Cahiers GUTenberg

VOYAGE AU CENTRE DE TEX : COMPOSITION, PARAGRAPHAGE, CÉSURE

Yannis HARALAMBOUS

Cahiers GUTenberg, nº 44-45 (2004), p. 3-53.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2004___44-45_3_0>

Voyage au centre de TEX : composition, paragraphage, césure

Yannis HARALAMBOUS

Département Informatique
ENST Bretagne
CS 83818, 29238 Brest,
yannis.haralambous@enst-bretagne.fr

Τοὺς Λαιστρυγόνας καὶ τοὺς Κύκλωπας,
τὸν ἄγριο Ποσειδῶνα δὲν θὰ συναντήσεις,
ἂν δὲν τοὺς κουβανεῖς μὲς στὴν ψυχή σου,
ἂν ἡ ψυχή σου δὲν τοὺς στήνει ἐμπρός σου.

Κ. Καβάφης, *Ἰθάκη* [1]

La vie est pleine de paradoxes. Ainsi, alors que Knuth a développé tout un système de « programmation littéraire » pour faire de TEX le premier logiciel auto-documenté, alors qu'il a publié le code de TEX sous forme d'un magnifique ouvrage relié — *TEX, The Program* [6], qui constitue le tome B de la série *Computers & Typesetting* —, alors qu'il y a soigneusement planifié et agrémenté de sagesse et d'humour le parcours à travers son logiciel favori, il s'avère qu'extrêmement peu nombreux sont ceux qui l'ont suivi dans cette aventure. Pourtant, grâce au système `WEB` [2], le code est présenté de manière typographiquement irréprochable, il est segmenté en petits modules faciles à comprendre et bien documentés, et ces modules sont présentés au lecteur dans un ordre pédagogique, totalement indépendant de leur ordre d'exécution. Des conditions optimales pour la lecture et la compréhension de ce logiciel, écrit par l'un des plus grands du XXe siècle.

Pourquoi les gens, d'ordinaire friands d'aventure et d'émotions fortes, évitent-ils ce parcours ? Est-ce la vétusté du langage fonctionnel Pascal face aux éblouissants

1. « Les Lestrygons et les Cyclopes,/tu ne les rencontreras pas, ni l'irascible Poséidon,/si tu ne les transportes pas dans ton âme/si ton âme ne les fait pas surgir devant toi », C. Cavafis, *Ithaque* [1].

2. Le système de « programmation littéraire » de Knuth [9], à ne pas confondre avec le Web de Tim Berners-Lee. Les deux n'ont en commun que la référence à la toile d'araignée.

langages orientés objet dont nous disposons aujourd'hui ? Est-ce l'absence de textes introductifs, d'exégèses, de « guides de voyage » ? On n'en sait rien. Pourtant, à une époque où l'on parle de plus en plus de « textes fondamentaux » des diverses religions, le code de TEX en est un, justement : il renferme la vérité absolue de tout TEXiste, il contient les réponses à toutes les questions (TEXniques) que l'on puisse se poser et il peut procurer la puissance à celui qui désire aller plus loin en étendant TEX ou en l'adaptant aux nouveaux besoins...

Ce texte se veut un « guide de voyage » dans une petite partie de cette forêt amazonienne qu'est le code de TEX. Plutôt que de survoler, en toute sécurité, cette forêt par avion en esquissant ses grandes lignes, nous avons choisi l'aventure : nous nous plongerons carrément dans la jungle et nous y suivrons un petit sentier qui nous mènera au centre de TEX. Au dépit des insectes voraces et des crocodiles, nous suivrons à la trace le parcours d'un mot à travers TEX, depuis sa lecture jusqu'au moteur de paragraphage et de césure. Inutile de dire que ce sentier laisse des centaines de km² de jungle inexplorés, notamment l'expansion de macros, la procédure de sortie, les insertions, le mode mathématique, les tableaux, sans parler des jungles avoisinantes : eTEX, Oméga, pdfTEX, et *tutti quanti*. Le défi que nous relevons est de donner au lecteur le goût de l'aventure et l'envie de découvrir les trésors qui y sont cachés.

1 Préliminaires

Le « tome B » de Knuth [6] est certes un ouvrage passionnant à lire et à consulter, mais il souffre d'un petit défaut : il est hélas obsolète, puisque la version de TEX qui y est décrite est la 2. Pour suivre donc notre cheminement, nous demandons au lecteur de produire sa propre copie actualisée du code commenté de TEX, en suivant la procédure suivante :

1. récupérer les fichiers `tex.web` (nous avons utilisé la version 3.141592, de décembre 2002) et `tex.ch` (qui se trouve dans l'archive `/systems/web2c/web2c.tar.gz` sur CTAN) ;
2. lancer `weave tex.web tex.ch`, qui va produire un fichier `tex.tex` ;
3. aller à la ligne 87 de ce fichier et changer `\iffalse` en `\iftrue` ;
4. le compiler.

Les numéros de section du type « §123 » et les extraits de code que nous donnons se réfèrent à ce document actualisé, et non pas à *TEX, The Program*. Par la notation « [TB123] » on se réfère à la page 123 du *TEXbook* [5].

2 Notions fondamentales

À partir du source que l'on lui fournit, TEX va d'abord produire des *tokens*[3] [TB38], c'est-à-dire des unités lexicales « caractère » ou « commande ».

Dans la syntaxe de TEX les caractères ont des tâches bien précises. Mais, contrairement à la plupart des langages de programmation, les différents caractères d'échappement ne sont nullement codés en dur : si l'on utilise le pour-cent pour les commentaires et l'antislash pour les noms de commande, ce n'est que par convention, on aurait très bien pu permuter les sémantiques de ces deux caractères, ou choisir tout autre caractère pour celles-ci. Pour indiquer à TEX quel caractère est utilisé pour quelle tâche, on se sert du « code de catégorie » (ou « catcode » en jargon) [TB37], attribué à l'aide de la primitive `\catcode`. Ainsi, les « lettres » sont de catcode 11, l'espace de 10, les caractères actifs de 13, etc.

D'autre part, il y a deux types de commandes : les *primitives* [TB9] (qui sont définies en dur dans le code de TEX) et les *macros* [TB199] (définies à l'aide de primitives, par le format, les paquetages ou l'utilisateur). Après la lecture des tokens a lieu l'expansion des macros [TB212], celles-ci vont produire d'autres tokens « caractère » ou « commande ». Après un certain nombre d'expansions on finit par n'avoir que des tokens de caractère et de primitive.

La prochaine étape est la création des *listes* [TB64]. Qu'est-ce qu'une liste ? Une idée géniale de Knuth a été de décrire la page comme un ensemble de boîtes qui contiennent d'autres boîtes. La page entière en est une, chaque glyphe peut être considéré comme une boîte minimale. Pour créer ces boîtes on peut agir horizontalement (par exemple dans le cas de la composition d'une ligne), ou verticalement (lorsqu'on assemble les paragraphes pour former une page).

Que met-on dans une boîte ? Les différents ingrédients forment des *nœuds* [§133-159] et dans une boîte on place une liste chaînée de nœuds. Par « création de listes » on entend donc le fait de passer des tokens aux nœuds, en mobilisant toutes les ressources : primitives, fontes, etc. À la fin de cette étape on a un certain nombre de listes de nœuds, qui correspondent aux contenus des différentes boîtes. Mais attention : un « paragraphe » n'est encore qu'une très longue liste horizontale.

C'est à ce moment qu'on fait appel au moteur de paragraphage [§813]. Son rôle est de couper la liste horizontale du paragraphe en un certain nombre de listes horizontales plus petites, les lignes du paragraphe. Ces listes sont placées dans la

3. Dans ce texte nous avons promu les mots « token » et « glue » au rang d'anglicisme de la langue française. D'autres auteurs ont traduit *token* par « unité lexicale » et *glue* par « ressort » — même si ces traductions reflètent assez bien le sens des termes, il est indéniable que dans la communauté TEX francophone on a depuis longtemps (à tort ou à raison) acquis l'habitude d'utiliser les termes anglais.

liste verticale globale, qui à son tour est coupée en portions de hauteur égale, les pages du document.

Ayant obtenu toutes les listes horizontales et verticales il ne reste plus qu'à traduire les nœuds en instructions DVI [§592], c'est la dernière tâche qu'effectue TEX avant de s'arrêter.

2.1 Tokens, nœuds, listes, pile sémantique

Les *tokens* sont les atomes de la segmentation que TEX effectue sur le flot de caractères entrant. Un token est un triplet (*cur_cmd*, *cur_chr*, *cur_cs*) [§§289, 297] où *cur_cmd* est un « code de commande » c'est-à-dire : un catcode s'il s'agit d'un caractère, ou un numéro de primitive s'il s'agit d'une commande. Ainsi, la primitive `\char` a le numéro 16, la `\radical` le 65, `\multiply` le 90 et ainsi de suite [§§207-210]. *cur_chr* est la position d'un glyphe dans la table de fonte si *cur_cmd* est entre 0 et 15, et un modificateur de code de commande sinon. Enfin, *cur_cs* est 0 s'il s'agit d'un token de caractère — par contre s'il s'agit d'une commande, alors c'est la position de celle-ci dans la table des équivalents [§220].

À partir des tokens, TEX produit des *nœuds* qui forment des listes chaînées [§133]. On distingue les nœuds par leurs *type* et *sous-type* (les deux premiers octets d'un mot à 32 bits, les 16 bits restants formant un pointeur vers le nœud suivant de la liste). Voici les types de nœud définis dans TEX :

1. le « nœud de caractère »[4] 

char_node, d'une longueur d'exactement un mot (32 bits) [§134]. On stocke ce type de nœud dans une partie réservée de la mémoire et on utilise leurs type et sous-type pour *font* (le numéro de fonte) et *character* (la position du glyphe dans la table de fonte). Pour savoir si un nœud est un « nœud de caractère » on dispose de la fonction de test *is_char_node* ;
2. le « nœud de liste horizontale » 

hlist_node [§135] est le nœud d'une boîte faite à partir d'une liste horizontale de glyphes. Il a une longueur de $7+1$ mots[5], les suivants : la largeur, la profondeur et la hauteur de la boîte, le *shift_amount* qui est, en somme, l'ordonnée de l'intersection entre la ligne de base et la boîte, un pointeur vers le premier nœud de la liste, et enfin trois mots qui correspondent à la glue : *glue_set* (quantité de glue), *glue_sign* (0 = *normal* si la glue est rigide, 1 = *stretching* si elle est dilatante, 2 =

4. Entre guillemets, car en réalité il s'agit de glyphe et non pas de caractère. Cf. [3, p. 54-58] pour plus d'informations sur la différence entre les notions de caractère et de glyphe.

5. Dans la suite, le mot « +1 » est toujours celui contenant le type *type* du nœud et le lien *link* vers le nœud suivant.

shrinking si elle contractante), *glue_order* (l'ordre de grandeur[6] de la glue : 0 = *normal*, 1 = *fil*, 2 = *fill*, 3 = *filll*). Ce type de nœud n'a pas de sous-type. On obtient la valeur de chaque mot par une macro[7] de même nom. La procédure *new_null_box* [§136] crée un nouveau nœud de ce type ;

3. le « nœud de liste verticale » 

vlist_node [§137] est identique au précédent sauf que la liste est verticale (et que le *shift_amount* est le décalage par rapport à l'axe vertical de composition) ;
4. le « nœud de filet » 

rule_node [§138], utilisé pour produire une boîte noire dans le DVI, n'a que 4 + 1 mots : la largeur, la profondeur et la hauteur. Il a une particularité : si l'on est dans une liste horizontale, alors la profondeur et la hauteur peuvent être extensibles, dans le sens qu'elles peuvent prendre les dimensions de la boîte qui contient le filet. Il en est de même de la largeur quand on est dans une liste verticale. Pour obtenir une dimension extensible il faut lui donner une valeur de 2^{-30}. Pour créer un nœud de filet on utilise *new_rule* ;
5. le « nœud d'insertion » 

ins_node [§140] est utilisé pour les insertions, par exemple les notes de bas de page ou les figures. Il est d'une longueur de 5 + 1 mots ;
6. le « nœud de marque » 

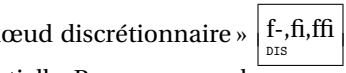
mark_node [§141] est utilisé pour les `\mark` ;
7. le « nœud d'ajustement » 

adjust_node [§142] est utilisé pour les `\vadjust` qui servent aux alignements ;
8. le « nœud de ligature » 

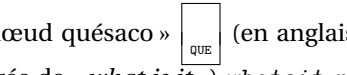
ligature_node [§143] est utilisé pour les ligatures. Il contient 1 + 1 mots, dont le deuxième, appelé *lig_char*, contient trois informations : un octet *font*, un octet *character* (comme dans les nœuds de caractère), ainsi qu'un demi-mot *lig_ptr* qui est un pointeur vers une liste chaînée de nœuds de caractère (il s'agit des nœuds qui ont été « remplacés » par la ligature). Le sous-type de ce nœud peut être 0, 1, 2 ou 3 selon le type de ligature intelligente ([3, p. 620]). Pour obtenir un nœud de ligature on se sert de la fonction *new_ligature* [§144] ;

6. En mathématiques on a divers ordres d'infini : le cardinal de $\mathbb{R}$ ($\aleph\_1$) est d'ordre supérieur de celui de $\mathbb{N}$ ($\aleph\_0$). Il en est de même dans TEX : une glue *fil* va écraser toute glue rigide, une glue *fill* va écraser toute glue *fil*, et ainsi de suite…

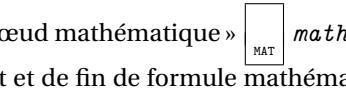
7. Quand nous parlons de « macro », il s'agit de macro-commande de `WEB`, un concept similaire à celui de macro de préprocesseur C. Il s'agit d'une construction qui disparaît lorsqu'on produit du code Pascal à partir du fichier `WEB`. À ne pas confondre avec les procédures Pascal que l'on trouve également dans le code de TEX, et qui subsistent dans le code Pascal produit.

9. le « nœud discrétionnaire » 

`disc_node` [§145] indique une césure potentielle. Pour comprendre comment il fonctionne, prenons le cas du mot allemand « backen » qui se coupe « bak-ken ». Pour obtenir ce comportement, on écrit `ba\discretionary{k-}{k}{ck}en` [TB95] dans le source TEX, où les deux premiers arguments correspondent aux chaînes produites lors d'une césure et le dernier à la chaîne obtenue en l'absence de césure. Eh bien, le nœud discrétionnaire contient deux liens `pre_break` et `post_break` vers des listes horizontales produites à partir des deux premiers arguments. D'autre part, son sous-type contient le nombre de nœuds consécutifs à remplacer par les deux listes, en cas de césure. Il s'agit donc du nombre de nœuds générés à partir du dernier argument de la primitive. Rappelons que `\-` est une primitive dont l'effet est l'équivalent de `\discretionary{}{-}{}`. Pour obtenir un nœud discrétionnaire on se sert de `new_disc` ;

10. le « nœud quésaco » 

(en anglais « quésaco » se dit « *whatsit* », version abrégée de « *what is it* ») `whatsit_node` [§146] est un nœud joker pour les extensions de TEX. Ainsi, les `\write` et `\special` se servent de ce type de nœud ;

11. le « nœud mathématique » 

`math_node` [§147] est une sorte de balise de début et de fin de formule mathématique (il se trouve aux endroits où l'on rencontre le dollar dans le source TEX). Son sous-type est `before` ou `after` selon le fait si c'est une balise d'ouverture ou de fermeture. Enfin, on a un mot de « largeur » `width` qui indique la valeur de blanc entourant la formule (blanc inséré par `\mathsurround`) ;

12. le « nœud de glue » 

`glue_node` [§149] est utilisé pour insérer de la glue dans une liste. Il n'est en fait qu'un pointeur vers une *spécification de glue*, cela permet de gagner un peu d'espace mémoire. Il contient donc `glue_ptr`, qui pointe vers une spécification de glue [§150]. Cette dernière occupe $4+1$ mots, les trois premiers contiennent la largeur (`width`), la dilatation (`stretch`) et la contraction (`shrink`). Enfin, les champs de type et de sous-type contiennent resp. l'ordre de grandeur de dilatation (`stretch_order`) et celui de contraction (`shrink_order`). Ainsi, un `\hskip 1pt plus 1fil minus 2fill` va générer un nœud de glue qui va pointer vers une spécification ayant un `width` égal à 1pt, un `stretch` égal à 1, un `shrink` égal à 2, et des `stretch_order` et `shrink_order` resp. égaux à 1 et à 2. Notons que le champ `link` de cette spécification contient une référence vers le numéro de paramètre de glue qui est à l'origine de la spécification. Ainsi, s'il s'agit, par exemple, d'un `\baselineskip`, le numéro de ce paramètre s'y trouve. Pour obtenir un

nœud de glue, on utilise *new_glue* [§153] si c'est une glue « anonyme », et *new_param_glue* [§152] si c'est une glue associée à un paramètre ;

13. le « nœud de crénage » 

kern_node [§155] est utilisé pour insérer un crénage. C'est le mot de largeur *width* qui contient la quantité de crénage (une quantité positive produit un rapprochement !). Le sous-type peut être *normal* (un crénage implicite obtenu par les métriques de fonte ou par des calculs internes de composition de formule mathématique), *explicit* (produit par un `\kern` ou un `\/`), *acc_kern* (produit par un `\accent`), *mu_glue* (produit par un `\mkern` dans une formule mathématique). Pour obtenir un nœud de crénage on utilise *new_kern* [§156] ;
14. le « nœud de pénalité » 

penalty_node [§157] associé à une pénalité de coupure de ligne ou de page. Une pénalité supérieure à 10 000 est considérée comme infinie, le mot-clé correspondant est *inf_penalty* ;
15. le « nœud d'annulation » 

unset_node [§159], utilisé dans les alignements ;
16. le « nœud de style » 

style_node [§688], utilisé pour indiquer le style de formule mathématique (le sous-type étant *display_style*, *text_style*, *script_style* ou *script_script_style*) ;
17. le « nœud de choix » 

choice_node [§689], qui contient des pointeurs vers différentes listes mathématiques selon la primitive `\mathchoice`.

