7.le right side bearing : c'est seulement utilisé les dispositions horizontales pour décrire la distance du bord droit de la bbox à la advance width. Cela peut être calculé par

advance_width - left_side_bearing - (xMax-xMin)



4.Les effets du remplissage de grille (grid-fitting)

Toutes ces mesures sont stockées dans les unités de fonte du fichier de fonte. Elles doivent être mise dans la grille proprement pour être utilisé en tant qu'instance spécifique.

Par conséquent, un programme de glyphe ne doit pas seulement aligner le long de la grille, mais il doit aussi calculer le left side bearing et le advance width.

Un programme de glyphe peut aussi décider d'étendre ou de raccourcir l'une de ces deux mesures si besoin.

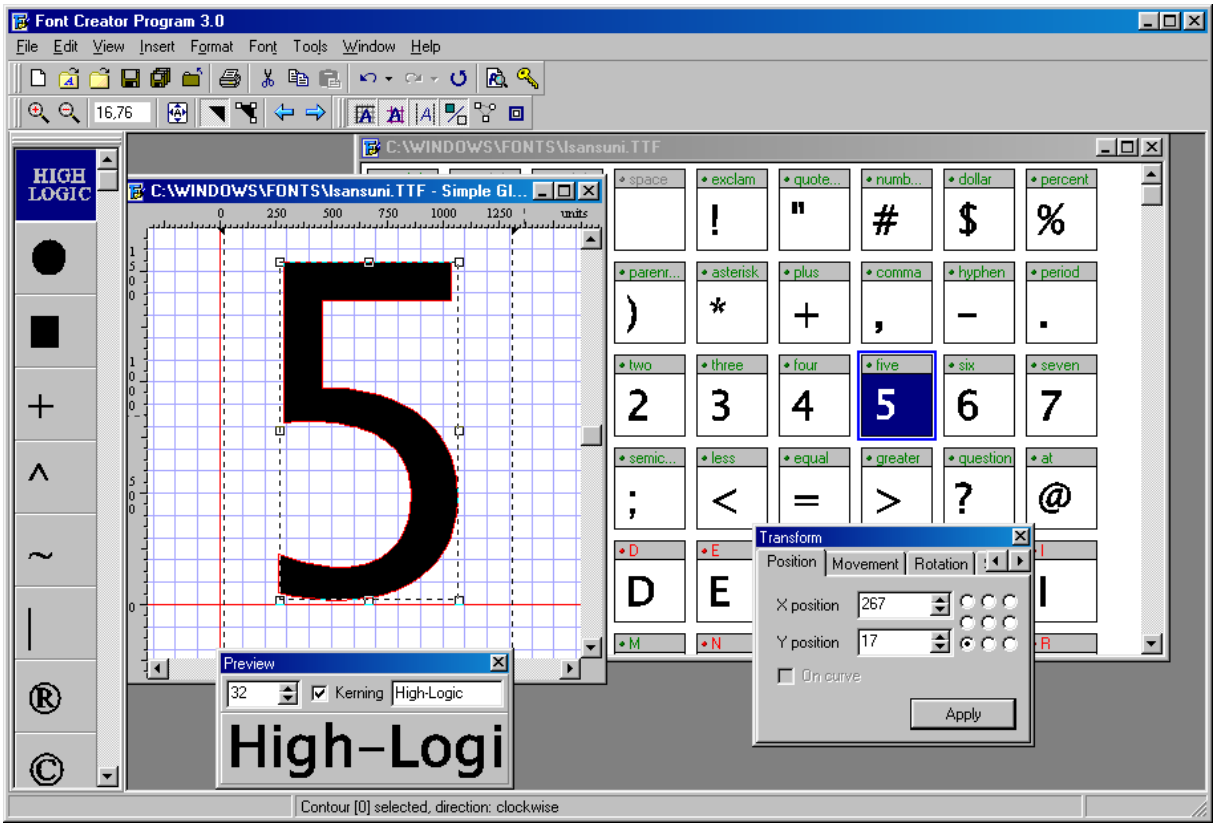
Demander toutes les mesures de tous les glyphes pour une instance est vraiment lent car il faut dans ce cas charger et traiter chaque glyphe indépendamment. Pour cette raison, on ma a disposition des mesures pre-calculées pour des valeur données (taille 8,9,10,12,14 par exemple) mais cela n'est pas toujours disponible.

3/ la recherche de la fonte

Nous avons donc vu les bases de l'affichage de caractères grâce aux fontes. Le problème est donc maintenant de trouver une de ces fontes qui gèrent les caractère qui nous intéressent (nous les avons donnés dans la partie Unicode). A ce stade, nous avions deux choix qui se présentaient : soit nous trouvions une fonte déjà existante, soit nous en créons une nouvelle.

