LUA(LA)TEX : COMMENT AVOIR ACCÈS À LA FABRICATION D'UN PARAGRAPHE

Comme nous l'a expliqué Didier Verna dans son exposé donné dans le cadre des exposés mensuels de GUTenberg[41] « *TEX est aujourd'hui encore un standard de-facto en matière de mise en forme typographique. Une part non négligeable de son succès est due à l'algorithme de justification de paragraphe dont il est équipé, le fameux "Knuth-Plass", conçu et développé entre 1977 et 1982, et que Donald Knuth lui-même a décrit comme "probablement l'algorithme le plus intéressant de TEX".* » Une des choses fabuleuses avec LuaTEX [1] est qu'il donne accès à quelques étapes de cet algorithme et permet d'en modifier le comportement. Pour un tour d'horizon, nous vous renvoyons au *Cahiers* dédié à LuaTEX [2], qui, malgré ses quelques années, reste une très bonne introduction.

Nous allons ici reprendre[42] en grande partie l'excellent article de Paul Isambert dans le *TUGboat* « LuaTEX : What it takes to make a paragraph » [3] en l'adaptant à LuaLATEX et à la situation en 2024. Cet article se situe entre le plagiat et la traduction donc…

Exécuter du code Lua

Avec le moteur LuaTEX, on a accès au langage Lua. Nous ne pouvons pas proposer ici une introduction à ce langage, et nous supposerons que les bases de programmation sont connues (la syntaxe n'est cependant pas trop difficile à comprendre).

La commande (primitive) du moteur LuaTEX[43] qui permet d'exécuter du code Lua dans un fichier source (LA)TEX est `\directlua`. L'argument de cette commande est le code Lua à exécuter. L'exemple classique d'utilisation de cette commande est le suivant :

Exemple 14

code

```
$\pi=\directlua{tex.print(math.pi)}$
```

résultat

π = 3.1415926535898

On peut ainsi très simplement afficher les valeurs de la suite de Fibonacci :

Exemple 15

code

```
\directlua{
function fibonacci (n)
    if n == 0 then
        return 0
    elseif n == 1 then
        return 1
    else
        return fibonacci(n-1) + fibonacci(n-2)
```

41. https://www.gutenberg-asso.fr/11-janvier-2024-Expose-sur-l-algorithme-de-Knuth-Plass.

42. Vous pouvez aussi visionner la vidéo d'un exposés mensuel GUTenberg que j'ai donnée le 12 septembre 2024 : https://www.gutenberg-asso.fr/12-septembre-2024-LuaLaTeX-introduction-a-l-utilisation-de-quelques-callbacks.

43. Et donc celle accessible peu importe le format : plain TEX, LATEX, etc.

code (suite)

```
    end
end
}

Le terme de rang $16$ de la suite de Fibonacci est :
\[u_{16}=\directlua{tex.print(fibonacci(16))}.\]
```

résultat

Le terme de rang 16 de la suite de Fibonacci est :

$$u_{16} = 987.$$

On voit au passage la fonction `print` Lua du module `tex` (et donc l'appel `tex.print`) qui permet de transmettre un affichage de Lua à TEX.

Comme il s'agit de code Lua dans un fichier source LATEX, on se doute bien qu'il peut y avoir des soucis dès qu'on utilise des caractères spéciaux de TEX dans le code Lua. Nous ne nous attarderons pas sur ces problèmes, cependant nous conseillons l'utilisation du package luacode [4] qui fournit des commandes et des environnements bien pratiques pour exécuter du code Lua en gérant peut-être de façon plus intuitive les divers développements et autres caractères spéciaux. De plus, luacode fournit deux commandes `\LuaCodeDebugOn` et `\LuaCodeDebugOff` qui permettent de faire afficher dans le fichier de log, le code Lua écrit dans le source TEX tel qu'il sera lu par l'interpréteur Lua. Ainsi, dans cet article, nous utiliserons principalement l'environnement `luacode` et sa variante étoilée pour les longs blocs de code Lua, et `\luadirect` et `\luaexec` pour les commandes analogues à `\directlua`. Sauf pour l'environnement `luacode*`, les arguments sont développables et donc on peut transférer l'effet d'une commande TEX à Lua.

Le tableau de la documentation de luacode fait un résumé des différences entre ces environnements et commandes :

	`\luadirect`	`\luaexec`	`luacode`	`luacode*`
Macros	Oui	Oui	Oui	Non
Simple contre-oblique	`\string\`	`\string\`	`\string\`	Simplement `\`
Double contre-oblique	`\string\\`	`\\`	`\\`	`\\`
Tilde	`\string~`	`~`	`~`	`~`
Dièse	`\string#`	`\#`	`\#` ou `#`	`#`
Pourcent	Pas de façon simple	`\%`	`\%` ou `%`	`%`
Commentaire TEX	Oui	Oui	Non	Non
Commentaire Lua	Non	Non	Oui	Oui

Exemple 16

code

```
\begin{luacode}
-- fonction Lua définie par récurrence pour le calcul
-- du terme de rang n de la suite de Fibonacci
function fibonacci(n)
    if (n==0) then
        return 0
```

code (suite)

```
    elseif (n==1) then
        return 1
    else
        return fibonacci(n-1)+fibonacci(n-2)
    end
end
\end{luacode}
\def\NN{16}
Le terme de rang $16$ de la suite de Fibonacci est :
\[u_{16}=\luaexec{% on affiche le résultat calculé par Lua
    tex.print(fibonacci(\NN))}.\]
```

résultat

Le terme de rang 16 de la suite de Fibonacci est :

$$u_{16} = 987.$$

Du source au paragraphe

Pour produire un paragraphe, de nombreuses étapes sont nécessaires : les caractères sont lus, interprétés, exécutés, les glyphes sont produits, les lignes sont mesurées, ajustées, etc. Avec LuaTEX, comme nous le disions, on peut avoir accès à toutes ces étapes, décortiquer ce qui se passe, et, dans une certaine mesure, en modifier le comportement. C'est ce que nous allons voir dans cet article.

Callbacks

Les *callbacks* sont des fonctions accessibles par l'utilisateur qui implémentent les opérations internes de TEX ; celles-ci peuvent être améliorées en les combinant avec d'autres fonctions *ad hoc*, mais on peut aussi les remplacer complètement. Tous les *callbacks* sont écrits en Lua.

Nous allons voir que, comme les fonctions enregistrées comme des *callbacks* interviennent dans une séquence d'actions qui constitue le traitement par le moteur TEX, beaucoup d'entre elles recevront des arguments (et doivent donc être définies pour les gérer) et, surtout, beaucoup sont censées renvoyer des valeurs. Les arguments passés, le cas échéant, et les valeurs retournées varient d'un *callback* à l'autre.

Alors qu'il existe de très nombreux *callbacks* pour énormément d'opération du moteur[44], nous ne nous intéresserons ici qu'aux principaux *callbacks* qui servent à la production d'un paragraphe :

`process_input_buffer` qui détermine comment TEX lit chaque ligne en entrée (source) ;

`hyphenate` qui détermine où et comment sont insérées les césures ;

`ligaturing` qui détermine où les ligatures doivent être produites ;

`kerning` qui détermine où les crénages sont insérés ;

`pre_linebreak_filter` qui permet de faire des opérations avant que le paragraphe soit construit ;

`linebreak_filter` qui construit le paragraphe ;

44. Pour trouver des fichiers, lire des fichiers, gérer l'affichage, et traiter des données : https://wiki.luatex.org/index.php/Callbacks.

post_linebreak_filter qui permet de faire des opérations après que le paragraphe est construit.

