Apprendre à programmer en TEX

Apprendre à programmer en TEX

Christian Tellechea

Septembre 2014

Ce livre est « autoédité », c'est-à-dire que la conception, les relectures et les corrections ont été faites par l'auteur. Malgré le soin qui y a été apporté, des fautes et des erreurs peuvent encore exister.

Toute erreur, omission ou toute proposition d'amélioration peut être signalée.

Email de l'auteur :

unbonpetit@netc.fr

Dépôt du code source, fichiers pdf et fichiers annexes :

https://framagit.org/unbonpetit/apprendre-a-programmer-en-tex/

Remontées de bugs, améliorations :

https://framagit.org/unbonpetit/apprendre-a-programmer-en-tex/issues

Les codes et algorithmes figurant dans ce livre sont donnés sans *aucune garantie* en vue de leur utilisation dans le cadre d'une activité professionnelle ou commerciale.

ISBN 978-2-9548602-0-6

Révision n° 2b, 20/12/2020

Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça, ce sont les attributs de la magie).

Dave Small

Préface

Il est déjà bien difficile de maitriser TEX comme langage de composition… C'est un vrai défi que d'en connaitre suffisamment les méandres pour l'utiliser comme langage de programmation [1]. Et, c'est une mission quasi impossible que d'écrire un livre là-dessus, j'en parle en connaissance de cause ! D'ailleurs, bien peu de choses existent sur le sujet. Non pas que les gens sachant programmer en TEX soient rares, ils sont simplement assez sages et modestes pour ne pas s'embarquer dans une telle galère. Inconscient, insouciant et trop sûr de moi, j'ai commencé à écrire sans mesurer les difficultés auxquelles j'allais me heurter. Je ne sais toujours pas quelle folie m'a poussé à poursuivre cette aventure jusqu'au bout. Même si un peu de vanité a certainement joué un rôle, ma motivation n'a jamais été de montrer que « moi, je peux » ; il ne s'agit pas d'un péché d'orgueil. Le but poursuivi, moteur plus fort que les difficultés rencontrées au cours de cette aventure, a été d'aider ceux qui, comme moi lorsque j'étais débutant en programmation TEX, restent devant un code écrit en TEX comme devant un texte en chinois ancien : plongés dans un abime d'incompréhension.

Quiconque connait un peu TEX a dû sourire en lisant la citation de Dave Small. Et a dû se dire qu'avec TEX bien plus qu'avec d'autres langages, on allait forcément parler de magie noire à un moment ou à un autre ! Car il est vrai que TEX est un langage difficile, tortueux et réputé comme tel. On ne compte plus ceux qui font tout pour ne pas s'y frotter, ceux qui le haïssent ou le rejettent ni ceux qui, bien que l'utilisant, se traitent de fous tant des choses apparemment simples sont en réalité compliquées. TEX est bourré de défauts, collectionne les lacunes, a une syntaxe à coucher dehors tant elle est irrégulière, mais malgré tout cela, on se prend à l'apprécier un peu comme on pourrait apprécier les défauts d'un ami de longue date, taciturne et bourru. Tout comme cet ami pourrait envier certains côtés aux « nouveaux communicants » dont nous sommes désormais entourés, TEX ne soutient pas non plus la comparaison avec les langages modernes. Mais, l'utiliser pour programmer procure un plaisir certain ainsi qu'une sensation de liberté et de créativité qu'on ne retrouve que dans une moindre mesure avec les langages de programmation récents.

1. TEX a été créé pour la composition et si son créateur l'a été doté de ce qu'il faut pour programmer, ce n'est pas par choix, mais parce qu'il a été obligé de le faire (un peu à contrecœur) !

J'invite donc mon lecteur, armé de patience et de persévérance, à entreprendre ce périple qui s'annonce ardu, mais riche de nouveautés et d'originalité. La meilleure chose que je lui souhaite est d'arriver à ses fins et par là même, atteindre aussi le but dans lequel ce livre a été écrit : acquérir une autonomie dans la programmation en TEX. Ce livre, qui n'est qu'une simple introduction à la programmation en TEX, sera peut-être un départ pour de nouveaux horizons qui, après beaucoup de pratique et sans utilisation de magie noire – c'est garanti ! –, le mèneront, qui sait, à devenir un TEXgourou…

Table des matières

Conventions adoptées dans ce livre

Les codes

Par souci de pédagogie, les codes sont très nombreux. Ils sont présentés de telle sorte que le code précède l'affichage qu'il génère. En voici un, le premier d'une longue série :

Code 1

```
1 Voici le code en \TeX{} du premier exemple.% ceci est un commentaire
2
3 On peut observer l'affichage qu'il produit juste au-dessous !
```

Voici le code en TEX du premier exemple.
On peut observer l'affichage qu'il produit juste au-dessous !

Sauf pour des cas très exceptionnels qui seront signalés, il ne peut y avoir de « triche » sur l'affichage : il est très exactement le résultat du code qui se trouve au-dessus. En effet, les codes sont tapés *une seule fois* dans le code source de ce livre. Une macro est chargée de stocker ce code pour ensuite fonctionner en deux temps. Elle l'affiche tel quel dans la partie haute du cadre puis laisse TEX le compiler et afficher le résultat dans la partie basse.

Les codes donnés produisent donc les résultats affichés, mais le lecteur doit garder en tête les limitations suivantes :

- malgré les relectures effectuées sur chaque code, ils ne sont pas garantis sans « bogue ». Certains bogues étant particulièrement difficiles à débusquer, il serait même très étonnant que certains n'en contiennent pas !
- dans la plupart des cas, les codes donnés ne sont qu'une façon de faire *parmi beaucoup d'autres*, le nombre d'alternatives tendant à augmenter avec la complexité de la tâche à exécuter. Ces codes sont donc à considérer comme *une* solution et non pas comme *la* solution. Ils n'ont donc pas valeur de méthode absolue qui serait meilleure que toute autre ;
- ils ne fonctionnent que pour un certain domaine de validité (souvent implicite) des arguments de leurs macros, car élargir ce domaine suppose des

complications qu'il est long et fastidieux de programmer. On peut donc toujours trouver des cas plus ou moins spéciaux où les arguments contiennent des choses qui feront « planter » une macro.

Par exemple, si une macro attend un nombre entier et qu'on lui donne la lettre « a » comme argument, TEX va protester avec l'erreur « `missing number` ». Par ailleurs, des caractères spéciaux comme « # » seul, « % », etc. vont inévitablement provoquer des plantages s'ils sont écrits dans les arguments des macros. Sauf dans les cas où ces caractères spéciaux sont gérés, les macros attendent des arguments « gentils », c'est-à-dire constitués de caractères ayant un code de catégorie de 10, 11 ou 12 (voir page 27) ;

– les codes proposés ne sont pas toujours *optimisés*, c'est-à-dire que leur vitesse d'exécution tout comme la consommation de mémoire qu'ils requièrent peuvent être améliorées. Ils répondent avant tout à deux contraintes, celle d'illustrer la notion qui est en train d'être abordée tout en utilisant des notions précédemment vues et celle de ne pas inutilement compliquer le code.

J'invite le lecteur à *lire* les codes proposés, même s'ils sont rébarbatifs. Lire n'est pas survoler ! Lorsque les codes deviennent assez complexes, il faut vraiment faire un effort mental important et se plonger dedans afin de comprendre comment ils fonctionnent. Les commentaires insérés dans le code, toujours très nombreux, fournissent une aide pour cela, mais la gymnastique mentale consistant à analyser l'enchainement des instructions et comprendre comment évoluent les arguments des macros reste nécessaire.

Les exercices

Les exercices, également en grand nombre et de difficulté variable, sont souvent proposés au lecteur au fur et à mesure que de nouvelles notions sont présentées. Ils sont précédés d'un « ■ » et immédiatement suivis de leur solution, choix discutable, mais qui s'est imposé. Voici à ce propos le premier exercice :

■ **Exercice 1**

Quel argument principal peut-on trouver pour justifier que les exercices soient immédiatement suivis de la solution ?

□ **Solution**

Se décider a été difficile, mais qu'y a-t-il de plus lassant que de passer son temps à tourner les pages pour aller débusquer à la fin du livre la solution d'un exercice que, bien souvent, on n'arrive pas à résoudre ? Ou qu'on ne *souhaite* pas résoudre ! Sur ce point, l'expérience un peu pénible du TEXbook (que vous avez bien sûr déjà lu et relu !) a été décisive et m'a poussé à présenter les choses différemment. Il me semble que le lecteur, supposé adulte et ayant un certain contrôle de lui-même, peut se retenir de lire la solution s'il souhaite vraiment chercher un exercice. ■

Éléments de code

Les chevrons "⟨" et "⟩" ont vocation à symboliser des éléments dont la nature est exprimée par les mots se trouvant entre ces chevrons. Ainsi, ⟨*nombre*⟩ est censé représenter… un nombre ! Selon la même notation, on aurait \⟨*macro*⟩, ⟨*texte*⟩, ⟨*code*⟩ (pour du code source en TEX), ⟨*car*⟩ (pour un caractère), ⟨*dimension*⟩, etc.

Un espace dans le code source peut être mis en exergue, soit parce qu'il revêt une importance particulière, soit parce sa présence est importante et qu'il doit être clairement être visible. Dans ces cas, il est noté « ␣ ».

Un caractère isolé peut parfois être encadré pour qu'il soit aisément identifiable, comme l'apostrophe inverse « `` ` `` ».

Lorsqu'il peut y avoir ambigüité sur les caractères composant le nom d'une commande ou bien parce que nom contient des caractères inhabituels, le nom est encadré comme dans « `\a_1\b7` » (car une commande peut tout à fait porter un nom pareil !).

Les macros propres à ce livre

Tout au long de ce livre, des macros sont définies dans des codes ou dans des exercices. Lorsqu'elles sont définies, elles sont suivies du signe « ⋆ » mis en exposant comme dans `\gobone`⋆. Elles disposent d'un index à part (voir page 559) où l'on peut aisément trouver le numéro de page où chacune d'entre elles a été mentionnée. Lorsqu'elles sont mentionnées dans l'index général, elles sont également suivies de « ⋆ ».

Anglicismes

Si à mon grand regret l'anglais – que dis-je, le *globish* ! – tend à s'imposer comme langue universelle de communication entre personnes ne parlant pas la même langue, il s'est depuis longtemps imposé comme langue en programmation. Il est en effet indéniable que les mots-clés de presque tous les langages de programmation sont en anglais. Ceci a conduit à employer des mots anglais pour désigner des notions couramment utilisées dans le monde de la programmation. J'ai donc fait le choix facile, pour quelques mots, d'utiliser la version anglaise.

Anglais	Français	Signification
token	entité lexicale	séquence d'un ou plusieurs caractères du code source reconnue et assimilée par TEX comme un élément indivisible selon les règles de l'analyseur lexical qui seront exposées dans ce livre.
tokenization	segmentation	s'applique à une portion du code source : transformation des caractères du code source en tokens.
catcode	code de catégorie	s'applique à un token : entier sur 4 bits (0 à 15) affecté à un token lorsqu'il est lu par TEX.
package	extension	mot générique signifiant « fichier externe permettant d'accroitre les possibilités de TEX ». Bien souvent, il s'agit de code source mettant à disposition des macros permettant de réaliser des tâches complexes.

Les règles

Au fur et à mesure de la progression, des règles précisant la façon de fonctionner de TEX sont inscrites dans des cadres sur fond gris. Toutes ces règles sont reprises en annexe à partir de la page 527.

Orthographe rectifiée

Lorsque j'ai vu la moue un peu dubitative des rares personnes ayant lu quelques passages de ce livre, j'y ai vu l'embarras de ceux qui, me découvrant incapable d'écrire quelques phrases sans faute, se préparaient à me faire des remarques sur mon orthographe fautive.

J'ai fait le choix, comme l'indique le logo ci-dessous, d'utiliser l'orthographe rectifiée de la réforme de 1990.

Par conséquent, la phrase « *nous sommes surs que la boite va disparaitre* » est orthographiquement correcte. Il ne manque aucun accent circonflexe, même s'il en fallait trois avant la réforme. Que l'orthographe rectifiée soit encore actuellement considérée comme fautive par la majorité des personnes me laisse un peu interrogatif. Malgré plus de deux décennies pendant lesquelles elle a été *recommandée*, le fait qu'elle soit si peu reconnue et pratiquement absente de l'enseignement ne s'explique pour moi que par le rapport excessivement rigide et crispé que nous entretenons avec notre orthographe.

Bien évidemment, ce livre étant 100% *fait-maison*, je ne peux le garantir exempt de coquilles, d'accords fautifs ou de lourdeurs. Je présente d'ores et déjà mes excuses à mes lecteurs pour les désagréments que cela leur procurera.

Introduction

Sommaire

Chapitre 1

TEX, LA BRUTE ET LE TRUAND

1.1. Les spécificités de TEX

Le programme `tex` est conçu pour lire du code source (le plus souvent enregistré dans un fichier de type « texte ») puis « compiler » ce code afin de produire un document affichable disponible dans un fichier de type `dvi`. Un tel programme, transformant du code source en document affichable, est appelé « *moteur*[1] ». Le langage TEX[2] met à disposition des commandes qui sont appelées *primitives*, tout en permettant de *définir* d'autres commandes, appelées « *macros-instructions* » ou « *macros* ». Compte tenu de leur puissance, ces dernières sont d'un emploi extrêmement courant et sont donc pratiquement indispensables. Elles peuvent être définies par l'utilisateur à l'intérieur même du code source, mais peuvent aussi être contenues dans un fichier externe appelé extension ou « *package* » qui sera lu à un endroit du code source choisi par l'utilisateur. Dernière possibilité, des macros peuvent également être précompilées pour être incorporées dans un exécutable appelé « *format* » qui embarque le langage TEX dans sa totalité ainsi que des macros précompilées. Ainsi, lorsqu'on lance l'exécutable d'un format, ces macros sont déjà définies avant que le début du code source ne soit lu.

Les formats les plus connus et les plus utilisés[3] sont :

– plain-TEX, dont l'exécutable est « `tex` », où les macros « `plain` » ont été dé-

1. On dit aussi parfois « *compilateur* ».

2. Il ne faut pas confondre le *moteur* `tex` qui est exécuté par un ordinateur et le *langage* TEX. Ce langage met à disposition des commandes pour produire un document, mais aussi des commandes pour *programmer*. C'est d'ailleurs le but de ce livre que d'expliquer comment utiliser ces dernières.

3. On ne peut pas vraiment parler de fréquence d'utilisation des formats les plus « célèbres », car n'importe qui peut se construire son propre format à la carte. La manœuvre à suivre ne sera pas décrite dans ce livre.

finies par l'auteur de TEX lui-même. Il est donc à noter que lorsqu'on lance le programme `tex`, en plus du langage TEX *nu*, on dispose aussi de toutes les macros de `plain` ;

- LATEX, dont l'exécutable est « `latex` », qui est un recueil de macros, écrites et mises en cohérence à l'origine par Leslie LAMPORT et dont le but est de fournir des macros de haut niveau pour la composition ;
- ConTEXt qui est un peu à part des deux précédents en ceci qu'il n'est pas basé sur le moteur `tex` mais sur un autre moteur permettant l'accès au langage *lua* ;
- pour mémoire, en voici de très peu connus ou très anciens, mais qui ont le mérite d'exister : `phyzzx`, `psizzl`, `Lollipop` et `TeXsis`. Le plus original de la bande est sans doute `StarTeX` qui a une approche très éloignée de tous les autres puisque la syntaxe des commandes se rapproche au plus près de celle de l'`HTML` !

Le langage TEX et ses formats sont réputés produire des documents de grande qualité typographique, mais TEX est vieux. Son concepteur, Donald KNUTH, l'a publié en 1982 [4], époque où aucun des langages « modernes » n'existait encore. Pour le programmer, il a utilisé un vieux langage s'il en est, le Pascal. TEX est assurément très stable, probablement exempt de bug, mais indéniablement *très vieux* à l'échelle du temps informatique. On peut d'ailleurs s'étonner qu'entre 1982 et le début des années 2000, une aussi longue période se soit écoulée sans évolution majeure. Certes, entre temps, un nouveau moteur « ε-TEX » a vu le jour où de nouvelles primitives ont ajouté des fonctionnalités bien pratiques qui n'existaient pas dans le langage TEX initial. Plus récemment encore, le moteur « pdfTEX » a été écrit par Hàn Thế THÀNH et de nouvelles fonctionnalités ont été ajoutées, tout en conservant celles de ε-TEX. Mais ce qui fait la particularité de pdfTEX est qu'il est capable de produire *directement* un fichier `pdf` à partir du code source sans passer par le fichier `dvi` jusqu'alors incontournable.

Au-delà de son âge, mais peut-être en sont-ce des conséquences, TEX a deux particularités qui le rendent peu attractif, voire carrément repoussant pour beaucoup de personnes qui s'y essaient.

Tout d'abord, la façon intrinsèque que TEX a de fonctionner, certes très originale, ne ressemble pas à celle qui est devenue la norme dans la plupart des langages de programmation. Ceci est *extrêmement* déroutant et conduit bien souvent à une incompréhension puis un rejet de ce langage, même pour des programmeurs avertis. Car avec TEX, les méthodes de programmation changent et donc, c'est aussi la façon de penser un algorithme qui change !

Pour résumer à l'extrême, TEX est un langage *de macros*, c'est-à-dire que les appels aux macros provoquent le remplacement de ces macros (et de leurs arguments) par un nouveau code. Ce code – ou « texte de remplacement » – est celui qui a été donné à la macro lors de sa définition. De plus, et c'est une originalité supplémentaire en TEX, lorsqu'on définit une macro, non seulement on définit son texte de remplacement, mais on définit *aussi* la syntaxe avec laquelle elle devra être appelée. Enfin, et c'est là un des points les plus difficiles, TEX ne procède à *aucune*

4. Une version plus récente « TEX90 » a été publiée en 1990 où de nouvelles primitives ont été incorporées et où la lecture du code est désormais sur 8 bits au lieu de 7 pour « TEX82 ».

évaluation, ni du texte de remplacement lors de la définition d'une macro, ni des arguments qui lui sont donnés lorsqu'elle est appelée.

Voici un exemple qui peut illustrer cette absence d'évaluation. Pour comprendre le problème, tenons-nous à l'écart de la syntaxe de TEX où les commandes doivent commencer par « \ » et les arguments doivent être entre accolades. Adoptons plutôt une syntaxe « générique » où les commandes sont écrites normalement et où les arguments se trouvent entre parenthèses. Imaginons qu'une macro « `left` » ait été créée et qu'elle admette deux arguments. Le premier de type chaine de caractères alphanumériques, noté « `A` » et le second de type entier, noté « `i` » : le résultat de cette macro est constitué par ce qui se trouve à gauche du caractère n° `i` de la chaine `A`. Par exemple, le code :

```
left(bonjour)(4)
```

donnerait les caractères se trouvant à gauche de 4e lettre, c'est-à-dire « bon ».

De la même façon, admettons qu'il existe aussi une macro « `right` » qui elle, renvoie ce qui se trouve à droite du caractère n° `i` dans la chaine `A`. Et donc :

```
right(bonjour)(4)
```

donnerait les caractères « our ».

Voici maintenant un raisonnement extrêmement répandu en programmation et c'est d'ailleurs le cœur du problème. N'importe quel programmeur pense – avec raison – que pour isoler ce qui se trouve entre les caractères n° 2 et n° 5 dans la chaine « `bonjour` » (c'est-à-dire les caractères « `nj` »), il suffit d'écrire :

```
right(left(bonjour)(5))(2)
```

Ce raisonnement suppose que les arguments de `right`– ici en particulier le premier – ont été évalués *avant* que la macro n'entre en jeu. Ce premier argument qui est « `left(bonjour)(5)` » serait donc remplacé par son résultat « `bonj` » et finalement, après cette évaluation, tout se passe comme si on avait l'appel :

```
right(bonj)(2)
```

Or, cette évaluation n'a pas lieu avec TEX ! Essayons de nouveau d'appréhender ce code, mais adoptons un esprit TEXien :

```
right(left(bonjour)(5))(2)
```

TEX exécute le code de gauche à droite donc, lorsqu'il exécute la macro `right`, il va lui transmettre ses deux arguments qui sont « `left(bonjour)(5)` » et « `2` ». Et donc, les caractères renvoyés seront tout ce qui se trouve à droite de la 2e lettre du premier argument, c'est-à-dire « `ft(bonjour)(5)` ».

Le simple fait que le premier argument ne soit pas évalué conduit non seulement à un résultat radicalement différent, mais surtout implique que si l'on souhaite un comportement « normal », c'est au programmeur d'intégrer à la macro un mécanisme qui se charge d'évaluer cet argument avant que le code utile de la macro ne traite le résultat. Pour ceux qui viennent d'autres langages, cette charge de travail supplémentaire est souvent ressentie comme une corvée, sans compter que le code s'en trouve alourdi et donc moins lisible.

Une bonne partie des problèmes que rencontrent les personnes non habituées à TEX vient de la confusion qu'ils font entre :

– l'affichage qui résulte d'un code ;

– ce code lorsqu'il est dans l'argument d'une macro.

L'explication de cette confusion tient au fait que la plupart des langages ont un mécanisme – appelé « préprocesseur » – qui *lit, analyse, évalue puis remplace les arguments par leurs résultats*, et ce avant que la macro ne soit exécutée. Avec TEX, ce préprocesseur est réduit à sa plus simple expression : il ne fait que *lire* et *tokenizer* cet argument [5].

Quelle que soit l'affinité – ou l'adoration – que l'on éprouve envers TEX, on est bien obligé de reconnaitre que cette façon de faire n'est pas *naturelle*. Car notre cerveau est lui aussi doté d'un préprocesseur intelligent qui agit de façon semblable à celui des langages de programmation « normaux ». Par exemple, n'apprend-on pas en mathématiques que :

$$\text{si } f(x) = x^2 \quad \text{alors} \quad f(2 \times 3 - 1) = 25$$

Avant tout calcul concernant la fonction f, ne procède-t-on pas spontanément à l'évaluation de l'argument « $2 \times 3 - 1$ » pour le remplacer par le résultat, c'est-à-dire le nombre 5 ? Ce faisant, ne procède-t-on pas comme les langages de programmation modernes qui, eux aussi, évaluent les arguments des macros ?

Tout ceci conduit donc à une première règle, probablement une des plus importantes de toutes :

1 - Règle
Si un code quelconque, notons-le <*x*>, produit un affichage <*y*> alors, sauf cas très particulier, il n'est pas équivalent d'écrire <*x*> ou <*y*> dans l'argument d'une macro.

L'autre inconvénient de TEX qui rebute les personnes habituées à des langages plus modernes est que, pris comme un langage de programmation, TEX est doté du *strict minimum* ! Pour parler avec modération, l'*extrême minimalisme* [6] de ce langage est totalement déroutant, car toutes les facilités auxquelles les langages modernes nous ont habituées depuis longtemps ne sont tout simplement pas implémentées dans TEX. On ne parle pas ici de structures évoluées comme de pointeurs ou pire, de programmation orientée objet. Non, on parle de structures de contrôle des plus *basiques*. Il va falloir renoncer à la création de variables typées, aux opérateurs logiques bien pratiques lors des tests (« `or` », « `and` », « `not` »), aux boucles (« `for` », « `while` », etc.), aux tableaux, aux nombres décimaux et aux opérations scientifiques, ainsi qu'à beaucoup d'autres choses. En TEX, on doit se contenter de récursivité, de tests simples, de nombres entiers et des quatre opérations arithmétiques, et encore avec quelques limitations pour ces dernières. Et si l'on veut profiter de structures plus évoluées, il faut les programmer soi-même ! Heureusement, cela est toujours possible, car TEX, dans son minimalisme, a été doté de tout ce qu'il faut pour construire n'importe quel algorithme, aussi complexe soit-il. On qualifie un tel langage de « complet ». On aborde d'ailleurs dans ce livre la programmation de structures de contrôle plus évoluées que celles directement accessibles par TEX.

Évidemment, lorsqu'on programme en TEX, on ne se trouve pas au niveau de

5. Dans ce cas, on a affaire à un préprocesseur de bas niveau, le plus bas qui soit, et on parle de « préprocesseur lexical ».

6. D'autres, moins bienveillants, parleraient de « pauvreté ».

l'assembleur [7] car tout de même, TEX est un langage de bien plus haut niveau. En revanche, ce qui est certain, c'est qu'on programme à un niveau beaucoup plus bas que celui de tous les langages actuels. Cela oblige donc à revenir aux fondamentaux de la programmation, et ce retour aux sources est parfois vécu comme une régression par les personnes venant d'autres langages. Pourtant, contrairement à ce que l'on peut penser, revenir aux concepts de base de la programmation aide à élaborer des algorithmes plus efficaces et mieux construits (ceci étant vrai quel que soit le langage utilisé), oblige à mieux comprendre les méthodes sous-jacentes aux structures complexes et en fin de compte, fait accomplir des progrès en algorithmique tout comme en compréhension de la façon dont fonctionne un ordinateur.

1.2. La relève

Ces contraintes intrinsèques à TEX font que, petit à petit, de plus en plus de gens gravitant autour de la sphère TEX – des utilisateurs de LATEX pour la plupart – ont admis que ce langage était devenu obsolète, tant pour la composition que pour la programmation. À leurs yeux, il devenait convenable, sinon de le cacher aux yeux des utilisateurs, au moins de proposer des alternatives considérées comme viables.

Le problème est que le format `latex` offre une collection de macros précompilées appelées LATEX dont le but est de proposer des commandes de haut niveau pour la composition mais hélas, *presque rien* pour la programmation ! Pour pallier ce manque, les utilisateurs de LATEX se sont vus proposer des extensions spécialisées qui rendent la vie du programmeur plus simple et qui mettent à disposition des commandes permettant de créer facilement des boucles ou autres structures de contrôle de plus haut niveau.

Le progrès que cela représente est considérable, car la facilité d'utilisation est au rendez-vous. Il devient ainsi *très* facile de faire appel à une boucle de type « `for` ». Bien sûr, chaque package a sa propre syntaxe et ses propres instructions. Mais finalement, beaucoup d'entre eux ont une intersection non vide qui est la mise à disposition de commandes sur les boucles et les tests notamment. Étant donné le grand nombre de packages disponibles pour la seule programmation, il est presque impossible de connaitre les instructions de tous les packages. Le travers que cela créé est que certains utilisateurs deviennent totalement dépendants d'un package auquel ils sont habitués, quitte à ce que cela confine parfois à l'absurde. Un des exemples les plus frappants est le package « ti*k*z » dont le but est de proposer des commandes de très haut niveau pour faire des dessins vectoriels. C'est un package d'une taille absolument gigantesque qui compte à peu près 300 000 [8] lignes de code, écrit en TEX et en perpétuelle évolution et qui propose, en plus de commandes de dessin vectoriel, des outils qui facilitent la vie en programmation. Bien entendu, on y trouve des commandes qui permettent de construire des boucles. L'absurde de la situation est que parfois, certains utilisateurs dépendants de ti*k*z et incapables de programmer par ignorance des commandes de TEX, en viennent à charger un package aussi volumineux pour ne se servir que d'une instruction en vue de créer une

7. Le « langage machine » est le langage de plus bas niveau qui soit, c'est-à-dire le moins compréhensible par un humain : ce sont des commandes directement compréhensibles par le microprocesseur qui sont des assemblages de bits ou si l'on préfère, des suites de 0 et de 1.

Ces suites de bits, traduites en instructions aisément mémorisables par un humain, constituent le langage « assembleur ».

8. À comparer avec les 8 000 lignes du code de LATEX !

boucle, le tout en dehors de tout contexte de dessin. Certes, la solution fonctionne, mais l'efficacité n'est pas vraiment optimale.

Le projet qui est actuellement en pleine phase d'expérimentation est LaTeX3. Il s'agit d'un projet qui relève un défi très ambitieux, celui de remplacer et d'améliorer l'actuelle version de LaTeX. Ce projet, en maturation intense depuis de nombreuses années et animé par une équipe de codeurs très expérimentés, arrive actuellement à un stade où des pans entiers, bien que toujours en évolution, deviennent utilisables. Le pan qui émerge en ce moment est carrément un nouveau langage de programmation entièrement écrit en TeX ! Comme TeX est un langage complet, tout est possible, même le plus inattendu ! Ce nouveau langage de programmation offre d'innombrables possibilités que TeX ne proposait pas, certaines extrêmement ardues à programmer. De plus, les auteurs ont essayé autant qu'ils le pouvaient d'inclure un préprocesseur, actionnable à volonté et agissant sur les arguments des macros individuellement. Un code écrit dans ce nouveau langage devient donc beaucoup plus épuré et lisible que du code TeX, autant par la présence de ce préprocesseur que par l'extraordinaire diversité des macros mises à disposition. Celles-ci couvrent des domaines aussi variés que :

- toutes les structures de contrôle basiques : tests, boucles ;
- nombres décimaux en virgule fixe ou flottante et les opérations arithmétiques et scientifiques sur ces nombres ;
- recherche, remplacement dans une chaine de caractères avec la possibilité de faire appel à des critères dont la syntaxe est proche de celle des « expressions régulières [9] » ;
- tri de données ;
- etc.

Le revers de la médaille est qu'il faut apprendre ce nouveau langage dont la syntaxe est suffisamment éloignée de celle de TeX pour dérouter, voire rebuter un grand nombre d'utilisateurs habitués depuis des années voire des décennies à celle de TeX ou LaTeX. Ce nouveau langage se veut en effet beaucoup plus régulier que ne l'est TeX, à tel point qu'il peut apparaitre fade, aseptisé et domestiqué en regard de l'original qui, par ses défauts un peu folkloriques, peut être ressenti comme davantage original et attachant. Le point le plus discutable est que TeX, le langage sous-jacent, est volontairement rendu totalement invisible pour l'utilisateur final de LaTeX3, toujours selon la volonté que TeX doit être caché à l'utilisateur, car jugé trop difficile, obsolète et contraignant. LaTeX3 créé finalement une dichotomie entre les utilisateurs. Les uns, pragmatiques, désireux de savoir et prêts à apprendre un nouveau langage, se moquent éperdument du *vrai* langage qui se trouve sous cette « couche », trop contents d'avoir enfin à leur disposition un langage au comportement quasi « normal ». Les autres, par curiosité intellectuelle et par désir de comprendre, ou tout simplement par rejet de LaTeX3, n'y adhèreront pas et préfèreront ouvrir le capot afin de voir ce qui se joue en coulisses.

Il va sans dire que ces derniers devront, si ce n'est déjà fait, se mettre à TeX et que les autres doivent sans tarder lire la documentation de LaTeX3.

9. Une expression régulière est une suite de signes, appelés « motif », qui décrit selon une syntaxe précise des règles auxquelles doit satisfaire une chaine de caractère lors d'une recherche.

Un livre entier serait à peine suffisant pour explorer le monde des expressions régulières. Ce qu'il faut savoir est qu'elles sont très largement utilisées autant pour leur compacité que pour leur puissance.

1.3. La révolution

Mais que ces utilisateurs ne se ruent pas trop vite sur LaTeX3, car il y a autre chose. Depuis l'année 2005, un nouveau moteur est en train de voir le jour. Il s'agit de luaTeX, un moteur où bien sûr le langage TeX est toujours accessible, mais où est également rendu possible l'accès au langage « lua ». Ce langage, créé en 1993 a justement été écrit pour être embarqué dans des applications pour en étendre les possibilités. Lua est un langage *interprété* mais surtout, il possède tout ce que TeX n'a pas et donc tout ce que les langages modernes ont ! Parmi les fonctionnalités les plus essentielles, on peut citer :

- accès facile et « naturel » aux structures de contrôle habituelles : boucles (`repeat`, `while`, `for`), tests simples, tests composés avec des opérateurs logiques sur les conditions ;
- fonctions dont les arguments sont évalués par un préprocesseur ;
- nombres, qui sont d'un seul type, les décimaux à virgule flottante. Ceux-ci sont assortis des opérations arithmétiques et d'une bibliothèque de fonctions mathématiques ;
- variables typées (chaines de caractères, booléens, tables) ;
- accès aux fonctions du système (accès aux entrées/sorties, aux fonctions de date, d'heure) ;
- etc.

L'accès à `lua` peut donc grandement faciliter la vie d'un programmeur pour écrire des algorithmes qui autrement, seraient difficiles à coder en TeX. Cela ne doit tout de même pas faire oublier que certains algorithmes *doivent* être codés en TeX, car c'est le langage qui a le dernier mot en matière de composition.

Mais il y a bien plus important. Le plus étonnant est que `lua` permet aussi l'accès aux mécanismes les plus intimes de TeX. Les auteurs ont réussi le tour de force d'implémenter `lua` à un niveau tellement bas qu'il est possible d'agir sur le programme `tex` lui-même et d'accéder à des endroits auparavant inaccessibles et non modifiables (on dit qu'ils sont codés « en dur »). Cet accès ouvre des possibilités inouïes, car lors de la compilation et par l'intermédiaire de `lua`, il devient possible de modifier ou contrôler des comportements très fins de TeX, le rendant extrêmement configurable, bien plus qu'il ne l'était. Il devient possible, par exemple, d'effectuer ces actions qui étaient impossibles avant :

- relever les positions de toutes les espaces inter-mots d'un paragraphe afin de vérifier que certaines d'entre elles, au-delà d'un certain nombre, ne soient pas adjacentes et alignées verticalement, créant ainsi ce que l'on appelle en typographie des « rivières » ;
- lors de l'affichage, remplacer chaque occurrence d'un groupe de caractères par un autre, par exemple, tous les « onsieur » par « adame » ;
- etc.

1.4. Et le bon dans tout ça ?

Bien évidemment, de TeX, LaTeX3 et luaTeX, aucun n'est le bon, la brute ou le truand ! Chacun a ses propres caractéristiques et chacun cohabite avec les deux autres sans qu'aucun ne soit en compétition avec un autre. Au contraire, il faut

se réjouir puisque le monde de TEX, déjà d'une extrême complexité, mais presque figé depuis 1982 est en pleine effervescence et promet maintenant de façon certaine que des choses impossibles auparavant ne le seront plus. L'avenir s'ouvre riche en perspectives et sans rêver, ne peut-on pas imaginer, d'ici quelques années, un format basé sur le moteur luaTEX contenant les macros précompilées de LATEX3, le tout formant une unité réunissant les trois protagonistes qui sera d'une puissance et d'une souplesse jusqu'alors inconnues ? Peut-on déjà mesurer l'immense bond en avant que constituera cette symbiose ? Et que feront les packages de cette nouvelle ère s'ils savent exploiter les trois langages, quels miracles seront-ils capables de réaliser ?

Chapitre 2

Avant de commencer

2.1. La programmation TEX oui, mais pour qui ?

Finalement, à qui s'adresse ce livre ?

Tout d'abord aux utilisateurs de plain-TEX, LATEX ou d'un autre format qui souhaitent acquérir un peu d'autonomie dans la programmation en TEX. Car un jour ou l'autre c'est inévitable, on en a besoin, ne serait-ce que parce que les packages disponibles n'ont pas ne couvrent pas tous les cas ou parce qu'on ignore qu'un package répondant à un besoin précis existe. Qu'y a-t-il de plus frustrant que de renoncer en se sachant incapable d'écrire le moindre code non linéaire, surtout lorsqu'on en a un urgent besoin ? Quelle satisfaction intellectuelle retire-t-on lorsqu'au moindre problème urgent et incontournable, on est contraint de poser une question de programmation TEX sur un forum spécialisé ? Est-ce vraiment valorisant d'attendre une réponse, « copier-coller » le code donné par un connaisseur sans en comprendre le fonctionnement et se dire qu'au prochain problème, il faudra poser une question à nouveau ?

Ce livre s'adresse aussi aux curieux, à ceux désireux d'apprendre, ceux qui souhaitent soulever un coin du voile. En effet, TEX est réputé comme étant un langage réservé à quelques « initiés », voire carrément inaccessible si l'on n'est pas « TEX-compatible ». La lecture de ce livre dissipera un peu l'aura quasi mystique qui l'enveloppe – c'est en tout cas un des buts recherchés –, assouvissant les légitimes interrogations que se posent les « non-initiés » à son sujet.

Il s'agit aussi de faire redécouvrir qu'il existe d'autres voies en programmation et celle que prend TEX est des plus originales. Car finalement, tous les « poids lourds » des langages modernes ne tendent-ils pas à se ressembler tous ? Ne proposent-ils

pas un peu tous les mêmes fonctionnalités, au moins jusqu'à un certain niveau ? Il est assez excitant de découvrir un langage de macros tellement ceux-ci sont rares [1]. L'apprentissage d'un tel langage est excitant d'abord parce que cela étend la culture informatique, mais surtout parce cela ouvre de nouvelles voies en programmation en suscitant des méthodes de programmation nouvelles. Il est d'ailleurs reconnu que les langages de macros sont puissants, même s'ils procèdent à des remplacements *aveugles* de macros par leur texte de remplacement. Ce manque d'évaluation se paie par des erreurs parfois difficiles à corriger, car surgissant à des moments ou à des endroits très éloignés de l'origine du problème.

Bien sûr, nulle part nous n'apprendrons à programmer des macros aussi complexes que celles de LaTeX3, nous resterons beaucoup plus modestes. Le but de ce livre est de donner les clés pour comprendre *comment* il faut s'y prendre. La compréhension de quelques concepts doit pousser, au fur et à mesure de la lecture, à coder soi-même et essayer de voler de ses propres ailes. Il est en effet primordial, quel que soit le niveau atteint, de toujours expérimenter par soi-même, car se contenter d'une lecture linéaire ne peut suffire. Imaginerait-on qu'un joueur débutant de bridge, jeu complexe s'il en est, deviendrait subitement un bon joueur après avoir simplement lu d'un trait un ouvrage sur le sujet ? La réponse est évidemment négative, car tester, modifier, adapter, expérimenter, se « planter », apprendre de ses erreurs, recommencer, recommencer encore fait partie de tout apprentissage. Cela est valable dans tous les domaines, et en programmation – surtout en TeX – peut-être encore davantage !

2.2. Ce qu'il faut savoir

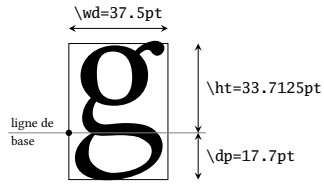
Il ne serait pas raisonnable d'aborder le côté langage de *programmation* de TeX sans avoir jamais pratiqué TeX, LaTeX ou un autre format en tant que langage de *composition*. Non, vraiment pas raisonnable. Cette pratique est indispensable, ne serait-ce que pour s'imprégner de la syntaxe et se familiariser avec l'outil. Ce livre ne s'adresse donc pas au débutant de la première heure, sauf inconscient ou masochiste qui souhaiterait se dégouter à tout jamais de TeX !

Mais tout compte fait et nous allons le voir ci-dessous, les choses à savoir ne sont pas légion.

2.2.1. TeX et la composition

Il faut avoir une petite idée de la façon dont TeX compose du texte. Le schéma mental le plus proche de la réalité est de se dire que TeX fonctionne par placement de boites sur la page. La géométrie des boites de TeX est assez facile à comprendre. Voici la lettre « g », écrite en très grand et dont on a tracé les contours de la *boite englobante* qui traduit les dimensions avec lesquelles TeX voit une boite :

1. Le langage « `M4` » est également un langage de macros.



Comme on peut le voir, la boite englobante a trois dimensions, tout comme n'importe quelle boite dans le monde de TEX :

- une largeur qui est ici de 37.5pt ;
- une hauteur, de 33.7125pt
- une profondeur égale à 17.7pt

TEX exprime naturellement les dimensions en *points* (mais nous le verrons, il y a bien d'autres unités possibles), et un centimètre contient exactement `28.45274pt`.

La hauteur et la profondeur s'étendent verticalement de part et d'autre de la *ligne de base*. C'est une ligne horizontale théorique autour de laquelle s'organise la géométrie d'une boite. Pour les boites contenant les caractères, sa position pratique coïncide avec le bas des caractères sans jambage comme « a, », « b », « c », etc.

Programmer en TEX est *facile.*

Chaque boite a un point particulier, appelé *point de référence* situé sur la frontière gauche de la boite englobante. Ce point de référence est, dans la boite contenant « g », représenté par un point « • ». Les boites sont donc correctement placées verticalement parce que leur point de référence vient sur la ligne de base de la ligne courante, et sont positionnées les unes à la suite des autres parce que le point de référence de la boite suivante est situé sur la frontière droite de la boite précédente.

TEX procède par à-coups, composant tour à tour des paragraphes. La mécanique fonctionne de cette manière : lorsque la commande `\par` est rencontrée, le paragraphe qui vient d'être lu et qui est jusqu'alors stocké en mémoire est « composé ». C'est le moment où TEX appréhende le paragraphe dans sa totalité pour décider où se feront les coupures de ligne. Chaque ligne de texte est ainsi constituée

- de boites contenant les caractères [2]. Dans un mot, chaque boite est adjacente à la précédente ;
- d'espaces qui peuvent, dans une certaine mesure, s'étirer ou se comprimer de telle sorte que la largeur de la ligne soit exactement celle de la largeur du paragraphe. La dernière ligne du paragraphe, moins large que ne l'est le paragraphe, a des espaces qui ont leur largeur naturelle.

Pendant tout le paragraphe, TEX positionne cette *liste* d'éléments à la suite les uns des autres de façon horizontale pour former chaque ligne : on dit qu'il fonctionne en « mode horizontal ». Ce mode est d'ailleurs celui dans lequel il se trouve pour la plupart du temps lorsqu'il compose du texte.

La composition proprement dite du paragraphe enferme chaque ligne dans une boite et les boites contenant les lignes sont positionnées *verticalement* les unes sous les autres, chacune étant séparée de sa voisine par un « ressort d'interligne » (page 245).

On peut observer ce ressort d'interligne dans ce paragraphe où la boite englobante de chaque ligne est tracée. On peut constater que la boite englobante d'une

2. Pour être clair, TEX ne connait à aucun moment le dessin de chaque caractère. Il ne connait que les dimensions de la boite qui le contient, qui est la boite englobante.

ligne ne touche pas celle de ses voisines : l'espace vertical qui les sépare est l'espace interligne. Si cet espace était nul, les lignes seraient beaucoup trop serrées et cela rendrait la lecture très pénible.

Après avoir formé le paragraphe (et donc après avoir exécuté la commande `\par`), le mode vertical est en vigueur, puisque TEX vient de placer les lignes du paragraphe précédent verticalement : il est donc prêt à positionner d'autres éléments verticalement les uns sous les autres. Bien évidemment, si un nouveau paragraphe commence juste après, TEX repasse en mode horizontal dès le premier caractère rencontré.

Au fur et à mesure que les lignes et autres objets remplissent verticalement la page, il arrive un moment où un certain seuil de remplissage est atteint et TEX décide où se situe la coupure de page. La page destinée à être composée est à ce moment enfermée dans une ultime boite. Cette boite signe la fin de l'imbrication des boites en poupées gigognes et est envoyée vers le fichier de sortie.

2.2.2. Quelques commandes utiles

`\par` et les changements de mode

La plus courante de toutes les primitives est sans doute `\par` puisque sans elle, il n'y aurait pas de paragraphe et donc pas de typographie. Ajoutons à ce qui a été dit ci-dessus la règle suivante :

2 - Règle

Lorsque TEX est en mode vertical, la commande `\par` est sans effet.

Par conséquent, si l'on se trouve en mode horizontal et si plusieurs commandes `\par` se suivent, seule la première est utilisée pour composer le paragraphe en cours et les autres sont ignorées.

On peut quitter le mode vertical sans afficher aucun caractère avec la macro de plain-TEX `\leavevmode`.

D'autres primitives commandent à TEX le passage en mode horizontal, mais font aussi autre chose :

- `\indent` insère au début de la liste horizontale une espace de valeur `\parindent`, qui est la primitive contenant la dimension du retrait de la première ligne d'un paragraphe (plain-TEX initialise cette dimension à `20pt` en procédant à l'assignation « `\parindent=20pt␣` ») ;
- `\noindent` demande à ce que le prochain paragraphe ne soit pas indenté, c'est-à-dire que sa première ligne ne commence pas en retrait.

Espaces sécables et insécables

La primitive `\kern`, suivie d'une dimension, insère dans la liste courante une espace insécable de la dimension voulue. Cette espace est insécable, car aucune coupure (de page ou de ligne) ne pourra se faire sur cette espace. Il est important de noter que `\kern` obéit au mode en cours ; l'espace est horizontale si TEX est en mode horizontal et sera verticale sinon. Voici ce qu'il se passe si l'on écrit `A\kern 5mm B` :

A B

L'espace de `5mm` se trouve entre le bord droit de la boite englobante de « A » et le bord gauche de la boite englobante de « B ».

Pour insérer une espace verticale supplémentaire de 5 mm entre deux lignes, il suffit de passer en mode vertical avec `\par` pour que `\kern` agisse dans ce mode. Ainsi, `A\par\kern 5mm B` se traduit par :

A

B

L'espace entre le bas de la lettre « A » et le haut de la lettre « B » est légèrement supérieure à 5 mm car le ressort d'interligne s'ajoute à l'espace spécifiée par `\kern`.

Si l'on veut insérer des espaces *sécables*, c'est-à-dire susceptibles d'être *remplacées*[3] par une coupure de ligne ou de page, il faut utiliser soit `\hskip`, soit `\vskip`. La première lettre désignant le mode dans lequel on veut que cette espace soit insérée et commandant le passage dans ce mode. Ces deux primitives doivent être suivies d'une dimension de ressort (voir page 242). Plain-TEX met à disposition deux macros `\quad` et `\qquad` insérant respectivement une espace horizontale sécable de `1em` et de `2em`. L'unité `em` dépend du contexte : c'est la dimension horizontale de la boite englobante de lettre « `m` », composée dans la fonte en cours.

La macro `\smallskip` insère verticalement à l'aide de `\vskip` l'espace élastique stockée dans le registre `\smallskipamount` et qui vaut dans plain-TEX « `3pt plus 1pt minus 1pt` ». Les variantes `\medskip` et `\bigskip` insèrent des espaces verticales égales respectivement à 2 fois et 4 fois `\smallskipamount`.

Les macros `\smallbreak`, `\medbreak` et `\bigbreak` se comportent comme leurs sœurs en « skip » mais elles vérifient tout d'abord que la précédente espace verticale qui les précède est inférieure à celle qui doit être insérée. Dans ce cas seulement, ces macros favorisent une coupure de page (par l'insertion d'une pénalité de -50, -100 ou -200), suppriment la précédente espace verticale et insèrent l'espace par l'intermédiaire de `\smallskip`, `\medskip` et `\bigskip`.

`\relax`

La primitive `\relax` est la plus simple à comprendre puisqu'elle ne provoque aucune action. Elle est cependant utile notamment pour marquer clairement la fin d'un nombre ou d'une dimension et évite que TEX n'aille chercher des choses au-delà. Elle fait aussi partie, mais de façon optionnelle, de la syntaxe des primitives `\numexpr` et `\dimexpr` de ε-TEX (voir pages 141 et 234).

Commandes de fontes

Les commandes suivantes, programmées dans le format plain-TEX, agissent comme des « bascules » et sélectionnent une forme de fonte :

3. Il faut bien lire « remplacées », car une espace sécable est supprimée si elle se trouve à un endroit de coupure. Les espaces sécables les plus courantes sont les espaces tapés dans le code pour séparer les mots.

Commande TEX	Équivalent LATEX	Exemple
\bf	\bfseries	**fonte en gras**
\it	\itshape	*fonte italique*
\tt	\ttfamily	fonte à chasse fixe
\sc	\scshape	FONTE PETITE MAJUSCULE

Par ailleurs, en mode mathématique, la primitive \scriptstyle fonctionne également comme une bascule au niveau de la taille de la fonte qui est affichée dans la taille des exposants. De la même façon, \scriptscriptstyle sélectionne une taille encore plus petite, celle des « exposants d'exposants » :

Code 2

```
3 tailles : $% entre en mode mathématique (espaces ignorés)
1^{2^3}% 2 est en taille "\scriptstyle" et 3 en taille "\scriptscriptstyle"
$% fin du mode math

normal $1$, petit $\scriptstyle 2$, très petit $\scriptscriptstyle 3$.
```

3 tailles : 1^{2^3}
normal 1, petit 2, très petit 3.

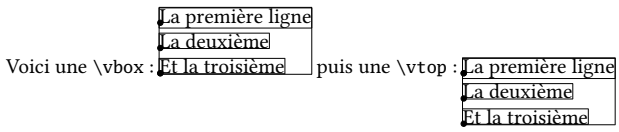
Les boites

Venons-en maintenant aux boites, non pas celles que TEX construit automatiquement lors de la composition, mais de celles qui sont mises à disposition de l'utilisateur pour qu'il puisse enfermer dedans un contenu arbitraire.

TEX met à disposition trois primitives permettant d'enfermer du matériel dans des boites :

- \hbox{⟨*contenu*⟩} enferme le ⟨*contenu*⟩ dans une boite horizontale, étant entendu que ce contenu ne peut être composé qu'en mode horizontal et doit donc être exempt de commande spécifiquement verticale (\par ou \vskip y sont interdits puisque passant en mode vertical) ;
- \vbox{⟨*contenu*⟩} construit une boite verticale dans laquelle chaque élément du ⟨*contenu*⟩ est empilé sous l'élément précédent et où règne le mode vertical *interne*. Le point de référence de la boite ainsi obtenue coïncide avec celui du dernier élément (le plus bas donc). Autrement dit, la ligne de base de la \vbox coïncidera donc avec la ligne de base de l'élément le plus bas. Par conséquent, s'il y a n éléments dans la \vbox, les $n-1$ premiers seront placés au-dessus de la ligne de base de la ligne courante comme le montre la figure ci-dessous ;
- \vtop{⟨*contenu*⟩} procède comme \vbox sauf que le point de référence de la \vtop coïncide avec celui du *premier* élément (le plus haut). Les $n-1$ derniers éléments sont donc au-dessous de la ligne de base de la ligne courante.

Voici deux constructions, l'une avec \vbox et l'autre avec \vtop où les frontières des boites ont été rendues visibles et où les points de références de chaque boite ont été symbolisés par un point « • ». Pour les besoins de l'exemple, chaque boite contient trois éléments.



Le mode en cours dans ces boites est le mode *interne*. Il s'agit du mode horizontal interne pour `\hbox` et du mode vertical interne pour `\vbox` et `\vtop`.

Une boite construite avec l'une des trois primitives `\hbox`, `\vbox` ou `\vtop` est *insécable*. Il n'existe donc aucun mécanisme qui permet de couper une boite ainsi construite. Il est donc nécessaire d'être prudent sur les débordements qui peuvent en résulter :

- en mode horizontal, ces débordements ont lieu dans la marge de droite ;
- en mode vertical interne, ils ont lieu dans la marge inférieure.

Pour illustrer ce propos, voici ce qui arrive si on tente d'enfermer ces **mots en gras** dans une `\hbox`. On peut constater que la coupure de ligne n'a pu se faire et que la boite dépasse dans la marge de droite. Cela signifie que malgré tous ses efforts pour comprimer ou étirer les espaces, couper des mots, mixer ces possibilités pour examiner plusieurs solutions, TEX n'a pu composer le paragraphe pour faire en sorte que cette `\hbox` se trouve entre les frontières du texte. Dans le cas présent, cet échec est dû à la trop grande largeur de la boite et au fait qu'elle se situe à un endroit proche du début du paragraphe, où la latitude d'organiser différemment ce qui précède est assez réduite. TEX avertit l'utilisateur qu'une anomalie dans la composition du paragraphe a eu lieu par un message dans le fichier `log` : c'est le fichier produit par TEX lors de la compilation d'un code source. Ce fichier contient des informations de compilation et a le même nom que le fichier compilé mais porte l'extension « `.log` ». Concernant ce débordement, on lit dans ce ficher « `Overful \hbox (9.74199pt too wide) in paragraph` », ce qui signifie qu'une boite horizontale (la boite horizontale dont il est question est ici la ligne en cours) est trop large de 9.74199pt par rapport à la largeur du texte.

2.3. Demandez le programme

Le chemin choisi pour aborder la programmation en TEX va sembler, c'est sûr, long et ingrat. Il obéit cependant à une logique qui apparaitra plus tard au lecteur. Les parties I et II ne sont pas consacrées à la programmation, mais aux nombreuses spécificités du langage TEX sans lesquelles il est impossible de construire le moindre programme. C'est par souci de clarté et pour bâtir des bases solides que ces spécificités sont abordées *avant* d'entrer dans le vif du sujet. Renoncer à ces explications préliminaires aurait conduit à d'innombrables digressions, chacune apparaissant à une nouvelle spécificité, et troublant plus le discours qu'autre chose. Ainsi, la partie I montrera comment et selon quelles règles TEX lit le code source pour le transformer en matière exploitable que sont les tokens. La partie suivante exposera les nombreuses spécificités des macros de TEX et comment les construire afin de maitriser cet outil fondamental de programmation.

Ce n'est qu'ensuite, à la partie III, que nous nous lancerons dans le grand bain et commencerons à bâtir des algorithmes. Nous verrons, à l'aide d'algorithmes relativement peu complexes, quelles méthodes de programmation nous pouvons mettre en œuvre en TEX et quels écueils nous devons éviter. La partie IV s'inscrit dans la continuité de la partie III, mais les méthodes employées diffèrent en ce sens que la granularité de la lecture du code effectuée par les macros sera plus fine.

Enfin, la dernière partie sera consacrée à des algorithmes un peu plus complexes mettant à profit les notions vues jusqu'alors.

Première partie

TEX et le code source

Sommaire

AVANT d'entrer dans le vif du sujet, il nous faut comprendre un des mécanismes fondamentaux de TEX, le premier dont il est question lorsque TEX fonctionne : comment lit-il le code qui est tapé par l'utilisateur ? Et qu'est-ce réellement du code tapé par l'utilisateur ? Les réponses à ces questions non triviales permettent non seulement de lever une petite partie du mystère qui entoure le fonctionnement de TEX mais surtout posent des bases essentielles.

Chapitre 1

Comment TEX lit-il le code source ?

Pour bien répondre à cette question, il faut d'abord définir ce qu'est du « code tapé par l'utilisateur ». Ce code, la plupart du temps stocké dans un fichier, est considéré et lu par TEX octet [1] par octet ; c'est d'ailleurs ainsi qu'il est stocké sur les disques durs de nos ordinateurs. Ainsi la résolution de lecture de TEX est toujours la même, le code tapé par l'utilisateur est vu par TEX comme une suite d'octets.

Un « éditeur » est un logiciel totalement indépendant de TEX dont les fonctions les plus basiques sont de permettre la saisie de *texte* et la sauvegarde de ce texte dans un fichier. Il existe des quantités d'éditeurs [2] différents, avec des fonctions plus ou moins nombreuses, parfois pléthoriques, certains étant même spécialisés dans la saisie de code TEX [3]. Lorsqu'on demande à l'éditeur d'enregistrer dans un fichier le texte que l'on a tapé, l'éditeur a la charge de *convertir* les caractères tapés par l'utilisateur en une suite d'octets qui seront enregistrés sur le disque dur sous forme d'un fichier texte.

Pour comprendre comment cette conversion est faite et quelle est la signification de ces octets, il faut savoir à quel « encodage » obéit l'éditeur lors de cette conversion. Un encodage est une convention qui lie de façon biunivoque un ca-

1. Un octet est une suite de 8 bits. Cette suite peut être comprise comme un nombre constitué de 8 chiffres en base 2. Si l'on convertit ce nombre en base 10 alors, ce nombre peut varier de $2^0 - 1$ à $2^8 - 1$, c'est-à-dire de 0 à 255 ou 00_h à FF_h en hexadécimal.

2. Citons pour les plus reconnus vim et emacs du côté de UNIX/GNU-linux et notepad++ du côté de Windows.

3. Là aussi, il y a du monde : aucTEX qui est une extension d'emacs pour le spécialiser pour TEX/LATEX, TEXmacs, kile, TEXmaker, TEXworks, etc.

ractère à un (parfois plusieurs) octet. Il est important de comprendre que c'est l'*utilisateur*, en paramétrant son éditeur, qui décide quel encodage il désire utiliser, même si évidemment, un choix par défaut est toujours sélectionné si l'on n'a pas fait la démarche de modifier le paramètre adéquat.

L'encodage ASCII [4] est commun à l'immense majorité des autres encodages ce qui fait que l'ASCII joue le rôle de plus petit dénominateur commun. Cette convention ASCII ne s'occupe que des octets compris entre 0 et 127 (ou 00_h et $7F_h$ en hexadécimal). TEX reconnait nativement l'ASCII dont on voit dans la table ci-dessous la correspondance entre octets et caractères. Pour toutes les implémentations de TEX, la relation entre octet et caractère affichable qui figure dans cette table est valide :

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*NUL* 0	*SOH* 1	*STX* 2	*ETX* 3	*EOT* 4	*ENQ* 5	*ACK* 6	*BEL* 7	*BS* 8	*HT* 9	*LF* 10	*VT* 11	*FF* 12	*CR* 13	*SO* 14	*SI* 15
1x	*DLE* 16	*DCI* 17	*DC2* 18	*DC3* 19	*DC4* 20	*NAK* 21	*SYN* 22	*ETB* 23	*CAN* 24	*EM* 25	*SUB* 26	*ESC* 27	*FS* 28	*GS* 29	*RS* 30	*US* 31
2x	SP 32	! 33	" 34	# 35	$ 36	% 37	& 38	’ 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3x	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4x	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5x	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6x	‘ 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7x	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	*DEL* 127

Dans ce tableau, les cases grisées portent les deux chiffres hexadécimaux qui formeront le numéro de l'octet. Le nombre en bas à droite de chaque case est le numéro de l'octet en base 10. Par exemple, le caractère « A » correspond à l'octet 41_h dont la valeur décimale est 65.