Mis à part ces types de nœud, il existe également des nœuds de coupure potentielle, ceux-ci n'apparaissent que lors du paragraphage.

Revenons maintenant aux listes. À tout moment TEX est en train de construire une liste, c'est la *liste courante*. Et à tout moment TEX est susceptible d'interrompre la construction de cette liste pour entamer celle d'une autre, et y revenir aussitôt terminé. Par exemple, une page est une liste verticale que l'on interrompt pour créer des paragraphes formés à partir de listes horizontales. Pour se retrouver dans cette multitude de listes en cours de construction, on utilise la « pile sémantique » [§211]. Il s'agit d'une pile de structures du type suivant :

1. un *mode* (horizontal, vertical, mathématique, interne ou externe) ;
2. un pointeur *head* vers le premier nœud de la liste, c'est-à-dire vers un nœud de liste horizontale ou verticale, qui à son tour pointe vers le premier nœud *link*(*head*) de la liste ;
3. un pointeur *tail* vers le dernier nœud de la liste ;
4. deux informations auxiliaires de moindre intérêt : *prev_graf* et *aux*.

3 Par où commencer ?

Tout programme Pascal commence par une instruction `program`, celle de TEX se trouve au §4. Après un certain nombre d'initialisations on se retrouve au §1332 qui est une synthèse du cycle de vie de TEX : initialisation, grande boucle de lecture de token, de création de liste, de paragraphage et de sortie DVI, arrêt. En particulier, on trouve dans ce paragraphe la procédure *main_control* qui contient l'essentiel du code de TEX. Celle-ci est décrite à partir du §1029 (section appelée « l'exécutif-chef »).

4 La composition : *main_control* et « exécutif-chef »

La procédure *main_control* est une immense boucle : on récupère un à un les tokens, à l'aide de la procédure *get_x_token* [§380] et selon les valeurs du mode courant et du type de token, on fait un certain nombre de choses.

Parmi les dizaines de cas qui peuvent se présenter, voici les plus fréquents [§1030] :

1. on est en mode horizontal et le token est une lettre ou un caractère de type « autre » (catcodes 11 et 12), ou alors un caractère défini par `\chardef` : on passe alors au label *main_loop* ;
2. on est en mode horizontal et le token est donné par `\char` : on lance d'abord la procédure *scan_char_num* [§434] pour obtenir la position du glyphe et ensuite on fait comme dans (1) ;
3. on est en mode horizontal et le token est la primitive `\noboundary` dont le but est de désactiver les ligatures de début de mot : on récupère un nouveau token par *get_x_token*, si celui-ci est une « lettre », un « autre », un `\chardef` ou un `\char`, alors on active le drapeau *cancel_boundary* et on retourne au début de la boucle pour traiter le token que l'on vient de récupérer ;
4. on est en mode horizontal et le token est un espace (catcode 10) : si le facteur d'espace est 1 000, c'est-à-dire si c'est une espace inter-mots normale, on va au label *append_normal_space* (qui suit immédiatement *main_loop*). Par contre, si le facteur d'espace est différent de 1 000 (par exemple si on est en typo anglo-saxonne et l'espace suit un point de fin de phrase), alors on exécute la procédure *app_space* [§1043] ;
5. on est en modes horizontal ou mathématique, et le token est un blanc explicite produit par `\␣` : alors on fait comme dans le premier cas de (4).

Les cas ci-dessus relèvent de la micro-typographie, ils gèrent la composition des glyphes et des blancs horizontaux. Nous allons poursuivre cette piste pour comprendre comme TEX compose à partir des informations que l'on lui fournit.

4.1 Le label *main_loop*

Pour arriver ici [§1030], il a fallu que l'on soit en mode horizontal et que les tokens soient de type « lettre » ou « autre » (catcodes 11 et 12). Rappelons que la principale différence entre ces deux types de token est que contrairement aux « autres », les « lettres » peuvent faire partie des noms de commande [TB45].

Le problème que l'on se pose est le suivant : comment, à partir des tokens de caractère du flux entrant, et à partir des informations fournies par la fonte courante, TEX va-t-il va construire une liste horizontale ?

Ce processus, « l'exécutif-chef », est un véritable parcours du combattant que nous nous proposons de décortiquer. La fig. 1 en montre un organigramme. Elle est complétée par la fig. 2 qui est l'organigramme d'une procédure Pascal utilisée à plusieurs endroits dans l'exécutif-chef, la *wrap_up*.

Dans la figure 1 — que le lecteur peut également télécharger en PDF [8] pour l'imprimer au format A3, ce qui en améliore sensiblement la lisibilité — on voit le point d'entrée en haut à droite, et la sortie en bas à gauche. Entre ces deux on trouve une boucle : TEX va en effet parcourir la chaîne de tokens jusqu'à arriver à un token qui n'est pas caractère.

Voici la stratégie générale : TEX utilise un « curseur », c'est-à-dire une paire de variables « gauche » *cur_l* et « droite » *cur_r*, dans lesquelles on stocke successivement les paires de caractères de chaîne à composer. S'il y a un comportement spécial entre deux caractères (un crénage ou une ligature), alors sont exécutées les opérations que l'on voit dans la partie droite de l'organigramme. Dans le cas d'une ligature, on garde en mémoire aussi bien les caractères d'origine que les glyphes de ligature obtenus. On commence par ajouter à la liste des nœuds de caractère et à chaque fois qu'une formation de ligature se termine, un nœud de ligature est créé et les nœuds de caractère qui « sautent » deviennent le *lig_ptr* de ce nœud ligature.

Avant même de commencer à décortiquer le code de l'exécutif-chef, il nous faut parler de la structure interne des fontes. Celle-ci est *a priori* d'un ennui mortel, mais sans dire quelques mots sur la manière de TEX de coder les ligatures on ne pourra jamais comprendre son code. Ouvrons donc une parenthèse et parlons de la structure d'un fichier TFM et de la manière dont ces informations sont codées dans la mémoire de TEX.

4.2 Structure de fichier TFM

Un fichier TFM [§539-548] est formé de mots de 32 bits. Après un préambule de six mots, arrivent dix tableaux : *header* (infos générales), *char_info* (indices des

8. À l'URL `http://omega.enstb.org/yannis/chief-executive.pdf`.

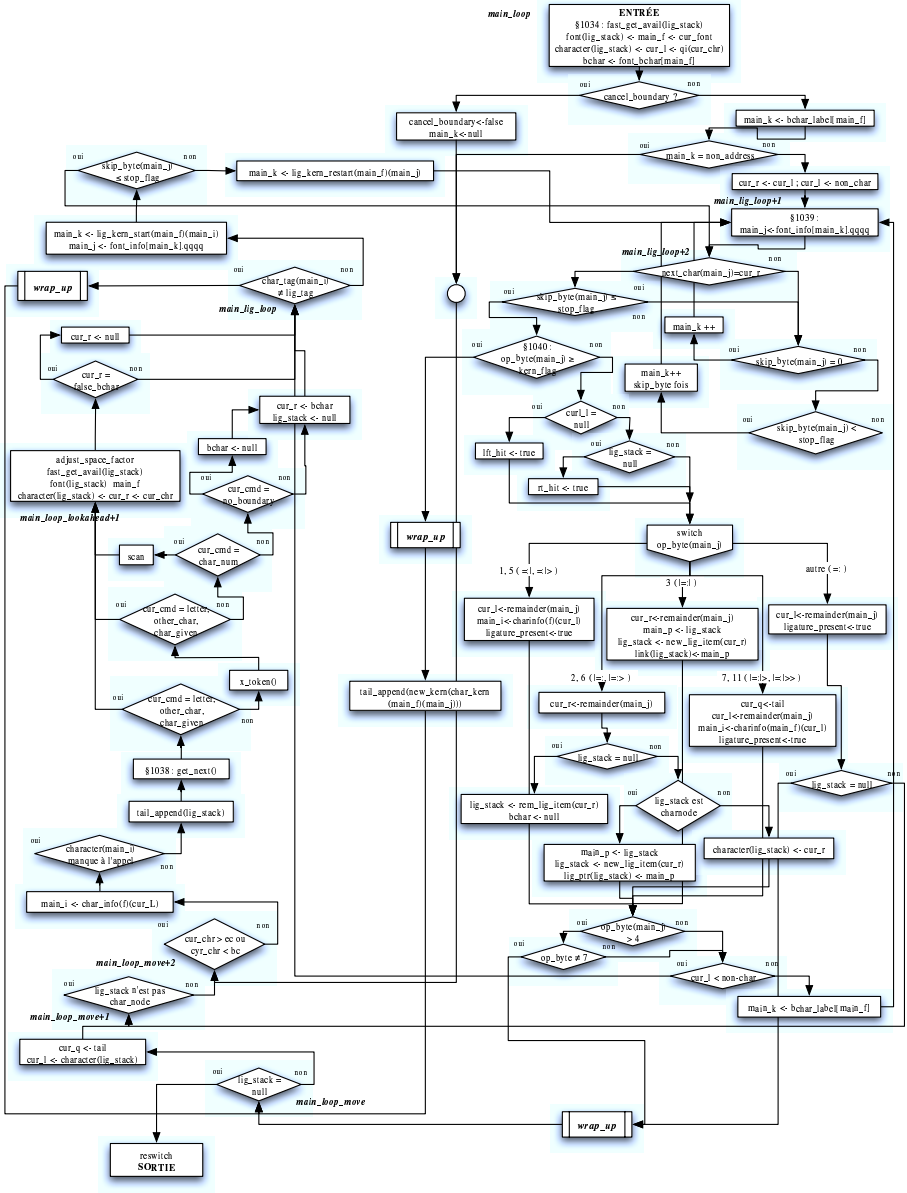


FIGURE 1 – L'organigramme de la section « exécutif chef ».

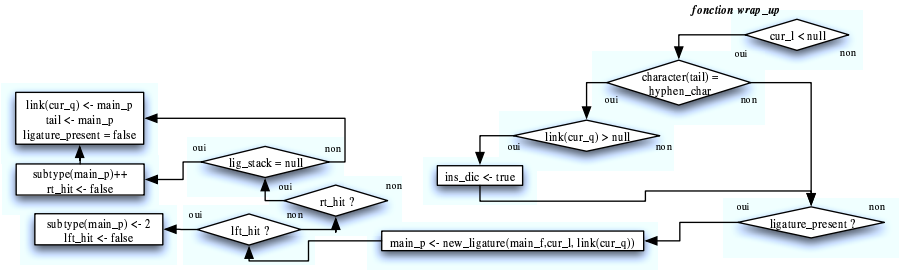
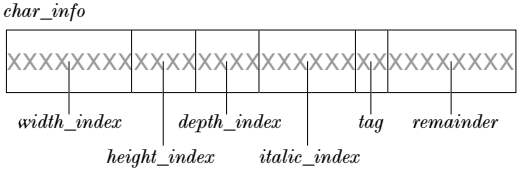


FIGURE 2 – L'organigramme de la macro *wrap_up*.

métriques d'un glyphe donné dans les autres tableaux), `width` (les différentes chasses), `height` (les différentes hauteurs), `depth` (les différentes profondeurs), `italic` (les différentes corrections italiques), `lig_kern` (les « programmes de ligatures et de crénages », que nous allons par la suite abréger « PLC »), `kern` (les différentes valeurs de paires de crénage), `exten` (les valeurs de glyphes extensibles), `param` (les paramètres de fonte).

Parmi ces tableaux, ceux qui vont nous intéresser le plus sont `char_info` et `lig_kern`. Dans le premier on trouve exactement un mot pour chaque glyphe de la fonte. Chaque mot de cette table contient les six informations suivantes :



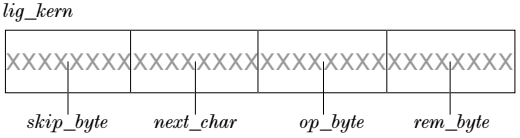
1. `width_index` (8 bits) : l'indice de la chasse du glyphe dans le tableau `width` ;
2. `height_index` (4 bits) : itou, mais pour la hauteur ;
3. `depth_index` (4 bits) : itou, mais pour la profondeur ;
4. `italic_index` (6 bits) : itou, mais pour la correction italique ;
5. `tag` (2 bits) : un drapeau qui indique la signification du dernier octet `remainder`. Il peut prendre les 4 valeurs suivantes :
   - `no_tag` : `remainder` non utilisé,
   - `lig_tag` : `remainder` contient l'indice du début d'un PLC dans le tableau `lig_kern`,
   - `list_tag` : il s'agit d'un glyphe qui fait partie d'une famille de glyphes de taille variable et `remainder` représente le code du prochain glyphe dans cette famille ;

– *ext_tag* : il s'agit d'un glyphe extensible et *remainder* est un indice d'élément du tableau *exten* ;

6. *remainder* (8 bits) : le dernier octet, dont la signification dépend de *tag*.

Hélas, la description d'un mot du tableau *lig_kern* est moins aisée[9]. Il contient les *programmes de ligatures et de crénages* (en abrégé, PLC) de certains glyphes de la fonte. Chaque PLC consiste en une série de mots : ces *instructions*.

Un mot du tableau *lig_kern*, autrement dit : une instruction de PLC, est constitué des quatre octets suivants :



1. *skip_byte* : si cet octet est supérieur ou égal à 128 (le nombre est représenté par la constante *stop_flag*), alors il s'agit de la dernière instruction du PLC ; s'il est strictement inférieur à 128, il faut sauter *skip_byte* mots pour arriver à la prochaine instruction du programme. Nous donnerons plus d'explications par la suite ;
2. *next_char* : si le glyphe suivant est *next_char* alors on effectue la ligature ou le crénage indiqué par ce mot et puis on quitte le programme ;
3. *op_byte* : cet octet peut prendre les valeurs suivantes :
   – 0 : ligature simple du type *LIG* : on remplace les glyphes g_1g_2 par le glyphe g_3,
   – 1 : ligature intelligente du type *LIG/* : $g\_1g\_2 \mapsto g\_3g\_2$ et ensuite on examine la paire g_3g_2,
   – 2 : itou, mais du type */LIG* : $g\_1g\_2 \mapsto g\_1g\_3$ et ensuite on examine la paire g_1g_3,
   – 3 : itou, mais du type */LIG/* : $g\_1g\_2 \mapsto g\_1g\_3g\_2$ et ensuite on examine les paires g_1g_3 et g_3g_2,
   – 5 : comme 1, mais on n'examine pas la paire g_3g_2 (type *LIG/>*),
   – 6 : comme 2, mais on n'examine pas la paire g_1g_3 (type */LIG>*),
   – 7 : comme 3, mais on n'examine que la paire g_3g_2 (type */LIG/>*),
   – 11 : comme 3, mais on n'examine aucune des deux paires (type */LIG/>>*),
   – supérieur ou égal à 128 : il s'agit d'une paire de crénage, dont la valeur se trouve à l'indice *256*(op_byte-128)+remainder* du tableau *kern*. Le nombre 128 est représenté par la constante *kern_flag* ;

9. Dans TEX version 2, la structure de ces mots était encore relativement rationnelle, mais avec les nouvelles fonctionnalités de TEX 3, et pour ne pas changer le format de fichier TFM lors de cette mise à jour, on a malheureusement atteint un niveau de technicité gratuite fort désagréable.

4. *rem_byte* : la valeur de cet octet est utilisée dans la formule ci-dessus, dans le cas du crénage, ou alors pour accéder à un PLC qui se trouve assez loin dans le tableau (cf. ci-dessous).

Revenons maintenant à *skip_byte* et aux sauts (d'humeur) des fontes. À quoi bon de sauter des étapes dans un PLC ? La réponse est simple : il s'agit d'une astuce d'optimisation des fichiers TFM. Knuth (et son élève Ramshaw) ont eu l'idée suivante : supposons que deux glyphes ont pratiquement les mêmes paires de crénage, alors on écrira les paires qui ne sont pas communes au début, et on demandera au programme de chaque glyphe de sauter celles qui ne le concernent pas, pour aboutir à la fin aux paires communes (que l'on n'écrira alors qu'une seule fois). On gagne ainsi quelques dizaines d'octets tout au plus, mais dans les années 70 cela était considérable.

Autre détail gênant : si la toute première instruction d'un programme de ligatures et de crénages a un *skip_byte* strictement supérieur à 128, alors le reste du programme se trouve à la position *256*op_byte+remainder* de la fonte, cela permet d'avoir des tableaux *lig_kern* très grands.

Nous savons que depuis TEX 3, les fontes peuvent posséder des ligatures ou des crénages de début et de fin de mot. La technique utilisée est différente dans les deux cas. Pour le début de mot, on a un PLC qui n'appartient à aucune position de glyphe. Pour obtenir ce programme — attention les yeux ! — il faut que la dernière instruction du tableau *lig_kern* ait un *skip_byte* de 255, et le programme commence alors à *256*op_byte+remainder*.

Pour la fin de mot, la situation est différente : on définit un glyphe comme étant le « glyphe de fin de mot » (en anglais : *right boundary character*). Quand on arrive à une fin de mot, TEX fait comme si il avait rencontré ce glyphe, et applique les ligatures correspondantes. Pour obtenir le code de ce glyphe, il faut que la toute première instruction du tableau *lig_kern* ait un *skip_byte* de 255, le code est alors donné par l'octet *next_char*.

