CAHIERS GUTenberg

PONCTUATION FRANÇAISE AVEC LUATEX

Paul ISAMBERT

Cahiers GUTenberg, nº 54-55 (2010), p. 87-100.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2010___54-55_87_0>

PONCTUATION FRANÇAISE AVEC LUATEX

Paul ISAMBERT

RÉSUMÉ. — Si TEX était l'œuvre d'un français, il existerait peut-être une primitive pour ajouter une espace avant la ponctuation haute (point d'interrogation, point d'exclamation, point-virgule, deux-points) comme le veut la tradition française – mais tel n'est pas le cas. LuaTEX n'apporte pas une telle primitive, mais il permet de manipuler les listes de caractères au cours de la composition du texte. Cet article se propose d'illustrer cela par la présentation d'algorithmes en Lua destinés à insérer des espaces à bon escient devant les signes de ponctuation qui le requièrent.

ABSTRACT. — If TEX had been created by a French man, maybe it would have a primitive dedicated to insert spaces before some punctuation signs (question mark, exclamation mark, colon, semicolon) as is usual in the French typographical tradition–but this wasn't the case. LuaTEX is not written by a French team either, but it enables handling character lists while texts are being typeset. The goal of this work is to illustrate its power by presenting Lua algorithms meant to insert the proper space before those signs that require it.

1. LE PROBLÈME

Il existe en TEX une primitive qui permet d'accommoder une particularité de la typographie anglaise : `\sfcode`, qui à chaque caractère peut attribuer une certaine valeur (le *space factor*) affectant l'espace qui suit (s'il y en a) ; ainsi après un point ou toute autre ponctuation forte l'espace peut-elle être plus large que la normale, comme il est souvent d'usage dans la typographie anglaise (bien que l'usage puisse être condamné).

Si TEX était l'œuvre d'un français, il existerait peut-être une primitive similaire ajoutant une espace avant la ponctuation haute (point d'interrogation, point d'exclamation, point-virgule, deux-points), comme le

Cahiers GUTenberg n° 54-55 — Octobre 2010, p. 87-100.

veut la tradition française – mais tel n'est pas le cas. LuaTEX n'apporte pas une telle primitive ; cependant, il permet de manipuler les listes de caractères et d'y ajouter une espace au besoin.

2. Solutions avec d'autres moteurs

Pour résoudre ce problème, les divers moteurs (TEX, PDFTEX, XETEX) offrent des possibilités différentes. Avec TEX, la solution est de rendre actifs les caractères concernés, les transformant ainsi en macros dont le développement produit une espace (un `\kern`) et le caractère (non actif). Par exemple, l'extension `babel` utilise le code suivant :

```
\declare@shorthand{french}{?}{%
  \ifhmode
    \ifdim\lastskip>\z@
      \unskip\penalty\@M\thinspace
    \else
      \FDP@thinspace
    \fi
  \fi
  \string?}
```

Ce code définit le point d'interrogation comme un caractère actif qui marche comme suit : si on est en mode horizontal, et si un blanc précède le signe de ponctuation (inséré à la main par l'utilisateur), on le retire (`\unskip`) et on ajoute une pénalité inviolable (`\penalty\@M`) et une espace fine (`\thinspace`) [1]. Si aucun caractère d'espace ne précède le point d'interrogation, on appelle la macro `\FDP@thinspace`, définie par défaut comme `\penalty\@M\thinspace` ; le résultat est donc identique au cas précédent, mais l'utilisateur est invité à redéfinir cette macro pour distinguer, s'il le souhaite, les ponctuations hautes précédées d'une espace dans le fichier source de celles qui ne le sont pas. Dans tous les cas, y compris celui où on n'était pas en mode horizontal, on ajoute une version non active du point d'interrogation.

1. La macro `\thinspace` est définie dans le noyau LATEX comme `\kern.16667em`, ce qui ne devrait nécessiter aucune pénalité, puisque TEX ne peut pas couper une ligne sur un kern (sauf s'il est suivi d'une *glue*, ce qui n'est pas le cas ici). Nous ignorons pourquoi une pénalité a néanmoins été ajoutée dans le code présenté ici ; peut-être pour permettre à l'utilisateur de redéfinir `\thinspace` comme une *glue*.

