Spécification de MaTEX version (alpha)

Par :
Éric Marcotte

avril 2015

1 Introduction

1.1 Présentation

Le but de ce document est de décrire les idées générales d'un langage qui permettrait de générer du code source LaTeX à l'aide d'une syntaxe moins lourde et plus simple. LaTeX étant beaucoup utilisé dans le domaine scientifique, il existe une foule de petits plugiciels ou de petits programmes facilitant l'édition de code LaTeX dans plusieurs contextes. Peu de ces plugiciels ou de paquetages servent pour l'édition de texte dans un contexte scientifique. Voici donc une spécification sommaire pour un langage spécialisé dans la génération de code source LaTeX. Nous allons appeler ce langage MaTeX pour le reste du document.

1.2 Objectif

Comme une équation mathématique peut rapidement devenir illisible en LaTeX, il devient pénible d'essayer de manipuler des équations directement dans un éditeur. Le but de MaTeX n'est pas de redéfinir LaTeX autrement, mais bien de développer un langage syntaxiquement plus simple que LaTeX pour générer des structures. Au lieu de se rappeler par cœur la structure, l'utilisateur pourra alors générer des morceaux de code source LaTeX qu'il intégrera au fur et à mesure dans son code source LaTeX. Comme une des forces de LaTeX est de pouvoir définir en profondeur la structure du texte, MaTeX ne visera pas cela. Il visera plutôt à créer le brute du code latex rapidement et efficacement puis il sera fignolé par la suite.

Notons qu'il est aussi possible de créer des macros en LaTeX pour obtenir des résultats similaires. Toutefois, en plus d'être difficile à écrire, elles peuvent avoir des résultats inattendus. En effet, l'analyse du code source LaTeX n'est pas optimal et les erreurs générées sont souvent inutiles. MaTeX réglera ce problème en effectuant une meilleure analyse de son code source pour pouvoir indiquer précisément à l'usager quelle est son erreur.

1.3 Structures

Pour commencer voici un exemple simple illustrant bien le but et la problématique que MaTeX s'attaquera.

1.3.1 Exemple de base

Considérons cette section de code source LaTeX suivante :

```
\[\sum\limits_{i=1}^{\lfloor log_{2}e\rfloor}\left(\frac{e}{2^{i}
}\right)^{2} \lfloor log_{2}b\rfloor^{2}+\left(\frac{e}{2^{i-1}}-
1\right)\lfloor log_{2}b \rfloor \ast \lfloor log_{2}b \rfloor\]
```

Si vous pouvez visualiser rapidement le résultat exact qui sera généré, votre langue maternelle est sans doute LaTeX. Voici le résultat une fois compilé :

$$\sum_{i=1}^{\lfloor log_2 e \rfloor} \left(\frac{e}{2^i}\right)^2 \lfloor log_2 b \rfloor^2 + \left(\frac{e}{2^{i-1}} - 1\right) \lfloor log_2 b \rfloor * \lfloor log_2 b \rfloor$$

Maintenant imaginez que cette équation est dans le milieu d'une suite d'équation semblable et que vous devez modifier un des exposants dans le code source. La lourdeur syntaxique rendra la tâche plus compliquée qu'elle ne devrait l'être.

Imaginons maintenant comment on pourrait utiliser MaTeX pour générer du code source latex plus efficacement. Le problème consiste à pouvoir écrire les structures sous-jacente de l'organisation du rendu LaTeX sans avoir à les connaître toutes par cœur. Nous pouvons décomposer l'expression ci-haut en morceaux. Premièrement, on a une somme mathématique dont la structure est :

```
\sum\limits_{ index = indice }^{ borne } expression
```

Lorsqu'on doit réécrire la structure d'une somme très souvent, il serait pratique d'avoir un raccourci. En MaTeX, il suffirait d'écrire :

```
\[%% Sum();
\]
```

pour obtenir :

```
\[ \sum\limits_{i = index\_start}^{index\_end}
\]
```

ce qui donne :

$$\sum_{i=index\_start}^{index\_end}$$

MaTeX génèrera ce code pour une sommation par défaut. Bien sûr, il est possible de passer des paramètres en plus, chose que nous étudierons plus loin. Avant de continuer avec d'autres exemples plus intéressants, voici comment MaTeX sera structuré.

1.3.2 Fonctionnement général

Comme MaTeX est un langage qui sert à générer un autre langage de manière littérale, son code source sera étranger au langage. En effet, on utilise le commentaire de LaTeX pour changer de lexeur dans la grammaire du langage MaTeX. Pour distinguer entre un commentaire LaTeX et le code source, on doublera le caractère du commentaire LaTeX. Les commentaires LaTeX commencent par le caractère '%' et se terminent à la fin de la ligne. Donc le code source LaTeX commencera par '%%' et se terminera par un point virgule. Après le point virgule, le lexeur LaTeX revient en fonction et continu

l'analyse. Donc il y a deux lexeurs, un pour LaTeX et un pour MaTeX. Tout le code LaTeX sera copié textuellement et les instructions MaTeX seront transformées en code LaTeX puis ajoutées au fichier. Cette manière de procéder a un avantage très important ; il permet de tout faire dans le même fichier. L'utilisateur pourrait simplement ajouter le compilateur MaTeX avant celui de LaTeX et remplacer le fichier source originale. Ce qui est vraiment pratique pour un utilisateur. Toutefois, la version actuelle génère le code dans un autre fichier appeler "output.tex".