4.3 Représentation des fontes en mémoire

Si la structure des fontes en tant que fichiers TFM peut encore nous paraître rationnelle, les choses se gâtent vraiment quand on passe à la représentation des mêmes informations dans la mémoire interne de TEX [§549-559].

Toutes les données de toutes les fontes sont stockées dans un énorme tableau appelé *font_info*. D'autre part, on a un certain nombre de tableaux d'indices de *font_info* qui correspondent aux tableaux des fichiers TFM : *char_base*, *width_base*, *height_base*, *depth_base*, *italic_base*, *lig_kern_base*, *kern_base*, *exten_base*, *param_base*. Ces tableaux sont indexés

par les numéros de fonte, ainsi pour une fonte f, le début de la table des *char_info* dans *font_info* se trouve à l'indice *char_base*[f] et le mot *char_info* correspondant au glyphe de position c se trouve dans :

font_info[*char_base*[f]+c]

Pour obtenir la chasse de ce glyphe, il faut prendre le premier octet de son *char_info* et l'ajouter à *width_base*. On obtient ainsi l'indice de la chasse correspondante dans *font_info*. La chasse du glyphe c de la fonte f se trouve donc dans :

font_info[*width_base*[f]+*font_info*[*char_base*[f]+c].*qqqq.b0*]

où .*qqqq.b0* signifie que l'on prend le premier octet du mot. On écrit *char_info*(f)(c) [§554] pour *font_info*[*char_base*[f]+c].*qqqq*, et si on note par q cette quantité, alors *char_width*(f)(q) sera la chasse de c. De même, *char_tag*(f)(q) sera le drapeau *tag* de ce glyphe, et *rem_byte*(f)(q) le dernier octet du mot *char_info*.

On a défini de telles macros également pour le PLC. Ainsi, *lig_kern_start*(f)(q) est l'indice de *font_info* qui pointe vers le début du PLC, si *skip_byte*(f)(q) est inférieur ou égal à *stop_flag*. On a une autre macro, appelée (assez maladroitement, il faut l'avouer) *lig_kern_restart*(f)(q) pour obtenir ce même début de programme lorsque *skip_byte*(f)(q)>*stop_flag*, c'est-à-dire lorsque le programme est éloigné de sa première instruction.

En ce qui concerne le PLC de début de mot et le glyphe de fin de mot, ceux-ci sont heureusement stockés de manière plus élégante dans la mémoire de TEX que dans le fichier TFM. Le début du PLC de début de mot est stocké dans *bchar_label*. Le code du glyphe de fin de mot se trouve dans *font_bchar*. Il est intéressant de noter que cette variable est de type *nine_bits*, c'est-à-dire « neuf bits » : en effet elle peut être un code normal entre 0 et 255, mais elle peut aussi prendre la valeur 256 (poétiquement appelée *non_char*[10] [§549]) si la fonte ne possède pas de glyphe de fin de mot.

4.4 Suite de l'exploration de l'exécutif-chef

Nous sommes maintenant suffisamment armés pour attaquer le code de l'exécutif-chef. Un mot d'avertissement : il ne faut pas se laisser distraire par les noms de variable du type *main_f*, *main_k*, *main_i*, etc. qui semblent importants à cause du « *main* ». Il s'agit simplement de variables temporaires utilisées à l'intérieur de cette boucle, et c'est la raison pour laquelle leurs noms sont préfixées par *main*.

10. À ne pas confondre avec les *non-caractères* d'Unicode, qui sont des positions interdites dans ce codage [3, p. 103].

On entre dans le *main_loop* par le §1034 (en haut à droite de la fig. 1). Arriver ici signifie que l'on vient de lire un token qui est la première lettre d'un mot. On commence par une série d'initialisations :
– on crée un nouveau nœud de caractère appelé *lig_stack* en utilisant *fast_get_avail*, une procédure [§122] fournissant un nouveau nœud vierge ;
– on stocke dans ce nœud les valeurs de la fonte courante et de la position du glyphe que l'on vient de lire. Mais ces valeurs sont également stockées dans les variables temporaires *main_f* et *cur_l*. Cette dernière est la « partie gauche » du « curseur » que l'on va utiliser tout au long de cette boucle pour parcourir la chaîne de caractères ;
– on stocke dans *bchar* la position du glyphe de fin de mot de la fonte courante, ou alors *non_char* si celui n'existe pas ;
– *last but not least*, on stocke dans *cur_q* un lien vers le dernier nœud de la liste horizontale. C'est ce *cur_q* qui va sauver les meubles à chaque fois que l'on va rencontrer une ligature. À suivre.

Suit un test : est-ce que la chaîne de caractères est précédée de la primitive `\noboundary` qui signifie que l'on veut éviter les éventuelles ligatures de début de mot ? Si oui, on saute toute la partie droite de l'organigramme et on se retrouve au label *main_loop_move+2*, en bas à gauche. Si non, on cherche à savoir s'il y a des interactions entre le « début de mot » et le premier glyphe.

On continue donc en stockant dans *main_k* le début du PLC de début de mot. On teste si *main_k* est effectivement une adresse de début de programme, sinon on va directement au label *main_loop_move+2*. Supposons que *main_k* existe. On déplace alors, en quelque sorte, le curseur vers la gauche : *cur_r* (la partie droite du curseur) contient désormais le premier glyphe, et *cur_l* prend la valeur, ou plutôt la « non-valeur » *non_char*.

On entre alors dans la boucle, par le label *main_lig_loop+1* [§1039] où l'on récupère la première instruction du PLC des glyphes qui se trouvent de part et d'autre du curseur. Dans notre cas on a à droite le premier glyphe et à gauche le glyphe bidon *non_char* (puisqu'il s'agit du programme de début de mot) mais par la suite nous repasserons par ici avec des véritables paires de glyphes.

On commence donc par stocker dans *main_j* la première instruction du PLC.

La partie en haut à droite de l'organigramme consiste à explorer le PLC, jusqu'à tomber sur une instruction qui concerne les deux glyphes qui entourent le curseur. Le test se trouve au label *main_lig_loop+2* : est-ce que le *next_char* de *main_j* est bien le glyphe de droite, c'est-à-dire *cur_r* ? Si non, on lit les instructions une par une, en prenant soin d'en sauter certaines si l'octet *skip_byte* est non nul.

Si une telle instruction se présente, on va au §1040, avec un nouveau test : est-ce qu'il s'agit d'une ligature ou d'une paire de crénage, autrement dit : *op_byte*(*main_j*) est-il plus grand que *kern_flag* ?

S'il s'agit d'une paire de crénage alors on exécute d'abord la macro *wrap_up* (dont l'effet est négligeable dans ce cas) et ensuite la macro :

tail_append(*new_kern*(*char_kern*(*main_f*)(*main_j*)))

qui signifie que l'on ajoute à la liste horizontale un nœud de crénage dont la valeur de crénage est égale à celle de l'instruction *main_j* du programme de la fonte *main_f*.

Si par contre il s'agit d'une ligature, alors l'aventure peut commencer. On va d'abord activer les booléennes *lft_hit* et *rt_hit* s'il s'agit d'un début ou d'une fin de mot. Ensuite on est redirigé vers d'autres opérations selon la valeur de *op_byte*, c'est-à-dire selon le type de ligature.

Dans le cas le plus simple, c'est-à-dire la ligature « non intelligente » *LIG* de TEX 2, on place le glyphe résultant de la ligature dans *cur_l*. Cela peut paraître bizarre parce qu'à cet instant on a dans *cur_r* (et dans *lig_stack*) le glyphe de droite du curseur, et dans *cur_l* le glyphe de ligature. Mais cela va s'arranger par la suite. On active également la booléenne *ligature_present*, sans laquelle la macro *wrap_up* qui va effectivement créer le nœud de ligature, n'aura aucun effet.

Poursuivons maintenant le cheminement du programme. Suit un test qui nous demande si *lig_stack* est nul. Un des cas où cela peut se produire est quand on est en fin de mot. Supposons que ce n'est pas le cas. Alors on est redirigés vers *main_loop_move+1* [§1036], en bas à gauche de l'organigramme.

On commence alors à remonter la partie gauche de l'organigramme. On se prépare à récupérer le prochain token, sans passer par la grande boucle de TEX, et si celui-ci donne lieu à un nœud de caractère alors on déplace le curseur vers la droite.

Premier test, assez énigmatique : est-ce que *lig_stack* est bien un nœud de caractère ? Si oui, on peut avancer. Prochain test : est-ce que le code du glyphe courant, c'est-à-dire *cur_chr* est bien présent dans la fonte ? Si non, on a un avertissement et le jeu s'arrête.

Prochaine étape : on stocke dans *main_i* le mot *char_info* de *cur_l* (c'est-à-dire le glyphe à gauche du curseur, dans notre cas il s'agit du glyphe de ligature). On teste si ce glyphe se trouve bien dans la fonte, et si oui...

Le lecteur s'attend à ce que l'on ajoute ce glyphe à la liste horizontale. Eh bien pas du tout, c'est *lig_stack* que l'on ajoute à la liste horizontale, et *lig_stack* contient précisément le glyphe original — que l'on ne s'attendait pas de voir dans la liste horizontale puisqu'il est censé produire une ligature. Mais Knuth est un joueur habile : il met les glyphes originaux dans la liste horizontale tant qu'il n'est pas sûr que le nœud de ligature va effectivement être créé. Après tout, il se peut bien qu'après cinq glyphes « consommés » pour produire une ligature, on tombe sur un glyphe absent du TFM ; on serait alors bien embarrassé si on n'avait pas à

ce moment-là la possibilité de changer d'avis et de ne pas produire de nœud de ligature du tout. Assez anticipé, revenons à notre processus.

On exécute par la suite *get_next* () [§1038], c'est la procédure qui fournit un nouveau token sans faire d'expansion. S'il s'agit d'un nœud de caractère on passe au label *main_loop_lookahead+1* (à gauche, à mi-hauteur de l'organigramme). S'il s'agit d'une commande, on en fait l'expansion et on vérifie de nouveau si, oui ou non, on a créé un nœud de caractère. De même si la commande est un `\char` on obtient le glyphe correspondant et on avance.

Si l'on arrive à *main_loop_lookahead+1* c'est que l'on a fini par obtenir un nouveau nœud de caractère. À ce stade on donne la valeur du nouveau glyphe obtenu à *cur_r* et à *lig_stack* celle du nouveau glyphe. Après un test pour savoir si *cur_r* est le glyphe de fin de mot (dans lequel cas on change la valeur de celui-ci en *non_char*), on passe à l'autre label primordial de cette boucle : *main_lig_loop* (en haut à gauche).

Celui-ci commence par un test : est-ce que le champ *tag* du *char_info* du glyphe que l'on a stocké dans *cur_l* (dans notre cas : la ligature) indique que celui-ci possède un PLC ?

Si oui, alors le ligaturage n'est pas fini et on recommence la boucle, en stockant dans *main_k* l'indice du PLC de *cur_r* et dans *main_j* la première instruction de celui-ci. Et on se retrouve donc au label *main_lig_loop+2*, en haut à droite de l'organigramme, comme avant.

Si non, alors on lance la macro *wrap_up* (cf. fig. 2). Que fait celle-ci ? Après quelques tests dont le but est de déceler un *hyphen_char* (c'est-à-dire le glyphe choisi comme trait de césure), on arrive au test crucial : est-ce que *ligature_present* est vrai ? Autrement dit : est-ce qu'on a une ligature à traiter ?

Dans notre cas c'est oui, et on crée donc un nouveau nœud de ligature dont la fonte est *main_f*, le glyphe *cur_l* et dont le *lig_ptr* pointe vers *link*(*cur_q*). Le voilà de nouveau, ce *cur_q* que nous avons rencontré lors de l'initialisation. Et l'on voit maintenant sa double utilité : même si entre temps nous avons ajouté un nœud de caractère à la liste horizontale courante, *cur_q* n'a pas bougé, il nous permettra donc de placer le nœud ligature au bon endroit et d'ignorer par la même occasion les nœuds caractère insérés. En même temps, *cur_q* pointe vers les nœuds de caractère « remplacés » par le nœud ligature, on stocke donc cette information dans le nœud de ligature nouvellement créé, et ainsi on garde *ad vitam aeternam* la « décomposition » de la ligature en glyphes simples.

On utilise une nouvelle variable temporaire *main_p* pour stocker l'adresse du nouveau nœud. Ensuite on essaie de se remémorer s'il s'agissait d'un début ou d'une fin de mot. Si oui, cela a comme effet de changer le *subtype* du nœud ligature. Et vient le moment tant attendu : *link*(*cur_q*)←*main_p* et *tail*←*main_p*, autrement dit :

on insère notre nœud ligature à l'endroit de la liste horizontale où l'on était quand on a donné une valeur à *cur_q*. Et puisque le traitement de la ligature est fini, on désactive *ligature_present*.

Reprenons le processus, en nous servant d'un exemple : le mot « fil », dans une fonte qui possède une ligature ordinaire « fi » et qui ne possède pas de PLC de début ou de fin de mot. Dans la suite nous allons noter par x|y la situation du « curseur », autrement dit le fait qu'à l'instant t, les caractères *cur_l* et *cur_r* sont resp. x et y. Allons-y :

1. Initialisation ***main_loop*** : on crée un nœud  que l'on stocke dans *lig_stack*. On stocke le caractère f dans *cur_l*, le curseur devient donc f|$\emptyset$, *cur_q* pointe vers la fin actuelle de la liste horizontale, notons-la **h** ;
2. il n'y a pas de \noboundary et donc *main_k* pointe vers le PLC de début de mot, c'est-à-dire vers *null* puisqu'on n'a pas de tel programme ;
3. on teste si *main_k* pointe vers quoi que ce soit d'intéressant, il s'avère que non. On entre donc dans la boucle à partir du label ***main_loop_move+2*** (en bas à gauche de l'organigramme) ;
4. ***main_loop_move+2*** : *main_i* récupère le *char_info* de « f » ;
5. on ajoute à la liste horizontale le nœud  , elle devient donc **h**  ;
6. on récupère le token suivant, c'est « i » et son catcode est bien 11 ;
7. ***main_loop_lookahead+1*** : on crée un nouveau nœud  et on le stocke dans *lig_stack*. *cur_r* prend la valeur i, le curseur est donc f|i ;
8. ***main_lig_loop*** : on teste si « f » possède un PLC, la réponse est oui ;
9. on récupère donc dans *main_j* la première instruction de ce programme et on se retrouve au label ***main_lig_loop+2*** ;
10. après avoir parcouru quelques instructions de ce programme on arrive à celle qui concerne la paire (« f », « i »), c'est-à-dire celle où *next_char*(*main_j*) est égal à *cur_r*. On se demande si c'est une paire de crénage ou une ligature, c'est le deuxième cas. On arrive donc au *switch* :
11. *switch* : le *op_byte* de l'instruction est 0, on stocke donc dans *cur_l* le code du glyphe « fi » et on active *ligature_present*. Le curseur est maintenant fi|i ;
12. le *lig_stack* contient toujours  et donc le test *lig_stack=null* ? échoue et on part de nouveau vers ***main_loop_move+1*** :
13. ***main_loop_move+1*** : *lig_stack* est bien un nœud caractère et on part donc vers ***main_loop_move+2*** :

14. ***main_loop_move+2*** : on place le `char_info` de « fi » dans `main_i` et on ajoute `lig_stack` à la liste horizontale courante qui devient alors **h** [f CHR] [i CHR] ;
15. on récupère le token suivant, c'est « l », son catcode est bien 11 ;
16. ***main_loop_lookahead+1*** : on re-initialise `lig_stack` et on y stocke le nœud [l CHR]. `cur_r` prend la même valeur, le curseur est maintenant `fi|l` ;
17. on teste si « fi » a un PLC, la réponse est : non. On va donc au label ***main_loop_wrapup*** :
18. ***main_loop_wrapup*** : on lance la macro `wrap_up`. Celle-ci teste la booléenne `ligature_present`, qui est vraie. On crée alors un nouveau nœud de ligature [fi → (f,i) LIG] et on remplace tout ce qui a été ajouté à la liste horizontale après `cur_q` par celui-ci : elle devient alors **h** [fi → (f,i) LIG] ;
19. ***main_loop_move*** : on teste si le `lig_stack` est vide, ce n'est pas le cas puisqu'on y a placé [l CHR] ;
20. on ré-initialise `cur_q` (qui pointe donc vers **h** [fi → (f,i) LIG]) et `cur_l` devient « l ». Le curseur est maintenant `l|l` (!!) ;
21. ***main_loop_move+2*** : on place le `char_info` de « l » dans `main_i` et on ajoute `lig_stack` à la liste horizontale courante qui devient alors **h** [fi → (f,i) LIG] [l CHR] ;
22. on récupère le token suivant, c'est sans doute un blanc puisqu'on a déjà récupéré tout le mot « fil ». Son catcode n'est certainement pas 11 ou 12 ;
23. on ne passe donc pas par ***main_loop_lookahead+1*** et à la place on donne à `cur_r` la valeur `bchar` (dans notre cas : `non_char` puisqu'on n'a pas de glyphe de fin de mot), le curseur est donc `l|`$\emptyset$. D'autre part on annule `lig_stack` ;
24. ***main_lig_loop*** : on teste si « l » a un PLC, ce n'est pas le cas. On part donc de nouveau vers ***main_loop_wrapup*** ;
25. ***main_loop_wrapup*** : on entre dans la macro `wrap_up`, mais cette fois-ci `ligature_present` est faux. On arrive directement à ***main_loop_move***, en bas à gauche de l'organigramme ;
26. ***main_loop_move*** : on teste si `lig_stack` est nul, il l'est effectivement. On sort de la boucle en allant au label `reswitch`.

