CAHIERS GUTenberg

LUATEX: VUE D'ENSEMBLE

Paul ISAMBERT

Cahiers GUTenberg, nº 54-55 (2010), p. 3-12.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2010___54-55_3_0>

LuaTeX: vue d'ensemble

Paul Isambert

1. Introduction

Quand Donald Knuth a inventé TeX, il ne s'agissait au départ que de mettre en page ses propres ouvrages ; ceux-ci étant écrits en anglais, bien des problèmes ne se posaient pas : direction de l'écriture, correspondance entre les caractères et les glyphes (le codage)… Qui plus est, la typographie numérique existait à peine, et ce que nous prenons aujourd'hui pour acquis (fontes, langages de description de page…) était à inventer : d'où, par exemple, les fontes TFM et les fichiers DVI.

L'utilisation de TeX se répandant, les limitations sont apparues : d'abord, l'ASCII ne pouvait pas suffire, et TeX est passé, à la fin des années 80, de 7 à 8 bits pour accommoder d'autres codages et permettre par exemple d'entrer des caractères accentués tels quels dans les fichiers sources. La typographie numérique évoluait aussi, et des standards se sont imposés pour le document électronique (PostScript, puis PDF) et les fontes qui le composent (fontes PostScript, TrueType, OpenType, parmi d'autres) ; avec l'apparition d'Unicode, même le codage 8 bits est devenu insuffisant.

Au fil des années, un certain nombre d'extensions de TeX sont nées pour tenter de résoudre ces problèmes : parmi les plus connues, ε-TeX préparait la naissance d'un « New Typesetting System » (qui n'a jamais vu le jour) en étendant les capacités et la syntaxe de TeX ; PDFTeX a permis

Cahiers GUTenberg nº 54-55 — Octobre 2010, p. 3-12.

de produire directement des fichiers PDF (avec des fontes PostScript T1) sans passer par un fichier DVI intermédiaire ; XeTeX enfin a fait entrer TeX dans le monde d'Unicode et des polices modernes.

Qu'apporte LuaTeX dans ce contexte ? Les évolutions précédentes (que LuaTeX intègre bien sûr) sont venues de l'extérieur : TeX s'est seulement adapté à certains contextes d'utilisation. LuaTeX entend modifier le programme plus en profondeur, sur deux points principaux : d'abord en embarquant un nouveau langage (Lua, un « vrai » langage, diraient certains) qui rend la programmation extrêmement aisée ; ensuite en « ouvrant » TeX grâce aux *callbacks*, c'est-à-dire en donnant accès à des opérations jusque là cachées (construction d'un paragraphe, lecture d'un fichier…) qu'on peut réécrire comme on l'entend.

Le reste de cette introduction passe en revue certain des points forts du nouveau moteur.

2. Codage

Comme XeTeX, LuaTeX est orienté vers Unicode. Il l'est en fait bien plus que XeTeX, car si ce dernier accepte divers types de codages (qu'ils relèvent d'Unicode ou pas), LuaTeX n'en autorise qu'un : UTF-8. Et impossible de tricher : alors que TeX3 tolère toute suite d'octets, et qu'il appartient à l'utilisateur de lui faire comprendre ce que ces octets signifient, LuaTeX produit une erreur dès qu'une suite d'octet n'est pas valide en UTF-8. D'ailleurs, tout document produit par une version antérieure de TeX, à moins qu'il ne se limite à l'ASCII (un sous-ensemble d'UTF-8), est invalide en LuaTeX.

Cela semble, à première vue, terriblement limité. Après tout, d'innombrables personnes produisent des documents qui ne sont pas codés en UTF-8, ne serait-ce que parce que tous les éditeurs de texte ne le font pas. Mais cette limitation n'est qu'apparente. Parmi les opérations inaccessibles que LuaTeX permet maintenant d'atteindre, il en existe une que sans doute peu d'utilisateurs ont jamais voulu contrôler, mais qui risque maintenant de se trouver au centre d'une grande attention : c'est l'opération qui consiste à passer une ligne d'*input* à TeX. Avec LuaTeX, il est possible d'intervenir à ce niveau, et de transformer la suite d'octets qui constitue une ligne réelle dans un fichier en une suite d'octets acceptable pour LuaTeX ; en d'autres termes, on peut transformer une ligne écrite

dans n'importe quel codage en une ligne écrite en UTF-8. Un exemple d'une telle manipulation est donnée à la section 5 ci-dessous.