1.3.3 Retour sur l'exemple

Dans l'exemple ci-dessus, on remarque qu'il y a une chaîne qui est répétée souvent : $\lfloor log_2 b \rfloor$. Le code LaTeX donne :

```
\lfloor log_{2} b \rfloor
```

Imaginez que vous devez retaper cela des dizaine de fois au travers un fichier. MaTeX possède un mécanisme simple pour accélérer l'écriture. Voici comment cela serait exprimé en MaTeX :

```
%% var lgfloor = "\lfloor log_{2} b \rfloor";
```

Quand MaTeX rencontrera cette instruction, il la recopiera dans le fichier sortie pour quelle soit de nouveaux accessible. Cela constitue la manière de déclarer une variable. Lorsqu'on voudra faire un appel à cette variable, on écrira "%% lgfloor ;" puis lors de la compilation cela sera substitué par la chaine entre les guillemets. Par exemple :

```
\[ %% Sum(); %% lgfloor;*%% lgfloor; \]
```

génère :

```
\[ \sum\limits_{i = 0}^{\infty}
\lfloor log_{2} b \rfloor *
\lfloor log_{2} b \rfloor  \]
```

Bien qu'il s'agit d'exemple trivial, on peut voir les avantages et le fonctionnement de MaTeX.

1.3.4 Exemple 2

Avant de regarder la syntaxe du code plus en profondeur, nous allons développer le langage avec d'autres exemples. Voici une autre problématique avec LaTeX qui survient lorsqu'on manipule des structures physiques. Voici du code source LaTeX et essayez de visualiser rapidement la structure que cela génèrera :

```
\[\begin{array}{rcl} \left[ \begin {array}{cc} a+b&a\\ \noalign{\medskip}a&b
\end {array} \right]  \left[ \begin {array}{cc} c+d&c\\ \noalign{\medskip}c&d
\end {array} \right] & = &  \left[ \begin {array}{cc}  \left( a+b \right)
\left( c+d \right) +ac& \left( a+b \right) c+ad\\ \noalign{\medskip}a \left(
c+d \right) +bc&ac+bd\end {array} \right] \\ \end{array}\]\\
```

Une autre horreur signée LaTeX. Voici le résultat une fois compilé :

$$\begin{bmatrix} a+b & a \\ a & b \end{bmatrix} \begin{bmatrix} c+d & c \\ c & d \end{bmatrix} = \begin{bmatrix} (a+b)(c+d)+ac & (a+b)c+ad \\ a(c+d)+bc & ac+bd \end{bmatrix}$$

Le problème provient du fait que LaTeX doit contenir beaucoup d'information concernant la structure de la matrice. Lorsqu'on travail sur des matrices et que l'on doit modifier un terme ou que nous effectuons les calculs directement dans l'éditeur, cela devient rapidement ardu.

Pour générer un matrice avec MaTeX, il suffit d'utiliser l'instruction prévue. Par exemple pour obtenir une matrice identité de taille 10 en MaTeX, on écrit :

```
\[ %%MatrixI(10);
\]
```

Ce qui génère :

```
\[
% AUTOMATIC CODE GENERERATION FOR: MatrixI(10);
\left[ {\begin{array}{cccccccccc}
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0  \\
  0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0  \\
  0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0  \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0  \\
  0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0  \\
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0  \\
  0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0  \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0  \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0  \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1  \\
\end{array} } \right]% END OF AUTOMATIC CODE GENERERATION
\]
```

Voici le rendu :

$$\left[\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

Pour générer le code ci-haut nous aurions besoin d'une fonction qui permet de placer les matrices dans l'espace puis d'une autre fonction qui permettrait de générer des matrices vides. Il faudrait alors remplir la matrice dans LaTeX, soit directement ou avec d'autres fonctions MaTeX.

2 Structures

Pour activer MaTeX, le fichier devra avoir une entête soit : "%% use_matex ;" sur la première ligne du fichier. Sinon, MaTeX ne fera aucun traitement. Après l'entête voici les structures de bases du langage qui seront explicitées chacune en détail dans les sections suivantes.

— Les fonctions primitives.
— Les fonctions définissables de manière dynamique.
— Les variables.

Pour chacun de ces éléments, nous allons maintenant faire une liste des cas possibles ainsi qu'une description de leur rôle.