L'option création

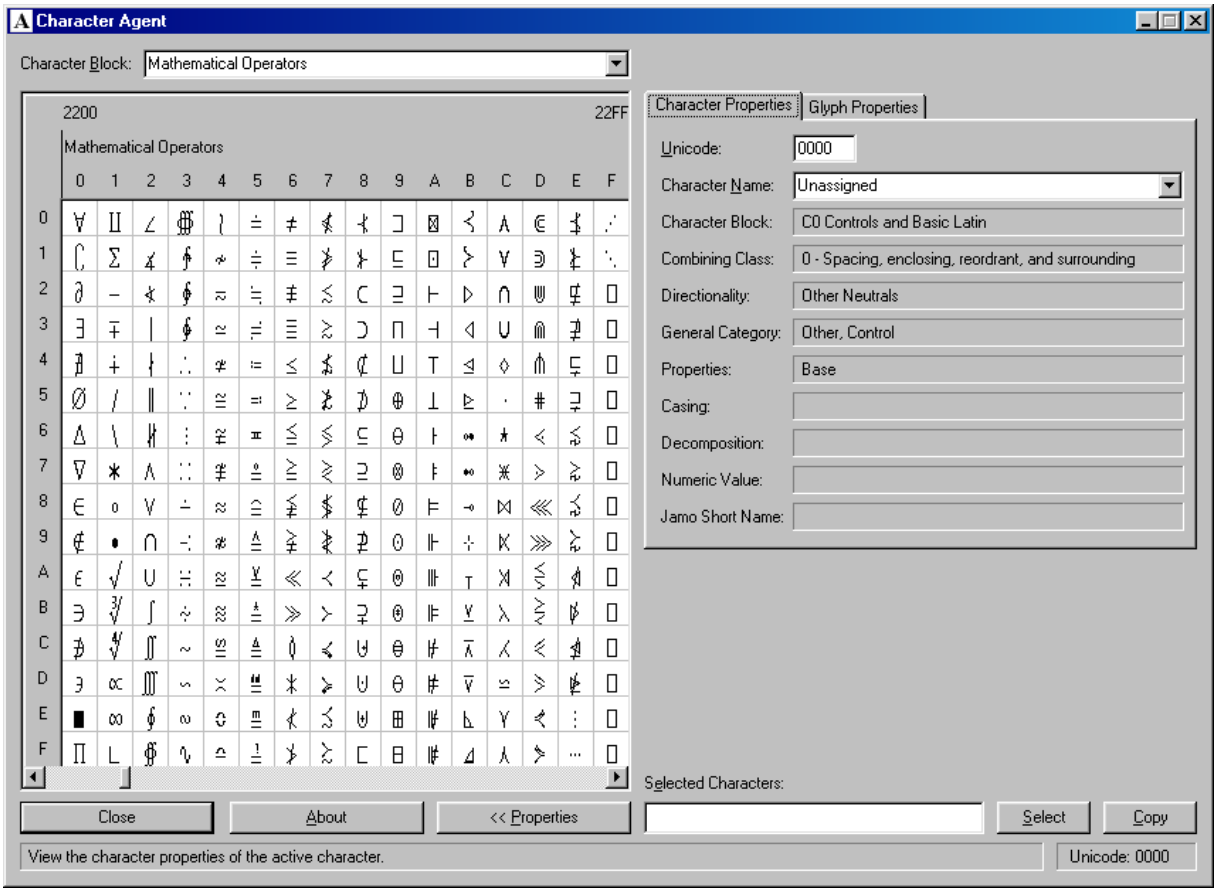
Il est possible de créer une nouvelle fonte en partant de zéro. Pour cela, nous avons trouvé plusieurs outils gratuits sur le marche. Nous avons donc regardé le fonctionnement de Font Creator Program (www.high-logic.com). Ce logiciel est assez complet et a l'avantage d'être gratuit.



Toutefois, il n'est pas aussi efficace que les éditeurs professionnels. L'usage semblait assez aisé mais le problème réside dans le fait que la création d'une fonte nécessite un temps assez important et un talent dans l'art graphique. En effet, il faut générer des fontes qui soient agréables tout en étant efficaces. De plus, nous avions besoins de plusieurs centaines de glyphes différentes pour touts les caractères à traiter. Ce travail, bien que possible, aurait été trop coûteux en temps et nous avons donc préféré chercher une fonte déjà existante. Par la suite, il est envisageable de créer notre propre fonte pour MathML. De cette manière nous pourrions perfectionner l'apparence de notre formule.

L'utilisation de fonte existante.

Cette option semblait donc être la plus intéressante pour nous. Toutefois, la recherche d'une telle fonte est au début assez difficile. En effet, il existe de nombreuses fontes disponibles sur Internet. Il est très facile de trouver une fonte payante qui gère les caractères désirés. Le but étant d'avoir une application libre, nous avons continué notre recherche. En parallèle, afin de vérifier la validité de notre fonte, nous avons utilisé un autre logiciel, appelé Character Agent (créé par le groupe Bjondi). Celui ci permet de vérifier rapidement qu'un caractère peut être affiche par la fonte. De plus il donne le code Unicode correspondant.



Avec cet outil, nous avons sélectionné 4 fontes gratuites qui gèrent les bons caractères :

- Arial Unicode MS : une fonte unicode Microsoft qui gère une grande quantité de caractères et même trop. En effet cette fonte fait 13 mo, ce qui est bien trop contraignant à charger ou à installer dans le cadre d'une application Internet.
- Caslon : cette fonte est assez intéressante ; elle ne pèse que 300 ko et gère une bonne partie des caractères classiques
- Code 2000 : elle gère plus de 32000 glyphes mais par la suite elle va s'étendre au caractères chinois, ce qui ne nous intéresse pas. Elle pèse tout de même 2 400 ko, ce qui est encore acceptable.
- Lucida Sans Unicode MS : c'est la fonte que nous avons retenue. Celle ci gère tous les caractères mathématiques, les alphabets et les symboles que nous avons soulignés dans le chapitre sur Unicode. De plus elle est rapide à charger (300 ko) et donc ne nécessite pas une connexion Internet trop gourmande.