3. Fontes

Comme XeTeX encore, LuaTeX permet d'utiliser les formats de fontes modernes que sont TrueType et OpenType. À la différence de XeTeX, cependant, il n'y a aucun chargement automatique des fontes (sauf les TFM) : LuaTeX lit un fichier de fonte et retourne à l'utilisateur une table représentant cette fonte, table qui doit encore être transformée en une autre de telle sorte que le programme puisse l'utiliser dans un document [1].

Cela peut sembler bien compliqué, et surtout bien inutile. À quoi bon devoir dire à LuaTeX que tel glyphe représente tel caractère, à quoi bon devoir devoir mettre en place soi-même les fonctionnalités Opentype (*features* : ligatures automatiques, petites capitales…) que souhaite employer un utilisateur ? LuaTeX ne le ferait-il pas mieux tout seul ?

Ces questions sont légitimes, mais ignorent ce qui fait la particularité et l'intérêt de LuaTeX : il ne s'agit pas d'apporter des solutions, mais des outils. En cela il est pleinement dans la lignée de TeX, programme quasi inutilisable tel quel mais offrant des briques de construction avec lesquelles il revient à l'utilisateur de bâtir sa maison (généralement via les éléments préfabriqués fournis par les formats).

Qu'on veuille augmenter artificiellement la chasse des caractères (produisant une version un peu primitive de l'interlettrage), mélanger plusieurs polices dans une fonte virtuelle, retoucher les paires de crénage, utiliser une correspondance caractères-glyphes particulière, voire corriger les fontes mal conçues (il y en a !), tout cela n'est possible que parce que le chargement des fontes n'est pas automatisé.

4. Lua

Comme son nom l'indique, LuaTeX incorpore un langage de programmation, Lua. Bien qu'on ne puisse mélanger les deux langages TeX et

1. Cette transformation est longue et complexe, mais que le lecteur se rassure, du code a déjà été écrit à cet usage : il s'agit du package `luaotfload`, qui adapte pour plain TeX et LaTeX le chargeur de fontes du format ConTeXt.

Lua, on peut passer du premier au second avec (principalement) la commande `\directlua`, et du second au premier avec (principalement) la function `tex.print`. Par exemple :

```
${3 \over 2} = \directlua{tex.print(3/2)}$
```

donne $\frac{3}{2} = 1.5$.

Une telle extension serait déjà beaucoup pour TEX : outre des opérations mathématiques que TEX n'a jamais su faire, Lua permet aussi de créer des index ou des bibliographies (puisque trier des données est très simple), sans parler des diverses facilités de programmation : boucles, tables, etc., tout cela sans le moindre `\expandafter`. L'article « LuaLATEX pour les non-sorciers, deux exemples » illustre une telle utilisation, montrant ainsi que si LuaTEX semble rendre TEX encore plus complexe, l'intégration d'un langage de programmation « digne de ce nom » en facilite aussi l'exploitation à bien des égards.

Mais Lua sait aussi manipuler les objets dont TEX se sert pour construire une page : ce qu'on appelle les nœuds (voir la section 6) ; c'est Lua aussi qui fait l'interface avec les « tripes » de TEX, et les opérations vues dans les sections précédentes, et plus généralement les *callbacks*, abordés dans la section suivante, se rédigent en Lua.

On pourrait avoir des réticences à apprendre un nouveau langage, surtout si on considère avoir déjà suffisamment peiné pour apprendre TEX. Mais Lua est facile à comprendre et à utiliser (une des raisons pour lesquelles il a été choisi) ; le maîtriser est donc assez aisé, ce qui est d'autant plus souhaitable que les opérations auxquelles il donne accès sont à l'inverse parfois complexes.