Cette solution, la plus ancienne et la plus répandue, souffre de plusieurs défauts : premièrement, les caractères actifs risquent toujours d'introduire des complications imprévues (développement à un moment inopportun, conflit possible avec un code utilisant les mêmes caractères mais avec d'autres codes de catégorie) ; deuxièmement, l'approche est trop rigide : par exemple, un point d'interrogation entre parenthèses sera précédé d'une espace (?), alors que tel ne devrait pas être le cas, l'ajout de l'espace dépendant du caractère qui précède.

Avec PDFTEX, il existe une primitive (pas documentée jusqu'à récemment) `\knbccode` (*kern before character code*), qui assigne à un caractère une certaine quantité d'espace (insécable) à ajouter à gauche (il faut aussi que le paramètre `\pdfprependkern` ait la valeur 1). Si elle évite le problème des caractères actifs, cette solution a la même rigidité que la précédente, et l'espace est ajoutée quel que soit le contexte. (Bien que LuaTEX soit le successeur de PDFTEX, cette primitive, comme d'autres, n'y a pas été reprise.)

C'est avec XETEX qu'apparaît une solution solide, grâce à l'utilisation des classes de caractères. On peut assigner à tout caractère une classe, et pour toute paire de classes on peut spécifier du code TEX à introduire automatiquement. Il suffit alors d'assigner l'essentiel des caractères à une classe *a*, la parenthèse gauche à une classe *b*, et les signes de ponctuation haute à une classe *c*, et demander qu'entre *a* et *c* soit introduit une espace, mais pas entre *b* et *c* ni entre *c* et *c*. On obtient ainsi l'effet escompté : une espace est insérée avant une ponctuation double, sauf si elle suit une parenthèse ouvrante ou une autre ponctuation haute (des combinaisons plus subtiles sont bien sûr possibles).

On notera une dernière possibilité, indépendante de TEX (et adoptée par Robert Bringhurst) : modifier l'approche des caractères directement dans la fonte ; sa radicalité la rend peu souhaitable, même si LuaTEX permet une solution similaire sans toucher à la fonte mais en insérant des paires de crénage au moment de son chargement. Cette solution ne marcherait cependant pas entre des glyphes de fontes différentes.

3. Nœuds

L'approche proposée ici est toute différente des précédentes : il s'agit d'examiner la liste de caractères et autres nœuds que TEX produit quand il s'apprête à construire un paragraphe (et aussi, d'ailleurs, quand il

construit une liste horizontale dans une `\hbox`), et d'insérer une espace seulement aux bons endroits. On évite ainsi le danger des caractères actifs, et on peut à loisir examiner le contexte [2].

Quand LuaTEX produit un paragraphe, il crée d'abord une liste de nœuds horizontale contenant essentiellement des glyphes et des espaces ; cette liste est soumise à trois opérations essentielles :

— des points de césure sont introduits (opération accessible dans le callback `hyphenate`) ;

— les ligatures sont formées (dans le callback `ligaturing`) ;

— le crénage des caractères est opéré (dans le callback `kerning`).

Ce n'est qu'après que ces trois opérations ont eu lieu que le paragraphe est construit. On peut illustrer ce qui se passe en prenant l'exemple d'un paragraphe ne contenant qu'un seul mot, *Vérification*, sans même de point final ; si on se restreint aux seuls nœuds représentant ce mot, la liste horizontale pourrait être dessinée comme suit, chaque carré représentant un nœud :

V é r i f i c a t i o n

Après l'insertion des points de césure, la liste devient :

V é - r i - f i - c a - t i o n

Après la formation des ligatures (les ligatures dépendent bien sûr de la fonte ; on indique ici une ligature fréquente) :

V é - r i - fi - c a - t i o n

Enfin, après le crénage, c'est-à-dire le rapprochement ou l'éloignement de deux glyphes contigus, produit en TEX par l'insertion de l'équivalent d'un `\kern` (le crénage dépend aussi de la fonte, mais le crénage entre *V* et *é* est assez courant) :

V *kern* é - r i - fi - c a - t i o n

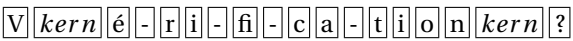
Ces représentations ne prétendent pas à l'exactitude : il faudrait indiquer la largeur (négative) du nœud de crénage ; quant aux nœuds de césure, ce

2. Ici nous ne distinguerons pas les ponctuations hautes précédées d'une espace dans le fichier source de celles qui ne le sont pas, par souci de simplicité ; il ne serait cependant pas difficile d'exécuter deux codes différents selon la présence ou non de l'espace.

ne sont pas des glyphes, ce que pourrait laisser penser la manière dont nous les avons représentés.