Il existe sûrement d'autres fontes, peut être même plus spécialisée pour MathML mais au moment de l'écriture de ce rapport, c'est encore Lucida Sans Unicode qui est la plus intéressante. Ce plus, étant développée par Microsoft, on peut s'attendre à des améliorations futures.

Toutefois, ce choix de fonte n'est pas définitif et notre application n'est pas dépendante de la fonte retenue. Du moment que la fonte choisie gère la bonne quantité de glyphes, il est tout à fait possible de prendre une autre fonte.

Les problèmes de configuration

Installation de la fonte

Pour installer une fonte, c'est assez simple. Sous windows, il faut juste aller dans le répertoire Fonts de Windows (c:\Windows\Fonts par défaut) et choisir le menu Fichier/Installer Fonte. Ensuite, il faut sélectionner le fichier Lsanuni.ttf (c'est le nom de Lucida Sans Unicode) et installer celle ci. Le procédé est très facile

Les différents navigateurs

Après s'être assuré de la validité de la fonte, nous avons eu un autre problème à affronter : la gestion des fontes par les différents navigateurs et Java. Au début du développement, je n'avais installé que Internet Explore v4.0 ce qui est assez ancien.

On peut configurer le navigateur pour utiliser une certaine fonte par défaut de cette manière :

- 1.Lancer le navigateur
- 2.Choisir le menu Outils et le sous menu Options internet
- 3.Sous le panneau général, cliquer sur le bouton Polices
- 4.Dans la case Langage, choisir la langue à voir
- 5.Mettre en valeur la nouvelle police dans les différentes cases pour choisir la fonte adéquate (mettre Lucida Sans Unicode partout par exemple)
- 6.Valider

Pour tester cette application nous avons utilisé une petite applet qui devait afficher certains caractères comme l'intégrale. Cependant, le résultat obtenu sous Internet Explorer 4.0 n'etait pas satisfaisant. Malgré la configuration, le navigateur n'affichait rien que des caractères inconnus.

Donc il fallait trouver une configuration qui marche. J'ai donc mis à jour Internet Explorer en premier lieu à la version 5.5. Cela ne marchait toujours pas mais cela doit être contournable. Il semblerait que IE ne prenne pas en compte ou ne connaisse pas la fonte.

Par la suite, j'ai donc installé un autre navigateur assez répandu : Netscape Communicator 4.76. Sur celui ci, tous les caractères s'affichaient sans problème et donc la faisabilité st dont bien prouvée. Sachant cela, les restes des tests se font directement avec l'appletviewer. Ensuite il suffira de l'intégrer dans une page HTML que nous pourrons visualiser avec Netscape. Toutefois, IE devra être capable de gérer notre applet par la suite. Il doit certainement exister une technique permettant de réaliser cela. Cela consistera sûrement à mettre à jour la Java Virtual Machine de IE, ce qui semble être très difficile pour le moment.

Ces différents tests de configuration ont demandé assez de temps pour bien cerner les problèmes à ce niveau. Il etait très important de s'assurer que cet affichage serait possible. Il ne fallait donc pas négliger cette partie pour éviter de se retrouver devant une application qui ne tournerait que sur nos machines de développement. Il ne faut pas oublier le but final qui est d'assurer une utilisations pour tous les utilisateurs d'Internet et voire même peut être, être intégré dans les futures versions des navigateurs.

L'implémentation du moteur de rendu

Nous allons tout d'abord montrer les différentes classes et méthodes fournies en Java qui nous permettent de travailler sur les fontes et les glyphes. Ensuite, à l'aide d'exemples simples nous allons expliquer les étapes successives qui nous permettent de partir d'un code unicode et d'arriver à un glyphe pouvant être placé où nous voulons sur l'écran et auquel nous aurons pu appliquer une ou plusieurs transformations.

Comment spécifier une fonte avec Java.

Utilisation de la classe Java.awt.Font

Java fournit de nombreuses fonctionnalités concernant les fontes. Ci dessous nous allons juste spécifier une fonte pour une chaîne de caractère :

Code :

```
import java.awt.*;
import java.awt.font.*;
import java.awt.Graphics.*;
import javax.swing.*;


public class PApplet extends java.applet.Applet {
   public void init() {

      Font f ;
      String text;
      f=new Font("Lucida Sans Unicode", Font.PLAIN, 18);
      setFont(f);
      //affichage en spécifiant une chaine de carctères
      text1="Chaine 1";
      add(new Label (text1));
      //affichage en spécifiant un code Unicode
      text2="\u2201";
      add(new Label (text2));

   }
}
```

On voit donc que l'utilisation est assez simple. On peut spécifier la fonte à utiliser et en changer très facilement. On peut aussi spécifier le type et la taille de la fonte. Il existe encore d'autres fonctionnalités proposées pour les fontes que nous verrons par la suite. On peut aussi citer la classe Java.awt.FontMetrics qui permet de récupérer des informations sur les fontes.