On remarque que quelques octets codent des caractères non affichables. Ceux-ci sont affichés en italique [5] dans la table. Ils ont des codes hexadécimaux compris entre 00_h et $1F_h$ inclus auxquels il faut rajouter le caractère de code $7F_h$.

Pendant la compilation, à chaque fois que TEX lit un octet – pour l'instant, disons par abus de langage un « caractère » –, il lui affecte une bonne fois pour toutes un « code de catégorie » ou « catcode » et de fait, chaque caractère lu dans le fichier a donc *deux* codes pour TEX :

– le code de caractère, c'est-à-dire la valeur de l'octet lu dans le code source et qui est lié à la table ASCII vue précédemment ;

4. ASCII est l'acronyme pour « American Standard Code for Information Interchange ».

5. Les acronymes dans les cases des caractères non imprimables correspondent à des noms qui ont été donnés à ces octets lors de la création de cet encodage. Pour la plupart, ces octets étaient insérés dans des flux de données lors de télétransmission et avaient une signification liée à la communication entre deux ordinateurs distants. Par exemple, l'octet *EOT* signifie « End Of Transmission » c'est-à-dire « Fin de Transmission », *NAK* signifie « Negative Acknowledge » pour « Accusé de réception négatif ». Ces fonctions sont évidemment sans objet pour un traitement de texte comme TEX. Ces caractères n'ont donc pas de signification particulière pour TEX à l 'exception de quelques caractères non imprimables dont la fonction est restée la même : *HT* pour « tabulation horizontale » (le caractère inséré avec la touche « Tab » ↹), *LF* pour « saut de ligne » et *CR* pour « retour charriot ».

– le code de catégorie qui est affecté par TEX lors de sa lecture. Ces codes de catégorie sont des entiers qui vont de 0 à 15 et qui donnent au caractère des propriétés qui sont celles de la catégorie à laquelle il appartient.

Voici les différentes catégories ainsi que pour chacune, les caractères qui les peuplent par défaut :

N° de catégorie	Signification	Caractères concernés
0	caractère d'échappement	\
1	début de groupe	{
2	fin de groupe	}
3	délimiteur de mode mathématique	$
4	tabulation d'alignement	&
5	retour charriot	*LF*
6	caractère de paramètre	#
7	exposant	^
8	indice	_
9	caractère ignoré	*NUL*
10	espace	SP et *HT*
11	lettres	`A...Z` et `a...z`
12	autres caractères	
	– chiffres	`0 1 2 3 4 5 6 7 8 9`
	– signes de ponctuation	`: . ; , ? ! ' "`
	– signes mathématiques	`+ - * / = < >`
	– autres signes	`[ ] ( ) | @ ‘`
13	caractère actif	~
14	caractère de commentaire	%
15	caractère invalide	*DEL*

Dans toute la suite du livre, sauf indication contraire, lorsqu'il est dit par abus de langage « lettre » ou bien « accolade ouvrante », il est sous-entendu « caractère de catcode 11 » ou « caractère de catcode 1 ».

Comme on le voit, certaines catégories sont très peuplées (celles dont le numéro est 11 ou 12) alors que d'autres ne contiennent qu'un seul élément. Ceci s'explique par le fait que seules les catégories de numéro 10, 11 et 12 comportent des caractères *affichables* pour TEX. Les autres ne comportent que des caractères qui commandent une action à TEX, par exemple de passer en mode mathématique pour le caractère « $ ». Comme il n'y a pas de raison pour que plusieurs caractères revêtent cette propriété, il est logique de ne trouver qu'un seul caractère dans la catégorie n°3. Le même raisonnement s'applique aux autres catégories ne comportant qu'un seul caractère.

L'espace, bien qu'affichable, se trouve dans une catégorie à part, car il obéit à des règles différentes que celles des autres caractères affichables, notamment lors de la lecture du code source :

3 - Règle

Lorsque plusieurs espaces (ou plus généralement « caractères de catcode 10 ») se suivent dans le code source, TEX ne prend en compte que le premier d'entre eux et ignore les autres.

Si une ligne de code source commence par des espaces, ceux-ci sont ignorés.
Plain-TEX et LATEX assignent au caractère de tabulation (*HT*) le catcode de 10 ce qui signifie que ce caractère revêt toutes les propriétés de l'espace.

Pour fixer les idées, supposons que TEX ait à lire le code suivant :

```
\hbox{Et $x$ vaut  3,5.}
```

Au fur et à mesure de sa lecture, il attribuera à chaque caractère le code de catégorie en vigueur au moment où il le lit et agira en conséquence. Ce code de catégorie est attribué de façon *définitive* et ne peut pas être changé ensuite ; il s'agit d'un mécanisme incontournable sur lequel il faut insister :

4 - Règle
Dès que TEX lit un caractère, il lui affecte de façon inaltérable un code de catégorie.

Si lors de la lecture, les catcodes en vigueur sont ceux exposés dans le tableau ci-dessus, voici sous chaque caractère, le catcode qui lui sera affecté :

\	h	b	o	x	{	E	t	␣	$	x	$	␣	v	a	u	t	␣	3	,	5	.	}
0	11	11	11	11	1	11	11	10	3	11	3	10	11	11	11	11	10	12	12	12	12	2

Un caractère peut être tapé sous forme habituelle comme lorsqu'on tape un « a », mais on peut aussi utiliser « ^^ ». En effet, si TEX voit deux caractères *identiques* dont le code de catégorie est 7 (celui du caractère de mise en exposant) et si ces deux caractères sont suivis d'un caractère dont le code – appelons-le c – est compris entre 0 et 63 inclus, alors TEX lit le tout comme étant le caractère dont le code est $c + 64$. Par exemple, « ^^: » est le caractère « z ». Il est rigoureusement équivalent de taper « z » ou « ^^: » et ce, quel que soit l'endroit où l'on se trouve dans le code source.

Symétriquement, si le code de caractère c est compris entre 64 et 127 inclus, TEX remplace le tout par le caractère de code $c - 64$. Et donc, il est équivalent de taper « ^^z » ou « : ».

Ce qui est écrit dans les deux paragraphes précédents nous conduit donc à cette règle :

5 - Règle
Lorsqu'on écrit « ^^⟨*car*⟩ » dans le code source, tout se passe donc comme si TEX voyait le caractère se trouvant 4 lignes plus haut ou plus bas dans la table ASCII, selon que ⟨*car*⟩ se trouve dans la partie basse ou haute de cette table.
Si les *deux* caractères qui suivent ^^ forment un nombre hexadécimal à deux chiffres minuscules pris parmi « `0123456789abcdef` », alors TEX remplace ces 4 caractères par le caractère ayant le code hexadécimal spécifié. Avec cette méthode, on peut donc facilement accéder à n'importe quel caractère. Le caractère « z » peut aussi s'écrire « `^^7a` ».

Ainsi, le caractère « t » étant apparié via ^^ avec un caractère affichable (en l'occurrence « 4 » qui se trouve 4 lignes plus haut dans la table ASCII), « t » peut donc être tapé dans le code de 3 façons équivalentes : « `t` » ou « `^^4` » ou « `^^74` ». Si l'on voulait rendre le code mystérieux ou difficilement lisible, voici comment on pourrait écrire le mot « tex » :

– ^^4^^%^^8
– ^^74^^65^^78

La seule petite ambigüité qui subsiste dans cette belle mécanique est que l'on ne peut pas écrire un « t » sous la forme « ^^4 » puis le faire suivre du caractère « 5 ». En effet, cela formerait la suite de caractères « ^^45 » que TEX interprèterait comme étant le caractère de code hexadécimal 45_h qui est « E ».

Au-delà de l'originalité de cette méthode alternative pour écrire des caractères dans le code source, ce qu'il faut en retenir est qu'elle peut être très utile pour écrire des caractères non affichables qui de toute façon, seraient difficilement accessibles au clavier. Ainsi, le caractère de tabulation *HT* s'écrit « ^^I », le retour charriot *CR* « ^^M », etc.

On peut aussi *afficher* un caractère avec la primitive \char suivie du numéro du code de caractère. Ainsi, \char97 affiche un « a » mais au contraire de la méthode avec ^^, \char97 ne peut pas être utilisé dans le code source à la place d'une lettre. Autant « \p^^61r » et « \par » sont équivalents, autant il n'est pas question d'écrire « \p\char65r » à la place de \par dans le code source, car ceci serait interprété comme la commande \p, suivie de \char65, suivi de « r ».

Enfin, il faut définir ce qu'est une « ligne de code source ». Pour TEX, une ligne de code est constituée par les caractères qui se trouvent avant la prochaine « marque de fin de ligne ». Cette marque est une convention qui dépend du système d'exploitation :

– pour Windows, une marque de fin de ligne est constituée des octets $0D_h0A_h$, c'est-à-dire de *CR* suivi de *LF* (noté « CRLF ») ;
– pour GNU-linux, cette marque est l'octet $0A_h$ (LF) ;
– enfin, pour MAC OS X, cette marque est l'octet $0D_h$ (CR).

L'exécutable pdftex considère comme marque de fin de ligne les séquences suivantes : LF, CR, CRLF et LFCR. Les espaces qui précèdent la marque de fin de ligne sont supprimés en même temps que cette marque et TEX insère à la place un caractère spécial dont le code de caractère est contenu dans la primitive \endlinechar. Habituellement, ce code vaut 13, celui du caractère retour charriot.

6 - Règle

Dans la ligne courante, si un caractère de catcode 14 (caractère de commentaire, généralement %) ou 5 (retour charriot, généralement ^^M) apparait, alors, ce caractère ainsi que tout ce qui va jusqu'à la fin de la ligne est ignoré, y compris le caractère de code \endlinechar inséré à la fin de la ligne.

Les caractères de catcode 5 obéissent à une règle particulière :

7 - Règle

Un caractère de catcode 5 est interprété comme un espace.

Deux caractères de catcode 5 consécutifs sont transformés en la séquence de contrôle \par .

Cela signifie qu'un retour charriot est vu comme un espace, sauf s'il est précédé du caractère % et que deux retours charriots consécutifs (symbolisés dans le code source par une ligne vide) seront équivalents à \par.

On peut localement modifier le caractère de code \endlinechar pour obtenir des effets spéciaux en tenant compte de la règle suivante :

8 - Règle

Si \endlinechar est négatif ou supérieur à 255, aucun caractère n'est inséré aux fins de lignes.

Voici une illustration de cette règle où l'on teste le comportement normal, puis on met \endlinechar à −1 et enfin à 88 (qui est le code de caractère de « X ») :

Code nº I-3

```
%%% comportement normal %%%
a) comportement normal :
Ligne 1
Ligne 2

Ligne 3
\medbreak
b) Aucun caractère de fin de ligne :
%%% aucune insertion en fin de ligne %%%
\endlinechar=-1
Ligne 1
Ligne 2

Ligne 3\endlinechar13

\medbreak
c) "X" comme caractère de fin de ligne :
%%% "X" est inséré à chaque fin de ligne
\endlinechar=88
Ligne 1
Ligne 2

Ligne 3
```

a) comportement normal : Ligne 1 Ligne 2
Ligne 3

b) Aucun caractère de fin de ligne : Ligne 1Ligne 2Ligne 3

c) "X" comme caractère de fin de ligne : Ligne 1XLigne 2XXLigne 3X

On peut notamment remarquer que lorsque \endlinechar est négatif, tout se passe comme si les lignes se terminaient toutes par le caractère de commentaire %, de catcode 14.

Chapitre 2

Octets dont le bit de poids fort est 1

Venons-en maintenant à la partie de la table des caractères qui se situe après le numéro 128, c'est-à-dire dont la représentation en base 2 est *1xxxxxxx* et dont le bit le le plus à gauche, celui qui a le plus de « poids [1] », vaut 1. Cette partie de la table n'est pas dans la convention ASCII et donc, dépend de la convention que l'on veut se donner. Pour nous en France, deux de ces conventions sont utilisées et dépendent du codage avec lequel a été enregistré le code source. Comme on l'a dit, cet encodage doit être un surensemble de l'ASCII qui est commun à tous les encodages.

Historiquement, plusieurs façons de compléter la table ASCII ont vu le jour, chacune donnant naissance à un encodage à part entière. Aucune harmonisation n'a jamais été faite et plusieurs encodages sur 8 bits cohabitent encore à l'heure actuelle. Les plus connus pour les langues occidentales sont :

- `ansinew` a émergé comme encodage des systèmes d'exploitation Windows ;
- `applemac` a été choisi dans la sphère de la marque Apple jusqu'à un passé récent ;

1. Dans l'écriture moderne des nombres, dite « de position », les chiffres ont une influence liée à leur position dans le nombre. En écriture décimale, nous avons pris comme convention qu'un nombre, par exemple écrit avec 3 chiffres « $\overline{abc}$ », est égal à la somme $a \times 100 + b \times 10 + c \times 1$. Le coefficient d'un chiffre (ou son « poids ») est une puissance de 10 dont l'exposant est d'autant plus grand que la place du chiffre est à gauche : 10^2, 10^1 et 10^0. On dit que le chiffre a a un « poids » fort puisque plus important que celui des chiffres b ou c.

En binaire, c'est le même principe sauf que les poids sont les puissances de 2. Si un octet s'écrit $\overline{abcdefgh}$, sa valeur en base 10 est $a \times 2^7 + b \times 2^6 + \cdots + g \times 2^1 + h \times 2^0$. Le bit a est le bit de poids fort tandis que h est celui de poids faible.

– `latin1` a été utilisé dans le monde Unix/Linux.

Selon ces trois encodages, le caractère « é » est respectivement placé à l'octet $E9_h$, $8E_h$ et $E9_h$.

Intéressons-nous à l'encodage « `latin1` » ou « ISO 8859-1 ». Cet encodage a été créé pour être utilisé avec les langues européennes. Par exemple, l'octet 224 (ou $E0_h$ en hexadécimal) code la lettre « à ». On dispose ainsi de tous les caractères nécessaires pour la langue française [2]. Il faut noter que l'extension pour LaTeX « `inputenc` » chargée avec l'option « `latin1` » rend plusieurs caractères actifs (et donc met leur catcode à 13) lorsque l'encodage est `latin1` . Le caractère « à » en fait partie et donc, ce caractère se comporte comme une commande : `inputenc` lui fait exécuter un code qui au final, imprime le caractère « à ». Il est immédiat que du fait qu'il est actif, le caractère « à » n'appartient pas à la catégorie des lettres.

Voici la table de l'encodage `latin1`. Seuls les octets supérieurs ou égaux à 128 y figurent puisque ceux qui sont inférieurs à 128 sont ceux de la table ASCII :

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
8x	*PAD* 128	*HOP* 129	*BPH* 130	*NBH* 131	*IND* 132	*NEL* 133	*SSA* 134	*ESA* 135	*HTS* 136	*HTJ* 137	*VTS* 138	*PLD* 139	*PLU* 140	*RI* 141	*SS2* 142	*SS3* 143
9x	*DCS* 144	*PU1* 145	*PU2* 146	*STS* 147	*CCH* 148	*MW* 149	*SPA* 150	*EPA* 151	*SOS* 152	*SGCI* 153	*SCI* 154	*CSI* 155	*ST* 156	*OSC* 157	*PM* 158	*APC* 159
Ax	NBSP 160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	¦ 166	§ 167	¨ 168	© 169	ª 170	« 171	¬ 172	- 173	® 174	¯ 175
Bx	° 176	± 177	2 178	3 179	´ 180	181	¶ 182	· 183	¸ 184	1 185	º 186	» 187	¼ 188	½ 189	¾ 190	¿ 191
Cx	À 192	Á 193	*A* 194	Ã 195	Ä 196	Å 197	Æ 198	Ç 199	È 200	É 201	*E* 202	Ë 203	Ì 204	Í 205	*I* 206	Ï 207
Dx	Ð 208	Ñ 209	Ò 210	Ó 211	*O* 212	Õ 213	Ö 214	× 215	Ø 216	Ù 217	Ú 218	*U* 219	Ü 220	Ý 221	Þ 222	ß 223
Ex	à 224	á 225	*a* 226	ã 227	ä 228	å 229	æ 230	ç 231	è 232	é 233	*e* 234	ë 235	ì 236	í 237	*ı* 238	ï 239
Fx	ð 240	ñ 241	ò 242	ó 243	*o* 244	õ 245	ö 246	÷ 247	ø 248	ù 249	ú 250	*u* 251	ü 252	ý 253	þ 254	ÿ 255

Ici encore, cette partie de la table comporte des caractères non affichables. Ils se situent entre les codes 80_h et $9F_h$ inclus auxquels il faut rajouter le caractère de code AD_h. Ceux qui se trouvent dans des cases grisées sont affichables, mais ils doivent être utilisés en mode mathématique.

Bien sûr, certains de ces caractères ne sont pas directement accessibles au clavier. En fait, les caractères accessibles au clavier dépendent de la configuration de chacun. Par exemple, sur *mon* clavier, avec *mon* système d'exploitation et le pilote correspondant au modèle du clavier, avec le mappage du clavier sélectionné dans les options du système d'exploitation, je peux accéder au caractère « æ » avec la combinaison de touches AltGr+A. Pour le caractère « ¶ », c'est avec AltGr+R. Et ainsi de suite… En revanche, certains caractères dans la table ci-dessus me sont inaccessibles, sauf à les taper via ^^.

L'encodage « `UTF8` [3] » permet de coder beaucoup plus que 256 caractères avec l'inévitable conséquence que certains d'entre eux seront codés sur plusieurs octets. Le principe de cet encodage est assez simple et on peut le décrire ici :

2. À l'exception notable de « œ », « Œ » et « Ÿ ». Il manque également quelques caractères pour d'autres langues.

3. L'acronyme `UTF8` signifie « Unicode Transformation Format 8 bits ».

- si un caractère a un code inférieur à 128, alors il est codé par le même octet que dans le encodage ASCII ;
- sinon, il est codé sur plusieurs octets : le nombre de bits de poids fort non nuls (au maximum 4) du premier octet indique le nombre d'octets pris au total pour l'encodage du caractère. Par exemple, si un octet vaut *1110xxxx* en binaire, les 3 bits de poids fort suivis d'un 0 indiquent que ce caractère est codé sur 3 octets, les 2 octets restants ayant obligatoirement les bits de poids fort *10*, c'est-à-dire qu'ils sont de la forme *10xxxxxx*.

  Finalement, ce caractère sera donc codé sur 3 octets ayant cette forme : *1110xxxx 10xxxxxx 10xxxxxx*. Il y a donc 16 bits non contraints sur 24. Le « code propre » du caractère sera constitué des 16 bits *x*.

Si l'on s'intéresse par exemple au caractère « à », celui-ci est codé en `UTF8` par les 2 octets $C3_h$ $A0_h$. Écrits en binaire, on obtient ***110***00011 ***10***100000 où les bits contraints sont écrits en gras. Il en résulte que le code propre du caractère « à » est *00011100000*. Si on élimine les zéros inutiles de gauche, on obtient *11100000* qui est l'octet $E0_h$, celui que l'on retrouve dans la table de l'encodage `latin1` pour la lettre « à ». En effet, en `UTF8`, les caractères ayant des codes propres compris entre 128 et 255 *doivent* être les mêmes que les caractères qui ont ces codes dans la table de `latin1`. Enfin, le symbole « € » est codé sur 3 octets en UFT8 : $E2_h$ 82_h AC_h qui s'écrit en binaire ***1110***0010 ***10***000010 ***10***101100 et qui donne un code propre de *10000010101100* soit $20AC_h$.

Pour TEX, le principe est le même qu'avec l'encodage `latin1`, il faut rendre actifs tous les octets qui ont une écriture binaire *110xxxxx* ou *1110xxxx* ou *11110xxx* et dont les bits non contraints « *x* » correspondent au début du code des caractères que TEX va prendre en charge. Ensuite, ces caractères actifs vont lire le (ou les) octet(s) suivant(s) et traduire le tout de façon à faire imprimer le caractère voulu.

Ce qu'il faut retenir est que, aussi bien en `latin1` qu'en `UTF8`, les lettres accentuées et les caractères ne figurant pas dans la table ASCII ne sont pas des octets dont le code de catégorie est 11 et n'appartiennent donc pas à la catégorie des lettres. De plus, pour ceux qui utilisent l'encodage `UTF8` qui tend à devenir la norme, il ne faut pas perdre de vue que lorsqu'on écrit la lettre « à » dans le code, nous ne voyons qu'un seul caractère, mais celui-ci est stocké sur *deux* octets dans le fichier et donc, TEX voit *deux* choses. Il n'y a donc pas équivalence entre « octet » et « caractère » et dans tous les cas, TEX ne lit pas le code caractère par caractère, mais octet par octet.

Cette dernière affirmation n'est vraie que pour les trois moteurs TEX (celui écrit par Donald Knuth), ε-TEX et pdfTEX. Outre ces « ancêtres », encore majoritairement utilisés malgré leur âge, il existe désormais des moteurs plus récents qui gèrent nativement l'encodage `UTF8` : XƎTEX et luaTEX. Ces deux moteurs, entre autres possibilités, lisent le code source caractère `UTF8` par caractère `UTF8` et il peut donc arriver que plusieurs octets soient lus en un coup pour former une entité unique. Malgré le progrès que cela constitue, on considèrera dans ce livre que l'on utilise un moteur fonctionnant octet par octet.

Chapitre 3

Codes de catégorie

Le sujet est vaste et technique puisqu'il touche à l'un des mécanismes les plus intimes de TEX. Nous nous contenterons pour l'instant de voir l'essentiel, quitte à y revenir plus loin au fur et à mesure de nos besoins. Rappelons que les « catégories » de TEX sont au nombre de 16 et regroupent des caractères ayant des propriétés communes aux yeux de TEX. Le code de catégorie d'un octet lu est la première chose que TEX examine, avant même le nombre correspondant à l'octet lui-même puisque le code de catégorie détermine la façon dont TEX doit réagir face à cet octet.

Tout d'abord, pour accéder au code de catégorie d'un caractère, il faut utiliser la primitive `\catcode` suivie du *code de caractère*. L'ensemble constitue une représentation interne de l'entier étant le code de catégorie du caractère concerné. La commande `\number` convertit la représentation interne d'un nombre en la représentation affichable habituelle, c'est-à-dire en base 10 et en chiffres arabes. Par exemple, si on veut afficher le code de catégorie de « $ » dont le code de caractère est 36, on va écrire « `\number\catcode36` » et on obtiendra « 3 ».

On peut trouver gênant de mettre explicitement le code du caractère après la primitive `\catcode`. En effet, cela oblige à aller le chercher dans la table des caractères ASCII puis le convertir en décimal. Cette dernière étape, la conversion hexadécimal vers décimal, n'a pas lieu d'être faite car TEX comprend nativement les nombres hexadécimaux [1]. Pour indiquer qu'un nombre est écrit en base 16, il faut le faire précéder du caractère « " » et pour traduire le tout en nombre en écriture arabe en base 10, il faut mettre devant l'ensemble la primitive `\number` :

1. Où les lettres qui le composent *doivent* être majuscules.

Code n° I-4

```
a) \number"1A \qquad b) \number"AB3FE \qquad c) \number"78
```

a) 26 b) 701438 c) 120

Pour obtenir le code de catégorie du caractère « $ » dont le code de caractère est 24_h, on peut donc écrire `\number\catcode"24`.

Dans le même ordre d'idée et bien que cela soit assez rarement utile, il faut savoir que TEX comprend aussi les nombres en notation *octale*, c'est-à-dire écrits en base 8. Pour indiquer qu'un nombre est écrit en base 8, il faut le faire précéder de l'apostrophe « ' » :

Code n° I-5

```
a) \number'15 \qquad b) \number'674 \qquad c) \number'46
```

a) 13 b) 444 c) 38

Malgré ce côté pratique, il faut toujours aller voir la table ASCII pour chercher le code de caractère. Heureusement, on peut facilement convertir un caractère en son code écrit en base 10. Pour cela, s'agissant des caractères de catcode 11 ou 12, il faut faire précéder le caractère de l'apostrophe inverse ` ce qui donnerait « `\number‘a` » pour afficher le code de caractère de « a ». Un moyen encore plus sûr qui fonctionne pour *tous* les caractères, quels que soient leurs catcodes, est de faire précéder le caractère de l'apostrophe inverse *et* du caractère d'échappement comme dans cet exemple :

Code n° I-6

```
a) \number‘\a \quad %code de caractère de "a"
b) \number‘\\ \quad % code de caractère de "\"
c) \number‘\$ \quad % code de caractère de "$"
d) \number‘\  \quad % code de caractère de l'espace
e) \number‘\5 \quad % code de caractère de "5"
f) \number‘\{ \quad % code de caractère de l'accolade ouvrante
g) \number‘\}       % code de caractère de accolade fermante
```

a) 97 b) 92 c) 36 d) 32 e) 53 f) 123 g) 125

Il devient facile d'afficher le code de catégorie de n'importe quel caractère :

Code n° I-7

```
a) \number\catcode‘\a \quad %code de catégorie de "a"
b) \number\catcode‘\\ \quad % code de catégorie de "\"
c) \number\catcode‘\$ \quad % code de catégorie de "$"
d) \number\catcode‘\  \quad % code de catégorie de l'espace
e) \number\catcode‘\5 \quad % code de catégorie de "5"
f) \number\catcode‘\{ \quad % code de catégorie de l'accolade ouvrante
g) \number\catcode‘\}       % code de catégorie de accolade fermante
```

a) 11 b) 0 c) 3 d) 10 e) 12 f) 1 g) 2

Afficher un code de catégorie n'est pas réellement utile. Ce qui l'est plus est de *modifier* le code de catégorie d'un caractère, nous verrons dans quels buts plus loin. Pour ce faire, on écrit :

`\catcode‘\`⟨*caractère*⟩`=`⟨*nombre*⟩

où ⟨*nombre*⟩ est un entier de 0 à 15. Après que TEX a exécuté ce code, à chaque fois que TEX lira le ⟨*caractère*⟩, il lui assignera le code de catégorie égal au ⟨*nombre*⟩. Supposons par exemple que nous souhaitions que le caractère « W » ne soit plus dans la catégorie des lettres, mais revête le code de catégorie de 3 pour utiliser indifféremment « W » ou « $ » pour basculer en mode mathématique. Voici comment procéder :

Code n° I-8

```
Ici, W est une lettre...\par
\catcode`\W=3 % W devient le signe de bascule en mode math
Wx+y=3W\par
$a+b=cW\par
W2^3=8$\par
\catcode`\W=11 % W redevient une lettre
De nouveau, W est une lettre...
```

Ici, W est une lettre...
$x + y = 3$
$a + b = c$
$2^3 = 8$
De nouveau, W est une lettre...

Il est important de restaurer le code de catégorie initial à la fin de la zone où l'on souhaite avoir le comportement voulu sous peine de s'exposer à des erreurs de compilation ou des effets indésirables plus tard.

Symétriquement, on aurait aussi bien pu assigner le code de catégorie 11 (ou 12) au caractère « $ » de façon à pouvoir l'afficher comme on le fait d'une lettre ou d'un caractère affichable :

Code n° I-9

```
$3+4=7$ \par % $ est la bascule math
\catcode`\$=12 % $ devient un caractère affichable
$ s'affiche sans problème : $\par
\catcode`\$=3 % $ redevient la bascule math
$4+3=7$
```

3 + 4 = 7
$ s'affiche sans problème : $
4 + 3 = 7

■ Exercice 2

Inventer un procédé pour que, dans une zone, seuls les premiers mots de chaque ligne soient affichés.

□ Solution

Il suffit de modifier le code de catégorie de l'espace pour qu'il devienne 5. D'après la règle vue à la page 29, dans une ligne, tout ce qui se trouve après un caractère de catcode 5 est ignoré :

Code n° I-10

```
Voici les premiers mots de chaque ligne :

\catcode`\ =5 % l'espace est déormais de catcode 5
Cette première ligne sera tronquée...

La deuxième aussi !
```

```
7
8 Et la dernière également.
9 \catcode`\ =10 % l'espace reprend son catcode
10
11 Le comportement normal est restauré.
```

Voici les premiers mots de chaque ligne :
Cette
La
Et
Le comportement normal est restauré.

■

Il faut retenir que les codes de catégorie attachés à chaque caractère de la table des catcodes de la page 27 ne sont que des valeurs *par défaut* qui peuvent être facilement modifiées comme on l'a vu dans les exemples précédents. On peut donc parler de « régime de catcode », c'est-à-dire qu'à chaque moment, chacun des 256 octets aura un catcode qui lui sera attribué. En règle générale, si l'on modifie le catcode d'un octet, il est sage de restaurer les codes de catégorie de la table des catcodes au-delà de la zone où l'on veut faire des modifications. En effet, il est admis que les catcodes « naturels » de cette table doivent rester la règle.

Deuxième partie

Commandes

Sommaire

Dans cette partie, nous allons nous intéresser à ce qui fait l'essentiel du code lorsqu'on programme, les commandes. Contrairement à d'autres langages de programmation où leur comportement est assez intuitif, elles revêtent avec TEX des propriétés originales. Par leur intermédiaire, nous découvrirons l'essentiel de ce qu'il faut savoir pour commencer à programmer c'est-à-dire la façon intrinsèque qu'a TEX de fonctionner et comment contrôler ce fonctionnement. Après ces prérequis, nous pourrons alors aborder la partie suivante où nous commencerons les premiers exercices de programmation.

Chapitre 1

Qu'est-ce qu'une commande ?

Lorsqu'on utilise TEX, les séquences de contrôle sont des mots constitués de lettres (caractères de catcode 11) qui débutent par le caractère d'échappement « \ ». Ce caractère spécial peut être changé pour en utiliser un autre, mais cette manœuvre ne sera pas décrite pour l'instant. Ainsi, lorsque TEX lit le code que l'on a tapé, toujours linéairement de gauche à droite, forme une séquence de contrôle à l'aide des lettres, majuscules ou minuscules, qui suivent le caractère d'échappement « \ », la casse étant prise en compte. TEX considère que le nom de la séquence de contrôle se termine au dernier caractère qui est une lettre . Une fois que ce nom est construit, il devient une entité unitaire indivisible, une « unité lexicale » ou « *token* » au même titre qu'un octet dans le code si celui-ci ne sert pas à écrire une séquence de contrôle.

9 - Règle
Une séquence de contrôle commence par le caractère d'échappement de catcode 0 qui est habituellement « \ » et se termine à la fin de la plus longue série de lettres (caractères de catcode 11) qui le suit.

Les séquences de contrôle sont de deux types :

- les « commandes », que l'on appellera aussi « macros », ont été définies soit par l'utilisateur, par un format ou par une extension. Ces commandes ont un texte de remplacement, c'est le code TEX qui les définit, ou si l'on préfère, le code TEX qu'elles « contiennent ». Ce texte de remplacement peut évidemment contenir d'autres séquences de contrôle ;
- les primitives, qui sont codées en dur dans le programme binaire `tex`, et qui représentent en fait le « vocabulaire » de base avec lequel on doit se débrouiller

pour faire tout le reste. Les primitives n'ont pas de texte de remplacement et sont destinées à être exécutées de façon binaire par le programme `tex`.

10 - Règle

Voici des règles concernant les espaces et les séquences de contrôle :

1. dans le code qui est tapé, tout espace qui suit une unité lexicale de type séquence de contrôle est ignoré ;
2. si plusieurs espaces se suivent *dans le code*, seul le premier est pris en compte et les autres sont ignorés (ceci est un rappel de la règle page 27) ;
3. il découle des deux premiers points que, quel que soit le nombre d'espaces qui suivent une séquence de contrôle, ceux-ci sont ignorés.

Lorsqu'un utilisateur définit une commande, il lui donne un *texte de remplacement* qui sera utilisé à la place de cette commande lorsqu'elle sera « exécutée » par TEX. La primitive qui permet de définir ce texte de remplacement est « `\def` ». Elle doit être suivie du nom de la commande et du texte de remplacement entre accolades :

```
\def\foo{Bonjour}
```

Après cette définition, à chaque fois que TEX « exécutera » le token `\foo` dans le code source, il le remplacera dans sa mémoire par son texte de remplacement qui est « Bonjour » et continuera à lire le code obtenu en tenant compte de ce remplacement. Cet endroit dans sa mémoire s'appelle « pile d'entrée ». Pour écrire un espace après `\foo` qui ne soit pas ignoré, il faut trouver un moyen, invisible à l'affichage, pour que l'espace affiché après la macro ne lui soit pas consécutif. On peut envelopper la macro dans des accolades ou la faire suivre de `{}` de façon à ce que l'espace se situant *après* l'accolade fermante ne soit pas ignoré comme il le serait après une séquence de contrôle. On peut également mettre après `\foo` la macro `\space` dont le texte de remplacement est un espace. Dans ce cas, le dernier point de la règle précédente ne s'applique pas, car ce qui suit `\foo` n'est pas un espace mais une séquence de contrôle, et peu importe son texte de remplacement.

Code n° II-11

```
\def\foo{Bonjour}% définit le texte de remplacement de \foo
a) \foo Alice.\qquad% espace ignoré
b) {\foo} Bob.\qquad% espace non ignoré
c) \foo{} Chris.\qquad% espace non ignoré
d) \foo\space Daniel.% \space est remplacé par un espace
```

a) BonjourAlice. b) Bonjour Bob. c) Bonjour Chris. d) Bonjour Daniel.

11 - Règle

Lorsque TEX assigne un texte de remplacement à une macro avec `\def`, ce texte de remplacement n'est pas exécuté, il est juste converti en tokens et rangé quelque part dans la mémoire de TEX pour être ressorti plus tard pour venir en replacement de la macro, et éventuellement être exécuté à ce moment.

Le texte de remplacement est très peu *analysé* lorsqu'il est stocké avec `\def`, ce qui veut dire que si une erreur est contenue dans le texte de remplacement, elle ne sera

pas détectée au moment où \def agit mais plus tard lors du remplacement et de l'exécution proprement dite.

Une des rares choses que TEX vérifie dans le texte de remplacement d'une macro est qu'un caractère de catcode 15 n'y figure pas (nous verrons plus loin les autres vérifications que TEX effectue).

1.1. Accolades et groupes

Les accolades des lignes 3 et 4 délimitent des « groupes [1] ». Un groupe est une zone où les modifications effectuées et où les définitions faites restent *locales* à ce groupe et sont détruites lors de sa fermeture pour revenir à l'état antérieur à l'ouverture du groupe. En revanche, les accolades obligatoires utilisées avec la primitive \def de la ligne 1 ne délimitent pas un groupe. Elles délimitent la portée du texte de remplacement. On peut donc considérer qu'il y a deux types d'accolades. D'une part celles qui sont obligatoires et font partie de la syntaxe d'une primitive comme celles de la primitive \def qui délimitent le texte de remplacement et d'autre part celles, non obligatoires, qui servent de frontières à un groupe.

Un groupe semi-simple [2] est compris entre les instants où TEX exécute les primitives \begingroup et \endgroup. Comme dans les groupes, les modifications et les définitions dans un groupe semi-simple sont locales et sont restaurées à leur état antérieur à sa fermeture.

12 - Règle

On peut créer des zones où les modifications faites aux macros et autres paramètres de TEX sont locales. Ces zones portent le nom de groupes.

Un groupe est délimité :

- soit par une accolade ouvrante et une accolade fermante auquel cas le groupe est qualifié de « simple » ;
- soit par les primitives \begingroup et \endgroup et dans ce cas, le groupe est dit « semi-simple ».

Il est entendu qu'un groupe ouvert avec une accolade ne peut être fermé qu'avec une accolade et il en est de même avec \begingroup et \endgroup.

À l'intérieur d'un groupe, les assignations sont locales à ce groupe et sont restaurées à leur état antérieur lors de la fermeture du groupe. Pour qu'une assignation soit *globale*, c'est-à-dire pour qu'elle survive à la fermeture du groupe, il faut faire précéder la commande d'assignation de la primitive \global.

Les groupes délimités par accolades et les groupes semi-simples peuvent être emboités, mais *ne doivent pas* se chevaucher.

1. On dit aussi parfois « groupe simple ».

2. Le nom « groupe simple » étant déjà pris pour les groupes entre accolades, il y a fort à parier que D. Knuth, par jeu et par affinité pour les mathématiques, ait baptisé « groupe semi-simple » un groupe délimité par \begingroup et \endgroup.

■ Exercice 3

Puisqu'un groupe délimité par accolades et un groupe semi-simple ont les mêmes fonctionnalités, pourquoi existe-t-il deux façons différentes de délimiter un groupe ?

□ Solution

Tout d'abord, abondance de biens ne nuit pas !

Ensuite, il y a des cas où une accolade ne peut pas délimiter un début ou une fin groupe, c'est dans le texte de remplacement d'une macro. Par exemple, si l'on voulait qu'une macro `\startbold` ouvre un groupe et écrive en gras et que la macro `\stopbold` stoppe ce fonctionnement, on serait obligé d'utiliser `\begingroup` et `\endgroup` :

Code n° II-12

```
1 \def\startbold{\begingroup \bf}
2 \def\stopbold{\endgroup}
3 Voici \startbold du texte en gras\stopbold{} et la suite.
```

Voici **du texte en gras** et la suite.

En effet, cela constituerait une erreur que d'écrire :

```
\def\startbold{{\bf}
\def\stopbold{}}
```

car le texte de remplacement de `\startbold` irait jusqu'à la prochaine accolade équilibrée et serait

```
{\bf} \def\stopbold{}
```

Il n'y aurait pas d'erreur de compilation, mais la portée de `\bf` serait nulle puisque cette macro est seule dans son groupe et `\stopbold`, ayant un texte de remplacement vide, n'aurait aucune action. ■

■ Exercice 4

Décrire ce qui se passe exactement et quel va être l'affichage si l'on définit une macro `\foo` dans ce contexte :

```
\begingroup
\def\foo{Hello \endgroup world !}
\foo
```

□ Solution

Déjà, la définition sera *locale* au groupe semi-simple initié par `\begingroup` et la macro `\foo` sera détruite dès qu'un `\endgroup` sera exécuté pour revenir à ce qu'elle était avant le groupe semi-simple (éventuellement indéfinie). Le `\endgroup` de la ligne 2 n'est pas « exécuté » par TEX puisque ce `\endgroup` est stocké dans le texte de remplacement de la macro `\foo`. Il ne ferme donc pas le groupe semi-simple lorsque TEX exécute la ligne n° 2 du code.

Lorsque TEX rencontre `\foo` à la 3e ligne du code, il se trouve toujours dans le groupe semi-simple ouvert par `\begingroup` et la définition faite la ligne au-dessus est toujours en vigueur. TEX remplace donc `\foo` par son texte de remplacement, ce remplacement se faisant dans la pile d'entrée de TEX :

```
Hello \endgroup world !
```

TEX va continuer à lire ce qu'il y a dans sa pile et va donc afficher `Hello␣` puis, en rencontrant `\endgroup`, détruira la définition locale de `\foo` enfin continuera à lire ce qui est dans sa pile et affichera : `world !`.

On voit donc que dans ce cas, la macro `\foo` porte dans son texte de remplacement la primitive `\endgroup` qui provoquera son autodestruction ! ■

13 - Règle

La primitive \aftergroup⟨*token*⟩ permet de stocker un ⟨*token*⟩ dans une pile spéciale de la mémoire de TEXpour que ce token soit lu et exécuté juste après être sorti du groupe courant, que ce groupe soit simple ou semi-simple.

Si plusieurs commandes \aftergroup sont rencontrées dans un même groupe, les tokens mis en mémoire seront lus dans l'ordre où ils ont été définis. Ainsi, écrire

\aftergroup⟨*x*⟩\aftergroup⟨*y*⟩

se traduira par « ⟨*x*⟩⟨*y*⟩ » après la fermeture du groupe.

Code n° II-13

```
\def\foo{foo}
\begingroup A\aftergroup\foo B\endgroup\par
{A\aftergroup X\aftergroup\foo B}
```

ABfoo
ABXfoo

La primitive \aftergroup trouve une utilité lorsque la composition en italique est demandée dans un groupe (qui en délimite la portée). En effet, pour éviter que le dernier caractère en italique (qui est donc penché vers la droite) ne soit trop près du caractère du mot suivant (qui est vertical), il convient d'insérer à la fin du groupe une correction d'italique, surtout si les deux caractères sont des lettres hautes comme « l », « f » ou une parenthèse fermante. Cette correction, effectuée en TEX par la primitive \/, augmente légèrement l'espace entre ces deux caractères. La différence est très fine, mais elle existe :

Code n° II-14

```
1) (un {\it cheval})\par% sans correction d'italique
2) (un {\it\aftergroup\/cheval})\par% avec correction d'italique
% on peut définir une macro \itacorr qui effectue automatiquement la correction
\def\itacorr{\it\aftergroup\/}
3) (un {\itacorr cheval})% avec correction d'italique
```

1) (un *cheval*)
2) (un *cheval*)
3) (un *cheval*)

1.2. Primitive \let

Pour copier le texte de remplacement d'une macro vers une autre, on utilise la primitive \let qui prend comme arguments deux unités lexicales de type séquence de contrôle. Par exemple, écrire \let\foo=\bar copiera le texte de remplacement de la macro \bar vers \foo en écrasant celui de cette dernière si elle était déjà définie. Le signe « = » entre les deux séquences de contrôle est facultatif et sera souvent omis désormais [3]. Une fois cette copie faite, peu importe ce que sera le futur de \bar, la copie a lieu à l'instant où \let est rencontré et si \bar est redéfinie par la suite, cela n'affectera pas le texte de remplacement de \foo.

3. En réalité, le signe égal peut aussi être suivi d'*un* espace facultatif.

Code n° II-15

```
\def\bar{Je suis bar.}
\let\foo\bar % \foo devient égal à \bar
\def\bar{ABC}% \bar est redéfinie
\foo\par% affiche "Je suis bar"
\bar% affiche "ABC"
```

Je suis bar.
ABC

Lorsqu'on écrit `\let\foo=\bar`, si `\bar` est une primitive, alors `\foo` devient un *alias* pour cette primitive et pour TEX, elles sont identiques. Il est important de noter que ni `\def` ni `\let` ne préviennent d'aucune façon l'utilisateur qu'il écrase une séquence de contrôle et que sa définition précédente sera perdue. Et tant pis si c'est une primitive. Si par malheur on écrit `\let\def\foo`, alors, à partir de ce moment, la primitive `\def` sera écrasée et il ne sera donc plus possible de définir de commande.

■ **Exercice 5**

Que fait le code suivant : `\let\foo\let\foo\bar\foo` ?

□ **Solution**

En procédant comme TEX et en ne lisant que ce dont on a besoin, on a d'abord `\let\foo\let` et donc `\foo` devient égale à la primitive `\let`.

Ensuite, on a `\foo\bar\foo` mais comme `\foo` est devenue un alias de `\let`, tout se passe comme si l'on écrivait `\let\bar\let` et donc, `\bar` devient aussi égale à `\let`.

Le code rend donc `\foo` et `\bar` égales à `\let`. ■

La commande `\let` peut aussi être utilisée pour définir des « caractères implicites ». La syntaxe ne change pas, mais au lieu de la seconde séquence de contrôle, on met un caractère. Si on écrit par exemple `\let\foo=a`, la séquence de contrôle `\foo` devient un « a implicite ». Cela signifie qu'elle produira un « a » à l'affichage et que lorsque TEX teste cette séquence de contrôle (nous verrons comment plus tard), il la voit égale à « a ». Il est bien évident que `\foo` ne peut pas être utilisée à la place de « a » n'importe où dans le code. Si on veut écrire `\par`, on ne peut pas écrire `\p\foo r` !

La définition d'un caractère implicite via `\let` ne peut se faire que pour les caractères *inoffensifs* que l'on peut rencontrer, c'est-à-dire ayant un catcode 11 ou 12, ainsi que pour ceux ayant le catcode 1, 2, 3, 4, 6, 7, 8. Il est par exemple impossible de définir un « % » ou un « \ » implicite avec `\let`. Pour l'espace, cela requiert un plus haut niveau de TEXnicité que nous verrons plus loin (page 51).

Pour les accolades ouvrantes et fermantes (dont les catcodes sont respectivement 1 et 2), il existe des tokens implicites prédéfinis très importants qui sont `\bgroup` et `\egroup`. Ils sont définis par plain-TEX de la façon suivante :

```
\let\bgroup={        \let\egroup=}
```

Ces *accolades implicites* peuvent jouer le même rôle de délimiteur de groupe que les accolades explicites « { » et « } » mais dans la grande majorité des cas, elles ne sont *pas* interchangeables avec les accolades explicites. Par exemple, elles ne peuvent pas être utilisées pour dans la syntaxe de `\def` pour délimiter le texte de remplacement d'une macro.

À ce stade, on peut donc faire un point TEXnique sur tout ce qui sert à délimiter un groupe à savoir :

- la paire `\begingroup ...\endgroup` ;
- les accolades explicites « { » et « } » ;
- les accolades implicites `\bgroup ...\egroup`.

Tout d'abord, un groupe ouvert par `\begingroup` ne peut être refermé que par `\endgroup` et donc, le groupe semi-simple est totalement indépendant du groupe ouvert par des accolades. Les choses sont différentes pour les groupes ouverts et fermés par « { » et « } » ou par « `\bgroup` » et « `\egroup` ». Tout d'abord, s'il n'existe aucune contrainte d'équilibrage d'accolades, « { » et `\bgroup` sont interchangeables pour ouvrir un groupe et « } » et `\egroup` le sont pour le fermer. Enfin, il y a quelques *rares* primitives où les accolades explicites et implicites sont interchangeables dans leur syntaxe :

- les primitives `\hbox`, `\vbox` et `\vtop` attendent après elles un texte entre accolades pour le composer dans une boite (pour en savoir plus, voir la section 8.3 page 250 et suivantes).
  Ce sont les seules primitives où l'on peut indifféremment utiliser des accolades explicites ou implicites, aussi bien pour ouvrir que pour fermer. Par conséquent, « `\hbox{a}` » et « `\hbox\bgoup a\egroup` » sont équivalents ;
- certaines primitives que nous verrons et utiliserons plus loin ont une propriété encore plus curieuse…
  Ces primitives doivent être immédiatement suivies d'une accolade ouvrante. Les plus connues sont `\toks`⟨*nombre*⟩, `\lowercase`, `\uppercase` et pour celles de ε-TEX, `\detokenize` et `\unexpanded`. Elles tolèrent indifféremment « { » ou « `\bgroup` » pour l'accolade ouvrante. En revanche, l'accolade fermante *doit* être une accolade explicite.
  Ainsi, « `\lowercase{Foo}` » et « `\lowercase\bgroup Foo}` » sont acceptés et sont équivalents.

■ **Exercice 6**

Si l'on écrit le code ci-dessous, va-t-on obtenir « $a^3 = b_5$ » ?

```
\let\swichmath=$ \let\expo= ^ \let\ind _ \let\egal= =
\swichmath a\expo3\egal b\ind5$
```

□ **Solution**

Oui car tous les tokens implicites définis ici le sont avec la bonne syntaxe (le signe = et l'espace facultatif qui le suit sont présents ou pas) et surtout, les catcodes des tokens implicites permettent une telle définition. ■

■ **Exercice 7**

Que va t-il se passer si l'on définit la macro `\foo` ainsi : `\let\foo={hello}`

□ **Solution**

TEX va d'abord lire `\let\foo={` et donc, `\foo` va devenir une accolade implicite.

Ensuite, le reste du code à lire est « `hello}` ». Tout va bien se passer jusqu'à l'accolade fermante. La première accolade ouvrante ayant été lue et absorbée par le `\let`, celle-ci n'a pas été prise en compte dans le comptage interne à TEX concernant l'équilibrage des accolades, l'accolade fermante devient orpheline et TEX va nous gratifier d'un « `Too many }'s` ». Cette

dernière erreur ne surviendra pas si auparavant, une accolade *x* a été ouverte sans encore être fermée par une accolade fermante ; dans ce cas, l'accolade qui suit « `hello` équilibrera l'accolade *x* ouverte préalablement à ce code.

Pour *définir* une macro et lui donner un texte de remplacement, on ne doit utiliser que `\def` puis le nom de la macro puis le texte de remplacement entre accolades. ■

La primitive `\let` est parfois utile pour faire une sauvegarde d'une commande de façon à pouvoir la restaurer à son état initial après y avoir apporté des modifications. Prenons un mauvais exemple et décidons de modifier la macro `\TeX` pour qu'elle n'affiche plus « TEX » mais « tEx ». Nous allons d'abord définir avec `\let` un alias de `\TeX` que nous appelons `\TeXsauve` puis nous pourrons modifier la commande `\TeX` et utiliser cette commande avec sa nouvelle définition. À la fin, avec `\let` à nouveau, nous la restaurerons pour revenir au comportement initial :

Code n° II-16

```
1 Initialement, c'est \TeX.\par
2 \let\TeXsauve\TeX% sauvegarde
3 \def\TeX{tEx}% redéfinition
4 Ici, on a modifié \TeX.\par
5 \let\TeX\TeXsauve% retauration
6 De nouveau, c'est \TeX.
```

Initialement, c'est TEX.
Ici, on a modifié tEx.
De nouveau, c'est TEX.

1.3. Les primitives \csname et \endcsname

Pour écrire une séquence de contrôle, on peut bien sûr faire précéder son nom du caractère « \ », c'est que nous avons fait jusqu'ici… Mais il y a une autre méthode pour créer une séquence de contrôle. On peut utiliser la paire de primitives `\csname` et `\endcsname` qui utilisent les caractères qui se trouvent entre elles pour les convertir en séquence de contrôle. Par exemple, si TEX rencontre `\csname␣foo \endcsname` [4], dans un premier temps il construira l'unité lexicale « `\foo` » et dans un deuxième temps, il lui substituera son texte de remplacement, tout ceci étant effectué dans la mémoire de TEX, dans la pile d'entrée. À l'affichage, l'effet est le même que si l'on avait directement écrit `\foo`, mais dans les entrailles de TEX, il y a une étape de plus, celle de former la séquence de contrôle avec ce qui se trouve entre `\csname` et `\endcsname`. L'avantage de cette méthode est que l'on peut utiliser des caractères normalement interdits dans le nom de commandes, c'est-à-dire des tokens dont le catcode est différent de 11, celui des lettres. On peut donc construire des séquences de contrôle dont le nom comporte des espaces, des chiffres, des signes d'opérations, de ponctuation ainsi que d'autres caractères réservés dont les codes de catégorie sont spéciaux (`&`, `#`, `^`, `_`, `$`).

Un autre avantage non négligeable est que si jamais la séquence de contrôle formée par `\csname…\endcsname` n'est pas définie, TEX la rend égale à `\relax` qui est une primitive dont l'action est de ne rien faire. Ainsi, écrire `\foobar` dans le code alors que cette macro n'a pas été définie par `\def` ou `\let` provoquera une erreur de compilation « `Undefined control sequence` ». En revanche, écrire `\csname`

4. L'espace qui suit `\csname` est ignoré en vertu de la règle vue à la page 42

`foobar\endcsname` revient à invoquer la primitive `\relax` et il ne se passera donc rien.

Le revers de la médaille est que, pour TEX, former une séquence de contrôle avec `\csname…\endcsname` est *beaucoup* plus lent que de l'écrire directement avec le caractère d'échappement. Si c'est possible, il vaut donc mieux éviter d'utiliser cette méthode dans les boucles des programmes.

■ Exercice 8

Après avoir défini la macro `\foo` avec `\def\foo{Bonjour}`, pourquoi rien ne s'affiche lorsqu'on écrit « `\csname foo \endcsname` » ?

□ Solution

Parce que l'espace qui suit le « foo » est pris en compte pour construire le nom de la séquence de contrôle. Ainsi, écrire `\csname␣foo␣\endcsname` construit la commande `\foo` qu'il serait fastidieux de construire autrement. Celle-ci étant indéfinie, la construire avec `\csname…\endcsname` la rend égale à `\relax` et il ne se passe donc rien. ■

■ Exercice 9

Pourquoi écrire `\let\salut\csname foo\endcsname` pour rendre égale `\salut` à `\foo` provoque une erreur de compilation ?

□ Solution

L'erreur survient parce que TEX lit le code de gauche à droite en prenant en compte uniquement ce dont il a besoin. Il lit donc d'abord `\let\salut\csname`. Ce faisant, il rend donc la commande `\salut` égale à `\csname`. Ayant effectué cette opération, il poursuit avec la suite du code qui est `foo\endcsname`. Les lettres « foo » ne vont pas lui poser de problème, il les affichera normalement, mais il va rencontrer la commande `\endcsname` sans avoir rencontré le `\csname` qui initiait le début de la construction du nom de la commande. Il va donc émettre le message d'erreur « `Extra \endcsname` ».

Pour faire ce que nous voulions, il faudrait forcer TEX à construire la séquence de contrôle `\foo` à partir de `\csname␣ foo\endcsname` avant qu'il ne passe la main à `\let`. Pour provoquer cette action, il faudrait sauter les deux tokens `\let\salut` pour provoquer cette action et revenir où l'on était avant les sauts pour reprendre la lecture du code dans l'ordre normal. Pour effectuer cette manœuvre non linéaire qui déroge à la règle de lecture du code, il faut avoir recours à la primitive `\expandafter` que nous verrons plus loin. ■

■ Exercice 10

On a vu comment définir une commande avec `\def`. Existe-t-il un moyen d'annuler une définition et faire qu'une commande préalablement définie redevienne indéfinie ?

□ Solution

Il n'existe pas de primitive `\undef` qui rendrait une commande indéfinie *et* l'enlèverait de la table de hashage [5].

On peut cependant combler ce manque. En effet, si l'on définit une commande à l'intérieur d'un groupe simple ou semi-simple, la définition est locale au groupe et est perdue à sa fermeture. Par conséquent, si la commande était indéfinie avant l'ouverture du groupe, elle le redevient après sa fermeture.

Si l'on souhaite définir une commande de façon globale, la primitive `\global`, lorsqu'elle est placée devant une assignation, rend cette assignation globale c'est-à-dire qu'elle survit à la fermeture du groupe. La primitive `\gdef` se comporte comme `\global\def`. ■

5. Il s'agit d'une structure interne à TEX où sont stockés les appariements entre les noms des macros et l'endroit de la mémoire où sont stockés leurs textes de remplacement.

1.4. Caractère de contrôle

14 - Règle
Lorsque le caractère d'échappement « \ » est suivi d'un caractère dont le catcode n'est pas 11, celui des lettres, seul ce caractère est pris en compte et TEX forme ce que l'on appelle un « caractère de contrôle ».

Pour former un caractère de contrôle, on peut donc utiliser des caractères de catcode 12 comme par exemple « \. », « * », « \5 » ou « \@ ». Il devient surtout possible d'employer des caractères dont le catcode est plus sensible comme « \\ », « \{ », « \} », « \^ », « _ », « \& », « \~ », « \% ». Ces derniers, à l'exception de \\ [6], sont définis par plain-TEX pour afficher le caractère qui forme leur nom, caractère qu'il serait plus difficile d'afficher sinon. Pour cela, la primitive `\chardef` est employée avec cette syntaxe :

`\chardef\`⟨*macro*⟩`=`⟨*code de caractère*⟩

Si l'on s'intéresse au code des macros du format plain-TEX qui est contenu dans le fichier « `plain.tex` », on découvre par exemple à la ligne n° 651 que le caractère de contrôle \& est défini par

```
\chardef\&=`\&
```

ce qui a pour effet de rendre équivalent (à la manière de `\let`) le caractère de contrôle \& à « `\char`\& ».

15 - Règle
Contrairement aux séquences de contrôle dont le nom est formé de lettres, les espaces qui suivent les « caractères de contrôle » ne sont pas ignorés. La seule exception est la primitive \ , qui ajoute un espace à la liste courante. En effet, si cette dernière est suivie d'un espace, alors deux espaces se suivent dans le code et une autre règle stipule que le second est ignoré.

■ **Exercice 11**
En ayant fait les définitions « `\def\9{\␣\:}` » et « `\def\:{␣\␣␣X}` », quel affichage obtiendra-t-on avec « `a\9␣fin` » ?

□ **Solution**
On obtient la lettre « a » suivie du texte de remplacement de \9. Celui-ci commence par \␣ (espace qui sera affiché) suivi de \:. Le texte de remplacement de \: commence par : ␣ (espace qui sera affiché) puis \␣␣ (le deuxième espace est ignoré) suivi d'un « X ». Et enfin, l'espace après \9␣ sera affiché avant le mot « fin ». On obtient donc « a␣␣␣X␣fin ». ■

6. La macro \\ est définie par plain-TEX car elle est utilisée comme macro auxiliaire pour définir une autre macro et la définition qui lui est donnée à cette occasion est laissée en l'état. Toujours est-il qu'elle n'affiche pas « \ », qui s'obtient avec « `\char92` » ou mieux avec « `\char`\\ ».