2.1 Fonctions primitives

Les fonctions primitives du langage en seront une partie cruciale de MaTeX. En effet, plus elle sera riche plus MaTeX sera utile pour la composition de code source LaTeX. Elles serviront à rapidement décrire une structure que l'on souhaite avoir et y mettre des valeurs importantes. Elles sont composées par un mot clef réservé puis par des parenthèses qui contiendront des valeurs. Plusieurs vérifications sémantiques seront faites pour vérifier que les paramètres sont compatibles avec la fonction.

— Sum() : créer une somme par défaut.
— Matrix(taille , taille) : créer une matrice rectangulaire.
— MatrixI(taille) : créer une matrice identité.

— Integral(bornesup , borneinf, expression , variable) : créer une intégrale.
— ...

2.2 Fonctions dynamiques

Les fonctions primitives ne suffiront dans toutes les situations. C'est pourquoi il serait utile de pouvoir créer des fonctions avec un comportement identique, mais créer dynamiquement. Éventuellement, les fonctions dynamiques qui seront les plus pertinentes pourront être intégrées en tant que fonction primitive. Voici un exemple de déclaration d'une fonction dynamique :

```
%% fun itemize2(5) = "
%\begin{itemize}
% \item \"$arg[1]$\"
% \item $arg[1]$
% \item $arg[2]$

%\end{itemize}
$arg[3] \\
arg[4]
$
arg[5]
%";
```

Ce qui génèrera :

```
%% fun itemize2(5) = "
%\begin{itemize}
% \item \"$arg[1]$\"
% \item $arg[1]$
% \item $arg[2]$
%
%\end{itemize}
%$arg[3] \\
%arg[4]
%$
%arg[5]
%";
```

La seule différence entre le code avant et après la compilation est que MaTEX a ajouté des signes de commentaires au début de chaque lignes. Ceci a pour but de laisser l'usager voir le rendu avant de l'intégrer à MaTEX. Le code est recopié pour réutiliser la fonction à chaque appel. Le nombre 5 entre parenthèses après l'identifiant sert à vérifier qu'un bon nombre de paramètres est utilisé dans la fonction dynamique. On

permet un maximum de 10 paramètres. Lorsqu'on appellera la fonction, les paramètres seront substitués dans l'ordre de la signature de la fonction a endroit indiqué dans la définition de la fonction dynamique. Ces endroits sont désignés par arg[1], arg[2] , ... , arg[10] dans le code. Voici maintenant un exemple d'appel de la fonction dynamique :

```
%% var test2 = MatrixI(1);
%% var allo = "allo";
%% itemize2(lgfloor,"alslo",test2,Sum(),"OK");
```

Ce qui générera :

```
%% var test2 = MatrixI(1);
%% var allo = "allo";

\begin{itemize}
\item "$\lfloor log_{2} b \rfloor$"
\item $\lfloor log_{2} b \rfloor$
\item $alslo$

\end{itemize}
$
% AUTOMATIC CODE GENERERATION FOR: MatrixI(1);
\left[ {\begin{array}{c}
1  \\
\end{array} } \right]
% END OF AUTOMATIC CODE GENERERATION \\
\sum\limits_{i = index_start}^{index_end}
$
OK
```

Voici le rendu :

— "$\lfloor log_2 b \rfloor$"
— $\lfloor log_2 b \rfloor$
— $alslo$

$$\left[1 \right] \sum_{i=index_start}^{index_end} \text{OK}$$

On voit dans cet exemple que l'on peut passer en paramètre soit un identifiant de variable, une fonction primitive ou une chaine de caractère. Cette fonctionnalité sera vraiment intéressante pour un usager désirant faire ses propres fonctions. Voici les signatures pour l'appel et la définition des fonctions dynamique :

— fun indentifiant(nombre)= ¨du codeLaTeX ¨ ; Pour la dénition.
— id (arg[1], arg[2], ...) ; Pour l'appel.

2.3 Les variables

Les variables internes permettent de remplacer rapidement une instruction MaTEX par du code source LATEX. Le code source MaTEX est conservé dans le document originale lors de la génération. Les identifiants sont uniques car le compilateur vérifie s'il y a duplication de variable. Le code, pour définir un variable, est recopié dans le fichier pour pouvoir le réutiliser. Les variables peuvent contenir soit une fonction primitive ou une chaine de caractère.

3 Conclusion

La difficulté principale de la conception du langage réside surtout dans le fait de vouloir à la fois un langage simple, mais aussi performant au niveau de la génération d'erreur sémantique. Il est dangereux de faire du copier-coller surtout pour du code source LATEX car, les erreurs générés sont de piètre qualité. Il devient avantageux de générer le code avec MaTEX. Maintenant que nous avons une manière simple, rapide et efficace pour générer du code LATEX, il suffit d'y ajouter le plus de fonctions primitives possibles pour qu'il devienne attrayant. Les fonctions dynamiques sont aussi bien défini et offre une flexibilité à l'usager.