Dans cet exemple nous voyons aussi que on doit pouvoir spécifier indifféremment une chaîne de caractères, ou bien un code unicode pour l'affichage. Java gère le standard Unicode. En fait nous verrons par la suite que certaines parties ne sont pas encore très bien implémentées.

L'utilisation des glyphes

L'autre partie importante est l'utilisation de la classe java.awt.font.GlyphVector. Comme nous l'avons vu plus haut, une fonte est composée de glyphes. La fonte se charge alors d'elle même, en respectant certaines règles de typographie, d'afficher ce qu'on lui demande. Toutefois, cela n'est satisfaisant que dans le cas d'une utilisation simple. Dans certains cas, il est intéressant de pouvoir spécifier nous même la manière dont nous voulons que les éléments soient afficher. Dans notre cas par exemple, si on prend un intégrale avec ses bornes inf et bornes sup, on doit pouvoir dire d'afficher les bornes dans un taille plus petite et à une position particulière (en haut et en bas du glyphe intégrale). Nous allons donc traduire notre expression mathématique sous forme d'un vecteur de Glyphes d'où l'utilité de la classe GlyphVector. Pour chaque élément de notre vecteur, nous pourrons spécifier

la position et la taille du glyphe. Grâce aux outils proposés par Java, on peut contrôler parfaitement ces mesures. Nous allons montrer un petit exemple montrant l'utilisation de ces GlyphVector.

Exemple :

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.font.*;
import java.awt.Graphics;
import java.awt.geom.*;
import java.io.*;
import mathit.font.util.*;

public class GlyphDraw {
   static Tableau frame;
      //en argument on donne deux valeurs correspondant à la
valeur décimale du code unicode
      public static void main( String[] args ) {
         int test[]=new int[2];
         test[0] = new Integer(args[0]).intValue();
         test[1] = new Integer(args[1]).intValue();
         System.out.println(test[0]);
         System.out.println(test[1]);
         frame = new Tableau(test);
         frame.setVisible(true);
      }
}

class Tableau extends JFrame {

   static final Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
   static final int largeurEcran = screenSize.width;
   static final int hauteurEcran = screenSize.height + 2;
   int l = 150;
   int h = 150;
   Container pane;

   public Tableau(int[] indice) {
      super();
      pane = getContentPane();
      pane.setLayout(new FlowLayout());
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);}});
      setBounds ((largeurEcran - l) / 2, (hauteurEcran - h) / 2,
l, h);
      setBackground(Color.white);

      //on initialise les caractères unicodes à afficher
      MathML texte = new MathML(indice);

      pane.add(texte);
   }
}

class MathML extends Canvas {
      int l = 0, h = 0, xt = 0, yt = 0, lt = 0;
      Font police;
```

```
         FontMetrics métriques;
      Color fCouleurC = new Color(255, 255, 255),fCouleurF = new
Color(0, 0, 0);
      protected int matrice[];
         protected char matriceChar[];
         protected Vector glyphVector ;
         protected int charIndice[];
         protected UnicodeToGlyphCode unicodeToGlyphCode;

   public MathML(int[] indice) {
      super();

      charIndice = indice;
      police = new Font("Lucida Sans Unicode", Font.PLAIN, 40);
      métriques = getFontMetrics(police);
      setFont(police);
      setBackground(fCouleurC);
      setSize(100,100 );
   }

   public void paint(Graphics g) {

      //on convertit pour pouvoir utiliser la methode
drawGlyphVector()
      Graphics2D g2 = (Graphics2D)g;

      Font police=getFont();
      System.out.println(police.toString());

      g2.setFont(police);
      g2.setColor(fCouleurF);
      FontRenderContext frc = g2.getFontRenderContext();
      GlyphVector essai;

      int[] t =new int[2];
      //on convertit les entiers en code de la fonte
      t[0] = unicodeToGlyphCode.getGlyphCode((char)charIndice[0]);
      t[1] = unicodeToGlyphCode.getGlyphCode((char)charIndice[1]);
      //on crée le vecteur de glyphes
      essai = police.createGlyphVector(frc,t);

      //on déplace le premier glyphe du vecteur
      Point2D position = essai.getGlyphPosition(0);
      double x = position.getX();
      double y = position.getY();
      position.setLocation(x,y+10);

      //on applique une transformation affine au deuxième élément
(agrandissement *4
      AffineTransform transfo = essai.getGlyphTransform(0);
      transfo.setToScale(4,4);
      System.out.println(""+transfo);
      essai.setGlyphTransform(1,transfo);
      System.out.println(""+essai.getGlyphTransform(1));

      essai.setGlyphPosition(0,position);