Si on veut connaître l'ensemble des *callbacks* fournis par LuaTEX, on peut utiliser la fonction Lua `callback.list()`. L'exemple suivant permet d'afficher la liste dans le terminal à la compilation du fichier avec LuaLATEX.

Exemple 17

```
\directlua{
    info = callback.list()
    for k, v in pairs(info) do
        print(k)
    end
}
```

Enregistrer un *callback*

Il y a deux types de *callbacks* : ceux qui étendent la fonctionnalité existante avec une nouvelle, et ceux qui (quand cela est possible) remplacent la fonctionnalité existante. Évidemment, lorsqu'on remplace la fonctionnalité existante, il est attendu que notre nouvelle définition fonctionne de façon similaire.

Pour enregistrer un des *callbacks* listés ci-dessus, on utilisera la syntaxe (Lua) suivante :

Exemple 18

```
id, error = callback.register(<string> callback_name, <function>
    func)
```

Si l'on souhaite rétablir le comportement par défaut, il suffira d'exécuter le code suivant :

Exemple 19

```
id, error = callback.register(<string> callback_name, nil)
```

L'argument à `nil` rétablira le comportement par défaut du *callback*.

Ces fonctions sont celles du moteur LuaTEX. L'utilisation du format LATEX peut poser quelques problèmes. David Carlisle et Joseph Wright ont développé le package ltluatex (présent dans le noyau LATEX) qui fournit des commandes TEX et des fonctions Lua adaptées au format. En particulier, on utilisera les fonctions, aux noms plus explicites :

Exemple 20

```
luatexbase.add_to_callback(<string> callback_name, <function>
    func, <string> description)
```

et

Exemple 21

```
luatexbase.remove_from_callback(<string> callback, <string>
    description)
```

Lire les lignes en entrée

Le *callback* `process_input_buffer` est exécuté lorsque TEX a besoin d'une ligne en entrée. Cette fonction Lua a donc un argument qui se trouve être la chaîne de caractères (le type Lua `string`) de la ligne et elle doit retourner une ligne, possiblement la même. Dans le fonctionnement par défaut de LuaTEX, rien ne se passe à cette étape et TEX lit la chaîne de caractères de la ligne du document source. À ce stade de la construction du paragraphe, la ligne n'a pas été traitée du tout : le contenu mis après un signe de commentaire est bien présent, les espaces multiples n'ont pas été réduites à des espaces uniques, etc. La chaîne de caractères contient donc exactement la ligne de texte du document source.

On peut se demander si avoir accès à ce *callback* est utile. Paul Isambert, dans son article, propose deux exemples d'utilisations : lire une entrée dans n'importe quel codage (UTF-8, ASCII, etc.) ; composer du texte verbatim. Nous allons étudier ici cette deuxième application.

Une des principales difficultés de l'écriture de texte verbatim avec TEX réside dans la gestion des *catcode*[45]. Tout devient encore plus compliqué lorsque l'on souhaite composer le texte verbatim et l'exécuter (on utilise souvent pour cela un fichier externe).

Avec le callback `process_input_buffer` tout devient plus simple. Les lignes du source peuvent être stockées et utilisées ensuite de différentes manières.

Nous allons voir qu'on peut définir un couple de commande `\myVerbatim` et `\Endverbatim`, la première enregistrant une nouvelle fonction Lua au *callback* qui stockera les lignes dans une table (Lua) jusqu'à ce que le *callback* soit réinitialisé lorsque la ligne `\Endverbatim` est rencontrée. La table ainsi construite pourra être utilisé soit pour afficher le code, soit pour l'exécuter.

Le code Lua permettant de stocker les lignes entre `\myVerbatim` et `\Endverbatim` est le suivant :

Exemple 22

```
local verb_table
local function store_lines (str)
    if str == "\\Endverbatim" then
        luatexbase.remove_from_callback("process_input_buffer",
            "Store lines of verbatim")
    else
        table.insert(verb_table, str)
    end
    return ""
end
```

Ce code est fait pour être exécuté avec l'environnement `luacode` et donc pour obtenir la contre-oblique de la macro TEX `\Endverbatim`, il nous faut utiliser `\string\`. On peut aussi utiliser l'environnement `luacode*` pour que le code Lua ne soit pas développé au sens de TEX.

45. La page « Que sont les catcodes ? » (https://faq.gutenberg-asso.fr/2_programmation/syntaxe/catcodes/que_sont_les_catcodes.html) nous explique ce que sont les *catcodes* : « plutôt que de définir des primitives pour des tâches aussi courantes que le passage en mode mathématique ou la mise en exposant ou en indice, Donald Knuth a préféré réserver certains caractères à ces tâches. Par exemple le dollar pour passer en mode mathématique, l'underscore pour passer en indice. D'autres caractères ont des significations particulières lorsqu'on écrit un document TEX : les accolades et crochets pour délimiter les arguments des commandes, ou simplement l'antislash pour appeler les commandes. Certains caractères sont différents des autres pour TEX. »

La fonction `store_lines` ajoute donc chaque ligne dans la table *globale* `verb_table` sauf si la ligne contient uniquement `\Endverbatim` (on pourrait aussi utiliser une *expression régulière* pour détecter `\Endverbatim` parmi du texte), auquel cas le *callback* est réinitialisé[46]. Très important : cette fonction retourne une chaîne de caractères vide car si elle ne retournait rien, LuaTEX traiterait cela comme si le *callback* n'avait pas été utilisé, et la ligne originale serait affichée.

Pour rendre cette fonction active dans le callback `process_input_buffer`, on va définir la fonction Lua `register_verbatim` suivante :

Exemple 23

```
function register_verbatim ()
    verb_table = {}
    luatexbase.add_to_callback("process_input_buffer",
        store_lines, "Store lines of verbatim")
end
```

Cette fonction réinitialise la table `verb_table` dans laquelle nous stockons les lignes de texte verbatim et appelle la fonction `callback.register` vue plus haut.

On va alors définir la macro `\myVerbatim` qui active le *callback* modifié :

Exemple 24

```
\newcommand\myVerbatim{\luadirect{register_verbatim()}}
```

Avec cette façon de faire, nous n'avons pas besoin de définir `\Endverbatim` car cette ligne sera « mangée » par notre *callback*, et donc jamais vue par TEX.

On va maintenant fabriquer une fonction Lua qui permet d'afficher les lignes stockées dans `verb_table` soit après traitement par (LA)TEX, soit verbatim.

Exemple 25

```
function print_lines (catcode)
    if catcode then
        tex.print(catcode, verb_table)
    else
        tex.print(verb_table)
    end
end
```

La fonction Lua `print` du module `tex`, fourni donc dans le moteur LuaTEX, accepte comme argument soit une chaîne de caractères soit une table de chaînes de caractères (ici notre `verb_table`). De plus, il est possible d'avoir un argument optionnel (un entier relatif) permettant de définir le régime de *catcode* à utiliser pour les caractères. Pour définir une table de *catcode* (le régime de *catcode*), on utilisera le code suivant :

46. La description `"Store lines of verbatim"` est celle qui a été utilisée pour enregistrer `store_lines` dans le *callback* `process_input_buffer` (voir plus bas). Ainsi, lorsque la fonction `store_lines` arrive à `\Endverbatim`, elle se supprime elle-même du *callback* `process_input_buffer` et n'est donc plus appelée quand la ligne suivante est lue.