Du côté de LATEX, la macro \\ est définie pour commander d'aller à la ligne pendant le paragraphe en cours et est redéfinie dans plusieurs environnements (centering, raggedright, raggedleft, eqnarray, tabbing, array, tabular).

■ **Exercice 12**

Quelle différence y t-il entre « \+ » et « \csname+\endcsname » ?

□ **Solution**

Il n'y a pas de différence à l'affichage sauf si le caractère qui suit dans le code est un espace. Il sera comptabilisé avec \+ et ignoré avec \csname+\endcsname car l'espace qui suit la séquence de contrôle \endcsname est ignoré.

De plus, si le caractère de contrôle \+ n'a pas été défini, écrire \+ provoque une erreur à la compilation de type « `Undefined control sequence` ». Au contraire, avec \csname+\endcsname, il n'y aurait pas d'erreur car lorsque \csname construit une séquence de contrôle non définie, elle la rend égale à \relax, et ici, tout se passerait comme si on avait écrit \let\+=\relax. ■

■ **Exercice 13**

Quelle différence y a-t-il entre « \def\foo{A} » et « \let\foo=A » ?

□ **Solution**

À l'affichage, dans les deux cas, \foo produit un « A ». Par contre, en interne, les choses sont différentes…

Lorsque \foo est définie avec \def, TEX la *remplace* par un « A ».

Lorsque \foo est définie avec \let, aucun remplacement n'est fait puisque \foo *est* un « A » et n'a aucun texte de remplacement. ■

Le fait que les commandes caractères prennent en compte les espaces qui les suivent permet de définir un espace implicite \sptoken* de cette façon [7] :

```
\def\:{\let\sptoken= }
\:
```

Toute l'astuce réside dans le fait que l'espace qui suit \: à la 2e ligne est pris en compte. La manœuvre mérite une petite explication. La première ligne définit un texte de remplacement pour la macro \:. À la deuxième ligne, TEX va remplacer \: par son texte de remplacement ce qui va donner :

```
\let\sptoken= ␣
```

Pour plus de clarté, le texte de remplacement de \: est encadré. Dans ce texte de remplacement, l'espace qui suit le signe = est l'espace facultatif qui est permis avec la primitive \let. Cet espace, faisant partie de la syntaxe de \let, est ignoré et donc, \let rend \sptoken égal au token suivant qui est « ␣ », l'espace qui suit le caractère de contrôle \:.

Dans le code ci-dessus, le second espace n'est pas ignoré car les deux espaces ne sont pas consécutifs dans le code source. Ils sont consécutifs après que TEX ait remplacé \: par son texte de remplacement. La règle de la page 42 ne s'applique donc pas.

■ **Exercice 14**

Comment serait défini \sptoken si l'on écrivait naïvement \let\sptoken=␣␣ ?

7. C'est ainsi que \@sptoken est défini dans le code du noyau LATEX.

□ **Solution**

Comme on l'a vu, le premier espace, facultatif, fait partie de la syntaxe de `\let` et ne peut donc pas servir à définir `\sptoken`. Le second est ignoré puisque la règle veut que deux espaces consécutifs dans le code n'en forment qu'un.

Dans ce cas, `\sptoken` sera un alias pour le token qui *suivra dans le code* et que l'on n'a pas précisé dans l'énoncé de l'exercice. Il faut noter que si ce qui suit est un saut de ligne, `\sptoken` sera un alias de la commande `\par` car deux sauts de lignes consécutifs sont équivalents à la commande `\par` :

Code n° II-17

```
1 \let\sptoken=  %2 espaces avant le "%"
2
3 La commande \sptoken compose le paragraphe
```

La commande
compose le paragraphe

■

1.5. Caractères actifs

Un caractère est dit *actif* lorsqu'il revêt les mêmes propriétés qu'une commande : on peut le définir avec `\def` et, lorsqu'il est rencontré par TEX, son texte de remplacement lui est substitué. En contrepartie, il ne fait plus partie de la catégorie de lettres et n'est donc pas autorisé dans les noms des séquences de contrôle.

16 - Règle

Un caractère est *actif* lorsque son code de catégorie vaut 13. Dans ce cas, on peut lui donner un texte de remplacement comme on le fait pour une macro.

Par défaut, aussi bien avec plain-TEX qu'avec LATEX, seul le caractère « ~ » est actif fait les choses suivantes :

1. interdire une coupure en spécifiant une haute pénalité (avec la primitive `\penalty` suivie d'un entier élevé, par exemple 1 000) ;
2. appeler la primitive `\ ` qui affiche une espace.

Ainsi, plain-TEX dit :

```
\def~{\penalty1000 \ }
```

La forte pénalité de `1000` interdit toute coupure et le tout forme donc une « espace insécable », où l'adjectif *insécable* exprime que cette espace ne pourra donner lieu à une coupure de ligne.

À notre tour, supposons que l'on souhaite rendre le caractère « `W` » actif. Nous le ferons à l'intérieur d'un groupe, car il est prudent de limiter la portée des modifications des codes de catégorie.

Nous savons que `\catcode‘\`⟨*car*⟩=⟨*nombre*⟩ modifie le catcode d'un caractère, où ⟨*nombre*⟩ est un nombre de 0 à 15 [8]. Ici, on va donc écrire `\catcode‘\W=13`. Et

8. Le caractère « ‘ » est l'apostrophe *inverse* et s'obtient parfois en tapant la combinaison de touches AltGr+7. Pour exprimer le nombre égal au code de catégorie d'un caractère, on peut écrire ^^<car> ou ‘<car>. La seconde écriture est cependant moins recommandable puisque l'on s'expose à des erreurs lorsque <car> a un catcode spécial (0 ou 14).

ensuite, il suffit de définir le texte de remplacement du caractère actif W, par exemple « wagons » :

Code n° II-18

```
{% début du groupe
\catcode`\W=13 \def W{wagons}
Les W constituent le train.
}% fin du groupe
```

Les wagons constituent le train.

Voici la règle corroborant ce que l'on constate dans l'exemple ci-dessus :

17 - Règle

Un espace après un caractère actif est pris en compte.

Il est plus difficile est de créer une commande qui rende le caractère W actif. Appelons \actiw cette commande et donnons-nous le cahier des charges suivant : elle devra rendre le caractère W actif et lui donner le texte de remplacement « wagons ». Nous placerons un \begingroup avant d'appeler cette macro et un \endgroup à la fin du texte où l'on souhaite que le caractère W soit actif.

Pour définir cette commande \actiw, on ne peut pas écrire directement ceci :

```
\def\actiw{\catcode`\W=13 \def W{wagons}}
```

En effet, lorsque TEX remplace \actiw par son texte de remplacement, *tous* les tokens dans ce texte de remplacement ont leurs catcodes figés depuis que la définition a été faite. Et donc, le « W » qui suit le \def a un catcode de 11.

Par conséquent, lorsque ce texte de remplacement sera exécuté, TEX va trouver \def W{wagons} où W est une lettre (de catcode 11), et il va se plaindre de ne pas trouver une séquence de contrôle après le \def en affichant le message d'erreur « Missing control sequence ».

Il faut comprendre que l'ordre « \catcode`\W=13 » qui est dans texte de remplacement ne peut pas agir sur les « W » de ce texte de remplacement puisque leurs catcodes sont figés, mais agira sur le code qui reste à lire *après* \actiw.

Pour sortir de ce mauvais pas, il faut rendre W actif *avant* que TEX ne lise le texte de remplacement de \wagon. Pour limiter la portée de cette opération, on va donc le faire dans un groupe et utiliser \gdef, pour que la définition soit globale et survive à la fermeture du groupe. Voici la façon correcte de définir \actiw :

Code n° II-19

```
\begingroup \catcode`\W=13
\gdef\actiw{%
  \catcode`\W=13
  \def W{wagons}}
\endgroup
a) Les trains ne sont pas constitués de W !\par
b) \begingroup\actiw Ici, les W forment les trains.\endgroup\par
c) Ici, les W sont redevenus des lettres.
```

a) Les trains ne sont pas constitués de W !
b) Ici, les wagons forment les trains.
c) Ici, les W sont redevenus des lettres.

■ **Exercice 15**

Définir deux séquences de contrôle \> et \< telles qu'entre elles, les mots soient espacés de 5 mm. Le comportement normal doit être rétabli ensuite.

On utilisera la primitive \hskip suivie d'une dimension pour ordonner à TEX d'insérer une espace horizontale de la valeur de la dimension.

□ **Solution**

Comme on l'a vu, on ne peut pas écrire directement ce code comme ci-dessous pour définir la macro \> :

```
\def\>{\begingroup\catcode`\ =13 \def {\hspace{5mm}}}
```

puisqu'au moment où cette ligne est lue par TEX, l'espace a son code de catégorie naturel de 10. Celui-ci ne pourra plus être changé par la suite.

Voici la bonne façon de procéder : il suffit de rendre la commande \< égale à \endgroup ce qui clôturera le processus initié par \> qui avait ouvert le groupe semi-simple :

Code n° II-20

```
\begingroup
  \catcode`\ =13 % rend l'espace actif
  \gdef\>{\begingroup
    \catcode`\ =13
    \def {\hskip5mm\relax}}
\endgroup
\let\<=\endgroup
a) Du texte normal\par
b) \>Des mots très espacés\<\par
c) Du texte normal
```

a) Du texte normal
b) Des mots très espacés
c) Du texte normal

■

La présence d'un \relax après la dimension de 5mm stoppe la lecture de la dimension. En effet, comme nous le verrons plus loin, une dimension peut avoir des composantes étirables qui commencent par le mot-clé « plus ». Sans le \relax, si jamais un espace actif était suivi du mot « plus », alors TEX penserait que ce mot fait partie intégrante de la dimension et irait chercher encore plus loin la composante étirable. Le \relax agit donc ici comme une sécurité.

Une application utile des caractères actifs est de rendre les signes de ponctuation haute actifs. C'est ce que fait l'extension pour LATEX « babel » chargée avec l'option « frenchb » de Daniel Flipo. Les signes de ponctuation *haute* « ! », « ? », « : » et « ; » sont rendus actifs de façon à insérer une espace fine insécable avant eux. Nous allons imiter ce comportement et rendre la virgule active. L'idée est de la programmer pour qu'elle supprime un éventuel espace avant elle, affiche le caractère « , », insère une espace justifiante (c'est-à-dire étirable) et enfin, ne tienne pas compte des espaces qui pourraient suivre. Pour effectuer ces actions, certaines primitives sont nécessaires :

- si ce qui précède est un espace étirable, la primitive \unskip le supprime ;
- \string transforme le token qui suit en un ou plusieurs tokens dont les codes de catégorie sont 12. Cette primitive est utile pour rendre n'importe quel token inoffensif et donc immédiatement affichable. Par exemple, « \string# »

produit « # » de même que « \string\def » produit « \def ». Lorsque la virgule sera active, « \string, » affichera une virgule non active ;

– \ignorespaces demande à TEX d'ignorer tous les espaces qui vont suivre et d'avancer sa tête de lecture jusqu'au prochain caractère qui n'est pas un espace.

Code n° II-21

```
\begingroup
  \catcode`\,=13 \def,{\unskip\string,\space\ignorespaces}
  La rue assourdissante autour de moi hurlait.

  Longue , mince,en grand deuil  ,  douleur majestueuse ,

  Une femme passa   ,d'une main fastueuse

  Soulevant,balançant le feston et l'ourlet ;

\endgroup\medbreak\hfill Charles {\sc Baudelaire}
```

La rue assourdissante autour de moi hurlait.
Longue, mince, en grand deuil, douleur majestueuse,
Une femme passa, d'une main fastueuse
Soulevant, balançant le feston et l'ourlet ;

Charles Baudelaire

■ **Exercice 16**

Par défaut, le caractère inséré par TEX à chaque fin de ligne est le retour charriot, qui s'écrit « ^^M » et qui a comme code de catégorie 5. La règle de TEX vue page 29 veut que *deux* retours charriots consécutifs (ou plus généralement deux tokens consécutifs de catcode 5) soient équivalents à la commande \par .

Reprendre l'exemple précédent en faisant en sorte qu'un seul retour charriot suffise à aller à la ligne suivante.

□ **Solution**

Il suffit de rendre le caractère ^^M actif et le rendre égal à \par avec un \let :

Code n° II-22

```
\begingroup
  \catcode`\,=13 \def,{\unskip\string, \ignorespaces}
  \catcode`\^^M=13 \let ^^M\par  % rend le retour charriot égal à \par
  La rue assourdissante autour de moi hurlait.
  Longue , mince,en grand deuil  , douleur majestueuse ,
  Une femme passa   ,   d'une main fastueuse
  Soulevant,balançant le feston et l'ourlet ;
\endgroup\medbreak\hfill Charles {\sc Baudelaire}
```

La rue assourdissante autour de moi hurlait.
Longue, mince, en grand deuil, douleur majestueuse,
Une femme passa, d'une main fastueuse
Soulevant, balançant le feston et l'ourlet ;

Charles Baudelaire

■

■ Exercice 17

Dans un texte à l'intérieur d'un groupe, inventer un procédé qui effectue un remplacement par permutation circulaire des voyelles, c'est-à-dire qui remplace a par e, e par i, i par o, o par u, u par y et y par a.

□ Solution

La difficulté vient du fait qu'une fois qu'un caractère est rendu actif, il devient semblable à une séquence de contrôle et on ne peut plus s'en servir pour former le nom d'une séquence de contrôle. C'est donc une méthode peu recommandable, il en existe d'ailleurs de meilleures. Il n'empêche qu'en l'état actuel de nos connaissances, nous ne pouvons procéder qu'en touchant aux codes de catégorie.

Avant de toucher aux catcodes des voyelles, on va donc définir des séquences de contrôle `\AA`, `\EE`, etc. qui, à l'aide de la commande `\let` vont devenir des lettres implicites, « a » pour `\AA`, « e » pour `\EE`, etc.

Ensuite, les commandes `\catcode` et `\let` contenant des voyelles, on ne pourra pas y faire appel après avoir modifié les codes de catégorie. On va donc définir avec `\let` des séquences de contrôle équivalentes dont les voyelles seront en majuscule. On aura donc `\lEt` pour `\let` et `\cAtcOdE` pour `\catcode`.

On peut ensuite rendre chaque voyelle active et la mettre `let`-égale à `\AA`, `\EE`, etc. selon la lettre que l'on veut obtenir.

Code n° II-23

```
{% ouvre un groupe
  \let\AA=a \let\EE=e \let\II=i \let\OO=o \let\UU=u \let\YY=y
  % sauvegarder avant de modifier le catcode de a et e :
  \let\lEt=\let \let\cAtcOdE=\catcode
  \cAtcOdE`\a=13 \lEt a=\EE \cAtcOdE`\e=13 \lEt e=\II
  \cAtcOdE`\i=13 \lEt i=\OO \cAtcOdE`\o=13 \lEt o=\UU
  \cAtcOdE`\u=13 \lEt u=\YY \cAtcOdE`\y=13 \lEt y=\AA
  Ce texte devenu \`a peine reconnaissable montre que le r\'esultat
  contient des sonorit\'es catalanes, corses ou grecques assez
  inattendues.
}% ferme le groupe
```

Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís cetelenis, cursis uy gricqyis essiz onettindyis.

Les contorsions nécessaires pour arriver à nos fins montrent que l'on ne touche pas aux codes de catégorie sans se créer de réelles difficultés et sans prendre de gros risques si l'on contrôle mal la portée des modifications. Changer un code de catégorie de caractère doit donc rester exceptionnel et rester réservé à des buts très spécifiques. En effet, des méthodes plus sures existent, même si elles ont aussi d'autres inconvénients.

Une optimisation du code serait d'utiliser le caractère déjà actif « ~ » comme alias de `\catcode`. Bien sûr, on y perd en lisibilité… On peut aussi profiter de la permutation circulaire pour définir chaque voyelle rendue active comme alias de la prochaine voyelle, non encore rendue active. On économise ainsi des séquences de contrôle pour les lettres implicites. On n'en a besoin que d'une seule, `\AA`, lettre implicite pour « a » :

Code n° II-24

```
{%
  \let\AA=a \let\lEt=\let \let~=\catcode
  ~`a=13 \lEt a=e ~`e=13 \lEt e=i ~`i=13 \lEt i=o
  ~`o=13 \lEt o=u ~`u=13 \lEt u=y ~`y=13 \lEt y=\AA
  Ce texte devenu \`a peine reconnaissable...
}
```

Ci tixti diviny è pioni ricunneossebli...

■

1.6. Signification d'une commande

Il est possible de demander à TEX la « signification » d'une séquence de contrôle. Pour cela, la primitive \show écrit dans le fichier log le texte de remplacement de la macro ou simplement son nom si c'est une primitive. Cette primitive, lorsqu'elle est utilisée à bon escient dans le code permet un débogage efficace par la lecture du fichier log. Pour diriger les informations délivrées par \show dans le flux de lecture de TEX pour par exemple les afficher dans le document final, il faut utiliser la primitive \meaning.

Code n° II-25

```
a) \def\foo{Bonjour}\meaning\foo\par
b) \let\bar=\foo\meaning\bar\par% \foo est "copiée" vers \bar
c) \def\baz{\foo}\meaning\baz\par
d) \catcode`\W=13 \def W{Wagons}% W est un caractère actif
   \meaning W\par
e) \meaning\def% \def est une primitive
```

a) macro:->Bonjour
b) macro:->Bonjour
c) macro:->\foo
d) macro:->Wagons
e) \def

Il est important d'insister entre la différence qui existe entre \let\bar\foo (cas b) et \def\baz{\foo} (cas c). Dans le premier cas, \bar est rendue égale à \foo et à partir de ce moment, \bar a comme texte de remplacement « Bonjour », indépendamment de ce que devient \foo. En revanche, lorsqu'on écrit \def\baz{\foo}, la macro \baz a comme texte de remplacement \foo et donc, à chaque fois qu'elle est exécutée, elle dépend de ce qu'est \foo à ce moment-là.

■ **Exercice 18**

Mettre en évidence à l'aide de la primitive \meaning que 2 retours charriots consécutifs sont interprétés par TEX comme étant la primitive \par.

□ **Solution**

Il suffit de définir une commande dont le texte de remplacement est constitué de 2 retours charriots consécutifs et de faire afficher à TEX la signification de cette commande, voire faire suivre \meaning de deux retours à la ligne :

Code n° II-26

```
a) \def\foo{

}Signification de \string\foo{} : \meaning\foo

b) Signification de deux retours charriots consécutifs : \meaning

```

a) Signification de \foo : macro:-> \par
b) Signification de deux retours charriots consécutifs : \par

■

18 - Règle

La primitive \show écrit dans le fichier log la « signification » du token qui la suit :

1. si ce token est une macro (ou un caractère actif), le texte « macro–> » suivi du texte de remplacement de cette macro est écrit ;
2. si ce token est une primitive, qui par définition n'a pas de texte de remplacement, le nom de la primitive est écrit ;
3. sinon, TEX écrit un texte bref (caractérisant le catcode du token) suivi du token.

La primitive \meaning écrit les mêmes choses que \show sauf qu'elle les écrits dans le flux de lecture de TEX, dans la pile d'entrée.

Voici ce que provoque \meaning pour les tokens de chaque catégorie qu'il est possible de mettre après \meaning :

Code n° II-27

```
a) \meaning\relax\par% primitive
b) \meaning {\par% catcode 1
c) \meaning }\par% catcode 2
d) \meaning $\par% catcode 3
e) \meaning &\par% catcode 4
g) \meaning #\par% catcode 6
h) \meaning ^\par% catcode 7
i) \meaning _\par% catcode 8
j) \meaning a\par% catcode 11 (une lettre)
k) \meaning +\par% catcode 12 (caractère "autre")
l) \meaning ~\par% catcode 13 (caractère actif)
```

```
a) \relax
b) begin-group character {
c) end-group character }
d) math shift character $
e) alignment tab character &
g) macro parameter character #
h) superscript character ^
i) subscript character _
j) the letter a
k) the character +
l) macro:->\penalty \@M \
```

■ **Exercice 19**

On ne peut pas afficher la signification d'un espace (catcode 10) puisqu'écrire « \meaning␣ » ferait que l'espace qui suit la primitive serait ignoré et \meaning prendrait le token d'après. Comment s'y prendre pour afficher la signification d'un espace avec \meaning ?

□ **Solution**

On peut employer la même astuce qu'avec \sptoken de la page 51. On s'y prend ici avec \let, en enfermant le tout dans un groupe semi-simple :

Code n° II-28

```
\begingroup% dans un groupe
  \let\*=\meaning% rendre \* égal à \meaning
  \* %<- espace pris en compte
\endgroup% fermer le groupe
```

blank space

En revanche, on ne peut pas écrire après `\meaning` les caractères de catcode suivants :

- 0 car le caractère d'échappement s'apparie toujours avec autre chose pour former une séquence de contrôle. Un token de catcode 0 ne peut exister comme entité seule ;
- 5 car ce caractère est vu comme un espace ;
- 9 car c'est le numéro de la catégorie des caractères « ignorés » et le token serait aussi ignoré, même après `\meaning` ;
- 14 car un caractère de commentaire est ignoré ainsi que tout ce qui va jusqu'à la fin de la ligne ;
- 15 car un caractère « invalide » n'est pas permis et provoque une erreur de compilation lorsqu'il est vu par TEX. ■

1.7. Le caractère d'échappement

Par défaut, le caractère d'échappement est « \ », c'est à ce caractère que TEX reconnait le début d'une séquence de contrôle. Comme presque tout dans TEX, ce caractère est modifiable. En effet, tout caractère de catcode 0 est, par définition, un caractère d'échappement. Supposons qu'à l'intérieur d'un groupe semi-simple, on donne à « * » le code de catégorie 0 et à « \ » le code de catégorie 12 qui est celui des caractères « autres ». Ayant fait ces modifications, nous devrons mettre « * » devant les séquences de contrôle et pourrons alors afficher « \ » directement puisqu'il sera devenu un caractère inoffensif :

Code n° II-29

```
\begingroup
\catcode`\*0
\catcode`\\12 % \ n'est plus un caractère d'échappement
*def*foo{bar}
*LaTeX{} et la macro *string*foo : *foo.

L'ancien caractère d'échappement  "\" et \TeX.
*endgroup
```

LaTeX et la macro \foo : bar.
L'ancien caractère d'échappement "\" et \TeX.

■ **Exercice 20**

Concernant l'exemple précédent :

1. Aurait-on pu écrire à la ligne 3 ceci : « `*catcode`*\12` » ?
2. Si l'on avait utilisé `*gdef` pour que la macro `*foo` survive à la fermeture du groupe, aurait-on pu l'appeler par la suite avec `\foo` ?

□ **Solution**

1. Après la ligne n° 2, il y a 2 caractères d'échappement qui sont « \ » et « * ». Tant qu'ils ont ce pouvoir, ils sont interchangeables. La réponse est donc oui.
   En revanche, la ligne 2 fait perdre ce pouvoir au caractère « \ » ce qui fait qu'ensuite, seul « * » peut être utilisé comme caractère d'échappement jusqu'à ce que le groupe soit fermé par *endgroup.
2. N'importe quel caractère ayant le catcode 0 peut être utilisé pour amorcer le nom d'une macro. En temps normal, seul « \ » est disponible donc la réponse est oui. ■

On peut observer que \string, qui convertit le nom d'une séquence de contrôle en caractères de catcode 12, ne tient pas compte de la modification du caractère d'échappement puisque l'on obtient \foo et non pas *foo. Cela dit, ce comportement est un peu normal car s'il y avait plusieurs caractères d'échappement, TEX serait bien ennuyé pour en choisir un. Pour schématiser, on peut considérer que le choix d'un caractère d'échappement par son catcode concerne un mécanisme d'*entrée* pour TEX puisqu'il s'agit d'une règle concernant la lecture du code tapé par l'utilisateur. La primitive \string appliquée aux macros, au contraire, concerne un mécanisme de sortie puisqu'elle provoque la conversion d'une donnée interne de TEX (le nom de la macro) en caractères immédiatement affichables. Ces deux mécanismes se représentent le caractère d'échappement par des moyens indépendants. Pour \string, il faut le définir via la primitive \escapechar qui attend après elle le code de caractère du caractère d'échappement que l'on souhaite. Dans l'exemple précédent pour que le caractère d'échappement affiché par \string soit « @ », il suffit de rajouter juste après le \begingroup :

```
\espcapechar=64        ou bien        \escapechar=‘\@
```

Notons que tout comme \string, les primitives \meaning et \show tiennent compte de la modification du \escapechar. Enfin, si l'on assigne à \escapechar un code inférieur à 0 ou supérieur à 255, alors aucun caractère d'échappement ne précèdera le nom de la macro.

Code n° II-30

```
1 \def\foo{\bar}
2 \begingroup
3 a) \escapechar=-1  \meaning\foo\qquad% pas de caractère d'échappement
4 b) \escapechar=‘\@ \meaning\foo\qquad% "@" est le caractère d'échappement
5 \endgroup
6 c) \meaning\foo% caractère d'échappement par défaut
```

a) macro:->bar b) macro:->@bar c) macro:->\bar

■ **Exercice 21**

Définir une commande dont le nom est \1\2\a et dont le texte de remplacement est « foo » et vérifier en appelant cette commande que la définition est bien correcte.

□ **Solution**

Il va falloir agir dans un groupe, modifier les catcodes de 1, 2 et \ pour les mettre à 11, catégorie des lettres, seules autorisées dans les noms de macros. Comme « \ » est le caractère d'échappement, il va aussi falloir en définir un autre, choisissons « | ». Comme toutes ces manœuvres ont lieu dans un groupe, nous utiliserons |gdef pour que la définition de la macro survive à la fermeture du groupe.

Le problème ici est que l'on doit donner *en dernier* à « 1 » le catcode 11. En effet, si on le faisait avant, « 1 » deviendrait pour TEX une *lettre* et lorsqu'il rencontrerait |catcode‘|2=11, le « 11 » serait, à ses yeux, deux lettres qui ne peuvent former un nombre, car les tokens qui forment les nombres doivent avoir un catcode de 12 ! Nous aurions droit à une erreur de compilation du type « Missing number ».

Enfin, pour appeler la macro à l'extérieur du groupe, on doit utiliser la paire \csname... \endcsname dans laquelle on prendra soin de transformer le caractère d'échappement \ en un caractère inoffensif de catcode 12 avec la primitive \string.

Code n° II-31

```
\begingroup
  \catcode‘\|=0 |catcode‘|\=11 |catcode‘|2=11 |catcode‘|1=11
  |gdef|1\2\a{foo}
|endgroup
Voici la macro : \csname 1\string\2\string\a\endcsname
```

Voici la macro : foo

Dans ce cas, ces manipulations de catcode, assez dangereuses et plutôt chatouilleuses, sont fort heureusement inutiles avec la primitive \expandafter que nous verrons plus loin. Il s'agit d'un exemple purement pédagogique… ■

1.8. Une autre façon de stocker des tokens

Arrivés à ce point, nous savons qu'une macro peut servir à stocker des tokens. Ces tokens sont placés dans le texte de remplacement de la macro lors de sa définition avec \def. Bien qu'on les utilise aussi dans ce but, les possibilités d'une macro ne se limitent pas à ce simple stockage et c'est ce que nous allons découvrir dans les chapitres suivants.

Si l'on recherche uniquement le côté « stockage de tokens », TEX propose un autre type de structure : les registres de tokens. Ce sont de zones de mémoire (au nombre de 256 pour TEX et 32 768 pour ε-TEX) auxquelles on accède à l'aide de la primitive \toks suivie du numéro de registre. Ainsi, \toks17 fait référence au registre de token n° 17. Il est admis que le registre n° 0 sert de registre brouillon et peut être utilisé sans trop de risque n'importe où.

TEX met aussi à disposition la primitive \toksdef et par exemple, écrire

```
\toksdef\foo=17
```

rend \foo équivalent à \toks17 de telle sorte que le registre n° 17 peut être désigné par la séquence de contrôle \foo. L'équivalence dont il est question ici ressemble à celle de \let en ce sens que \foo n'aura pas de texte de remplacement, mais *sera* \toks17.

Il est déconseillé d'utiliser un numéro non nul arbitrairement et sans précaution, car il se pourrait que ce registre soit déjà utilisé et on pourrait écraser son contenu et provoquer des dysfonctionnements par la suite. Plain-TEX propose la macro \newtoks\⟨*nom*⟩ où le \⟨*nom*⟩ sera une séquence de contrôle créée par TEX et qui, par l'intermédiaire de la primitive \toksdef, rendra \⟨*nom*⟩ équivalent à \toks suivi du prochain numéro de registre inutilisé. Grâce à ce système, on peut donc demander en toute sécurité l'allocation d'un registre de tokens et y faire référence ensuite par un nom, ce qui est bien plus pratique à retenir qu'un numéro.

Pour effectuer une assignation à un registre de tokens, il faut écrire

⟨*registre de token*⟩= {⟨*ensemble de token*⟩}

ou bien

⟨*registre de token*⟩= ⟨*registre de token*⟩

où ⟨*registre de token*⟩ est soit une séquence de contrôle définie par \toksdef ou \newtoks, soit explicitement \toks⟨*numéro*⟩. Le signe « = » et l'espace qui le suit sont facultatifs et seront souvent omis.

Pour extraire du registre ce qu'il contient, on doit faire appel à la primitive \the suivie du ⟨*registre de token*⟩. Voici un exemple où est sont créés deux registres \foo et \bar :

Code n° II-32

```
\newtoks\foo% allocation d'un nouveau registre
\foo={Bonjour le monde}% assignation
Contenu du registre {\tt\string\foo} : \the\foo.

\newtoks\bar% allocation d'un autre registre
\bar=\foo% assigne à \bar les tokens du registre \foo
Contenu du registre {\tt\string\bar} : \the\bar.
```

Contenu du registre \foo : Bonjour le monde.
Contenu du registre \bar : Bonjour le monde.

Comme pour les séquences de contrôle, si l'assignation d'un registre à tokens se fait à l'intérieur d'un groupe, elle est annulée lors de sa fermeture à moins de faire précéder l'assignation de \global .

Chapitre 2

Arguments d'une commande

Les commandes vues jusqu'à présent étaient immuables en ce sens que leur texte de remplacement était figé une bonne fois pour toutes. Avec cette limite, elles ne seraient ni très puissantes ni très intéressantes. Heureusement, les commandes peuvent admettre des « arguments », variables eux, qui seront lus chaque fois que la commande sera appelée et qui seront insérés à des endroits du texte de remplacement choisis par l'utilisateur.

2.1. Qu'est ce qu'un argument ?

19 - Règle
Un argument d'une macro est :

- soit un token c'est-à-dire un caractère (un octet) ou une séquence de contrôle ;
- soit le code qui se trouve entre deux accolades, étant entendu que ce code est un ensemble de tokens équilibrés en accolades ouvrantes et fermantes.

Par exemple, si on les considère comme des arguments de commandes :

- « a » représente un argument qui est « a » ;
- « abc » représente 3 arguments qui sont « a », « b » et « c » ;
- « ab␣c » constitue 3 arguments aussi, les mêmes que ci-dessus. En effet, dans une liste d'arguments, un espace sous forme d'unité lexicale est ignoré. Pour qu'il soit pris en compte l'espace *doit* être mis entre accolades : « ␣ » n'est pas un argument alors que « {␣} » en est un.
- « {abc} » est un argument unique qui est « abc » ;

- « \foo\bar » sont deux arguments, « \foo » et « \bar » ;
- « {\foo\bar} » est un seul argument qui est « \foo\bar » ;
- enfin, « {\def\AA{a}}{␣}{c{d}e}{f} » est constitué de quatre arguments :
  1. \def\AA{a}
  2. ␣
  3. c{d}e
  4. f

20 - Règle
Un espace *non entouré d'accolades* est ignoré en tant qu'argument.

Il faut retenir que les accolades ne servent qu'à délimiter l'argument et n'en font pas partie. Lorsque TEX lit cet argument, il lit le texte se trouvant à intérieur des accolades et dépouille donc l'argument de ses éventuelles accolades extérieures.

21 - Règle
Lorsque des accolades délimitent un argument d'une macro, elles ne jouent pas le rôle de délimiteur de *groupe*.

■ **Exercice 22**
Si on les considère comme arguments d'une macro, « a » et « {a} » sont-ils différents ? Et si oui, en quoi ?

□ **Solution**
Non, s'ils sont les arguments d'une macro, « a » et « {a} » sont des arguments rigoureusement identiques et donc, si \foo est une macro admettant un seul argument, il sera équivalent d'écrire

- \foo␣a
- \foo{a}

En revanche, si l'on souhaite donner la macro \bar comme argument, il y aura une différence entre

- \foo\bar
- \foo{\bar}

Dans le premier cas, si un espace est à suivre, il sera ignoré tandis qu'il sera bien pris en compte dans le second cas puisque venant à la suite d'une accolade et non pas d'une séquence de contrôle.

Certains prennent l'habitude de ne jamais entourer d'accolades un argument constitué d'une seule unité lexicale. Il reste à savoir si cette habitude est bonne ou pas… Je pencherais plutôt pour dire qu'elle est bonne puisqu'elle facilite la lisibilité en ne surchargeant pas le code d'accolades inutiles. Cependant, la présentation du code étant souvent une affaire de gouts personnels, d'autres trouveraient des arguments imparables pour préférer le contraire ! ■

Pour définir une commande qui admet des arguments, on utilise le token « # », de catcode 6, dit « token de paramètre » que l'on fait suivre d'un entier qui représente le numéro de l'argument. Cet entier doit être compris entre 1 et 9, ce qui signifie qu'une macro ne peut admettre que 9 arguments au maximum.

Voici une macro « \hello », très simple, qui admet deux arguments et qui affiche « Bonjour ⟨*argument 1*⟩ et ⟨*argument 2*⟩. » et qui compose le paragraphe courant à l'aide de \par :

Code n° II-33

```
\def\hello#1#2{Bonjour #1 et #2.\par}% définition
\hello ab% #1=a et #2=b
\hello a b% #1=a et #2=b
\hello{foo}{bar}% #1=foo et #2=bar
\hello foobar% #1=f et #2=o
% (le reste "obar" est lu après que la macro est terminée)
```

```
Bonjour a et b.
Bonjour a et b.
Bonjour foo et bar.
Bonjour f et o.
obar
```

Comme on le constate, écrire

```
\def\hello#1#2{Bonjour #1 et #2.\par}
```

signifie que la commande \foo admet deux arguments qui seront écrits « #1 » et « #2 » dans le texte de remplacement de la macro. Lorsque TEX va rencontrer \hello, il lira *aussi* les deux arguments qui suivent la commande. Il remplacera ces trois choses par le texte de remplacement de la macro où chaque #1 sera l'exact contenu du premier argument et chaque #2 celui du second.

Lorsqu'on utilise \def pour définir une macro, ce qui est situé entre le nom de la macro et l'accolade ouvrante qui délimite son texte de remplacement est appelé « texte de paramètre ». Dans l'exemple ci-dessus, ce texte de paramètre est #1#2 et spécifie que la commande \foo admet 2 arguments. Dans le texte de paramètre, les entiers qui suivent les caractères # doivent commencer à 1 pour le premier et aller en croissant de un en un. Il faut également que les arguments rencontrés dans le texte de remplacement existent *aussi* dans le texte de paramètre sinon, TEX émet un message d'erreur « `Illegal parameter number in definition of` \⟨*macro*⟩ [1] ». On ne peut donc pas écrire

```
\def\foo#1#2{Bonjour #1, #2 et #3}
```

car l'argument #3 n'est pas spécifié dans le texte de paramètre.

22 - Règle

Une macro peut accepter de 0 à 9 arguments.

Pour le spécifier, on indique juste après \def, dans le « texte de paramètre », les arguments les uns à la suite des autres sous la forme « #1 », « #2 », et ainsi de suite jusqu'au nombre d'arguments que l'on veut donner à la macro.

Le texte de paramètre est strict sur l'ordre des arguments, mais en revanche, il n'y a aucune contrainte sur l'ordre et l'existence des arguments dans le texte de remplacement de la macro. On peut donc les y écrire dans l'ordre que l'on veut, autant de fois que l'on souhaite, voire ne pas en écrire certains.

1. Nombre de paramètres incorrect dans la définition de \⟨*macro*⟩.

■ **Exercice 23**

Comment s'y prendre pour qu'une commande traite plus de 9 arguments, par exemple 10 ?
Programmer une macro `\tenlist` qui lit 10 arguments (qui peuvent être par exemple les 10 premières lettres de l'alphabet) et les affiche de cette façon :

(a,b,c,d,e,f,g,h,i,j)

□ **Solution**

On ne peut pas, *stricto sensu*, programmer une macro qui admet 10 arguments, c'est une limitation intrinsèque à TEX. Mais rien n'empêche d'en programmer une qui admet 9 arguments, traiter ces arguments comme on l'entend et une fois fini, passer la main à une autre commande qui admettra un argument, le lira et le traitera à son tour.
On va donc écrire :

```
\def\tenlist#1#2#3#4#5#6#7#8#9{%
    (#1,#2,#3,#4,#5,#6,#7,#8,#9\endlist
}
\def\endlist#1{,#1)}
```

Lorsque TEX rencontrera la macro `\tenlist`, il l'absorbera ainsi que les 9 arguments suivants (et uniquement eux) et il aura donc avancé juste devant le 10e argument. Il procèdera à la substitution de la macro et ses arguments par le texte de remplacement et poursuivra sa lecture en tenant compte de la substitution. Ce texte de remplacement va procéder à l'affichage des 9 premiers arguments en les séparant par une virgule, puis TEX rencontrera à la fin du texte de remplacement la macro `\endlist`. Comme elle n'est suivie de rien dans le texte de remplacement, elle ira chercher son argument plus loin dans le code, et ce qu'il y a après justement, c'est le 10e argument.

Code no II-34

```
\def\tenlist#1#2#3#4#5#6#7#8#9{(#1,#2,#3,#4,#5,#6,#7,#8,#9\endlist}
\def\endlist#1{,#1)}
Une liste \tenlist abcdefghij de lettres.
```

Une liste (a,b,c,d,e,f,g,h,i,j) de lettres.

■

Voici l'exemple de la macro `\foo` que l'on a défini précédemment et que l'on met à l'épreuve dans différents cas :

Code no II-35

```
\def\foo#1#2{Bonjour #1 et #2.\par}
\begingroup\tt% passer en fonte à chasse fixe
a) \foo{monsieur}{madame}
b) \foo{{\bf toi}}{moi}
c) \foo{}{{}}
d) \foo{ }{ }
e) \foo{$\pi$} {$\delta$}
f) \foo ab
g) \foo X      Y
\endgroup% fin de la fonte à chasse fixe
```

a) Bonjour monsieur et madame.
b) Bonjour **toi** et moi.
c) Bonjour et .
d) Bonjour et .
e) Bonjour π et δ.
f) Bonjour a et b.
g) Bonjour X et Y.

Le cas b contient un argument lui-même entre accolades, c'est-à-dire dans un groupe. En effet, si les accolades intérieures n'existaient pas, cet argument serait « `\bf moi` » et la macro `\bf` qui ordonne le passage en fonte grasse ne serait pas contenue dans un groupe pour en limiter sa portée. Tel qu'il est écrit à la ligne 4, le second argument est donc `{\bf moi}` et donc le texte de remplacement de `\foo{toi}{{\bf moi}}` est :

```
Bonjour toi et {\bf moi}.\par
```

On remarque ensuite au cas c que les deux arguments ne produisent rien. Le premier parce que c'est un argument vide et le second parce que c'est `{}` qui est un groupe vide qui lui non plus ne produit non plus aucun affichage. Pour mieux comprendre ce qui se passe à cette ligne 5, le texte de remplacement de `\foo{}{{}}` est :

```
Bonjour  et {}.\par
```

L'argument `#1`, qui est vide, se trouve entre les deux premiers espaces qui par conséquent deviennent des espaces consécutifs, mais ils ne le sont pas *dans le code source* ! La règle de TEX concernant les espace consécutifs ne s'applique donc pas comme on peut l'observer à l'affichage.

23 - Règle
Si l'argument ⟨*i*⟩ d'une macro est vide, cela équivaudra dans le texte de remplacement de la macro à ce que #⟨*i*⟩ n'existe pas.

Le sens d'un argument « vide » n'est pas évident. On entend ici que l'argument vide est ce qui est entre les accolades dans « `{}` ». Si l'on préfère, un argument vide n'est constitué d'aucun token.

Ce qui se passe au cas d est tout aussi intéressant. Les deux arguments sont des espaces *entre accolades*, ils sont donc pris en compte. Le texte de remplacement est ici :

```
Bonjour␣□␣et␣□.
```

Les arguments `#1` et `#2`, qui sont des espaces, ont été encadrés. L'espace du code « ␣ », suivi de l'espace qui provient de l'argument □, ne sont pas des espaces consécutifs *du code*. La règle qui veut que deux espaces consécutifs du code n'en font qu'un *ne s'applique pas*. À l'affichage, on a donc *trois* espaces entre « Bonjour » et « et » puis deux entre « et » et le point final.

Lorsque TEX lit le texte de paramètre puis le texte de remplacement d'une macro pour la définir et stocker les tokens qui résultent de cette lecture, il entre dans ce qu'on appelle la *lecture à grande vitesse*. Très peu de vérifications sont faites à ce moment :

1. dans le texte de paramètre, la déclaration des arguments #⟨*chiffre*⟩ n'est acceptée que si les ⟨*chiffres*⟩ commencent à 1 et vont en croissant de un en un ;
2. une occurrence de #⟨*chiffre*⟩ dans le texte de remplacement, symbolisant l'emplacement d'un argument, est acceptée uniquement si cet argument a été déclaré dans le texte de paramètre ;

3. TEX vérifie que les macros rencontrées dans le texte de remplacement ne sont pas déclarées « \outer » (voir page 294) ;
4. la lecture à grande vitesse s'arrête lorsque l'accolade ouvrante marquant le début du texte de remplacement est équilibrée par une accolade fermante. À ce moment, le stockage du texte de remplacement sous forme de tokens a lieu dans la mémoire de TEX.

L'argument d'une commande ne peut pas contenir la primitive \par. Cette interdiction est en fait un mécanisme de sécurité pour que, si jamais on a oublié de fermer l'accolade qui marque la fin d'un argument, TEX ne tente pas de lire indéfiniment (jusqu'à la fin du document) cet argument : au premier \par rencontré, la compilation est stoppée avec affichage d'un message d'erreur, ce qui limite la zone où l'on doit chercher où il manque une accolade fermante dans le code source. Il est parfois nécessaire de passer outre cette protection si l'on souhaite écrire une macro dont un argument est susceptible de contenir \par. Pour déclarer une macro de ce type, il faut faire précéder \def de la primitive \long. Dans ce cas, on dit que la macro est « longue » par opposition aux autres macros « courtes ».

Il faut noter que si on définit un alias pour \par comme plain-TEX le fait avec \let\endgraf=\par, alors, l'alias \endgraf est permis dans l'argument d'une macro même si celle-ci n'est pas déclarée \long. En effet, les arguments d'une macro sont eux aussi lus à grande vitesse avant d'être insérés sous forme de tokens aux endroits qui leur sont réservés dans le texte de remplacement. Lors de cette lecture à grande vitesse, TEX ne tient pas compte de la \let égalité ; il ne fait que chercher \par. L'alias \endgraf est parfois bien pratique puisqu'avec cet artifice, tout se passe comme si l'on pouvait mettre un \par dans les macros courtes sans avoir besoin de les réécrire pour qu'elles soient longues.

24 - Règle

Une macro ne peut admettre la primitive \par dans aucun de ses arguments. Elle admet en revanche toute séquence de contrôle \let-équivalente à \par, par exemple \endgraf, définie par plain-TEX.

Pour définir une macro capable d'accepter \par dans ses arguments, on doit faire précéder \def de la primitive \long.

2.2. Perte d'informations lors de la « tokénization »

Examinons maintenant comment et quand sont lues les « composantes » d'une macro, à savoir son texte de remplacement et ses arguments, et surtout quelles sont les conséquences de cette lecture.

Tout d'abord, il est utile de comprendre que plusieurs moments différents interviennent lorsqu'on utilise une macro :

- le moment où elle est définie ;
- le (ou les) moment(s) où elle est appelée.

25 - Règle

Lorsqu'une macro est *définie*, les arguments figurant dans le texte de remplacement sous la forme #⟨*chiffre*⟩ (où « # » symbolise un token ayant le catcode 6 au moment

de la définition) sont des endroits où TEX mettra les arguments lus plus tard lorsque la macro sera appelée.
Lors de sa lecture à grande vitesse, le reste du texte de remplacement est « tokénizé » c'est-à-dire que les caractères composant le code source sont transformés en tokens et les catcodes qui leur sont assignés obéissent au régime en cours lorsque la définition a lieu. Ces tokens sont stockés dans la mémoire de TEX.

Lorsqu'une macro est *appelée* et qu'elle *lit* ses arguments, ceux-ci sont lus à grande vitesse et sont tokénizés selon le régime de catcode en vigueur cette macro est appelée (et qui peut donc différer de celui en cours lors de sa définition). Ils sont ensuite insérés aux endroits où #⟨*chiffre*⟩ était écrit dans le texte de remplacement. Les tokens du texte de remplacement ne sont pas modifiés et restent ce qu'ils étaient lors de la définition.

Plus généralement, quels que soient le moment et les circonstances, lorsque TEX lit du code source, les caractères du code source sont transformés en tokens. Cette transformation est faite selon le régime de catcode en vigueur au moment de la lecture et des règles spécifiques à chaque catégorie sont alors appliquées. Il est important de savoir que transformer des caractères en tokens provoque d'irréversibles pertes d'informations. En voici quelques-unes :

1. plusieurs espaces consécutifs sont lus comme un espace unique ;
2. les espaces au début d'une ligne du code source sont ignorés ;
3. les espaces qui suivent une séquence de contrôle sont ignorés ;
4. un caractère de catcode 5 (en général le retour charriot « `^^M` ») est lu comme un espace ;
5. deux caractères consécutifs de catcode 5 sont lus comme la primitive `\par` ;
6. deux caractères identiques de catcode 7 (en général le caractère de mise en exposant « ^ ») suivis d'un caractère ou de deux chiffres hexadécimaux sont lus comme un caractère unique ;
7. le caractère de code `\endlinechar` est inséré à chaque fois qu'une fin de ligne de code source est rencontrée ;
8. dans une ligne de code source, tout ce qui se trouve après un caractère de catcode 14 est ignoré (le caractère de commentaire est en général « `%` »).

Par exemple, supposons qu'une macro `\foo` est définie de cette façon sous le régime de catcode par défaut :

```
\def\foo{Programmer  en \TeX {} est   facile}
```

Par la suite, en examinant le texte de remplacement de `\foo`, il est impossible de savoir combien d'espaces il y avait après « `est` » dans le code source. Il est également impossible de savoir si `\TeX` était suivi d'un espace, de plusieurs ou d'aucun. Et donc, une façon *strictement équivalente* de définir `\foo` aurait été :

```
\def\foo{Program^^6der en   \TeX{} est facile}
```

La primitive `\meaning` peut nous aider à le constater.

Code n° II-36

```
\def\foo{Programmer  en \TeX {} est   facile}
\meaning\foo\par
```

```
\def\foo{Program^^6der en   \TeX{} est facile}
\meaning\foo
```

```
macro:->Programmer en \TeX {} est facile
macro:->Programmer en \TeX {} est facile
```

26 - Règle

Lorsque TEX lit du code source, d'irréversibles pertes d'informations ont lieu lorsque des *caractères* sont transformés en *tokens*.

Cette règle s'applique bien évidemment lorsque TEX lit du code pour le stocker (que ce soit dans une macro ou dans un registre de tokens) ou pour l'exécuter.

■ **Exercice 24**

Donner un autre exemple de perte d'information lorsque TEX lit du code.

□ **Solution**

Si deux caractères ont le catcode 0 (par exemple « \ » et « | »), voici deux façons parfaitement équivalentes de définir une macro `foo` :

```
\def\foo{Programmer en |TeX{} est facile}
```

et

```
\def\foo{Programmer en \TeX{} est facile}
```

Si par ailleurs, "<" et ">" ont comme catcode 1 et 2, alors, cette façon est également équivalente :

```
\def\foo<Programmer en \TeX<> est facile}
```

Une autre perte d'information est que les caractères ayant un catcode égal à 9 sont ignorés. ■

■ **Exercice 25**

Comment faudrait-il s'y prendre pour que la lecture du code soit *réversible*, c'est-à-dire pour qu'aucune perte d'information n'ait lieu ?

□ **Solution**

Il faudrait que le régime de catcode en vigueur soit très spécifique et que tous les caractères ayant un catcode de 0, 1, 2, 5, 6, 9, 10, 13, 14 et 15 aient un catcode de 12. On peut aussi carrément utiliser le régime où *tous* les octets ont un catcode égal à 12. ■

2.3. Quelques commandes utiles

Prenons maintenant le temps de définir des commandes utiles en programmation. Certes, il est encore tôt pour comprendre leur intérêt, mais nous le verrons plus tard, leur emploi est fréquent.

La plus simple de toutes est pourtant très utile : elle lit un argument qu'elle fait disparaitre. Appelons-la [2] `\gobone*` pour « gobble one » qui signifie « mange un ». Le code qui permet sa définition est d'une simplicité extrême :

2. Cette macro est appelée `\@gobble` dans le noyau LATEX.

```
\def\gobone#1{}
```

Le texte de remplacement est vide, ce qui revient à dire que lorsque TEX rencontre \gobone, il lit également l'argument qui suit et remplace le tout par le texte de remplacement, c'est-à-dire rien.

Pour rester dans la simplicité, voici une commande qui admet un argument et dont le texte de remplacement est cet argument. Appelons-la [3] \identity* :

```
\long\def\identity#1{#1}
```

Voici maintenant deux commandes très utilisées. Elles admettent deux arguments. Le texte de remplacement de \firstoftwo* est son premier argument et celui \secondoftwo* est son second argument [4]

```
\long\def\firstoftwo#1#2{#1}
\long\def\secondoftwo#1#2{#2}
```

On pourrait aller plus loin et définir, pourquoi pas, \thirdofsix qui irait chercher le 3e argument d'une série de 6 :

```
\long\def\thirdofsix#1#2#3#4#5#6{#3}
```

Mais ces variantes sont un peu inutiles, car ce sont les macros \firstoftwo et \secondoftwo qui sont le plus souvent utilisées. Pourquoi est-il plus utile et fréquent de choisir entre *deux* arguments ? L'explication tient au fait que ces macros sont utilisées après un test pour choisir une des alternatives, l'une correspondant au fait que le test est vrai et l'autre qu'il est faux.

■ Exercice 26

Pour manger les deux arguments {⟨*arg1*⟩} et {⟨*arg2*⟩}, le code suivant ne fonctionne pas :

```
\gobone\gobone{⟨arg1⟩}{⟨arg2⟩}
```

Pourquoi ? Comment faudrait-il programmer la macro \gobtwo* pour qu'elle mange et fasse disparaitre les deux arguments qui la suivent ?

□ Solution

TEX lit le code *de gauche à droite*, il remplace donc dans un premier temps \gobone et son argument, le deuxième \gobone, par le texte de remplacement de \gobone, c'est-à-dire rien. Les deux \gobone disparaissent et il reste les deux arguments entre accolades qui seront affichés tous les deux !

La macro \gobtwo se programme ainsi : \def\gobtwo#1#2{} ■

■ Exercice 27

Qu'obtient-on à l'affichage si on écrit :

```
\gobone1\secondoftwo2{3\gobtwo}\firstoftwo45\secondoftwo{67}{\gobone890}
```

3. Son nom dans le noyau LATEX est \@firstofone
4. Elles sont appelées \@firstoftwo et \@secondoftwo dans le noyau LATEX.

□ **Solution**

Procédons de gauche à droite, comme TEX :

- « `\gobone 1` » disparait ;
- « `\secondoftwo2{3\gobtwo}` » a comme texte de remplacement son 2e argument qui est « `3\gobtwo` ». « 3 » est donc affiché et le `\gobtwo` va manger les 2 arguments qui le suivent. Ils se trouvent en dehors de l'argument en cours et ce sont « `\firstoftwo4` » qui sont donc mangés ;
- « 5 » est affiché ;
- « `\secondoftwo{67}{\gobone890}` » a pour texte de remplacement son 2e argument qui est « `\gobone890` ». Le `\gobone` mange le « 8 » qui disparait et il reste donc les tokens « 90 » qui seront affichés.

On obtient donc « `3590` » ■

■ **Exercice 28**

On constate que les macros `\gobone` et `\secondoftwo` donnent les mêmes résultats :

Code n° II-37

```
a) \gobone XY - \secondoftwo XY\par
b) \gobone{ab}{xy} - \secondoftwo{ab}{xy}\par
c) \gobone{123}4 - \secondoftwo{123}4\par
d) \gobone A{BCD} - \secondoftwo A{BCD}
```

a) Y - Y
b) xy - xy
c) 4 - 4
d) BCD - BCD

Existe-t-il une différence entre les macros `\gobone` et `\secondoftwo` et si oui, laquelle ?

□ **Solution**

Il y a une différence, et de taille ! On peut le voir sur cet exemple :

Code n° II-38

```
\gobone     {a}{\catcode`\~12 Le <<~>> est un espace ?} Et ici <<~>> ?

\secondoftwo{a}{\catcode`\~12 Le <<~>> est un espace ?} Et ici <<~>> ?
```

Le «~» est un espace ? Et ici « » ?
Le « » est un espace ? Et ici «~» ?

À la première ligne, lorsque `\gobone` est développée, « `\gobone{a}` » disparait et il reste ceci qui n'a pas encore été lu :

```
{\catcode`\~12 Le <<~>> est un espace ?} Et ici <<~>> ?
```

Il est clair que la modification de catcode a lieu *à l'intérieur d'un groupe* et que dans ce groupe, ~ devient un caractère « autre » que l'on peut afficher tel quel. En revanche, une fois sorti du groupe, il reprend son catcode naturel de 13 et redevient donc actif pour afficher une espace insécable.

Tout est différent dans la deuxième ligne. La macro `\secondoftwo` lit ses *deux* arguments et « redonne » le second une fois qu'il a été lu. Voici donc ce qu'il reste après que `\secondoftwo` ait été remplacé pare son 2e argument :

```
\catcode`\~12 Le <<~>> est un espace ? Et ici <<~>> ?
```

Le 2e argument, ici encadré et qui s'étend jusqu'au premier « ? » a été lu par \secondoftwo et donc, le catcode de « ~ » est figé à 13. Par conséquent, « ~ » gardera dans tout l'argument encadré le catcode 13 qu'il avait lorsque \secondoftwo a été appelé. L'ordre \catcode`\~12 rencontré dans l'argument ne pourra agir qu'après cet argument : ~ prendra donc le catcode 12 après la fin de l'argument et ce en-dehors de tout groupe. Il gardera donc ce catcode indéfiniment c'est-à-dire jusqu'à ce qu'il soit volontairement modifié.

Pour résumer :

- de « \gobone{⟨*a*⟩}{⟨*b*⟩} », il reste {⟨*b*⟩}qui n'est pas encore lu ;
- tandis que \secondoftwo{⟨*a*⟩}{⟨*b*⟩} lit le second argument, le dépouille de ses accolades et le tout devient donc ⟨*b*⟩ où les catcodes sont figés.

■

■ Exercice 29

Le but est d'afficher l'énoncé et la correction d'une courte interrogation de grammaire élémentaire, tous deux stockés dans la macro \interro ainsi définie :

```
\def\interro{J'ai ét\?{é} invit\?{é} à goût\?{er} chez Max.
             On s'est bien amus\?{é} et ensuite, il a fallu
             rentr\?{er}.}
```

Écrire une macro \? qui admet un argument. Cet argument sera affiché tel quel si la commande \visible a été rencontrée auparavant, et si c'est la commande \cache, cet argument sera remplacé par « ... ». Les commandes \cache et \visible n'admettent pas d'argument.

On écrira « \cache\interro » ou « \visible\interro » selon que l'on souhaite afficher l'énoncé ou la correction.

□ Solution

L'idée est que le texte de remplacement de \?{⟨*argument*⟩} doit être

\choix{⟨*argument*⟩}{...}

où \choix sera rendu égal à \firstoftwo par \visible et à \secondoftwo par \cache :

Code n° II-39

```
\def\visible{\let\choix=\firstoftwo}
\def\cache{\let\choix=\secondoftwo}
\def\?#1{\choix{#1}{...}}%
\def\interro{J'ai ét\?{é} invit\?{é} à gout\?{er} chez Max. Après s'être
            bien amus\?{é}, nous avons rang\?{é} et il a fallu rentr\?{er}.}
Compléter avec << é >> ou << er >> :\par
\cache\interro\par\medskip
Correction :\par
\visible\interro
```

Compléter avec « é » ou « er » :
J'ai ét... invit... à gout... chez Max. Après s'être bien amus..., nous avons rang... et il a fallu rentr....