Imaginons maintenant que notre simple paragraphe se finisse par un point d'interrogation ; il a alors la représentation suivante :

Cela correspond à « Vérification? ». Or on cherche à obtenir « Vérification ? », c'est-à-dire :



L'opération à effectuer est donc similaire à celle du callback `kerning`, et pour cause : il s'agit d'un crénage entre deux caractères.

4. Le code

Nous devons donc faire la chose suivante : insérer des nœuds de crénage avant la ponctuation haute, sauf dans certains cas particuliers, c'est-à-dire quand le glyphe qui précède est une parenthèse ou un crochet ouvrants un point (dans le cas des points de suspension) ou une autre ponctuation haute [3]. En jargon LuaTEX, il s'agit d'insérer un nœud de crénage devant tout nœud de glyphe dont le code de caractère est 33 (point d'exclamation) 58 (deux-points), 59 (point-virgule), ou 63 (point d'interrogation), si et seulement si le nœud qui précède n'est pas un nœud de glyphe dont le code de caractère est l'une des valeurs mentionnées ou bien 40 (parenthèse ouvrante), 46 (point) ou 91 (crochet ouvrant). L'expression « nœud qui précède » doit être précisée : il s'agit du nœud qui précède, sauf si celui-ci est un nœud d'espace, qui peut provenir d'un document source tel que :

```
... vérification ?
```

3. Il y a peu de risque qu'une ponctuation haute suive un deux-points ou un point-virgule (pas plus qu'une virgule, par exemple), mais comme dans ce cas il faudrait quand même interdire le crénage, nous maintenons cette formulation, qui simplifie l'implémentation.

où l'auteur insère (consciemment ou inconsciemment) une espace devant le point d'interrogation pour imiter les règles de ponctuation française ; dans ce cas, il faut supprimer le nœud et recommencer l'opération, c'est-à-dire considérer le nœud qui précède [4]. Enfin, comme il est affaire de crénage, on insérera le code tout naturellement dans le callback `kerning`.

Commençons par définir les trois variables suivantes :

```
local GLUE = node.id("glue")
local KERN = node.id("kern")
local GLYF = node.id("glyph")
```

Les types de nœuds se distinguent par des valeurs numériques dans le champ `id` ; par exemple, un nœud d'espace (`glue`) a un champ `id` dont la valeur est 10, un nœud de crénage (`kern`) a la valeur 11 et un nœud de glyphe a la valeur 37. Cependant, ces valeurs risquent de changer d'ici la première version stable de LuaTEX ; c'est pourquoi on utilise la fonction `node.id`, qui prend un nom de nœud en argument et retourne un numéro d'`id` : les noms étant fixés, on est sûr que les variables `GLUE`, `KERN` et `GLYF` auront toujours la valeur associée aux types de nœud correspondants, quelle que soit cette valeur. Qui plus est, il est plus lisible de manipuler des variables avec un nom signifiant que des nombres.

On crée ensuite une table contenant les signes qui nécessitent un crénage. Cette table a deux fonctions :

— identifier les signes en question : ne seront considérés que les nœuds de glyphe dont le numéro de caractère correspond à un numéro d'entrée dans la table ;

— donner la largeur de l'espace voulue (en proportion de l'espace-mot de la fonte), qui n'est pas la même pour le deux-points et les autres ponctuations hautes ; cela sera la valeur de l'entrée.