Exemple 26

```
\newcatcodetable\verbcatcode
\newcommand\createcatcodes{
    \begingroup
    \catcode`\\ = 12
    \catcode`\{ = 12
    \catcode`\} = 12
    \catcode`\$ = 12
    \catcode`\& = 12
    \catcode`\# = 12
    \catcode`\^ = 12
    \catcode`\_ = 12
    \catcode`\% = 12
    \catcode`\  = 13
    \catcode`\^^M=13
    \savecatcodetable\verbcatcode
\endgroup}
\createcatcodes
```

Ici, on crée donc un contexte dans lequel tous les caractères spéciaux de TEX deviennent des caractères normaux. Les commandes `\newcatcodetable` et `\savecatcodetable` ont été utilisées pour créer cela. On déclare donc la table que l'on nomme `\verbcatcode` et on sauve les enregistrements de catcodes grâce à `\savecatcodetable`. Le passage par une commande `\createcatcodes` est une façon de gérer le fait que, dès lors qu'on utilise `\catcode`\\ =12` qui transforme donc la contre-oblique en caractère normal, on perd la possibilité d'écrire des commandes.

Ceci fait, il ne nous reste qu'à définir deux commandes : `\useverbatim` et `\printverbatim` pour respectivement produire le résultat du code et afficher le verbatim du code.

Exemple 27

```
\newcommand\useverbatim{%
    \luadirect{print_lines()}%
}
\newcommand\printverbatim{%
    \bgroup\parindent=0pt \ttfamily
    \luadirect{print_lines(luatexbase.catcodetables.verbcatcode)}
    \egroup
}
```

On remarquera que notre `\savecatcodetable\verbcatcode` a généré, du côté de Lua, la valeur (un entier) `luatexbase.catcodetables.verbcatcode`.

Une fois tout cela fait, on pourra utiliser notre machinerie comme l'illustre l'exemple suivant.

Exemple 28

code

```
\myVerbatim
\newcommand\myluatex{%
  Lua\kern-.01em\TeX
}%
```

code (suite)

```
\Endverbatim
\useverbatim % fabriquer vraiment la commande
Avec le code :\par
\printverbatim\par
on définit la commande permettant de générer \myluatex.
```

résultat

Avec le code :
\newcommand\myluatex{%
Lua\kern-.01em\TeX
}%
on définit la commande permettant de générer LuaTEX.

Tout cela ouvre la porte à de nombreuses améliorations. On peut imaginer, grâce à un paramètre ajouté aux commandes `\printverbatim` et `\useverbatim`, ne spécifier qu'une partie du code à afficher ou utiliser. On pourrait aussi, au moment de la composition du verbatim, entrecouper les lignes de verbatim de macros qui, elles, obéiraient à un régime classique de *catcode*.

Insérer des césures

Nous allons désormais nous intéresser aux nœuds que TEX a créés et concaténés dans une liste horizontale, c'est-à-dire, pour simplifier, la liste de mots et de blancs qui constituent le prochain paragraphe[47]. C'est à ce moment que la composition commence réellement. Le *callback* `hyphenate` reçoit une liste de nœuds (*nodes* en anglais) qui est le matériel de base à partir duquel est construit un paragraphe. Ce *callback* doit ajouter les points de césure, ce qu'il fait par défaut si aucune fonction n'a été enregistrée comme *callback*.

Dans ce *callback*, mais aussi dans d'autres, il est souvent intéressant de savoir quels nœuds sont passés en argument. Pour cela, voici une fonction Lua qui prend une liste de nœuds et qui affiche leur champs `id` (d'identification) dans le terminal et dans le fichier de journal `.log`[48] ou, si le nœud est un glyphe (`id=37`, mais il est préférable de récupérer la valeur avec la fonction `node.id`), affiche directement le caractère.

Exemple 29

```lua
local GLYF = node.id("glyph")
function show_nodes (head)
    local nodes = ""
    for item in node.traverse(head) do
        local i = item.id
        if i == GLYF then
            i = unicode.utf8.char(item.char)
        end
        nodes = nodes .. i .. " "
    end
    texio.write_nl(nodes)
```

47. Dans son article, Paul Isambert illustre avant l'utilisation du callback `token_filter`. Cependant, depuis l'écriture de cet article, ce callback a été enlevé de LuaTEX.

48. La correspondance entre nombre et type de nœud est spécifiée dans le chapitre 8 du manuel de référence de LuaTEX [1].

```
12  end
```

On va enregistrer cette fonction dans le callback `hyphenate` comme nous avons fait pour les callbacks précédemment. Ainsi, le code suivant :

Exemple 30

```
\luadirect{
    luatexbase.add_to_callback("hyphenate",show_nodes,"Show nodes
    ")
}
Est-ce effectif?

\luadirect{
    luatexbase.remove_from_callback("hyphenate","Show nodes")
}
```

produira dans le terminal la sortie suivante :

```
41 9 0 E s t - c e 12 e f f e c 7 t i f ?
```

Notre fonction montre donc que le callback `hyphenate` reçoit la liste :

41 9 0 E s t - c e 12 e f f e c 7 t i f ?

Le premier nœud, identifié par le nombre 41, est un nœud présent pour des raisons techniques qui ne nous intéressent pas ici. S'ensuit le nœud d'identification 9 qui est un *whatsit* (élément extraordinaire[49]), et son sous-type est 6 ce qui indique que c'est un *whatsit* `local_par` qui contient, parmi d'autres choses, la direction de composition du paragraphe (gauche-droite ici). Le troisième nœud est une liste horizontale (identificateur 0), c'est-à-dire une `hbox`. Son sous-type est 3 ce qui indique que c'est une boîte d'indentation, et si on demande sa largeur, la valeur sera celle de `\parindent` en *scaled point*.

Les nœuds représentant des caractères ont de nombreux champs parmi lesquels le champ `char` (un nombre) que notre fonction `show_nodes` utilise pour afficher quelque chose de plus parlant qu'un simple entier (grâce à la fonction `unicode.utf8.char()`). Les autres champs sont `width`, `height`, `depth` (des nombres aussi en *scaled points*) et `font` (encore un nombre car les fontes sont aussi représentées en interne par des nombres). Leurs sous-types nous intéresseront plus tard.

Enfin, les nœuds avec l'identificateur 12 sont des *glues*, qu'on appellera *ressorts* en français, c'est-à-dire l'espace entre deux mots et l'espace qui se trouve à la fin de ligne du paragraphe (qui n'est pas là si le dernier caractère est immédiatement suivi par un `\par` ou un signe de commentaire). Les ressorts sont, avec les boîtes et les dimensions, parmi les types d'objet de données dans TEX traditionnel qui ne sont pas des valeurs simples. Ils sont insérés lorsque TEX voit un espace dans le flux de texte, mais aussi par `\hskip` et `\vskip`. La structure qui représente les composants des ressorts est appelée `glue_spec`, et elle a les champs `width`, `stretch`, `stretch_order`, `shrink` et `shrink_order`.

Maintenant, que peut-on faire avec ce *callback*? Bien, tout d'abord, on peut insérer des points de césure dans notre liste de nœuds comme LuaTEX le fait par lui-même, si nous n'avions pas modifié le *callback*. La fonction `lang.hyphenate` permet de faire cela :

49. Voir le lexique français-anglais de la FAQ GUTenberg : https://faq.gutenberg-asso.fr/1_generalites/documentation/comment_traduire_ce_terme.html.

Exemple 31

```
\begin{luacode}
function myhyph (head, tail)
    lang.hyphenate(head)
    show_nodes(head)
end
\end{luacode}
\luadirect{
    luatexbase.add_to_callback("hyphenate", myhyph, "My Hyphenate
    ")
}
Est-ce effectif?

\luadirect{
    luatexbase.remove_from_callback("hyphenate", "My Hyphenate")
}
```

Le terminal nous montre alors la liste suivante :

41 9 0 E s t 7 c e 12 e f 7 f e c 7 t i f ?