5. *Callbacks*

Qu'est-ce qu'un *callback*[2] ? C'est une porte donnant sur les opérations internes de TEX. On peut, avec un *callback*, enrichir ce que TEX fait lui-même, mais aussi le faire à sa place. Par exemple, on peut récupérer

2. « Fonction de rappel », dira le francophone ; nous gardons le terme *callback* parce qu'il est régulièrement utilisé, même par des locuteurs français, et que c'est aussi le nom de la bibliothèque Lua associée à ces opérations. D'ailleurs, on peut très bien savoir ce qu'est un *callback* en LuaTEX sans connaître plus généralement le concept de fonction de rappel en informatique.

et manipuler les lignes de texte produites par le constructeur de paragraphes — et aussi remplacer ce dernier par un code que l'on aura conçu soi-même.

Prenons l'exemple (beaucoup plus simple) du codage, et supposons que nous voulions écrire en ISO-8859-1 (familièrement appelé Latin-1 et contenant les caractères utilisés par un certain nombre de langues d'Europe de l'Ouest). Un codage est la représentation numérique d'un ensemble de caractères, nombres eux-même représentés aujourd'hui par des octets (un octet par nombre dans le cas de Latin-1, un ou plusieurs octet(s) dans le cas d'UTF-8). Passer d'un codage à un autre nécessite donc d'établir une correspondance entre deux ensembles de nombres, mais aussi d'avoir une traduction des octets vers les nombres (pour le codage source) et des nombres vers les octets (pour le codage cible). Dans le cas qui nous occupe, la correspondance numérique est immédiate, puisque les 256 caractères de Latin-1 sont les 256 premiers caractères d'Unicode, donc d'UTF-8. Il suffit alors de passer d'un octet à un nombre (ce qui est facile, puisque chaque octet est un nombre en représentation binaire), puis d'un nombre à un ou des octet(s). C'est ce que fait le code suivant :

```
latin1_vers_utf8 = function (character)
  return unicode.utf8.char(string.byte(character))
end

recodage = function (ligne)
  return string.gsub(ligne, ".", latin1_vers_utf8)
end

callback.register("process_input_buffer", recodage)
```

La première fonction, `latin1_vers_utf8`, prend un caractère, c'est-à-dire un octet, le transforme en nombre avec `string.byte`, traduit ce nombre en octets dans le codage UTF-8 avec `unicode.utf8.char`, et retourne le résultat. La deuxième fonction, `recodage`, prend une ligne et la retourne, mais seulement après avoir substitué à chaque caractère son correspondant UTF-8 via la fonction précédente, appliquée grâce à `string.gsub`, qui se lit comme suit : dans `ligne`, prendre chaque caractère (représenté par le point), et lui appliquer `latin1_vers_utf8`.

Enfin, puisque c'est ce qui nous intéresse ici, on enregistre `recodage` dans le *callback* appelé `process_input_buffer` ; celui-ci se déclenche à chaque fois que TEX lit une ligne dans le fichier source.

Il existe un grand nombre de callbacks, correspondant à autant d'opérations en TEX : trouver un fichier et le lire (par exemple avec `\input`), écrire vers un fichier extérieur (avec `\write`), charger une fonte (avec `\font`), intercepter les lexèmes (*tokens*) avant que TEX ne les traite, créer les ligatures, construire un paragraphe… Par défaut, ces callbacks ne sont pas définis et TEX travaille comme il l'a toujours fait, ce qui veut dire qu'on peut (heureusement !) utiliser LuaTEX sans avoir à se soucier de ces subtilités

6. Nœuds

TEX a toujours travaillé avec des nœuds (*nodes*) : il s'agit des briques qu'il manipule pour construire une page, qui n'est, dans les cas les plus simples, qu'une pile de boîtes horizontales séparées par des espaces (*glues*) ; les boîtes elles-mêmes sont faites essentiellement de glyphes et d'espaces. Tout cela, boîtes, espaces, glyphes, c'est ce qu'on appelle des nœuds.