Correction :
J'ai été invité à gouter chez Max. Après s'être bien amusé, nous avons rangé et il a fallu rentrer.

■

2.4. Définitions imbriquées

À l'exercice précédent, on aurait pu procéder de façon plus « économique », sans avoir besoin de définir une macro auxiliaire \choix. La macro \visible aurait pu être programmée pour définir \? \let-égale à \identity (qui affiche son argument tel quel) alors que la macro \cache aurait défini la macro \? de cette façon :

```
\def\?#1{...}
```

Mais si on écrit

```
\def\cache{\def\?#1{...}}
```

TEX émet un message d'erreur : « `Illegal parameter number in definition of \cache` ». L'explication est que l'argument #1 est compris comme étant celui de la macro *extérieure* \cache et non pas comme celui de la macro intérieure \?. L'erreur survient, car la macro extérieure \cache est définie comme n'ayant *aucun* argument.

Lorsque des définitions de macros sont imbriquées, la règle veut que pour distinguer les arguments des macros, il faut doubler les tokens de paramètre # à chaque niveau d'imbrication supplémentaire. Ainsi, la bonne façon de définir \cache est :

```
\def\cache{\def\?##1{...}}
```

Voici le code complet :

Code n° II-40

```
\def\visible{\let\?=\identity}
\def\cache{\def\?##1{...}}
\def\interro{J'ai ét\?{é} invit\?{é} à goût\?{er} chez Max. On s'est
             bien amus\?{é} et ensuite, il a fallu rentr\?{er}.}
Compléter avec << é >> ou << er >> :\par
\cache\interro\par\medskip
Correction :\par
\visible\interro
```

Compléter avec « é » ou « er » :
J'ai ét... invit... à goût... chez Max. On s'est bien amus... et ensuite, il a fallu rentr....

Correction :
J'ai été invité à goûter chez Max. On s'est bien amusé et ensuite, il a fallu rentrer.

27 - Règle

Lorsqu'on définit une macro « fille » dans le texte de remplacement d'une macro « mère », les # concernant les arguments de la macro fille sont doublés, aussi bien dans le texte de paramètre que dans le texte de remplacement.

Ils seront doublés à nouveau à chaque imbrication supplémentaire pour éviter de confondre les arguments des macros imbriquées.

À la lumière de ce qui vient d'être dit, examinons maintenant le code suivant :

```
\def\makemacro#1#2{\def#1##1{##1 #2}}
```

On constate que \makemacro admet deux arguments et que le premier est nécessairement une séquence de contrôle puisque cet argument #1 se trouve juste après le \def dans le texte de remplacement.

Faisons mentalement fonctionner la macro \makemacro lorsqu'on l'appelle de cette façon : \makemacro\foo{BAR}. Comme on le sait, le texte de remplacement de ce code va être obtenu en remplaçant chaque #1 par \foo et chaque #2 par BAR ce qui donne :

```
\def\foo##1{##1 BAR}
```

Puisqu'on examine le code contenu dans le texte de remplacement, il devient inutile de doubler les tokens de paramètre et donc, tout se passe donc comme si la macro \foo était définie par :

```
\def\foo#1{#1 BAR}
```

Le code obtenu ne présente pas de difficulté particulière, l'argument #1 de \foo est placé devant l'argument #2 de \makemacro qui est BAR. Par exemple, si l'on écrivait « \makemacro\mai{mai 2014} », alors par la suite, \mai{10} aurait comme texte de remplacement « 10 mai 2014 ».

Code n° II-41

```
\def\makemacro#1#2{\def#1##1{##1 #2}}
\makemacro\LL{Lamport}
\makemacro\juin{juin 2014}
Paris, le \juin{3}.\medbreak
\LL{Cher Monsieur},\smallbreak
Vous êtes convoqué à un examen sur \TeX{} le \juin{21} à
9h00 précise dans le bureau de D. Knuth.\smallbreak
Veuillez croire, \LL{Monsieur}, en mes sentiments \TeX iens.
```

Paris, le 3 juin 2014.

Cher Monsieur Lamport,

Vous êtes convoqué à un examen sur TEX le 21 juin 2014 à 9h00 précise dans le bureau de D. Knuth.

Veuillez croire, Monsieur Lamport, en mes sentiments TEXiens.

2.5. Programmer un caractère actif

Le niveau de TEXnicité atteint dans cette section est plutôt relevé. On pourra donc, lors d'une première lecture, se dispenser de se plonger dans cette section, quitte à y revenir plus tard.

Avant de rentrer dans le vif du sujet, rappelons ce qu'est réellement un caractère. Aux yeux de TEX lorsqu'il le lit, un caractère (c'est-à-dire un token n'étant pas une séquence de contrôle) est constitué de :

- un code dit « code de caractère » qui représente l'ordre du caractère dans la table des caractères. Par exemple, le caractère « A » a le code de caractère 65 tandis que « ^ » a le code 94. Il y a 256 codes [5] possibles ;
- un code dit « code de catégorie » ou « catcode » qui met le caractère dans une des 16 catégories et lui confère les propriétés communes à tous les caractères de cette catégorie. Ainsi, « A » a le code de catégorie 11, et « ^ » le code de catégorie 7.

5. Pour les moteurs 8 bits, il y a effectivement 256 codes possibles. En revanche, pour les moteurs UTF8, il y a en a bien plus.

Chaque caractère lu possède donc intrinsèquement une *paire de codes*. On peut convenir d'une notation et les mettre entre parenthèses, le premier étant le code de caractère et le second le catcode. Cela donnerait : A(65 ; 11) et ^(94 ; 7).

L'objet ici est de découvrir comment on peut programmer une macro – appelons-la \defactive* – qui rend actif un caractère choisi par l'utilisateur et qui lui donne un texte de remplacement donné. Ainsi, « \defactive W{wagons} » fera qu'après cet appel, le caractère « W » sera actif et son texte de remplacement sera « wagons ».

Comme nous l'avons vu, si on modifie un catcode dans le texte de remplacement d'une macro, la modification n'entrera en vigueur qu'après ce texte de remplacement. En effet, la règle vue à la page 28 veut que lorsqu'un argument est lu, les catcodes des tokens qui le composent sont figés. Il est donc exclu de programmer la macro de cette façon

```
\def\defactive#1#2{\catcode`\#1=13 \def#1{#2}}
```

car, malgré l'ordre \catcode`\#1=13, le catcode de #1 ne serait pas modifié à l'intérieur du texte de remplacement de \defactive. Par exemple, si #1 est « W », son catcode restera 11 (celui que W a par défaut) et TEX va échouer lorsqu'il rencontrera \def W{wagons}, car « W » ne sera pas encore actif.

Évidemment, on l'a vu aussi, on pourrait utiliser un \gdef et agir dans un groupe où l'on aurait déjà modifié le catcode de « W » à 13. Cette méthode est possible lorsqu'on connaît *à l'avance* le caractère à rendre actif. Or, ce n'est pas le cas ici puisque l'argument #1 de la macro est choisi par l'utilisateur.

Intéressons-nous par exemple à la primitive \lowercase car les fonctionnements de \lowercase et \uppercase sont symétriques : il suffit de comprendre comment fonctionne l'une pour comprendre comment fonctionne de l'autre.

Avant d'entrer dans le vif du sujet, il faut mentionner qu'en plus du code de catégorie et du code de caractère, chaque caractère possède deux autres codes :

- le code minuscule accessible par la primitive \lccode ;
- le code majuscule accessible par la primitive \uccode.

Par exemple, le code de caractère de la lettre « A » est 65 et son code minuscule est 97 qui est celui de la lettre « a ». Il en est ainsi pour chaque lettre majuscule de l'alphabet qui possède un code minuscule correspondant à celui de la lettre minuscule. Grâce à la primitive \the ou \number, on peut afficher les codes de caractère et les codes minuscules des lettres A, B, et même de a, b. On va utiliser ici une macro \showcodes pour alléger le code :

Code n° II-42

```
\def\showcodes#1{\string#1 : $\number`#1 \rightarrow \number\lccode`#1$}
\showcodes A\qquad \showcodes B\qquad
\showcodes a\qquad \showcodes b\qquad \showcodes ^
```

A : 65 → 97 B : 66 → 98 a : 97 → 97 b : 98 → 98 ^ : 94 → 0

On observe que logiquement, le code minuscule de « a » est le sien propre. Quant au code minuscule de « ^ », il est égal à 0 ce qui signifie que ce caractère n'a pas de code minuscule prédéfini.

28 - Règle

La primitive \lowercase doit être suivie de son argument entre accolades et cet argument, constitué d'un texte dont les accolades sont équilibrées, est lu de la façon suivante :

- les séquences de contrôles restent inchangées ;
- pour chaque caractère, le code de caractère est remplacé par le code minuscule, étant entendu que le code de catégorie *ne change pas*.
  Si le code minuscule d'un caractère est nul, ce caractère n'est pas modifié par \lowercase.

Voici la primitive \lowercase en action :

Code n° II-43

```
\lowercase{CACAO foobar}\par
\lowercase{\def\foo{Bonjour}}% définit une commande \foo
\foo % la commande définie précédemment
```

cacao foobar
bonjour

On peut détourner la primitive \lowercase de son usage premier qui est de mettre un texte en minuscule. Pour cela, il suffit de modifier le code minuscule d'un caractère. Supposons que nous souhaitions que le code minuscule de « A » ne soit plus 97 qui est celui de la lettre « a », mais 42 qui est celui de « * ». Pour cela, il va falloir modifier le \lccode de « A » à l'intérieur d'un groupe semi-simple, puisqu'il est toujours prudent de confiner les modifications lorsqu'on touche aux mécanismes internes de TEX :

Code n° II-44

```
\begingroup
  \lccode`A=`* % change le code minuscule de A
  \lowercase{CACAO ABRACADABRA}\par
\endgroup
\lowercase{CACAO ABRACADABRA}
```

c*c*o *br*c*d*br*
cacao abracadabra

Les « * » que l'on obtient ont le catcode de A, c'est-à-dire 11.

29 - Règle

Si l'on écrit « \lccode⟨*entier1*⟩=⟨*entier2*⟩ » alors, dans l'argument de \lowercase, tous les caractères de code ⟨*entier1*⟩ seront lus comme des caractères de code ⟨*entier2*⟩ tout en gardant leur catcode originel.

En pratique, si l'on écrit

\lccode`\⟨*car1*⟩=`\⟨*car2*⟩

puis

\lowercase{⟨*texte*⟩}

alors, dans le ⟨*texte*⟩, tous les ⟨*car1*⟩ sont lus comme ⟨*car2*⟩, mais en conservant le catcode de ⟨*car1*⟩.

Nous sommes prêts pour écrire la macro \defactive et nous allons exploiter le fait que le caractère « ~ » est naturellement actif. Grâce à la primitive \lowercase, on va le transformer dynamiquement pour qu'il devienne le caractère #1 *actif* de notre macro \defactive :

```
\def\defactive#1#2{%
    \catcode`#1=13 % #1 sera actif après la fin
                   % du texte de remplacement
    \begingroup
    \lccode`~=`#1  % dans \lowercase, changer les ~ (actifs)
                   % en "#1", ceux-ci étant actifs
    \lowercase{\endgroup\def~}{#2}%
}
```

- « \catcode`#1=13 » rend le caractère #1 actif, mais comme nous l'avons vu, cette modification n'entrera en vigueur qu'après être sorti du texte de remplacement ;
- Le \endgroup n'étant pas modifié par \lowercase, le groupe semi-simple est donc bien refermé *avant* d'effectuer la définition de #1 ;
- on peut remarquer que l'argument #2 se trouve *en dehors* de l'argument de \lowercase car sinon, toutes les lettres majuscules de #2 seraient transformées en minuscules.

  D'ailleurs, comme {#2} est à la fin du texte de remplacement de la macro, il devient inutile de le lire puisqu'il sera lu juste après. On peut optimiser la macro de cette façon :

```
\def\defactive#1{%
    \catcode`#1=13
    \begingroup
    \lccode`~=`#1
    \lowercase{\endgroup\def~}}
```

Voyons cette macro à l'œuvre dans un exemple :

Code n° II-45

```
\def\defactive#1{%
  \catcode`#1=13
  \begingroup
  \lccode`~=`#1
  \lowercase{\endgroup\def~}%
}
\begingroup% les modifications restent locales
  \defactive W{wagons}% définit le caractère actif "W"
  Les W constituent les trains.
\endgroup %fermeture du groupe, "W" redevient une lettre
\par Les W sont des lettres
```

Les wagons constituent les trains.
Les W sont des lettres

■ Exercice 30

Supposons que « W » soit rendu actif à l'aide de la macro \defactive pour lui donner le texte de remplacement « wagons ». Imaginons ensuite que l'on rende « W » non actif avec \catcode‘\W=11.

Question : si l'on rend à nouveau W actif, le texte de remplacement précédemment défini existe-t-il encore ou faut-il redéfinir son texte de remplacement ?

□ Solution

Il suffit de faire l'expérience :

Code n° II-46

```
\defactive W{wagon}
1) Les W constituent le train.\par
2) \catcode‘W=11 Les W sont des lettres.\par
3) \catcode‘\W=13 Les W constituent-ils le train ?
```

1) Les wagon constituent le train.
2) Les W sont des lettres.
3) Les wagon constituent-ils le train ?

La réponse est donc « oui ».

Le texte de remplacement d'un caractère actif n'est pas perdu si ce caractère change de catcode et redevient actif ensuite. ■

■ Exercice 31

Décrire ce qui va se passer si on définit le caractère actif « W » de la façon suivante :

```
\defactive W{Wagon}
```

□ Solution

Tout dépend de la version de \defactive. La réponse est très différente selon que \defactive lit 1 ou 2 arguments, c'est-à-dire si elle lit le texte de remplacement du caractère actif ou pas.

Si elle le lit, le « W » de « Wagon » aura le catcode en vigueur au moment ou \defactive lit son argument, c'est-à-dire 11. Le texte de remplacement de « W » sera « Wagon » où toutes les lettres ont le catcode 11.

Code n° II-47

```
\def\defactive#1#2{%
  \catcode‘#1=13 % #1 sera actif après la fin du texte de remplacement
  \begingroup
  \lccode‘~=‘#1  % dans \lowercase, changer les ~ (actifs) en "#1", ceux-ci étant actifs
  \lowercase{\endgroup\def~}{#2}%
  }
\defactive W{Wagon}
Un W.
```

Un Wagon.

Si \defactive ne lit pas le texte de remplacement (version « optimisée »), le caractère « W » sera actif à la fin du texte de remplacement et donc, lorsque \def W lira le texte de remplacement entre accolades, le « W » de « Wagon » sera actif. On entrevoit ce qui va se passer lorsque W sera exécuté. Dans un premier temps, W va être remplacé dans la pile d'entrée par

```
Wagon
```

puis le premier « W », actif, sera exécuté et donc, nous aurons dans la pile d'entrée

```
Wagonagon
```

Le `W` actif généré sera à nouveau exécuté et ainsi de suite… Une boucle infinie va s'amorcer au cours de laquelle les caractères « agon » seront ajoutés sur la pile d'entrée à chaque boucle, provoquant rapidement le dépassement de sa capacité et générant le message d'erreur

Code n° II-48

```
\def\defactive#1{%
  \catcode`#1=13 % #1 sera actif après la fin du texte de remplacement
  \begingroup
  \lccode`~=`#1 % dans \lowercase, changer les ~ (actifs) en "#1", ceux-ci étant actifs
  \lowercase{\endgroup\def~}%
  }
\defactive W{Wagon}
Un W.
```

```
! TeX capacity exceeded, sorry [input stack size=5000]
```

La morale que l'on peut retirer de cet exercice est qu'une « optimisation », qui de prime abord peut sembler anodine (ne pas faire lire un argument par une macro), conduit à des limitations et à des erreurs de compilation qui n'existaient pas dans une version « non optimisée ». La programmation en TEX est pavée d'embuches… Il est donc nécessaire de parfaitement savoir ce que l'on fait, de connaitre le régime de catcodes en cours lorsqu'une lecture est faite et toujours rester vigilant dès lors qu'il est question de lecture d'argument. ■

Les manœuvres décrites dans cette section reposent sur la supposition que « ~ » est actif lorsque la macro `\defactive` est *définie*. Si ce n'est pas la cas et si, pour une raison ou pour une autre, « ~ » a été neutralisé et mis dans la catégorie des caractères « autres » de catcode 12, la macro `\defactive` ne pourra et provoquera une erreur de compilation lors de son exécution.

Tout ce qui a été dit avec les primitives `\lowercase` et `\lccode` est valable pour leurs symétriques, les primitives `\uppercase` et `\uccode` qui agissent sur le « code majuscule » d'un caractère. On aurait d'ailleurs tout aussi bien pu procéder avec `\uppercase` et `\uccode`.

Code n° II-49

```
\def\defactive#1#2{%
  \catcode`#1=13 % #1 sera actif après la fin du texte de remplacement
  \begingroup
  \uccode`~=`#1 % dans \uppercase, changer les ~ (actifs) en "#1", ceux-ci étant actifs
  \uppercase{\endgroup\def~}{#2}%
}
\defactive W{Wagon}
Un W.
```

```
Un Wagon.
```

Une macro à peu près similaire `\letactive`⋆⟨*token*⟩⟨*token*⟩ est facile à programmer :

Code n° II-50

```
\def\letactive#1{%
  \catcode`#1=13 % #1 sera actif après la fin du texte de remplacement
  \begingroup
  \uccode`~=`#1 % dans \uppercase, changer les ~ (actifs) en "#1", ceux-ci étant actifs
```

```
  \uppercase{\endgroup\let~}%
}
\def\W{wagon}%
\letactive X\W
Un X.
```

Un wagon.

2.6. Afficher du code tel qu'il a été écrit

Le but est d'inventer une commande qui désactive toutes les règles de TEX pour que son argument soit affiché tel qu'il a été écrit dans le code source, ce que l'on appelle du « verbatim ». Cette commande « `\litterate*` » va donc rendre inoffensifs tous les tokens de catcode autre que 11 ou 12, en changeant leur catcode pour 12. Ces tokens *spéciaux* sont ␣, \, {, }, $, &, #, $, _, % et ~. Plain-TEX définit [6] la macro `\dospecials` de cette façon :

```
\def\dospecials{\do\ \do\\\do\{\do\}\do\$\do\&%
  \do\#\do\^\do\^^K\do\_\do\^^A\do\%\do\~}
```

La macro `\do` peut être programmée comme on l'entend pour agir sur son argument. Ici, comme le but est de changer le catcode de cet argument pour 12, on va donc la programmer ainsi :

```
\def\do#1{\catcode`#1=12 }
```

et il suffira d'appeler `\dospecials` pour qu'à partir de ce moment, tous les tokens spéciaux soient inoffensifs et deviennent affichables.

En ce qui concerne la syntaxe de `\litterate`, il serait légitime d'écrire son argument entre accolades, mais cela rendrait la tâche compliquée. En effet, cette macro doit rendre tous les tokens spéciaux inoffensifs *avant* de lire son argument. Par conséquent, les accolades délimitant l'argument ne seraient pas de *vraies* accolades de catcode 1 et 2, mais des accolades de catcode 12. Pour savoir où se situe la fin de l'argument, il faudrait parcourir le code qui se trouve après `\litterate` token par token, compter les accolades ouvrantes et fermantes jusqu'à obtenir l'égalité entre ces deux quantités qui signifierait la fin de l'argument. C'est faisable, mais inutilement compliqué et implique comme limitation que le code à afficher soit équilibré en accolades ouvrantes et fermantes. Le mieux est de décider à chaque fois d'un token (de catcode 11 ou 12) et de placer l'argument entre deux occurrences de ce token [7]. On peut par exemple écrire

```
\litterate|\foo # #34{ ~} &_\ |
```

et obtenir l'affichage « `\foo # #34{ ~} &_\` ». La limitation évidente est que le token choisi comme délimiteur ne doit pas figurer dans l'argument.

Du côté de la programmation, la macro `\litterate` doit ouvrir un groupe semi-simple puis rendre les tokens spéciaux inoffensifs. À l'aide de la macro `\defactive` vue à la section précédente, elle doit aussi rendre actif le retour à la ligne « `^^M` » et lui donner comme texte de remplacement `\par \leavevmode`. Le `\leavevmode`, en quittant le mode vertical mis en place par `\par`, entre dans le mode horizontal et

6. En plain-TEX, les tokens `^^K` et `^^A` sont des équivalents pour « ^ » et « _ ».
7. C'est la syntaxe adoptée par la macro `\verb` de LATEX.

commence un nouveau paragraphe. Sans ce \leavevmode, deux retours à la ligne consécutifs seraient équivalents à deux \par et le second serait ignoré (voir règle page 18). De plus, le token frontière choisi doit aussi être rendu actif et pour devenir \let-égal à \endgroup ; ainsi, lorsque le second token frontière sera rencontré, le groupe semi-simple sera fermé et tous les catcodes seront restaurés comme avant que la macro \litterate n'entre en action.

Code n° II-51

```
\def\litterate#1{% #1=lit le token frontière choisi
  \begingroup% ouvre un groupe pour y faire les modifications
    \def\do##1{\catcode`##1=12 }% \do change le catcode de son argument à 12
    \dospecials% rend inoffensifs tous les tokens spéciaux
    \defactive\^^M{\leavevmode\par}% définit le retour charriot
    \letactive#1\endgroup% définit #1 qui sera un \endgroup
    \tt% passe en fonte à chasse fixe
}
essai 1 : \litterate*\foo # #34{ %} &_\ *
\medbreak
essai 2 : \litterate/une première ligne  ,, >>

un saut de ligne et la seconde --/
```

```
essai 1 : \foo # #34{ } &_\

essai 2 : une première ligne  „ »

un saut de ligne et la seconde –
```

L'ensemble fonctionne bien à part les espaces qui sont rendus avec le mauvais caractère et les certaines ligatures qui sont faites à l'essai n° 2.

Pour l'espace tout d'abord, cela s'explique car son catcode a été rendu égal à 12 par \dospecials et le caractère « ␣ » se trouve à la position de l'espace dans la fonte à chasse fixe. Pour contourner ce problème, il faut rendre l'espace actif après l'appel à \dospecials et lui donner comme texte de remplacement la primitive \ .

Ensuite, pour les ligatures, il faut déjà identifier quels sont les caractères doublés qui forment une ligature en fonte à chasse fixe. Ces caractères sont : `, <, >, ,, ' et -. Pour éviter qu'ils ne forment une ligature, il va falloir les rendre actifs et les faire suivre soit de « {} », soit de \relax soit encore de \kern0pt, ou plus généralement de quelque chose ne se traduisant par aucun affichage. Ainsi, ils sembleront se suivre à l'affichage, mais aux yeux de TEX, ils seront séparés par un nœud [8] qui empêchera la formation de la ligature.

Code n° II-52

```
\def\litterate#1{% #1=lit le token frontière choisi
  \begingroup% ouvre un groupe pour y faire les modifications
    \def\do##1{\catcode`##1=12 }% \do change le catcode de son argument à 12
    \dospecials% rend inoffensifs tous les tokens spéciaux
    \defactive\^^M{\leavevmode\par}% définit le retour charriot
    \defactive\ {\ }% l'espace est actif et devient "\ "
    \defactive<{\string<{}}\defactive>{\string>{}}% empêche
    \defactive-{\string-{}}\defactive`{\string`{}}% les
    \defactive,{\string,{}}\defactive'{\string'{}}% ligatures
    \letactive#1\endgroup% #1 sera un \endgroup
```

8. Un nœud est un élément, visible ou pas, qui entre dans la liste horizontale ou verticale que TEX est en train de construire.

```
    \tt% passe en fonte à chasse fixe
}
essai 1 : \litterate*\foo # #34{ %} &_\ *
\medbreak
essai 2 : \litterate/une première      ligne  ,, >>

un saut de ligne et la seconde --/
```

essai 1 : \foo # #34{ %} &_\

essai 2 : une première   ligne ,, >>

un saut de ligne et la seconde --

Chapitre 3

Arguments délimités

3.1. Théorie

Les commandes que l'on définit avec `\def` peuvent également avoir des « arguments délimités ». Ces arguments-là ne sont pas lus de la même façon que les arguments « non délimités » que nous avons vus précédemment. On indique à TEX qu'une macro a un ou des arguments délimités lorsque son texte de paramètre contient d'autres unités lexicales que #*i* où *i* est un entier de 1 à 9. Ces unités lexicales sont appelées les délimiteurs.

30 - Règle
Un argument #<*i*> est dit « délimité » si, dans le texte de paramètre, il est suivi par autre chose que #<*i*+1> ou que l'accolade « { » qui marque le début du texte de remplacement.

Voici un exemple :

```
\def\foo.#1**#2#3/{Bonjour #1, #2 et #3}
```

Ici, `#1` et `#3` sont les arguments délimités puisqu'ils sont suivis par respectivement « ** » et « / ». Par la suite, lorsque TEX va exécuter `\foo`, il s'attendra à ce qu'elle soit immédiatement suivie de « `.` » puis de quelque chose (qui sera l'argument `#1`) qui s'étendra jusqu'à la première occurrence de « ** », puis autre chose (qui deviendra les arguments `#2#3`) qui s'étendra jusqu'à la première occurrence de « / ». Ayant défini la commande `\foo` avec le code précédent, supposons que l'on écrive :

```
\foo.ab{123}c**xyz/
```

Dans ce cas, #1 sera « ab{123}c ». Quant aux tokens « xyz », ils sont à répartir entre #2 et #3 selon la règle suivante :

31 - Règle
Lorsqu'une macro est exécutée et qu'elle doit lire une suite de n arguments dont le dernier est délimité, les $n-1$ premiers arguments sont assignés comme des arguments non délimités. Le dernier argument reçoit ce qui reste jusqu'au délimiteur, ceci étant éventuellement vide.

D'après cette règle, l'argument #2 de l'exemple précédent n'est pas délimité et sera « x ». L'argument #3 qui est délimité sera donc qu'il reste jusqu'au délimiteur « / » c'est-à-dire « yz ». Le texte de remplacement sera donc :

```
Bonjour ab123c, x et yz.
```

32 - Règle
Dans tous les cas, qu'il soit délimité ou pas, si un argument est de la forme {⟨*tokens*⟩}, où ⟨*tokens*⟩ est un ensemble d'unités lexicales où les accolades sont équilibrées, alors TEX *supprime les accolades extérieures* et l'argument lu est « ⟨*tokens*⟩ ».

Un cas particulier de cette règle engendre une autre règle :

33 - Règle
Un argument délimité est vide si l'ensemble des tokens qui lui correspond est vide ou « {} ».

Ainsi, si on définit la macro \bar ainsi :

```
\def\bar#1#2#3+{L'argument délimité est : "#3".}
```

Alors, l'argument #3 est vide dans ces deux cas :

- « \bar{123}x+ » car « 123 » et « x » sont les 2 premiers arguments et il ne reste rien pour le troisième ;
- « \bar{a}{b}{}+ » car #3 reçoit « {} ».

En revanche, si on avait écrit \bar{a}{b}{}{}+, l'argument délimité aurait été « {}{} » ce qui n'est pas un argument vide.

Plutôt que d'envisager un par un de nombreux cas, voici un tableau qui affiche ce que seraient les arguments #1, #2 et #3 dans plusieurs cas de figure avec la définition de \foo rappelée en haut du tableau :

```
\def\foo.#1**#2#3/{...}
```

	#1	#2	#3
1. \foo.␣abc**123/	abc	1	23
2. \foo.**k/	<vide>	k	<vide>
3. \foo.1{**}2**{ab}{cd}/	1{**}2	ab	cd
4. \foo.{ab}**1{cd}␣/	ab	1	{cd}
5. \foo.␣**xy/		x	y
6. \foo.xy**␣z/	xy	z	<vide>
7. \foo.xy**␣/	<Erreur de compilation>		
8. \foo**/	<Erreur de compilation>		

Voici maintenant quelques observations :

- la ligne 1 met en évidence que #1 récolte ce qui est entre « . » et « ** », espaces compris ;
- on observe à la ligne 2 que le « . » est immédiatement suivi du délimiteur « ** » et donc, l'argument délimité #1 est vide. L'argument #3, qui est délimité, est vide puisqu'il ne reste rien après #2 qui est « k » ;
- la ligne 3 montre que si le délimiteur « ** » est entre accolades, il n'est pas reconnu comme délimiteur et ne stoppe pas la lecture de l'argument #1 ;
- à la ligne 4, l'argument délimité #1 est « {ab} » et comme cet argument est un texte entre accolades, les accolades sont supprimées lors de la lecture de cet argument. Par contre, l'autre argument délimité #3 est « {cd}␣ » et lui n'est pas réduit à un texte entre accolades, il y a l'espace en plus. Les accolades ne sont donc pas supprimées ;
- à la ligne 6, on peut se demander pour quelle raison #2 n'est pas « ␣ » et #3 n'est pas « z ». En fait, l'argument non délimité #2 cherche son assignation dans « ␣z » et comme on l'a vu au chapitre précédent, un espace est ignoré dans une liste d'arguments non délimités. Donc #2 devient « z », ce qui explique pourquoi l'argument délimité #3 est vide puisqu'il ne reste plus rien entre la fin de #2 et le délimiteur « / »
- l'erreur qui survient à la ligne 7 est due à l'argument non délimité #2 qui cherche son assignation dans « ␣/ ». Ignorant l'espace, #2 devient « / » et capture le délimiteur qui ne sera plus vu par \foo. TEX va donc lire du code jusqu'à ce qu'il rencontre une autre « / », ce qui est peu probable et qui, si cela arrivait, donnerait un argument #3 beaucoup plus étendu que prévu. Si TEX ne trouve pas d'autre « / », la lecture peut se poursuivre jusqu'à ce qu'il arrive sur un \par et comme la macro n'est pas déclarée \long, il va se plaindre d'un « `Paragraph ended before \foo was complete` » [1]. Si l'on est proche de la fin du fichier, TEX peut même vainement chercher son délimiteur « / » jusqu'à la fin du fichier et émettre un « `File ended while scanning use of \foo` » [2].
- à la ligne 8, ce qui suit la commande \foo n'est pas « . » donc TEX nous informe que ce qu'il trouve ne correspond pas à ce qu'il attend avec un « `Use of \foo doesn't match its definition` » [3].

■ Exercice 32

Voici une commande à arguments délimités \bar :

```
\def\bar#1#2:#3\relax#4{⟨code⟩}
```

1. Quels sont les arguments délimités ?
2. Que valent les 4 arguments dans les cas suivants :
   a) `\bar 123:456\relax789`
   b) `\bar\def\relax:\relax\relax\relax`
   c) `\bar:\relax:{\bar\relax}\bar\relax\bar`
   d) `\bar0:a\relax`
3. Donner un cas où l'on est sûr d'obtenir une erreur de compilation.

1. Le paragraphe s'est terminé avant que la lecture de \foo n'ait été achevée.
2. Fichier terminé pendant la lecture de \foo.
3. L'emploi de \foo ne correspond pas à sa définition.

4. Donner un cas où #1 vaut « \bar12:34\relax5 » et #3 vaut « \relax ».
5. Donner un cas où #1 et #4 sont des arguments vides, mais pas #2 et #3.

□ **Solution**

Rappelons la définition de \bar : \def\bar#1#2:#3\relax#4{⟨*code*⟩}

1. Les arguments délimités sont le deuxième et le troisième. Il est important de noter que le 4e n'est pas délimité.
2. Voici les résultats présentés dans un tableau :

	#1	#2	#3	#4
a)	1	23	456	7
b)	\def	\relax	<vide>	\relax
c)	:	\relax	{\bar\relax}\bar	\bar
d)	0	<vide>	a	<inconnu>

   Les chiffres 8 et 9 au premier exemple et le dernier \relax au deuxième exemple ne sont pas pris en compte par la macro \bar.

   L'argument #4 de la dernière ligne est qualifié d'« inconnu », car l'argument qui suit « \bar0:a\relax » n'est pas précisé dans l'énoncé. Si cet argument est \par ou deux retours à la ligne consécutifs (ce qui est la même chose), il y aura une erreur de compilation puisque dans ce cas, l'argument #4 serait ce \par ce qui est interdit puisque la macro \bar n'est pas déclarée \long.
3. Ce code « \bar:a\relax bcd\par » provoquera une erreur de compilation, car l'argument #1 qui n'est pas délimité va capturer le « : » qui ne sera plus visible par la macro \bar. Celle-ci va donc poursuivre sa lecture jusqu'à trouver un autre « : » pour affecter à l'argument délimité #2 ce qu'elle a emmagasiné jusque là. Ce faisant, elle va tomber sur le \par et comme cette macro \bar n'a pas été définie \long, TEX va émettre une erreur de compilation.
4. Par exemple « \bar{\bar12:34\relax5}X:{\relax}\relax Y » où ici les arguments 2 et 4 sont « X » et « Y ».
5. Par exemple « \bar{}X:Y\relax{} » où les arguments 2 et 3 sont « X » et « Y ». ■

■ **Exercice 33**

Quel sera l'affichage produit par le code suivant ?

```
\def\bar/#1#2/{*#1*#2*}
\bar////
```

□ **Solution**

TEX attend « / » juste après \bar. Ce caractère est donc absorbé. L'argument #1 n'est pas délimité, il reçoit donc le second « / ». Comme #2 est un argument délimité, il sera vide puisque le second délimiteur arrive immédiatement après #1. Il reste le dernier « / » qui n'a pas été lu avec la macro \bar et ses arguments ; il sera donc affiché tel quel juste après la fin du texte de remplacement de \bar.

On obtient donc : */**/ ■

■ **Exercice 34**

Soit une macro \baz définie de cette façon : \def\baz#1\relax{#1}. Elle ressemble un peu à la commande \identity que nous avons déjà vue à la page 71 sauf qu'ici, l'argument #1 est *délimité.*

Lorsqu'elles sont exécutées, expliquer s'il y a des différences entre ces différentes lignes de code et si oui, lesquelles :

```
\baz\let\bar=A\let\bidule=B\relax
\baz{\let\bar=A\let\bidule=B}\relax
\baz{\let\bar=A}{\let\bidule=B}\relax
```

□ **Solution**

Pour la première ligne, l'argument #1 est ce qui se trouve entre \baz et \relax, donc le texte de remplacement de ce code est « \let\bar=A\let\bidule=B ». TEX va donc définir deux lettres implicites.

À la deuxième ligne, TEX lit l'argument « {\let\bar=A\let\bidule=B} ». Comme il est réduit à un texte entre accolades, ces accolades sont supprimées et donc cette ligne de code est strictement équivalente à la première.

Pour la troisième, l'argument est « {\let\bar=A}{\let\bidule=B} » qui n'est pas réduit à un texte entre accolades et ces accolades ne seront donc pas supprimées lors de la lecture de l'argument. TEX va donc définir des lettres implicites à l'intérieur de deux groupes. Ces définitions seront perdues à la fin de chaque groupe, c'est-à-dire immédiatement après avoir été définies. Cette troisième ligne de code n'a donc aucun effet qui survit à la fermeture des deux groupes. ■

Le délimiteur du dernier argument peut également être une accolade ouvrante. Lorsque la macro sera exécutée, cela signifiera que TEX s'attendra à ce que cet argument s'étende jusqu'à la prochaine accolade ouvrante.

34 - Règle

Pour spécifier que le dernier argument ⟨*i*⟩ d'une macro doit être délimité par une accolade ouvrante, il faut écrire dans le texte de paramètre

#⟨*i*⟩#{

L'accolade qui suit le # sert de délimiteur et marque aussi le début du texte de remplacement.

Lorsque la macro sera exécutée, l'argument #⟨*i*⟩ s'étendra jusqu'à la prochaine accolade ouvrante.

Voici une macro \baz où son 3e argument est délimité par une accolade ouvrante :

Code no II-53

```
1 \def\baz#1#2#3#{"#1", puis "#2" et enfin "#3".}
2 \baz{abc}def{gh}ij
```

```
"abc", puis "d" et enfin "ef".ghij
```

D'après les règles vues dans ce chapitre, #1 est « abc », #2 est « d » et l'argument délimité #3 est bien « ef ». La suite du code, « {gh}ij », ne sera pas lu par la macro \baz. C'est pourquoi l'affichage produit par ce code se trouve après le point qui marque la fin de texte de remplacement de la macro \baz.

Voici enfin deux macros à usage « généraliste » en programmation, toutes les deux à arguments délimités. La première `\firstto@nil*` lit tout ce qui se trouve jusqu'au prochain « `\@nil` » et en prend le premier argument non délimité. La seconde, `\remainto@nil*` prend ce qui reste. Elles [4] ne sont *pas* à confondre avec `\firstoftwo` et `\secondoftwo` qui elles, ne sont pas à arguments délimités.

On remarque que le caractère « `@` » fait partie du nom de ces deux macros ainsi que de `\@nil`, ce qui est normalement interdit. Pour que cela soit possible, il faut modifier le code de catégorie de l'arobase qui est naturellement 12 pour le mettre à 11, celui des lettres, afin qu'il puisse servir à constituer un nom de macro. En TEX, on doit écrire « `\catcode‘\@=11` » pour que `@` devienne une lettre puis « `\catcode‘\@=12` » pour revenir au régime de catcode de l'arobase par défaut. En LATEX, les macros `\makeatletter` et `\makeatother` effectuent ces actions.

La zone de ces modifications sera limitée aux définitions des macros de telle sorte qu'il ne sera plus possible de se référer à ces macros par la suite. L'intérêt est qu'elles deviennent « privées », terme qu'il faut comprendre au sens de « envers du décor réservé à un usage de programmation », ou encore « à ne pas mettre entre toutes les mains ». Ainsi, l'utilisateur final ne peut pas y accéder, sauf évidemment s'il effectue la manœuvre de modification du catcode de `@` [5].

Voici les définitions et quelques exemples d'utilisation de `\firstto@nil` et `\remainto@nil` :

Code n° II-54

```
\catcode‘\@11
\long\def\firstto@nil#1#2\@nil{#1}
\long\def\remainto@nil#1#2\@nil{#2}
a) \firstto@nil foobar\@nil   \qquad b) \remainto@nil foobar\@nil  \par
c) \firstto@nil {foo}bar\@nil \qquad d) \remainto@nil {foo}bar\@nil
```

a) f b) oobar
c) foo d) bar

On voit souvent cette macro `\@nil` dans le code de LATEX et on la verra aussi fréquemment dans les macros de ce livre. On peut se demander ce qu'elle fait vraiment et quelle est sa définition. La réponse est très simple : il est admis que `\@nil` est une macro *non définie* (et doit le rester) dont le seul usage est de servir de délimiteur pour argument délimité. Ici, une précision s'impose : le fait qu'elle ne soit pas définie *n'implique pas* que, en tant que délimiteur, elle soit interchangeable avec une autre macro non définie.

35 - Règle

Lorsqu'un délimiteur d'argument est une séquence de contrôle, TEX ne prend pas en compte la signification de cette séquence de contrôle : seuls les caractères qui composent son nom sont examinés.

Si par exemple, on définit une macro `\foo` ainsi :

`\def\foo#1\truc{`⟨*code*⟩`}`

alors par la suite, TEX s'attendra à ce que `\foo` soit suivie d'un code (qui deviendra l'argument `#1`) dont la fin sera marquée par `\truc` (c'est-à-dire le caractère d'échap-

4. Ces deux commandes sont `\@car` et `\@cdr` du noyau LATEX.
5. LATEX3 utilise le caractère « _ » pour ses macros privées.

pement suivi des lettres t, r, u et c). Même si une autre macro \machin est rendue \let-égale à \truc, cette macro \machin ne sera pas comprise comme le délimiteur de la macro \foo.

■ **Exercice 35**

Existe-t-il une différence de comportement entre ces deux lignes de code et si oui, laquelle :

```
\firstoftwo{⟨code1⟩}{⟨code2⟩}
\firstto@nil{⟨code1⟩}{⟨code2⟩}\@nil
```

□ **Solution**

Il n'y a pas de différence de comportement. Ces deux lignes se comportent de façon identique. Dans les deux cas, l'argument #1 n'est pas délimité et va être dépouillé de ses accolades. Leur texte de remplacement est donc pour toutes les deux : ⟨*code1*⟩. De plus, elles ont toutes les deux lu et absorbé l'argument ⟨*code2*⟩ qui disparait dans leur texte de remplacement.

Cette équivalence serait également valable entre ces 3 lignes qui toutes exécutent ⟨*code2*⟩ :

```
\secondoftwo{⟨code1⟩}{⟨code2⟩}
\remainto@nil{⟨code1⟩}{⟨code2⟩}\@nil
\remainto@nil{⟨code1⟩}⟨code2⟩\@nil
```

■

3.2. Mise en pratique

Les arguments délimités ne présentent pas de difficulté théorique spéciale, aussi il est temps de les utiliser sur un exemple.

3.2.1. Afficher ce qui est à droite d'un motif

Nous allons nous employer à programmer une macro\rightof* dont la syntaxe est

```
\rightof{⟨chaine⟩}{⟨motif⟩}
```

et qui affiche ce qui est à gauche du ⟨*motif*⟩ dans la ⟨*chaine*⟩, étant entendu que si ⟨*motif*⟩ ne figure pas dans la ⟨*chaine*⟩, rien ne doit être affiché.

Du côté de la mécanique interne, cette macro se contentera de lire ses deux arguments, de définir et d'appeler une macro \right@of. Cette macro, à arguments délimités, sera chargée d'effectuer le travail.

L'intérêt du « @ » dans « \right@of est de marquer la frontière entre macro publique et macro privée pour délimiter ce qui a une syntaxe clairement définie et stable dans le temps pour les macros publiques de ce qui est de la cuisine interne pour les macros privées. Ces dernières sont susceptibles d'être modifiées, voire supprimées pourvu que la syntaxe et les fonctionnalités de la macro publique soient conservées.

Voici comment doit être définie la macro \right@of :

```
\def\rightof#1⟨motif⟩#2\@nil{#2}
```

où le délimiteur ⟨*motif*⟩ est l'argument #2 de la macro principale.

Voici donc le code de \rightof où, conformément à ce qui a été dit, les tokens de paramètre # sont doublés pour la macro « fille » \right@of :

Code n° II-55

```
\catcode`\@=11 % "@" devient une lettre
\def\rightof#1#2{%
  \def\right@of##1#2##2\@nil{##2}% définit la macro auxiliaire
  \right@of#1\@nil% appelle la macro auxiliaire où #1 est la <chaine>
}
\catcode`\@=12 %"@" redevient un signe
--\rightof{Programmation}{g}--\par
--\rightof{Programmation}{gram}--\par
--\rightof{Programmation}{on}--
```

```
–rammation–
–mation–
––
```

3.2.2. Les cas problématiques

La macro fonctionne bien… En apparence seulement, car le cas où ⟨*chaine*⟩ ne contient pas le ⟨*motif*⟩ n'a pas encore été examiné :

Code n° II-56

```
\catcode`\@=11 % "@" devient une lettre
\def\rightof#1#2{%
  \def\right@of##1#2##2\@nil{##2}% définit la macro auxiliaire
  \right@of#1\@nil% appelle la macro auxiliaire
}
\catcode`\@=12 %"@" redevient un signe
--\rightof{Programmation}{z}--\par
--\rightof{Programmation}{Gram}
```

```
! Paragraph ended before \right@of was complete. <to be read again>
```

TEX explique qu'un \par a été rencontré avant que l'argument ne soit entièrement lu, ce qui est interdit puisque la macro n'est pas déclarée \long. Le texte de remplacement de \rightof{Programmation}{z} (cas de la ligne n° 7) permet d'analyser ce qui se passe. Le texte de remplacement est écrit ci-dessous (les doubles ## ont été remplacés par des simples puisque nous écrivons le texte de remplacement) :

```
\def\right@of#1z#2\@nil{#2}
\entre@ Programmation\@nil
```

La première ligne spécifie donc que la macro \right@of doit être suivie de quelque chose (l'argument #1) qui s'étend jusqu'à « z » puis de quelque chose (l'argument #2) qui s'étend jusqu'au \@nil. Or, « z » ne figure pas entre \right@of et \@nil puisque ce qui s'y trouve est l'argument #1 de la macro rightof, c'est-à-dire « Programmation » ! C'est pourquoi TEX le cherche jusqu'à tomber sur le \par qui stoppe cette vaine recherche puisque la macro n'est pas déclarée \long.

Tout serait bien plus facile si on disposait d'une macro qui teste si le ⟨*motif*⟩ est contenu dans ⟨*chaine*⟩ : dans l'affirmative, il suffirait d'appeler la macro \right@of précédemment définie et sinon, ne rien faire ce qui revient à ne rien afficher. Une telle macro, capable de faire un test, sera écrite plus tard (voir page 192). En attendant, il faut trouver une astuce pour se protéger de cet inconvénient.

En fait aucune erreur ne serait émise si, quelle que soit la ⟨*chaine*⟩, \right@of voyait toujours le ⟨*motif*⟩ avant le \@nil. Afin qu'il en soit ainsi, on va délibérément écrire ce ⟨*motif*⟩ quelque part entre \right@of et \@nil lors de l'appel à la macro

\right@of. L'écrire juste avant le \@nil nous assure que rien ne sera renvoyé dans le cas où ⟨*chaine*⟩ ne contient pas ⟨*motif*⟩ puisque dans ce cas, ##2 est vide :

Code n° II-57

```
\catcode`\@=11 % "@" devient une lettre
\def\rightof#1#2{%
  \def\right@of##1#2##2\@nil{##2}% définit la macro auxiliaire
  \right@of#1#2\@nil% appelle la macro auxiliaire avec le motif #2
}
\catcode`\@=12 %"@" redevient un signe
--\rightof{Programmation}{g}--\par
--\rightof{Programmation}{ti}--\par
--\rightof{Programmation}{z}--\par
--\rightof{Programmation}{Gram}--
```

```
–rammationg–
–onti–
––
––
```

L'inconvénient qui saute aux yeux est que le motif #2, ajouté juste avant \@nil, est dirigé vers l'affichage lorsque la ⟨*chaine*⟩ contient le ⟨*motif*⟩ ! Tout simplement parce que l'argument ##2 de \right@of qui est affiché est ce qui se trouve entre la *première* occurrence du ⟨*motif*⟩ et \@nil.

Pour mieux comprendre ce qui se joue en coulisses, examinons ce qui se passe lorsque l'appel « \rightof{Programmation}{g} » est exécuté :

1. la macro \right@of est définie par : \def\right@if#1g#2\@nil{#2} à la ligne n° 3 ;
2. écrire « \right@of Programmationg\@nil » donne ce qui se trouve entre la première occurrence de « g » et \@nil, c'est donc « rammationg » qui est affiché.

Procédons de même avec l'appel « \rightof{Programmation}{z} », cas où le ⟨*motif*⟩ n'est pas contenu dans la ⟨*chaine*⟩ :

1. la macro \right@of est définie par : \def\right@if#1z#2\@nil{#2} ;
2. écrire « \right@of Programmationz\@nil » donne ce qui se trouve entre « z » et \@nil, c'est-à-dire rien.

Comment faire en sorte que le ⟨*motif*⟩ rajouté juste avant le \@nil ne soit pas affiché dans le cas où ⟨*motif*⟩ est contenu dans ⟨*chaine*⟩ ? Réponse : en incluant dans l'argument ##2 de \right@of une autre macro à argument délimitée qui fera disparaitre ce qui est indésirable. Cette macro à argument délimité sera \gobto@@nil* et absorbera tout ce qui se trouve entre elle et le prochain \@@nil rencontré. On note ici qu'il ne s'agit pas de \@nil mais de \@@nil afin que le délimiteur de cette macro lui soit spécifique :

```
\def\gobto@@nil#1\@@nil{}
```

On va donc ajouter un \@@nil lors de l'appel à la macro \right@of :

```
\right@of#1#2\@@nil\@nil
```

Et il nous reste à correctement placer la macro \gobto@@nil. Pour que ce \@@nil soit toujours apparié avec une macro \gobto@@nil, que ⟨*motif*⟩ soit contenu dans chaine ou pas, il faut mettre \gobto@@nil juste avant et juste après #2 :

Code n° II-58

```
\catcode`\@=11 % "@" devient une lettre
\def\gobto@@nil#1\@@nil{}
\def\rightof#1#2{%
  \def\right@of##1#2##2\@nil{##2}% définit la macro auxiliaire
  \right@of#1\gobto@@nil#2\gobto@@nil\@@nil\@nil% appelle la macro auxiliaire
}
\catcode`\@=12 %"@" redevient un signe
--\rightof{Programmation}{g}--\par
--\rightof{Programmation}{ti}--\par
--\rightof{Programmation}{z}--\par
--\rightof{Programmation}{Gram}--
```

```
–rammation–
–on–
––
––
```

Faisons fonctionner la macro `\rightof` dans le cas où ⟨*motif*⟩ est contenu dans ⟨*chaine*⟩. Comme précédemment, examinons quels sont ses arguments lorsque l'on écrit « `\rightof{Programmation}{g}` » :

1. l'appel à la macro `\right@of` se fait ainsi :

   ```
   \right@of Programmation\gobto@@nil g\gobto@@nil\@@nil\@nil
   ```

2. son argument #2, qui est entre le premier « g » et `\@nil` est :

   ```
   rammation\gobto@@nil g\gobto@@nil\@@nil
   ```

3. l'affichage qui en résulte est bien « rammation » car la macro `\gobto@@nil` absorbe tout ce qui se trouve jusqu'au `\@@nil`. TEX ne tient aucun compte de ce qu'il absorbe. Que la macro `\gobto@@nil` y figure ou pas n'y change rien : la macro à argument délimité effectue « bêtement » son travail sans analyser ce qui se trouve dans son argument.

Voici maintenant le cas où ⟨*motif*⟩ n'est contenu dans ⟨*chaine*⟩, comme lorsqu'on écrit « `\rightof{Programmation}{z}` » :

1. la macro `\right@of` est appelée par

   ```
   \right@of Programmation\gobto@@nil z\gobto@@nil\@@nil\@nil
   ```

2. l'argument #2 se trouve entre le « z » et le `\@nil` :

   ```
   \gobto@@nil\@@nil
   ```

3. grâce à la définition de `\gobto@@nil`, ce code ne se traduit par aucun affichage.

Impliquer ainsi plusieurs macros à arguments délimités et faire en sorte que la suivante hérite de l'argument de la précédente pour le traiter à son tour est couramment utilisé en programmation et démontre, comme dans ce cas particulier, la puissance des arguments délimités. En revanche, la gymnastique mentale requise pour mettre en place un tel dispositif et correctement envisager tous les cas est loin d'être naturelle et demande une certaine habitude… Le plus difficile est d'analyser un code déjà écrit, voire de comprendre un code que l'on avait écrit soi-même et oublié entre temps. Qui peut se vanter de comprendre du premier coup d'œil l'appel à la macro auxiliaire mise au point précédemment ?

```
\right@of#1\gobto@@nil#2\gobto@@nil\@@nil\@nil
```

Le manque de lisibilité est criant et démontre l'importance de commenter son code, autant pour soi-même que pour les autres. Pour augmenter la lisibilité du code, il serait préférable de recourir à une macro `\ifin` (dont l'élaboration sera expliquée page 192) qui teste si l'argument #1 contient l'argument #2 et ayant pour syntaxe

```
\ifin{#1}{#2}
    {⟨code à exécuter si vrai⟩}
    {⟨code à exécuter si faux⟩}%
```

On peut ainsi programmer plus confortablement et utiliser la toute première version de `\rightof` et se dispenser d'envisager le cas problématique où ⟨*chaine*⟩ ne contient pas le ⟨*motif*⟩ :

Code n° II-59

```
\catcode`\@=11 % "@" devient une lettre
\def\ifin#1#2#3#4{<code vu plus tard>}
\def\rightof#1#2{%
  \ifin{#1}{#2}%
    {% si #1 contient le #2
    \def\right@of##1#2##2\@nil{##2}% définit la macro auxiliaire
    \right@of#1\@nil% appelle la macro auxiliaire
    }%
    {}% sinon, ne rien faire
}
\catcode`\@=12 %"@" redevient un signe
--\rightof{Programmation}{g}--\par
--\rightof{Programmation}{z}--\par
--\rightof{Programmation}{o}--
```

```
–rammation–
––
–grammation–
```

3.2.3. À gauche toute

Ne nous arrêtons pas en si bon chemin et venons-en à la macro symétrique de `\rightof`, la macro `\leftof*` qui sera chargée d'afficher ce qui est à gauche du ⟨*motif*⟩ dans la ⟨*chaine*⟩.

Pour se prémunir de l'erreur de compilation rencontrée dans le cas où le ⟨*motif*⟩ (argument #2) n'est pas contenu dans la ⟨*chaine*⟩ (argument #1), procédons comme avec `\rightof` et ajoutons ce ⟨*motif*⟩ lors de l'appel à la macro auxiliaire :

Code n° II-60

```
\catcode`\@=11
\def\leftof#1#2{%
  \def\leftof@i##1#2##2\@nil{##1}% renvoie ce qui est à gauche de #2
  \leftof@i #1#2\@nil% ajoute #2 après #1
}
\catcode`\@=12
--\leftof{Programmation}{ram}--\par
--\leftof{Programmation}{zut}--
```

```
–Prog–
–Programmation–
```

Cela était attendu : lorsqu'on écrit `\leftof@i #1#2\@nil`, si `#1` ne contient pas `#2`, alors le `#2` ajouté en fin d'argument devient la première occurrence du ⟨*motif*⟩ et donc, `#1` est affiché en entier.

À nouveau, il va falloir déployer une méthode astucieuse pour que, lorsque `#1` est renvoyé en entier (ce qui correspond au cas problématique), rien ne soit affiché. Pour cela, la macro auxiliaire `\leftof@i` ne va pas afficher, `##1` mais transmettre ce « `##1#1` » à une autre macro auxiliaire `\leftof@ii`. Cette dernière aura comme délimiteur d'argument est `#1` tout entier et se chargera d'afficher ce qui est à gauche de la ⟨*chaine*⟩ `#1` :

```
\def\leftof@ii##1#1##2\@nil{##1}
```

Ainsi, la macro `\leftof@i` devient :

```
\leftof@i##1#2##2\@nil{\leftof@ii##1#1\@nil}
```

et on ajoute ici aussi `#1` après `##1` pour que la macro `\leftof@ii` voie toujours la ⟨*chaine*⟩ `#1` :

Code n° II-61

```
\catcode`\@=11
\def\leftof#1#2{%
  \def\leftof@i##1#2##2\@nil{\leftof@ii##1#1\@nil}%
  \def\leftof@ii##1#1##2\@nil{##1}%
  \leftof@i #1#2\@nil
}
\catcode`\@=12
--\leftof{Programmation}{ram}--\par
--\leftof{Programmation}{zut}--
```

```
–Prog–
––
```

Compte tenu de la faible lisibilité de ce code et de la difficulté à suivre comment se transmettent les arguments, voici un tableau montrant comment sont lus et isolés les arguments délimités dans le cas où ⟨*chaine*⟩ contient ⟨*motif*⟩, c'est-à-dire le premier cas. On a encadré ce qui délimite les arguments des macros :

Appel macro principale	`\leftof{Programmation}{ram}`
Appel à `\leftof@i` Argument `##1` retenu par `\leftof@i`	`\leftof@i`Prog`ram`mationram`\@nil` Prog
Appel à `\leftof@ii` Argument `##1` retenu par `\leftof@ii`	`\leftof@ii`Prog`Programmation`\@nil Prog
Affichage	Prog

Voici maintenant le déroulement des opérations lorsque ⟨*chaine*⟩ ne contient pas ⟨*motif*⟩ :

Appel macro principale	`\leftof{Programmation}{zut}`
Appel à `\leftof@i` Argument `##1` retenu par `\leftof@i`	`\leftof@i`Programmation`zut`\@nil Programmation
Appel à `\leftof@ii` Argument `##1` retenu par `\leftof@ii`	`\leftof@ii`Programmation`Programmation`\@nil <vide>
Affichage	<vide>

Il est bon de noter que la macro `\leftof`, bien que requérant elle aussi deux macros auxiliaires, est programmé différemment de `\rightof`. Dans le cas de `\leftof`, la deuxième macro auxiliaire `\leftof@ii` est appelée *dans l'argument* de `\leftof@i`.

Dans le cas de \rightof, la deuxième macro auxiliaire \gobto@@nil était écrite dans l'appel à la macro \right@of, ce qui est un cas particulier pas toujours faisable.

■ **Exercice 36**

Modifier la macro \rightof pour utiliser la méthode employée avec \leftof : la première macro auxiliaire \rightof@i appellera *dans son argument* une deuxième macro auxiliaire \rightof@ii.