      //on affiche le vecteur
      g2.drawGlyphVector(essai,30,50 );
   }
}
```

Dans cet exemple nous voyons toutes les principales fonctions sur les glyphes qui peuvent nous intéresser. Rien que dans cet exemple, nous avons trouvé deux bugs que nous allons vous expliquer.

En premier lieu, nous avons donc crée un vecteur de Glyphes à partir d'un tableau de code unicode :

GlyphVector essai = police.createGlyphVector(frc,t);

Il existe de nombreux constructeurs mais c'est celui la qui est le plus intéressant. En effet comme nous le disons depuis le début, c'est le code unicode qui nous intéresse car il assure une identification unique.

En fait à ce stade, nous avons rencontré un premier problème qui s'est révélé être un problème de la JDK. Nous avons voulu envoyer un bug report mais en fait le problème avait déjà été soulevé par d'autres personnes. Le problème est le suivant :

La JDK ne peut pas créer directement un GlyphVector à partir d'un code Unicode. Par contre on peut obtenir le code dy glyphe correspondant dans la Fonte (qui est différent du code Unicode) et utiliser cet indice de la fonte (le GlyphCode) pour créer notre GlyphVector, car de ceete manière on affiche bien ce que nous voulions à la base. Nous avons donc créé une bibliothèque permettant de faire comme si nous pouvions utiliser directement le code Unicode. Ce package s'appelle mathit*.font.util.* Il suffit f'utiliser la commande *unicodeToGlyphCode.getGlyphCode(CodeUnicode);*

Nous n'allons pas nous etendre trop longuement sur cela mais il faut juste noter que nous utilisons un fichier contenant toutes les correspondances Unicode →Glyphcode pour une certaine fonte. Nous créons ce fichier une fois (ce qui peut prendre du temps machine) et par la suite, nous lisons directement ce fichier pour éviter de devoir générer ce fichier à chaque appel.

Il nous a donc fallu utiliser cette méthode pour créer et afficher notre vecteur à partir des codes unicodes. Nous utilisons donc une conversion intermédiaire qui par la suite devra être ôtée une fois le bug de la JDK corrigé

L'exemple montre aussi la manière de changer la position par défaut d'un glyphe en utilisant la méthode *setGlyphPosition(GlyphIndex,Point2D)* cela se fait très facilement et ce parfaitement.

Enfin, nous avons essayé d'appliquer une transformation affine à un glyphe. Pour cela on peut utiliser une matrice de transformation (classe AffineTransform) qui consiste en matrice 3*3. Celle ci permet de définir des transformations comme les translations, les rotations, les agrandissements/réductions… Son utilisation est aussi très simple. Cela nous permet donc de changer facilement la taille d'un élément du vecteur. Par exemple dans le cas de l'intégrale, on pourra spécifier que les bornes seront 4 fois plus petites que le signe intégrale lui même (et on coordinera le tout avec en plus un positionnement différent des bornes). Toutefois, nous nous sommes une fois de plus heurté à un bug de la JDK. Il est apparu que la méthode *setGlyphTransform(GlyphIndex,AffineTransform)* n'etait pas implémentée. En tout cas la modification ne prend pas effet et on se retrouve toujours avec une matrice de transformation identité. Ce but fait déjà l'objet d'un bug report et nous avons pu voir des personnes confrontées au mêmes problèmes. Donc nous avons tout de même implémenté ces changement d'échelles qui sont primordiaux pour la compréhension d'une expression mathématique même si pour l'instant, cela ne peut pas apparaître. En effet, si tous les membre d'une expression mathématique avaient tous la même taille, on aurait de grandes difficultés à saisir le sens de la formule.

Nous avons alors pensé alors à un solution de secours. Celle ci est moins « propre » mais en cas de besoin, elle permet de changer les tailles. Cela consiste décomposer notre GlyphVector en autant de GlyphVector unité. Pour chacune de ceux–ci on a la possibilité de changer la taille de la police. On peut donc obtenir un résultat assez clair. Toutefois, cette méthode n'est pas exactement équivalente à celle voulue. En Effet on ne peut pas répondre à des règles typographiques telles que : dans une intégrale, les bornes doivent être 4 fois plus petite que le signe intégrale. C'est une solution envisageable mais il est préférable d'utiliser la méthode setGlyphTransform définie. Encore une fois, le but étant de montrer la faisabilité, on peut pour le moment utiliser la deuxième solution en attendant la mise à jour de la JDK.

Donc au terme de ce chapitre, nous voyons que nous pouvons afficher n'importe quel caractère de la manière que nous voulons. Il ne reste alors plus qu'à appliquer à chaque balise MathML que nous rencontrerons, une méthode d'affichage qui pourra prendre en compte les arguments ainsi que les règles de position à respecter.

Le moteur de rendu final