Les nœuds ont des champs (*fields*) qui nous renseignent sur leur propriétés ; en premier lieu, tout nœud a un champ `id` qui indique son type (une boîte horizontale, un espace, un glyphe, mais aussi un *whatsit*, un tiret de césure, une pénalité…), et souvent un champ `subtype` qui précise sa nature (par exemple, un glyphe simple et une ligature sont tous deux des glyphes, mais se distinguent par leur sous-type). En fonction de ces deux champs, un nœud peut en avoir d'autres. Par exemple, un glyphe a les champs `char`, qui donne le numéro Unicode de son caractère, `font`, qui indique la fonte dont est tiré ce glyphe, `width`, `height` et `depth`, ses dimensions, etc.

Sauf si on le crée soi-même, on ne trouve jamais un nœud isolé : les nœuds vivent naturellement dans des listes. Par exemple, une boîte horizontale est une liste composée essentiellement de glyphes et d'espaces. Chaque nœud a un champ `prev`, qui pointe sur le nœud précédent dans la liste, et un champ `next`, qui point sur le suivant (ces deux champs ont la valeur `nil` pour le premier et le dernier nœud d'une liste, respectivement).

Par exemple :

```
\setbox0 = \hbox{des mots}
```

assigne au registre de boîte (*box register*) 0 une boîte horizontale, qui est elle-même un nœud de type `hlist`, et dont le contenu est une liste composée d'un premier nœud (qu'on appelle la tête, *head*) de type `glyph`, dont le champ `char` est 100 (le code de caractère de *d* en Unicode), dont le champ `prev` vaut `nil` et dont le champ `next` pointe sur le nœud suivant, etc.

L'accès aux nœuds rend extrêmement aisées des opérations typographiques quasi impossibles auparavant. N'importe quel glyphe, n'importe quelle ligne texte, peuvent être manipulés depuis leur création jusqu'au moment où ils sont écrits dans le fichier de sortie. Qu'il s'agisse de souligner du texte, le mettre en couleur, encadrer un bloc, balancer des colonnes, récupérer des inserts (comme des notes de bas de page) normalement inaccessibles parce qu'ils ne sont pas dans le bloc de texte principal, les possibilités semblent sans fin, et surtout l'interface, en Lua, est très simple.

On trouvera un exemple de manipulation de nœuds dans l'article « Ponctuation française avec LuaTEX ».

7. Attributs

Les attributs sont une innovation simple mais extrêmement puissante. En apparence, ce sont des compteurs, et c'est ainsi qu'on les manipule. Mais la différence est qu'à chaque fois que LuaTEX crée un nœud, celui-ci retient la valeur de chaque attribut. Par exemple

```
a\attribute0=10 b\attribute0=-"7FFFFFFF c
```

Ici, LuaTEX va créer trois nœuds, chacun contenant un glyphe. La valeur de chaque attribut est attachée à chacun, à savoir `-"7FFFFFFF` par défaut (`-2147483647` en notation décimale), ce qui équivaut à un attribut non-spécifié (*unset*), sauf pour le nœud `b`, qui a la valeur `10` pour l'attribut `0`, et qui va conserver cette valeur même si l'attribut en question change.

À quoi cela peut-il bien servir ? L'objectif est de « marquer » les nœuds afin de les identifier à une étape ultérieure, qui peut être très éloignée du moment où ces nœuds sont créés ; n'importe quel composant d'une page peut ainsi être repéré à n'importe quel moment, que ce soit juste après que le paragraphe a été construit ou au moment d'envoyer la

page au fichier de sortie. Une opération donnée peut être appelée à un certain moment mais réalisable (ou plus souhaitable) seulement après ; avec les attributs on peut signaler que tel nœud ou suite de nœuds devra recevoir un traitement particulier plus tard, au moment opportun. L'article « Attributs et couleurs » montre une application pour les couleurs.

8. Mathématiques

Pour la typographie des mathématiques, LuaTEX garde sa ligne de conduite : il offre de nombreux outils plutôt que des solutions, et donne accès à un grand nombre de paramètres auparavant intouchables, particulièrement en matière d'espacement. Ainsi peut-on contrôler le placement vertical des exposants et des indices, l'espace autour d'une barre de fraction, la hauteur de l'axe mathématique par rapport à la ligne de base, ou encore toutes les espaces possibles entre les huit types d'atomes servant à construire les formules. Par exemple, avec :

```
\Umathordbinspacing\displaystyle=8mu plus 4mu minus 8mu
\Umathbinordspacing\displaystyle=8mu plus 4mu minus 8mu
```

L'espace entre un « atome ordinaire » (un nombre, par exemple) et un « opérateur binaire » (comme le symbole +), et vice-versa, aura, dans une formule en style *display*, les valeurs indiquées plutôt que la valeur par défaut (indexée sur `\medmuskip`).