□ **Solution**

On utilise une astuce similaire à celle vue avec \leftof : la deuxième macro auxiliaire \rightof@ii va renvoyer ce qui est à *gauche* du ⟨*motif*⟩ puisque celui-ci, indésirable, était rajouté à la toute fin de l'appel à \rightof@i :

Code n° II-62

```
\catcode`\@=11
\def\rightof#1#2{%
  \def\rightof@i##1#2##2\@nil{\rightof@ii##2#2\@nil}%
  \def\rightof@ii##1#2##2\@nil{##1}%
  \rightof@i#1#2\@nil
}
\catcode`\@=12
--\rightof{Programmation}{g}--\par
--\rightof{Programmation}{z}--
```

```
–rammation–
––
```

Il est intéressant de noter que lorsque le ⟨*motif*⟩ est « z », l'argument ##2 de \rightof@i est vide et donc, l'argument ##1 de \rightof@ii est vide aussi. ■

3.2.4. Autre problème : les accolades dans le motif

On n'en a pas fini avec les cas problématiques… Examinons maintenant pourquoi la macro \rightof ne fonctionne pas lorsqu'elle est appelée avec un argument #2 contenant un (ou des) groupe(s) entre accolades. Voici un exemple simple où une erreur de compilation est produite :

Code n° II-63

```
\catcode`\@=11
\def\gobto@@nil#1\@@nil{}
\def\rightof#1#2{%
  \def\right@of##1#2##2\@nil{##2}% définit la macro auxiliaire
  \right@of#1\gobto@@nil#2\gobto@@nil\@@nil\@nil% appelle la macro auxiliaire
}
\catcode`\@=12
\rightof{abc{1}def}{c{1}d}% affiche ce qui est à droite de "c{1}d"
```

```
You can't use 'macro parameter character #' in horizontal mode.
```

Pour comprendre ce qu'il se passe avec le token « # » qui est incriminé dans le message d'erreur, écrivons le texte de remplacement de \rightof{abc{1}def}{c{1}d} :

```
\def\right@of##1c{1}d##2\@nil{##2}
\right@of abc{1}def\gobto@@nil c{1}d\gobto@@nil\@@nil\@nil
```

Le problème se trouve à la première ligne. TEX va lire « `\def\right@of##1c{1}` » ce qui va définir la macro auxiliaire `\rightof@` : elle sera donc à argument délimité par « `c` » et son texte de remplacement sera « `1` ». Par la suite, les tokens qui restent à lire sont « `d##2\@nil{##2}` ». Ceux-ci n'étant plus dans le texte de paramètre d'une macro, TEX bute « `#` », car ce token est justement attendu dans le texte de paramètre d'une macro [6]. Le message d'erreur signale donc que TEX rencontre ce token dans un contexte (ici, le mode horizontal) dans lequel il est interdit. D'ailleurs, si le mode en cours avait été le mode vertical, le message aurait été le même : « `You can't use 'macro parameter character #' in vertical mode.` »

Il est difficile de modifier la macro `\rightof` pour qu'elle puisse aussi trouver ce qui est entre deux arguments qui comportent des groupes entre accolades, car dans ces cas, les arguments délimités sont interdits. Nous emploierons une tout autre méthode, bien plus lente, qui consisterait à s'y prendre token par token (voir page 366).

36 - Règle

Les délimiteurs figurant dans le texte de paramètre d'une macro ne peuvent pas contenir de tokens de catcode 1 ou 2 équilibrés (accolade ouvrante et fermante) car celles-ci seraient interprétées non pas comme faisant partie d'un délimiteur, mais comme marquant les frontières du texte de remplacement.

La conséquence est donc que lorsqu'une macro définit une macro fille à argument délimité où les délimiteurs sont les arguments de la macro mère, ceux-ci ne peuvent pas contenir un groupe entre accolades.

6. On le rencontre aussi dans le préambule d'un alignement initié par la primitive `\halign` ou `\valign`.

Chapitre 4

Développement

TEX est un langage particulier et parmi d'autres subtilités, le développement est un de ses mécanismes intimes. Maitriser le développement est nécessaire pour programmer, car cette notion occupe une place centrale dans la programmation TEX. Certains diraient que la gestion du développement fait partie des choses assez pénibles, mais lorsqu'on programme, il est souvent nécessaire de savoir contrôler finement ce mécanisme fondamental. Ce chapitre, consacré à l'apprentissage des outils permettant de contrôler le développement, nous ouvrira la voie pour appréhender la programmation abordée dans les parties suivantes.

4.1. Développement des commandes

Intéressons-nous aux commandes (qui ont un texte de remplacement donc), par opposition aux primitives qui sont « codées en dur » dans TEX. Le texte de remplacement est, en quelque sorte, la définition de la commande, c'est-à-dire la représentation interne qu'en a TEX. Reprenons par exemple la commande très simple `\foo` définie par :

```
\def\foo{Bonjour}
```

Son texte de remplacement est « Bonjour », c'est-à-dire que dans le code, lorsque TEX va rencontrer `\foo`, alors il va aller chercher sa signification dans sa mémoire et va remplacer cette séquence de contrôle par son texte de remplacement qui est « Bonjour ». Ce remplacement, cette substitution, cette transformation du code s'appelle le « développement », et on dit que la macro `\foo` a été développée par TEX. Pour être tout à fait clair, cette « transformation du code », a lieu dans la mémoire de TEX (nommée « pile d'entrée »). Le code source n'est bien évidemment pas modifié.

37 - Règle et définition

TEX est un langage de macros, c'est-à-dire qu'il agit comme un outil qui *remplace* du code par du code. Ce remplacement, aussi appelé développement, a lieu dans une zone de mémoire nommée « pile d'entrée ».

Ainsi, lorsque TEX exécute une macro, elle (ainsi que ses éventuels arguments) est d'abord remplacée par le code qui lui a été donné lorsqu'on l'a définie avec `\def`. Ce code est le « texte de remplacement ». Une fois que ce développement a été fait, TEX continue sa route en tenant compte du remplacement effectué.

Prenons le temps d'examiner en détail ce qui se passe réellement et définissons les conventions suivantes : le symbole ⋙ représente l'endroit où TEX s'apprête à lire la suite du code. On a vu qu'il ne lisait que ce dont il avait besoin, et on va donc encadrer en pointillés le ou les tokens que TEX va lire à la prochaine étape. Lorsque le développement se fait, la pile est encadrée. Observons donc, étape par étape ce qui se passe lorsqu'on écrit « `\foo le monde` » (l'espace après `\foo` sera ignoré en vertu de la règle concernant les espaces du code vue à la page 42) :

```
⋙\foo le monde
⋙\foo␣le monde
⋙ Bonjour le monde
```

Ayant développé la commande `\foo` dans sa pile, TEX continue à lire le code qu'il a obtenu. Ici, cela se solderait par la lecture des caractères « Bonjourle monde » et par leur affichage.

Il est important de comprendre que la notion de développement ne s'applique qu'à des séquences de contrôle ou des caractères actifs. On peut donner la définition suivante :

38 - Définition

Une séquence de contrôle est développable si TEX peut la remplacer par un code différent.

Les primitives n'ont pas de texte de remplacement. Cependant, certaines d'entre elles sont développables. Nous en avons rencontré une qui est la paire `\csname` ... `\endcsname` dont l'argument est le code qui se trouve entre elles. En effet, lorsque TEX les exécute, il transforme leur argument en une séquence de contrôle. Cette action recoupe la définition donnée juste au-dessus : le code qui est substitué est bien différent de ce qu'il y avait avant le développement.

La plupart des primitives ne sont pas développables en ce sens que si l'on force leur développement (nous verrons comment), on obtient la primitive elle-même. L'action de ces primitives est de provoquer une action par le processeur lorsqu'elles sont exécutées par TEX. Par exemple, pour la primitive `\def` qui n'est pas développable, son action est de ranger dans la mémoire de TEX le texte de remplacement d'une macro. Les caractères de code de catégorie 11 comme « `A` » ou même ceux dont le code de catégorie est moins inoffensif comme « `$` » ne sont pas développables non plus. Comme les primitives non développables, ils provoquent une action via le programme `tex` qui est de placer ce caractère dans la page pour les lettres et passer en mode mathématique pour `$`. Seuls les caractères rendus actifs dont le code de catégorie est 13 peuvent se développer puisque leur comportement est similaire à celui des commandes.

Prenons maintenant l'exemple suivant où la macro \foo admet 2 arguments :

```
\def\foo#1#2{Bonjour #1 et #2}
```

Faisons l'hypothèse que l'on stocke deux prénoms « Alice » et « Bob » dans les deux macros suivantes :

```
\def\AA{Alice } \def\BB{Bob }
```

Pour bien comprendre le mécanisme du développement, prenons à nouveau le temps d'examiner en détail ce qui va se passer lorsqu'on écrit :

```
\csname foo\ensdcsname\AA\BB et les autres
```

.

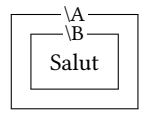
Dans ce cas, voici les étapes par lesquelles TEX va passer :

1. lecture du code dont il a besoin pour former la commande \foo :
   ⋙\csname foo\endcsname\AA\BB et les autres.
2. développement de ce qui a été lu, ce qui provoque la construction de la macro \foo dans la pile :
   ⋙\foo\AA\BB et les autres.
3. la pile ne contenant pas les deux arguments requis pour la macro \foo, ceux-ci sont donc lus à l'extérieur de la pile :
   ⋙\foo\AA\BB et les autres.
4. développement de \foo et ses deux arguments qui provoque sa substitution par son texte de remplacement ;
   ⋙Bonjour \AA et \BB et les autres.
5. affichage de « Bonjour » et lecture du code jusqu'à \AA :
   Bonjour ⋙\AA et \BB et les autres.
6. développement de \AA :
   Bonjour ⋙Alice et \BB et les autres.
7. affichage de « Alice et » et lecture du code jusqu'à \BB ;
   Bonjour Alice et ⋙\BB et les autres.
8. développement de \BB :
   Bonjour Alice et ⋙Bob et les autres.
9. lecture du code qui ne contient plus que des caractères inoffensifs jusqu'au « . » :
   Bonjour Alice et Bob et les autres.⋙

Il arrive qu'une séquence de contrôle puisse être développée plusieurs fois. Imaginons que l'on définisse deux séquences de contrôle \A et \B par ce code :

```
\def\A{\B}
\def\B{Salut}
```

Si l'on écrit \A dans le code, TEX va procéder à son développement et va la remplacer par \B. Cette séquence de contrôle va être développée à son tour pour être remplacée par « Salut » dans la pile. Dans ce cas, on dit que la macro \A a été 2-développée, c'est-à-dire qu'elle a dû subir 2 développements successifs. On pourrait se représenter la situation par des paquets cadeaux : la séquence de contrôle est le paquet cadeau (le contenant) et son développement est le contenu. Pour les deux macros \A et \B, il s'agit d'un système de poupées gigognes où le paquet cadeau \A contient le paquet \B qui lui-même contient « Salut ». On comprend qu'il faut ouvrir 2 paquets pour prendre connaissance du contenu final :



Si au lieu de \A, on avait écrit « \csname A\endcsname », il aurait fallu ajouter un développement de plus au début pour transformer ce code en \A. Et donc c'est après 3 développements que « \csname A\endcsname » donne « Salut ».

■ **Exercice 37**

On définit trois macros de cette façon :

```
\def\aa{\bb}
\def\bb{\csname cc\endcsname}
\def\cc{Bonjour}
```

Si on écrit dans le code \csname aa\endcsname, combien de développements successifs TEX doit-il effectuer pour que le code dans la pile soit « Bonjour » ?

□ **Solution**

Il suffit de compter les développements. Dans le schéma suivant, les flèches signifient « se développe en… » :

\csname aa\endcsname ⟶ \aa ⟶ \bb ⟶ \csname cc\endcsname ⟶ \cc ⟶ Bonjour

Il faut donc 5 développements. ■

■ **Exercice 38**

On redéfinit les macros \aa et \bb de cette façon :

```
\def\aa{\bb}
\def\bb{\def\cc{Bonjour}}
```

Quel est le 2-développement de \aa. Et le 3-développement ?

□ **Solution**

Le 2-développement de \aa est « \def\cc{Bonjour} ».

Pour le 3-développement, la question n'est pas claire. En effet, le 2-développement que l'on voit à la ligne ci-dessus n'est pas un token unique est constitué de 11 tokens. D'ailleurs, le seul qui est susceptible de se développer est \cc puisque les autres sont soit une primitive (\def), soit des accolades, soit des lettres. On va considérer que le développement ne s'applique qu'au premier token du code obtenu et la réponse va donc être : le 3-développement est identique au 2-développement, car la primitive \def se développe en elle-même. ■

■ **Exercice 39**

Redéfinissons les macros \aa, \bb et \cc de cette façon :

```
\def\aa{X}
\def\bb{Y}
\def\cc{\secondoftwo\aa\bb 123}
```

Quel code est obtenu sur la pile après 1, 2 et 3 développements de la macro \cc ?

□ **Solution**

État de la pile après :

- 1 développement : \secondoftwo\aa\bb 123
- 2 développements : \bb 123
- 3 développements : Y123

■

Les règles suivantes découlent de ce qui a été dit :

39 - Règles
La lecture du code se fait séquentiellement de gauche à droite et TEX ne cherche pas à lire plus de tokens que ce dont il a besoin.

Lorsque TEX rencontre une séquence de contrôle, il lit aussi ses éventuels arguments pour procéder à son développement qui consiste à remplacer le tout par du code. Si la pile ne contient pas assez d'arguments, TEX va chercher ces arguments manquants juste après la pile, c'est-à-dire dans le code source.

La première partie de cette règle concernant la lecture séquentielle du code est extrêmement contraignante et rend certaines choses impossibles. Par exemple définir une séquence de contrôle formée avec `\csname` et `\endcsname`, comme la macro `\ab1c`. En effet, comme on l'a vu à la page 60, c'est une erreur que d'écrire :

```
\def\csname ab1c\endcsname{⟨code⟩}
```

Nous avons vu également à la page 49 que l'on ne peut écrire

```
\let\salut\csname foo\endcsname
```

pour rendre la macro `\salut` égale à `\foo`.

Concernant le développement, TEX permet d'enfreindre la première partie de la règle en offrant la possibilité, dans une faible mesure, de développer le code *non linéairement* et c'est ce que nous allons découvrir avec la primitive `\expandafter`.

4.2. La primitive \expandafter

Il n'est pas exagéré de dire qu'aux yeux de débutants en TEX, `\expandafter` est une des primitives les plus mystérieuses qui soient. Certains lui prêtent des vertus quasi magiques et en viendraient presque à croire que disséminer quelques `\expandafter` dans un code qui ne fonctionne pas le ferait fonctionner ! Ici encore, il ne sera pas question de magie, mais au contraire de démystification puisque cette primitive `\expanfafter` fait une chose extrêmement simple.

Tout d'abord, elle est développable, c'est-à-dire que lorsque TEX va la développer, il va la remplacer par du code dans la pile. L'action de cette primitive se passe au niveau des tokens et non pas à celui des arguments. Si $\langle x \rangle$ et $\langle y \rangle$ sont deux tokens et si l'on écrit :

```
\expandafter⟨x⟩⟨y⟩
```

cela a pour effet de 1-développer le token $\langle y \rangle$ avant de lire $\langle x \rangle$. En fait, tout se passe comme si le `\expandafter` permettait de « sauter » $\langle x \rangle$ pour 1-développer $\langle y \rangle$ et revenir ensuite au point de départ.

40 - Règle
Si $\langle x \rangle$ et $\langle y \rangle$ sont deux *tokens*, écrire

```
\expandafter⟨x⟩⟨y⟩
```

a pour effet, lorsque TEX développe \expandafter, de 1-développer ⟨y⟩ sans pour autant que la tête de lecture ne bouge de place (et donc reste devant ⟨x⟩ qu'elle n'a toujours pas lu).

Le \expandafter disparait après s'être développé et on obtient donc

$$\langle x\rangle\langle *y\rangle$$

où « * » indique le 1-développement du token ⟨y⟩.

Attention, ⟨y⟩ a été 1-développé, mais n'a pas été *lu*. La tête de lecture de TEX n'a pas bougé dans l'opération. On peut se représenter la situation en imaginant que la tête de lecture dispose d'une « tête de développement », capable sur ordre de se désolidariser pour aller développer du code non encore lu. La tête de développement, une fois sa mission accomplie, revient sur la tête de lecture afin que cette dernière poursuive son travail.

Une application immédiate est que désormais, nous pouvons définir la séquence de contrôle \ab1c avec \csname : il suffit de sauter le \def avec un \expandafter pour provoquer la formation du nom de la macro avant que \def ne la voit :

Code n° II-64

```
\expandafter\def\csname ab1c\endcsname{Bonjour}
Texte de remplacement de la macro : \csname ab1c\endcsname
```

Texte de remplacement de la macro : Bonjour

Au lieu d'écrire « \expandafter\def\csname ab1c\endcsname », il est préférable de définir une macro \defname⋆ dont l'argument est le nom de la macro. Il est facile de faire de même avec \letname⋆.

Code n° II-65

```
\def\defname#1{\expandafter\def\csname#1\endcsname}
\defname{ab1c}{Bonjour}
Texte de remplacement de \litterate|\abc1| : \csname ab1c\endcsname\par
\def\letname#1{\expandafter\let\csname#1\endcsname}
\def\foo{abc}\letname{foo1}=\foo% rend la macro "\foo1" égale à \foo
Texte de remplacement de \litterate|\foo1| : \csname foo1\endcsname
```

Texte de remplacement de \abc1 : Bonjour
Texte de remplacement de \foo1 : abc

Afin de comprendre l'utilité du développement, tentons maintenant une expérience en utilisant la macro à arguments délimités \firstto@nil. Nous l'avons définie de la façon suivante à la page 90 :

```
\def\firstto@nil#1#2\@nil{#1}
```

son texte de remplacement est donc le premier argument de ce qui se trouve entre cette macro et \@nil.

Maintenant, si on définit la macro \foo par \def\foo{Bonjour} et que l'on écrit \firstto@nil\foo\@nil, qu'obtient-on ?

Code n° II-66

```
 \def\foo{Bonjour}\catcode`@=11
Résultat : \firstto@nil\foo\@nil
```

```
\catcode`@=12
```

Résultat : Bonjour

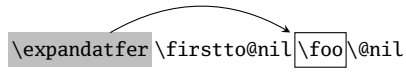
Contrairement à ce que l'on pourrait penser, on n'obtient pas la lettre « B » mais le mot entier. C'est que, fort logiquement, `\firstto@nil` a pris comme argument `#1` la macro `\foo` en entier tandis que l'argument délimité `#2` est resté vide. Il *essentiel* de ne pas confondre macro et texte de remplacement : dans un cas, il y a un seul token alors que dans l'autre, il y a tous les tokens du texte de remplacement, c'est-à-dire ici les 7 lettres du mot « Bonjour ». Pour que `\first@tonil` agisse sur le texte de remplacement de `\foo` et non pas sur `\foo` elle-même, il faut développer `\foo` *avant* que TEX ne lise la macro `\firstto@nil`. Nous allons utiliser `\expandafter` pour sauter `\firstto@nil` et provoquer ce développement :

Code n° II-67

```
\def\foo{Bonjour}\catcode`@=11
Résultat : \expandafter\firstto@nil\foo\@nil
\catcode`@=12
```

Résultat : B

Le mécanisme est schématisé ci-dessous :



La flèche représente le développement du token encadré. Ici donc, le développement de `\expandafter` va provoquer le 1-développement de `\foo` et donne :

```
\firsto@nil Bonjour\@nil
```

Cette expérience faite avec `\firsto@nil` peut être généralisée à n'importe quelle autre macro :

41 - Règle

Lorsqu'on stocke des tokens dans une \⟨*macroA*⟩ (comme cela arrive très souvent), si l'on veut mettre ces tokens dans l'argument d'une autre macro \⟨*macroB*⟩, il *faut* développer \⟨*macroA*⟩ avant que \⟨*macroB*⟩ n'agisse.

■ **Exercice 40**

Au lieu d'une macro, utiliser un nouveau registre de token `\testtoks` pour stocker les caractères « Bonjour ». Comment faut-il alors utiliser `\firstto@nil` pour obtenir la lettre « B » ?

□ **Solution**

Pour extraire le contenu du registre, il faut développer la primitive `\the` lorsqu'elle précède le registre. Ici, au lieu de 1-développer la macro, il faut 1-développer `\the \toks 0` :

Code n° II-68

```
\newtoks\testtoks \testtoks={Bonjour}
\catcode`@=11
Résultat : \expandafter\firstto@nil\the\testtoks\@nil
\catcode`@=12
```

Résultat : B

■

Afin de bien maitriser le développement, abandonnons la macro \firstto@nil pour construire une macro plus généraliste \> qui agit comme un révélateur universel et qui affiche littéralement tout ce qui se trouve entre \> et < :

```
\>ensemble de tokens<
```

On voudrait que \>12^&$~\foo$< affiche « 12^&$~\foo $ ». Pour cela, la primitive \detokenize va bien nous aider. Cette primitive développable admet un argument entre accolades et le transforme en caractères de code de catégorie 12 (et 10 pour l'espace). Par exemple, si on écrit \detokenize{12^&$~\foo$}, on obtient « 12^&$~\foo $ ».

Il est utile de faire quelques remarques concernant \detokenize : une espace est *toujours* rajoutée après une séquence de contrôle ; ici, on peut la voir après \foo. Enfin, l'argument doit contenir un texte où les accolades sont équilibrées. Voici le code qui permet de programmer la macro \> évoquée plus haut :

```
\long\def\>#1<{\detokenize{#1}}
```

Expérimentons cette macro et profitons de \expandafter pour « sauter » la macro \> afin de développer le premier token qui se trouve juste après :

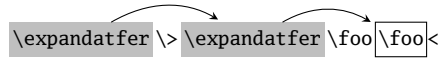
Code n° II-69

```
\def\foo{bonjour}
a) \>\csname foo\endcsname< \par
b) \>\foo< \par
c) \expandafter\>\foo< \par
d) \>\foo\foo< \par
e) \expandafter\>\foo\foo<
```

a) \csname foo\endcsname
b) \foo
c) bonjour
d) \foo \foo
e) bonjour\foo

Chaque fois qu'un \expandafter précède \> (cas c et e), on constate que le développement du \foo a bien lieu tout de suite après la balise. À la dernière ligne, on voit que le premier \foo est développé alors que le second, non concerné par le \expandafter reste tel quel.

Maintenant, essayons de 1-développer le *second* \foo de la dernière ligne tout en laissant le premier intact. Le principe est que le développement d'un premier \expandafter provoque le développement d'un autre situé un token après, et ainsi de suite pour finir par 1-développer le token voulu. Cette capacité que possèdent les \expandafter à se transmettre de proche en proche le développement conduit à appeler ce genre de structure un « pont d'\expandafter » :



Une fois que les \expandafter se sont passé le relai pour développer le second \foo, ceux-ci disparaissent, et le code qui est prêt à être lu est :

```
\>\foo Bonjour<
```

ce que l'on vérifie dans cet exemple :

Code n° II-70

```
\def\foo{Bonjour} \expandafter\>\expandafter\foo\foo<
```

\foo Bonjour

La règle fondamentale est la suivante :

42 - Règle

Pour 1-développer un token précédé de *n* tokens, il faut mettre un `\expandafter` devant chacun de ces *n* tokens.

Par exemple, en partant de `\foo`, si l'on veut définir une macro `\bar` dont le texte de remplacement soit « `\foo Bonjour` », alors on doit mettre un `\expandafter`, représenté par un « • » devant les 4 tokens `\def`, `\bar`, `{` et `\foo` pour provoquer le 1-développement du second `\foo` :

```
•\def•\bar•{•\foo\foo}
```

Code n° II-71

```
\def\foo{Bonjour}
\expandafter\def\expandafter\bar\expandafter{\expandafter\foo\foo}
\meaning\bar
```

macro:->\foo Bonjour

Une application de ceci, très utile en programmation, est l'ajout d'un ⟨*code*⟩ au texte de remplacement d'une macro. Appelons `\addtomacro*` la commande qui admet deux arguments dont le premier est une macro et le second est un texte qui sera ajouté à la fin du texte de remplacement de la macro. On doit pouvoir écrire :

```
\def\foo{Bonjour}
\addtomacro\foo{ le monde}
```

pour qu'ensuite, la macro `\foo` ait comme texte de remplacement « Bonjour le monde ». La méthode consiste à partir de cette définition :

```
\long\def\addtomacro#1#2{\def#1{#1#2}}
```

Si on écrit « `\addtomacro\foo{ le monde}` », le texte de remplacement devient

```
\def\foo{\foo Bonjour}
```

On le voit, le texte de remplacement va conduire à une définition *récursive* de `\foo` puisque le texte de remplacement de `\foo` contient `\foo`. Ouvrons une parenthèse et intéressons-nous à ce qui va se passer.

L'exécution de `\foo` telle que définie ci-dessus va conduire à une erreur de compilation. En effet, à chaque développement, non seulement `\foo` est créée dans la pile, mais 7 tokens (les 7 lettres du mot « Bonjour ») y sont aussi ajoutés. La pile va donc croitre très rapidement, voici son contenu lors des premiers développements :

1. `\foo`
2. `\foo Bonjour`
3. `\foo BonjourBonjour`
4. `\foo BonjourBonjourBonjour`

5. etc.

Cette pile n'étant pas infinie, le maximum va être rapidement atteint :

Code n° II-72

```
\def\foo{\foo Bonjour}% définition récursive
\foo% l'exécution provoque un dépassement de la capacité de la pile
```

```
! TeX capacity exceeded, sorry [input stack size=5000].
```

Augmenter cette taille ne changerait pas l'issue, même si des développements supplémentaires étaient faits.

Fermons la parenthèse et revenons à `\def\foo{\foo Bonjour}`. Afin d'éviter l'écueil du dépassement de la capacité de la pile, il faut 1-développer `\foo` à l'intérieur des accolades pour qu'elle soit remplacée par « Bonjour » et ce, *avant* que `\def` n'entre en action. Comme nous l'avons vu, il suffit de placer un `\expandafter` devant chaque token qui précède ce `\foo`. Voici le code correct qui permet de programmer la macro `\addtomacro` :

Code n° II-73

```
\long\def\addtomacro#1#2{%
  \expandafter\def\expandafter#1\expandafter{#1#2}%
}
\def\foo{Bonjour}
\addtomacro\foo{ le monde}
\meaning\foo
```

```
macro:->Bonjour le monde
```

■ **Exercice 41**

En utilisant `\addtomacro`, écrire une macro `\eaddtomacro*` qui agit comme `\addtomacro` sauf que le second argument est 1-développé. Ainsi, si on écrit

```
\def\foo{Bonjour} \def\world{ le monde}
\eaddtomacro\foo\world
```

alors, le texte de remplacement de la macro `\foo` doit être « Bonjour le monde ».

□ **Solution**

Si nous partons de ceci :

```
\long\def\eaddtomacro#1#2{\addtomacro#1{#2}}
```

le texte de remplacement est de `\eaddtomacro` est

```
\addtomacro#1{#2}
```

Pour que `\addtomacro` accède au texte de remplacement de la macro #2, nous savons qu'à l'intérieur des accolades, #2 doit être 1-développée. Pour ce faire, il faut placer un `\expandafter` devant chacun des 3 tokens qui précèdent #2 :

Code n° II-74

```
\long\def\eaddtomacro#1#2{%
  \expandafter\addtomacro\expandafter#1\expandafter{#2}%
}
\def\foo{Bonjour} \def\world{ le monde}
```

```
\eaddtomacro\foo\world
\meaning\foo
```

macro:->Bonjour le monde

■

Nous allons maintenant construire les mêmes macros sauf qu'elles agiront sur le contenu de registres de tokens. Mais pour atteindre ce but, une règle va nous être utile…

43 - Règle

Les primitives qui doivent immédiatement être suivies d'une accolade ouvrante ont la particularité suivante : elles développent au maximum jusqu'à trouver l'accolade ouvrante.

Nous connaissons :

- `\lowercase` et `\uppercase` ;
- `\detokenize` ;
- `\toks`⟨*nombre*⟩=␣ (ou `\`⟨*nom*⟩=␣ si on est passé par `\newtoks\`⟨*nom*⟩), sachant que le signe = et l'espace qui le suit sont facultatifs.

La conséquence est que l'on peut placer un (ou plusieurs) `\expandafter` entre ces primitives et l'accolade ouvrante pour développer leur argument avant qu'elles n'agissent.

La méthode va consister à tirer parti de cette règle. Si `#1` est un registre de tokens, on va partir de

```
#1= {\the#1#2}
```

et placer un `\expandafter` entre `#1=␣` et l'accolade ouvrante pour 1-développer `\the#1` afin de délivrer le contenu du registre `#1` :

Code n° II-75

```
\long\def\addtotoks#1#2{#1= \expandafter{\the#1#2}}
\newtoks\bar
\bar={Bonjour}% met "Bonjour" dans le registre
\addtotoks\bar{ le monde}% ajoute " le monde"
\the\bar% affiche le registre
```

Bonjour le monde

En copiant la méthode utilisée avec `\eaddtomacro`, il est facile de construire la macro `\eaddtotoks*` qui 1-développe son second argument avant de l'ajouter au contenu du registre à token `#1` :

Code n° II-76

```
\long\def\eaddtotoks#1#2{\expandafter\addtotoks\expandafter#1\expandafter{#2}}
\bar={Bonjour}
\def\macrofoo{ le monde}
\eaddtotoks\bar\macrofoo
\the\bar
```

Bonjour le monde

4.3. Développer encore plus avec \expandafter

Jusqu'à présent, nous nous sommes limités à des \expandafter provoquant des 1-développements, mais on peut 2-développer, voire plus. Pour voir comment, définissons les macros

```
\def\X{Bonjour} \def\Y{\X}
```

et intéressons-nous au problème suivant : essayons de 2-développer le « \Y » qui se trouve entre \> et < dans « \>\Y< » de façon à obtenir l'affichage « Bonjour ».

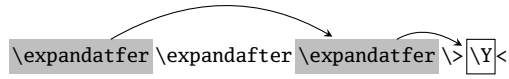
Ce que nous savons faire, c'est 1-développer \Y :

```
\expandafter\>\Y<
```

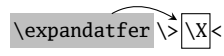
En l'état, ce code ne fait que 1-développer \Y. Pour 2-développer, il faudrait que le \expandafter ci-dessus agisse sur le 1-développement de \Y. On va donc partir du code ci-dessus et faire en sorte que préalablement, le \Y soit 1-développé. Nous savons comment placer les \expandafter pour que la macro \Y soit 1-développée, il suffit d'en mettre un devant chaque token qui précède \Y. Il faut donc placer un \expandafter devant le premier \expandafter et un autre devant \>. On arrive au code suivant :

```
\expandafter\expandafter\expandafter\>\Y<
```

Pour nous assurer que ce code va conduire au résultat escompté, examinons ce qui se passe avec les conventions graphiques vues auparavant. Voici ce qui se passe dans un premier temps :



La macro \Y est développée en \X et on obtient le code « \expandafter\>\X< » dans la pile. Voici maintenant le deuxième temps :



\X se développe bien en « Bonjour » ce qui donne :

```
\>Bonjour<
```

que nous cherchions à obtenir. Vérifions-le sur cet exemple :

Code n° II-77

```
1 \def\X{Bonjour} \def\Y{\X}
2 \expandafter\expandafter\expandafter\>\Y<
```

Bonjour

Ce pont d'\expandatfer a provoqué *deux* développements pour arriver au code final et on peut parler de « pont à plusieurs niveaux ».

Retenons que lorsque deux tokens $\langle x\rangle\langle y\rangle$ sont dans le code, pour 2-développer $\langle y\rangle$ avant de lire $\langle x\rangle$, nous devons placer trois \expandafter devant $\langle x\rangle$ de cette façon :

```
\expandafter\expandafter\expandafter⟨x⟩⟨y⟩
```

Et si nous voulions 3-développer $\langle y \rangle$, il faudrait ajouter des \expandafter dans le code ci-dessus pour, au préalable, 1-développer $\langle y \rangle$. Pour cela, il faudrait placer un \expandafter devant chaque token qui précède $\langle y \rangle$ et donc en ajouter 4 (un devant chacun des 3 \expandafter existants et un devant $\langle x \rangle$). Nous aurions alors 7 \expandafter en tout :

```
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter⟨x⟩⟨y⟩
```

Pour un développement de plus de $\langle y \rangle$ de façon à le 4-développer, il faudrait 1-développer $\langle y \rangle$ dans le code ci-dessus et donc, ajouter 8 \expandatfer de plus (un devant les 7 \expandafter existants plus un devant $\langle x \rangle$) ce qui en ferait 15.

Essayons d'en tirer une règle générale : soit u_k le nombre d'\expandafter placés devant les tokens $\langle x \rangle\langle y \rangle$ qui provoquent le k-développement de $\langle y \rangle$. Si on veut procéder à un développement de plus de $\langle y \rangle$, il faut ajouter $u_k + 1$ \expandafter devant $\langle x \rangle\langle y \rangle$ (un devant chacun des u_k \expandafter existants et un devant $\langle x \rangle\langle y \rangle$). Pour traduire ceci par une relation de récurrence mathématique, il suffit de partir de $u_1 = 1$ puisque un \expandafter placé devant $\langle x \rangle\langle y \rangle$ provoque le 1-développement de $\langle y \rangle$. On a donc :

$$u_1 = 1 \qquad \begin{aligned} u_{k+1} &= u_k + u_k + 1 \\ &= 2u_k + 1 \end{aligned}$$

À l'aide d'une démonstration mathématique facile, on en tirerait que :

$$u_k = 2^k - 1$$

Le raisonnement pourrait facilement être généralisé si plusieurs tokens précédaient $\langle y \rangle$. La règle générale est la suivante et il est bon de s'en souvenir :

44 - Règle

Pour provoquer le k-développement d'un token $\langle y \rangle$ précédé par n autres tokens, il faut placer $2^k - 1$ \expandafter devant chacun des n tokens qui le précèdent.

■ **Exercice 42**

Reprenons les macros suivantes :

```
\def\X{Bonjour}
\def\Y{\X}
```

Placer les \expandafter dans le code ci-dessous pour que le texte de remplacement de la macro \foo soit « Bonjour » :

```
\def\foo{\Y}
```

□ **Solution**

La réponse se trouve dans la règle précédente : il faut 2-développer \Y et donc ajouter $2^2 - 1$, c'est-à-dire 3 \expandafter devant chaque token qui précède \Y. Ces tokens sont au nombre de 3 (« \def », « \foo » et « { ») et donc, il faut remplacer chaque « • » par 3 \expandafter dans ce code :

```
•\def•\foo•{\Y}
```

Cela donne un code un peu indigeste :

```
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\foo
\expandafter\expandafter\expandafter{\Y}
```

■

■ **Exercice 43**

Placer correctement des \expandafter dans la dernière ligne du code ci-dessous pour que \bar ait comme texte de remplacement « Bonjour le monde » :

```
\def\foo{Bonjour}
\def\world{ le monde}
\def\bar{\foo\world}
```

□ **Solution**

Si on part de

```
\def\bar{\foo\world}
```

il faut 1-développer \foo et 1-développer \world dans le texte de remplacement de \bar.

Nous allons construire un pont d'\expandafter en deux passes : la première 1-développera une des deux macros et la deuxième 1-développera l'autre.

Dans un cas comme celui-ci qui est un cas particulier d'un cas plus général, il faut commencer par placer des \expandafter pour 1-développer la macro la plus à gauche. Ceci étant fait, la passe suivante consistera à placer des \expandatfer pour développer la macro suivante. Et ainsi de suite pour finir, lors de la dernière passe, par le développement de la macro la plus à droite. On progresse donc de gauche à droite, c'est-à-dire dans l'ordre inverse de celui dans lequel elles seront développées par TEX lorsqu'il lira le code.

Ici donc, la première passe consiste à placer des \expandafter pour 1-développer \foo, ce qui revient à en placer un devant chaque token qui précède \foo :

```
\expandafter\def\expandafter\foo\expandafter{\foo\world}
```

Dans une deuxième passe, nous allons 1-développer \world et placer aussi un \expandafter devant chaque token qui la précède, en tenant compte de ceux déjà mis en place :

```
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\foo
\expandafter\expandafter\expandafter
{\expandafter\foo\world}
```

Bien que ne répondant pas aux contraintes de l'énoncé, la macro \eaddtomacro peut se révéler utile dans un cas comme celui-ci. En effet, elle améliore la lisibilité du code puisqu'elle évite ces nombreux \expandafter. Une autre façon de faire aurait été :

```
\def\bar{}%
\eaddtomacro\bar\foo
\eaddtomacro\bar\world
```

■

La primitive \expandafter intervenant au niveau des *tokens*, il est intéressant en programmation de disposer d'une macro, appelons-la \expsecond*, qui intervient au niveau des *arguments*. On voudrait qu'elle soit capable de sauter un argument entier pour 1-développer le premier token de l'argument qui suit. On pourrait donc écrire :

\expsecond{⟨*arg1*⟩}{⟨*arg2*⟩}

pour que le code finalement lu par TEX soit

⟨*arg1*⟩{*⟨*arg2*⟩}

où l'étoile indique que le premier token de ⟨*arg2*⟩ est 1-développé.

Exercice 44

Construire la macro \expsecond en utilisant la macro auxiliaire \swaparg* qui échange 2 arguments en mettant le premier entre accolades :

```
\long\def\swaparg#1#2{#2{#1}}
```

Solution

La méthode consiste à partir de

```
\long\def\expsecond#1#2{\swaparg{#2}{#1}}
```

qui, après que \swaparg ait joué son rôle, donnerait « #1{#2} ». Pour que le premier token de #2 soit 1-développé, il faut placer des \expandafter dans le texte de remplacement de \expsecond pour provoquer ce développement. Cela donne :

```
\long\def\swaparg#1#2{#2{#1}}
\long\def\expsecond#1#2{\expandafter\swaparg\expandafter{#2}{#1}}
```

■

Exercice 45

Combien de développements doit subir \expsecond{⟨*arg1*⟩}{⟨*arg2*⟩} pour que le code qui soit effectivement à lire soit ⟨*arg1*⟩{*⟨*arg2*⟩} ?

Placer correctement des \expandafter dans le 2e ligne de ce code

```
\def\X{Bonjour}
\>\expsecond{\X}{\X}<
```

pour qu'il affiche « \X Bonjour ». La macro \>, programmée à la page 106, affiche tel quel le code qu'elle rencontre jusqu'au prochain <.

Solution

Développons \expsecond{⟨*arg1*⟩}{⟨*arg2*⟩} comme le fait TEX :

1. \expandafter\swaparg\expandafter{⟨*arg2*⟩}{⟨*arg1*⟩}
2. \swaparg{*⟨*arg2*⟩}{⟨*arg1*⟩} (les \expandafter ont joué leur rôle)
3. ⟨*arg1*⟩{*⟨*arg2*⟩}

Il faut donc 3 développements.

Dans « \>\expsecond{\X}{\X}< », si l'on veut 3-développer \expsecond, il faudra mettre $2^3 - 1 = 7$ \expandafter devant \> :

Code n° II-78

```
\long\def\swaparg#1#2{#2{#1}}
\long\def\expsecond#1#2{\expandafter\swaparg\expandafter{#2}{#1}}
\def\X{Bonjour}
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter
\>\expsecond{\X}{\X}<
```

```
\X {Bonjour}
```

■

■ **Exercice 46**

Dans quels cas \expsecond ne fonctionne pas comme attendu ?

□ **Solution**

Si l'on examine la définition

```
\long\def\expsecond#1#2{\expandafter\swaparg\expandafter{#2}{#1}}
```

on constate que le 1-développement de l'argument #2 a lieu alors que cet argument est déplacé et n'est plus en dernier, c'est-à-dire qu'il ne précède pas le code qui suit. Dès lors, si cet argument doit agir sur le code qui suit (avec \gobone, \identity, \firstoftwo ou \secondoftwo), il n'est plus en mesure de le faire. Dans l'exemple ci-dessous, on souhaite 1-développer \gobone pour donner à \foo le texte de remplacement « B » :

Code n° II-79

```
\expsecond{\def\foo}\secondoftwo{A}{B}
```

```
Argument of \secondoftwo has an extra }
```

Le 2e argument de \expsecond est \secondoftwo donc, après avoir 1-développé \expsecond dans « \expsecond{\def\foo}\secondoftwo{A}{B} », voici ce que l'on obtient :

```
\expandafter\swaparg\expandafter{\secondoftwo}{\def\foo}
```

Lorsque \secondoftwo est 1-développé, en cherchant son premier argument, il voit « } » et c'est là que TEX proteste car une accolade fermante ne peut pas être le début d'un argument. Pour que « B » soit le texte de remplacement de \foo, il aurait fallu que « \secondoftwo{A}{B} » soit le 2e argument de \expsecond :

Code n° II-80

```
\expsecond{\def\foo}{\secondoftwo{A}{B}}
\meaning\foo
```

```
macro:->B
```

■

La macro \expsecond peut être utilisée si son premier argument est une macro :

```
\expsecond\⟨macro⟩{⟨argument⟩}
```

Dans ce cas, \expsecond 1-développe l'argument de la macro avant d'exécuter la macro. C'est pourquoi la macro \expsecond a un alias « \exparg⋆ » défini par \let dont le nom suggère davantage que l'argument d'une macro a été développé.

■ **Exercice 47**

En reprenant le même principe que \exparg, définir une macro

```
\exptwoargs\⟨macro⟩{argument 1}{argument 2}
```

qui 1-développe les 2 premiers tokens des deux arguments avant de lancer la \⟨*macro*⟩

□ **Solution**

Deux appels imbriqués à \exparg conduisent au résultat :

```
\def\exptwoargs#1#2#3{\expsecond{\expsecond{#1}{#2}}{#3}}
```

Comme l'argument #3 se trouve une seule fois dans le texte de remplacement de la macro et à la fin de celui-ci, il est possible de ne pas le faire lire par \exptwoargs :

Code n° II-81

```
\def\exptwoargs#1#2{\expsecond{\expsecond{#1}{#2}}}
\def\foo{Bonjour}\def\bar{ le monde}
\def\displaytwoargs#1#2{\detokenize{#1#2}}% affiche tels quels les arguments
\exptwoargs\displaytwoargs\foo\bar
```

Bonjour le monde

■

Une macro plus générale, capable de développer plusieurs arguments sera programmée plus loin (voir page 394).

4.4. La primitive \edef

On a vu que \expandafter permettait de contrôler finement la profondeur de développement moyennant, c'est vrai, de longs et laborieux ponts d'\expandafter. Mais il existe des endroits où TEX développe au maximum ce qu'il rencontre. Attention, il s'agit bien d'un développement et non pas d'une exécution. Le plus célèbre de ces endroits est le texte de remplacement des macros définies avec la primitive \edef. Cette primitive fonctionne comme sa sœur \def sauf que tout ce qui se trouve à entre les accolades est développé au maximum et ce qui en résulte est le texte de remplacement de la macro définie par \edef. Essayons :

Code n° II-82

```
\def\aa{Bonjour}\def\bb{\aa}
\def\cc{ le monde}
\edef\foo{\bb\cc}
\meaning\foo
```

macro:->Bonjour le monde

On constate que le texte de remplacement de \foo contient le 2-développement de \bb et le 1-développement de \cc. Ces développements sont les plus profonds que l'on puisse faire puisqu'ensuite on ne trouve plus que des lettres ou des espaces. Le texte de remplacement de \truc est bien « Bonjour le monde ».

Comme pour \def, on peut aussi spécifier qu'une macro définie avec \edef admet des arguments, et on utilise pour cela la même syntaxe #⟨*chiffre*⟩, aussi bien pour le texte de paramètre que pour le texte de remplacement. Le développement maximal fait lors de la définition concerne tout ce qui se trouve dans le texte de remplacement *à l'exception* des emplacements des arguments qui ont la syntaxe #⟨*chiffre*⟩. Les arguments, qui seront lus plus tard lorsque la macro sera appelée, seront insérés tels quels à la place de #⟨*chiffre*⟩ : ils ne sont donc pas soumis au développement maximal.

On peut le constater ici où la macro \ee est définie avec un \edef et admet un argument. À l'aide d'\expandafter, on provoque le 1-développement de la macro \ee dont l'argument est la macro \dd. On peut constater que cet argument « \dd » reste non développé bien qu'il soit l'argument d'une macro définie avec \edef :

Code n° II-83

```
\def\aa{Bonjour } \def\bb{\aa} \def\cc{le } \def\dd{ monde}
\edef\ee#1{\bb\cc#1 !!!}
\meaning\ee\par
```

```
\expandafter\>\ee{\dd}<
```

```
macro:#1->Bonjour le #1 !!!
Bonjour le \dd  !!!
```

45 - Règle

La primitive \edef a la même syntaxe et les mêmes actions que \def, c'est-à-dire qu'elle stocke un texte de remplacement associé à une séquence de contrôle. La différence réside dans le fait que la totalité du texte de remplacement est développée *au maximum* avant que l'assignation soit faite.

Le développement s'applique au texte de remplacement *hors arguments* symbolisés par #⟨*chiffre*⟩. Ceux-ci, qu'ils soient délimités ou pas, seront insérés tels qu'ils seront lus dans le texte de remplacement (qui lui aura été préalablement développé).

La primitive \xdef effectue les mêmes actions que \edef sauf que la définition faite est globale.

4.4.1. \noexpand pour contrer \edef

Cette primitive \edef se révèle fort utile, et rend de grands services. Mais dans son argument où tout est développé au maximum, on voudrait pouvoir bloquer le développement d'une séquence de contrôle. Par exemple, si l'on écrit

```
\edef\foo{\aa\bb\cc}
```

comment laisser \aa et \cc se développer au maximum tout en bloquant le développement de \bb ? C'est là qu'une nouvelle primitive entre en jeu, il s'agit de \noexpand.

46 - Règle

La primitive développable \noexpand bloque le développement du token x qui la suit. Elle a pour effet, lorsqu'elle se 1-développe, d'être le token x lui-même. Si x est une séquence de contrôle qui aurait été développée, x est rendu temporairement égal à \relax.

Code n° II-84

```
\def\foo{123xyz}
a) signification du 1-développement d'un \litterate-\noexpand- :
   \expandafter\meaning\noexpand\foo

b) exécution d'un \litterate-\noexpand- : \noexpand\foo% identique à \relax

c) 1-développement de \litterate-\noexpand- dans le texte de remplacement de
   \litterate-\bar- :
   \expandafter\def\expandafter\bar\expandafter{\noexpand\foo}
   \meaning\bar
```

```
a) signification du 1-développement d'un \noexpand : \relax
b) exécution d'un \noexpand :
c) 1-développement de \noexpand dans le texte de remplacement de \bar :  macro:->\foo
```

Le dernier cas montre qu'entre le moment du 1-développement de \noexpand \foo et le moment où TEX lit ce 1-développement, plusieurs choses sont faites (lecture et exécution du \def suivie de la lecture \bar et de l'accolade ouvrante). Entre temps, le développement initié par le pont d'\expandafter s'est arrêté, signant la mort du \relax temporaire qui est donc redevenu \foo.

Dans le code « \edef\foo{\aa\bb\cc} », tentons maintenant de bloquer le développement de \bb :

Code n° II-85

```
1 \def\aa{Bonjour}\def\bb{\aa}\def\cc{\bb}
2 a) \edef\foo{\aa\noexpand\bb\cc}\meaning\foo  \qquad b) exécution : \foo\par
3 c) \edef\foo{\aa\noexpand{\aa\cc}}\meaning\foo\qquad d) exécution : \foo
```

a) macro:->Bonjour\bb Bonjour b) exécution : BonjourBonjourBonjour
c) macro:->Bonjour{BonjourBonjour} d) exécution : BonjourBonjourBonjour

Tout se passe comme prévu au a) pour la première définition de \foo avec \edef : le \noexpand qui précède \bb empêche bien son développement.

À la deuxième ligne (cas c), on a tenté bien maladroitement de bloquer le développement de ce qui se trouve entre accolades. Ceci échoue puisque \noexpand bloque le développement du *token* qui suit et qui est dans ce cas « { ». Cette accolade ouvrante n'a d'ailleurs nul besoin d'être protégée d'un développement puisqu'elle n'est pas développable (et elle se développe donc en elle-même). Pour bloquer le développement de plusieurs séquences de contrôle, il *faut* mettre un \noexpand devant chacune d'elles !

4.4.2. \unexpanded pour contrer \edef

Multiplier les \noexpand et en écrire autant qu'il y a de tokens à protéger peut être lourd, voire impossible si on ne connait pas le nombre de tokens lorsqu'on manipule l'argument d'une macro par exemple. Pour contourner cette difficulté, on peut utiliser la primitive \unexpanded de ε-TEX dont le développement est de bloquer le développement du texte entre accolades qui suit la primitive.

Voici deux façons de faire équivalentes, la première en plaçant un \noexpand devant \bb et \aa la seconde en mettant ces deux tokens dans l'argument de la primitive \unexpanded :

Code n° II-86

```
1 \def\aa{Bonjour}\def\bb{Au revoir}
2 a) \edef\truc{\aa\noexpand\bb\noexpand\aa\bb}\meaning\truc \par
3 b) \edef\truc{\aa\unexpanded{\bb\aa}\bb}\meaning\truc
```

a) macro:->Bonjour\bb \aa Au revoir
b) macro:->Bonjour\bb \aa Au revoir

La primitive \unexpanded, tout comme \detokenize et d'autres, fait partie des primitives qui doivent être suivies d'une accolade ouvrante. Ces primitives initient un développement maximal jusqu'à ce qu'elles rencontrent une accolade ouvrante (voir règle page 109). On peut donc intercaler un ou plusieurs \expandafter entre ces primitives et l'accolade ouvrante de façon à procéder au développement du (ou des) premiers tokens du texte entre accolades.

4.4.3. \the\toks⟨*nombre*⟩ pour contrer \edef

Il existe un autre moyen qui ne fait pas appel à ε-TEX et qui permet de bloquer le développement dans un \edef. Il s'agit du contenu d'un registre de tokens : dans le texte de remplacement d'un \edef, le contenu d'un registre de tokens obtenu par le développement de \the \toks ⟨*nombre*⟩ (ou par « \the⟨*nom d'un registre*⟩ ») n'est pas développé.

Voici l'exemple précédent traité avec un registre à token (ici, nous prenons le registre n° 0) :

Code n° II-87

```
\def\aa{Bonjour}\def\bb{Au revoir}
\toks0={\aa\bb}
\edef\foo{\aa\the\toks0 \bb}
\meaning\foo
```

macro:->Bonjour\aa \bb Au revoir

4.4.4. Protéger une macro avec \protected

Tout comme on peut déclarer une macro « \long » ou « \global » lors de sa définition, on peut aussi déclarer qu'on souhaite définir une macro « protégée ». On entend « protégée contre le développement maximal dans l'argument d'une primitive ». Pour ce faire, il faut utiliser la primitive de ε-TEX « \protected » et la placer avant le \def (ou \edef) qui effectue la définition. La macro ainsi définie ne se développera pas dans un \edef :

Code n° II-88

```
\def\aa{Bonjour}
\protected\def\bb{ le}% \ bb est protégée !
\def\cc{ monde}

\edef\foo{\aa\bb\cc}
Signification de \string\foo~: \meaning\foo \par
Exécution de \string\foo~: \foo
```

Signification de \foo : macro:->Bonjour\bb monde
Exécution de \foo : Bonjour le monde

Il faut bien comprendre que la macro est protégée \bbd'un développement est maximal, mais pas de l'action de \expandafter. Voici comment on peut 1-développer \bb dans le texte de remplacement de \edef avec un \expandafter :

Code n° II-89

```
\def\aa{Bonjour}
\protected\def\bb{ le}
\def\cc{ monde}

\edef\foo{\expandafter\aa\bb\cc}% le \expandafter 1-développe \bb
\meaning\foo
```

macro:->Bonjour le monde

4.4.5. Les moyens de bloquer le développement maximal

Il faut savoir est que certaines primitives (\edef, \xdef, \csname ...\endcsname que nous connaissons mais aussi \message, \write et \errmessage que nous verrons plus tard) développent au maximum ce qui est dans leur argument [1]. Les méthodes pour contrer ce développement maximal valent aussi bien pour \edef que pour n'importe quelle autre de ces primitives. Voici le résumé des méthodes que nous avons vues :

47 - Règle

Il existe 4 moyens de bloquer le développement maximal dans les arguments de certaines primitives :

1. mettre un \noexpand devant chaque token dont on veut bloquer le développement ;
2. placer les tokens dont on veut bloquer le développement dans l'argument de la primitive \unexpanded de ε-TEX ;
3. stocker au préalable tous les tokens dont on veut bloquer le développement dans un ⟨*registre de token*⟩ et écrire dans l'argument de la primitive \the⟨*registre de token*⟩ ;
4. lorsqu'on définit une macro, il est possible de faire précéder le \def de la primitive \protected de ε-TEX pour empêcher le développement maximal de cette macro.

4.5. Code purement développable

On pourrait penser que la primitive \edef peut simuler une « exécution » pour mettre dans la séquence de contrôle qu'elle définit le « résultat » de l'exécution du texte de remplacement [2]. Il s'agit généralement d'une erreur, et pour comprendre pourquoi, il faut parler de ce qu'est du « code purement développable ».

Afin de mieux comprendre de quoi il s'agit, prenons un exemple simple, n'ayant aucun intérêt si ce n'est pédagogique. Supposons que la macro \foo a comme texte de remplacement « abc », ce qui suppose qu'un \def\foo{abc} a été rencontré auparavant. Voici un code élémentaire qui définit une macro \foo et qui l'exécute :

Code n° II-90

```
1 \def\foo{Bonjour}% définition
2 \foo% exécution
```

Bonjour

Ce code de deux lignes n'est pas purement développable. Essayons de comprendre pourquoi en le mettant dans un \edef pour définir la macro \bar :

```
\edef\bar{\def\foo{Bonjour}\foo}
```

1. Il y a aussi \special, \mark et \marks qui ne seront pas étudiées dans ce livre. Pour être complet, il faut signaler que le développement maximal est aussi en marche dans un alignement lorsque TEX recherche un \omit ou un \noalign.

2. On peut stocker l'affichage résultant d'un code arbitraire à l'aide des registres de boites que nous verrons plus tard.

Voici ce que va faire TEX lorsqu'il va développer au maximum le code dans le texte de remplacement de \edef :

1. la primitive non développable \def se développe en elle-même et reste donc inchangée ;
2. les deux occurrences de \foo vont être développées et donc remplacées par « abc ».

Le texte de remplacement de \bar sera donc :

```
\def abc{Bonjour}abc
```

Vérifions-le :

Code n° II-91

```
\def\foo{abc}
\edef\bar{\def\foo{Bonjour}\foo}
\meaning\bar
```

```
macro:->\def abc{Bonjour}abc
```

Ce texte de remplacement comporte une erreur, car après un \def, on *doit* trouver une séquence de contrôle (et non pas des caractères normaux comme a, b et c). Mais cette erreur n'apparait pas lors de la définition de\bar car TEX effectue les assignations sans procéder à l'analyse du texte de remplacement. Ce n'est que si l'on cherche à exécuter la macro \bar, et donc à exécuter son texte de remplacement que TEX émettra le message d'erreur « `Missing control sequence` ».

■ **Exercice 48**

Décrire ce qui se passerait si l'on écrivait le code ci-dessous alors que la macro \foo n'est pas définie.

```
\edef\bar{\def\foo{Bonjour}\foo}
```

□ **Solution**

Lors du développement maximal de \foo, celle-ci n'étant pas définie, TEX émettrait le message d'erreur « `Undefined control sequence` ». ■

La conclusion est donc sans appel : que \foo soit définie ou pas, le code n'est pas purement développable. On pourrait rétorquer que pour ce code soit purement développable, il suffirait de bloquer le développement de \foo en se servant de la primitive \noexpand :

Code n° II-92

```
\def\foo{abc}
\edef\bar{\def\noexpand\foo{Bonjour}\noexpand\foo}
Signification : \meaning\bar\par
Exécution : \bar
```

```
Signification : macro:->\def \foo {Bonjour}\foo
Exécution : Bonjour
```

Peut-on dire que le code, enrichi de \noexpand, est purement développable ? En fait, tout dépend de ce que l'on attendait ! Si l'on voulait simplement construire un code exécutable stocké dans une macro, alors oui. Mais dans la grande majorité des cas, on attend qu'un code purement développable se réduise, lors de son développement maximal, aux caractères affichés si ce code était exécuté.

48 - Règle

Un code est purement développable si son développement maximal se réduit aux caractères qui auraient été affichés s'il avait été exécuté par TEX.

Dans la majorité des cas, un tel code ne doit être constitué que caractères (ceux que l'on souhaite dans l'affichage final) ainsi que de séquences de contrôle ou caractères actifs *développables*. Citons :

- du côté des primitives : `\expandafter`, `\number`, `\the`, `\string` ainsi que les primitives de ε-TEX `\detokenize` et `\unexpanded`. Tous les tests de TEX que nous verrons plus tard sont développables. Il y a aussi deux primitives de ε-TEX, `\numexpr` et `\dimexpr` qui seront étudiées dans la partie suivante ;
- les macros définies par l'utilisateur, qu'elles soient à arguments délimités ou pas, pourvu que leur développement maximal soit constitué des caractères que l'on souhaite dans l'affichage final.