À la fin de ces opérations on a obtenu une liste horizontale **h** [fi → (f,i) LIG] [l CHR]. Et bingo !

5 Le paragraphage

Nous avons vu comment créer des listes horizontales à partir des tokens du flot d'entrée. Maintenant passons à l'étape suivante : à partir d'une liste horizontale, former un paragraphe, c'est-à-dire une suite de boîtes (les lignes de texte) qui deviennent nœuds de la liste verticale environnante.

5.1 Présentation du problème

On a donc une liste horizontale qui comporte toutes sortes de nœuds et que l'on veut couper en lignes de manière optimale aussi bien en ce qui concerne le temps de calcul que la qualité du résultat. Pour TEX, un bon résultat est un paragraphe visuellement homogène où certaines lignes peuvent être sacrifiées pour obtenir une mise en page globale optimale.

TEX va couper en trois endroits possibles : des nœuds de glue, des nœuds de pénalité (si cette pénalité n'est pas infinie, c'est-à-dire interdits de coupure) et des nœuds discrétionnaires. On appelle ces nœuds les « coupures légales ». La méthode sera la suivante : à partir de coupures légales on va construire une nouvelle liste de nœuds, appelés *nœuds actifs*. Un nœud actif est plus qu'une coupure légale, c'est une « coupure faisable », un véritable candidat à la coupure. Ce qui les distingue c'est qu'ils se trouvent aux bons endroits pour former des lignes de bonne longueur et de composition acceptable.

On va donc commencer par le début de paragraphe que l'on déclare « nœud actif », et ensuite s'éloigner jusqu'à être à une distance « acceptable » pour former une ligne. Si l'on trouve un nœud légal à cet endroit, il devient nœud actif. Quand on sera trop loin de notre nœud actif placé à l'origine, on le désactive, il devient alors « nœud passif ». On continue, et dès que l'on est de nouveau à une distance acceptable d'un nœud actif, on crée des nouveaux nœuds actifs — de même, dès que l'on est trop loin d'un nœud actif, on le désactive. En continuant ainsi, on arrive à la fin de paragraphe, que l'on déclare également nœud actif.

Mais qu'est-ce qui rend une distance « acceptable » ?

Nous savons que la glue peut être extensible (à plusieurs ordres de grandeur) et que les blancs inter-mot sont également susceptibles de ce dilater et de se comprimer. De ce fait, la largeur totale d'une suite de nœuds peut être assez variable, on peut calculer ses maxima et minima. Si la justification souhaitée se trouve entre les maxima et minima de largeur totale d'une chaîne de nœuds, on dira que les extrémités de la chaîne sont à une distance acceptable.

Arrivés à la fin du paragraphe on se retrouve avec un grand nombre de nœuds passifs ou actifs, comment choisir les bons ? En fait, à chaque fois que l'on crée un nouveau

nœud actif, on lui associe le nœud actif qui a servi à le créer, c'est-à-dire celui qui était à une distance acceptable de lui. Ainsi, arrivé à la fin du paragraphe on a au moins une, et souvent plusieurs manières de remonter jusqu'au début. Ce sont les différentes versions du paragraphe, toutes acceptables.

Toutes acceptables, mais comment choisir la meilleure ?

C'est que l'on tient pas seulement compte des extrema de largeur d'un intervalle entre deux nœuds actifs, mais que l'on va également mesurer la laideur (*badness*) de chaque composition obtenue. Et mis à part les variations de largeur, il y a aussi des pénalités qui sont appliquées à certaines situations : pénalité de césure, pénalité de double césure, pénalité de ligne orpheline, etc. TEX peut être d'une grande sévérité et distribuer des pénalités à tout bout de champ. En combinant laideur et pénalités on obtient une autre mesure de qualité de composition, appelée « démérite » (*demerits*).

À chaque fois que l'on crée un nœud actif à partir d'un autre nœud actif, on attache à cette paire de nœuds un démérite. Chaque version du paragraphe correspond donc à une somme de démérites, le « démérite total ». On prendra la version du paragraphe qui a le moindre démérite total.

Pour mieux comprendre le fonctionnement du moteur de paragraphage, prenons un exemple : le premier paragraphe de [3], composé en lmr10, dans une justification de 10 cm :

L'homme écrit. Et parmi le grand nombre d'outils utilisés pour écrire, le dernier en date et le plus complexe n'est autre que l'ordinateur. L'ordinateur qui est en même temps outil d'écriture et de lecture, lieu de stockage, moyen de transmission. Il est devenu un véritable espace à l'intérieur duquel vit le texte, espace qui, comme l'avait prédit, entre autres, MacLuhan, a réussi à s'affranchir des barrières géographiques et à englober toute la planète.

Nous avons noté les nœuds actifs par un petit trait suivi du numéro de nœud. TEX va parcourir la liste horizontale jusqu'à trois fois de suite : la première sans césure, la deuxième avec césure et la troisième avec des paramètres exceptionnels pour « sauver les meubles ». Commençons directement par la deuxième passe, qui est la plus intéressante.

TEX dispose d'un moyen de nous dévoiler exactement ce qu'il est en train de faire, il s'agit de la commande `\tracingparagraphs=1`. Voici ce que l'on peut lire dans le fichier log, à condition d'avoir utilisé cette commande :

```
@secondpass
[]\x L'homme écrit. Et parmi le grand nombre d'outils uti-li-
```

Sans le dire explicitement, un premier nœud actif a été créé, au début du paragraphe : le nœud 0. Nous parcourons donc la liste horizontale pour trouver un nœud légal qui soit à une distance acceptable du nœud 0. Le premier qui satisfasse à cette condition est le nœud de césure n° 1 « utili*sés » :

```
@\discretionary via @@0 b=158 p=50 d=40724
@@1: line 1.0- t=40724 -> @@0
```

Voici ce que signifient ces deux lignes de code : la première nous dit que l'on a trouvé un nœud discrétionnaire à distance acceptable du nœud actif 0, la laideur (« b » comme *badness*) est alors de 158, la pénalité de 50 (car c'est une césure) et le démérite de 40 724 (une quantité non négligeable).

La deuxième ligne nous dit que nous venons de créer un nouveau nœud actif, le nœud 1, qui termine la ligne 1 avec une qualité de composition « 0 » (qui signifie : « composition extrêmement lâche »). Le démérite total pour arriver au nœud courant est de 40 724, et pour avoir ce démérite il faut (évidemment) passer par le nœud 0. Continuons :

```
sés
@ via @@0 b=24 p=0 d=1156
@@2: line 1.1 t=1156 -> @@0
```

Le prochain nœud est donc le blanc qui suit le mot « utilisés ». On est toujours à une distance raisonnable du nœud 0 et cette fois-ci la situation est bien meilleure que pour le nœud 1 : la laideur est de 24, il n'y a aucune pénalité et le démérite est de 1 156 seulement. Le nœud 2 termine donc, lui aussi, la ligne 1, mais cette fois-ci avec une qualité de « 1 » (c'est-à-dire : « composition lâche »).

```
pour écrire, le der-nier en date et le plus com-plexe n'est autre
@ via @@1 b=0 p=0 d=10100
@ via @@2 b=75 p=0 d=7225
@@3: line 2.1 t=8381 -> @@2
```

On continue, et le prochain nœud à une distance acceptable d'un nœud actif quelconque est le blanc qui suit le mot « autre ». Le nœud 0 est hors d'atteinte, on le désactive. Mais les nœuds 1 et 2 sont tous les deux à une distance acceptable de

notre nœud courant : si l'on terminait ici une ligne commencée par le nœud 1, on aurait alors une laideur de 0 (!!), une pénalité nulle et un démérite de 10 100 ; par contre si l'on terminait ici une ligne commencée par le nœud 2, on aurait une laideur de 75, une pénalité également nulle, et un démérite de 7 225, ce qui légèrement mieux.

Le deuxième cas est plus intéressant, et dans les deux cas on aurait des lignes *de même qualité de composition*. TEX décide alors d'oublier le premier cas. On crée un nouveau nœud 3, qui produit une ligne de qualité 1 quand il suit le nœud 2. Le démérite total est de 8 381. Continuons :

```
que
@ via @@2 b=0 p=0 d=100
@@4: line 2.2 t=1256 -> @@2
l'ordinateur. L'ordinateur qui est en même temps ou-til d'écriture
@ via @@4 b=0 p=0 d=100
@@5: line 3.2 t=1356 -> @@4
et de lec-ture, lieu de sto-ckage, moyen de trans-mis-sion. Il est
@ via @@5 b=97 p=0 d=11449
@@6: line 4.1 t=12805 -> @@5
de-
@\discretionary via @@5 b=1 p=50 d=2621
@@7: line 4.2- t=3977 -> @@5
venu un vé-ri-table es-pace à l'intérieur du-quel vit le texte, es-
@\discretionary via @@6 b=33 p=50 d=4349
@@8: line 5.1- t=17154 -> @@6
pace
@ via @@6 b=6 p=0 d=256
@ via @@7 b=4 p=0 d=196
@@9: line 5.2 t=4173 -> @@7
qui, comme l'avait pré-dit, entre autres, Ma-cLu-han, a réussi
@ via @@8 b=0 p=0 d=100
@@10: line 6.2 t=17254 -> @@8
```

On continue donc notre parcours en créant successivement des nœuds actifs 4, 5, 6, 7, 8, 9, 10. Observons, en passant, la mauvaise coupure du nom « MacLuhan » par les motifs de césure français…

Autre cas intéressant :

```
à
@ via @@8 b=21 p=0 d=10961
```

```
@ via @@9 b=100 p=0 d=22100
@@11: line 6.0 t=26273 -> @@9
@@12: line 6.3 t=28115 -> @@8
```

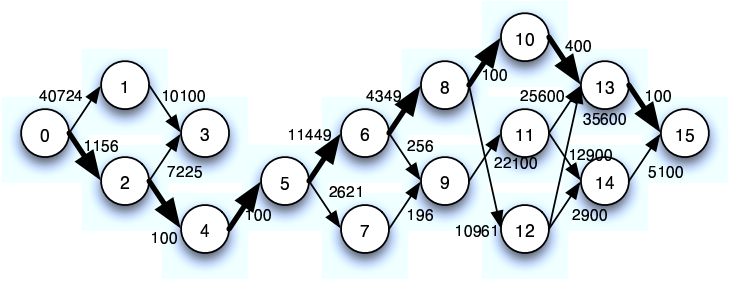
Il s'agit du blanc qui suit le mot « à » (dans : « a réussi à s'affranchir »). De nouveau on est à distance raisonnable de plus d'un nœud actif : il y a aussi bien le nœud 8 (la césure « es*pace ») que le 9 (le blanc qui suit « espace ») qui sont des candidats. Mais les qualités de composition des deux lignes ainsi produites sont différentes, on crée donc *deux* nouveaux nœuds actifs, associés au même nœud de glue de la liste horizontale. Le nœud 11, utilisé comme successeur du nœud 9 produit une ligne 6 de qualité 0 (« extrêmement lâche »). Le nœud 12, successeur du nœud 8, produit une ligne 6 de qualité 3 (« serrée »). Poursuivons :

```
s'affranchir des bar-rières géo-gra-phiques et à en-glo-ber toute
@ via @@10 b=10 p=0 d=400
@ via @@11 b=150 p=0 d=25600
@ via @@12 b=150 p=0 d=35600
@@13: line 7.2 t=17654 -> @@10
pla-
@\discretionary via @@11 b=10 p=50 d=12900
@\discretionary via @@12 b=10 p=50 d=2900
@@14: line 7.2- t=31015 -> @@12
nète.
```

On a bien de choix pour les nœuds actifs qui suivent : le nœud 13 est le blanc qui suit le mot « la », et le nœud 14 est la césure « pla*nète ». Le premier peut former des lignes avec les nœuds 10, 11 et 12, avec des laideurs et des démérites bien différents, mais toujours en restant dans la qualité de composition 2 (« compo décente »). Le nœud 14 peut se combiner avec les 11 et 12, et la qualité est encore 2. Enfin, arrivés au terme du paragraphe nous créons un dernier nœud actif artificiel :

```
@\par via @@13 b=0 p=-10000 d=100
@\par via @@14 b=0 p=-10000 d=5100
@@15: line 8.2- t=17754 -> @@13
```

La dernière ligne étant creuse, elle est évidemment de qualité 2 (« compo décente »). Nous pouvons dessiner le graphe des différentes combinaisons de nœuds obtenues par les données ci-dessus :



Il s'agit d'un graphe acyclique (= où il n'y a aucun parcours possible qui nous ramène au même endroit) et les démérites que l'on a noté à côté des flèches peuvent être considérés comme formant une « distance ». Le problème revient alors à trouver le plus court chemin entre deux nœuds d'un graphe acyclique, un problème dont Knuth donne la solution dans son *Art de programmer l'ordinateur* et une implémentation pratique dans le moteur de paragraphage de TEX. Nous avons noté dans la figure ci-dessus par des traits plus épais ce chemin le plus court, et c'est celui qui a effectivement été choisi par TEX pour la mise en page du paragraphe de texte.

5.2 Le code

Nous sommes maintenant armés pour attaquer le code du moteur de paragraphage.

Supposons que notre paragraphe de texte se termine par une ligne blanche ou une primitive `\par`. Pour TEX la ligne blanche ou la commande `\par` produisent un token *par_end* (fin de paragraphe). La boucle principale *main_loop* qui lit les tokens nous propose justement dans le §1094 le cas « mode horizontal + *par_end* ». On exécute alors la procédure *end_graf* [§1096].

Cette procédure teste d'abord si la liste horizontale courante est nulle (autrement dit : si *head* est égal à *tail*) et ensuite lance deux procédures : *line_break* et *normal_paragraph*. La deuxième est juste une ré-initialisation de certaines valeurs.

La première, *line_break*, est le véritable noyau dur de TEX.

5.2.1 line_break

Voici comment on procède : on va créer un nouvelle liste de nœuds, associés à certains nœuds de la liste horizontale : des *nœuds de coupure* [§817]. Ceux-ci contiennent trois informations : un pointeur vers l'endroit où la coupure est possible (cet endroit est un nœud de glue, un nœud mathématique, un nœud de pénalité ou un nœud discrétionnaire), le numéro d'ordre de la ligne ainsi produite et un paramètre entre 0 et 3 qui indique la qualité de composition de la ligne : *tight_fit* (très

serrée), *loose_fit* (beaucoup de blancs, mais on reste dans les limites de l'acceptable données par la fonte), *decent_fit* (composition décente), *very_loose_fit* (trop de blancs).

La structure générale de *line_break* [§815] est la suivante :

1. les préliminaires (§816) ;
2. on trouve les endroits de coupure optimaux (§863) ;
3. on coupe la ligne horizontale à ces endroits pour obtenir des lignes, on justifie chacune de ces lignes, on les place dans des boîtes, et on attache les boîtes à la liste verticale en cours (§876) ;
4. on supprime les nœuds de coupure pour libérer la mémoire (§865).

Bien évidemment, l'étape la plus intéressante est la deuxième. Mais on en est encore loin.

TEX va parcourir la liste horizontale jusqu'à trois fois : la première fois on ne fait aucune césure et l'on considère qu'une ligne est acceptable si sa « laideur » (en anglais : *badness*) n'excède pas *pretolerance* (spécifiée par la primitive `\pretolerance`) ; la deuxième fois on passe par l'algorithme de césure pour trouver plus de points de coupure et on prend comme laideur acceptable *tolerance* ; la troisième fois on essaie de sauver les meubles en augmentant la glue infinie placée des deux côtés du paragraphe (`\leftskip` et `\rightskip`) à l'aide de la commande `\emergencystretch`. Cette dernière vise à relativiser les laideurs des lignes et d'éviter ainsi un problème connu de TEX≤ 3 : préférer une ligne très mauvaise à plusieurs lignes médiocres. Enfin, si rien ne marche, on revient aux résultats de la deuxième passe, on prend une justification de ligne plus grande que la souhaitée et on insère dans la marge « boîte noire de dépassement » (« *overfull box* »), ce qui a déjà consterné plus d'un TEXiste…

Commençons les préliminaires. Tout d'abord [§816] on ajoute un nœud de glue de fin de paragraphe (glue donnée par `\parfillskip`) à la fin de la liste horizontale (en supprimant préalablement des éventuels espaces ou glues qui puissent s'y trouver). Cela explique le fait que l'on puisse allègrement ajouter des blancs ou des `\hskip` à la fin d'un paragraphe, sans changer en rien le résultat de la composition.

Les nœuds de coupure se matérialisent sous deux formes : les « actifs » et les « passifs ». L'information que contiennent ces deux types de nœud est différente — au fur et à mesure du parcours du paragraphe, on crée des nœuds actifs qui ensuite deviennent passifs.

Un nœud actif est long de trois mots. Il comporte les six informations suivantes [§819] :

1. *link* est un pointeur vers le prochain nœud actif. Le dernier nœud actif pointe vers le premier (appelé *active*), la chaîne des nœuds de coupure est donc circulaire ;

2. *break_node* pointe vers le nœud passif correspondant ;
3. *line_number* est le numéro d'ordre de la ligne qui suit ce point de coupure ;
4. *fitness* indique la qualité de composition (*tight_fit*, etc.) ;
5. *type* est soit *hyphenated*, soit *unhyphenated*, selon le fait si le nœud de coupure en question est un nœud discrétionnaire ou pas ;
6. *total_demerits* est le démérite total minimum obtenu en parcourant les nœuds de coupure à partir du début du paragraphe et jusqu'au nœud courant.