Nouas maintenant tous les éléments nécessaires à l'écriture du moteur de Rendu. En fait il suffit de regrouper toutes ces différentes méthodes. Nous partons donc d'un document MathML. Le fonctionnement est le suivant : pour chaque balise que nous rencontrons, nous récupérerons les différents nœuds fils et pour chacun de ces éléments, nous définissons une procédure d'affichage spécifique. Selon le type du nœud nous aurons une taille et un positionnement dépendant du nœud père. Par exemple pour l'intégrale, lorsque nous rencontrons la balise « msubsup », nous récupérons en même temps les 2 fils et nous affichons dans l'ordre de haut en bas : argument2, le contenu de msubsup, argument1. Ensuite pour chaque argument, nous continuons la lecture et selon le type, nous utiliserons l'affichage correspondant. Nous construisons donc la formule mathématique en parcourant l'arbre MathML au fur et à mesure Nous avons implémenté une bonne dizaine de balises MathML et le fonctionnement est toujours similaire donc il n'est ps difficile d'implémenter les autres balises. Toutefois cela prend du temps. En effet, il faut penser à la manière dont on veut afficher les élément les ns par rapport aux autres et donc il y aura des réglages à faire afin de proposer un affichage qui soit lisible et qui respecte la formulation sur papier. Nous avons donc bien prouvé que ce moteur de rendu marchait sur deux exemples : une intégrale et une fraction. Nous avons donc développé une première version qui servira de référence pour les évolutions futures de ce moteur de rendu. Pour le moment, il n'est pas complet mais nous avons traité tous les types de balises. Il reste à définir certaines méthodes d'affichage.

/////METTRE LA LISTE DES BALISES GEREES

Conclusion

La réalisation du moteur de rendu a été très intéressante et nous a permis d'apprendre de nombreuses choses sur l'utilisation de certaines entités que nous ne pensions pas aussi complexes au début. Nous avons utilisé deux standards très utilisés dans la programmation : le standard Unicode qui assure une reconnaissance unique des symboles, et les Fontes qui sont les collections d'images permettant l'affichage de ces symboles.

De plus, nous avons enrichi nos connaissances en Java et nous avons même pu soulever deux bugs important de la JDK, ce qui nous a appris que nous ne devions pas nous fier aveuglément à ce qui devait marcher. Nous donc réalisé des tests et cherché des solutions de rechange ce qui est tres enrichissant. La pahse de debuggage est donc très longue mais est aussi incontournable.

Nous avons aussi pu voir les problèmes de configuration posés par les navigateur Internet qui n'étaient pas forcément entièrement compatibles avec les standards que nous utilisions.

Finalement nous avons réussi à créer un moteur qui tourne et cela est assez gratifiant.

Conclusion

Au final nos travaux ont abouti à la réalisation :

- D'un parseur MathML gérant l'ensemble des spécifications concernant la DOM MathML,
- D'un moteur de rendu MathML gérant une dizaine d'éléments de présentation MathML,
- D'un parseur TeX traitant la plupart des sigles mathématiques mais ne gérant pas les autres balises de mise en page de document,
- D'un convertisseur MathML-TeX composé du transformeur Xalan et d'un ensemble de feuilles de style XSLT.

Concernant le parseur TeX, le domaine mathématique n'a pas été traité en entier. Cependant des algorithmes génériques permettent de traiter bon nombre de fonctions et commandes. Le domaine de mise en forme de Tex reste à être étudié ;nous pensons que la modélisation de ce domaine ne doit pas être mise en rapport avec celle du domaine mathémathique.

La conversion de l'arbre TeX en MathML de présentation a été réalisée partiellement pour les fonctions et termes principaux. Elle ne l'a pas été pour le MathML de contenu ni dans l'autre sens, i.e. de MathML en TeX.

Bibliographie

De l'usage intensif des livres et d'Internet…

Livres

- "XML, le guide de l'utilisateur", Elliote Rusty Harold, Ed. OEM
- "XML, langage et application", Alain Michard, éditions Eyrolles
- "Java & XML", Brett McLaughlin, éditions O'Reilly
- "XML, précis et concis", Robert Eckstein et Michel Casabianca, éditions O'Reilly
- introduction à LaTeX2e, Tobias Oetoker, Hubet Partl, Irene Hyna et Elisabeth Schlegl

Sites Internet

- http://www.w3c.org :LA référence en matière de recommandation et de spécifications Internet. En anglais.
- http://www.zvon.org: un site de tutorats sur XML, et les standards associés, dont un excellent sur MathML. Possède également de bons tutorats sur le browser Mozilla. En anglais.
- http.//xml.apache.org :ce site regroupe l'ensemble des travaux du groupe XML Apache Project. En anglais.
- http://www.mutu-xml.org: un excellent site regroupement un grand nombre d'informations, de présentation et de tutorats sur XML, sur ses applications, et sur les standards associés. L'ensemble de la présentation des XML Schemas de ce rapport est inspiré par ce site. En français.