La table a donc comme entrées des codes de caractères, obtenus grâce à la fonction `unicode.utf8.byte` (ici sous une forme « localisée », plus

4. Si on cherche un peu, on trouvera des complications ou cas particuliers qui ne nous semblent pas pertinents ; par exemple, étant donné :

```
... \hbox{vérification !} ?
```

il faudrait analyser la boîte (ou liste horizontale) précédant le point d'interrogation pour savoir si son dernier glyphe n'interdit pas le crénage (comme c'est le cas ici). Le risque d'une telle situation est cependant si réduit que nous l'ignorons ici.

simple à écrire et plus rapide)[5], et comme valeurs des nombres; créons aussi une seconde table identifiant les caractères (outre les ponctuations hautes elles-mêmes) qui, s'ils précèdent une ponctuation haute, interdisent le crénage, c'est-à-dire la parenthèse et le crochet ouvrants, et le point; pour cette table peu importent les valeurs des entrées, ce qui compte est que les entrées existent.

```
local byte = unicode.utf8.byte
local ponc_haute = {
  [byte(" ?")] = .4, [byte(" !")] = .4,
  [byte(" ;")] = .4, [byte(" :")] = 1 }
local pas_de_kern = {
  [byte("(")] = true, [byte("[")] = true,
  [byte(".")] = true }
```

Voici enfin la fonction qui insère les nœuds de crénage dans une liste de nœuds :

```
local function crenage (head)
  for item in node.traverse_id(GLYF, head) do
    if ponc_haute[item.char] then
      local prev = item.prev
      if not (prev.id == GLYF) or
       ( not ponc_haute[prev.char] and
         not pas_de_kern[prev.char] ) then
        if prev.id == GLUE then
          node.remove(head, prev)
        end
        local f = font.getfont(item.font)
        local kern = node.new(KERN, 1)
        kern.kern = ponc_haute[item.char]
                        * f.parameters.space
        node.insert_before(head, item, kern)
```

5. Cette fonction provient de la bibliothèque `slnunicode`, que LuaTEX intègre, et est équivalente à `string.byte`, plus couramment utilisée en Lua, sauf qu'elle comprend l'UTF-8 alors que `string.byte`, comme tout Lua, ne gère que Latin-1. Avec les caractères utilisés ici, cela ne fera aucune différence puisqu'ils relèvent de l'ASCII, dont le codage est identique en UTF-8 et Latin-1.

```
      end
    end
  end
  return head
end
```

Cette fonction se lit comme suit : on traverse une liste de nœuds (dénotée par sa tête `head`), mais en ne considérant, grâce à `node.traverse_id`, que les nœuds de glyphe (ceux dont le champ `id` a la même valeur que la variable `GLYF`) ; parmi ceux-ci, on ne s'intéresse qu'aux ponctuations hautes, c'est-à-dire ceux dont la code de caractère (champ `char`) correspond à une entrée dans la table `ponc_haute` (c'est la première fonction de cette table) ; enfin, on se restreint aux cas où le nœud précédent[6] n'est pas un glyphe, ou bien, si c'est un glyphe, son numéro de caractère ne correspond pas à une entrée dans les tables `ponc_haute` et `pas_de_kern`[7]. Ce n'est que si toutes ces conditions sont remplies qu'on insérera un nœud de crénage.

Avant de faire cette insertion, on vérifie si le nœud précédent est une espace, auquel cas on le retire de la liste avec `node.remove`. Ensuite, on

6. Le lecteur qui connaît déjà un peu LuaTEX, ou qui a exercé ses capacités de déduction, se demandera peut-être ce qui se passe si le nœud que nous considérons est le premier de la liste : le code sera-t-il invalide ? Dans l'absolu, ce serait le cas ; LuaTEX produira une erreur pour, par exemple, `prev.id` si `prev` vaut `nil` (c'est-à-dire qu'il n'y avait pas de nœud précédent). Cependant, cette situation est impossible ici, pour la raison suivante : quand LuaTEX passe une liste de nœuds aux callbacks `hyphenate`, `ligaturing` et `kerning`, il ajoute automatiquement un nœud temporaire en tête de liste, si bien qu'on n'a jamais à se soucier de savoir s'il y a un bien un nœud avant celui que nous considérons (étant entendu que nous ne considérons jamais ce nœud temporaire, puisque ce n'est pas un nœud de glyphe).

Pour la même raison, il est inutile de réassigner à `head` la tête de liste retournée par les fonctions `node.remove` et `node.insert_before` ; et la fonction `crenage` elle-même ne retourne rien : la tête de la liste est le nœud temporaire, aucunement affecté par les opérations effectuées ici, et ne risque donc pas de changer ; dans toute autre situation (c'est-à-dire en dehors des callbacks mentionnés), cette réassignation serait impérative.