Un nœud passif [§821] contient des informations tout à fait différentes qui visent à établir un graphe de coupures faisables du paragraphe :

1. *link* pointe vers le nœud passif précédent, ou alors il est nul ;
2. *cur_break* pointe vers le nœud correspondant dans la liste horizontale ;
3. *prev_break* pointe vers le nœud passif qui devrait précéder celui-ci dans le paragraphe coupé de manière optimale ;
4. *serial*, le numéro d'ordre du nœud, est utilisé uniquement par `\tracingparagraphs=1` à des fins de traçage.

Les nœuds actifs servent à faire des calculs de largeur de lignes potentielles. À l'issue de ces calculs, on obtient des nœuds passifs avec des champs *prev_break*. Arrivés à la fin du paragraphe, nous obtenons un nœud actif avec un *total_demerits* minimal, il suffit de suivre la chaîne des nœuds passifs en parcourant les *prev_break* pour obtenir tous les points de coupure optimaux.

Mais comment les calculs se font-ils ? Pour nous aider à calculer les largeurs de tronçons de ligne entre deux nœuds actifs, on introduit encore un type de nœud, appelé *nœud* δ. Un nœud δ q inséré dans la chaîne circulaire de nœuds actifs entre les nœuds p et r contient la largeur de la chaîne de glyphes produits par notre liste horizontale entre les nœuds de coupure correspondants.

Mais comment calculer la largeur d'une liste de nœuds alors qu'entre ceux-ci on trouve de la glue fixe ou variable, avec différents ordres de grandeur ? On fait une sorte de calcul vectoriel, où chaque ordre de grandeur est un vecteur indépendant. Un nœud δ occupe, ni plus ni moins, 7 (!) mots, dont un mot pour la largeur idéale, quatre mots pour la dilatation maximale en termes de *pt*, *fil*, *fill*, et *filll*, et un mot pour la contraction maximale (en *pt*, puisque la contraction infinie n'est pas autorisée dans un paragraphe).

Pour faire nos calculs, nous allons utiliser quatre variables globales. Comme leurs valeurs peuvent contenir différents degrés d'infinité de glue, ces variables sont en fait des tableaux à six entrées : une pour la taille idéale, quatre pour les différents degrés d'infinité de dilatation, une pour la contraction. Ces quatre variables « tableaux » sont *active_width*, *cur_active_width*, *background* et *break_width*. Pour la

première, on écrit *act_width* (en abrégé) lorsqu'il s'agit de la première entrée du tableau uniquement (c'est-à-dire la dimension fixe).

C'est la variable *cur_p* [§862] qui va parcourir la liste horizontale à la recherche de nœuds de coupure légaux. La boucle de ce parcours de la liste horizontale se trouve dans la section §863. On a une variable *treshold* qui va prendre les valeurs de pré-tolérance ou de tolérance, selon le fait si c'est une première ou une deuxième/troisième passe.

On crée un premier nœud actif en début de paragraphe [§864], avec un démérite total nul. Ensuite on place *cur_p* au début de la liste horizontale. Il y a une autre variable, *prev_p*, qui va suivre *cur_p* au pas, mais avec un nœud de retard. C'est ce nœud qui nous permettra de décider si un nœud de glue est une coupure de ligne légale ou pas. On commence en mettant *prev_p* égal à *cur_p*.

Et ensuite, tout naturellement, on entre dans une boucle où les nœuds de la liste horizontale sont examinés un à un. Le cas le plus fréquent est celui de nœud de caractère, il est traité en premier [§867] : on met *prev_p* égal à *cur_p*, on augmente *act_width* d'une quantité égale à la chasse du glyphe que l'on vient de lire, et ensuite on passe au nœud suivant par un *cur_p* ← *link*(*cur_p*).

Les autres cas de nœuds sont rassemblés dans un switch [§866] :
- si c'est un nœud de liste (horizontale ou verticale) ou de filet, on fait comme pour les glyphes, c'est-à-dire que l'on augmente *act_width* de la chasse du nœud (la chasse d'une liste est en fait la chasse de la boîte qui contient la liste, celle-ci n'a pas de quantité infinie) ;
- si c'est un nœud quésaco, on le traverse [§1362] ;
- si c'est un nœud de glue [§868], alors c'est un point de coupure légal si et seulement si *prev_p* n'est pas nœud de glue, de pénalité, de crénage explicite, ou un nœud mathématique. Reste à trouver si c'est une coupure « faisable », pour en faire un nœud actif. Pour le savoir on passe par la procédure *try_break*() que nous verrons par la suite. Ensuite on augmente toutes les entrées de *active_width* par la quantité de glue apportée par le nœud.
  Si c'est notre deuxième passe, alors c'est à cet endroit que l'on applique l'algorithme de césure au mot, c'est-à-dire à la sous-liste de nœuds de caractère qui suit immédiatement le nœud de glue [§894]. Cet algorithme (que nous verrons plus loin) va insérer des nœuds discrétionnaires aux endroits de césure potentielle.
  Le fait que l'algorithme est appliqué uniquement aux mots qui suivent un nœud de glue explique certains problèmes que l'on rencontre souvent : ainsi, le premier mot d'un paragraphe n'est pas coupé puisqu'il n'est pas précédé par un nœud de glue, un mot précédé directement d'une boîte n'est jamais coupé, la partie d'un mot qui suit une lettre accentuée formée par la primitive `\accent` n'est jamais coupée, etc. ;

– si c'est un nœud de crénage implicite (c'est-à-dire obtenu par la fonte), alors on fait comme pour les nœuds de caractère : on ajoute sa chasse et on avance ;
– si c'est un nœud de crénage explicite (donné par un `\kern`), alors on a un point de coupure légal si et seulement si le nœud suivant est de type glue. On lance donc *try_break*`()`. Et quelque soit le résultat, on ajoute la chasse et on avance ;
– si c'est un nœud de ligature alors on fait comme pour les nœuds de caractère ;
– si c'est un nœud discrétionnaire, alors [§869] des choses bien intéressantes se passent. Rappelons qu'un nœud discrétionnaire peut provenir d'une commande du type `\discretionary{k-}{k}{ck}` qui porte trois arguments : les deux sous-chaînes produites en cas de césure, et la chaîne « au repos », c'est-à-dire en cas de non césure. Un nœud discrétionnaire contient deux pointeurs *pre_break* et *post_break* vers les listes horizontales des deux derniers arguments, et un nombre : le nombre de nœuds contenus dans le premier argument (et que l'on supprime en cas de coupure).
  Le problème est donc le suivant : selon que l'on coupe ou pas un mot à un endroit donné, la commande `\discretionary` peut produire des nouvelles listes horizontales avec un apport de chasse significatif. Comment intégrer cette variation de chasse dans l'algorithme de paragraphage ?
  Il s'agit de faire « comme si il y a avait coupure », et de tenir compte des listes *pre_break* et *post_break*, en ignorant les nœuds que l'on supprimerait en cas de coupure. Mais cela ne doit se faire que pour les calculs qui impliquent le nœud actif courant.
  Voici comment on fait cette simulation [§869] : on stocke le nœud de liste horizontale *pre_break* dans *s*, et on initialise une variable *disc_width* (chasse de discrétionnaire). Si *s* est nul, on lance *try_break* (pour voir si le point de coupure est faisable), avec une pénalité de *ex_hyphen_penalty*. Si *s* n'est pas nul, on calcule la chasse totale de la liste horizontale initiée par *pre_break* et on la met dans *disc_width*. Ensuite on augmente *act_width* de *disc_width* (on simule donc la coupure) et on essaie de nouveau *try_break* avec une pénalité de *hyphen_penalty*. Le test étant fait, on veut rétablir la situation : on soustrait alors *disc_width* de *act_width* et on avance, en ajoutant à *act_width* la chasse des nœuds qui forment le premier argument de `\discretionary` ;
– si c'est un nœud mathématique, on fait comme pour les nœuds de crénage ;
– si c'est un nœud de pénalité, alors c'est un point de coupure légal (à condition que la pénalité ne soit pas infinie), et on lance *try_break* avec cette pénalité-là ;
– si c'est un nœud de marque, d'insertion ou d'ajustement, on ne fait rien.

Ayant parcouru ainsi toute la liste horizontale, on arrive à sa fin. On lance alors *try_break* une dernière fois [§873], avec comme pénalité *eject_penalty*. On peut donc enfin récolter les fruits de nos efforts : on prend le nœud actif avec le moins de démérite total [§874] et ce nœud actif nous fournit, en remontant les flèches, les nœuds passifs optimaux jusqu'au du paragraphe.

C'est la procédure *post_line_break* [§877] qui prend alors la relève. On commence par monter une nouvelle liste, commençant par la première coupure et allant vers le bas, jusqu'à la fin de paragraphe. *cur_p* devient donc la première coupure, et c'est cette variable qui va parcourir les nœuds de coupure optimale.

Maintenant il ne reste plus que les étapes suivantes :

1. on enlève tout ce qui indésirable de la fin de ligne [§881] et on ajoute la glue de `\rightskip` [§886] ;
2. on place la glue de `\leftskip` au début de la ligne et on détache la ligne du reste [§887] ;
3. on met le tout dans une boîte de largeur égale à la justif, en passant par une procédure appelée *just_box*, qui prend en charge l'équilibrage de tous les blancs, etc. [§889] ;
4. on ajoute cette boîte à la liste verticale courante [§888] ;
5. et enfin, on ajoute une pénalité si cela s'avère nécessaire, comme par exemple dans le cas des lignes orphelines [§890].

Le travail étant fait, on libère la mémoire en supprimant les nœuds actifs, passifs et δ.

5.2.2 try_break

Si le switch de la section précédente nous permet d'identifier les points de coupure « légaux », c'est *try_break* qui va décider s'ils sont « faisables ». L'idée est la suivante : *try_break* parcourt la liste de nœuds actifs $\alpha\_1, \ldots, \alpha\_n$ et répertorie les lignes que l'on peut composer entre chaque α_i et le nœud courant *cur_p*. Quand on peut former une ligne entre un nœud actif et *cur_p*, alors ce dernier devient actif à son tour et s'ajoute à la chaîne. Les nœuds actifs qui sont trop éloignés de *cur_p* sont désactivés.

Les démérites s'accumulent lorsqu'on crée des nouveaux nœuds actifs. Quand on arrive à la fin du paragraphe, on a donc un certain nombre de nœuds actifs qui ont survécu. Chacun est l'aboutissement d'une chaîne de nœuds passifs, et donc d'une version de paragraphe obtenue à partir de la liste horizontale courante. On prend celle qui a le moins de démérites, et l'on a gagné.

C'est la variable *r* qui va parcourir les nœuds actifs. On définit également *prev_r* et même *prev_prev_r* qui sont les nœuds précédant *r*. Il y a également une variable *old_l* dans laquelle on stocke le numéro de ligne courante. Si le tableau *active_width* indique la distance (avec différents ordres de grandeur pour la dilatation des nœuds de glue) entre le premier nœud actif et *cur_p*, le tableau *cur_active_width* va évoluer pendant le parcours des nœuds actifs.

On commence en posant *prev_r* égal au premier nœud actif, *old_l* égal à 0, et *cur_active_width* égal à *active_width*. On entre alors dans la boucle suivante [§829] :

1. on met *r* égal à *link*(*prev_r*). Si *r* est un nœud δ, on l'ajoute à *cur_active_width* pour avoir ainsi la distance entre le prochain nœud actif et *cur_p*, et on définit *prev_prev_r* comme étant *prev_r* et *prev_r* comme étant *r* [§832] ;
2. on calcule le démérite d'une ligne hypothétique entre *r* et *cur_p*. Pour cela on calcule d'abord [§851] la laideur produite par la dilatation ou la contraction de cette ligne pour la faire tenir dans la justif souhaitée. On calcule également la classe de qualité de composition de la ligne ;
3. ici Knuth utilise une astuce assez drôle qui montre ses origines mathématiciennes : que l'on dilate ou que l'on contracte une ligne, la laideur peut dans les deux cas être ∞. Mais pour nous il importe de distinguer les cas, puisque dans le premier cas la ligne est trop courte et donc le nœud *r* peut rester actif, alors que dans le deuxième elle est trop longue et le nœud *r* doit être désactivé. TEX va calculer une laideur ∞ en cas de ligne trop courte, mais une laideur $\infty+1$ (!!) en cas de ligne trop longue ;
4. dans le cas de laideur $\infty+1$, on se prépare à désactiver *r*. Pourquoi ne pas le désactiver tout de suite ? Parce qu'en cas de dépressurisation de la cabine comme on dit dans les avions, on veut avoir un masque d'oxygène sous la main. Autrement dit, si le paragraphe est impossible à composer décemment, et donc sa laideur est bien $\infty+1$, on peut vouloir garder un nœud pour composer une ligne trop longue, suivie d'une boîte de dépassement. Knuth avertit le lecteur que ce bout de code [§854] est extrêmement fragile et que la moindre modification peut entraîner une catastrophe ;
5. revenons à la normalité : si tout se passe bien, et la laideur est inférieure à la (pré)tolérance, c'est que *cur_p* est une coupure faisable. On va alors calculer le démérite *d* de *r* à *cur_p* [§859] : si b est la laideur, p la pénalité de coupure au nœud *cur_p*, p_l la valeur de `\linepenalty` (par défaut $p\_l = 10$), d_* le démérite éventuel provenant d'un `\doublehyphendemerits` (deux césures consécutives) ou d'un `\finalhyphendemerits` (césure en dernière ligne du paragraphe) ou d'un `\adjdemerits` (deux lignes consécutives de qualités de composition différentes), alors deux cas se présentent [§859] :
   (a) si $p > -\infty$ (l'infini étant 10 000 pour TEX) alors $d = (b+p\_l)^2 - p^2 + d\_*$,
   (b) si $p > 0$ alors $d = (b+p\_l)^2 + p^2 + d\_*$;
   (c) et enfin, si $p = -\infty$, alors $d = (b+p)^2 + d\_*$;
6. c'est à ce moment [§856] que TEX va enregistrer dans le log la description de la coupure faisable, du type `@ via @@5 b=97 p=0 d=11449` ;

7. au fur et à mesure que l'on trouve des nœuds actifs pour lesquels *cur_p* est une coupure faisable, on compare les démérites et on garde une trace du démérite minimal, pour chaque qualité de composition [§855].

Cette boucle se poursuit jusqu'à ce que les nœuds actifs *r* soient épuisés, où jusqu'à ce que l'on ait changé de ligne, car il est tout à fait possible quand le paragraphe est assez long que le même *cur_p* soit coupure faisable pour des numéros de ligne différents.

On s'arrête alors [§835] et on crée un nouveau nœud actif en *cur_p*, en choisissant le meilleur *r*, c'est-à-dire celui avec le moindre démérite. Si différents *r* produisent des lignes avec des qualités de compo différentes, alors on crée autant de nœuds actifs : c'est pour cela que dans notre exemple le même nœud de glue s'est trouvé associé à deux nœuds actifs (11 et 12) : les lignes produites étaient resp. de qualités 0 (« extrêmement lâche ») et 3 (« serrée »).

Créer un nœud actif implique aussi les opérations suivantes :
– calcul du tableau *break_width* [§837] : il s'agit de la glue qui s'ajoute ou qui disparaît lors d'une coupure. Ce qui peut s'ajouter, c'est le `\rightskip`, le `\leftskip` et les listes *pre_break* et *post_break* d'un `\discretionary` ; ce qui peut disparaître, ce sont des glues ou des crénages « parasites » en fin de ligne, ou alors la liste de nœuds du premier argument de `\discretionary` ;
– insertion d'un nœud δ dans la liste de nœuds actifs, entre le nœud actif qui a servi à obtenir le démérite minimal et le nœud actif nouvellement créé [§843]. Ce nœud δ contient la distance entre *r* et *cur_p* exprimée comme une glue, c'est-à-dire avec valeur au repos, quatre ordres de grandeur de dilatation maximale, et valeur de contraction maximale ;
– création des nœuds actif et passif correspondant [§845] ;
– c'est à ce moment [§846] que TEX va enregistrer dans le log la description du (ou des) nouveau(x) nœud(s) actif(s), du type `@@11 : line 6.0 t=26273 -> @@9`.

Et cela clôt la description de *try_break*, et par cela-même de *line_break* et donc du moteur de paragraphage. Il ne reste plus qu'un point obscur : la césure.

6 La césure

Nous avons vu dans la section précédente que quand *line_break* parcourait la liste horizontale pour la deuxième fois et tombait sur un nœud de glue, alors TEX se proposait d'appliquer l'algorithme de césure au mot suivant. Dans cette section nous allons explorer cette opération qui est un des grands atouts de TEX (au point où des outils actuels comme FOP se servent de ce même algorithme pour couper les textes en différentes langues).

Le but est d'examiner le « mot », c'est-à-dire une petite sous-liste de la liste horizontale, et d'y insérer des nouveaux nœuds discrétionnaires. Quand *line_break* examinera ce « mot » il ne pourra plus faire la distinction entre les nœuds discrétionnaires insérés explicitement par des `\-` ou des `\discretionary`, et des nœuds discrétionnaires obtenus par l'algorithme de césure.