- http://java.sun.com: le site officiel de Sun sur Java, qu'il n'est plus utile de présenter.
- http://www.megginson.com/
- http://rpmfind.net/veillard/Talks/200011XML/all.htm
- http://www.mathtype.com/

Glossaire

De l'utilisation intempestive des abréviations…

- **API** : Application Program Interface
  Ensemble de classes dédiées à une tâche particulière
  La plupart des API sont composées des interfaces qui définissent leur comportement mais aussi de leurs implémentations

- **CSS** :Cascading Style Sheet
  Liste de règles d'affichage appliqués à un document HTML ou XML

- **DOM** : Document Object Model
  Langage d'interfaçage permettant un accès et une mise à jour dynamique du contenu et de la structure d'un document XML.

- **DTD** : Document Type Definition
  Elle spécifie le contenu autorisé et la structure des éléments d'un fichier XML

- **HTML** : HyperText Markup Language
  Langage textuel à base de balises. Application du standard SGML.
  Utilisé pour la publication sur Internet

- **HTTP** : Hyper Text Transfert Protocol
  Protocole de transport des données hypertextes (HTML) basé sur TCP

- **Infoset** :Norme en cours de validation par le W3C. Le terme abrégé *infoset* désigne une arborescence abstraite de document au moyen de 17 catégories que peuvent prendre les items d'information attachés aux nœuds de cet arbre : document, élément, attribut, instruction de traitement, référence d'entité non actualisée, caractères, commentaire, déclaration de type de document, entité interne, entité externe, entité non parsée, notation, délimiteur ouvrant d'entité, délimiteur fermant d'entité, espace nominatif, délimiteur ouvrant de CDATA, délimiteur fermant de CDATA (CDATA : les données qui sont des caractères).

- **SAX** : Simple API for XML
  Interface standard pour parser des documents XML
  Basé sur un Mode événementielle

- **SGML** : Standard Generalized Markup Language
  Métalangage standard à base de balises

- **URL** : Uniform Resource Location
  Sorte d'adresse qui permet de trouver le document désigné.

- **XML** : Extensible Markup Language
  Métalangage à base de balises. Application du standard SGML.
  Utilisé pour la représentation des données

- **Xpath :** langage qui permet d'adresser, de désigner, des objets structurels contenus dans un document XML

- **Xpointer :** Construite sur XPath, définit de *nouvelles fonctionnalités*, telle la possibilité de définir une région (un ensemble de mots), sur laquelle doit être faite la mise en relation. se définit alors en un ensemble d'extensions à Xpath

- **NameSpaces :** « Espaces de Noms »
  Permettent de qualifier de manière unique des éléments et des attributs. On sait alors à quel domaine de définition se rapporte un objet et comment il doit être interprété, selon sa spécification.

- **XSL** : Extensible Stylesheet Language
  Application du standard XML.
  définit le vocabulaire permettant de transformer un document XML, puis le vocabulaire pour le formater après transformation.

Appendice A

La DOM MathML

La hiérarchie des interfaces

- MathMLDOMImplementation
- MathMLDocument
- MathMLNodeList
- MathMLElement
  - MathMLSemanticsElement
  - MathMLAnnotationElement
  - MathMLXMLAnnotationElement
  - MathMLPresentationElement
    - MathMLGlyphElement
    - MathMLSpaceElement
    - MathMLPresentationToken
      - MathMLOperatorElement
      - MathMLStringLitElement
    - MathMLFractionElement
    - MathMLRadicalElement
    - MathMLScriptElement
    - MathMLUnderOverElement
    - MathMLMultiScriptsElement
    - MathMLTableElement
    - MathMLTableRowElement
      - MathMLLabeledRowElement
    - MathMLAlignGroupElement
    - MathMLAlignMarkElement
  - MathMLContentElement
    - MathMLContentToken

        - MathMLCnElement
        - MathMLCiElement
        - MathMLCsymbolElement
      - MathMLPredefinedSymbol
      - MathMLIntervalElement
      - MathMLConditionElement
      - MathMLDeclareElement
      - MathMLVectorElement
      - MathMLMatrixElement
      - MathMLMatrixrowElement
      - MathMLPiecewiseElement
      - MathMLCaseElement
- MathMLContainer
  - MathMLMathElement
  - MathMLPresentationContainer
  - MathMLContentContainer

Tables des éléments et de leur interface DOM associée

Eléments MathML	Interface DOM
math	MathMLMathElement
mi	MathMLPresentationToken
mn	MathMLPresentationToken
mo	MathMLOperatorElement
mtext	MathMLPresentationToken
mspace	MathMLSpaceElement
ms	MathMLStringLitElement
mglyph	mglyph MathMLGlyphElement
mrow	MathMLPresentationContainer
mfrac	MathMLFractionElement
msqrt	MathMLRadicalElement
mroot	MathMLRadicalElement
mstyle	MathMLStyleElement
merror	MathMLPresentationContainer
mpadded	MathMLPaddedElement
mphantom	MathMLPresentationContainer