Dans la fonction Lua `myhyph`, nous n'avons pas besoin de retourner une liste puisque LuaTEX s'occupe de cela dans à l'intérieur de ce *callback*, comme c'est aussi le cas dans les *callbacks* `ligaturing` et `kerning`. Aussi, ces trois *callbacks* prennent deux arguments : `head` et `tail`, repectivement le premier et dernier nœud de la liste à traiter. Le dernier nœud peut en général être ignoré.

Maintenant, regardons ce que l'insertion de points de césure a produit. Comme attendu, des éléments ont été ajoutés avec l'identification 7 : ce sont des nœuds de césure avec trois champs `pre`, `post` et `replace` qui sont les équivalents des premier, deuxième et troisième arguments de la macro `\discretionary`. Le `pre` est la liste des nœuds à insérer avant une coupure de ligne, le `post` est la liste des nœuds à insérer après une coupure de ligne, et le `replace` est la liste de nœuds à insérer si ce point de césure n'a pas été choisi. Ici, nous avons trois nœuds avec l'identificateur 7 :

entre `Est` et `ce` : le caractère initial est un tiret, et ainsi, le nœud devient un point de césure potentiel. Le nœud a donc un champ `replace` contenant le caractère `-` à afficher lorsqu'il n'y pas de coupure à cet endroit, et classiquement, un champ `pre` contenant lui aussi `-` (à n'afficher que lorsque la ligne se coupe à cet endroit) et un champ `post` vide.

entre les deux `f` et entre le `c` et le `t` : Ici, il s'agit de point de césure très classique, sans champ `replace`, un champ `pre` contenant `-`, et un champ `post` vide.

Un dernier mot sur les césures. Avec TEX, il est possible d'ajouter des mots dans le dictionnaire d'exception avec la commande `\hyphenation`. Sa syntaxe est simple, il suffit d'écrire les mots en indiquant les points de césures par un tiret :

Exemple 32

```
\hyphenation{man-u-script man-u-scripts ap-pen-dix}
```

Cette primitive a été étendue avec LuaTEX et permet désormais d'utiliser l'équivalent de la structure `pre`, `post` et `replace` en insérant à la place du simple tiret une séquence de trois groupes : `{pre}{post}{replace}`. Les utilisateurs allemands (et sans doute d'autres

nombreuses nationalités) seront ravis de ne plus avoir à donner une attention particulière à leur *backen* dans leur document : une simple déclaration comme la suivante devrait suffir :

Exemple 33

```
\hyphenation{ba{k-}{}{c}ken}
```

Enfin, comme on le voit avec `Est-ce`, avec un trait d'union en premier et troisième argument, on configure très simplement les points de césure des mots composés.

Ligatures

Comme son nom[50] l'indique, le *callback* `ligaturing` doit s'occuper d'insérer les ligatures (et ceci se produit par défaut si aucune fonction additionnelle n'est enregistrée au *callback*). Comme précédemment, on va utiliser notre fonction `show_nodes` juste après la fonction Lua `node.ligaturing` lors d'un enregistement dans le *callback* `ligaturing`. Cela donne le code suivant :

Exemple 34

```
\begin{luacode}
function mylig (head, tail)
    node.ligaturing(head)
    show_nodes(head)
end
\end{luacode}
\luadirect{
    luatexbase.add_to_callback("ligaturing", mylig, "My
    Ligaturing")
}
Est-ce effectif?

\luadirect{
    luatexbase.remove_from_callback("ligaturing", "My Ligaturing")

}
```

Cependant, ce code, écrit initialement pour LuaTEX par Paul Isambert, ne produit pas le résultat escompté avec LuaLATEX. En effet, LuaLATEX charge désormais par défaut une fonte OTF, et le travail de ligature est alors laissé à la fonte. Le callback `ligaturing` devient alors une coquille vide qui n'a pas vraiment d'utilité. Pour retrouver le fonctionnement de LuaTEX de base, on pourra (mais ce n'est pas à conseillé) forcer l'utilisation d'une fonte T1 avec le chargement du package fontenc :

Exemple 35

```
\usepackage[T1]{fontenc}
```

Si l'on fait cela, alors, nous aurons le résultat suivant : 41 9 0 E s t 7 c e 12 e 7 e c 7 t i f ? □

50. Même en anglais, cela se laisse entendre.

Que s'est-il passé ? Pourquoi *effectif* a-t-il été raccourci de la sorte ? Simplement parce qu'il y a une interaction entre la création des césures et la création des ligatures. Si le point de césure dans *effectif* est choisi, le résultat sera alors `ef-fectif`. Si le point n'est pas choisi alors le résultat sera `e<ff>ectif` où `<ff>` représente la ligature. En d'autres termes, la présence de ligatures dépend des césures. Ici, le nœud de césure a son champ `pre` égal à `f-`, `f` dans le champ `post` et `<ff>` dans le champ `replace`.

Les nœuds de ligatures sont des nœuds de glyphe avec un `subtype` qui vaut 2, alors qu'un nœud de glyphe classique a un `subtype` de 1. Ainsi, ils ont un champ spécial, `components`, qui pointe vers une liste de nœud composée des glyphes individuels qui constituent la ligature. Les composants de `<ff>` sont donc `f` et `f`. Les ligatures peuvent ainsi être décomposées quand cela est nécessaire.

Comment LuaTEX (soit avec le fonctionnement par défaut du *callback* `ligaturing`, soit avec la fonction Lua `node.ligaturing`) fait-il pour savoir quelle séquence de nœuds de glyphes doit-être transformée en ligature ? L'information est codée dans la fonte : LuaTEX regarde dans la table `ligatures` associée (si elle existe) à chaque caractère, et si le caractère suivant est présent dans cette table, alors la ligature est créée. Par exemple, pour le `f` de *Computer Modern Roman*, la table `ligatures` a une entrée à l'index 105, ce qui correspond au `i`, et qui contient la ligature `<fi>`. Ainsi, LuaTEX ne traite finalement pas autre chose que des ligatures de deux glyphes. La ligature `<ffi>` est le résultat d'une ligature entre `<ff>` et `i`.

Cependant, les fontes, et particulièrement les fontes OTF, peuvent définir des ligatures avec plus de deux glyphes ; par exemple, l'entrée `3/4` peut produire quelque chose comme ¾ (un glyphe unique). On peut choisir, lorsqu'on crée la fonte à partir du fichier OTF, de définir une ligature *fantôme* `<3/>` et de configurer ¾ comme la ligature entre notre ligature fantôme `<3/>` et `4`. Avec ce mécanisme LuaTEX peut gérer cela automatiquement. Il est sans doute plus élégant et générateur de moins d'erreurs, de gérer ce type de ligatures à la main, c'est-à-dire d'enregistrer une fonction dans le *callback* `ligaturing` qui à partir d'une suite de nœud, crée la ligature. C'est cependant plus lent.

C'est aussi dans ce *callback* que des choses comme les substitutions contextuelles peuvent être orchestrées. Par exemple, les formes initiales ou finales des glyphes, qui sont différentes par exemple en arabe ou dans quelques fontes latines sophistiquées, peuvent être gérées ici. En théorie, c'est assez simple : il suffit d'analyser le contexte d'un nœud particulier, c'est-à-dire les nœuds l'entourant ; si ceux-ci correspondent à un certain contexte d'une certaine forme, alors il suffit de l'utiliser. Par exemple, avec la fonte Minion (disponible avec Adobe Reader), et l'option OTF `ss02` activée, les `r` et les `e` suivis d'une espace (un nœud *glue*) peuvent être remplacés par leurs variantes finales. Dans la pratique, les choses sont un peu plus compliquées, en particulier parce que l'on doit lire ces contextes de substitution depuis les fichiers de fontes.