4.6. Propager le développement

4.6.1. Lier des zones de développement maximal

Dans certaines zones bien spécifiques, le développement est maximal et donc, dans ces zones, le code doit être purement développable.

L'expression « propager le développement » signifie mettre en œuvre des méthodes pour que ces zones de développement maximal se transmettent de proche en proche le développement. Le but est de développer des territoires très éloignés de l'endroit où le premier développement a eu lieu.

Parfois, du code TEX est constitué de zones de développement maximal entrecoupées de zones « normales » où ce développement maximal n'existe pas. Le schéma ci-dessous représente une de ces situations où les zones de développement maximal sont encadrées et les tokens des zones normales sont représentés par « $*$ ». La tête de lecture de TEX est représentée par $\ggg$:

Il est possible, sans déplacer la tête de lecture, de 1-développer le token x. Pour cela, on va amorcer le développement de la première zone de développement maximal en mettant un `\expandafter` juste avant le premier token « $*$ ». Par la suite, il faudra exporter le développement hors des zones encadrées à l'aide d'un `\expandafter` placé en dernière position. Un pont d'`\expandafter` prend ensuite le relai dans la zone normale pour communiquer le développement à la zone encadrée suivante. À la fin de la dernière zone, un pont d'`\expandafter` permettra d'atteindre le token x pour le 1-développer. En notant « $\bullet$ » la primitive `\expandafter`, voici ce que devient le schéma :

■ **Exercice 49**

Que faudrait-il modifier dans ce schéma pour provoquer le 2-développement du token x ?

□ **Solution**
Tout restant égal par ailleurs, il faut initier un pont d'\expandafter à deux passes à la fin de la dernière zone de développement maximal, c'est-à-dire mettre 3 \expandafter au lieu d'un seul :

4.6.2. Étude d'un cas

Pour illustrer ce propos, passons à un cas plus concret, quoiqu'encore très théorique. Supposons qu'avant de lire les macros \foo et \bar, nous devions provoquer le 2-développement de la macro \wee et le 1-développement de \fin dans le code suivant :

```
\foo\bar\wee\fin
```

Le nombre d'\expandafter nécessaires est assez important. Essayons de les compter ! Plaçons mentalement les \expandafter pour provoquer tour à tour le développement des macros prises *de gauche à droite*, c'est-à-dire que nous allons d'abord chercher à développer \wee qui est la plus à gauche des deux macros. Ensuite, nous partirons du code obtenu et chercherons à développer \fin.

Pour 2-développer \wee, il faut placer $2^2 - 1 = 3$ \expandafter devant chaque token qui la précède ce qui fait $3 \times 2 = 6$ \expandafter, notés • :

```
••••\foo••••\bar\wee\fin
```

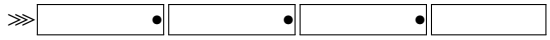
Dans ce code, le nombre de tokens qui précèdent \fin devient égal à 9 : nous avons les 6 \expandafter que nous venons de placer et les 3 séquences de contrôle \foo, \bar et \wee. Pour 1-développer \fin, il faut mettre un \expandafter devant chaque token qui précède \fin ce qui fait 9 \expandatfer de plus et porte donc leur nombre total à 6 + 9 = 15. Voici le code obtenu :

```
•••••••\foo•••••••\bar•\wee\fin
```

Venons-en maintenant aux zones de développement maximal et au lieu d'écrire directement le nom des 4 macros, aidons-nous de la paire \csname et \endcsname :

```
\csname foo\expandafter\endcsname
\csname bar\expandafter\endcsname
\csname baz\expandafter\endcsname
\csname fin\endcsname
```

Le rôle des \expandafter mis à chaque fin de zone de développement maximal (sauf la dernière) est de transmettre le développement en dehors de la zone de développement maximal pour toucher le \csname suivant qui commande à son tour le développement maximal jusqu'au prochain \endcsname. L'\expandafter se trouvant en fin de zone transmet à son tour le développement au \csname suivant, et ainsi de suite jusqu'au dernier \endcsname qui n'est pas précédé d'un \expandafter et stoppe donc cet enchainement. La structure est clairement celle que l'on voyait dans le schéma de la section précédente : externaliser le développement pour déclencher le développement d'une nouvelle zone de développement maximal pour, en réitérant la manœuvre, transmettre très loin la flamme du développement. Voici le schéma qui traduirait cette situation (ici, aucun token ne se trouve entre les zones de développement maximal) :



Le code donné ci-dessus se 1-développe en « \foo\bar\wee\fin ». En voici la preuve :

Code n° II-93

```
\long\def\>#1<{\detokenize{#1}}
\expandafter\>% provoque le 1-développement de :
\csname foo\expandafter\endcsname
\csname bar\expandafter\endcsname
\csname wee\expandafter\endcsname
\csname fin\endcsname<
```

\foo \bar \wee \fin

Dans le code ci-dessus, aucune séquence de contrôle n'est développée. Pour provoquer le 1-développement de \fin, il suffit de mettre 3 \expandafter après « wee » au lieu d'un seul. En suivant le même raisonnement, pour 2-développer \wee, il faut 3-développer \csname baz\endcsname et donc placer 7 \expandafter après « bar ». Cela donne le code :

```
\csname foo\expandafter\endcsname
\csname bar\expandafter\expandafter\expandafter\expandafter
   \expandafter\expandafter\expandafter\endcsname
\csname wee\expandafter\expandafter\expandafter\endcsname
\csname fin\endcsname
```

Pour nous assurer que ce code fonctionne correctement, définissons \wee pour que son 2-développement soit « XX » et \fin pour que son 1-développement soit « YY » :

Code n° II-94

```
\long\def\>#1<{\detokenize{#1}}
\def\fin{YY} \def\wee{\weeA} \def\weeA{XX}
\expandafter\>%
\csname foo\expandafter\endcsname
\csname bar\expandafter\expandafter\expandafter\expandafter
          \expandafter\expandafter\expandafter\endcsname
\csname wee\expandafter\expandafter\expandafter\endcsname
\csname fin\endcsname<
```

\foo \bar XXYY

Le premier \expandafter qui précède \> suffit à atteindre, de zone de développement en zone de développement, la toute fin de ce code avant même que le premier nom de séquence de contrôle \foo ne soit formé.

■ Exercice 50

Où faut-il placer des \expandafter dans « \>\foo\bar\wee\fin< » pour obtenir la même chose que dans l'exemple précédent ?

□ Solution

Il faut les 15 \expandafter déjà vus sauf qu'il faut sauter le token \> au départ. On a déjà calculé qu'il faut 7 \expandafter avant \foo, il en faut également 7 pour sauter \> qui vient en première position. Cela nous fait 15 + 7 = 22 \expandafter (au lieu de 12 avec \csname et \endcsname) :

Code n° II-95

```
\long\def\>#1<{\detokenize{#1}}
\def\fin{YYY} \def\wee{\weeA} \def\weeA{XXX}
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter
\>%
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter
\foo
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter
\bar
\expandafter\wee\fin<
```

\foo \bar XXXYYY

■

4.6.3. Qu'est-ce qu'un nombre ?

La notion de nombre (compris comme nombre *entier*) étant étroitement liée à celle de développement, il est temps d'aborder ce sujet.

Tout d'abord, il est faux de croire que lorsque TEX exécute le code

```
le nombre 123 est impair
```

il *lit* le nombre 123. Il ne fait qu'afficher séquentiellement les caractères « 1 » puis « 2 » et « 3 » sans pour autant que ces caractères ne soient isolés et analysés comme un tout formant un nombre entier. Pour que TEX déclenche la lecture d'un nombre, il faut qu'il se trouve à un endroit où il s'attend à en lire un.

Nous avons déjà rencontré plusieurs cas où TEX doit lire un nombre. C'est le cas pour les primitives \catcode, \lccode et \uccode dont la syntaxe complète est

$$\left.\begin{array}{l}\texttt{\textbackslash catcode}\\ \texttt{\textbackslash lccode}\\ \texttt{\textbackslash uccode}\end{array}\right\}\langle \textit{nombre}\rangle\texttt{= }\langle \textit{nombre}\rangle$$

où le signe = et l'espace qui le suit sont facultatifs.

Nous avons vu qu'après la primitive \toks, TEX attend le ⟨*nombre*⟩ correspondant au registre de tokens auquel on veut accéder. Nous savons également qu'après la primitive \number, TEX s'attend à lire un nombre pour le convertir en chiffres arabes et en base 10. Un ⟨*nombre*⟩ est également attendu après la primitive \char et le tout se traduit par l'affichage du caractère dont le code de caractère est ⟨*nombre*⟩.

Pour être complet concernant les caractères affichés via \char, il est possible d'utiliser la primitive \chardef dont la syntaxe est

$$\texttt{\textbackslash chardef\textbackslash}\langle \textit{macro}\rangle\texttt{= }\langle \textit{nombre}\rangle$$

Comme pour \catcode, \lccode et \uccode, le signe « = » et l'espace qui le suit sont facultatifs. L'action qu'effectue ce code est de rendre la \⟨*macro*⟩ équivalente à \char⟨*nombre*⟩. L'équivalence dont il est question ici ressemble à celle effectuée avec \let. Ainsi, si on écrit :

```
\chardef\foo= 65
```

alors, la macro \foo sera équivalente à \char65 et produira un « A » à l'affichage.

Une \⟨*macro*⟩ définie avec \chardef revêt une dualité intéressante : lorsque TEX s'attend à lire un nombre, cette \⟨*macro*⟩ est alors comprise comme un nombre. Avec la macro \foo ci-dessus, le nombre lu serait 65.

Code n° II-96

```
\chardef\foo65
a) |\foo|\qquad % employée seule, \foo affiche un A
b) |\number\foo|\qquad % si TeX lit un nombre, \foo est le nombre 65
```

a) |A| b) |65|

Comme il n'y a que 256 caractères possibles [3], les primitives \char et \chardef ne peuvent prendre en compte que des nombres compris entre 0 et 255. Le mode mathématique requiert d'afficher beaucoup plus de caractères que ces 256, aussi il existe les primitives \mathchar et \mathchardef qui sont au mode mathématique ce que sont \char et \chardef au mode texte. Le caractère dual d'une macro définie par \mathchardef existe sauf que le nombre qu'elle peut représenter est compris entre 0 et 32 767 [4] et lorsqu'elle est utilisée pour afficher un caractère, le mode mathématique doit être en vigueur.

Code n° II-97

```
\mathchardef\foo4944
a) |$\foo$|\qquad % en mode math, affiche le signe "somme" (uniquement en mode maths)
b) |\number\foo|\qquad % si TeX lit un nombre, \foo est 4944
```

a) |∑| b) |4944|

Il est temps de donner la définition d'un nombre et montrer en quoi elle est liée au développement.

49 - Règle

Un « nombre » est un entier relatif compris entre $-2^{31}+1$ et $2^{31}-1$, c'est-à-dire entre −2 147 483 647 et 2 147 483 647. Tout nombre hors de cet intervalle provoque une erreur de compilation.

Lorsque TEX s'attend à lire un nombre, il entre dans une phase de développement maximal. Dans les résidus de ce développement, il cherche tout d'abord une série de signes optionnels de catcode 12 : « + » et « − ». Le signe du nombre final dépendra de la parité du nombre de signes « − » lus au total. Ensuite, plusieurs cas peuvent se présenter :

1. le nombre est explicitement écrit en
   - base 8 sous la forme '⟨*signes*⟩ ;
   - base 10 sous la forme ⟨*signes*⟩ ;
   - base 16 sous la forme "⟨*signes*⟩,

   où les ⟨*signes*⟩ sont une suite de chiffres de catcode 12 allant de 0 à 7 pour la base 8, de 0 à 9 pour la base 10 et pour la base 16, de 0 à 9 auxquels s'ajoutent les lettres *majuscules* de catcode 11 ou 12 allant de A à F.

3. Pour les moteurs 8 bits, car les moteurs UTF8 en ont beaucoup plus.
4. Voir le TEXbook, page 181.

TEX stoppe la lecture du nombre au premier token qui n'est pas un ⟨*signe*⟩ valide. Si ce token est un espace, il sera absorbé. Le nombre lu comprend donc la plus longue série de ⟨*signes*⟩ valides possible ;
2. un « compteur » se présente (nous verrons les compteurs plus loin). Le nombre qu'il contient est lu et la lecture du nombre est stoppée ;
3. un « registre de dimension » se présente. Dans ce cas, la dimension est convertie en entier (nous verrons comment plus loin) et la lecture du nombre s'achève ;
4. une séquence de contrôle définie avec `\chardef` pour les nombres compris entre 0 à 255 ou avec `\mathchardef` pour ceux compris entre 0 à 32767 se présente, le nombre correspondant est lu et la lecture du nombre est stoppée ;
5. un nombre écrit sous la forme ‘⟨*car*⟩ ou ‘\⟨*car*⟩ se présente, où « ‘ » est l'apostrophe inverse. Le nombre lu est le code de caractère de ⟨*car*⟩. Le *développement maximal se poursuit* jusqu'au premier token non développable qui signe la fin de la lecture du nombre. *Si ce token est un espace, il est absorbé.*

Plain-TEX définit la séquence de contrôle `\z@` comme étant un registre de dimension égal à la dimension nulle `0pt`. Cette séquence de contrôle est souvent employée en programmation, car elle présente 3 avantages :

1. elle est convertie en l'entier 0 lorsque TEX lit un nombre entier ;
2. elle est la dimension `0pt` lorsque TEX lit une dimension ;
3. dans les deux cas précédents, elle stoppe la lecture et le développement maximal.

La primitive `\number` est développable et son 1-développement provoque un travail assez conséquent : lecture du nombre selon la règle vue ci-dessus et conversion de ce nombre en base 10 sous la forme la plus simple en tokens de catcode 12. Par « forme la plus simple », on entend que le signe du nombre n'est donné que s'il est négatif et que les 0 inutiles de gauche sont supprimés si le nombre était explicitement écrit.

Lorsque `\the` est immédiatement suivi d'un compteur, d'un registre de dimension ou d'une séquence de contrôle définie avec `\chardef` ou `\mathchardef`, elle se comporte comme `\number`.

Les compteurs, les registres de dimensions et les séquences de contrôle définies avec `\chardef` ou `\mathchardef` sont des représentations *internes* d'entiers qui nécessitent une primitive développable (`\the` ou `\number`) pour obtenir les chiffres formant ce nombre en base 10. C'est donc une erreur que de les en dépourvoir pour les afficher.

4.6.4. Nombres romains

Tout comme `\number` et `\the` sont capable de produire des nombres arabes, la primitive `\romannumeral` produit des nombres romains.

50 - Règle

La primitive `\romannumeral` doit être suivie d'un ⟨*nombre*⟩ valide.

Son 1-développement est

– l'écriture en chiffres romains de catcode 12 si le ⟨*nombre*⟩ est strictement positif ;
– vide si le nombre est négatif ou nul.

Si le ⟨*nombre*⟩ est supérieur à 1000, la lettre « m » de catcode 12 sera produite autant de fois qu'il y a de milliers dans le nombre.

Nous faisons fonctionner ici la primitive \romannumeral en l'encadrant des caractères « " » pour mettre en évidence que les espaces qui suivent les nombres disparaissent bien. Vérifions également que le développement maximal a bien lieu avec les macros \foo et \bar :

Code n° II-98

```
a) "\romannumeral 27 "\quad
b) "\romannumeral 4687 "\quad
c) "\romannumeral 0 "\quad% 0 donc développement vide
d) "\romannumeral -2014 "\quad% négatif donc développement vide
e) \def\foo{7}\def\bar{\foo}%
   "\romannumeral \foo\foo\bar"\quad% 777
f) "\romannumeral 1\bar8\foo"\quad% 1787
g) "\romannumeral 1\bar8 \foo"% 178 (stoppé par l'espace) puis 7
```

a) "xxvii" b) "mmmmdclxxxvii" c) "" d) "" e) "dcclxxvii" f) "mdcclxxxvii" g) "clxxviii7"

Remarquons la différence entre les cas f et g où le simple espace en plus après « 8 » au cas g stoppe la lecture du nombre et le développement maximal. On constate que cet espace est absorbé, car aucun espace n'existe à l'affichage entre « clxxviii » et « 7 ».

4.6.5. Le développement de \romannumeral

La primitive \romannumeral est très intéressante, car son 1-développement déclenche un développement maximal dont on peut contrôler la portée. Surtout, ce développement maximal peut ne laisser aucune trace si le nombre que lit finalement \romannumeral est négatif ou nul.

Si dans ce schéma

1. la zone grise ne contient que des tokens purement développables dont le développement est vide ;
2. x est un nombre négatif ou nul

alors, 1-développer \romannumeral revient à développer au maximum tous les tokens se trouvant dans la zone grise et à supprimer le nombre x et l'espace qui le suit.

La zone encadrée devient une zone de développement maximal dont la portée est contrôlée par le placement du nombre négatif ou nul x.

Certes, cela implique une contrainte forte : tous les tokens dans la zone grise doivent être purement développables et une fois ceci fait, ils ne doivent rien laisser. En pratique, cette zone grise contient des appels à des macros et des tests.

Pour mettre en évidence l'utilité d'une telle structure, reprenons l'exercice de la page 113 où l'on a vu que \expsecond{⟨*arg1*⟩}{⟨*arg2*⟩} devait se développer 3

fois pour que le code effectivement dans la pile soit ⟨*arg1*⟩{*arg2} (l'étoile indique que le premier token de l'argument 2 est 1-développé). Nous avions calculé qu'il fallait placer 7 \expandafter devant \> pour parvenir à nos fins :

Code n° II-99

```
\long\def\>#1<{\detokenize{#1}}
\def\swaparg#1#2{#2{#1}}
\def\expsecond#1#2{\expandafter\swaparg\expandafter{#2}{#1}}
\def\X{Bonjour}
\expandafter\expandafter\expandafter\expandafter
\expandafter\expandafter\expandafter
\>\expsecond\X\X<
```

\X {Bonjour}

Avec \romannumeral, un seul \expandafter sera requis pour sauter \> et provoquer le 1-développement de \romannumeral qui va tout développer au maximum et s'arrêter au bon moment. Ce *bon moment* sera juste avant le \X observé à l'affichage. Comme ce \X est l'argument #2 de \swaparg, un simple 0␣, placé avant cet argument #2 stoppe le développement maximal initié par \romannumeral :

Code n° II-100

```
\long\def\>#1<{\detokenize{#1}}
\def\swaparg#1#2{0 #2{#1}}% le "0 " stoppe ici le développement maximal
\def\expsecond#1#2{\expandafter\swaparg\expandafter{#2}{#1}}
\def\X{Bonjour}
\expandafter\>\romannumeral\expsecond\X\X<
```

\X {Bonjour}

■ **Exercice 51**

À quel autre endroit aurait-on pu placer « 0␣ » pour que le développement se passe comme on le souhaite ?

□ **Solution**

On veut stopper la lecture du nombre juste avant l'argument #2 de la commande \swaparg. Cet argument est l'argument #1 de la commande \expsecond et il suffit donc de mettre « 0␣ » au début du premier argument de \expsecond lorsqu'elle est appelée.

Code n° II-101

```
\long\def\>#1<{\detokenize{#1}}
\def\swaparg#1#2{#2{#1}}
\def\expsecond#1#2{\expandafter\swaparg\expandafter{#2}{#1}}
\def\X{Bonjour}
\expandafter\>\romannumeral\expsecond{0 \X}\X<
```

\X {Bonjour}

Contrairement à celle de l'exemple précédent, cette méthode présente l'avantage de ne pas avoir à modifier les macros \swaparg et \expsecond.

On aurait donc remplacer 0␣ par \z@ avec l'avantage pour ce dernier de stopper la lecture du nombre sans avoir à insérer un espace :

Code n° II-102

```
\long\def\>#1<{\detokenize{#1}}
\def\X{Bonjour}
\catcode`\@=11
\expandafter\>\romannumeral\expsecond{\z@\X}{\X}<
\catcode`\@=12
```

\X {Bonjour}

■

4.7. Programmation d'une variable unidimensionnelle

Finissons ce chapitre en programmant une « variable », constituée de cellules qui auront chacune un nom et qui contiendront un code choisi par l'utilisateur. Il s'agit ici de copier les variables de type tableau (ou array) à une dimension qu'offrent la plupart des langages de programmation.

Supposons que l'on veuille définir la variable unidimensionnelle `\foobar`. Pour définir la variable et remplir les cases dont les noms sont « `0` », « `1` », « `3` » et « `toto` », on doit pouvoir écrire :

```
\newunivar\foobar
\defunivar\foobar[0]{abcd}
\defunivar\foobar[1]{1 23}
\defunivar\foobar[3]{XY Z}
\defunivar\foobar[toto]{Bonjour}
```

Et par la suite, il suffira que l'on écrive `\foobar[0]` pour que le code « abcd », contenu dans la cellule « `0` » soit restitué et exécuté.

Passons maintenant à la mécanique interne. Si la variable est « `\foobar` », décidons que le contenu des cellules sera stocké dans des séquences de contrôle auxiliaires. Par exemple, le code de la cellule `0` sera contenu dans la macro `\\foobar[0]`. Autrement dit, le *nom* de la macro doit être constitué des 10 caractères (inoffensifs) `\`, `f`, `o`, `o`, `b`, `a`, `r`, `[`, `0` et `]`. Pour former un tel nom, nous ferons appel à la paire `\csname ...\endcsname`.

L'ordre « `\newunivar\foobar` » doit donc définir la macro `\foobar` de cette façon :

```
\def\foobar[#1]{\csname\string\foobar[#1]\endcsname}
```

Si on place ce texte de remplacement dans la définition de `\newunivar`, il suffit de remplacer `\foobar` par l'argument `#1` de `\newunivar` et de doubler les « # ». La macro `\newunivar` sera donc définie ainsi :

```
\def\newunivar#1{\def#1[##1]{\csname\string#1[##1]\endcsname}}
```

On pourra ainsi appeler par la suite `\foobar[0]` qui se 2-développera en la macro `\\foobar[0]` tout en offrant la protection, au cas où la cellule n'a pas été définie, que la paire `\csname...\endcsname` génère l'inoffensif `\relax`. Aucune erreur de compilation ne sera donc provoquée en cas d'appel à une cellule non définie.

Pour la macro \defunivar\⟨*macro*⟩[⟨*nom*⟩], un \expandafter forcera la formation de la séquence de contrôle avant que \def, qui procède l'assignation, ne voie cette séquence de contrôle. Par conséquent, \defunivar\foo[0] doit avoir le texte de remplacement suivant :

```
\expandafter\def\csname\string\foo[0]\endcsname
```

Et donc, \defunivar doit être définie comme suit :

```
\def\defunivar#1[#2]{\expandafter\def\csname\string#1[#2]\endcsname}
```

ou encore, en utilisant \defname

```
\def\defunivar#1[#2]{\defname{\string#1[#2]}}
```

Il est inutile de lire l'argument du \def qui est le contenu de la cellule. En effet, comme on l'a déjà vu à la page 78, lire cet argument serait redondant puisqu'il faudrait le placer entre accolades à la fin du texte de remplacement de \defunivar. Voici donc comment procéder :

Code n° II-103

```
\def\newunivar#1{\def#1[##1]{\csname\string#1[##1]\endcsname}}
\def\defunivar#1[#2]{\defname{\string#1[#2]}}
\newunivar\foobar
\defunivar\foobar[0]{abcd}
\defunivar\foobar[1]{1 23}
\defunivar\foobar[3]{XY Z}
Cellule 0 : \foobar[0]\par
Cellule 1 : \foobar[1]\par
Cellule 2 : \foobar[2]\par% cellule non définie : \foobar[2] donne \relax
Cellule 3 : \foobar[3]\bigbreak

\newunivar\client
\defunivar\client[nom]{M. Raymond {\sc  Tartempion}}
\defunivar\client[adr]{5 rue de la paix}
\defunivar\client[cod_post]{75000}
\defunivar\client[ville]{Paris}
% fin des définitions, affichage de l'adresse :
\client[nom]\par
\client[adr]\par
\client[cod_post] \client[ville]
```

Cellule 0 : abcd
Cellule 1 : 1 23
Cellule 2 :
Cellule 3 : XY Z

M. Raymond Tartempion
5 rue de la paix
75000 Paris

On remarque que la programmation de cette variable unidimensionnelle est ici, grâce à la richesse et la puissance des instructions de TEX, relativement facile et extrêmement concise puisqu'elle tient en deux lignes !

■ **Exercice 52**

Comment modifier les macros \newunivar et \defunivar vues précédemment pour que les cellules ne soient pas stockées dans la macro \\foobar[0] mais dans la macro \foobar@0 ?

□ **Solution**

Faisons la supposition fondée que \escapechar définit « \ » comme caractère d'échappement. Lorsque \string est développée à l'intérieur de \csname...\endcsname, il convertit le token \foobar en 7 tokens de catcode 12 qui sont \ f o o b a r. Il suffit de manger le premier token \ avec un \gobone en ayant pris soin de 1-développer \string avant que \gobone n'entre en action :

```
\def\newunivar#1{%
    \def#1[##1]{\csname\expandafter\gobone\string#1@##1\endcsname}}
\def\defunivar#1[#2]{%
    \defname{\expandafter\gobone\string#1@#2}}
```

■

★
★ ★

Voici la fin de cette longue et difficile partie où, en nous intéressant aux commandes, nous avons découvert comment TEX fonctionne. Pour résumer à l'extrême, TEX est un langage de macros et donc, se comporte comme une machine qui remplace du code par du code jusqu'à arriver à des tokens non développables qui seront « exécutés » (des primitives, des caractères ordinaires de catcode 10, 11 ou 12 qui seront affichés ou des caractères revêtant des propriétés liées à leur catcode, comme $).

Si l'on entre un peu plus dans les détails et en résumant ce qui a été dit, voici comment on peut se représenter le fonctionnement de TEX : comme nous l'avons vu à la première partie, le code TEX est constitué de cases chacune formée d'un octet pour les moteurs 8 bits ou par une séquence d'octets codant un caractère UTF8 pour les moteurs UTF8. Lorsqu'ils sont transformés en tokens lorsque TEX les lit, chacun de ces caractères obéit à des règles particulières propres à leurs catégories. Voici les cas particuliers qui sont transformés lors de la lecture :

- lorsque des lettres sont précédées du caractère d'échappement « \ » de catcode 0, le tout ne forme qu'un seul token, une séquence de contrôle ;
- lorsqu'un espace (catcode 10) dans le code est suivi d'autres espaces, les autres espaces sont ignorés ;
- un seul retour charriot ^^M (catcode 5) est compris comme étant un espace ;
- deux retours charriots consécutifs sont équivalents au token \par ;
- deux caractères identiques de catcode 7 consécutifs (comme « ^^ ») sont convertis en un caractère selon les règles exposées à la 1re partie ;
- le caractère « % » (catcode 14) ainsi que tous les caractères jusqu'au retour charriot suivant sont ignorés.

Ainsi, on peut considérer le code TEX comme étant des caractères écrits les uns à la suite des autres comme s'ils étaient sur un ruban et, après leur lecture, ces caractères sont transformés en de la matière assimilable par TEX, les tokens. TEX fonctionne à la manière d'une tête de lecture qui se déplace toujours linéairement de gauche à droite sur le ruban où est écrit le code. Voici, comment on peut se représenter schématiquement le fonctionnement de TEX :

1. lire un token : utiliser en priorité les tokens stockés dans la pile d'entrée si elle n'est pas vide et dans le cas contraire, construire le token à partir des caractères du code source en appliquant les règles de lecture du code en vigueur à ce moment-là.

Si ce token lu requiert la lecture d'autres tokens (lecture d'un nombre, cas d'une macro à plusieurs arguments, cas de `\def` qui doit être suivi d'une macro, d'un texte de paramètre et d'un texte de remplacement, etc.), recommencer en 1 jusqu'à ce que les tokens requis soient lus ;

2. figer les catcodes de tous les tokens lus et les stocker en mémoire ;
    a) si ce qui vient d'être lu n'est pas développable, exécuter l'action que commande le tout ;
    b) sinon, 1-développer le premier token et insérer devant la tête de lecture (dans la pile d'entrée) ce qui en résulte ;
3. retourner en 1.

Lorsque TEX est en phase de développement et si `\expandfter` est la primitive à développer, convenons que 0 est la position de cet `\expandafter` dans la liste des tokens à lire. Lorsque `\expandafter` est développé, la tête de lecture de TEX a la capacité, sans bouger de place, de « détacher » une « tête de développement ». Celle-ci, en se désolidarisant de la tête de lecture, ira 1-développer le token se trouvant plus loin dans le code à la position 2 et suivra éventuellement les ordres de développement que son 1-développement implique. Une fois qu'il n'y a plus d'ordre de développement à suivre, la tête de développement revient sur la tête de lecture qui reprend son travail où elle l'avait laissé et tiendra compte, dans le futur, des développements effectués par sa tête de développement.

On peut donc énoncer que le fonctionnement de TEX s'apparente à celui d'une « machine de Turing ».

Troisième partie

Structures de contrôle et récursivité

Sommaire

Après avoir abordé la plus grosse partie des spécificités de TEX, nous pouvons entrer dans le vif du sujet avec de la programmation non linéaire. Aucun algorithme complexe ne sera abordé dans cette partie, bien au contraire, nous nous en tiendrons aux plus élémentaires et aux plus passepartouts. Malgré leur simplicité, ils sont néanmoins difficiles à cause des spécificités du langage TEX.

Chapitre 1

Les outils de programmation de TEX

Le langage TEX a été doté des structures de contrôles nécessaires pour élaborer n'importe quel algorithme, aussi compliqué soit-il. À ce titre, c'est donc un langage *complet*. Bien sûr, il ne viendrait à l'idée de personne de coder en TEX un décompresseur zip ou un algorithme de jeu d'échecs [1] ! Dans le monde de TEX, le mot « compilation » signifie « traduction de la totalité code source en document affichable ». En TEX, lorsqu'on parle de programmation, ce mot devient ambigu, car dans le monde de la programmation, « compilation » signifie « transformation du code source en code exécutable », le fichier binaire exécutable obtenu étant d'une grande rapidité. Ce qu'il faut comprendre est que le langage TEX est *interprété*, c'est-à-dire « exécuté au fur et à mesure » lors de la compilation ce qui le rend lent !

Les principales structures de contrôle dont TEX est doté sont les suivantes :

– l'assignation, c'est-à-dire la possibilité de stocker des informations dans des « registres » (le mot « variables » est plus couramment utilisé dans d'autres langages), qui sont en quelque sorte des endroits de la mémoire accessibles et lisibles facilement. En TEX, les structures les plus couramment utilisées capables de stocker de l'information sont les suivants :

1. Il faut tout de même signaler une récente extension appelée « reverxii » pour TEX qui joue au jeu d'Othello et qui tient en moins de 1000 octets ! Cette très faible taille tient au fait que l'auteur a utilisé toutes les astuces TEXiennes (programmation de caractères actifs notamment) pour que la taille soit la plus petite possible, au prix d'une lisibilité nulle à un tel point qu'il est presque impossible de dire s'il s'agit de code TEX ou pas !

Le programme comporte une intelligence artificielle qui, bien que loin d'être redoutable, peut tenir tête à un joueur débutant.

  - les séquences de contrôle ;
  - les registres de tokens ;
  - les registres d'entiers ou « compteurs » qui accueillent les entiers et sont dotés des 4 opérations où la division n'est possible que par un entier, étant entendu que cette division donne la partie entière ;
  - les registres de dimensions stockent des longueurs et qui permettent, sous certaines réserves que nous verrons, de manipuler des nombres décimaux. Les 4 opérations sont disponibles, mais la division n'est possible que par un *entier* ;
  - les registres de boites qui stockent un contenu arbitraire sous forme affichable, mais il n'est plus possible d'accéder au code qui a généré ce contenu.
- les tests ;
- la récursivité, c'est-à-dire la possibilité pour une macro de s'appeler elle-même, que ce soit directement ou par l'intermédiaire d'une ou plusieurs autres macros ;
- la localité, c'est-à-dire la possibilité de définir une zone où les assignations resteront valables dans cette zone pour être détruites une fois sorti de cette zone, tout en permettant de procéder à des assignations *globales* qui survivent une fois que la zone est fermée ;
- la possibilité d'écrire dans un fichier, c'est-à-dire d'y stocker des *caractères* ;
- la possibilité de lire les caractères stockés dans un fichier.

Et c'est à peu près tout ! Cette liste est *minimaliste* et n'a rien à voir le pléthore de moyens que propose un langage de programmation actuel. Pour faire une comparaison dans les moyens simples et basiques, il n'y a par exemple, aucune structure de boucle (for, loop, while, etc.). Ni de « variables » de type évolué comme les tableaux à plusieurs dimensions par exemple. Aucune trace non plus d'opérations scientifiques. Mais qu'à cela ne tienne, les structures énoncées ci-dessus sont bien suffisantes pour en programmer soi-même. Il faut s'y habituer, en TEX, on ne dispose que du strict minimum et bien souvent, on doit construire soi-même des structures de contrôle plus évoluées comme les boucles. La plupart du temps avec TEX, tout est question d'assignations, de lecture raisonnée d'arguments, de comparaison, de développement et de récursivité. Pour autant que ça soit pénible au début, en contrepartie, c'est extrêmement souple puisqu'on peut bâtir des boucles sur mesure. C'est ce que nous allons apprendre à faire dans cette partie en étudiant plus spécialement les tests et la récursivité.

Mais avant de les aborder, il nous faut revenir aux nombres entiers et comprendre comment TEX calcule.

Chapitre 2

TEX ET LES ENTIERS

S'il y a bien un domaine où TEX n'excelle pas, c'est celui du calcul. On ne pourrait lui en vouloir puisque, conçu pour composer du texte, il n'était pas nécessaire de le doter de puissants outils de calcul. Mais depuis que TEX a été écrit, beaucoup de temps s'est écoulé à l'échelle informatique et désormais, tout langage embarque pléthore d'outils de calcul. Par conséquent, un programmeur habitué à ces outils ressent à un moment ou à un autre une légitime frustration envers TEX qui ne les propose pas. Voyons tout de même ce que TEX offre en matière de calculs arithmétiques.

2.1. L'arithmétique TEXienne

2.1.1. Les entiers

TEX manipule des entiers signés sur 32 bits. Ils sont donc compris entre $-2^{31}+1$ et $2^{31}-1$ c'est-à-dire entre −2 147 483 647 et 2 147 483 647. Ces entiers sont stockés dans des registres spéciaux appelés « compteurs ». 256 de ces registres sont disponibles avec TEX et 65 536 avec εTEX. Ainsi, `\count42` fait référence au compteur n° 42. Contrairement aux registres de tokens avec `\toks0`, aucune convention n'existe quant à un compteur portant un numéro spécial qui serait non utilisé et qui servirait de « compteur brouillon ».

TEX met aussi à disposition la primitive `\countdef`, qui ressemble à `\chardef` ou `\toksdef`. Par exemple, écrire

```
\countdef\foo=42
```

rend `\foo` équivalent à `\count42` de telle sorte que le registre n° 42 peut être désigné

par la séquence de contrôle \foo. L'équivalence dont il est question ici ressemble à celle de \let en ce sens que \foo n'est pas développable et n'a pas de texte de remplacement mais *sera* \count42.

Comme pour les registres de tokens avec \toksdef, plain-TEX fournit une macro \newcount qui, suivie d'une \⟨*macro*⟩, permet d'allouer un compteur pour y faire ensuite référence avec une macro plutôt que par un numéro. La macro \newcount se charge de trouver le numéro du prochain registre de compteur libre. Si par exemple, on écrit \newcount\foo et si 89 est le numéro du prochain registre libre, alors la macro \newcount procèderait à l'assignation suivante :

```
\global\countdef\foo=89
```

La primitive \global, placée devant l'assignation, rend celle-ci globale : on ne peut donc pas allouer localement un registre de compteur qui serait ensuite libéré après la sortie d'un groupe [1].

51 - Règle

Un compteur a vocation à contenir un nombre entier de $-2^{31}+1$ à $2^{31}-1$. On y fait référence par

\count⟨*nombre*⟩

Il est aussi possible d'utiliser la commande

\newcount\⟨*macro*⟩

qui fait le nécessaire pour lier la \⟨*macro*⟩ au prochain numéro de compteur non utilisé. Par la suite, la \⟨*macro*⟩ se comporte exactement comme \count⟨*nombre*⟩. Comme les registres de tokens, les ⟨*compteurs*⟩ peuvent donc indifféremment être désignés selon ces deux possibilités.

Pour assigner un ⟨*entier*⟩ (voir la définition d'un entier à la page 125) à un compteur, il faut écrire

⟨*compteur*⟩= ⟨*entier*⟩

où le signe = ainsi que l'espace qui le suit sont facultatifs.

Rappelons que pour *afficher* la valeur du compteur, nous ne pouvons pas écrire ⟨*compteur*⟩ dans le code sans risquer de provoquer une erreur de compilation. En effet, TEX attendrait une assignation qui commence de cette façon et si ce ⟨*compteur*⟩ n'était pas suivi d'un ⟨*nombre*⟩, TEX se plaindrait d'un « `Missing number, treated as zero` ». Un compteur étant une représentation interne d'un entier, pour *afficher* sa valeur, il faut traduire cette représentation interne en caractères affichables, et faire précéder le ⟨*compteur*⟩ de \the, \number ou \romannumeral :

Code n° III-104

```
\newcount\foo \newcount\bar
\foo=42 \bar=\foo \string\bar\ vaut : \the\bar \par
```

1. Bien évidemment, mobiliser un compteur localement est possible, mais il faudrait écrire une nouvelle macro le permettant. Ceci est fait par le package `etex.sty` pour LATEX qui, entre autres fonctionnalités, fournit une macro \loccount\⟨*macro*⟩ par laquelle le compteur est défini de façon *locale*.

```
3 \bar=\foo57 \string\bar\ vaut : \number\bar\par
4 \bar=2014 \string\bar\ vaut : \romannumeral\bar
```

```
\bar vaut : 42
57 \bar vaut : 42
\bar vaut : mmxiv
```

La ligne n° 3 montre que les caractères « 57 » ne sont pas pris en compte pour l'affectation. Comme cela est dit à la page 125, si le ⟨*nombre*⟩ que TEX lit est un registre de compteur, le nombre qu'il contient est lu, et la lecture du nombre s'arrête (laissant ici « 57 » hors du nombre pris en compte).

■ **Exercice 53**

Que fait le code suivant ?

```
\newcount\foo
\newcount\bar
\foo=57
\bar=9\foo8\relax
```

□ **Solution**

Tout d'abord, deux compteurs sont créés. Le nombre 57 est affecté au compteur `\foo`. Puis, à la dernière ligne, à droite de `\bar=`, l'entier lu est 9 car la lecture du nombre s'arrête sur `\foo` qui n'est pas développable et n'est pas un chiffre.

Le compteur `\bar` vaut donc 9 et ce qui est à la suite « `\foo8\relax` » est une affectation qui rendrait le compteur `\foo` égal à 8, écrasant la valeur 57 qu'il avait auparavant. Le code proposé fait 3 assignations et ne produit aucun affichage.

À la dernière ligne, si nous avions voulu assigner 9578 au compteur `\bar` (où 57 est la valeur du compteur `\foo`), il aurait fallu écrire :

```
\bar=9\number\foo8\relax
```

de telle sorte que le développement mis en place pour la lecture du nombre convertisse `\number\foo` en caractères valides pour un ⟨*nombre*⟩. ■

2.1.2. Les opérations

Venons-en aux opérations arithmétiques à notre disposition. Celles-ci opèrent sur des compteurs, c'est-à-dire qu'il n'est pas possible de faire des opérations sur des nombres explicitement écrits en chiffres.

52 - Règle

Dans les lignes ci-dessous, ⟨*n*⟩ est un nombre entier au sens de TEX :

- `\advance`⟨*compteur*⟩ `by` ⟨*n*⟩ : ajoute ⟨*n*⟩ au ⟨*compteur*⟩ ;
- `\multiply`⟨*compteur*⟩ `by` ⟨*n*⟩ : multiplie le ⟨*compteur*⟩ par ⟨*n*⟩ ;
- `\divide`⟨*compteur*⟩ `by` ⟨*n*⟩ : divise le ⟨*compteur*⟩ par ⟨*n*⟩ en lui assignant la troncature à l'unité si le résultat de la division n'est pas entier.

Dans tous les cas, le mot « by » et l'espace qui le suit sont facultatifs.

Avec ces trois opérations, il est possible de programmer une macro \fonction dont l'argument est un entier n et qui calcule et affiche la valeur de $(3n-4)^2$. Le principe est d'assigner l'entier n à un compteur et effectuer les opérations sur ce compteur (le multiplier par 3, lui ajouter −4, puis le multiplier par lui-même) pour l'afficher ensuite :

Code n° III-105

```
\newcount\foo
\def\fonction#1{%
  \foo=#1 % assigne l'entier #1 au compteur puis
  \multiply\foo3 % multiplie par 3
  \advance\foo-4 % soustrait 4
  \multiply\foo\foo% élève au carré (lmultiplie \foo par lui même)
  \number\foo% enfin, afficher le résultat
}
a) \fonction5\qquad b) \fonction{-13}
```

a) 121 b) 1849

La macro donne bien les résultats attendus, mais l'inconvénient est qu'elle nécessite un compteur et surtout qu'elle n'est pas purement développable. En effet, si nous écrivons \edef\bar{\fonction5}, la macro \bar n'aura pas 121 comme texte de remplacement. La raison est que le texte de remplacement de \fonction contient des primitives qui ne se développent pas (\multiply, \advance) et la séquence de contrôle \foo qui n'est pas développable non plus. Il s'agit ici d'un cas qui illustre la fondamentale différence entre « affichage produit par une macro après exécution » et « texte de remplacement de cette macro ». Ce n'est pas parce que « \fonction5 » affiche 121 que tout se passe comme si « \fonction5 » était équivalent (même après développement) aux caractères 121. Nous pouvons le constater sur cet exemple :

Code n° III-106

```
\newcount\foo
\def\fonction#1{%
  \foo=#1 \multiply\foo3 \advance\foo-4 \multiply\foo\foo
  \number\foo}
\edef\bar{\fonction{5}}%
a) Signification : \meaning\bar\par
b) Exécution : \bar
```

a) Signification : macro:->\foo =5 \multiply \foo 3 \advance \foo -4 \multiply \foo \foo 0
b) Exécution : 0

Pourquoi obtenons-nous 0 lorsque \bar est exécutée ? Parce que le code contenu dans \bar (et affiché au cas a) effectue dans un premier temps les calculs sur le compteur \foo, mais au lieu d'afficher le ce que contient ce compteur, elle affiche 0, qui est le résidu du développement de \number\foo.

Non contente de créer une macro non purement développable, la méthode précédente peut rapidement rendre l'enchainement des opérations lourd, fastidieux et peu lisible, pour peu que les expressions à calculer soient plus complexes. En l'état de TEX tel qu'il a été écrit par D. KNUTH, il n'est pas possible de s'y prendre autrement. Heureusement, avec le moteur ε-TEX , une autre alternative est possible.

2.1.3. La primitive \numexpr

Pour les entiers, la primitive \numexpr permet d'effectuer un enchainement d'opérations arithmétiques via une écriture mathématique en notation infixée où les priorités arithmétiques habituelles sont respectées. Dans ces expressions, les espaces sont ignorés, les signes d'opérations sont +, –, * et / et des entiers explicitement écrits en chiffres peuvent y figurer. Pour la division, la seule différence avec la division TEXienne réside dans le fait qu'elle donne *l'entier le plus proche* au lieu de *la troncature*. Les expressions aussi contenir des parenthèses.

53 - Règle

La primitive \numexpr permet de calculer des enchainements d'opérations sur les entiers, pris au sens de la définition de la page 125. Elle doit être suivie d'une expression arithmétique où les espaces sont ignorés, les signes opératoires sont +, –, *, / et où les parenthèses sont admises .

Elle cherche à évaluer ce qui la suit en amorçant un développement maximal jusqu'à rencontrer un token qui ne peut figurer dans une expression arithmétique. Si l'expression arithmétique est suivie d'un \relax, alors ce \relax est mangé par \numexpr. Ceci constitue une spécificité de la primitive \numexpr puisqu'avec TEX, seuls les *espaces* sont mangés après les nombres. C'est donc une excellente habitude de faire suivre une expression arithmétique évaluée par \numexpr d'un \relax.

La structure

\numexpr⟨*expression arithmétique*⟩\relax>

est un entier au sens de TEX, mais un peu comme l'est un compteur, c'est une représentation *interne* d'un entier et donc, pour l'afficher, il faut recourir à une primitive qui transforme la représentation interne en caractères affichables telle que \the, \number ou \romannumeral.

L'exemple suivant montre comment, avec \number, on peut afficher une expression évaluée par \numexpr :

Code nº III-107

```
a) \number\numexpr2+3*4\relax\qquad
b) \romannumeral\numexpr2*3\relax\qquad
c) \number\numexpr(6-2*4)*(1-(2-6))\relax\qquad
d) \number\numexpr7/4\relax\qquad %7/4 vaut 1,75
e) \edef\foo{\number\numexpr2+3*4\relax}\meaning\foo
```

a) 14 b) vi c) -10 d) 2 e) macro:->14

Une expression évaluée par \numexpr peut contenir une sous-expression, elle-même évaluée par un autre \numexpr. Voici comment calculer 3*(1+2*5)+2, en évaluant l'expression entre parenthèses avec un \numexpr imbriqué dans l'expression à calculer et en prenant soin de stopper sa portée par un \relax :

Code n° III-108

```
\number\numexpr3*\numexpr1+2*5\relax+2\relax
```

35

L'emploi de \numexpr peut rendre la macro \fonction vue précédemment *purement développable* :

Code n° III-109

```
\def\fonction#1{\number\numexpr(3*#1-4)*(3*#1-4)\relax}
a) \fonction5\qquad
b) \fonction{-13}\qquad
c) \edef\bar{\fonction5}\meaning\bar
```

a) 121 b) 1849 c) macro:->121

■ **Exercice 54**

Dans le code ci-dessus, l'expression « 3*#1-4 » est évaluée *deux* fois, ce qui est inutile et consommateur de temps. Comment modifier le code pour qu'elle ne soit évaluée qu'une seule fois ?

□ **Solution**

L'idée est de provoquer le calcul de « 3*#1-4 » et passer le résultat à une macro auxiliaire \carre qui affichera le carré de son argument :

Code n° III-110

```
\def\fonction#1{\exparg\carre{\number\numexpr3*#1-4\relax}}
\def\carre#1{\number\numexpr#1*#1\relax}
a) \fonction{5}\qquad
b) \fonction{-13}\qquad
c) \edef\bar{\fonction5}\meaning\bar
```

a) 121 b) 1849 c) macro:->121

Ici, la macro \exparg 1-développe et force le calcul de \number\numexpr3*#1-4\relax en un nombre signé explicitement écrit, qui est transmis à la macro \carre qui se charge de le multiplier par lui-même. ■

Il est parfois gênant que la primitive de TEX \divide sur les compteurs et la division « / » de \numexpr ne donnent pas les mêmes résultats. En effet, cela rompt la cohérence qui existe entre les trois autres opérations. Comment réconcilier les deux et faire en sorte d'inventer une division pour ε-TEX qui donne aussi la troncature ? Pour cela, il existe une formule générale. Si x et y sont deux entiers *positifs*, si « / » représente la division donnant l'entier le plus proche (la division de \numexpr donc), la troncature du quotient de x par y est

$$(x - (y-1)/2)/y$$

À l'aide de \numexpr, il est facile de programmer une macro \truncdiv{x}{y} qui calcule la troncature du quotient de x par y. Le résultat est donné sous la forme d'un entier *non explicite*, c'est-à-dire qu'il faut \number pour transformer cet entier en nombre affichable :

Code n° III-111

```
\def\truncdiv#1#2{\numexpr(#1-(#2-1)/2)/#2\relax}
8/3 :
a) \number\truncdiv83\qquad      % doit donner 2
b) \number\truncdiv{-8}3\qquad   % doit donner -2
c) \number\truncdiv8{-3}\qquad   % doit donner -2
d) \number\truncdiv{-8}{-3}\par  % doit donner 2
4/3 :
e) \number\truncdiv43\qquad      % doit donner 1
f) \number\truncdiv{-4}3\qquad   % doit donner -1
g) \number\truncdiv4{-3}\qquad   % doit donner -1
h) \number\truncdiv{-4}{-3}      % doit donner 1
```

8/3 : a) 2 b) -3 c) -3 d) 2
4/3 : e) 1 f) -2 g) -2 h) 1

La macro donne bien des résultats erronés lorsque les nombres sont de signes contraires. Nous verrons au prochain chapitre comment la modifier pour que le résultat soit correct dans tous les cas.

Bien entendu, `\truncdiv` peut être écrite dans une expression numérique évaluée par `\numexpr` puisque les `\numexpr` peuvent être imbriqués sous réserve que les `\numexpr` internes aient leur portée limitée par `\relax`, ce qui est le cas ici.

2.2. Le test `\ifnum`

Le test `\ifnum` est le premier test abordé puisqu'il a un étroit rapport avec le chapitre en cours sur les nombres entiers. Il est aussi l'un des plus fréquents. Il n'est cependant qu'un test parmi beaucoup d'autres et avant de s'intéresser plus en détail à `\ifnum`, un peu de théorie sur les tests de TEX doit être abordée.

2.2.1. Structure des tests

54 - Règle

Les tests de TEX sont exécutés par des primitives qui obéissent aux contraintes suivantes :

1. le nom des primitives exécutant un test commence par les lettres « `if` » suivies d'autres lettres notées ⟨*test*⟩ déterminant de quel test il s'agit ;
2. si `\if`⟨*test*⟩ est une primitive exécutant un test sur ses ⟨*arguments*⟩, sa structure est la suivante :

```
\if⟨test⟩⟨arguments⟩
    ⟨code exécuté si le test est vrai⟩
\else
    ⟨code exécuté si le test est faux⟩
\fi
```

   La branche entre `\else` et `\fi` est facultative et donc, un test peut également avoir la structure suivante :

```
\if⟨test⟩⟨arguments⟩
    ⟨code exécuté si le test est vrai⟩
\fi
```

Au lieu de \else et \fi, on peut mettre toute séquence de contrôle rendue \let-égale à \else ou \fi.
Les tests et les primitives \else et \fi sont développables.

Les codes exécutés selon l'issue du test peuvent aussi contenir des tests sans que TEX ne s'y perde dans les appariements entre les primitives \if⟨*test*⟩ et leur \else ou \fi. En effet, un peu comme le compteur d'accolades, TEX est doté d'un compteur interne d'imbrication de tests qui assure que chaque primitive de test est correctement appariée avec le bon \else ou \fi.

Il est en outre primordial de comprendre qu'à l'occasion d'un test, la tête de lecture de TEX *ne lit pas* tout le code dépendant du test (jusqu'au \fi) pour stocker en mémoire les codes à exécuter selon l'issue du test afin de mettre l'un ou l'autre sur la pile juste après avoir effectué le test. Non, après avoir exécuté le test elle continue sa lecture linéaire, mais elle *sait* qu'elle se trouve dans la portée d'un test et qu'à un moment ou à un autre, elle aura à lire \fi, éventuellement précédé d'un \else.

Pour que ce point soit clair, mettons-nous à la place de la tête de lecture de TEX qui doit exécuter le test suivant (cas n° 1) :

\if⟨*test*⟩⟨*arguments*⟩⟨*code vrai*⟩\else ⟨*code faux*⟩\fi

ou (cas n° 2) :

\if⟨*test*⟩⟨*arguments*⟩⟨*code vrai*⟩\fi

Test vrai

Supposons pour l'instant que le test est vrai. Voici l'enchainement des opérations qui vont avoir lieu :

1. tout va commencer par le 1-développement de « \if⟨*test*⟩⟨*arguments*⟩ » qui va disparaitre à l'occasion de ce développement ;
2. ce qui est à suivre, le ⟨*code vrai*⟩ est exécuté normalement ;
3. le 1-développement de qui suit est vide. Il s'agit de

\else ⟨*code faux*⟩\fi (cas n° 1) ou \fi (cas n° 2)

Test faux

Supposons à présent que le test est faux. Voici ce qui va se passer :

– cas n° 1 :
    1. le 1-développement de « \if⟨*test*⟩⟨*arguments*⟩⟨*code vrai*⟩\else » est vide (où le \else est apparié avec le test initial) ;
    2. ce qui suit, c'est-à-dire ⟨*code faux*⟩ est exécuté normalement ;
    3. le \fi final qui reste est 1-développé et disparait.
– cas n° 2 : le 1-développement de « \if⟨*test*⟩⟨*arguments*⟩⟨*code vrai*⟩\fi » est vide et la totalité de ce code disparait donc en 1-développement.

55 - Règle

Lorsqu'un test est exécuté par une primitive de test \if⟨*test*⟩, les primitives \else et \fi ne sont *pas* supprimées lorsque le test est fait. Elles restent en place et lorsqu'elles seront rencontrées plus tard, elles s'autodétruiront par un simple développement.

2.2.2. Comparer des entiers

Le test \ifnum

56 - Règle

La primitive \ifnum effectue une comparaison entre deux entiers. La syntaxe est de la forme

```
\ifnum⟨entier1⟩⟨signe⟩⟨entier2⟩
    ⟨code exécuté si le test est vrai⟩
\else
    ⟨code exécuté si le test est faux⟩
\fi
```

où le ⟨*signe*⟩, de catcode 12, est soit « = » si l'on veut tester l'égalité entre les deux entiers, soit « < » ou « > » s'il l'on cherche à tester une inégalité stricte.

Code n° III-112

```
a) \ifnum 15=14 vrai\else faux\fi\qquad% test faux
b) \ifnum4<200 vrai\else faux\fi\qquad% test vrai
c) \newcount\foo \foo=9
   \ifnum\foo>9 nombre\else chiffre\fi% test faux
```

a) faux b) vrai c) chiffre

Vérifions maintenant ce qui a été exposé en théorie sur le 1-développement d'un test. Pour ce faire, mettons à contribution la macro \>...< vue à la partie précédente :

Code n° III-113

```
\long\def\>#1<{\detokenize{#1}}
\expandafter\>\ifnum5=5 vrai\else faux\fi<\par% test vrai
\expandafter\>\ifnum5=6 vrai\else faux\fi<% test faux
\fi\fi% rétablir l'équilibre \ifnum et \fi
```

vrai\else faux\fi
faux\fi

Les trois premières lignes de ce code permettent de visualiser les choses, mais rompent l'équilibre entre les \ifnum et les \fi. En effet, les \ifnum sont exécutés par \expandafter, mais les \fi, détokénisés par \>, ne seront jamais exécutés par TEX. Pour rétablir l'équilibre et éviter que ce bug ne se propage jusqu'à la fin du code source de ce livre, deux \fi ont été rajoutés à la ligne n° 4.

■ Exercice 55

Vérifier que les appariements des tests avec leur `\else` et `\fi` sont corrects en programmant une macro `\numtest` qui admet un argument de type entier et qui affiche « chiffre » si son argument est compris entre −9 et 9 inclus. Dans les autres cas, la macro doit afficher « nombre positif » ou « nombre négatif ».

□ Solution

Voici une façon de programmer la macro `\numtest` qui, compte tenu de ce qui a été dit sur le développement des tests, est purement développable :

Code n° III-114

```
\def\numtest#1{%
  \ifnum#1>-10
    \ifnum#1<10
      chiffre%
    \else
      nombre positif%
    \fi
  \else
    nombre négatif%
  \fi
}
a) \numtest{3}\qquad
b) \numtest{-67}\qquad
c) \numtest{-8}\qquad
d) \numtest{21}\qquad
e) \edef\foo{\numtest{2014}}\meaning\foo
```

a) chiffre b) nombre négatif c) chiffre d) nombre positif e) macro:->nombre positif

■

■ Exercice 56

Écrire une macro `\normalise` qui admet un argument de type entier positif. Si l'entier est compris entre 0 et 999 compris, cette macro devra afficher son argument avec 3 chiffres, quitte à rajouter des 0 inutiles à gauche pour satisfaire cette exigence. Dans les autres cas, aucun affichage ne sera fait.

Par exemple, l'argument « 17 » donnera un affichage de « 017 » et « 8 » sera affiché « 008 ».

□ Solution

Il suffit d'imbriquer des tests comme à l'exemple précédent et afficher 00 ou 0 juste avant l'argument #1 selon le cas. Ici par contre, la branche `\else` n'est pas nécessaire :

Code n° III-115

```
\def\normalise#1{%
  \ifnum#1>-1 % ne faire quelque chose que si #1 est positif ou nul
    \ifnum#1<100 % si <100
      0% afficher un 0
      \ifnum#1<10 % si <10
        0%afficher un autre 0
      \fi
    \fi
    \number#1 %afficher le nombre
  \fi}% affiche le nombre
a) \normalise{749}\qquad
b) \normalise{0017}\qquad
c) \normalise8\qquad
```

```
14 d) \normalise{1789}\qquad
15 e) \normalise{-18}\qquad
16 f) \normalise{0}
```

a) 749 b) 017 c) 008 d) 1789 e) f) 000

Le \number qui se trouve devant l'entier #1 a deux fonctions :

- celle d'afficher #1 sans provoquer d'erreur de compilation si #1 est un compteur ;
- purger les éventuels 0 inutiles qui se trouvent à gauche de #1 si celui-ci est écrit explicitement. Ainsi, écrire \normalise{0017} mène au résultat correct.