Il y a nombreuses difficultés intrinsèques :
– primo, les règles de césure varient d'une langue à l'autre, elles peuvent être étymologiques, grammaticales ou un mélange des deux. Il faut donc un moyen de les décrire au niveau de TEX, et ceci de manière optimisée pour que l'algorithme soit efficace ;
– secundo, la césure est une opération linguistique qui se fait au niveau des caractères. Or, à ce stade on a à faire à une liste horizontale de nœuds : des glyphes, des ligatures, des glues, des crénages, et *tutti quanti*. On est donc très loin du concept de caractère et pourtant il nous faut retrouver les caractères qui ont servi à produire cette liste ;
– tertio, même au niveau des caractères certaines transformations sont nécessaires avant d'appliquer l'algorithme de césure. Ainsi, dans le cas du français, les mots « FILLE » et « fille » se coupent de la même manière, on a donc intérêt à convertir le premier en le deuxième ; dans le cas du grec ζήτω (adverbe « vive »), ζητῶ (verbe « je demande ») se coupent au même endroit, on a donc intérêt à faire abstraction des accents, etc.

À cela s'ajoutent les problèmes ouverts que les successeurs de TEX se proposent de résoudre : les problèmes allemands du type « backen → bak-ken », le problème néerlandais et grec du tréma qui disparaît, le problème polonais du trait de césure qui se retrouve en début de ligne quand le mot contient déjà un trait d'union, etc. Aussi le problème de césure préférentielle : pour des mots tels que « Wahrscheinlichkeitstheorie » (théorie de la relativité) on a quatre niveaux de préférence de césure :

« Wahr_2schein_4lich_4keits_1the_3o_3rie ».

Pour commencer, on a dit que l'on coupait le « mot » qui suit un nœud de glue, mais qu'est-ce qu'un « mot » ? Il s'agit d'une suite de nœuds $p\_0 p\_1 \ldots p\_m$ où p_0 est un nœud de glue, $p\_1, \ldots, p\_m - 1$ sont des nœuds de caractère, de ligature, de crénage implicite ou des nœuds quésaco, et p_m est un nœud de glue ou de pénalité ou d'insertion, ou de marque, ou d'ajustement, ou de crénage explicite.

C'est pour cette raison que, paradoxalement, dès qu'il y a un nœud discrétionnaire dans un mot, ce mot ne peut plus être coupé autrement qu'en ce nœud. De même, c'est pour cette raison que les lettres accentuées produites par des primitives `\accent` sonnent le glas du processus de césure pour le mot en question (ou du moins pour la partie du mot qui suit la lettre accentuée) : en effet, `\accent` fait appel à la procédure *make_accent* [§1123] qui crée des nœuds de crénage dont le sous-type est *acc_kern*. Or seuls les nœuds de crénage de sous-type *normal* sont

autorisés, ce sont les nœuds obtenus automatiquement par les paires de crénage contenues dans la fonte courante.

Revenons à notre suite $p\_0 p\_1 \ldots p\_m$. On va extraire de celle-ci une suite de *lettres*, c'est-à-dire de codes de nœud de caractère dont le catcode est 11 et le `lc_code` est non nul. Le `lc_code`, obtenu par la primitive `\lccode` fournit la « transformée » d'une lettre, en vue de sa césure. C'est ainsi que l'on transforme les majuscules en bas de casse, ou les lettres accentuées grecques en lettres non accentuées, avant d'appliquer l'algorithme de césure.

Soit $c\_1 \ldots c\_n$ la suite de lettres ainsi obtenues, et soient $p\_a \ldots p\_b$ les nœuds qui ont servi à obtenir ces lettres, avec $p\_a \ldots p\_b \subset p\_1 \ldots p\_m$. Comme p_m est par définition un nœud de glue, on a $1 \leq a < b < m$. Autres restrictions : $n \leq 64$ (les mots *très* longs ne peuvent pas être coupés au-delà de la 64e lettre), $n \leq l\_\text{hyf} + r\_\text{hyf}$ où ces deux quantités sont données par les primitives `\lefthyphenmin` et `\righthyphenmin`. Et, bien sûr, le trait de césure de la fonte doit être un code entre 0 et 255 (et non pas `non_char`, qui est le nombre 256), sinon pas de césure.

Malheureusement, les notations utilisées dans le code de TEX sont assez malheureuses, puisqu'elles prêtent à confusion. Les voici : `hc` est un tableau qui contient $c\_1 \ldots c\_n$, `ha` et `hb` sont des pointeurs vers les nœuds $p\_a - 1$ (c'est-à-dire le nœud qui précède immédiatement p_a, le générateur de c_1) et p_b, `hf` est la fonte courante. Et pour arriver à `hc` on passe d'abord par `hu`, c'est un tableau de même taille mais il contient les lettres avant leur transformation par `lc_code`.

Allons-y alors. On se trouve dans la boucle de `line_break`, dans la deuxième passe, et on vient juste de trouver et de traiter un nœud de glue `cur_p`. Que faire ?

On va se servir des variables `s` et `prev_s` pour avancer dans la liste horizontale à la recherche d'un « mot » à couper. On initialise donc `prev_s` comme étant `cur_p` et `s` comme étant `link(prev_s)` [§894]. On commence à parcourir les nœuds en faisant avancer `s` et `prev_s`. Dès que `s` est un nœud de caractère ou un nœud de ligature, on vérifie [§896] si le glyphe correspondant a un `lc_code` non nul[11], si c'est le cas on a trouvé notre `ha` : il s'agit de `prev_s` (le nœud avant `s`). Par contre si l'on tombe sur un nœud qui n'est pas caractère, ligature, crénage implicite, ou quésaco, alors on abandonne complètement la procédure de crénage et on retourne à la boucle de `line_break`.

Ensuite on se met à la recherche de `hb` de la même manière [§897], tout en alimentant `hc` et `hu` : on fait avancer `s` et `prev_s` et on place les glyphes des nœuds de caractère dans `hu` et leurs transformées par `lc_code` dans `hc`. Quand on tombe sur un nœud de ligature, on récupère [§898] la liste de nœuds pointée par le champ `lig_ptr` du nœud ligature. On ne les insère pas dans `hu` et `hc` avant d'avoir vérifié qu'ils sont tous des lettres avec des `lc_code` non nuls. Effectivement, rien ne sert de

11. En particulier il faut veiller à ne pas mettre de lettre à la position 0 de la fonte…

mettre une partie de ligature dans ces tableaux, et de laisser le reste dehors… Enfin, on finit nos tests en vérifiant que les nœuds entre `hb` et $p\_m - 1$ sont du bon type, ce qui normalement devrait être le cas, vu la manière dont on a choisi p_m.

Nous sommes prêts maintenant à attaquer la véritable opération de césure. Cela se fait dans la procédure `hyphenate`.

6.1 `hyphenate`

Le but de cette procédure est de trouver des points de césure dans `hc` et ensuite de remplacer la suite de nœuds entre `ha` et `hb` par une autre suite contenant des nœuds discrétionnaires.

Les points de césure sont stockés dans un tableau `hyf` de nombres entre 0 et 9. Pour comprendre la signification de ces nombres, il faut rappeler les principes des *motifs de césure*.

Les motifs de césure sont des chaînes de caractères fournies à IniTEX lors de la création d'un format par le biais de la primitive `\patterns`. IniTEX va lire ces motifs et va créer une structure de trie (cf. ci-dessous), qui sera stockée dans le format. Les motifs reflètent les règles de césure d'une langue donnée, il y aura donc autant de structures d'arbre de motifs que de langues définies.

Voici la syntaxe des motifs de césure : pour indiquer la possibilité de césure à droite ou à gauche d'une lettre `a` on écrira un chiffre impair entre 1 et 9 à droite ou à gauche de cette lettre. Pour indiquer une interdiction de césure, on utilisera un chiffre pair entre 0 et 8. L'absence de chiffre est équivalente au chiffre 0. Ainsi, en écrivant `mé1ta` la chaîne de caractères « méta » pourra toujours être coupée « mé-ta », où qu'elle se trouve dans un mot — autrement dit, on n'aura jamais de coupure « m-éta » ou « mét-a ».

Si le chiffre « 1 » suffit pour indiquer les césures potentielles, à quoi bon d'aller jusqu'à 9 ? Parce que, comme toujours dans la vie, les règles ont des exceptions, qui à leur tour ont des exceptions, et ainsi de suite. Ainsi, le motif `1t` indique que l'on préfère couper plutôt avant la consonne « t » qu'après (on coupera « appé-tit » et non pas « appét-it »). Mais si la consonne est suivie d'une autre consonne, alors on a plutôt envie de couper après : « met-tre ». On exprime cette exception par le motif `2t1t`. Le « 2 » est plus fort que le « 1 », et il prévaut.

En fait, TEX va prendre tous les motifs qui couvrent une partie du mot, et trouver pour chaque paire de lettres le chiffre maximal rencontré. Si le chiffre est impair, et seulement dans ce cas, il va insérer un point de césure.

On peut également utiliser le point `.` dans un motif pour indiquer le début ou la fin de mot. Le lecteur comprendra aisément le motif `.con4` que l'on trouve dans

les motifs français de Flipo et Gaulle. Ainsi, en français et uniquement dans cette langue, on peut couper « mé-con-nus » mais *pas* « con-nus »...

Revenons donc au code de TEX. Le tableau `hyf` contient justement les chiffres des motifs. L'essentiel de la procédure `hyphenate` va consister à remplir `hyf`. Mais avant de le faire, il faudrait que l'on sache comment les informations récupérées par les motifs de césure sont stockées dans la mémoire de TEX. Autrement dit : que fait IniTEX lorsqu'il lit des motifs dans l'argument de la primitive `\patterns`?

6.2 Création du trie de césure

Ne cherchez pas le mot « trie » dans le *Petit Robert*, il n'y est pas. En fait c'est un mot anglais artificiel qui a été repris tel quel dans des ouvrages français comme [4]. Le mot *trie*, à ne pas confondre avec « *tree* » (= « arbre ») vient du mot *retrieval* et se prononce en anglais « trii » ou alors « traï », selon l'auteur. Il a été inventé en 1960 par un certain Fredkin [2].

Un trie est un arbre de degré ≥ 2 dans lequel le branchement à tout niveau n'est pas déterminé par la valeur de la clé dans son entier mais par une partie de cette valeur. Ainsi, on va lier un nœud d'étiquette « la », à un nœud d'étiquette « las », lui-même lié à un nœud « lasse », puis « lasser », « lassera » et ainsi de suite. Mais on ne va pas lier « la » et « but », ni même « la » et « le ».

Le trie utilisé dans TEX est spécial puisqu'il s'agit d'un trie compressé et empaqueté d'après une méthode décrite dans la thèse de Liang [8]. C'est IniTEX qui va d'abord établir le trie de base pour ensuite le compresser et l'empaqueter, de manière à ce qu'il ne soit plus qu'une chaîne de caractères, contenue dans le fichier de format.

Voyons d'abord comment fonctionne le trie de base. Chaque nœud du trie est consiste en un caractère `trie_c`, en deux liens vers d'autres nœuds `trie_l` et `trie_r` (lien « gauche » et « droite ») et en une « opération » `trie_o`. Le parcours de ce trie se fait de la manière suivante : on compare le premier caractère du motif avec le premier nœud du trie, s'ils concordent on suit le nœud gauche, sinon on suite le nœud droite. On compare de nouveau et on continue. Les nœuds obtenus en allant « vers la droite », forment une « famille ». Ils sont classés dans l'ordre alphabétique des caractères qu'ils représentent. Ainsi on sait que si l'on a dépassé le caractère recherché, alors on peut aussi bien abandonner le parcours.

Quand on trouve les bons caractères, on avance « vers la gauche ». Un nœud dont le lien gauche est nul, est un nœud final. Quand on a obtenu la chaîne recherchée, ou quand on est tombé sur un nœud final, on arrête le parcours et on s'intéresse à `trie_o` (l'« opération ») du dernier nœud obtenu. Il s'agit d'un nombre qui, après une suite d'opérations, nous fournira les valeurs de césure de chaque caractère du motif obtenu.

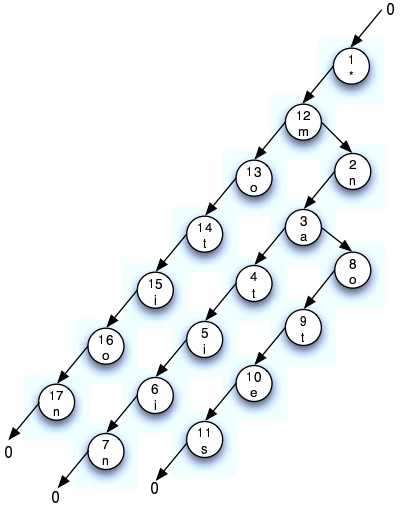


FIGURE 3 – Trie d'origine

Prenons un exemple. Soient les motifs `na1tion`, `no1tes` et `mo1tion`. Nous avons représenté le trie obtenu dans la figure 3.

Le premier nœud porte le numéro 1 et la valeur numérique du caractère associé est celle de la langue courante (un nombre à 1 octet, donné par la primitive `\language`). Il s'agit d'une astuce qui permet d'inclure dans un même trie [12] des motifs de langues différentes, sans ambiguïté, ni conflit. Pour indiquer qu'un motif appartient à une longue donnée, on préfixe le motif par le numéro de langue : *hc*`[0]=`*cur_lang*.

Le nœud qui suit est le 12, parce que le motif `mo1tion` a été lu après les autres. Son lien gauche nous fournit « motion » et son lien droit nous fournit un « n », qui à son tour nous fournit « nation » et « notes ». Par les calculs on obtient des valeurs *trie_o* nulles pour tout le monde, sauf pour les trois nœuds finaux : le *trie_o* de 17 et de 7 est 1, celui de 11 est 2. Que signifient ces valeurs ? À l'aide de deux tables *hyf_distance* et *hyf_num* on trouve que la valeur 1 signifie « une valeur de césure 1 après le 2e caractère d'un motif à 6 caractères », et 2 signifie : « une valeur de césure 1 après le 2e d'un motif à 5 caractères ».

12. Notons que même si TEX lit des motifs de plusieurs langues via une série de primitives `\pattern` entre lesquelles on a la possibilité de changer la langue courante par `\language`, il ne construit en fait qu'un seul trie. C'est pour cela que l'on ne peut stocker sur disque un trie pour chaque langue et le charger sur demande. On est alors obligé de créer une multitude de formats avec diverses combinaisons de motifs de césure…

Avant de passer à la compression et à l'empaquetage, voyons comment le code est structuré.

La création du trie de base se fait uniquement dans IniTEX, le code correspondant se trouve dans la section 43 qui commence par le §942. Il est entièrement placé entre **init** et **tini**, ce qui signifie qu'il ne peut être exécuté que lorsque l'option de ligne de commande `-ini` a été utilisée. Lorsque la primitive `\patterns` est lue [§1252], IniTEX va lancer la procédure *new_patterns*. Celle-ci est décrite dans le §960. Par le biais d'une boucle on lit les motifs, en séparant le point (début et fin de mot), les lettres et les chiffres (l'absence de chiffre équivaut au chiffre 0), jusqu'à arriver à une accolade droite. L'expansion de macros dans l'argument de `\patterns` est possible si elles ne produisent que des tokens de caractère.

Les lettres sont placées dans un tableau *hc* (d'indices ≥ 1, en mettant à l'indice 0 la valeur de la langue courante) et les valeurs de césure dans un tableau *hyf* [§962]. Après lecture d'un blanc ou d'une accolade droite, on arrive au §963 qui va insérer le motif dans le trie de la manière suivante :

1. on prend les lettres du motif une à une, et on vérifie si elles correspondent aux *trie_c* de nœuds existants dans le trie ;
2. tant que les lettres du motif se trouvent déjà dans l'arbre, on continue le parcours. Dans le cas contraire, on crée un nouveau nœud [§964], en prenant soin de respecter l'ordre alphabétique des nœuds d'une famille (c'est-à-dire des nœuds obtenus en suivant des liens de droite) ;
3. arrivés à la fin du motif on teste si le *trie_o* du nœud courant est nul. S'il ne l'est pas, cela ne peut être que parce que ce motif a déjà été lu et enregistré dans le trie, on affiche alors un message d'erreur ; s'il est nul, on calcule le *trie_o* du nœud final [§965] qui représente le motif tout entier, en utilisant la procédure *new_trie_op* [§944].

Jusqu'ici, aucune difficulté particulière, mais on constate que beaucoup de mémoire est gaspillée : on nécessite quatre tableaux pour décrire les nœuds (*trie_l*, *trie_r*, *trie_o*, *trie_c*), deux tableaux pour obtenir les valeurs de césure (*hyf_dimension* et *hyf_num*), et dans ces tableaux on rencontre beaucoup de redondance. Ainsi, dans notre exemple de trie, on constate que les nœuds 14–17 et 4–7 fournissent la même information : toutes leurs valeurs sont égales, y compris les *trie_o* des nœuds 17 et 7 qui auraient pu être différents puisqu'ils dépendent des motifs entiers. En effet, les mots « motion » et « nation » ont la même longueur et se coupent de la même manière (« un chiffre 1 après la 2e lettre du motif »), il est donc normal que leurs valeurs *trie_o* soient égales.

La compression du trie consiste à trouver des branches équivalentes dans le sens du paragraphe précédent, et de les fusionner. C'est le cas, dans notre exemple, des suites de nœuds 14–17 et 4–7, qui correspondent à des sous-motifs `**1tion` (les astérisques sont là pour indiquer que dans les deux motifs il y a le même nombre

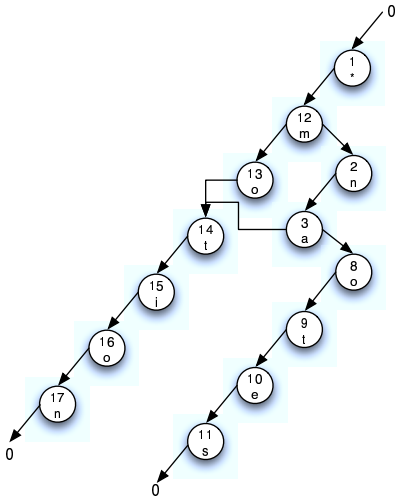


FIGURE 4 – Trie compressé

de caractères avant le début du sous-motif). Le lecteur trouvera le trie compressé représenté dans la fig. 4.