Cependant, on peut tout de même réaliser un type simple de substitution contextuelle. Le code utilisé pour charger une fonte avec LuaTEX[51] utilise souvent la fonctionnalité `trep` (inspirée de XETEX) pour que l'accent grave ` et les guillemets simples ' soient remplacés par les guillemets anglais simples gauche et droit. Cependant, on peut vouloir permettre une utilisation encore plus facile et utiliser les guillemets doubles droits " partout et qu'ensuite, suivant le contexte, les bons guillemets soient substitués.

Voici un code simple permettant de faire cela : si " est trouvé, le code le remplace par ' si le nœud précédent est un glyphe et que son caractère n'est pas une parenthèse ouvrante, et par ` sinon[52]. C'est ce que proposait Paul Isambert dans son article initial. Ici, nous

51. Et pas avec LuaLATEX et fontspec.

52. Ici, on a utilisé `not (x==y)` au lieu de `(x~=y)`, qui aurait été plus simple, pour éviter le problème du développement du caractère~.

allons le transformer légèrement pour utiliser des guillemets français. Évidemment, notre implémentation est donnée pour l'exemple : elle n'est absolument pas satisfaisante, puisqu'il faudrait ajouter des espaces fines insécables, permettre l'insertion d'espaces dans le source, etc.

Pour obtenir le code d'un caractère, il faut regarder la table unicode et convertir l'index écrit en hexadécimal en base 10. Ainsi, le double guillemets droit se trouve à la place U+0022 ce qui correspond au code décimal 34. Les doubles chevrons se trouvent en place U+00AB et U+00BB qui correspondent respectivement au nombre décimaux 171 et 187.

Exemple 36

code

```
\begin{luacode}
local GLYF = node.id("glyph")
function guillemets(head, tail)
    for glyph in node.traverse_id(GLYF, head) do
        if glyph.char == 34 then
            if glyph.prev and glyph.prev.id == GLYF
                and not (glyph.prev.char == 40)
            then
                glyph.char = 187
            else
                glyph.char = 171
            end
        end
    end
    node.ligaturing(head)
end
\end{luacode}
\luadirect{
    luatexbase.add_to_callback("ligaturing", guillemets, "
    Guillemets")
}
Voici un "test" !

\luadirect{
    luatexbase.remove_from_callback("ligaturing","Guillemets")
}
```

résultat

Voici un «test»!

Cette substitution conditionnelle est un simple exercice, et tout cela suppose de ne pas avoir de cas pathologique, et impose une forme stricte. On peut tout de même s'amuser à améliorer un peu la chose. En effet, les doubles chevrons sont normalement ajoutés avec des espaces fines insécables en français. On va donc les ajouter. Pour ce faire, on va reprendre le code précédent et ajouter un nœud de type `glyph` avec la fonte courante et ayant pour caractère l'espace fine insécable, c'est-à-dire le caractère Unicode U+202F ce qui correspond à l'entier 160 en décimal. Pour cela, à chaque itération lors du parcours des nœuds de la liste, si on trouve un double guillemet droit, nous allons définir un nouveau nœud :

Exemple 37

```
local fine = node.new(GLYF)
fine.font=font.current()
fine.char = 160
```

Ensuite, si on change le caractère pour le double chevrons fermant, on ajoutera le nœud nouvellement créé avant le nœud du double chevrons, et si le caractère est le double chevrons ouvrant, on ajoutera l'espace après. Cela se fait avec les fonctions `insert_before` et `insert_after` de la bibliothèque Lua de LuaTEX node. L'exemple précédent devient alors le suivant.

Exemple 38

code

```
\begin{luacode}
local GLYF = node.id("glyph")
function guillemets(head, tail)
    for glyph in node.traverse_id(GLYF, head) do
        if glyph.char == 34 then
            local fine = node.new(GLYF)
            fine.font=font.current()
            fine.char = 160
            if glyph.prev and glyph.prev.id == GLYF
                and not (glyph.prev.char == 40)
            then
                glyph.char = 187
                head,fine = node.insert_before(head,glyph,fine)
            else
                glyph.char = 171
                head,fine = node.insert_after(head,glyph,fine)
            end
        end
    end
    node.ligaturing(head)
end
\end{luacode}
\luadirect{
    luatexbase.add_to_callback("ligaturing", guillemets, "
    Guillemets")
}
Voici un "test" !

\luadirect{
    luatexbase.remove_from_callback("ligaturing","Guillemets")
}
```

résultat

Voici un « test » !

Insertion de crénages

De façon analogue au *callback* `ligaturing`, le *callback* `kerning` est supposé insérer les crénages de la fonte, et encore une fois, cela se produit si aucune fonction n'est enregistré dans le *callback*. On va de nouveau utiliser notre fonction `show_nodes` pour observer ce qu'il se passe.

Exemple 39

```
\begin{luacode}
function mykern (head, tail)
    node.kerning(head)
    show_nodes(head)
end
\end{luacode}
\luadirect{
    luatexbase.add_to_callback("kerning", mykern, "My Kerning")
}

\fontencoding{T1}
Vous? Est-ce effectif?
```

Le code ci-dessus produira dans le terminal ceci : 41 9 0 V 13 o u s ? 12 E s t 7 c e 12 e 7 e c 7 t i f 13 ?

On constate donc que des nœuds identifiés avec le nombre 13 ont été ajoutés entre le `V` et le `o` et entre le `f` et le `?`. Ces nœuds sont des crénages : le *o* derrière le *V* est plus joli s'il est légèrement rapproché. Comparez « Vo » à « Vo » (le cas de « f? » est beaucoup moins criant). Ces crénages sont définis par les fontes, comme les ligatures, et c'est bien normal puisque les crénages, comme les ligatures, vont dépendre de l'aspect des glyphes.

Comme les ligatures et les substitutions contextuelles, il existe un positionnement contextuel. Les crénages sont codés dans la fonte comme les ligatures, c'est-à-dire que les glyphes ont une table `kerns` indexée avec les index des caractères et les dimensions comme valeurs (en *scaled point*). Ainsi, le crénage est automatique avec les paires de glyphes seulement, et les positionnements contextuels doivent être traités à la main. Par exemple, dans « A.V. », un crénage (négatif) devrait être inséré entre le premier point et le *V* ; cependant, cela doit arriver uniquement lorsque que le point est précédé par *A* ; si la première lettre est un *T* (c'est-à-dire « T.V. »), le crénage n'est sans doute pas nécessaire (ou alors un crénage différent devrait être utilisé).

Enfin, ce *callback* est le bon endroit pour gérer à la main quelques crénages particuliers. Par exemple, selon les règles typographiques du français, une espace fine doit être insérée avant certains signes de ponctuation, sauf quelques exceptions : si le point d'interrogation doit en effet être précédé d'une telle espace, la règle ne s'applique pas si ce signe de ponctuation est placé juste après une parenthèse ouvrante (?) ou un autre point d'interrogation. Cela peut être codé dans la fonte, mais il est plus simple de gérer ce type de comportement dans le *callback* `kerning`. Si on choisit de procéder ainsi, il faudra s'assurer que tous les nouveaux crénages introduits sont de sous-type (`subtype`) 1, parce que les crénages de sous-type 0 sont des crénages de fonte et peuvent être réinitialisés si l'élargissement des glyphes (*font expansion*) est activé.