mfenced	MathMLFencedElementmenclose
menclose	MathMLEncloseElement
msub	MathMLScriptElement
msup	MathMLScriptElement

msubsup	MathMLScriptElement
munder	MathMLUnderOverElement
mover	MathMLUnderOverElement
munderover	MathMLUnderOverElement
mmultiscripts	MathMLMultiScriptsElement
mtable	MathMLTableElement
mlabeledtr	MathMLLabeledRowElement
mtr	MathMLTableRowElement
mtd	MathMLTableCellElement
maligngroup	MathMLAlignGroupElement
malignmark	MathMLAlignMarkElement
maction	MathMLActionElement
cn	MathMLCnElement
ci	MathMLCiElement
csymbol	MathMLCsymbolElement
apply	MathMLApplyElement
reln	MathMLContentContainer
fn	MathMLFnElement
interval	MathMLIntervalElement
inverse	MathMLPredefinedSymbol
condition	MathMLConditionElement
declare	MathMLDeclareElement
lambda	MathMLLambdaElement
compose	MathMLPredefinedSymbol
ident	MathMLPredefinedSymbol
domain	MathMLPredefinedSymbol
codomain	MathMLPredefinedSymbol
image	MathMLPredefinedSymbol
domainofapplication	MathMLContentContainer
piecewise	MathMLPiecewiseElement
piece	MathMLCaseElement
otherwise	MathMLContentContainer
quotient	MathMLPredefinedSymbol
exp	MathMLPredefinedSymbol
factorial	MathMLPredefinedSymbol
divide	MathMLPredefinedSymbol
max	MathMLPredefinedSymbol
min	MathMLPredefinedSymbol
minus	MathMLPredefinedSymbol
plus	MathMLPredefinedSymbol
power	MathMLPredefinedSymbol
rem	MathMLPredefinedSymbol
times	MathMLPredefinedSymbol
root	MathMLPredefinedSymbol
gcd	MathMLPredefinedSymbol
and	MathMLPredefinedSymbol
or	MathMLPredefinedSymbol
xor	MathMLPredefinedSymbol
not	MathMLPredefinedSymbol
implies	MathMLPredefinedSymbol
forall	MathMLPredefinedSymbol
exists	MathMLPredefinedSymbol
abs	MathMLPredefinedSymbol
conjugate	MathMLPredefinedSymbol
arg	MathMLPredefinedSymbol

real	MathMLPredefinedSymbol
imaginary	MathMLPredefinedSymbol
lcm	MathMLPredefinedSymbol
floor	MathMLPredefinedSymbol
ceiling	MathMLPredefinedSymbol
eq	MathMLPredefinedSymbol
neq	MathMLPredefinedSymbol
gt	MathMLPredefinedSymbol
lt	MathMLPredefinedSymbol
geq	MathMLPredefinedSymbol
leq	MathMLPredefinedSymbol
equivalent	MathMLPredefinedSymbol
approx	MathMLPredefinedSymbol
factorof	MathMLPredefinedSymbol
int	MathMLPredefinedSymbol
diff	MathMLPredefinedSymbol
partialdiff	MathMLPredefinedSymbol
lowlimit	MathMLContentContainer
uplimit	MathMLContentContainer
bvar	MathMLBvarElement
degree	MathMLContentContainer
divergence	MathMLPredefinedSymbol
grad	MathMLPredefinedSymbol
curl	MathMLPredefinedSymbol
laplacian	MathMLPredefinedSymbol
set	MathMLSetElement
list	MathMLListElement
union	MathMLPredefinedSymbol
intersect	MathMLPredefinedSymbol
in	MathMLPredefinedSymbol
notin	MathMLPredefinedSymbol
subset	MathMLPredefinedSymbol
prsubset	MathMLPredefinedSymbol
notsubset	MathMLPredefinedSymbol
notprsubset	MathMLPredefinedSymbol
setdiff	MathMLPredefinedSymbol
card	MathMLPredefinedSymbol
cartesianproduct	MathMLPredefinedSymbol
sum	MathMLPredefinedSymbol
product	MathMLPredefinedSymbol
limit	MathMLPredefinedSymbol
tendsto	MathMLPredefinedSymbol
exp	MathMLPredefinedSymbol
ln	MathMLPredefinedSymbol
log	MathMLPredefinedSymbol
sin	MathMLPredefinedSymbol
cos	MathMLPredefinedSymbol
tan	MathMLPredefinedSymbol
sec	MathMLPredefinedSymbol
csc	MathMLPredefinedSymbol
cot	MathMLPredefinedSymbol
sinh	MathMLPredefinedSymbol
cosh	MathMLPredefinedSymbol
tanh	MathMLPredefinedSymbol
sech	MathMLPredefinedSymbol

csch	MathMLPredefinedSymbol
coth	MathMLPredefinedSymbol
arcsin	MathMLPredefinedSymbol
arccos	MathMLPredefinedSymbol
arctan	MathMLPredefinedSymbol
arccosh	MathMLPredefinedSymbol
arccot	MathMLPredefinedSymbol
arccoth	MathMLPredefinedSymbol
arccsc	MathMLPredefinedSymbol
arccsch	MathMLPredefinedSymbol
arcsec	MathMLPredefinedSymbol
arcsech	MathMLPredefinedSymbol
arcsinh	MathMLPredefinedSymbol
arctanh	MathMLPredefinedSymbol
mean	MathMLPredefinedSymbol
sdev	MathMLPredefinedSymbol
variance	MathMLPredefinedSymbol
median	MathMLPredefinedSymbol
mode	MathMLPredefinedSymbol
moment	MathMLPredefinedSymbol
momentabout	MathMLContentContainer
vector	MathMLVectorElement
matrix	MathMLMatrixElement
matrixrow	MathMLMatrixRowElement
determinant	MathMLPredefinedSymbol
transpose	MathMLPredefinedSymbol
selector	MathMLPredefinedSymbol
vectorproduct	MathMLPredefinedSymbol
scalarproduct	MathMLPredefinedSymbol
outerproduct	MathMLPredefinedSymbol
annotation	MathMLAnnotationElement
semantics	MathMLSemanticsElement
annotation-xml	MathMLXMLAnnotationElement
integers	MathMLPredefinedSymbol
reals	MathMLPredefinedSymbol
rationals	MathMLPredefinedSymbol
naturalnumbers	MathMLPredefinedSymbol
complexes	MathMLPredefinedSymbol
primes	MathMLPredefinedSymbol
exponentiale	MathMLPredefinedSymbol
imaginaryi	MathMLPredefinedSymbol
notanumber	MathMLPredefinedSymbol
true	MathMLPredefinedSymbol
false	MathMLPredefinedSymbol
emptyset	MathMLPredefinedSymbol
pi	MathMLPredefinedSymbol
eulergamma	MathMLPredefinedSymbol
infinity	MathMLPredefinedSymbol