7. En Lua, l'opérateur `and` ayant priorité sur l'opérateur `or`, les expressions `A or B and C` et `A or (B and C)` sont équivalentes ; nous n'utilisons donc les parenthèses que par souci de lisibilité.

récupère la fonte du glyphe concerné, et on crée un nœud de crénage[8] dont la largeur (champ `kern`) est celle de l'espace intermot telle que définie par la fonte[9], pondérée par la valeur de l'entrée correspondant au code de caractère du glyphe dans `ponc_haute` (c'est la deuxième fonction de cette table). Enfin, on insère le nœud de crénage devant le nœud de glyphe dans la liste avec `node.insert_before`.

Il ne nous reste plus qu'à ajouter cette fonction dans le callback `kerning`. Cependant, celui-ci insère par défaut des nœuds de crénage, et comme tous les callbacks, quand on y enregistre une fonction ces opérations par défaut ne sont plus faites ; il faut donc les refaire soi-même, mais il existe heureusement la fonction `node.kerning`, qui justement insère des nœuds de crénage dans une liste de nœuds. On enregistre donc une fonction anonyme contenant les deux fonctions.

8. La fonction `node.new` crée un nœud dont le type (ou `id`) est le premier argument passé, et le sous-type (champ `subtype`, pas présent pour tous les types de nœuds) est le second argument. Ce sous-type n'est pas obligatoire, sauf si on crée un nœud de type `whatsit` ; cependant, dans notre cas, il est important d'indiquer le sous-type 1, car le sous-type 0 indique un nœud de crénage issu du crénage des caractères tel que défini par la fonte, nœud qui peut être remis à zéro si le paragraphe est justifié avec expansion des glyphes. Il n'empêche (à la différence d'un nœud `whatsit`) que nous aurions pu modifier la valeur du champ `subtype` après-coup plutôt qu'en appelant `node.new`.

9. La table `parameters` de la fonte contient sept valeurs équivalentes à celles accessibles avec `\fontdimen` ; cependant, la primitive dénote les valeurs par des nombres, la table Lua par des noms ; voici l'équivalence (« espace » signifie ici « espace intermot ») :

Clé `parameters`	Num. `\fontdimen`	Explication
slant	1	Pente de la fonte.
space	2	Taille naturelle de l'espace.
space_stretch	3	Largeur maximale de l'étirement de l'espace.
space_shrink	4	Largeur maximale du rétrécissement de l'espace.
x_height	5	Hauteur d'œil ; valeur de l'unité `ex`.
quad	6	Cadratin ; valeur de l'unité `em`.
extra_space	7	Largeur additionnelle de l'espace quand `\spacefactor` est supérieur à 2000.

```
callback.register("kerning",
  function (head)
    node.kerning(head)
    crenage(head)
  end)
```

Le code est maintenant terminé. Le lecteur se sera peut-être demandé à quoi servait `local`, qui apparaît régulièrement dans le code ci-dessus. Cela permet que les définitions restent circonscrites au fichier courant, ou branche de conditionnel, ou corps de fonction, dans lesquels elles sont données ; par exemple, dans la fonction `crenage`, la déclaration

```
local prev = item.prev
```

limite la portée de `prev` à la structure conditionnelle où cette variable apparaît. Pareillement, `GLUE` et les deux autres variables (ainsi que les deux tables) ne seront définies au mieux que pour le fichier (ou `\directlua`) dans lequel apparaît le code. Cela signifie aussi que la fonction `crenage` a une durée de vie limitée, ce qui pourrait paraître étrange puisqu'elle sera utilisée dans tout un document ; mais `callback.register` en fait une copie et utilise ainsi la fonction que nous avons créée sans en passer par la variable `crenage` (qui pourrait très bien être redéfinie à la fin de notre code).

5. Améliorations

Le code présenté ici a surtout pour but d'illustrer l'une des nouveautés majeures de LuaTEX : non pas la construction de listes de nœuds (TEX l'a toujours fait), mais la possibilité de les manipuler. En conséquence, le code est simple et pourrait être amélioré ; voici deux idées.