Un dernier *callback* avant de construire le paragraphe

Le *callback* précédent s'applique peu importe si on on est en train de construire un paragraphe ou simplement une `\hbox`. Les *callbacks* que nous allons voir dans la suite sont utilisés uniquement dans le premier cas, c'est-à-dire (TEXniquement parlant) lorsqu'une commande verticale est rencontrée en mode horizontal. Le premier d'entre eux est le *callback* `pre_linebreak_filter`, et on peut simplement, comme précédemment, regarder ce qu'il se passe avec notre fonction `show_nodes`. Cette fois ci, il faut cependant retourner la liste head des nœuds dans notre *callback*.

Exemple 40

```
\begin{luacode}
function pre_filter (head, groupcode)
    show_nodes(head)
    return head
end
\end{luacode}
\luadirect{
    luatexbase.add_to_callback("pre_linebreak_filter", pre_filter,
     "My Pre")
}

\fontencoding{T1}
Vous? Est-ce effectif?
\luadirect{
    luatexbase.remove_from_callback("pre_linebreak_filter", "My
    Pre")
}
```

Ce code donnera en sortie dans le terminal et le fichier de log la liste suivante : 9 0 V 13 o u s ? 12 E s t 7 c e 12 e 7 e c 7 t i f 13 ? 14 12

Tout d'abord, on s'apperçoit que le premier nœud temporaire au début de la liste (avec l'identifiant 41) a disparu. Ensuite, un nouveau nœud a été inséré avec l'identifiant 14 : il s'agit d'une pénalité. Si on demande d'afficher son champ `penality`, la valeur retournée sera de 10 000. D'où vient cette pénalité infinie ? Les lecteurs et les lectrices savent peut-être que, lorsque TEX prépare un paragraphe, il supprime le dernier espace (c'est-à-dire le dernier nœud `glue`) de la liste horizontale et le remplace par une *glue* de valeur `\parfillskip` qu'il préfixe avec une pénalité infinie pour qu'aucune coupure de ligne ne se produise. C'est ce qui se passe ici, le dernier nœud est une `glue` avec identifiant 12, mais il n'est pas identique aux nœeuds 12 précédents, puisque son sous-type indique qu'il s'agit d'un nœud `\parfillskip`.

Rien de particulier n'est supposé se produire dans le *callback* `pre_linebreak_filter` et TEX ne fait rien par défaut. Ce *callback* est utilisé pour introduire des effets spéciaux avant que la liste soit coupée en lignes : comme on peut le voir dans le code précédent, ses arguments sont la liste de nœuds à traiter et une chaîne de caractères indiquant dans quel contexte le paragraphe est construit. Les valeurs possibles pour ce deuxième argument sont la chaîne vide (si on se trouve dans la liste verticale principale), `"vbox"`, `"vtop"`, et `"insert"`.

Enfin, la construction d'un paragraphe !

Enfin, nous pouvons construire le paragraphe. On utilise alors le *callback* `linebreak_filter` ; par défaut, le paragraphe est construit automatiquement (heureusement), mais, si on enregistre une fonction dans le *callback*, on peut choisir où couper les lignes nous-mêmes. Enfin, plus ou moins : comme précédemment, il y a une fonction `tex.linebreak` qui fait cela.

Le *callback* reçoit deux arguments : une liste de nœuds et un booléen. Ce dernier est `true` si on construit la partie d'un plus grand paragraphe qui se trouve avant un élément mathématique hors ligne ; sinon, il est à `false`. Étant donné une liste de nœuds en entrée, une fonction pour ce *callback* doit retourner une autre liste d'une nature totalement différente : elle doit être une liste de boîtes horizontales (les lignes de texte), des *glues* (le ressorts entre les lignes), des pénalités (par exemple pour les lignes veuves et orpheline), et possiblement des insertions[53] ou du matériel pour ajuster verticalement le contenu, etc. Comme nous l'avons mentionné, la fonction `tex.linebreak` fait tout cela ; cette fonction peut même prendre un argument optionnel, une table avec des paramètres TEX comme clés (par exemple `hsize`, `tolerance`, `widowpenalty`), pour que les paragraphes puissent être construits en tenant compte de valeurs particulières affectées à ces paramètres.

Comme exemple d'utilisation de ce *callback*, nous allons essayer de construire un paragraphe dont la première ligne est en petites capitales, comme cela se fait parfois pour le premier paragraphe d'un chapitre. On utilise LuaLATEX, et donc, on ne souhaite pas utiliser trop d'astuces sophistiquées, et on souhaite que TEX construise quand même le *meilleur* paragraphe possible (c'est-à-dire qu'on ne souhaite pas ajuster simplement, mais probablement disgracieusement, les espaces de la première ligne) : par exemple, on souhaite laisser la possibilité qu'il y ait une césure à la première ligne. Le code qui va suivre est un simple prototype, mais il donne un aperçu d'une approche possible. Premièrement, nous avons besoin d'une fonte et d'une commande pour déclarer que le prochain paragraphe doit être traité spécifiquement. Pour déclarer cette fonte, on utilisera la commande `\font`, et dans notre cas, on utilisera *AlegreyaSC-regular* de la fonte du numéro de cette *Lettre*.

Exemple 41

```
\font\firstlinefont={file:AlegreyaSC-Regular.otf}
```

Ensuite, nous définissons un environnement qui désactive les trois *callbacks* de traitement des nœuds avec `luatexbase.reset_callback("<callback>", false)`, car on souhaite garder la liste originale des nœuds avec seulement les remplacements qui se font avant le *callback* `pre_linebreak_filter`, et nous allons opérer les césures, les ligatures et le crénage à la main. En pratique, il serait préférable de déterminer et sauvegarder les fonctions additionnelles enregistrées dans ces *callbacks*, et utiliser celles-là, pour rendre notre code plus robuste et permettre aux utilisateurs et utilisatrices de l'utiliser avec leurs modifications du comportement par défaut. Cela pourrait être fait, par exemple, avec la commande `callback.find`, mais nous ne nous attarderons pas sur ce point ici. Nous enregistrons ensuite notre fonction `mypar` au *callback* `linebreak_filter` et à la fin de l'environnement, nous rétablissons le comportement par défaut.

Exemple 42

```
\newenvironment{firstparagraph}{\luadirect{
    luatexbase.reset_callback("hyphenate", true)
    luatexbase.reset_callback("ligaturing", true)
```

53. Par exemple des notes de bas de page.

```
    luatexbase.reset_callback("kerning", true)
    luatexbase.add_to_callback("linebreak_filter",mypar,"MyPar")
    }%
}%
{\luadirect{
    luatexbase.reset_callback("hyphenate", false)
    luatexbase.reset_callback("ligaturing", false)
    luatexbase.reset_callback("kerning", false)
    luatexbase.reset_callback("linebreak_filter", false)
    }
}
```

Notre fonction Lua `mypar` est la suivante :

Exemple 43

```
function mypar (head, is_display)
    local par, prevdepth, prevgraf = check_par(head)
    tex.nest[tex.nest.ptr].prevdepth = prevdepth
    tex.nest[tex.nest.ptr].prevgraf = prevgraf
    return par
end
```

Notre fonction va appeler une autre fonction Lua `check_par` (que nous allons détailler plus tard) qui retourne un paragraphe et de nouvelles valeurs pour `prevdepth` et `prevgraf`, que nous pourront affecter au niveau courant d'*emboîtement* (*nesting level*), donc à la liste de boîtes dans laquelle on se trouve.