Il reste à remarquer que la macro \normalise, puisque son texte de remplacement n'est constitué que de tests et de caractères affichables, est purement développable. ■

Le test \ifcase

Pour seconder le test \ifnum, TEX met à disposition le test \ifcase qui opère également sur des entiers. Ce test fait un peu chambre à part dans le monde des tests, car sa syntaxe échappe à la syntaxe générale des tests et c'est pourquoi elle est expliquée en détail ici.

57 - Règle

Le test \ifcase teste si un entier est successivement égal à 0, 1, 2, ... selon la syntaxe suivante :

```
\or
  \ifcase⟨nombre⟩
      ⟨code exécuté⟩ si ⟨nombre⟩ = 0
  \or
      ⟨code exécuté⟩ si ⟨nombre⟩ = 1
  \or
      ⟨code exécuté⟩ si ⟨nombre⟩ = 2
  etc...
  \else
      ⟨code alternatif⟩% exécuté si aucune égalité précédente n'est vérifiée
  \fi
```

Les entiers auxquels le ⟨*nombre*⟩ est comparé commencent nécessairement à 0, sont consécutifs et vont aussi loin qu'il y a de branches \or. Il est donc nécessaire d'écrire autant de \or que l'on veut envisager de cas.

La branche \else⟨*code alternatif*⟩ est facultative.

Voici comment nous pourrions programmer une macro \mois{⟨*entier*⟩} qui affiche en toutes lettres le nom du mois dont le numéro est passé en argument. Il est même possible de donner le mois en cours comme argument avec la primitive \month, se comportant comme un compteur, qui contient le numéro du mois en cours :

Code n° III-116

```
1 \def\mois#1{%
2   \ifcase#1\relax
```

```
    \char`\#\char`\#% afficher "##" si argument = 0
  \or janvier\or f\'evrier\or mars\or avril%\or
  \or mai\or juin\or juillet\or aout%
  \or septembre\or octobre\or novembre\or d\'ecembre%
  \else
    \char`\#\char`\#% afficher "##" sinon
  \fi
}
a) \mois{-4}\qquad
b) \mois{3}\qquad
c) \mois{11}\qquad
d) \mois{20}\qquad
e) \edef\foo{\mois{\month}}\meaning\foo% mois en cours
```

a) ## b) mars c) novembre d) ## e) macro:->décembre

2.2.3. Programmer un test : syntaxe

Les primitives de TEX ne suffisent pas – et de loin – à couvrir tous les tests qu'il est possible de faire. On est donc parfois amené à programmer des *macros* qui effectuent des tests plus complexes à l'aide notamment d'imbrications de tests primitifs. Décidons dès à présent que les noms de ces macros commencent aussi par les lettres « `if` » suivies d'autres lettres notées ⟨*test*⟩. C'est un choix *arbitraire* et tout autre choix aurait également été légitime : certains font commencer les *macros* de test par les lettres *majuscules* « `IF` » pour ne pas qu'elles soient confondues avec les *primitive*s qui commencent avec des lettres *minuscules*.

La syntaxe de ces macros, différente de celle des primitives de test, fait davantage consensus. Si `\if`⟨*test*⟩ est une macro effectuant un test sur ses ⟨*arguments*⟩ (pris comme argument d'une macro), il est extrêmement courant d'adopter la syntaxe suivante, très facile d'utilisation :

```
\if⟨test⟩⟨arguments⟩
    {⟨code exécuté si le test est vrai⟩}
    {⟨code exécuté si le test est faux⟩}
```

Basons-nous sur cette syntaxe pour programmer une macro `\ifletter*` qui teste si son argument, que l'on suppose constitué d'un seul caractère, noté ⟨*car*⟩, est une lettre *minuscule*. Par souci de concision, notons ⟨*code vrai*⟩ et ⟨*code faux*⟩ les codes exécutés selon les issues des tests :

```
\ifletter{⟨car⟩}
    {⟨code vrai⟩}
    {⟨code faux⟩}
```

Pour effectuer le test avec `\ifnum`, seule primitive de test que nous connaissons pour l'instant, il va falloir exploiter une façon d'écrire un entier qui est `‘`⟨*car*⟩ où ⟨*car*⟩ est un caractère. L'entier que cela représente est le code du caractère ⟨*car*⟩ (voir page 125). Nous allons également tirer parti du fait que les codes des caractères de « a » à « z » sont des entiers consécutifs. Par conséquent, ⟨*car*⟩ est une lettre minuscule si l'entier `‘`⟨*car*⟩ est compris entre les entiers `‘a` et `‘z` (bornes comprises).

2.2.4. Programmer un test : méthodes

Compte tenu du fait qu'il existe plusieurs méthodes pour programmer la *structure* d'une telle macro, nous allons en envisager plusieurs et elles seront numérotées pour pouvoir facilement y faire référence plus tard.

Méthode nº 1 : intuitive

Une façon intuitive de le faire est de faire lire les trois arguments par la macro :
#1 : le caractère ⟨*car*⟩ à tester ;
#2 : le ⟨*code vrai*⟩, exécuté si le test est vrai ;
#3 : le ⟨*code faux*⟩, exécuté si le test est faux.

Ensuite, il suffit d'imbriquer 2 tests comme nous l'avons déjà fait et selon l'issue de ces tests, écrire à l'intérieur des branches des tests les arguments #2 ou #3 :

Code nº III-117

```
\def\ifletter#1#2#3{%
  \ifnum‘#1<‘a% si le caractère est avant "a"
    #3% exécuter "<code faux>"
  \else% sinon
    \ifnum‘#1>‘z% si le caractère est après "z"
      #3% exécuter "<code faux>"
    \else% dans tous les autres cas
      #2% "#1" est une lettre : exécuter "<code vrai>"
    \fi
  \fi}
a) \ifletter{4}{vrai}{faux}\qquad
b) \ifletter{t}{vrai}{faux}\qquad
c) \ifletter{D}{vrai}{faux}
```

a) faux b) vrai c) faux

Il y a une certaine inutilité dans ce code qui est de lire #2 et #3 par la macro `\ifletter` pour les réécrire plus loin dans le code. L'idéal serait de ne lire que l'argument #1, de faire le test et selon l'issue, de lire l'un des deux arguments qui suivent (et qui n'ont pas encore été lus).

Méthode nº 2 : `\firstoftwo` et `\secondoftwo`

Les macros `\firstoftwo` et `\secondoftwo` que nous avons écrites à la page 71 permettent de sélectionner un argument parmi deux. C'est d'ailleurs parce qu'elles sont utilisées à l'issue d'un test qu'elles sont si fréquemment rencontrées en programmation. La façon naïve de les placer dans les branches des tests serait d'écrire :

```
\def\ifletter#1{%
    \ifnum‘#1<‘a
        \secondoftwo
    \else
        \ifnum‘#1>‘z
            \secondoftwo
        \else
            \firstoftwo
        \fi
    \fi}
```

Mais agir de cette façon provoquerait des résultats inattendus et des erreurs de compilation. Il ne faut surtout pas oublier que les `\else` et les `\fi` ne sont pas résorbés lorsque le test est fait, mais restent en place pour être développés (et éliminés) plus tard ! Il faut donc faire disparaitre les `\else` et les `\fi` *avant* que `\firstoftwo` ou `\secondoftwo` n'entrent en jeu et ne les prennent à tort comme leurs arguments ! Pour cela, nous savons qu'il faut les 1-développer. Le cas du premier `\secondoftwo` est le plus simple : nous mettrons donc un `\expandafter` avant lui pour faire disparaitre les 7 dernières lignes.

Pour le test imbriqué, c'est un peu plus compliqué. Mettre un `\expandafter` devant `\firstoftwo` et `\secondoftwo` fait disparaitre que le `\else` ou le `\fi` intérieur. Il restera encore le `\fi` de la dernière ligne à 1-développer. Nous devons donc créer un pont d'`\expandafter` à deux passes pour 1-développer d'abord le `\else` ou le `\fi` intérieur puis le `\fi` final.

Voici donc comment coder la macro `\ifletter` :

Code n° III-118

```
\def\ifletter#1{%
  \ifnum‘#1<‘a
    \expandafter\secondoftwo% \expandafter 1-développe "\else...\fi"
  \else
    \ifnum‘#1>‘z
      \expandafter\expandafter\expandafter% 1-développe "\else...\fi" puis "\fi"
      \secondoftwo
    \else
      \expandafter\expandafter\expandafter%1-développe "\fi" puis "\fi"
      \firstoftwo
    \fi
  \fi}
a) \ifletter{4}{vrai}{faux}\qquad
b) \ifletter{t}{vrai}{faux}\qquad
c) \ifletter{D}{vrai}{faux}
```

a) faux b) vrai c) faux

La macro `\ifetter` est purement développable (à condition bien sûr que les instructions contenues dans ⟨*code vrai*⟩ et ⟨*code faux*⟩ le soient).

Méthode n° 3 : `\romannumeral`

La méthode précédente produit une macro purement développable, mais le nombre de développements à faire pour arriver au ⟨*code vrai*⟩ ou ⟨*code faux*⟩ dépend du nombre de tests faits et donc, dépend de l'argument. Ne pas savoir à l'avance combien de développements sont nécessaires pour arriver au résultat escompté peut se révéler gênant dans certains cas particuliers.

Quel que soit son argument, faisons en sorte que la macro `\ifletter` se développe en ⟨*code vrai*⟩ ou ⟨*code faux*⟩ en deux développements. L'astuce, nous l'avons déjà vu, consiste à initier une zone de développement maximal avec la primitive `\romannumeral` et stopper ce développement maximal au bon moment en ajoutant `0␣` ou `\z@` (ou tout autre nombre négatif). Ce moment est ici le début de ⟨*code vrai*⟩ ou ⟨*code faux*⟩ et donc, pour insérer cet ajout, il faut lire les deux derniers arguments de la macro `\ifletter` :

Code n° III-119

```
\def\ifletter#1#2#3{%
  \romannumeral % <- initie le développement maximal
  \ifnum‘#1<‘a
    \expandafter\secondoftwo
  \else
    \ifnum‘#1>‘z
      \expandafter\expandafter\expandafter\secondoftwo
    \else
      \expandafter\expandafter\expandafter\firstoftwo
    \fi
  \fi{0 #2}{0 #3}% <- "0 " stoppe le développement maximal
}
\long\def\>#1<{\detokenize{#1}}
a) \expandafter\expandafter\expandafter\>\ifletter{0}{vrai}{faux}<\qquad
b) \expandafter\expandafter\expandafter\>\ifletter{x}{vrai}{faux}<
```

a) faux b) vrai

Méthode n° 4 : macro auxiliaire

Pour s'épargner tous ces `\expandafter` et si nous n'avons pas besoin d'une macro purement développable, il est judicieux de s'aider d'une macro auxiliaire – appelons-la `\donext` – que l'on rend `\let`-égale à `\firstoftwo` ou `\secondoftwo` dans les branches des tests. Cette macro sera appelée à la fin, après être sorti des branches des tests :

Code n° III-120

```
\def\ifletter#1{%
  \ifnum‘#1<‘a
    \let\donext=\secondoftwo
  \else
    \ifnum‘#1>‘z
      \let\donext=\secondoftwo
    \else
      \let\donext=\firstoftwo
    \fi
  \fi
  \donext% exécuter l'action décidée ci-dessus
}
a) \ifletter{4}{vrai}{faux}\qquad
b) \ifletter{t}{vrai}{faux}\qquad
c) \ifletter{D}{vrai}{faux}
```

a) faux b) vrai c) faux

Méthode n° 5 : `\csname` et `\endcsname`

La méthode de loin la plus élégante, mais hélas la plus lente est d'écrire les tests *à l'intérieur* de la paire `\csname ...\endcsname`. Une fois que les tests seront développés, que les branches inutiles auront disparu suite au développement maximal initié par `\csname`, il devra rester entre `\csname` et `\endcsname` soit « `firstoftwo` », soit « `secondoftwo` ». La fin du mot « `oftwo` » étant commune à toutes les issues possibles, il est possible de *factoriser* le code en mettant cette partie de mot à la toute fin, juste avant le `\endcsname`. Cette méthode présente l'avantage de donner

⟨*code vrai*⟩ ou ⟨*code faux*⟩ en trois développements, quel que soit l'argument donné à `\ifletter` :

Code n° III-121

```
\def\ifletter#1{% \ifnum
  \csname
    \ifnum‘#1<‘a
      second% <- commenter la fin de ligne pour éviter un
    \else%          espace parasite dans le nom de la macro créée
      \ifnum‘#1>‘z
        second%
      \else
        first%
      \fi
    \fi
    oftwo%
  \endcsname
}
a) \ifletter{4}{vrai}{faux}\qquad
b) \ifletter{t}{vrai}{faux}\qquad
c) \ifletter{D}{vrai}{faux}
```

a) faux b) vrai c) faux

■ **Exercice 57**

Comment faudrait-il modifier ce dernier code pour que `\ifletter` teste si son argument est une lettre de l'alphabet, aussi bien minuscule que majuscule ?

□ **Solution**

La primitive `\lowercase` est utile dans ce cas puisqu'elle change toutes les lettres en lettres minuscules et laisse inchangées les séquences de contrôle :

Code n° III-122

```
\def\ifletter#1{%
  \lowercase{\csname
    \ifnum‘#1<‘a second%
    \else
      \ifnum‘#1>‘z second\else first\fi
    \fi
  oftwo\endcsname
  }%
}
a) \ifletter{4}{vrai}{faux}\qquad
b) \ifletter{t}{vrai}{faux}\qquad
c) \ifletter{D}{vrai}{faux}
```

a) faux b) vrai c) vrai

Hélas, la primitive `\lowercase` n'est *pas* développable ! Autrement dit, elle n'a pas vocation à être *développée,* mais *exécutée* c'est-à-dire que le remplacement des lettres par des lettres minuscules n'est pas fait sur la pile de TEX mais plus tard, en interne, lors de l'exécution. Le code suivant montre qu'un `\edef` ne provoque aucun développement sur cette primitive :

Code n° III-123

```
\edef\foo{\lowercase{AbCd}}\meaning\foo
```

macro:->\lowercase {AbCd}

Pour en revenir à la macro \ifletter, si nous souhaitons qu'elle reste purement développable, il faut lui faire exécuter deux tests de plus (ce qui suppose autant d'imbrications supplémentaires) pour tester la place qu'occupe l'entier ‘⟨*car*⟩ par rapport aux entiers ‘A et ‘Z :

Code n° III-124

```
\def\ifletter#1{%
  \csname
    \ifnum‘#1<‘A second%
    \else
      \ifnum‘#1<‘Z first%
      \else
        \ifnum‘#1>‘a
          \ifnum‘#1<‘z
            first%
          \else
            second%
          \fi
        \else
          second%
        \fi
      \fi
    \fi
    oftwo%
  \endcsname
}
a) \ifletter{4}{vrai}{faux}\qquad
b) \ifletter{t}{vrai}{faux}\qquad
c) \ifletter{D}{vrai}{faux}\qquad
d) \edef\foo{\ifletter{x}{vrai}{faux}}\meaning\foo\qquad
e) \edef\foo{\ifletter{=}{vrai}{faux}}\meaning\foo
```

a) faux b) vrai c) vrai d) macro:->vrai e) macro:->faux

Ici encore, 3 développements sont nécessaires pour que \ifletter{⟨*car*⟩}{⟨*vrai*⟩}{⟨*faux*⟩} se développe en ⟨*vrai*⟩ ou ⟨*faux*⟩ comme le montre ce schéma où une flèche représente un développement :

$$\texttt{\textbackslash ifletter\{}\langle car\rangle\texttt{\}} \longrightarrow \texttt{\textbackslash csname...\textbackslash endcsname} \longrightarrow \begin{array}{l}\texttt{\textbackslash firstoftwo}\\ \text{ou}\\ \texttt{\textbackslash secondoftwo}\end{array} \longrightarrow \begin{array}{l}\langle vrai\rangle\\ \text{ou}\\ \langle faux\rangle\end{array}$$ ■

Retour sur la méthode n° 1

Bien que la méthode n° 1 (dont le code est rappelé ci-dessous) semble identique aux autres méthodes, il n'en est rien !

```
\def\ifletter#1#2#3{%
    \ifnum‘#1<‘a
        #3%
    \else
        \ifnum‘#1>‘z
            #3%
        \else
            #2%
        \fi
    \fi}
```

Le fait que les arguments #2 et #3 soient insérés *dans* les branches du test (alors qu'avec \firstoftwo et \secondoftwo, ils restent au dehors pour les autres méthodes) signifie que ces arguments peuvent agir sur ce qui se trouve après eux dans ces mêmes branches… Et éventuellement tout casser ! Imaginons par exemple que l'argument #3 soit la macro \gobtwo et que le premier test soit vrai. Lorsque #3, c'est-à-dire la macro \gobtwo va être exécutée, elle va capturer les deux arguments qui la suivent pour les faire disparaitre. Tout va se passer comme si un \else et un \ifnum étaient retirés du texte de remplacement. Ces deux primitives n'existant plus, l'équilibre entre les \fi et les primitives de test est rompu et TEX, lorsqu'il rencontrera un \fi excédentaire, émettra l'erreur de compilation « Extra \fi ».

Il est donc particulièrement important qu'une macro effectuant un test avec la syntaxe

\if⟨*test*⟩⟨*arguments*⟩{⟨*code vrai*⟩}{⟨*code faux*⟩}⟨*code qui suit*⟩

rejette *au-dehors* des branches des tests les arguments ⟨*code vrai*⟩ et ⟨*code faux*⟩ de telle sorte que le code qu'ils contiennent puisse éventuellement agir sur le ⟨*code qui suit*⟩. La méthode n° 1, puisque contrevenant à ce principe, est une méthode à déconseiller.

2.2.5. Exercices sur les entiers

■ **Exercice 58**

Programmer une macro \ifinside* dont voici la syntaxe :

\ifinside *n*[*a*,*b*]{⟨*code vrai*⟩}{⟨*code faux*⟩}

où n, a et b sont 3 entiers avec $a \leqslant b$. La macro \ifinside teste si n appartient à l'intervalle fermé $[a\,;\,b]$.

□ **Solution**

La première idée est d'imbriquer des tests sur les inégalités. Il faut exécuter ⟨*code faux*⟩ si $n < a$ ou si $n > b$, et exécuter ⟨*code vrai*⟩ sinon :

Code n° III-125

```
\def\ifinside#1[#2,#3]{%
  \csname
    \ifnum#1<#2 second% si n<a, <code faux>
    \else
      \ifnum#1>#3 second% si n>b, <code faux>
      \else       first%  sinon, <code vrai>
      \fi
    \fi
  oftwo%
  \endcsname
}
a) \ifinside 3[1,8]{oui}{non}\qquad
b) \ifinside -7[-20,-11]{oui}{non}\qquad
c) \ifinside 9[0,6]{oui}{non}\qquad
d) \ifinside -1[-5,1]{oui}{non}
```

a) oui b) non c) non d) oui

Une autre approche serait de considérer l'entier

$$(n-a)(n-b)$$

Si cet entier est négatif ou nul, n est dans l'intervalle. Autrement dit, `\ifinside` doit exécuter ⟨*code faux*⟩ lorsque cet entier est strictement positif. En s'aidant de `\numexpr`, un seul test `\ifnum` devient alors nécessaire pour décider si n est dans l'intervalle ou pas :

Code n° III-126

```
\def\ifinside#1[#2,#3]{%
  \ifnum\numexpr(#1-#2)*(#1-#3)\relax>0
    \expandafter\secondoftwo
  \else
    \expandafter\firstoftwo
  \fi
}
a) \ifinside 3[1,8]{oui}{non}\qquad
b) \ifinside -7[-20,-11]{oui}{non}\qquad
c) \ifinside 9[0,6]{oui}{non}\qquad
d) \ifinside -1[-5,1]{oui}{non}
```

a) oui b) non c) non d) oui

■

■ Exercice 59

Programmer une macro `\absval*{entier}`, purement développable et qui se développe en la valeur absolue de l'entier.

□ Solution

La première idée est de tester si l'entier est négatif auquel cas, nous prendrons l'opposé de ce nombre en mettant un signe « – » devant :

Code n° III-127

```
\def\absval#1{%
  \ifnum#1<0 \number-#1
  \else      \number#1
  \fi}
a) \absval{-87}\qquad b) \absval{75}\qquad c) \absval{0}
```

a) 87 b) 75 c) 0

Mais la meilleure solution serait de se servir du fait que la primitive `\number` développe au maximum tout ce qu'elle trouve jusqu'à trouver un caractère qui ne peut pas appartenir à un nombre. Nous pouvons donc mettre le test *dans* le nombre évalué par `\number`, c'est-à-dire après cette primitive. Le développement maximal de ce test laissera « – » si le nombre est négatif et rien sinon :

Code n° III-128

```
\def\absval#1{\number\ifnum#1<0 -\fi#1 }
a) \absval{-87}\qquad b) \absval{75}\qquad c) \absval{0}
```

a) 87 b) 75 c) 0

Un fonctionnalité intéressante est d'autoriser des expressions arithmétiques dans l'argument de `\absval`. Pour ce faire, nous pouvons évaluer cet argument avec `\numexpr` :

Code nº III-129

```
\def\absval#1{\number\ifnum\numexpr#1\relax<0 -\fi\numexpr#1\relax}
a) \absval{-87}\qquad
b) \absval{75}\qquad
c) \absval{0}\qquad
d) \absval{-20+13}\qquad% -7 affiché 7
e) \absval{5-3*(-4)+10-50}% -23 affiché 23
```

a) 87 b) 75 c) 0 d) 7 e) 23

Le seul inconvénient est que nous évaluons *deux* fois l'expression arithmétique #1 ce qui est redondant et source de ralentissement. Un programmeur rigoureux décomposerait le travail et écrirait deux macros. La première serait chargée d'évaluer l'expression arithmétique et de passer comme argument à la seconde macro le *nombre explicitement écrit* qui a été obtenu. Celle-ci serait chargée de donner la valeur absolue de son argument :

Code nº III-130

```
\catcode`\@11
\def\absval#1{\exparg\absval@i{\number\numexpr#1\relax}}
\def\absval@i#1{\number\ifnum#1<\z@-\fi#1 }
\catcode`\@12
a) \absval{-87}\qquad
b) \absval{75}\qquad
c) \absval{0}\qquad
d) \absval{-20+13}\qquad% résultat -7 qui est affiché 7
e) \absval{5-3*(-4)+10-50}% résultat -23 qui est affiché 23
```

a) 87 b) 75 c) 0 d) 7 e) 23

■

■ Exercice 60

Nous avions vu au chapitre précédent la macro `\truncdiv` qui appliquait la formule suivante pour calculer la troncature du quotient de x par y en utilisant la division de `\numexpr` :

$$\frac{x - \frac{y-1}{2}}{y}$$

Voici le code de la macro tel que nous l'avions élaboré :

```
\def\truncdiv#1#2{\numexpr(#1-(#2-1)/2)/#2\relax}
```

La formule n'étant valable que pour x et y positifs, cette macro donnait des résultats erronés pour des arguments négatifs. Modifier la macro `\truncdiv` pour qu'elle donne le bon résultat, quels que soient les signes de ses deux arguments.

□ Solution

Déjà, la formule donnée est équivalente à

$$\frac{2x - y + 1}{2y}$$

qui sera sans doute un peu plus rapide puisqu'il n'y a qu'une seule division.

Puisqu'elle n'est valable que pour les entiers positifs, nous allons reprendre le code de la macro `\truncdiv` donné dans l'énoncé en remplaçant les arguments par leur valeur absolue. Pour que le quotient final tienne compte des signes des arguments, nous allons multiplier le quotient par 1, précédé du signe du premier argument et du signe du second. De cette façon,

s'ils sont tous les deux positifs, nous aurons −−1 et la primitive \number sait que ce nombre est 1. Si l'on note σ_x le signe de x, cela donne

$$\sigma_x\sigma_y 1 \times \frac{2|x| - |y| + 1}{2|y|}$$

Pour cela, reprenons la macro \absval et créons une macro \sgn⋆ qui se développe en « − » si son argument est strictement négatif :

Code n° III-131

```
\def\sgn#1{\ifnum#1<0 -\fi}
\def\truncdiv#1#2{%
  \numexpr
    \sgn{#1}\sgn{#2}1*% multiplie le quotient ci-dessous par +1 ou -1
    (2*\absval{#1}-\absval{#2}+1)/(2*\absval{#2})
  \relax
}
a) \number\truncdiv{-8}3\qquad
b) \number\truncdiv{-8}{-3}\qquad
c) \number\truncdiv8{-3}\qquad
d) \number\truncdiv{0}{-5}
```

a) -2 b) 2 c) -2 d) 0

Une optimisation possible de la formule est d'écrire au dénominateur $2y$ (qui contient le signe de y) et supprimer σ_y au début. Il est aussi possible de déplacer σ_x au dénominateur :

$$\frac{2|x| - |y| + 1}{\sigma_x 2y}$$

Si y est une expression arithmétique, il est plus prudent de mettre #2 (qui représente y) entre parenthèses au dénominateur. Cela nous donnerait la macro

```
\def\truncdiv#1#2{%
    \numexpr(2*\absval{#1}-\absval{#2}+1)/(\sgn{#1}2*(#2))\relax
}
```

En l'état, elle n'est pas satisfaisante, car si #1 est une expression arithmétique, \sgn{#1} va provoquer une erreur puisque rien n'est prévu pour évaluer l'argument de \sgn. De plus, l'argument #2 est évalué *deux* fois : une fois par \absval et une autre dans la parenthèse au dénominateur.

Il faut mettre un peu d'ordre et décider que les deux arguments de \truncdiv doivent tout d'abord être calculés par une première macro puis être transmis en tant que nombres explicitement écrits à une seconde macro qui fera le calcul. Comme les arguments sont déjà calculés, il ne faut pas que absval ne les calcule à nouveau. Abandonnons donc cette version de \absval pour écrire « \sgn{⟨*entier*⟩}⟨*entier*⟩ » ce qui arithmétiquement, est la valeur absolue de l'⟨*entier*⟩ :

Code n° III-132

```
\catcode`\@11
\def\sgn#1{\ifnum#1<\z@-\fi}
\def\truncdiv#1#2{%
  \exptwoargs\truncdiv@i{\number\numexpr#1\relax}{\number\numexpr#2\relax}%
}
\def\truncdiv@i#1#2{%
  \numexpr(2*\sgn{#1}#1-\sgn{#2}#2+1)/(\sgn{#1}2*#2)\relax
}
\catcode`\@12
```

```
a) \number\truncdiv{-8}3\qquad
b) \number\truncdiv{-8}{-3}\qquad
c) \number\truncdiv8{-3}\qquad
d) \number\truncdiv{0}{-5}\qquad
e) \number\truncdiv{20+2*3}{4-5*2}% 26/(-6) doit donner -4
```

a) -2 b) 2 c) -2 d) 0 e) -4

■

■ Exercice 61

Élaborer une macro `\siecle*{⟨nombre⟩}` qui, selon le nombre contenu dans son argument, adapte l'écriture des siècles :

`\siecle{19}` affiche XIX$^{\text{e}}$ siècle
`\siecle{-3}` affiche III$^{\text{e}}$ siècle avant J.C.
`\siecle{1}` affiche I$^{\text{er}}$ siècle
`\siecle{0}` affiche <vide>

Pour surélever un texte enfermé dans une boite (ici une \hbox), on aura recours à la primitive `\raise` et on écrira :

`\raise1ex \hbox{⟨exposant⟩}`

pour élever l'⟨*exposant*⟩ de 1ex.

□ Solution

Il s'agit de faire quelque chose sauf si l'argument est nul. Nous pouvons donc écrire

`\ifnum#1=0 \else⟨action à faire⟩\fi`

mais le moteur ε-TEX dispose de la primitive `\unless` qui, placée devant un test, échange les branches correspondant aux issues du test : ce qui est entre le test et `\else` est échangé avec ce qui est entre `\else` et `\fi`. Nous écrirons donc ici :

`\unless\ifnum#1=0 ⟨action à faire⟩\fi`

Pour obtenir des chiffres romains en majuscule, nous allons utiliser la primitive `\uppercase` en ayant pris soin de développer au préalable son argument traduit en chiffres romains par `\romannumeral`. Pour être surs que le nombre évalué est toujours positif, nous écrirons `\romannumeral \absval{#1}`.

Pour l'« exposant », nous allons donc utiliser la primitive `\raise` qui a la faculté d'élever – ici d'1ex – la boite horizontale qui la suit. Cette boite contiendra « e » suivi, si la valeur absolue de l'argument est 1, d'un « r ». Il faudra ensuite insérer « av. J.-C. » si l'argument #1 est strictement négatif. Le code est finalement assez facilement lisible :

Code n° III-133

```
\def\siecle#1{%
  \unless\ifnum#1=0 % ne faire quelque chose que si #1 différent de 0
    \uppercase\expandafter{\romannumeral\absval{#1}}% chiffres romains majuscules
    \raise 1ex \hbox{e\ifnum\absval{#1}=1 r\fi} siècle% affiche l'exposant puis "siècle"
    \ifnum#1<0 {} av. J.-C.\fi% affiche si besoin "av. J.-C."
  \fi
}
a) \siecle{19}\qquad
b) \siecle{-3}\qquad
c) \siecle{1}\qquad
d) \siecle{0}\qquad
```

```
12 e) \siecle{-1}
```

a) XIX^{e} siècle b) III^{e} siècle av. J.-C. c) I^{er} siècle d) e) I^{er} siècle av. J.-C.

Le `\expandafter` de la ligne n° 3 est automatiquement développé puisque `\uppercase` fait partie de ces macros qui doivent être suivie d'une accolade ouvrante et qui donc, développe au maximum ce qui est entre elle et cette accolade. Le développement de cet `\expandafter` provoque celui de `\romannumeral` qui convertit le nombre en chiffres romains avant que `\uppercase` ne joue son rôle et écrive les chiffres romains en majuscule. ■

■ Exercice 62

Programmer une macro `\hdif*{h:m:s}{h':m':s'}` qui calcule le temps qui s'est écoulé entre l'heure `h:m:s` et l'heure `h':m':s'` qui lui est antérieure. L'affichage se fera en heures, minutes et secondes :

`\hdif{18:51:20}{9:20:10}`	affiche	1h 31min 10s
`\hdif{14:20:0}{13:50:0}`	affiche	30min
`\hdif{7:50:20}{5:50:20}`	affiche	2h
`\hdif{7:50:10}{6:50:20}`	affiche	59min 50s
`\hdif{20:00:00}{19:59:15}`	affiche	45s
`\hdif{17:13:15}{14:12:45}`	affiche	3h 30s

□ Solution

La soustraction de durées exprimées en heures, minutes et secondes se fait comme au collège :

1. effectuer les soustractions indépendamment entre les heures, les minutes et les secondes ;
2. si le nombre de secondes est strictement négatif, l'augmenter de 60 et soustraire 1 au nombre de minutes ;
3. puis, si le nombre de minutes est strictement négatif, l'augmenter de 60 et soustraire 1 au nombre d'heures ;

L'algorithme s'annonce finalement assez simple.

Comme pour nous en France, le caractère « `:` » est parfois rendu actif[2], il faut ouvrir un groupe semi-simple et changer son code de catégorie *avant* de lire l'argument de `\hdif`. Le groupe semi-simple sera fermé à la fin, après l'affichage.

La principale méthode de programmation ici consiste à passer les arguments à une macro à arguments délimités `\hdiff@ii` qui isolera les heures, minutes et secondes, effectuera les soustractions entre nombres exprimés dans la même unité et ajustera les résultats s'ils sont négatifs. Pour ce qui est de l'affichage du résultat, les nombres d'heures, minutes et secondes ne seront affichés que s'ils ne sont pas nuls. Enfin, du côté de la typographie, une macro temporaire `\inter@space`, vide au début, sera rendue `\let`-égale à un espace dès qu'un nombre (d'heures ou de minutes) sera affiché. Elle sera insérée avant le nombre suivant et ainsi créera une espace avant minutes ou avant les secondes si celles-ci ont été précédées par l'affichage d'un nombre.

Voici le code commenté :

Code n° III-134

```
1 \catcode`\@11
2 \edef\saved@catcode{\number\catcode`\:}% sauvegarde du catcode de ":"
3 \catcode`\:12 % ":" est désormais un caractère non actif normal
4 \def\hdif{%
```

2. Afin qu'il insère une espace insécable du type `\kern`⟨*dimension*⟩ juste avant « `\string\:` ».

```
  \begingroup% ouvrir un groupe semi-simple
    \catcode`\:12 % modifier le catcode de ":"
    \hdif@i% aller lire les deux arguments
}

\def\hdif@i#1#2{% lit les 2 arguments
    \hdif@ii#1-#2\@nil% et les transmet à la macro à argument délimités
}

\def\hdif@ii#1:#2:#3-#4:#5:#6\@nil{%
    %%%%%% calcul des nombres à afficher %%%%%%
    \edef\nb@hrs{\number\numexpr#1-#4\relax}% différence des heures
    \edef\nb@min{\number\numexpr#2-#5\relax}% différence des minutes
    \edef\nb@sec{\number\numexpr#3-#6\relax}% différence des secondes
    \ifnum\nb@sec<\z@ % si la différence des secondes est <0
      \edef\nb@sec{\number\numexpr\nb@sec+60\relax}% ajouter 60 sec
      \edef\nb@min{\number\numexpr\nb@min-1\relax}% enlever 1 aux minutes
    \fi
    \ifnum\nb@min<\z@ % si les minutes sont <0
      \edef\nb@min{\number\numexpr\nb@min+60\relax}% ajouter 60 min
      \edef\nb@hrs{\number\numexpr\nb@hrs-1\relax}% enlever 1 aux heures
    \fi
    %%%%%% affichage du résultat %%%%%%
    \let\inter@space\empty% pas d'espace avant un nombre pour l'instant
    \ifnum\nb@hrs>\z@ % si les heures sont >0
      \nb@hrs h% afficher les heures et "h"
      \let\inter@space\space% espace pour plus tard
    \fi
    \ifnum\nb@min>\z@ % si les minutes sont >0
      \inter@space% afficher une espace éventuelle
      \nb@min min% afficher les minutes et "min"
      \let\inter@space\space
    \fi
    \ifnum\nb@sec>\z@ % si les secondes sont >0
      \inter@space% afficher une espace éventuelle
      \nb@sec s% afficher les secondes et "s"
    \fi
  \endgroup% fermer le groupe ouvert au début
}
\catcode`\:=\saved@catcode\relax% restaure le code de catégorie de ":"
\catcode`\@12
a) \hdif{10:51:20}{9:20:10}\par
b) \hdif{14:20:0}{13:50:0}\par
c) \hdif{7:50:20}{5:50:20}\par
d) \hdif{7:50:10}{6:50:20}\par
e) \hdif{20:00:00}{19:59:15}\par
f) \hdif{17:13:15}{14:12:45}
```

a) 1h 31min 10s
b) 30min
c) 2h
d) 59min 50s
e) 45s
f) 3h 30s

■

Chapitre 3

Une première récursivité

Le moment est venu de se jeter dans le grand bain de la récursivité ! En effet, nous en savons assez pour sortir de la programmation linéaire et passer au stade suivant.

La définition la plus communément admise, même si elle est un peu laconique tient en ces mots : un algorithme est récursif s'il s'appelle lui-même. Dans le cas d'une macro, cela signifie qu'une macro est récursive si elle s'appelle elle-même, que ce soit directement ou par le truchement d'autres macros intermédiaires. On comprend bien que cela implique qu'une *boucle* se met en place, c'est-à-dire qu'une portion de code sera lue plusieurs fois.

Pour prendre une analogie, nous mettrions en place une récursivité si, pour parcourir un trajet rectiligne de A vers B, nous nous fixions comme règle de reculer de 10 mètres à chaque fois que nous passons sur un point précis C du parcours. Le simple bon sens montre que nous n'arriverons jamais à la fin du parcours. En effet, nous serions prisonniers d'une boucle infinie, condamnés à parcourir éternellement la même portion de 10 mètres ! Pour éviter cet écueil, il faudrait une règle supplémentaire, appelée « point d'arrêt » qui permet de sortir de cette boucle. Par exemple

- ne rebrousser chemin si le nombre de fois que l'on a vu le point C est inférieur à 20 ;
- au point C, continuer sans rebrousser chemin si l'on marché plus d'une heure ;
- ne rebrousser chemin au point C que si l'on a parcouru moins de 1 km.

Il en va de même en informatique. Une récursivité *doit* comporter un point d'arrêt sans quoi elle devient une boucle infinie. Un point point d'arrêt est un test sur un paramètre qui évolue avec les itérations de telle sorte qu'à un moment, ce test change d'état et valide la sortie de boucle.

Ne nous lançons pas dans une macro de grande ampleur et dans un premier temps, restons tout de même modestes. Fixons-nous comme objectif de programmer une commande \ncar* qui admet 2 arguments, le premier est un nombre positif *n* et le second un caractère. Le but de cette macro sera d'afficher le caractère *n* fois.

Cette macro, effectuant un travail très simple, sera un prétexte pour envisager plusieurs méthodes de programmation.

Une première approche

Il suffit d'un peu de réflexion pour bâtir un algorithme effectuant l'action voulue. L'approche la plus naturelle fait appel à une variable qui, dans le monde de TEX sera un compteur :

1) assigner 0 au compteur ;
2) si le compteur est strictement inférieur à *n*
    a) afficher le caractère ;
    b) incrémenter le compteur de 1 ;
    c) retourner en 2 ;
3) fin de l'algorithme

Comment s'y prendre pour coder un tel algorithme en TEX ? Tout d'abord, nous voyons que le code correspondant au point 2 est parcouru plusieurs fois avec des valeurs différentes du compteur. Cette portion de code sera donc nécessairement *récursive*. Par conséquent, le point 1 sera dévolu à une macro qui initialisera le compteur à 0 et passera la main à une deuxième macro, amenée à s'appeler elle-même. Une macro qui procède à des initialisations et passe la main à la *vraie* macro principale s'appelle une macro « chapeau ». Décidons de nommer la macro chapeau \ncar et la macro récursive \ncar@i. Voici l'algorithme précédent présenté sous une forme d'un *pseudocode*, plus proche de la syntaxe avec laquelle on rédige un programme sans toutefois s'encombrer des subtilités syntaxiques :

Afficher *n* fois un caractère

```
macro ncar#1#2
    ii ← 0% initialise le compteur à 0
    val@max ← #1% stocke la valeur maximale
    car@save ← #2% stocke le caractère à afficher
    ncar@i% appeler la macro récursive
fin

macro ncar@i
    si ii < val@max% si la valeur maxi n'est pas atteinte
        afficher car@save
        ii ← ii + 1% incrémenter le compteur
        ncar@i% recommencer
    finsi
fin
```

Il est utile de remarquer que la boucle mise en place dans la macro ncar@i est de type « tant que ». En effet, dans un pseudocode adapté à un langage de plus haut niveau que ne l'est TEX [1], on aurait pu écrire :

1. Le format LATEX fournit une macro \@whilenum qui est un « tant que » où le test fait est une comparaison sur des entiers.

Macro ncar@i

```
macro ncar@i
    tant que ii < val@max% tant que la valeur maxi n'est pas atteinte
        afficher car@save
        ii ← ii + 1% incrémenter le compteur
    fin tant que
fin
```

Il suffit maintenant de traduire en TEX le premier pseudocode, plus proche du niveau de TEX que ne l'est le second :

Code n° III-135

```
\newcount\ii
\catcode`\@=11
\def\ncar#1#2{%
    \ii=0 % initialise le compteur à 0
    \def\val@max{#1}% stocke la valeur maximale
    \def\car@save{#2}% stocke le caractère à afficher
    \ncar@i% appelle la macro récursive
}
\def\ncar@i{%
  \ifnum\ii<\val@max% si la valeur maxi n'est pas atteinte
    \car@save% afficher le caractère
    \advance\ii 1 % incrémenter le compteur
    \ncar@i% recommencer
  \fi
}
\catcode`\@=12
\ncar{7}{*}\par
\ncar{13}W\par
\ncar{10}{foobar }
```

```
*******
WWWWWWWWWWWW
foobar foobar foobar foobar foobar foobar foobar foobar foobar foobar
```

Cela fonctionne comme attendu, et même au-delà puisque le code à reproduire est stocké dans une macro et donc, ce code peut être constitué de plusieurs caractères et pourrait aussi bien contenir des séquences de contrôle. Ce code comporte néanmoins un défaut : la récursivité mise en place n'est pas terminale !

58 - Règle

Une récursivité est terminale lorsque rien n'est laissé sur la pile lors de l'appel récursif.

Pratiquement, la récursivité est terminale si l'appel récursif intervient *en dernier* dans le code de la macro qui s'appelle elle-même ou intervient après avoir supprimé par développement tout ce qui se trouve après cet appel.

Ici ce n'est pas le cas puisque lorsque la macro `\ncar@i` s'appelle elle même à la ligne n° 13, il reste encore le `\fi` avant la fin du code de la macro. Ce `\fi`, résidu non encore résorbé du test `\ifnum`, va rester sur la pile pour être lu plus tard. Par conséquent, dans les exemples ci-dessus, il y aura 7, 13 ou 10 `\fi` qui vont s'accumuler sur la pile pour qu'après le dernier test (qui sera négatif), ils soient tous lus ce qui provoquera leur disparition par développement. Pour ne

pas surcharger inutilement la pile de TEX, il faut supprimer le `\fi` *avant* d'effectuer l'appel récursif. Il suffit ici de placer un `\expandafter` avant l'appel récursif `\ncar@i` pour 1-développer le `\fi`.

Mais il y a plus grave… En plus de la récursivité non terminale, le code donné ci-dessus comporte une faute de programmation. Comme nous l'avons vu, lorsque TEX lit le nombre qui suit le signe de comparaison <, il développe au maximum tout ce qu'il trouve jusqu'à trouver quelque chose qui ne peut entrer dans la composition d'un nombre. Il va donc développer `\val@max` (dont le développement est constitué de chiffres) et va poursuivre en développant *aussi* `\car@save`. Si jamais le développement de `\car@save` est un chiffre ou commence par un ou plusieurs chiffres, ceux-ci vont entrer dans la composition du nombre et, étant absorbés lors de cette lecture, ils ne seront plus disponibles pour l'affichage. Non seulement le test se fera sur une valeur fausse, mais un ou plusieurs caractères prévus pour être affichés seront capturés par le test et rendus indisponibles pour l'affichage.

Nous pouvons en faire l'expérience avec ce code :

Code nº III-136

```
\newcount\ii
\catcode`\@=11
\def\ncar#1#2{%
    \ii=0 % initialise le compteur à 0
    \def\val@max{#1}% stocke la valeur maximale
    \def\car@save{#2}% stocke le caractère à afficher
    \ncar@i% appelle la macro récursive
}
\def\ncar@i{%
  \ifnum\ii<\val@max% si la valeur maxi n'est pas atteinte
    \car@save% afficher le caractère
    \advance\ii 1 % incrémenter le compteur
    \ncar@i% recommencer
  \fi
}
\catcode`\@=12
\ncar{2}{3a}
```

```
aaaaaaaaaaaaaaaaaaaaaa
```

Ici, le premier test qui est fait, alors que `\ii` vaut 0, est :

```
\ifnum\ii<23
```

où le 2 est le développement de `\val@max` et le 3 est le début de celui de `\car@save`. Ce test est vrai et le restera jusqu'à ce que le compteur vaille 23. La macro `\ncar@i` va donc s'appeler 23 fois, et le « a », qui stoppe la lecture du nombre, sera donc affiché 23 fois.

Pour éviter que TEX ne développe trop loin, il *faut* mettre un `\relax`[2] après `\val@max` pour stopper la lecture du nombre. La bonne façon de coder la macro `\ncar@i` est donc :

2. On pourrait aussi mettre une macro qui se développe en un espace puisqu'un espace stoppe la lecture du nombre et, contrairement à `\relax`, est absorbé par cette lecture. On peut donc remplacer `\relax` par `\space`.

```
\def\ncar@i{%
  \ifnum\ii<\val@max\relax
    \car@save
    \advance\ii 1
    \expandafter\ncar@i
  \fi}
```

Voici maintenant deux autres variantes pour coder cette macro :

1. on peut utiliser une macro auxiliaire \donext qui sera égale, selon l'issue du test, aux instructions à effectuer s'il est positif et à \relax sinon. La macro sera appelée après être sorti des branches du test :

```
\def\ncar@i{%
    \ifnum\ii<\val@max\relax
        \def\donext{\car@save \advance\ii 1 \ncar@i}%
    \else
        \let\donext\relax
    \fi
    \donext
}
```

2. une autre méthode, plus élégante et plus économe, car elle se passe d'une macro temporaire consisterait, selon l'issue du test, à lire l'argument entre accolades se trouvant juste après le \fi ou à manger cet argument. Pour ces deux actions, on utilise les macros déjà définies \identity et \gobone (voir page 71) :

```
\def\ncar@i{%
    \ifnum\ii<\val@max\relax
        \expandafter\identity% exécute...
    \else
        \expandafter\gobone% ou mange...
    \fi
    {\car@save \advance\ii 1 \ncar@i}%... ce code
}
```

■ **Exercice 63**

Trouver un moyen pour programmer la macro \ncar{n}{⟨*car*⟩} sans programmer aucune boucle ni aucune récursivité.

L'exercice est vraiment difficile et tient davantage lieu de curiosité ou d'acrobatie TEXnique que de vraie méthode.

□ **Solution**

Nous allons mettre à contribution la primitive \romannumeral et utiliser une de ses propriétés : si un nombre est supérieur à 1000, alors cette macro produit un nombre en chiffres romains qui commence avec autant de fois le caractère « m » (de catcode 12) qu'il y a de milliers dans le nombre passé en argument :

Code n° III-137

```
a) \romannumeral 9600 \qquad b) \romannumeral 12000
```

a) mmmmmmmmmdc b) mmmmmmmmmmmm

Lorsque nous allons appeler la macro \ncar{#1}{#2} il va suffire d'ajouter « 000 » après le nombre #1 pour obtenir #1 fois le caractère « m » via la primitive \romannumeral. Il ne nous

restera plus qu'à « transformer » le caractère « m » vers le caractère que nous souhaitons afficher. Nous avons déjà vu comment le faire à l'aide de \lccode et la primitive \lowercase :

Code n° III-138

```
\def\ncar#1#2{%
  \begingroup
  \lccode`\m=`#2 % dans \lowercase, les "m" deviennent des "#2"
  \lowercase\expandafter{\expandafter\endgroup\romannumeral#1000 }%
}
a) \ncar{7}{*}\qquad
b) \ncar{15}{\%}\qquad
c) \ncar{10}{4}
```

a) ******* b) %%%%%%%%%%%%%%% c) 4444444444

Comme toutes les primitives devant être suivies d'une accolade ouvrante, \lowercase développe au maximum tout ce qui se trouve entre elle et cette accolade. Le développement d'un pont d'\expandafter exploite ici cette propriété pour propager le développement et 1-développer \romannumeral#1000␣. Ce développement est *nécessaire* car sans lui, l'argument de \lowercase serait :

```
\endgroup\romannumeral#1000
```

et comme cet ensemble de tokens ne contient pas le token « m », \lowercase n'aurait aucune action et ce code serait lu tel quel.

Nous répondons bien à l'énoncé de l'exercice, aucune récursivité ou boucle *explicite* n'a été utilisée. Il est évident qu'en coulisses, TEX utilise une boucle lorsqu'il exécute la primitive \romannumeral. ■

Quelle que soit la méthode employée pour parvenir à nos fins, aucune d'entre elles n'est *purement développable*. Si nous écrivions

```
\edef\foo{\ncar{3}{*}}
```

le texte de remplacement de \foo ne serait pas « *** ».. La raison tient au fait que les textes de remplacements de \ncar ou \ncar@i contiennent des séquences de contrôle non développables :

- un compteur \ii ;
- la primitive \def ;
- la primitive \advance.

Une approche purement développable

Pour construire une macro purement développable, il faut proscrire toutes les primitives non développables contenues dans le code actuel : \advance, \relax, \def et les appels aux compteurs. La question qui se pose naturellement est : « sans compteur, comment compter » ?

Le réponse tient dans la primitive de ε-TEX : \numexpr. L'idée est de programmer \ncar pour que cette macro s'appelle elle-même en décrémentant son premier argument avec \numexpr ; cet argument ira donc *en décroissant* de la valeur maxi à 0, alors que c'était l'inverse pour les versions vues jusqu'alors. Le fait que \ncar s'appelle directement elle-même rend inutile toute macro auxiliaire. Voici ce que devient l'algorithme, toujours du type « tant que », qui tend vers une extrême simplicité :

Décrémenter le 1er argument

macro **ncar**#1#2
    si #1 > 0
        afficher #2
        appeler **ncar**{#1-1}{#2}
    fin si
fin

Du côté de TEX, l'appel récursif prendra la forme suivante :

```
\exparg\ncar@i{\number\numexpr#1-1}{#2}
```

où \exparg 1-développe \number qui lance \numexpr qui à son tour, effectue le calcul #1-1. Tout cela donnera un argument de « 5 » si #1 vaut « 6 ». Regardons le code fonctionner sur un exemple :

Code n° III-139

```
\def\ncar#1#2{%
  \ifnum#1>0 % <- espace après le "0"
    #2% affiche le caractère
    \exparg\ncar{\number\numexpr#1-1}{#2}% appel récursif
  \fi
}
a) \ncar{7}{*}\qquad
b) \ncar{13}{W}\qquad
c) \edef\foo{\ncar{5}{7}}\meaning\foo
```

a) ******* b) WWWWWWWWWWWW c) macro:->77777

La première chose importante à noter que le 0 est suivi d'un espace ce qui stoppe la lecture du nombre et qui met à l'écart #2, même s'il commence par des chiffres. La seconde remarque est que cette récursivité n'est pas terminale puisque le \fi n'est pas mangé avant l'appel récursif. Comme cela a été vu précédemment, les macros \identity et \gobone règlent ce problème :

Code n° III-140

```
\def\ncar#1#2{%
  \ifnum#1>0 \expandafter\identity% si #1>0 exécuter...
  \else      \expandafter\gobone% ...sinon manger
  \fi% ce qui est entre ces accolades :
    {#2% afficher le caractère
    \exparg\ncar{\number\numexpr#1-1}{#2}% appel récursif
    }%
}
a) \ncar{7}{*}\qquad
b) \ncar{13}{W}\qquad
c) \edef\foo{\ncar{5}X}\meaning\foo
```

a) ******* b) WWWWWWWWWWWW c) macro:->XXXXX

Jouer avec le \fi

Une autre ruse de TEXpert consiste à écrire une macro à argument délimité qui *inverse* l'ordre d'un ⟨*code*⟩ quelconque et du prochain \fi, sous réserve que le ⟨*code*⟩ ne contienne pas un \fi :

```
\long\def\antefi#1\fi{\fi#1}
```

La macro \antefi lit tout le code #1 jusqu'au prochain \fi qui sert de délimiteur. Ce \fi, partie intégrante de la macro, n'est pas exécuté et n'est donc pas pas pris en compte par le compteur interne de \fi. Ce n'est que lorsque \antefi se développera que le \fi, token rendu en premier, marquera pour TEX la fin de la portée du test.

Code n° III-141

```
\long\def\antefi#1\fi{\fi#1}
\def\ncar#1#2{%
  \ifnum#1>0
    \antefi% comme si le \fi était "déplacé" ici
    #2% affiche le caractère
    \exparg\ncar{\number\numexpr#1-1}{#2}% appel récursif
  \fi
}
a) \ncar{7}{*}\qquad
b) \ncar{13}{W}\qquad
c) \edef\foo{\ncar{5}X}\meaning\foo
```

a) ******* b) WWWWWWWWWWWW c) macro:->XXXXX

La présence d'un \else provoquerait une erreur si le \antefi se trouvait avant le \else. Dans ce cas, le \antefi se développerait en \fi suivi d'un code #1 contenant un \else qui du coup, se retrouverait orphelin, en dehors du territoire du test. En revanche, \antefi fonctionne bien dans la branche \else...\fi d'un test.

■ **Exercice 64**

Comment peut-on modifier la définition de \antefi pour que l'astuce fonctionne aussi bien avec la présence d'un \else que sans ?

□ **Solution**

Nous pouvons définir comme argument *non délimité* le code à mettre après le \fi. Appelons \afterfi* cette macro qui aura pour mission de placer après le \fi son argument :

```
\long\def\afterfi#1#2\fi{#2\fi#1}
```

De cette façon, si l'argument délimité #2 contient un \else, ce \else est recopié tel quel et se trouve devant le \fi et donc n'occasionnera aucune erreur de compilation.

Code n° III-142

```
\long\def\afterfi#1#2\fi{#2\fi#1}
\def\ncar#1#2{%
  \ifnum#1>0
    \afterfi{\exparg\ncar{\number\numexpr#1-1}{#2}}% <- argument renvoyé après le \fi
    #2% <- ceci reste avant le \fi
  \fi
}
a) \ncar{7}{*}\qquad
b) \ncar{13}{W}\qquad
c) \edef\foo{\ncar{5}X}\meaning\foo
```

a) ******* b) WWWWWWWWWWWW c) macro:->XXXXX

■

Une solution avec \expandafter

Si nous étions était *certains* que l'argument #2 était composé d'un unique token, nous pourrions nous passer de \antefi ou \afterfi et sauter ce token pour

aller développer le \fi et rendre la récursivité terminale. Il suffirait de propager le développement avec des \expandafter :

```
\def\ncar#1#2{%
  \ifnum#1>0%
    #2%
    \expandafter\ncar\expandafter{\number\numexpr#1-1\expandafter}%
    \expandafter{\expandafter#2\expandafter}%
  \fi
}
```

Mais l'appel \ncar{5}{foobar} ne serait pas une récursivité terminale puisque dans

```
{\expandafter foobar\expandafter}
```

le pont d'\expandafter est très insuffisant pour sauter tous les tokens de « foobar ». Si nous voulons nous passer de \antefi, ce qu'il y a de mieux à faire est de mettre l'argument {#2} après le \fi. Si le test est faux, il faudra manger cet argument avec \gobone et s'il est vrai, {#2} reste en place et devient le 2e argument de \ncar. Pour cela, l'astuce consiste à exploiter le fait que la primitive \numexpr engage un développement maximal. Il suffit donc de placer un \expandafter à la fin de cette zone pour développer (et donc faire disparaitre) la branche \else...\fi :

Code nº III-143

```
\def\ncar#1#2{%
  \ifnum#1>0
    #2% afficher le caractère
    \expandafter\ncar\expandafter% le pont d'\expandafter lance \number et \numexpr
    {\number\numexpr#1-1\expandafter}% le dernier \expandafter mange "\else...\fi"
  \else
    \expandafter\gobone% si le test est faux, manger {#2} ci-dessous
  \fi{#2}%
}
a) \ncar{7}{*}\qquad
b) \ncar{13}{W}\qquad
c) \edef\foo{\ncar{5}X}\meaning\foo
```

a) ******* b) WWWWWWWWWWWW c) macro:->XXXXX

Définir une sous-macro

Examinons une dernière méthode pour construire une récursivité terminale. À l'intérieur de la macro \ncar, nous allons définir une « sous-macro » nommée \ncar@i qui elle, n'admettra qu'un seul argument, le nombre noté i qui compte les itérations. Grâce à l'imbrication des macros, les arguments de la macro mère \ncar seront accessibles dans le texte de remplacement de macro fille \ncar@i. Rappelons-nous que le token # doit être doublé à chaque niveau d'imbrication de \def pour distinguer les arguments de la macro mère de ceux de la macro fille.

Nous compterons dans *l'ordre croissant*, de 0 la valeur maxi.

Code nº III-144

```
\catcode`\@11
\def\ncar#1#2{%
  \def\ncar@i##1{%
    \ifnum##1<#1 % si i < valeur maxi
```

```
    #2% afficher le caractère
    \exparg\ncar@i{\number\numexpr##1+1\expandafter}% puis recommencer avec i+1
    \fi% \fi mangé par le \expandafter en fin de zone de \numexpr
  }%
  \ncar@i{0}% commence avec la valeur "compteur" 0
}
\catcode`\@12
a) \ncar{7}{*}\qquad
b) \ncar{13}{W}\qquad
c) \ncar{5}{X}
```

a) ******* b) WWWWWWWWWWWW c) XXXXX

L'inconvénient de cette méthode est que la macro \ncar n'est pas purement développable puisque son texte de remplacement contient la primitive \def \ncar.

■ Exercice 65

Écrire une macro \compte{⟨*nombre*⟩} qui affiche les entiers de 1 à ⟨*nombre*⟩, séparés les uns des autres par une virgule et une espace. Si le ⟨*nombre*⟩ est négatif ou nul, aucun affichage ne sera produit.