Pour commencer, ce code pourrait bien sûr être étendu aux guillemets ; la situation est en fait plus simple : on ajoute, sans avoir à contrôler le contexte, une espace après un guillemet ouvrant et une espace avant un guillemet fermant [10]. Il n'y a en effet aucun cas particulier où cette espace ne soit pas présente, l'insertion peut donc être automatique. À

10. On pourrait alors remplacer la table `ponc_haute` par une table plus générale, qui contiendrait aussi les guillemets, et dont les valeurs ne seraient plus des nombres mais des sous-tables contenant plusieurs informations : les nombres que nous avons déjà donnés, bien sûr, mais aussi le type de ponctuation, afin de distinguer les ponctuations hautes, qui nécessitent d'examiner le contexte, ainsi que le guillemet

l'inverse, pour les ponctuations hautes, on pourrait affiner le contexte et annuler l'insertion de l'espace si, par exemple, le nœud précédent était un nœud de crénage, ce qui permettrait à l'utilisateur d'utiliser `\kern` pour gérer les cas particuliers.

Ensuite, quelle que soit la largeur de l'espace, il pourrait être intéressant que, comme l'espace intermot, elle puisse s'étirer et se réduire (la justification du paragraphe ne pourrait d'ailleurs qu'en bénéficier). Dans ce cas, ce n'est pas un nœud de crénage qu'il faudra ajouter, mais un nœud d'espace (équivalent à `\hskip`), précédé d'une pénalité infinie (c'est-à-dire dont la valeur est 10000 au moins), ou plutôt, pour parler Lua, précédé d'un nœud de pénalité dont le champ `penalty` vaut 10000 au moins. Plus précisément, l'espace avant le deux-points sera une espace intermot normale, tandis que l'espace avant les autres ponctuations hautes sera « semi-justifiante », avec la largeur naturelle que nous avons déjà donnée, une petite quantité d'étirement possible, mais pas de rétrécissement (la largeur naturelle est trop petite pour l'autoriser).

Voici la table `ponc_haute` ainsi modifiée ; les valeurs des entrées ne sont plus des nombres, mais des sous-tables contenant une valeur pour chaque composant de l'espace :

```
local ponc_haute = {
  [byte(" ?")] = {.4, .2, 0},
  [byte(" !")] = {.4, .2, 0},
  [byte(" ;")] = {.4, .2, 0},
  [byte(" :")] = { 1,  1, 1} }
```

La convention est la suivante : pour un caractère donné, la valeur de la première entrée de la table correspondante est la largeur naturelle de l'espace associée, la seconde entrée est son étirement, et la troisième son rétrécissement, ces valeurs étant données comme des facteurs à appliquer aux paramètres de l'espace intermot. On voit donc que le deux-points sera précédé d'une espace identique à l'espace intermot, tandis

gauche, qui insère une espace après le nœud concerné et non avant. À noter que dans ce cas il serait impératif d'utiliser la fonction `unicode.utf8.byte` et non `string.byte`, puisque les guillemets ne sont pas encodés pareillement en UTF-8 et en Latin-1.

que les autres ponctuations hautes prendront une espace avec une largeur naturelle valant 40% de largeur de l'espace intermot, un élargissement maximal de 20% l'élargissement maximal de l'espace intermot, et pas de rétrécissement.

Il nous faut maintenant remplacer les trois lignes suivantes dans le code précédent (rappelons que `f` dénote la fonte du glyphe considéré) :

```
local kern = node.new(KERN, 1)
kern.kern = ponc_haute[item.char] * f.parameters.space
node.insert_before(head, item, kern)
```

Nous devons remplacer le nœud de kern par un nœud d'espace (une *glue*, selon les termes de TEX), et ne pas oublier d'insérer aussi un nœud de pénalité. Les nœuds d'espace ont une particularité : ils n'ont pas de champs indiquant leurs dimensions, mais un champ pointant vers un nœud d'un autre type (`glue_spec`), lequel contient ces dimensions. Il nous faut donc créer deux nœuds [11], dont un d'un type que nous n'avons pas encore rencontré, et qui nécessite de définir la variable suivante, similaire à `GLUE`, `KERN` et `GLYF` :

```
local SPEC = node.id("glue_spec")
```