Ces valeurs sont à retrouver dans la table Lua `tex.nest[tex.nest.ptr]`. Voici une description de l'utilité de ces deux valeurs :

`prevdepth` est la profondeur[54] de la dernière ligne dans une paragraphe construit ;

`prevgraf` est le nombre de lignes d'un paragraphe construit.

Avant d'expliquer en quoi consiste la fonction `check_par`, nous allons présenter ici la fonction auxiliaire qui est utilisée pour faire ce que nous avons empêche LuaTEX de faire : insérer les points de césure, les ligatures et les crénages, et ensuite fabriquer le paragraphe.

Exemple 44

```
local function do_par (head)
    lang.hyphenate(head)
    head = node.ligaturing(head)
    head = node.kerning(head)
    local p, i = tex.linebreak(head)
    return p, i.prevdepth, i.prevgraf
end
```

Il n'est pas nécessaire de réaffecter à `head` la sortie de `lang.hyphenate` car aucun point de césure ne peut être ajouté en premier nœud d'une liste (de toute manière, la sortie de `lang.hyphenate` est un booléen qui indique le succès ou l'échec de l'exécution de la fonction). En plus du paragraphe lui-même, `tex.linebreak` retourne une table avec les

54. Distance maximale sur laquelle le texte se prolonge sous la ligne de base (déterminée par exemple par la longueur du descendant de p s'il y en a un).

valeurs `prevdepth` et `prevgraf` (mais aussi les valeurs `looseness` et `demerits`, qui sont des valeurs utilisées pour la fabrication *optimale* du paragraphe).

On va aussi récupérer la fonte pour les petites capitales que l'on a définie plus tôt côté TEX. Pour cela, on utilisera le code suivant :

Exemple 45

```
local firstlinefont = font.id("firstlinefont")
```

qui nous fournit la représentation numérique de la fonte.

On peut maintenant s'attaquer à la fonction principale `check_par`. Le début du code est le suivant :

Exemple 46

```
local HLIST = node.id("hlist")
local GLYF = node.id("glyph")
local KERN = node.id("kern")
function check_par (head)
    local par = node.copy_list(head)
    par, prevdepth, prevgraf = do_par(par)
    local line = par
    while not (line.id == HLIST) do
        line = line.next
    end
```

Tout d'abord, on crée une copie de la liste de nœuds pour ne pas modifier l'originale, notamment ne pas lui ajouter des points de césure qui seront supprimés par la suite. Ensuite, on lance une première tentative de construction du paragraphe avec notre fonction `do_par`. On cherche alors la première ligne du paragraphe construit en parcourant la table `par` jusqu'à obtenir une boîte horizontale `hlist` (les premiers éléments de la liste du paragraphe peuvent être une *glue*, ou du matériel pour ajuster verticalement le paragraphe).

Exemple 47

```
    local again
    for item in node.traverse_id(GLYF, line.head)
        do if not (item.font == firstlinefont) then
            again = true
            for glyph in node.traverse_id(GLYF, head)
                do if not (glyph.font == firstlinefont) then
                    glyph.font = firstlinefont
                    break
                end
            end
            break
        end
    end
```

Ensuite, pour la première ligne (`line.head`), on va vérifier que tous les glyphes ont la bonne fonte (notre `\firstlinefont`) ; dès que l'on rencontre un glyphe qui n'est pas en petites capitales, on inspecte alors tous les glyphes de la liste originale (`head`) jusqu'à ce qu'on

recontre le premier qui n'est pas en petites capitales, et on modifie la fonte avec celle désirée. Le ou la lectrice peut sans doute voir où on va : nous allons reconstruire le paragraphe autant de fois que nécessaire, à chaque fois en passant un glyphe supplémentaire de la première ligne en petites capitales, jusqu'à ce que tous les glyphes de la première ligne soient en petites capitales. Cela explique pourquoi nous devons à chaque itération réinsérer les points de césures, les ligatures et les crénages : à chaque fois qu'une partie du texte passe en petites capitales, la largeur des glyphes concernés peut changer, si bien que la composition du paragraphe doit être calculée à nouveau.

Le booléen `again` indique qu'on a trouvé un glyphe en bas de casse que l'on a passé en petites capitales, et donc qu'il faut relancer l'algorithme. Quand c'est le cas, on retire le paragraphe de la mémoire de TEX et on relance la fonction avec la liste `head` modifiée ; c'est ce que fait le code suivant :

Exemple 48

```
    if again then
        node.flush_list(par)
        return check_par(head)
```

La suite du code traite le cas où il n'y a pas besoin de relancer la procédure de changement de fonte.

Exemple 49

```
    else
        local secondline = line.next
        while secondline
            and not (secondline.id == HLIST) do
            secondline = secondline.next
        end
        if secondline then
            local list = secondline.head
            for item in node.traverse_id(GLYF,list)
                do if item.font == firstlinefont then
                    item.font = font.current()
                else
                    break
                end
            end
```

Dans ce cas, tous les glyphes de la première ligne sont en petites capitales, mais il y a encore une chose à vérifier. Supposons que le dernier caractère ayant été mis en petites capitales est x. Par définition, x est à la fin de la première ligne avant que sa fonte ait été changée, mais est-ce le cas après le changement ? Pas nécessairement, il peut tout à fait être possible qu'avec les nouvelles dimensions de x, il faille[55] couper la ligne *avant*, et peut-être même pas juste avant, mais quelques glyphes avant. Ainsi, il est possible d'obtenir quelques petites capitales en deuxième ligne de paragraphe. Ces petites capitales doivent alors être changées de nouveau. Oui, mais comment ? On change leur fonte, et on construit le paragraphe de nouveau ? On ne peut pas faire cela au risque de se trouver bloqué dans une boucle inifine (des bas de casses dans la première ligne, des petites capitales dans le secondes, et encore et encore…). La solution que nous avons adoptée ici est de réaffecter la fonte originelle (avec

55. C'est l'optimisation de TEX qui détermine cela.

`font.current()`) à ces glyphes et de les laisser là.

La fin du code de la fonction permet de gérer encore quelques problèmes.

Exemple 50

```
            list = node.ligaturing(list)
            for kern in node.traverse_id(KERN, list)
                do if kern.subtype == 0 then
                    node.remove(list, kern)
                end
            end
            list = node.kerning(list)
            secondline.head = node.hpack(
                list, secondline.width, "exactly")
        end
        node.flush_list(head)
        return par, prevdepth, prevgraf
    end
end
```

En effet, si les premiers glyphes de la seconde ligne sont *f* et *i*, en petites capitales, ils ne forment sans doute pas de ligature, mais une fois la fonte courante rétablie ? Nous devons donc rétablir les ligatures. Et en ce qui concerne les crénages ? On doit supprimer tous les crénages de fonte (c'est-à-dire de sous-type 0) et reconstruire les crénages. On doit enfin ajuster la largeur de la deuxième ligne (qui a été modifié par le changement de fonte) pour qu'elle reprenne la largeur d'avant le changement de fonte, les *glues* vont alors être étirées ou compressées. Ce procédé n'est pas optimal, mais les cas où des petites capitales sont présentes en deuxième ligne sont très rares. Les dernières lignes de la fonction `check_par` suppriment la liste `head` originale et retournent le paragraphe et les deux paramètre `prevdepth` et `predgraf`.