Cela nous fait 4 nœuds de moins ! Voici comment se présente le code :

On définit une procédure `compress_trie` [§949] qui va compresser le trie de manière récursive. En commençant par les feuilles, on applique à chaque nœud la fonction `trie_node` [§948]. Celle-ci va examiner si un nœud avec les mêmes `trie_l`, `trie_r`, `trie_o`, et `trie_c` existe, et si c'est le cas, il remplace le nœud courant par le nœud trouvé. Les feuilles de l'arbre ont des valeurs `trie_l` et `trie_r` nulles, il n'y a donc que le caractère `trie_c` et la valeur de `trie_o` qui comptent. On peut donc identifier les feuilles qui représentent le même caractère et qui terminent des motifs de même longueur et qui se coupent de la même manière.

Une fois les feuilles fusionnées, on examine leurs nœuds parents (par un lien de gauche ou de droite). S'ils ont les mêmes propriétés, ils sont fusionnés également et c'est ainsi que l'on arrive à compresser assez efficacement le trie tout entier.

Passons maintenant à l'opération d'empaquetage. Le but est d'obtenir une représentation en mémoire du tri beaucoup plus compacte que les quatre tableaux `trie_l`, ..., `trie_o`. La première idée géniale est de dire que les nœuds d'une famille (c'est-à-dire ceux obtenus à partir d'un nœud en suivant des liens de droite) se distinguent par le caractère qu'ils représentent. Ces caractères ont des valeurs numériques : ce sont leurs positions dans un codage donné (« a » se trouve à la position 97 des

codages ASCII et Unicode, « n » à la position 110, etc.). Si l'on se met d'accord que la position d'un tel nœud dans une table est obtenue directement à partir de la valeur numérique du caractère, on fait l'économie du tableau *trie_r*.

Soit un nœud p et $q\_1, \ldots, q\_n$ une famille de nœuds (q_i est le lien de droite de $q\_i - 1$) qui se trouvent sous p, c'est-à-dire tels que q_1 est le lien de gauche de p. Alors on définit le *décalage* *trie_link* de p comme étant la valeur qu'il faut ajouter au code de caractère de q_i pour obtenir sa position dans le tableau. Voici ce que ça donne :

	0	1	2	3	4	5	6	7	8	9
10x : *trie_char*		a					i			
trie_link		1					2			
11x : *trie_char*		n		o	n			t		
trie_link		4		4	0			1		

Le nœud 1 du trie a un *trie_link* égal à la valeur de la langue courante plus 1, la première lettre du motif est donc à rechercher à la position du tableau qui est égale à sa valeur numérique plus 1. Prenons le motif `na1tion`. La première lettre est « n » dont la valeur est 110. On va donc à la position 111 du tableau. On y trouve un *trie_link* de 4, autrement dit : la deuxième lettre doit être recherchée avec un décalage de 4. Celle-ci est un « a » = 97, on va donc à la position 101 du tableau. Le *trie_link* à cet endroit est 1, on va donc à la position (« t » = 116) + 1 = 117, où le *trie_link* est 1, et on arrive ainsi jusqu'à la position 114 pour le dernier « n ».

Dans le tableau ci-dessus nous avons indiqué les lettres (ligne *trie_char*) et les *trie_link* correspondants. Pour vérifier qu'un motif se trouve dans le tableau, il suffit de le parcourir comme on l'a fait, en vérifiant à chaque étape que le caractère obtenu est le bon. Arrivé à la dernière lettre, il existe une correspondance *trie_op* avec les nœuds de valeurs de césure, comme dans le trie de base (*trie_op* provient directement de *trie_o*). Il suffit alors de désassembler la valeur de *trie_op* en se servant de *hyf_distance* et *hyf_num*, pour obtenir les valeurs de césure du motif, comme on l'a fait pour *trie_o*.

Il reste une question : supposons que dans notre texte à composer il y ait le mot « nations » (avec un « s »). On essaie donc de retrouver ce mot parmi les motifs. Arrivé au deuxième « n », on constate qu'il a un *trie_link* de 0. On pourrait alors prendre la valeur numérique de la lettre « s », qui est 115, l'ajouter à 0, et aller voir ce qui se trouve dans la case 115. Si par malheur à cet endroit il y a déjà une lettre « s », provenant d'un autre motif, cela peut nous induire en erreur : on va alors continuer le parcours, et le *trie_op* que l'on trouvera à la fin sera tout autre.

Autre manière de formuler la question : comment savoir que le contenu d'une case est un nœud final ? Aurions-nous gagné les 327,68 $ promis par Knuth pour toute

découverte de bogue dans TEX ? Y penser ne serait-ce qu'un instant serait gravement sous-estimer le génie du Grand Maître. Il y a une manière très simple de savoir si un nœud est final. Dans ce cas, son *trie_link* est nul [13].

Cela résout le problème. En effet, supposons qu'un autre motif ait mis une lettre « s » (= 115) à la case 115. La seule manière d'obtenir cette lettre dans cette case est d'utiliser un décalage nul, ce qui est impossible puisque seuls les liens finaux ont un *trie_link* nul. En ce qui concerne la première lettre du motif, on a pris soin d'ajouter systématiquement à sa valeur numérique la valeur de la langue courante (qui est ≥ 0) plus 1, on ne risque donc pas de retrouver une lettre dans la case de sa valeur numérique.

L'exercice ne s'arrête pas là : pour optimiser vraiment la taille de notre trie, il s'agit de mettre un maximum de motifs dans une table aussi petite que possible. La méthode de Knuth, décrite dans la thèse de Liang [8], donne de très bons résultats. Voici une table contenant nos trois motifs : `na1tion` (gras), `mo1tion` (italiques), `no1tes` (souligné) :

	0	1	2	3	4	5	6	7	8	9
10x : *trie_char*		**a**	e				***i***			
trie_link		1	1				2			
11x : *trie_char*	*m*	**n**	*o*	***o***	***n***	o	s	***t***	t	
trie_link	1	4	1	4	0	2	0	1	1	

On constate qu'il y a des nœuds utilisés par plus d'un motif : 111 « n » est le point de départ de `na1tion` et de `no1tes`, les nœuds 117 « t », 106 « i », 113 « o », 114 « n » sont partagés par `na1tion` et `mo1tion`.

Voyons maintenant comment se présente le code de l'empaquetage. Cette opération est effectuée au moment de la création d'un format, autrement dit : lorsque IniTEX rencontre la primitive `\dump`. Cela est décrit dans le §1324. On lance alors la procédure *init_trie*, et ensuite on affiche un message à l'utilisateur :

```
Hyphenation trie of length 15612 has 474 ops out of 35111
  207 for language 2
  86 for language 1
  181 for language 0
```

13. Dans la version 2 de TEX, la situation était toute autre : le *trie_link* des nœuds finaux prenait alors comme valeur le max+1 de tous les *trie_link* non finaux du trie. En revanche, on pouvait très bien avoir des *trie_link* nuls, sans que cela implique quoi que soit au niveau de la finalité du nœud.

Ce message nous apparaît moins cryptique maintenant : le trie que l'on vient de créer et d'écrire dans le fichier de format, possède 15 612 cases ; il nous a fallu 474 valeurs de *trie_op*, c'est-à-dire 474 combinaisons de valeurs de césure ; on avait donné un maximum de 35 111 telles valeurs. Dans l'exemple de motifs `na1tion mo1tion no1tes`, il nous aurait fallu 2 valeurs de *trie_op*, puisque `na1tion` et `mo1tion` utilisent la même valeur.

Que fait alors *init_trie* ? Cette procédure [§966] lance trois autres procédures :

1. *first_fit* [§963] va trouver la plus petite case vide z telle que $z + c\_i$ pour tous les c_i de la famille du nœud z soient également vides ;
2. *trie_pack* [§957] va poursuivre cette opération de manière récursive, en parcourant le trie ;
3. le code de §958 et la procédure *trie_fix* [§959] qui vont « nettoyer » le trie. En particulier c'est ici que l'on annule le *trie_link* des nœuds finaux.

6.3 Les exceptions de césure

Les exceptions de césure sont indiquées par l'utilisateur à l'aide de la primitive `\hyphenation`. Il s'agit d'une liste de mots dans lesquels les césures potentielles sont notées par des traits d'union : `conscien-cieux mé-con-tent`. Ces mots sont stockées dans une table de hachage à l'aide de la procédure *new_hyph_exceptions* [§934]. Si le mot contient n lettres, on en ajoute une $(n+1)$-ème qui représente la langue courante, ainsi les exceptions de césure sont classées par langue. Aussitôt un mot lu, il est placé dans le tableau *hc* et on calcule sa clé de hachage *h* [§939], par récurrence : si à l'étape j on a la clé h_j, alors à l'étape $j+1$ on a $h\_j+1 = 2h\_j + hc[j] \bmod 607$ (où 607 est un nombre premier). Ensuite on ajoute le mot à la table de hachage [§940] : les lettres vont être stockées dans *hyph_word*`[`*h*`]` et les césures potentielles dans *hyph_list*`[`*h*`]`.

La consultation de la table de hachage est tout aussi rapide [§930] : on ajoute la lettre artificielle qui indique la langue, on calcule *h*, et on parcourt les entrées trouvées à l'adresse *h*. S'il y en a une qui concorde avec le mot courant, alors on place des nombres 1 dans le tableau *hyf* des valeurs de césure de la procédure *hyphenate*. Notons que si une exception de césure concordante est trouvée, on ne va plus examiner le trie. Les valeurs de césure 1 sont alors amplement suffisantes pour permettre la césure du mot.

6.4 Suite de *hyphenate*

Après avoir expliqué la structure du trie de césure, revenons à *hyphenate* [§895]. Nous nous trouvons maintenant dans la procédure qui va insérer des nœuds discrétionnaires dans les mots qui suivent un nœud de glue. On a déjà placé les lettres

dans le tableau `hc`, l'étape suivante est de remplir le tableau `hyf` des valeurs de césure correspondantes.

La première étape de `hyphenate` se trouve dans le §923 : on teste d'abord si une exception de césure concorde avec le mot, si c'est le cas, on remplit `hyf` à partir du tableau `hyph_list` trouvé dans la table de hachage d'exceptions de césure.

Si ce n'est pas le cas, on vérifie que l'on a au moins un motif pour la langue donnée et on commence le parcours du trie avec un premier nœud `z` égal à `trie_link(cur_lang+1)+hc[0]` et ensuite en posant `z←trie_link(z)+hc[l]` où `hc[l]` parcourt le mot. Cette opération est ré-itérée : on commence par la première lettre et on avance en examinant de plus en plus de lettres, ensuite on recommence par la deuxième lettre, et ainsi de suite. Pour un mot `abcde` on va donc tester la présence dans le trie des chaînes `a`, `ab`, `abc`, `abcd`, `abcde` et ensuite `b`, `bc`, `bcd`, `bcde`, et `c`, `cd`, `cde`, et `d`, `de` et enfin `e` [14].

Chaque essai nous fournit des valeurs de césure dans `hyf` et on ne garde que les plus grandes valeurs pour chaque lettre. Après le dernier essai nous avons obtenu notre tableau `hyf` définitif. Si toutes les valeurs de `hyf` sont paires, on considère que le mot ne peut être coupé [§902] et on quitte `hyphenate`.

Si l'on a trouvé au moins une valeur impaire, alors on entre dans la phase de « post-césure » [§903], qui est la deuxième étape de `hyphenate`. Rappelons que si $p\_a \ldots p\_b$ sont les nœuds qui ont servi à obtenir `hu` et `hc`, alors on a appelé `ha` et `hb` des pointeurs vers les nœuds $p\_a - 1$ (c'est-à-dire le nœud qui précède immédiatement p_a) et p_b, et `hf` la fonte courante.

Quelques initialisations : on met `q←link(hb)` (c'est-à-dire p_b) et `r←link(ha)` (c'est-à-dire $p\_a - 1$) et `s←cur_p` (qui, rappelons-nous, est un nœud de glue qui précède le mot à couper). On fait avancer `s` jusqu'à `ha`.

Et c'est ici [§904] que Knuth nous prépare psychologiquement pour ce qui va suivre en disant : « il faut maintenant admettre le fait que la bataille n'est pas finie ». En effet, en repérant les césures potentielles on est passé des glyphes aux caractères, maintenant il faut revenir aux glyphes, et cela n'est pas toujours trivial. Ainsi, lorsqu'on coupe au milieu d'une ligature, d'autres ligatures peuvent se former. L'exemple typique est celui du mot « raffiné » qui contient une ligature « ffi » : lorsque ce mot est coupé « raf-finé », une nouvelle ligature « fi » apparaît. Et ce n'est qu'un exemple simpliste, puisque l'on peut très bien définir des ligatures ou des crénages avec le trait de césure, et ces ligatures peuvent en amener d'autres, ce qui peut déclencher une réaction en chaîne…

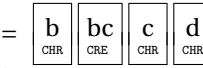
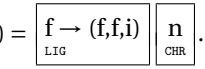
14. Le lecteur se demande peut-être à quoi peut bien servir un motif d'une seule lettre. Eh bien, cela peut être très utile : supposons que l'on veuille implémenter la règle « on peut toujours couper après une voyelle ». Pour cela il suffit d'écrire `a1 e1 i1 o1 u1`, ce sont des motifs d'une lettre.

La procédure `reconstitute` va effectuer le passage des caractères au glyphes.

Mais d'abord une définition. Soient $x\_1 \dots x\_n$ des caractères pris dans le tableau `hu` (l'ancêtre de `hc`, d'avant la transformation par `\lccode`). Supposons que ces caractères, pris ensemble, produisent des nœuds $y\_1 \dots y\_N$. Appelons $\mathscr{C}$ le passage des caractères aux nœuds, alors $\mathscr{C}(x\_1 \dots x\_n) = X\_1 \dots X\_N$. Partageons maintenant la chaîne en deux morceaux : $x\_1 \dots x\_\bullet$ et $x\_\bullet + 1 \dots x\_n$. Si l'on prend les images de ces deux sous-chaînes par $\mathscr{C}$ et que l'on les concatène, il n'est pas du tout sûr que l'on obtiendra de nouveau X_1…X_N (en guise d'exemple, il suffit de considérer qu'il y a une ligature entre $x\_\bullet$ et $x\_\bullet + 1$).

Supposons qu'il existe un $m > 1$ tel que au moins la deuxième sous-chaîne $x\_m + 1 \dots x\_n$ soit indépendante du contexte, dans le sens où $\mathscr{C}(x\_1 \dots x\_n) = y\_1 \dots y\_t + \mathscr{C}(x\_m + 1 \dots x\_n)$. Et s'il existe plusieurs m ayant cette propriété, prenons le plus petit. Alors $x\_1 \dots x\_m$ est appelé un *cut prefix*.

En voici quelques exemples (toujours dans la fonte *cmr10*) :

1. $\mathscr{C}(\texttt{abc}) =$  . Alors $m = 1$ et $y\_1 =$;
2. $\mathscr{C}(\texttt{bcd}) =$. Alors $m = 1$ et $y\_1 y2 =$. Pourquoi le nœud de crénage appartient-il au *cut prefix* ? Parce que la chaîne $x\_m + 1 \dots x\_n$ est ici simplement la lettre `d`, et $\mathscr{C}(\texttt{d}) =$  . Le nœud de crénage appartient donc forcément à $y\_1 \dots y\_t$;
3. $\mathscr{C}(\texttt{ffin}) =$  . Alors $m = 3$ et $y\_1 =$.

La procédure `reconstitute` examine la chaîne de caractères $x\_1 \dots x\_n$, calcule le nombre m et place les nœuds $y\_1 \dots y\_t$ dans une liste temporaire qui débute en `link(hold_head)` (`hold_head` [§162] est une sorte de point générique de rattachement de liste temporaire).

Le code de `reconstitute` est une version simplifiée de `try_break`, simplifiée parce que l'on n'a pas de lecture de tokens à faire ; en effet, les caractères qui vont générer nos nœuds se trouvent déjà bien rangés dans le tableau `hu`.

En §903 on commence par le cas spécial du premier caractère : il faut tester la présence d'une ligature ou d'un crénage de début de mot. On construit une liste horizontale `init_list` à partir des ces ligatures et/ou crénages, et on active une booléenne `init_lig` si l'on a effectivement une ligature à cet emplacement.

À la fin de la section §903, on passe en §913, où l'on lance une boucle sur les j. Pour chaque j on exécute `reconstitute` sur $x\_j \dots x\_m$, qui nous fournit donc un m. Pour passer à l'étape suivante, on fait $j \leftarrow m+1$. `reconstitute` laisse plusieurs traces de son passage, dont aussi la valeur de `hyphen_passed`. Il s'agit de l'indice du

premier caractère de la ligature qui possède une valeur de césure impaire. Ainsi, si $x\_1 \dots x\_n$ est `ffin` et si l'on peut couper « ffin » en « f-fin », on aura *hyphen_passed* égal à j.

Si *hyphen_passed* est nul, on fait avancer la variable s (qui pointe vers la fin de la liste que nous sommes en train de créer) jusqu'au dernier nœud créé par $x\_1 \dots x\_m$. Il reste un dernier cas de césure, non prévu par *reconstitute* : le cas où le dernier caractère du *cut prefix*, c'est-à-dire x_m, a une valeur de césure non nulle. On peut donc encore changer *hyphen_passed* et lui donner la valeur m.