Par ailleurs, le nœud de pénalité que nous utiliserons sera toujours le même ; plutôt que de le recréer à chaque fois, il est alors plus simple de n'en faire qu'un exemplaire, dont nous utiliserons des copies. Rajoutons donc les deux lignes suivantes (qui devraient se passer d'explication) à la précédente :

```
local penalite = node.new(node.id("penalty"))
penalite.penalty = 10000
```

Voici comment nous allons créer un nœud d'espace avec son nœud associé. Le code qui suit remplacera la création et l'insertion du nœud `kern` dans la fonction `crenage` (rappelons qu'`item` est nœud de glyphe de ponctuation haute que nous considérons).

```
local espace = node.new(GLUE)
local spec   = node.new(SPEC)
```

11. Cette particularité disparaîtra sans doute de LuaTEX, et les champs du nœud `glue_spec` seront directement accessibles dans le nœud d'espace.

```
spec.width   = ponc_haute[item.char][1]
                    * f.parameters.space
spec.stretch = ponc_haute[item.char][2]
                    * f.parameters.space_stretch
spec.shrink  = ponc_haute[item.char][3]
                    * f.parameters.space_shrink
espace.spec  = spec
node.insert_before(head, item, espace)
node.insert_before(head, espace, node.copy(penalite))
```

L'assignation des valeurs de l'espace se fait dans le nœud `spec`, comme expliqué plus haut; pour chaque composant, on applique le facteur enregistré pour le caractère concerné à la valeur de ce composant tel que défini dans la fonte [12]. On assigne ensuite ce nœud `spec` au champ correspondant dans le nœud d'espace préalablement créé. Enfin, on insère ce nœud avant le signe de ponctuation (comme avec `kern` précédemment), et on lui ajoute une copie du nœud de pénalité. Ç'aurait été une erreur d'insérer `penalite` directement comme on utiliserait une variable; il faut imaginer les nœuds comme de vrais caractères en plomb : chaque nœud ne peut apparaître qu'à un endroit; c'est pourquoi on utilise `node.copy`, qui crée un nouveau nœud, quoiqu'identique [13].

6. Conclusion

On peut faire bien d'autres choses sur les listes de nœuds; c'est là par exemple que se fait la gestion des ligatures contextuelles et autres substitutions complexes. C'est la grande force de LuaTEX : non pas apporter des solutions, mais donner accès aux opérations fondamentales et aux atomes qu'elles manipulent, ici les glyphes, espaces, etc. D'ailleurs, une

12. Un nœud `glue_spec` a deux autres champs, `stretch_order` et `shrink_order`, indiquant l'ordre de l'étirement et du rétrécissement : 0 si l'étirement ou le rétrécissement sont finis, et de 1 (`fi`, infini hérité d'Omega et d'ordre inférieur à `fil`) à 4 (`filll`) s'ils sont infinis. Ici, comme nous n'assignons rien, la valeur 0 est utilisée.

13. Incidemment, le même raisonnement vaut pour les tables en Lua. Si on assigne une table à une variable `table1`, et qu'ensuite on déclare `table2 = table1`, travailler avec une variable ou l'autre a le même résultat, car les deux variables pointent vers la même table.

page (comme toutes les boîtes de TEX) est elle-même une liste de nœuds, et peut être manipulée comme nous avons ici manipulé les paragraphes en devenir. L'ajout d'une espace n'est guère différent (dans le principe!) de l'ajout d'un cadre ou d'une note en marge. Le temps où il fallait ruser – le temps des *dirty tricks* – est bien fini [14].

Paul Isambert
Université Sorbonne Nouvelle
Paris
zappathustra@free.fr

14. L'auteur de cet article s'enorgueillait d'apporter une solution efficace au problème du guillemet continu, et s'apprêtait fièrement à la présenter dans cet article… quand il s'est rendu compte que le problème était déjà réglé : LuaTEX reprend en partie le moteur Omega, dont la primitive `\localleftbox`. Celle-ci peut être utilisée (par exemple, au début d'une citation), et l'argument qu'on lui passe (par exemple, un guillemet) sera inséré au début de toutes les lignes commençant après le point où la primitive est appelée dans le paragraphe.