Exemple 51

code

```
\begin{firstparagraph}
\ipsum<Lang=FR>[1]
\end{firstparagraph}

\ipsum<Lang=FR>[3]
```

résultat

SANS QU'ELLE DAIGNÂT LE DIRE À PERSONNE, UN ACCÈS DE FIÈVRE D'UN DE SES FILS LA mettait presque dans le même état que si l'enfant eût été mort. Un éclat de rire grossier, un haussement d'épaules, accompagné de quelque maxime triviale sur la folie des femmes, avaient constamment accueilli les confidences de ce genre de chagrins, que le besoin d'épanchement l'avait portée à faire à son mari, dans les premières années de leur mariage. Ces sortes de plaisanteries, quand surtout elles portaient sur les maladies de ses enfants, retournaient le poignard dans le cœur de Mme de Rênal. Voilà ce qu'elle trouva au lieu des flatteries empressées et mielleuses du couvent jésuitique où elle avait passé sa jeunesse. Son éducation fut faite par la douleur. Trop fière pour parler de ce genre de chagrins, même à son amie Mme Derville, elle se figura que tous les hommes étaient comme son mari, M. Valenod et le

résultat (suite)

sous-préfet Charcot de Maugiron. La grossièreté, et la plus brutale insensibilité à tout ce qui n'était pas intérêt d'argent, de préséance ou de croix ; la haine aveugle pour tout raisonnement qui les contrariait, lui parurent des choses naturelles à ce sexe, comme porter des bottes et un chapeau de feutre.
Mais la demoiselle du comptoir avait remarqué la charmante figure de ce jeune bourgeois de campagne, qui, arrêté à trois pas du poêle, et son petit paquet sous le bras, considérait le buste du roi, en beau plâtre blanc. Cette demoiselle, grande Franc-Comtoise, fort bien faite, et mise comme il le faut pour faire valoir un café, avait déjà dit deux fois, d'une petite voix qui cherchait à n'être entendue que de Julien : Monsieur! Monsieur! Julien rencontra de grands yeux bleus fort tendres, et vit que c'était à lui qu'on parlait.

Le lecteur ou la lectrice aura peut-être repéré quelques failles dans ce code. Une solution complète aurait largement dépassé les limites de cet article déjà assez long. Pour l'améliorer (ce que nous laissons en exercice), on peut essayer de proposer une solution qui ne repose pas sur l'hypothèse qu'aucune fonction n'est enregistrée dans les autres *callbacks*. On peut aussi mieux gérer l'introduction possible de petites capitales au début de la deuxième ligne (peut-être en reconstruisant le paragraphe à partir de cette ligne sans toucher à la première?). On peut aussi imaginer une solution qui change le caractère et non la fonte: il faut pour cela aller chercher le caractère en petite capitale dans la table Unicode (mais le test pour savoir si cela a déjà été fait est alors différent).

Traitement du paragraphe une fois construit

Le *callback* `post_linebreak_filter` est très calme après tout ce que nous venons de voir: par défaut, rien ne s'y passe. Il lui est donné comme premier argument ce que le *callback* `linebreak_filter` retourne, c'est-à-dire une liste de listes horizontales (les lignes du paragraphe), des pénalités, des *glues* et peut-être du matériel interligne. En second argument, il est possible de lui passer une chaîne de caractère à la manière du *callback* `pre_linebreak_filter`. Paul Isambert, dans son article [5], montre plusieurs utilisations de ce *callback*, en particulier pour souligner du matériel. Paul Isambert conseille aussi pour les lecteurs qui cherchent des défis, d'essayer d'adapter à LuaTEX le code de la section 5.9.6 de *TEX by Topic: A TEXnician's Reference*.

Le *callback* retourne un paragraphe, potentiellement le même que celui reçu. Le paragraphe est alors ajouté à la liste verticale englobante, et ce qui est suit est le travail du constructeur de page. Notre s'exploration s'arrête ici.

Conclusion

La plupart des opérations que nous avons vues ici ne sont pas nouvelle en TEX : LuaTEX donne simplement accès à ces opérations. Depuis le tout début, TEX lit des lignes et des *tokens* (lexèmes) et construit des listes de nœuds ; c'est son principal travail. Le contrôle du processus de composition est ce qui fait de TEX un si bon outil, peut-être meilleur que d'autres outils de composition ; LuaTEX amène le contrôle encore plus loin et permet la manipulation des plus petits atomes qui constituent la typographie numérique: les caractères et les glyphes, ainsi que quelques autres détails techniques. Paul Isambert conclut son article, source principale de cette adaptation, en disant que du point de vue de la liberté, de manière rétrospective, il peut trouver TEX82 et ses successeurs un peu dictatoriaux par comparaison avec LuaTEX.

Maxime Chupin

Références

[1] THE LUATEX TEAM. *The* luatex *package. The LuaTeX engine*. 9 déc. 2021. URL : https://ctan.org/pkg/luatex.

[2] *Cahiers GUTenberg : Introduction à LuaTEX* 2010.54-55 (2010). URL : http://www.numdam.org/issues/CG_2010___54-55/.

[3] Paul ISAMBERT. « LuaTEX: What it takes to make a paragraph ». In : *TUGboat* 32.1 (2011).

[4] Manuel PÉGOURIÉ-GONNARD. *The* luacode *package. Helper for executing lua code from within TEX*. Version 1.2a. 24 juin 2016. URL : https://ctan.org/pkg/luacode.

[5] Paul ISAMBERT. « Three things you can do with LuaTEX that would be extremely painful otherwise ». In : *TUGboat* 31.3 (2010).

[6] Victor EIJKHOUT. *TEX by Topic: A TEXnician's Reference*. Dante, 2014.

LA FONTE DE CE NUMÉRO : ALEGREYA

La super-famille[56] de fontes Alegreya est l'œuvre de Juan Pablo del Peral[57], pour la fonderie argentine Huerta Tipografica[58]. Cette fonderie distribue ces fontes sous la version 1.1 de la licence SIL Open Font[59] et en propose une version commerciale, dite *pro*, comprenant caractères grecs et cyrilliques ainsi que des caractères latins additionnels, permettant différents raffinements typographiques dont nous ne traiterons pas ici[60]. En l'état, Alegreya offre déjà de nombreuses fonctionnalités, qui permettent de composer un document satisfaisant ; cette *Lettre* en témoigne (nous espérons que vous partagez notre avis).

Grâce au très prolifique Bob Tennent[61, 62], Alegreya bénéficie d'un package support pour LaTeX, pdfLaTeX, XeLaTeX et LuaLaTeX, qui est très utile. Pour utiliser les fontes, il suffit d'ajouter au préambule de votre document l'appel suivant :

Exemple 52 : avec empattements

```
1 \usepackage{Alegreya}
```

Une vaste palette

Alegreya propose un grand choix de fontes que nous détaillons ici. Il est facile de les obtenir grâce au package dédié.

Les italiques sont intéressants — on appréciera notamment le magnifique *g* italique, que nous n'avons pas laissé passer (il figure en filigrane de la couverture de ce numéro) :

56. Voir, en anglais, https://en.wikipedia.org/wiki/Font_superfamily.
57. juan chez huertatipografica point com point ar
58. http://www.huertatipografica.com.ar
59. Le trésorier de l'association, François Druel, connaît bien cette licence et a prévu d'en parler dans un prochain article consacré à une autre fonte en bénéficiant.
60. Le logiciel `fontforge` nous permet de voir que de nombreux caractères grecs et cyrilliques sont présents dans la fonte à laquelle nos logiciels favoris donnent accès. C'est tant par manque de compétences que de temps que nous n'abordons pas dans cet article le cas des caractères grecs et, surtout, cyrilliques.
61. Bob Tennent a *packagé* de très nombreuses fontes pour (all)TEX. Voir https://ctan.org/author/tennent
62. Cet article est largement inspiré du README de la documentation du package Alegreya, dû à Bob Tennent. Merci à lui !