Le passage à Unicode touche aussi les mathématiques, et les primitives liées aux caractères, comme `\mathchardef` ou `\delimiter`, se voient adjoindre des équivalents Unicode comme `\Umathchardef` ou `\Udelimiter`.

On peut aussi signaler que la conversion opérée par TEX (et décrite dans l'annexe G du *TEXbook*) entre une liste mathématique et une liste horizontale est modifiable dans le *callback* `mlist_to_hlist` (et qu'une fonction associée existe, `node.mlist_to_hlist`) ; comme avec le constructeur de paragraphe, les opérations internes sont ici aussi pleinement accessible.

9. Bibliothèques

LuaTEX est équipé d'un certain nombre de bibliothèques (*libraries*) écrites en Lua, c'est-à-dire des ensembles de fonctions conçues avec un

objectif précis. Parmi les plus importantes, on notera :

`epdf`. — Une bibliothèque permettant d'explorer des documents PDF et d'en extraire les propriétés.

`kpse`. — Ces fonctions implémentent *kpathsea*, la bibliothèque externe utilisée par TEX pour trouver des fichiers ; on peut ainsi chercher soi-même des fichiers sans que TEX n'ait besoin de les lire, on peut créer ses propres messages d'erreur si un fichier n'existe pas, etc.

`mplib`. — LuaTEX intègre MetaPost, et on peut dessiner sans avoir à utiliser un programme externe.

`node`. — Cette bibliothèque contient toutes les fonctions utilisées dans la manipulation des nœuds.

`tex`. — Les paramètres, registres, boîtes, et autres, de TEX sont tous accessibles en Lua via cette bibliothèque. De nombreuses opérations peuvent ainsi être écrites plus simplement en Lua, surtout si elles impliquent de l'arithmétique (par exemple, déterminer la largeur du bloc de texte en fonction de la largeur de la page moins les marges).

`token`. — Ces fonctions permettent d'examiner, manipuler et créer des lexèmes (*tokens*), ces briques de l'utilisation de TEX.

10. Conclusion

Que va changer LuaTEX pour l'utilisateur ? Dans un premier temps, peut-être pas grand-chose : les fonctions les plus attendues (codages et fontes) sont déjà disponible avec XeTEX. Manipuler des nœuds, qui s'en soucie ? C'est aux auteurs de code qu'il va revenir d'être créatifs, et à n'en pas douter LuaTEX va ouvrir de nouvelles perspectives. Mais nous avons déjà remarqué que la simple intégration de Lua simplifiera bien des aspects de TEX, et qu'il n'est nullement besoin d'être expert pour en bénéficier.

Plus généralement, l'utilisateur de LATEX verra sans doute apparaître au fil des années de plus en plus d'extensions exploitant LuaTEX ; l'article « Un guide pour LuaLATEX » présente celles qui existent déjà (certaines fonctionnent en plain TEX).

Ceux qui utilisent ConTEXt, paradoxalement, peuvent ignorer complètement les mystères de LuaTEX, car c'est autour de ce moteur qu'est construit la dernière version (MkIV) de ce format, et qu'il y est déjà exploité à fond. En effet, deux des membres du noyau de l'équipe LuaTEX, Hans Hagen et Taco Hoekwater (le troisième étant Hartmut Henkel), sont

aussi respectivement le concepteur et l'un des principaux développeurs de ConTEXt.

Enfin, l'utilisateur de plain TEX, et tous les amateurs d'expérimentation en général, découvriront des possibilités si riches et des outils si puissants que TEX en sera complètement renouvelé.

◖ Paul Isambert
Université Sorbonne Nouvelle
Paris
zappathustra@free.fr