Si *hyphen_passed* n'est pas nul, soit parce qu'il y avait une césure potentielle dans la ligature, soit parce qu'il en avait une à la fin de la ligature, alors on est dans le vif du sujet [§914]. Rappelons que *reconstitute* nous laisse une liste de nœuds $y\_1 \dots y\_t$ commençant par *link*(*hold_head*). On va entrer dans une boucle qui nous permettra de traiter les deux cas [15] de césure potentielle.

En ce faisant, nous allons créer deux listes : une liste « majeure » terminée par *major_tail* et une liste « mineure » terminée par *minor_tail*. La liste mineure va contenir les nœuds produits en cas de césure et la liste majeure ceux produits en cas de non-césure.

Ainsi, dans le §915 on s'intéresse aux nœuds de $y\_1 \dots y\_t$ qui viennent avant la césure. Soit $i \leftarrow$*hyphen_passed*, la chaîne de caractères qui précède la césure est donc $x\_1 \dots x\_i$. À ce stade, il faut envisager un cas rare mais tout à fait possible : si l'on ajoute un trait de césure à cet endroit, il se peut que x_i et le trait de césure aient une interaction (ligature ou paire de crénage). Cela va influencer le *pre_break* de notre nœud discrétionnaire.

On va donc tricher : on pose temporairement $x\_i+1 =$*hyf_char*, où *hyf_char* est le code du trait de césure, et on lance de nouveau *reconstitute* sur toutes les sous-chaînes de $x\_1 \dots x\_i x\_i+1$. Celle-ci nous fournit des listes d'éventuelles ligatures ou paires de crénages, commençant par *link*(*hold_head*). On place donc les nœuds obtenus dans la liste mineure, ainsi que dans le *pre_break* de notre nœud discrétionnaire. Ensuite on rétablit la valeur originale de x_i+1.

Passons maintenant au *post_break* [§915]. On commence à x_i+1 et on monte jusqu'à x_j (qui n'est autre que x_m) en attachant des nœuds à la liste mineure (re-initialisée depuis que l'on a fini le *pre_break*), ainsi qu'au *post_break*. Pour que des éventuelles ligatures ou paires de crénage n'échappent pas, on avance en lançant de nouveau *reconstitute* sur toutes les sous-chaînes de $x\_i+1 \dots x\_j$.

Prenons un peu de recul pour voir où l'on est. Dans le §914 on avait créé un nœud discrétionnaire r, que nous avons immédiatement ajouté à la liste majeure. Les

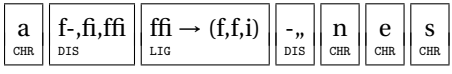
15. On peut se demander pourquoi il a fallu une boucle dans §914, alors qu'il n'y a que deux cas possibles. Peut-être que Knuth avait voulu traiter le cas de toutes les césures potentielles de la ligature, mais a trouvé que cela était trop difficile, mais que la boucle y soit restée…

pre_break et *post_break* que nous venons de créer, font partie de ce nœud. Ensuite nous avons continué d'attacher des nœuds à la liste majeure, en comptant les nœuds obtenus par les caractères qui ont généré la ligature. Le nombre de ces nœuds fait également partie du nœud discrétionnaire, c'est le nombre de nœuds à supprimer en cas de césure. Nous avons donc créé le nœud discrétionnaire avant même de commencer à explorer les ligatures de $x\_1 \ldots x\_j$, et nous avons alimenté, en même temps, les branches de pré-cesure et de post-césure de ce nœud, ainsi que la liste majeure. Arrivés à x_n, tous les nœuds discrétionnaires ont été ajoutés et il ne reste plus qu'à remplacer les nœuds *p_a . . . p_b* par la liste majeure obtenue. On revient à §913, où l'on trouve la dernière instruction *link(s)*$\leftarrow q$. En effet, la variable *s* a parcouru la liste majeure, et arrivés à la fin de l'opération, en faisant *link(s)*$\leftarrow q$ on rattache la liste majeure à la liste horizontale du paragraphe. Le tour est joué et ceci est la fin de *hyphenate*.

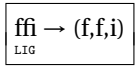
Prenons un exemple : en supposant que l'on a des motifs de césure `f1` et `i1`, le mot « affines » devient, lors de la deuxième passe du paragraphage :

```
...\tenrm a
...\discretionary replacing 1
....\tenrm f
....\tenrm -
...|\tenrm ^^L (ligature fi)
...\tenrm ^^N (ligature ffi)
...\discretionary
....\tenrm -
...\tenrm n
...\tenrm e
...\tenrm s
```

Ce code fait partie de la sortie de `\tracingparagraphs`, avec nos notations il devient :



On voit donc qu'après 

on a un nœud discrétionnaire qui remplace, les cas échéant, *un* nœud (le nœud qui le suit, c'est-à-dire : `^^N` = 

). Le *pre_break* de ce nœud est :

```
....\tenrm f
....\tenrm -
```

c'est-à-dire les nœuds $\boxed{\underset{\text{CHR}}{\text{f}}}$ et $\boxed{\underset{\text{CHR}}{\text{-}}}$, et le *post_break* est :

```
...|\tenrm ^^L (ligature fi)
```

c'est-à-dire $\boxed{\underset{\text{LIG}}{\text{fi} \rightarrow \text{(f,i)}}}$. Après $\boxed{\underset{\text{LIG}}{\text{ffi} \rightarrow \text{(f,f,i)}}}$ il y a encore un nœud discrétionnaire, mais celui ne remplace aucun nœud et a un *pre_break* égal à $\boxed{\underset{\text{CHR}}{\text{-}}}$.

Notons que pour TEX, une ligature ne peut malheureusement avoir que deux points de coupure : un à l'intérieur, et un à sa fin. Cela revient à placer au plus un nœud discrétionnaire « riche » (c'est-à-dire avec *pre_break* et *post_break*) *avant* le nœud de ligature, et au plus un autre, « simple », *après* la ligature. Un exemple : supposons que d'après nos motifs de césure, « ffi » peut être coupé aussi bien en « f-fi » qu'en « ff-i ». Seul la première coupure est prise en compte par TEX — pour obtenir la deuxième, il faudrait utiliser une fonte sans ligature « ffi ». Cela est clairement une faiblesse de TEX, et on peut espérer que ses successeurs pourront la pallier [16]. Quoi qu'il en soit, Knuth décide de ne pas traiter les ligatures à césures multiples en affirmant que « TEX évite cela en ignorant purement et simplement les césures additionnelles dans de tels cas bizarres » [§904].

Nous sommes arrivés à la fin de *hyphenate*. Il ne reste plus qu'à décrire *reconstitute*, que nous avions laissée en suspens.

6.5 *reconstitute*

Cette fonction Pascal, dont le code commence au §906, prend comme arguments les indices j et n d'une chaîne de caractères $x\_j \ldots x\_n$ et en calcule le plus petit *cut prefix*. Elle retourne m, c'est-à-dire l'indice du caractère x_m tel que $x\_m+1 \ldots x\_n$ est indépendante de $x\_j \ldots x\_m$. L'image de $x\_j \ldots x\_m$ par $\mathscr{C}$ est $y\_1 \ldots y\_t$. Elle est stockée par *reconstitute* dans une liste qui débute en *link*(*hold_head*).

Pour ce faire, on reprend les techniques de *try_break*. Ainsi, ici aussi on utilise un « curseur » (variables *cur_r* et *cur_l*) qui parcourt la chaîne de caractères, une

16. Si Knuth n'a pas prévu des coupures multiples de ligature, ce n'est pas par négligence. Il s'agit d'un véritable défi puisqu'on arrive ainsi à un *graphe de césures potentielles*. Ainsi, imaginons que les lettres abcde forment une ligature qui peut être coupée partout. On peut donc avoir des sous-ligatures ab-cde, abc-de, etc. mais aussi ab-(c-de), (ab-c)-de, et ainsi de suite. Premier problème : comment représenter toutes ces possibilités sous formes de nœuds discrétionnaires ? Deuxième problème : comment amener TEX à choisir parmi toutes ces possibilités au moment du paragraphage ? Ce n'est pas impossible, mais cela demanderait un bon effort de développement, probablement disproportionné avec l'utilité du résultat souhaité. Rares sont les ligatures de trois lettres ou plus.

pile de ligature *lig_stack* qui va contenir les caractères qui ont servi à créer une ligature, et un pointeur *cur_q* vers l'endroit où l'on place l'expansion de la ligature.

C'est la variable *j* qui va parcourir la chaîne de caractères et *t* qui va pointer vers les nœuds que nous allons créer. *t* est initialisé à *hold_head*. Le cas $j = 0$ est un peu particulier car il se peut que l'on ait une ligature ou un crénage de début de mot.

On commence en mettant *hu*[*j*] dans *cur_l*, en posant *cur_q* $\leftarrow$ *t*, et en créant un nouveau nœud de caractère, attaché à *t*, dont le glyphe est *cur_l* [§908]. On initialise *lig_stack*, et enfin on récupère *cur_r*, le caractère de droite du curseur, qui n'est autre que *hu*[*j*+1]. Ici on constate une différence avec *try_break* : il faut tenir compte du fait que certains caractères ont des valeurs de césure impaires et sont donc des points de césure potentiels. On définit alors *cur_rh* qui va être *hchar* (c'est-à-dire un trait de césure) quand *cur_l* est un point de césure potentielle, et *non_char* (c'est-à-dire la valeur absurde 256) sinon.

Puis [§909] on récupère les infos de glyphe de *cur_l* dans *q* et on vérifie si le glyphe en question possède des ligatures. Si ce n'est pas le cas, on termine *reconstitute* en retournant *j*. Par contre, si c'est le cas, on récupère dans *q* la première instruction du PLC de ce glyphe.

On définit une variable *test_char*, qui est *cur_rh* si *cur_l* est un point de césure potentielle, et *cur_r* sinon. On va d'abord tester une interaction possible entre *cur_l* et le trait de césure, et ensuite, une interaction entre *cur_l* et *cur_r*.

Nous entrons donc dans une boucle qui nous permet de parcourir toutes les instructions du PLC de *cur_l*, jusqu'à trouver une instruction dont le *next_char* soit *test_char*. Si *test_char* est *cur_rh*, on fait également *hyphen_passed* $\leftarrow$ *j*. Si l'on a trouvé un crénage, alors on pose *w* $\leftarrow$ *char_kern*(*hf*)(*q*) et on sort de la boucle. Si l'on a trouvé une ligature, on passe au §911 qui est un switch similaire à celui du §1040 de *try_break*. On y considère tous les cas de ligatures intelligentes, et on place, le cas échéant, certains nœuds sur *lig_stack*, pour les traiter lors de la prochaine itération.

Quand le *lig_stack* est non nul, on récupère le glyphe de ligature comme *cur_l* à la prochaine itération, et on teste son interaction potentielle avec le glyphe suivant. On vide la pile *lig_stack* et on active la booléenne *ligature_present*. À la prochaine itération, on passe par la macro *wrap_lig* (qui est l'équivalent de *wrap_up* de la procédure *try_break*) qui va effectivement crée un nœud de ligature, et l'ajouter à *t*. Il nous faut plusieurs itérations pour obtenir un nœud de ligature, cela est dû au fait que la ligaturation peut se poursuivre par l'ajout de nouveaux glyphes. Il faut donc être sûr qu'elle est complète avant de créer le nœud.

Nous allons illustrer le fonctionnement de *reconstitute* par un exemple, légèrement plus exotique que celui que nous avons choisi pour *try_break*. Soit une fonte avec les ligatures et crénages suivants :

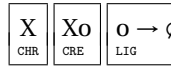
```
(LIGTABLE
   (LABEL C X)
   (/LIG/ C Y C o)
   (KRN C o R -0.319446)
   (STOP)
   (LABEL C o)
   (KRN C Y R -0.319446)
   (STOP)
   )
```

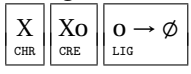
Cela signifie qu'à chaque fois que l'on lui fournit la chaîne XY, TEX va composer « XoY », avec des crénages entre « X » et « o », et entre « o » et « Y ».

Nous allons appliquer *reconstitute* à la chaîne XYetc, en ayant choisi cette fonte. Nous allons reprendre la notation utilisée dans la description de *try_break* : x|y est la situation du curseur à un moment donné (cela signifie que *cur_l* est x et *cur_r* est y). Allons-y :

1. §906. initialisations : $t \leftarrow$ *hold_head*, *link*(*hold_head*) $\leftarrow$ *null*, $w \leftarrow 0$;
2. §908. *cur_l* récupère la première valeur de *hu*, qui est X, *cur_q* est initialisé à t, on ajoute le nœud $\boxed{\underset{\text{CHR}}{\text{X}}}$ à t à l'aide de la macro *append_charnode*. On a donc une mini-liste de nœuds $\boxed{\underset{\text{CHR}}{\text{X}}}$;
3. *cur_r* récupère sa valeur de *hu*, c'est Y. *cur_rh* est *non_char*, puisqu'aucune césure n'a été constatée. Le curseur est maintenant X|Y ;
4. on entre pour la première fois dans la boucle du §909 : on récupère le PLC de « X », on teste avec « Y » et on s'aperçoit que l'on a une ligature de type 3 (/LIG/) ;
5. on est dans le switch du §911, *cur_r* prend la valeur du glyphe de cette ligature, c'est-à-dire o, le curseur est X|o ;
6. on attache un *new_lig_item* à la pile *lig_stack*. Il s'agit d'un nœud de « pseudo-ligature » $\boxed{\underset{\text{LIG}}{\text{o} \rightarrow \emptyset}}$, contenant simplement le glyphe de la ligature ;
7. on passe une deuxième fois par la boucle du §909 : on récupère (de nouveau) le PLC de « X », on teste avec « o » et on s'aperçoit que l'on a un crénage w ;
8. §910. On entre dans *wrap_lig* et on en ressort aussitôt puisque *ligature_present* est désactivée ;

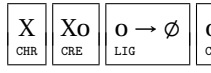
9. w étant non nul, on crée un nœud de crénage et on l'attache à t. Notre mini-liste devient : 

;
10. toujours dans §910, `lig_stack` étant non nul, il est temps de s'en débarrasser. On récupère le glyphe de la pseudo-ligature dans `cur_l`, le curseur est maintenant o|o (!) On active `ligature_present` et on entre dans la macro `pop_lig_stack` qui va éliminer le nœud de `lig_stack` et lancer `set_cur_r`. Cette dernière va de nouveau récupérer `hu[j+1]` dans `cur_r`, c'est Y. Le curseur est donc maintenant o|Y ;
11. on passe une troisième (et dernière) fois par la boucle du §909 : on récupère le PLC de « o », on teste avec « Y » et on s'aperçoit que l'on a un crénage w ;
12. §910. On entre dans `wrap_lig` et puisque `ligature_present` est activée, on crée un nouveau nœud de ligature avec `cur_l` comme glyphe (c'est-à-dire o) et `link(cur_q)` comme liste de nœuds supprimés. Dans notre cas, `link(cur_q)` est nul, il n'y a pas de nœud supprimé. On attache cette ligature à la mini-liste qui devient maintenant : 

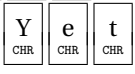
. On désactive `ligature_present` ;
13. puisque w est non nul, on attache également un nœud de crénage à la mini-liste, qui devient maintenant : 

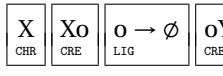
;
14. et on arrive à la fin de la procédure, qui retourne la valeur j. Puisque cette dernière n'a jamais été incrémentée, elle reste toujours égale à 1.

On peut se poser la question : comment est-ce possible d'avoir une valeur de `reconstitute` égale à 1, alors que la ligature intelligente génère trois glyphes ?

En fait, selon notre définition de *cut prefix*, on peut partager les nœuds générés par `XYetc` en deux parties, d'un côté on aura : 

, et de l'autre : 

. Donc la plus grande partie droite indépendante de la partie gauche est 

, et elle commence par le nœud généré par Y. On a donc bien $m+1=2$, et $y\_1 \dots y\_t =$ 

.

Ici se termine notre visite rapide du code de TEX. Nous tenons juste à rappeler au lecteur qu'il reste énormément d'aspects de TEX que nous n'avons pas effleurés, comme l'expansion de macros, l'alignement des tableaux, la composition des mathématiques, la routine de sortie, etc., sans parler des nouvelles fonctionnalités apportées par des successeurs de TEX tels que eTEX, pdfTEX, ou Oméga.

Dans une époque de mutation du livre, du document, de l'imprimé, les nouveaux impératifs nous invitent à reconsidérer les techniques du passé et à les adapter au présent, voire même à l'avenir. Tout comme la composition au plomb ou la monotype, TEX fait désormais partie de ce patrimoine de techniques et son étude ne peut être que bénéfique pour l'élaboration de nouveaux outils. À suivre…

Références bibliographiques

[1] Constantin Cavafis, *En attendant les barbares, et autres poèmes*, trad. par Dominique Grandmont, Gallimard, Paris, 1999.

[2] Edward Fredkin, *Trie Memory*, CACM, 3(9) :490-499, septembre 1960.

[3] Yannis Haralambous, *Fontes et codages*, O'Reilly, Paris, 2004.

[4] Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, *L'essentiel des structures de données en C*, Dunod, Paris, 1992.

[5] Donald Knuth, *The TEXbook*, Computers & Typesetting, vol. A, Addison Wesley, 1986.

[6] ————————, *TEX, The Program*, Computers & Typesetting, vol. B, Addison Wesley, 1986.

[7] ————————, *Digital Typography*, CSLI Lecture Notes Number 78, Stanford, 1999.

[8] Franklin Mark Liang, *Word Hy-phen-a-tion by Com-put-er*, thèse, département informatique, Stanford, août 1983.

[9] Wayne Sewell, *Weaving a Program*, van Nostrand Reinhold, New York, 1989.