On écrira un code non purement développable utilisant un compteur et un code purement développable. Dans tous les cas, la récursivité devra être terminale.

□ Solution

Voici un code qui n'est pas purement développable. Comme au premier exemple vu avec \ncar, nous utilisons un compteur et nous stockons la valeur maximale dans la séquence de contrôle \val@max.

Code n° III-145

```
\catcode`\@11 \newcount\compte@cnt
\def\compte#1{%
  \def\val@max{#1}\compte@cnt=0 % effectue les initialisations
  \compte@i% appelle la macro principale
}
\def\compte@i{%
  \ifnum\compte@cnt<\val@max% si compteur < valeur maxi
    \advance\compte@cnt1 % incrémenter le compteur
    \number\compte@cnt, % afficher le nombre, la virgule et l'espace
    \expandafter\compte@i %recommencer
  \fi
}
\catcode`\@12
a) \compte{9}\qquad
b) \compte{-4}\qquad
c) \compte{2}
```

a) 1, 2, 3, 4, 5, 6, 7, 8, 9, b) c) 1, 2,

Le problème saute aux yeux et révèle une erreur de programmation : la virgule et l'espace sont aussi affichés après le dernier nombre, ce qui ne devrait pas être le cas.

La méthode la plus immédiate est de tester à l'intérieur du test principal si, après incrémentation, \compte@cnt est inférieur à \val@max et si tel est le cas, afficher « , ␣ ». Voici donc le code corrigé :

Code n° III-146

```
\catcode`\@11 \newcount\compte@cnt
\def\compte#1{\def\val@max{#1}\compte@cnt=0 \compte@i}
\def\compte@i{%
  \ifnum\compte@cnt<\val@max\relax
    \advance\compte@cnt1
    \number\compte@cnt
    \ifnum\compte@cnt<\val@max , \fi% afficher ", " si le maxi n'est pas atteint
    \expandafter\compte@i
  \fi
}
\catcode`\@12
a) \compte{9}\qquad
b)\compte{-4}\qquad
c) \compte{2}
```

a) 1, 2, 3, 4, 5, 6, 7, 8, 9 b) c) 1, 2

Bien qu'il fonctionne parfaitement, ce code n'est pas satisfaisant parce que le test de la ligne n° 7 est fait à chaque itération. Certes, le ralentissement qu'il occasionne à chaque fois est insignifiant, mais il est réel et il se cumule au fur et à mesure des appels. Un programmeur rigoureux optimiserait ce code.

Essayons de supprimer ce test. Nous allons tout d'abord rajouter un test dans la macro `\compte` pour tester si son argument est strictement positif, seul cas où quelque chose sera affiché. Ce test ne sera donc fait qu'une fois. Dans le cas où il est positif, la macro récursive `\coompte@i` sera appelée et elle produira l'affichage en deux étapes :

- la valeur du compteur sera affichée ;
- le test décidant si la macro s'appelle elle-même sera fait et s'il est positif, cela signifie qu'il reste encore des valeurs à afficher. La virgule et l'espace seront donc affichés avant que le compteur ne soit incrémenté et l'appel récursif ne soit exécuté.

Code n° III-147

```
\catcode`\@11 \newcount\compte@cnt
\def\compte#1{%
  \ifnum#1>0 % ne faire quelque chose que si l'argument est positif
    \def\val@max{#1}\compte@cnt=1 % faire les initialisations
    \expandafter\compte@i% appeller la macro récrusive
  \fi
}
\def\compte@i{%
  \number\compte@cnt\relax % afficher le nombre
  \ifnum\compte@cnt<\val@max\relax % si valeur maxi n'est pas atteinte
    , % afficher la virgule et l'espace
    \advance\compte@cnt1 % incrémenter le compteur
    \expandafter\compte@i % recommencer
  \fi
}
\catcode`\@12
a) \compte{9}\qquad
b) \compte{-4}\qquad
c) \compte{2}
```

a) 1, 2, 3, 4, 5, 6, 7, 8, 9 b) c) 1, 2

Passons maintenant à une solution purement développable dans laquelle, pour compter les itérations, nous allons utiliser la primitive `\numexpr`. Nous allons traduire l'algorithme vu ci-dessus de façon à nous passer de compteur. Il suffit de passer à la macro `\compte@i` deux arguments, l'un qui contient la valeur à afficher (et qui compte les itérations à partir

de 1) et l'autre la valeur maximale qui est l'argument #1 de \compte. Comme vu dans un exemple précédent, nous rendons cette fois-ci la récursivité terminale en mettant le code *entier* contenant l'appel récursif dans un argument après le \fi : cet argument sera lu tel quel avec \identity si le test est positif et sera mangé avec \gobone dans le cas contraire.

Code n° III-148

```
\catcode`\@11
\def\compte#1{%
  \ifnum#1>\z@% ne faire quelque chose que si #1 > 0
    \antefi\compte@i{1}{#1}%
  \fi
}

\def\compte@i#1#2{%
  #1% afficher le nombre
  \ifnum#1<#2 % si le maxi n'est pas atteint
    \expandafter\identity% exécuter...
  \else
    \expandafter\gobone% sinon, manger...
  \fi% le code entre accolades ci-dessous
    {, % afficher la virgule et l'espace
    \exparg\compte@i{\number\numexpr#1+1}{#2}% appel récursif
    }%
}
\catcode`\@12
a) \compte{9}\qquad
b) \compte{-4}\qquad
c) \edef\foo{\compte{2}}\meaning\foo
```

a) 1, 2, 3, 4, 5, 6, 7, 8, 9 b) c) macro:->1, 2

Une variante plus concise aurait été possible en utilisant \antefi dans la macro auxiliaire :

Code n° III-149

```
\catcode`\@11
\def\compte#1{%
  \ifnum#1>\z@% ne faire quelque chose que si #1 > 0
    \antefi\compte@i{1}{#1}%
  \fi
}

\def\compte@i#1#2{%
  #1% afficher le nombre
  \ifnum#1<#2 % si le maxi n'est pas atteint
    \antefi% déplace le \fi ici
    , % affiche la virgule et l'espace
    \exparg\compte@i{\number\numexpr#1+1}{#2}% appel récursif
  \fi
}
\catcode`\@12
a) \compte{9}\qquad
b) \compte{-4}\qquad
c) \edef\foo{\compte{2}}\meaning\foo
```

a) 1, 2, 3, 4, 5, 6, 7, 8, 9 b) c) macro:->1, 2

■

Il faut tirer une leçon de ce chapitre. La programmation de macros récursives purement développables, disqualifiant de fait tout stockage d'information via un

compteur, `\def` ou `\toks`, oblige à transmettre ces informations sous forme d'arguments. Il devient donc nécessaire de fournir à chaque itération tous les arguments nécessaires au fonctionnement de la macro même si parfois, des arguments invariables (comme la valeur maximale #2 ci-dessus) doivent être lus à chaque fois.

Rendre une macro récursive purement développable est l'assurance qu'elle va fonctionner comme on l'attend dans tous les cas et à ce titre, tend à devenir le Graal que certains recherchent à tout prix. Il est cependant bon de garder à l'esprit que la facilité et la sécurité que l'on gagne d'un côté cachent une certaine lourdeur dans les mécanismes mis en jeu dans sa programmation. Les contorsions que l'on doit faire notamment au niveau des arguments pour rendre une macro purement développable impliquent généralement une programmation beaucoup plus délicate que si elle ne n'était pas.

Chapitre 4

Une boucle « for »

Nous savons à présent suffisamment de choses pour programmer une structure de contrôle de type « for » où une « variable » – une macro en réalité – va prendre les valeurs entières successives entre deux entiers n_1 et n_2. Fixons-nous par exemple la syntaxe sera la suivante :

$$\texttt{\textbackslash for}\langle \textit{macro} \rangle\texttt{=}n_1\ \texttt{to}\ n_2\texttt{\textbackslash do\{}\langle \textit{code} \rangle$$

où le ⟨*code*⟩ pourra contenir la \<macro qui tient lieu de variable et qui prend successivement toutes les valeurs de n_1 à n_2. Compte tenu de sa syntaxe, il est bien évident que la macro \for doit avoir un texte de paramètre contenant des arguments délimités par « = », « to » et par « \do ».

Une première version

Voici un premier jet, assez intuitif :

Code n° III-150

```
\catcode`\@11
\def\for#1=#2to#3\do#4{% #1=\<macro>   #2=n1   #3=n2   #4=<code>
  \def#1{#2}% initialise la \<macro> à l'entier n1
  \def\val@max{#3}% stocke n2
  \def\loop@code{#4}% stocke le <code>
  \for@i#1% appelle la macro récursive et passe la \<macro> en argument
}
\def\for@i#1{%
  \unless\ifnum#1>\val@max\relax% tant que la \<macro> est <= n2
    \loop@code% effectue le code précédemment sauvegardé
    \edef#1{\number\numexpr#1+1}% incrémenter la \<macro>
    \expandafter\for@i\expandafter#1% et recommencer après avoir mangé le \fi
  \fi
```

```
}
\catcode`\@=12
\for\ii=1to5\do{Maintenant \string\ii\ vaut : \ii.\endgraf}\medbreak
"\for\ii=1to0\do{A}"\medbreak% boucle parcourue 0 fois
"\for\ii=1to1\do{A}"\medbreak% boucle parcourue 1 fois
\for\ii = 0 to 10 \do {[\ii]}.
```

```
Maintenant \ii vaut : 1.
Maintenant \ii vaut : 2.
Maintenant \ii vaut : 3.
Maintenant \ii vaut : 4.
Maintenant \ii vaut : 5.

""

"A"

[ 0 ][1][2][3][4][5][6][7][8][9][10].
```

Remarquons tout d'abord que la séquence de contrôle `\do` n'est jamais définie et peu importe qu'elle le soit [1] : elle ne sert que de délimiteur à l'argument `#3` de la macro `\for`.

À la ligne n° 3, l'argument `#2`, qui est l'entier n_1, est assigné à `#1` qui est la `\⟨macro⟩`. C'est une mauvaise idée si on laisse des espaces – toujours utiles pour la lisibilité – comme c'est le cas à l'appel de la ligne n° 19. Dans ce cas, lors de la première itération, `#1` a comme texte de remplacement « ␣0␣ ». Pour se prémunir de cet inconvénient, il aurait fallu écrire à la ligne n° 3

```
\edef#1{\number\numexpr#2\relax}
```

et ainsi tirer parti du fait que non seulement `\numexpr` évalue l'éventuelle expression arithmétique `#1`, mais le fait en *ignorant les espaces*. La même méthode doit être utilisée pour stocker l'entier n_2 dans `\val@max`.

Plutôt que de définir une macro auxiliaire récursive `\for@i` *à l'extérieur* de la macro principale comme c'est le cas ici, nous avons tout à gagner à la définir à l'intérieur en tant que « macro fille ». De cette façon, tous les arguments de la macro chapeau sont accessibles. Nous économisons donc la macro `\loop@code`.

Remarquons enfin que la macro `\for` n'était pas déclarée `\long`, interdisant donc la primitive `\par` dans ses arguments. Ceci explique pourquoi il faut utiliser `\endgraf` (une macro `\let`-égale à `\par`) pour provoquer la fin du paragraphe.

Code n° III-151

```
\catcode`\@11
\long\def\for#1=#2to#3\do#4{%
  \def\for@i{%
    \unless\ifnum#1>\val@max\relax% tant que la \<macro> <= n2
      #4% code à exécuter
      \edef#1{\number\numexpr#1+1}% incrémenter \<macro>
      \expandafter\for@i% et recommencer après avoir mangé le \fi
    \fi
  }%
  \edef#1{\number\numexpr#2\relax}% initialise la variable à n1
  \edef\val@max{\number\numexpr#3\relax}% stocke n2
  \for@i% appelle la sous-macro récursive
```

1. Comme nous l'avons vu, elle est bel et bien définie dans plain-TEX notamment pour seconder la macro `\dospecials`.

```
}
\catcode`\@=12
\for\ii = 1 to 5 \do {Maintenant \string\ii\ vaut : \ii.\par}\medbreak
"\for\ii=1to0\do{A}"\medbreak% boucle parcourue 0 fois
"\for\ii=1to1\do{A}"\medbreak% boucle parcourue 1 fois
\for\ii = 0 to 10 \do {[\ii]}.
```

Maintenant \ii vaut : 1.
Maintenant \ii vaut : 2.
Maintenant \ii vaut : 3.
Maintenant \ii vaut : 4.
Maintenant \ii vaut : 5.

""

"A"

[0][1][2][3][4][5][6][7][8][9][10].

Mise en place d'un incrément

En l'état actuel, la macro `\for` incrémente sa « variable » de 1 à chaque itération. Comment s'y prendre pour que cet incrément puisse être spécifié par l'utilisateur ce qui constituerait en quelque sorte un argument « optionnel » ? Il est naturel de penser à cette syntaxe :

$$\texttt{\textbackslash for\textbackslash}\langle \textit{macro} \rangle\texttt{=}n_1\ \texttt{to}\ n_2\ \texttt{\textbackslash do}\ i\ \texttt{\{}\langle \textit{code} \rangle\texttt{\}}$$

Nous allons modifier la macro `\for` pour que l'incrément i, qui est un entier signé, soit pris en compte. Si celui-ci n'est pas spécifié, il sera pris égal à 1 si $n_1 < n_2$ et à -1 sinon. L'exercice est difficile, d'autant que nous ne savons pas encore tester si un argument est vide, il faudra donc trouver une autre méthode !

Il faut déjà modifier le texte de paramètre de la macro `\for` pour qu'elle prenne un 4e argument après le `\do`, suivi ensuite du 5e qui sera le code à exécuter à chaque itération. Ce serait une erreur que de la définir ainsi :

```
\def\for#1=#2to#3\do#4#5{...}
```

Le problème avec cette définition est que TEX va toujours lire *deux arguments* après le `\do` ! Et si nous omettons l'argument optionnel, `#4` va devenir le ⟨*code*⟩ et `#5` sera l'argument suivant, non destiné à être lu par la macro ! Pour éviter un tel mélange d'arguments, l'argument optionnel recevant *i doit* être délimité par l'accolade ouvrante qui marque le début du ⟨*code*⟩. Nous allons donc écrire :

```
\def\for#1=#2to#3\do#4#{...}
```

Ce faisant, `#4` sera donc tout ce qui se trouve entre `\do` et la prochaine accolade ouvrante, cet ensemble étant éventuellement vide. Mais en procédant de cette façon, l'argument `{`⟨*code*⟩`}` ne sera pas encore lu. Il faudra appeler une macro auxiliaire chargée de lire le code devant être exécuté à chaque fois.

L'autre problème qui surgit est que l'argument `#4` peut être vide, ou égal à +4, -2 ou n'importe quel entier signé. Comment s'y prendre pour transformer cet argument en un nombre, même s'il est vide ? L'astuce consiste à mettre un 0 devant cet argument et évaluer le tout avec `\numexpr`. Nous aurions donc :

```
\edef\for@increment{\number\numexpr0#4\relax}
```

Par acquit de conscience, assurons-nous que cette astuce fonctionne. Voici les cas qui peuvent se présenter :

- si #4 est vide ou réduit à des espaces, alors le « 0 » ajouté sera évalué par `\numexpr` et donc, `\for@increment` vaudra 0 ;
- si #4 vaut « 5 » alors, « 05 » sera évalué et nous obtiendrons 5 ;
- si #4 vaut « +4 » alors, « 0+4 » sera évalué et nous obtiendrons 4 ;
- enfin, si #4 vaut « -10 » alors, « 0-10 » sera évalué et vaudra -10.

Une fois ceci fait, si `\for@increment` est nul, cela traduit une absence d'argument optionnel ou pire, un incrément explicitement donné égal à 0 par l'utilisateur joueur ou inconscient ! Dans ces cas, il suffit de le redéfinir à 1 si $n1 < n_2$ (soit si #2 < #3) et à -1 sinon.

Passons à l'étape suivante : il faut vérifier que le signe de l'incrément (lorsqu'il est explicitement écrit) est « compatible » avec les valeurs de #2 et #3. En effet, si #2 vaut 4, #3 vaut 10 et l'incrément vaut -2, on comprend bien qu'additionner l'incrément -2 à 4 est une tâche vaine. On n'arrivera jamais à 10 et, après un *très grand* nombre d'itérations, on va dépasser la borne inférieure $-2^{31}+1$ permise pour un nombre, provoquant à ce moment (ou même avant) une erreur de compilation.

Un peu d'analyse montre que l'incrément est « incompatible » avec les bornes #2 et #3 si les entiers #3 − #2 et `\for@increment` sont de signes contraires, c'est-à-dire lorsque le signe de l'expression arithmétique suivante est strictement négatif :

```
\for@increment*(#3-#2)
```

Voici le code :

Code n° III-152

```
\catcode`\@11
\def\for#1=#2to#3\do#4#{%
  \edef\for@increment{\number\numexpr0#4}% lit et normalise l'argument optionnel
  \ifnum\for@increment=\z@% s'il est nul,
    \edef\for@increment{% le redéfinir :
      \ifnum\numexpr#3\relax<\numexpr#2\relax
        -% ajoute un signe - si #3<#2
      \fi
      1% devant "1"
    }% \for@increment vaut donc 1 ou -1
  \fi
  \ifnum\numexpr\for@increment*(#3-#2)\relax<\z@% si l'argument est incompatible
    \expandafter\firstoftwo% exécuter le 1er argument qui suit
  \else
    \expandafter\secondoftwo% sinon le second
  \fi
    {Incrément incompatible !\gobone %\gobone mangera le <code> qui à lire
    }%
    {\edef#1{\number\numexpr#2\relax}% initialise la \<macro>
    \edef\cmp@sgn{\ifnum\for@increment<\z@<\else>\fi}% stocke "<" ou ">" pour plus tard
    \expandafter\for@i\expandafter#1\expandafter% appelle la macro récursive
      {\number\numexpr#3\relax}% et lui lui passe la \<macro> (#1) et n2 (#3)
    }%
}

% #1 = \<macro>   #2 = n2
\long\def\for@i#1#2#3{% l'argument #3 (le <code>) est lu à ce moment-là
  \def\for@ii{%
    \unless\ifnum#1\cmp@sgn#2\relax% tant que la \<macro> n'a pas dépassé n2
      #3% exécute le <code>
```

```
31      \edef#1{\number\numexpr#1+\for@increment}% incrémente la \<macro>
32      \expandafter\for@ii% recommence après avoir mangé le \fi
33    \fi
34  }%
35  \for@ii% appelle la sous-macro récursive
36 }%
37 \catcode`\@=12
38 a) \for\ii = -4 to 7 \do{(\ii)}\par
39 b) \for\jj = 20 to -50\do-10 {(\jj)}\par
40 c) \for\xx = 8 to 185\do 20 {[\xx]}\par
41 d) \for\yy = 0 to 10\do -2{(\yy)}\par
42 e) "\for\ii = 1 to 0 \do1{XX}"\par
43 f) "\for\ii = 1 to 1 \do{A}"\par% boucle parcourue 1 fois
44 g) \for\ii=1to4\do{\for\jj=0to2\do{(\ii,\jj)}\par}% imbrication de boucles
```

a) (-4)(-3)(-2)(-1)(0)(1)(2)(3)(4)(5)(6)(7)
b) (20)(10)(0)(-10)(-20)(-30)(-40)(-50)
c) [8][28][48][68][88][108][128][148][168]
d) Incrément incompatible !
e) "Incrément incompatible !"
f) "A"
g) (1,0)(1,1)(1,2)

Il est utile de reprendre ce code en détail, car il comporte beaucoup d'astuces… Laissons pour l'instant le cas g où le problème de l'imbrication de boucles sera résolu un peu plus loin.

À la ligne n° 3, si le nombre `\for@increment`, résultat de l'évaluation de `0#4` est nul, cela signifie que l'argument optionnel `#4` est absent ou égal à `0`. Dans ce cas, la macro `\for@increment` est redéfinie par un `\edef` et un signe « – » est ajouté si `#3` est inférieur à `#2` (ligne n° 6) de telle sorte que `\for@increment` contienne `1` ou `–1`.

La ligne n° 12 teste si `\for@increment` est « compatible » avec les valeurs `#2` et `#3`. À l'aide des macros `\firstoftwo` et `\secondoftwo`, un des 2 arguments qui suit est exécuté :

- le premier argument se contente d'afficher « Incrément incompatible » et appelle la macro `\gobone` qui mange le ⟨*code*⟩ qui n'a pas encore été lu ce qui termine tout le processus. Bien évidemment, il est possible de ne rien afficher et se contenter de mettre `\gobone` dans le deuxième argument ;
- le deuxième argument définit la séquence de contrôle `\cmp@sgn` pour que son texte de remplacement soit « < » si l'incrément est négatif et « > » sinon. Ce signe sera utilisé dans un test plus tard. Il ne reste plus ensuite qu'à appeler la macro auxiliaire `\for@i` en lui passant la \⟨*macro*⟩ (argument `#1`) et l'entier n_2 (argument `#3`) préalablement évalué par `\numexpr`.

La macro `\for@i` lira *trois* arguments, les deux que lui passe la macro `\for` et le ⟨*code*⟩ qui doit être exécuté à chaque itération puisque celui-ci n'a pas été lu par la macro chapeau `\for`. Comme la macro `\for@i` est appelée à être récursive, il est très peu commode – et maladroit – d'installer une récursivité avec trois arguments à lire à chaque appel. Pour éviter cette lourdeur, la macro `\for@i` ne sert que de macro chapeau pour la vraie macro récursive `\for@ii` qui n'admet aucun argument puisque, étant *dans* la macro `\for@i`, elle peut accéder à ses arguments.

Le code de la macro `\for@ii` est assez simple, le ⟨*code*⟩ est exécuter à la ligne n° 30 puis la \⟨*macro*⟩ est incrémentée de la valeur signée `\for@increment`. La ligne n° 29 nécessite un petit commentaire : le test `\ifnum` s'attend à lire un nombre et

développe tout au maximum jusqu'à trouver un token qui ne puisse pas constituer un entier : la macro #1 va être développée et la lecture du nombre ne va pas s'arrêter. Le développement va continuer et la macro \cmp@sgn va être développée à son tour. Comme son texte de remplacement est « < » ou « > », la lecture du nombre va être stoppée, mais le but recherché, c'est-à-dire le développement de \cmp@sgn est effectué. Enfin, le test est construit avec \unless ce qui signifie que ce qui se trouve jusqu'au \fi est exécuté si le test *contraire* est vérifié. Ce test contraire correspond à une inégalité *large* ($\leqslant$ ou $\geqslant$) ce qui implique qu'une itération sera faite lorsque la \⟨*macro*⟩ est égale à n_2.

Imbrication de boucles : analyse du problème

Comme le cas g l'a mis en évidence dans le code précédent, lorsque deux boucles \for sont imbriquées le fonctionnement attendu n'est pas obtenu et il semble qu'une seule itération de la boucle la plus extérieure est faite entièrement :

```
\for\ii=1to4\do{\for\jj=0to2\do{(\ii,\jj)}\par}
```

se traduit par « (1,0)(1,1)(1,2) ».

Il faut inventer un remède qui permet d'imbriquer autant de boucles \for que l'on veut. Mais tout d'abord, il faut analyser le problème. Par chance, c'est assez simple ici. Dans un premier temps, \for@i est appelée par \for ce qui a pour effet de définir la macro \for@ii avec le texte de remplacement (auquel nous donnons le numéro 1) suivant :

```
\unless\ifnum\ii\cmp@sgn4\relax
  \for\jj=0to2\do{(\ii,\jj)}\par
  \edef\ii{\number\numexpr\ii+\for@increment}%
  \expandafter\for@ii
\fi
```

Lorsque la macro \for@ii est exécutée à la ligne n° 35, elle est développée et donc, tout ce code est placé sur la pile pour y être exécuté. Lorsque la 2e ligne de la pile est exécutée, une nouvelle boucle \for démarre :

```
\for\jj=0to2\do{(\ii,\jj)}
```

Comme \ii vaut 1, cette boucle va afficher « (1,0)(1,1)(1,2) » mais en coulisse, l'irréparable s'est déjà produit ! Ce nouvel appel à \for a redéfini la macro \for@ii avec ce texte de remplacement (appelé n° 2) :

```
\unless\ifnum\jj\cmp@sgn2\relax
  (\ii,\jj)%
  \edef\jj{\number\numexpr\jj+\for@increment}%
  \expandafter\for@ii
\fi
```

Que va-t-il se passer quand la boucle interne (dont la variable est \jj) sera terminée et que « (1,0)(1,1)(1,2) » sera affiché ? La macro \jj vaudra 3 et TEX va continuer à lire le code sur sa pile qui est le texte de remplacement n° 1 amputé de ses deux premières lignes :

```
\edef\ii{\number\numexpr\ii+\for@increment}%
  \expandafter\for@ii
\fi
```

Par conséquent, \ii va être incrémenté (et vaudra 2). La macro \for@ii (dont le texte de remplacement est le n° 2) va être appelée et le test

```
\unless\ifnum\jj\cmp@sgn2\relax
```

sera faux car \jj (qui vaut 3) a dépassé n_2 qui est 2. Par conséquent, aucun code ni aucune récursivité ne seront exécutés. Ce test faux signe donc la fin de la boucle extérieure.

Imbrication de boucles : résolution du problème

Tous les problèmes viennent donc de la redéfinition de \for@ii par la boucle intérieure et du fait que cette redéfinition survive à la fin de la boucle intérieure. Certes, exécuter chaque boucle dans un groupe règlerait le problème, mais cette solution de facilité empêcherait toute globalité au code qui est exécuté à chaque itération et donc diminuerait les fonctionnalités de la boucle.

Le remède serait donc de définir une macro \for@ii spécifique à chaque « variable ». Pour cela, la macro chapeau \for va appeler une macro \for@i chargée de définir la macro récursive \for@ii@\<macro> où \⟨*macro*⟩ est le développement de \string#1. Par exemple, si l'on exécute

```
\for\xx=1to5\do{X}
```

la macro \for va appeler \for@i qui va se charger de définir et lancer la macro récursive \for@ii@\xx. Plus aucun risque de redéfinition n'existe, sauf si l'utilisateur fait preuve d'inconscience et utilise le même nom de variable pour deux boucles imbriquées.

Du côté de la méthode à utiliser, nous allons faire en sorte que \for passe à la macro \for@i les arguments suivants :

- #1 : la macro récursive \for@ii@\<macro> ;
- #2 : le signe de comparaison explicitement écrit "<" ou ">" ;
- #3 : l'incrément sous forme d'un entier explicitement écrit ;
- #4 : la \⟨*macro*⟩ qui est l'argument #1 de \for ;
- #5 : l'entier n_2 qui est l'argument #3 de \for.

Par exemple, si nous écrivons

```
\for\xx=1to20\do4{X}
```

la macro \for@i doit être appelée ainsi :

```
\for@i\for@ii@\xx<{4}\xx{20}
```

Le fait que les arguments #2, #3 et #5 soient transmis explicitement sous forme de tokens de catcode 12 évite de créer une macro pour les stocker et nous met donc à l'abri de toute redéfinition de cette macro par une boucle imbriquée.

Prenons le parti de stocker ces 5 arguments dans une macro \macro@args définie avec \edef. Rappelons-nous que dans le texte de remplacement d'un \edef, un \noexpand est nécessaire pour bloquer le développement d'une séquence de contrôle : les arguments #1 et #4 sont ici concernés. Une fois la macro \macro@args définie, l'appel à \for@i se fera donc par

```
\expandafter\for@i\macro@args
```

Code n° III-153

```
\catcode`\@11
\def\for#1=#2to#3\do#4#{%
  \edef\for@increment{\number\numexpr0#4}% lit et normalise l'argument optionnel
  \ifnum\for@increment=\z@% s'il est nul,
    \edef\for@increment{% le redéfinir à -1 (si #3<#2) et 1 sinon
      \ifnum\numexpr#3-#2\relax<\z@ -1\else 1\fi
    }% \for@increment vaut donc 1 ou -1
  \fi
  \ifnum\numexpr\for@increment*(#3-#2)\relax<\z@
    \expandafter\gobone% si argument optionnel incompatible, manger le {<code>}
  \else
    \edef#1{\number\numexpr#2}% initialise la \<macro>
    \edef\macro@args{% définit et développe les arguments à passer à \for@i
      %#1=nom de la macro récursive :
      \expandafter\noexpand\csname for@ii@\string#1\endcsname
      \ifnum\for@increment<\z@ <\else >\fi% #2=signe de comparaison
      {\for@increment}% #3=incrément
      \noexpand#1% #4=\<macro>
      {\number\numexpr#3\relax}% #5=entier n2
    }%
    \antefi% externalise la ligne ci-dessous de la portée du test
    \expandafter\for@i\macro@args% appelle \for@i avec les arguments définis ci-dessus
  \fi
}

% #1=nom de la macro récursive de type "\for@ii@\<macro>"
% #2=signe de comparaison  % #3=incrément
% #4=\<macro>  % #5=entier n2  % #6=<code> à exécuter
\long\def\for@i#1#2#3#4#5#6{%
  \def#1{% définit la sous macro récursive
    \unless\ifnum#4#2#5\relax% tant que la \<macro> variable n'a pas dépassé n2
      \afterfi{% rendre la récursivité terminale
        #6% exécute le code
        \edef#4{\number\numexpr#4+#3\relax}% incrémente la \<macro>
        #1% recommence
      }%
    \fi
  }%
  #1% appelle la sous-macro récursive
}%
\catcode`\@=12
\for\ii=1to4\do{\for\jj=0to2\do{(\ii,\jj)}\par}\medbreak

\for\carA= `\A to `\E \do{\for\carB= `w to `z \do{\char\carA\char\carB}\quad}
```

(1,0)(1,1)(1,2)
(2,0)(2,1)(2,2)
(3,0)(3,1)(3,2)
(4,0)(4,1)(4,2)

AwAxAyAz BwBxByBz CwCxCyCz DwDxDyDz EwExEyEz

■ Exercice 66

En l'état, la boucle est parcourue autant de fois que l'imposent les bornes et l'incrément. Or, dans certains cas, une condition programmée par l'utilisateur devient vraie au cours des itérations et à partir de ce moment, exécuter les itérations suivantes devient indésirable. Comment programmer une macro `\exitfor*` qui, insérée dans le ⟨*code*⟩ à exécuter, permette de sortir prématurément de la boucle ?

□ **Solution**

La macro `\exitfor` est susceptible de se trouver dans le ⟨*code*⟩ à la ligne nº 33. Cette macro doit donc annuler l'appel récursif de la ligne nº 39. Il y a plusieurs moyens de le faire, le plus simple est de charger `\exitfor` de redéfinir la macro récursive `#1` comme ayant un texte de remplacement vide :

Code nº III-154

```
\catcode`\@11
\long\def\for@i#1#2#3#4#5#6{%
  \def\exitfor{\def#1{}}%
  \def#1{% définit la sous macro récursive
    \unless\ifnum#4#2#5\relax% tant que la variable n'a pas dépassé l'entier max
      \afterfi{% rendre la récursivité terminale
        #6% exécute le code
        \edef#4{\number\numexpr#4+#3\relax}% incrémente la variable #1
        #1% recommence
      }%
    \fi
  }%
  #1% appelle la sous-macro récursive
}%
\catcode`\@=12
\for\ii= 1 to 9 \do{\string\ii{} vaut \ii.\ifnum\ii=5 \exitfor\fi\par}
```

```
\ii vaut 1.
\ii vaut 2.
\ii vaut 3.
\ii vaut 4.
\ii vaut 5.
```

Cela fonctionne correctement car il n'y a aucune boucle `for` imbriquée.

Dans le cas de boucles imbriquées, la macro `\exitfor` va être définie plusieurs fois (une fois par imbrication). Par conséquent, la dernière définition, celle de la boucle la plus intérieure, écrase les autres. On peut donc sortir de la boucle la plus intérieure, mais plus des boucles de niveau supérieur.

Le remède le plus simple est de donner comme argument à `\exitfor` la `\`⟨*macro*⟩ correspondant à la boucle de laquelle on souhaite sortir. Ainsi, `\exitfor#1` redéfinirait la macro `\for@ii@#1` comme ayant un texte de remplacement vide. Pour être complets sur le sujet, nous créerons aussi la macro `\ifexitfor`⋆ de syntaxe

`\ifexitfor\`⟨*macro*⟩`{`⟨*code vrai*⟩`}{`⟨*code faux*⟩`}`

Ainsi, `\ifexitfor\`⟨*macro*⟩ permet de savoir, en dehors de la boucle de variable `\`⟨*macro*⟩, si on en est prématurément sorti ou pas. Définir `\ifexitfor` fait appel au test `\ifx` et à la macro `\empty` et devient donc, en anticipant un peu, une bonne transition vers le chapitre suivant qui aborde d'autres tests que propose TEX.

Le code ci-dessous permet de voir plusieurs façons, plus ou moins rapides, de sortir de 2 boucles imbriquées :

Code nº III-155

```
\def\exitfor#1{% #1=\<macro> correspondant à la boucle de laquelle on veut sortir
  \defname{for@ii@\string#1}{}
}

\def\ifexitfor#1{% envoie vrai si on est prématurément sorti de la boucle de \<macro> #1
  % si la macro récursive est égale à la macro "\empty"
  \expandafter\ifx\csname for@ii@\string#1\endcsname\empty
    \expandafter\firstoftwo% c'est qu'on est sortir prématurément, renvoyer "vrai"
```

```
  \else
    \expandafter\secondoftwo% sinon, renvoyer "faux"
  \fi
}

% on ne sort QUE de la boucle intérieure quand \ii=3 donc
% les boucles sont normalement exécutées pour \ii=4 et \ii=5
a. \for\ii=1 to 5\do1{%
     \for\jj=1 to 3\do1{(\ii,\jj) \ifnum\ii=3 \exitfor\jj\fi}%
     \qquad
   }

% on sort de la boucle extérieure lorsque \ii=3 donc la boucle
% intérieure est faite entièrement même lorsque \ii=3
b. \for\ii=1 to 5\do1{%
     \for\jj=1 to 3\do1{(\ii,\jj) \ifnum\ii=3 \exitfor\ii\fi}%
     \qquad
   }

% on sort de la boucle intérieure lorsque \ii=3
% et aussi de la boucle extérieure à l'aide de \ifexitfor
c. \for\ii=1 to 5\do1{%
     \for\jj=1 to 3\do1{(\ii,\jj) \ifnum\ii=3 \exitfor\jj\fi}%
     \ifexitfor\jj{\exitfor\ii}{}% si on est sorti de \jj, sortir aussi de \ii
     \qquad
   }
```

a. (1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (4,1) (4,2) (4,3) (5,1) (5,2) (5,3)
b. (1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (3,2) (3,3)
c. (1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1)

■

Chapitre 5

Quelques autres tests

Après avoir touché du doigt la syntaxe générale des tests et leurs particularités concernant la façon dont ils se développent, après avoir examiné plus particulièrement le test `\ifnum`, nous allons dans ce chapitre découvrir quelques autres tests de TEX. Le but n'est pas de les passer tous en revue, mais d'examiner ceux qui sont le plus utiles et les utiliser pour en créer d'autres.

5.1. Le test `\ifx`

5.1.1. Que teste `\ifx` ?

Un des tests les plus utilisés est sans doute le test `\ifx`.

59 - Règle

Le test `\ifx` compare les significations des deux *tokens* qui le suivent. Il a la syntaxe suivante :

```
\ifx⟨token1⟩⟨token2⟩
    ⟨code vrai⟩
\else
    ⟨code faux⟩
\fi
```

Il est vrai si la signification des deux tokens est la même. Parmi toutes les possibilités qui se présentent selon la nature des deux tokens, les trois cas suivants sont les plus courants :

1. si les deux tokens sont des caractères, alors le test est positif si les caractères

sont identiques, c'est-à-dire si leur code de caractère *et* leur catcode sont égaux ;
2. si les deux tokens sont des primitives, le test est positif si les primitives sont les mêmes ;
3. si les deux tokens sont des macros ou des caractères actifs, le test `\ifx` compare leurs textes de remplacement : le test est positif si les textes de remplacement sont identiques, c'est-à-dire constitués des mêmes tokens avec les mêmes catcodes. La comparaison tient également compte des propriétés de ces macros, notamment si elles sont `\long` ou `\outer`.
   Deux macros non définies sont égales pour `\ifx`, quels que soient leurs noms.

Contrairement à `\ifnum`, le test `\ifx` ne déclenche pas un développement maximal et donc, les deux tokens sont comparés tels quels.

Si deux tokens appartiennent à deux catégories différentes parmi les trois listées dans la règle ci-dessus, le test sera faux. Par exemple, si une macro `\foo` a le texte de remplacement « a », alors le test `\ifx a\foo` est faux :

Code n° III-156

```
\def\foo{a}\ifx a\foo vrai\else faux\fi
```

faux

En effet, le premier token « a » est un caractère, une sorte de primitive pour TEX alors que le second `\foo` est une macro. Notons que la primitive `\meaning` donne une bonne idée de ce qu'est la *signification* d'un token, même si elle ne dit rien sur les catcodes des tokens composant le texte de remplacement d'une macro (voir les cas n^{os} 6 et 7 ci-dessous) :

Code n° III-157

```
1) \meaning9\par% un caractère de catcode 12
2) \meaning a\par% une lettre
3) \def\foo{a}\meaning\foo\par% une macro
4) \long\def\foo{a}\meaning\foo\par% une macro déclarée \long
5) \meaning\sdlkfj\par% une macro indéfinie
6) \edef\foo{\string a}\meaning\foo\par%\foo contient un "a" de catcode 12
7) \def\foo{a}\meaning\foo\par%\foo contient un "a" de catcode 11
```

```
1) the character 9
2) the letter a
3) macro:->a
4) \long macro:->a
5) undefined
6) macro:->a
7) macro:->a
```

Pour fixer les idées, effectuons quelques autres tests qui montrent comment réagit `\ifx` :

Code n° III-158

```
a) \ifx abvrai\else faux\fi\quad% a est n'est pas égal à b
b) \ifx++vrai\else faux\fi\quad% le caractère "+" est égal à "+"
c) \ifx\relax\par vrai\else faux\fi\quad% \relax n'est pas la même primitive que \par
d) \ifx\par\par vrai\else faux\fi\quad% \par et \par sont les mêmes primitives
```

```
e) \ifx\sdfk\qlms vrai\else faux\fi\quad% 2 macros non définies sont égales
f) \def\foo{abcd}\def\bar{abc}% \foo et \bar ont des textes de remplacement différents
   \ifx\foo\bar vrai\else faux\fi\quad
g) \def\foo{abcd}\def\bar{abcd }% \foo et \bar ont des textes de remplacement différents
   \ifx\foo\bar vrai\else faux\fi\quad
h) \def\foo{abcd}\def\bar{abcd}
   \ifx\foo\bar vrai\else faux\fi\quad% \foo et \bar ont les mêmes textes de remplacement
i) \long\def\foo{a}\def\bar{a}
   \ifx\foo\bar vrai\else faux\fi\quad% \foo est \long, \bar ne l'est pas
j) \edef\foo{\string a}% \foo contient un "a" de catcode 12
   \def\bar{a}% \bar contient un "a" de catcode 11
   \ifx\foo\bar vrai\else faux\fi
```

a) faux b) vrai c) faux d) vrai e) vrai f) faux g) faux h) vrai i) faux j) faux

■ **Exercice 67**

Il a été dit que le test `\ifx` était sensible aux codes de catégorie. Écrire une macro

`\cmpmacro`⟨`\macroA`⟩⟨`\macroB`⟩{⟨*code vrai*⟩}{⟨*code faux*⟩}

qui comparera les macros contenues dans ses deux premiers arguments sans tenir compte des codes de catégorie des tokens qui composent leurs textes de remplacement.

□ **Solution**

L'idée et de détokéniser le texte de remplacement des deux macros, d'assigner ces deux textes détokénisés à des macros auxiliaires et enfin, de comparer ces deux macros avec `\ifx`. Pour ne pas laisser ces deux macros auxiliaires derrière nous, créons-les dans un groupe semi-simple que nous refermerons après la comparaison. Remarquons à cette occasion que la paire `\begingroup` et `\endgroup` peut être « imbriquée » avec les `\ifx`, `\else` et `\fi` d'un test.

Code nº III-159

```
\def\cmpmacro#1#2{%
  \begingroup
  \edef\tempa{\detokenize\expandafter{#1}}\edef\tempb{\detokenize\expandafter{#2}}%
  \ifx\tempa\tempb% si égalité
    \endgroup\expandafter\firstoftwo% ferme le groupe et lit 1er argument
  \else
    \endgroup\expandafter\secondoftwo% sinon, ferme le groupe et lit le 2e argument
  \fi
}
a) \edef\foo{\string a}\def\bar{a}
   \cmpmacro\foo\bar{vrai}{faux}\qquad
b) \edef\foo{\detokenize{$i^2=-1$\relax}}\def\bar{$i^2=-1$\relax}
   \cmpmacro\foo\bar{vrai}{faux}
```

a) vrai b) vrai

■

5.1.2. Le test `\ifx`, la `\let`-égalité et les quarks

Voici une règle particulière qu'il est utile de connaitre :

60 - Règle
Le test \ifx ne distingue pas deux tokens \let-égaux.

Autrement dit, si un token de type \⟨*macro*⟩ est rendu égal au token *x* avec \let, alors, \ifx verra \⟨*macro*⟩ et *x* égaux.

Nous pouvons l'observer ci-dessous où des tokens, à chaque fois \let-égaux, rendent tous les tests vrais :

Code n° III-160

```
a) \let\rien\relax      \ifx\rien\relax vrai\else faux\fi\qquad
b) \let\AA=a            \ifx a\AA vrai\else faux\fi\qquad
c) \let\foo=_\let\bar=_ \ifx\foo\bar vrai\else faux\fi
```

a) vrai b) vrai c) vrai

Pour exploiter la règle précédente, il est parfois commode de créer des macros spéciales dont le développement est elles-mêmes, autrement dit *invariantes* par développement ! Elles portent le nom de « quarks ». Voici comment nous pourrions définir un quark nommé \quark* :

```
\def\quark{\quark}
```

Bien évidemment, il ne faut jamais qu'un quark ne soit exécuté puisqu'en se développant, il ne change pas : cette propriété mènerait à une boucle infinie dont on ne peut sortir que par interruption manuelle. À vrai dire, les quarks ne sont utiles que pour les comparaisons via \ifx. Essayons de comprendre pourquoi en prenant par exemple une macro \test définie comme suit :

```
\def\test{\quark}
```

Dans ce cas, le test « \ifx\test\quark » est vrai parce que \quark est un quark. Il serait faux s'il ne l'était pas. En effet, \ifx compare le texte de replacement de \test et celui de \quark : ce texte de remplacement est bien \quark dans les deux cas.

Le test est également positif si \test est défini par

```
\let\test=\quark
```

puisque dans ce cas, deux macros \let-égales sont comparées.

Code n° III-161

```
\def\quark{\quark}% définit un quark
1) \def\test{\quark} \ifx\quark\test vrai\else faux\fi
\qquad
2) \let\test=\quark  \ifx\quark\test vrai\else faux\fi
```

1) vrai 2) vrai

Le fait que \ifx donne un test vrai aussi bien pour une macro définie par \def ou par \let se révèle bien pratique et constitue une particularité des quarks.

61 - Définition et propriétés
Un quark est une macro dont le texte de remplacement est constitué de la macro elle-même.

Un quark ne doit *jamais* être exécuté, car étant invariant par développement, son traitement par TEX engendrerait une boucle infinie.

Si \⟨*quark*⟩ est un quark et si \⟨*macro*⟩ est définie par

\def\⟨*macro*⟩{\⟨*quark*⟩} ou par \let\⟨*macro*⟩ = \⟨*quark*⟩

alors, le test \ifx\⟨*macro*⟩\⟨*quark*⟩ sera vrai dans les deux cas.

5.2. Applications du test \ifx

5.2.1. Tester si un argument est vide

Il arrive souvent qu'il faille tester si un argument est vide (c'est-à-dire constitué de 0 token). La méthode naïve, mais aussi la plus efficace est la suivante : on assigne avec \def cet argument à une \⟨*macro*⟩ temporaire. Ensuite, on compare avec \ifx cette \⟨*macro*⟩ et la macro nommée \empty, définie dans plain-TEX [1], dont le texte de remplacement est vide :

\def\empty{}

Cela donne le code suivant :

Code n° III-162

```
\def\ifempty#1{%
  \def\tempa{#1}% stocke l'argument #1 dans \tempa
  \ifx\tempa\empty% si \tempa = \empty
    \expandafter\firstoftwo% 1er argument
  \else
    \expandafter\secondoftwo% sinon, second
  \fi
}
a) \ifempty{abc}{vide}{pas vide}\qquad
b) \ifempty{}{vide}{pas vide}\qquad
c) \ifempty{ }{vide}{pas vide}\qquad
d) \ifempty{\empty}{vide}{pas vide}
```

a) pas vide b) vide c) pas vide d) pas vide

■ **Exercice 68**

Expliquer pourquoi le test est négatif à la ligne n° 12 du code ci-dessous

□ **Solution**

La ligne n° 2 effectue l'action suivante :

\def\tempa{\empty}

Le test \ifx\tempa\empty compare donc les deux textes de remplacement de ces macros qui sont \empty pour la première et un texte vide pour la seconde. Ces deux textes de remplacement ne sont pas égaux ce qui explique que le test est négatif. ■

1. La macro équivalente \@empty est définie dans le format LATEX.

La macro \ifempty fonctionne parfaitement, mais elle n'est *pas* purement développable à cause de la présence du \def dans son texte de remplacement. Pour rendre la macro développable, l'astuce consiste à utiliser le fait que lorsqu'un argument #1 d'une macro est vide, tout se passe comme s'il n'existait pas dans son texte de remplacement. Dans ce cas, les tokens qui sont juste avant et juste après #1 deviennent consécutifs. Il suffit que ces deux tokens soient les mêmes (par exemple « _ ») pour que le test « \ifx_#1_ » devienne positif lorsque #1 est vide :

Code n° III-163

```
\def\empty{}
\long\def\ifempty#1{%
  \ifx_#1_\expandafter\firstoftwo
  \else\expandafter\secondoftwo
  \fi}
a) \ifempty{abc}{vide}{pas vide}\qquad
b) \ifempty{}{vide}{pas vide}\qquad
c) \ifempty{ }{vide}{pas vide}\qquad
d) \ifempty{\empty}{vide}{pas vide}\qquad
e) \edef\foo{\ifempty{abc}{vide}{pas vide}}\meaning\foo
```

a) pas vide b) vide c) pas vide d) pas vide e) macro:->pas vide

Bien évidemment, au lieu de « _ », nous aurions aussi pu utiliser « | », \empty ou tout autre token. La macro fonctionne bien, mais il va se trouver des cas particuliers où elle va donner des faux positifs ! En effet, le test \ifx⟨*token*⟩#1⟨*token*⟩ est vrai si #1 est vide, mais il sera *aussi* vrai si #1 commence par le ⟨*token*⟩. Nous pouvons l'observer ici où « W » est pris comme token de comparaison qui, contrairement à « _ » peut être *affiché* par TEX. Le cas n° 4 met en évidence le problème :

Code n° III-164

```
\long\def\ifempty#1{%
  \ifx W#1W\expandafter\firstoftwo
  \else    \expandafter\secondoftwo
  \fi}
1) \ifempty{foobar}{vide}{pas vide}\qquad
2) \ifempty{}{vide}{pas vide}\qquad
2) \ifempty{\empty}{vide}{pas vide}\qquad
4) \ifempty{Wagons}{vide}{pas vide}
```

1) pas vide 2) vide 2) pas vide 4) agonsWvide

On peut observer que le test \ifx a comparé le « W » qui le suit et le « W » de « Wagons ». Comme il est vrai, tout ce qui est jusqu'au \else a été affiché.

Pour se prémunir de cet inconvénient, la méthode la plus communément utilisée consiste à *détokéniser* l'argument #1 et l'entourer de deux tokens qui ne sont pas de catcode 12 (\relax est habituellement utilisé dans ce cas). Pour que cela fonctionne, « \detokenize{#1} » doit être développé avant que le test n'entre en jeu et nécessite donc un pont d'\expandafter :

```
\expandafter\ifx\expandafter\relax\detokenize{#1}\relax
```

Avec cette méthode, le test n'est positif *que* lorsque #1 est vide. En effet,

1. si #1 commence par un espace, \detokenize{#1} se développe en un espace (de catcode 10) suivi d'autres tokens. Le test se fera donc entre \relax et ␣ et sera faux ;

2. si #1 commence par un autre token, le développement de \detokenize{#1} commence par un caractère de catcode 12. La comparaison entre \relax et ce token est fausse ;
3. enfin, si #1 est vide, \detokenize{} se développe en 0 token et donc, les primitives \relax deviennent consécutives et sont vues égales par le test \ifx qui est alors positif.

Code n° III-165

```
\long\def\ifempty#1{%
  \expandafter\ifx\expandafter\relax\detokenize{#1}\relax
    \expandafter\firstoftwo
  \else
    \expandafter\secondoftwo
  \fi
}
a) \ifempty{abc}{vide}{pas vide}\qquad
b) \ifempty{}{vide}{pas vide}\qquad
c) \ifempty{ }{vide}{pas vide}\qquad
d) \ifempty{\relax}{vide}{pas vide}\qquad
e) \edef\foo{\ifempty{abc}{vide}{pas vide}}\meaning\foo
```

a) pas vide b) vide c) pas vide d) pas vide e) macro:->pas vide

Utiliser \detokenize est la seule méthode développable qui soit vraiment infaillible, mais elle a le désavantage d'être un peu lourde à écrire, c'est pourquoi on lui préfère souvent

```
\ifx\empty#1\empty
```

tout en restant conscient des faux positifs qu'elle peut engendrer dans les cas – le plus souvent improbables – où #1 commence par la macro \empty ou toute autre macro \let-égale à \empty.

■ Exercice 69

Programmer une macro \reverse*, admettant comme argument un ⟨*mot*⟩ et dont le développement maximal est le ⟨*mot*⟩ écrit dans l'ordre inverse.

Par exemple, \reverse{foobar} donne « raboof ».

□ Solution

Pour y parvenir, il faut imaginer deux réservoirs, A et B recevant des caractères. L'algorithme est le suivant :

1) mettre le ⟨*mot*⟩ dans le réservoir A et initialiser le réservoir B à ⟨*vide*⟩ ;
2) prendre le premier caractère du réservoir A et le mettre en *première* position dans le réservoir B ;
3) si le réservoir A n'est pas vide, retourner au point 2.

Avec des arguments délimités, il est très facile de construire de tels réservoirs tout en gardant la macro purement développable. Décidons pour l'instant que ces délimiteurs sont des points. La structure des arguments sera donc :

⟨*réservoir A*⟩.⟨*réservoir B*⟩.

Voici ce que sont les étapes pour inverser le texte « abcd » :

abcd.. ⟶ bcd.a. ⟶ cd.ba. ⟶ d.cba. ⟶ .dcba.

Le choix du point interdit que le ⟨*mot*⟩ en contienne, ce qui n'est pas absurde en soi. Cependant, pour plus de sécurité, nous prendrons `\@nil` comme délimiteur. Le code de la macro `\reverse` ne présente aucune difficulté particulière :

Code n° III-166

```
\catcode`\@11
\def\reverse#1{%
  \ifempty{#1}
    {}% s'il n'y a rien à inverse, ne rien faire
    {\reverse@i#1\@nil\@nil}% initialisation des réservoirs et appeler \reverse@i
}
\def\reverse@i#1#2\@nil#3\@nil{% #1 est le 1er caractère du réservoir A
  \ifempty{#2}% si ce qui reste après ce 1er caractère est vide
    {#1#3}% renvoyer #1#3 qui est le résultat final
    {\reverse@i#2\@nil#1#3\@nil}% sinon, recommencer en déplaçant #1
                                % en 1re position du réservoir B
}
\catcode`\@12
a) \reverse{foobar}\qquad
b) \edef\foo{\reverse{Bonjour}}\meaning\foo\qquad
c) \reverse{Bonjour le monde}\qquad
d) \reverse{Bonjour{ }le{ }monde}
```

a) raboof b) macro:->ruojnoB c) ednomelruojnoB d) ednom el ruojnoB

Selon la règle de la page 64, les espaces qui ne sont pas entre accolades sont ignorés lorsque lus comme argument par une macro. Il est donc normal que « Bonjour le monde » donne un résultat inversé sans espace. ■

Le simple fait pour pouvoir tester de façon fiable si un argument est vide ouvre déjà la voie à d'autres tests…

5.2.2. Tester si un code contient un motif

Le but est d'élaborer un test qui vérifie si un ⟨*code*⟩ contient un autre code que l'on appelle ⟨*motif*⟩. Nous allons tenter d'écrire une macro `\ifin` dont voici la syntaxe :

`\ifin{`⟨*code*⟩`}{`⟨*motif*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}`

Le code ⟨*vrai*⟩ sera exécuté si ⟨*code*⟩ contient ⟨*motif*⟩ et le code ⟨*faux*⟩ le sera dans le cas contraire.

Comme nous l'avions fait avec les macros `\rightof` et `\leftof` du chapitre 3.2.1 et 3.2.3 de la partie 2, nous allons utiliser une macro auxiliaire à arguments délimités. Cette macro `\ifin@i` sera située à l'intérieur du texte de remplacement de `\ifin` de telle sorte qu'elle puisse accéder aux arguments de `\ifin`. Elle aura le texte de paramètre suivant :

```
\def\ifin@i##1#2##2\@nil{code à définir}
```

Ainsi, les arguments délimités `##1` et `##2` collectent ce qui se trouve respectivement avant et après le ⟨*motif*⟩ qui est l'argument `#2` de la macro chapeau.

L'idée est que la macro `\ifin` appelle la macro `\ifin@i` de cette façon :

```
\ifin@i#1#2\@nil
```

où le ⟨*motif*⟩ (argument #2) est ajouté après le ⟨*code*⟩ (argument #1) afin de satisfaire la macro à argument délimité \ifin@i même dans le cas où #1 ne contient pas #2. Avec cet appel, de deux choses l'une :

- soit #1 contient déjà #2 et donc la première occurrence de #2 est dans #1 et nous pouvons donc affirmer que ce qui se trouve après cette première occurrence (qui est l'argument ##2 de \ifin@i) n'est pas vide puisqu'au moins constitué de #2 que nous avons rajouté ;
- soit #1 ne contient pas #2, ce qui fait que la première occurrence de #2 est le #2 que nous avons ajouté juste devant le \@nil. Par conséquent, ce qui se trouve après cette occurrence est vide.

Il suffit donc de tester si ce qui est après la première occurrence de #2 est vide ou pas, ce que fait la ligne n° 4 du code ci-dessous :

Code n° III-167

```
\catcode`\@11
\def\ifin#1#2{%
  \def\ifin@i##1#2##2\@nil{% définit la macro auxiliaire
    \ifempty{##2}% si ce qu'il y a derrière le motif est vide
      \secondoftwo% aller à "faux"
      \firstoftwo% sinon à "vrai"
  }%
  \ifin@i#1#2\@nil% appel de la macro auxiliaire
}
\catcode`\@12
a) \ifin{abc\relax1}{bc}{vrai}{faux}\qquad
b) \ifin{abc \relax1}{c\relax}{vrai}{faux}\qquad
c) \ifin{abc \relax1}{ }{vrai}{faux}\qquad
d) \ifin{abc \relax1}{}{vrai}{faux}\qquad
e) \ifin{}{a}{vrai}{faux}
```

a) vrai b) faux c) vrai d) vrai e) faux

Le gros inconvénient des arguments délimités est le défaut que nous avions signalé avec la macro \rightof à la page 97. Le ⟨*motif*⟩ ne peut pas contenir d'accolades ouvrantes puisqu'elles se retrouveraient dans le texte de paramètre de \ifin@i et en fausseraient la définition. Nous allons donc nous limiter à un code et un motif sans accolades… Avant de trouver mieux !

■ Exercice 70

Écrire une macro \ifstart* de syntaxe

\ifstart{⟨*code*⟩}{⟨*motif*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

qui teste si le ⟨*code*⟩ commence par le ⟨*motif*⟩ et exécute ⟨*vrai*⟩ si c'est le cas et ⟨*faux*⟩ sinon.

□ Solution

La méthode est très similaire à celle déployée dans \ifin. Dans l'homologue de \ifin@i qui est \ifstart@i, il suffira de tester si ce qui est *avant* la première occurrence du motif (c'est-à-dire ##1) est vide ou pas. Ce qui donne ce code :

Code n° III-168

```
\catcode`\@11
\def\ifstart#1#2{%
  \def\ifstart@i##1#2##2\@nil{\ifempty{##1}}%
```

```
  \ifstart@i#1#2\@nil
}
\catcode`\@12
a) \ifstart{abc}{bc}{vrai}{faux}\qquad
b) \ifstart{abc}{ab}{vrai}{faux}\qquad
c) \ifstart{ abc}{ }{vrai}{faux}\qquad
d) \ifstart{abc}{}{vrai}{faux}\qquad
e) \ifstart{}{a}{vrai}{faux}
```

a) faux b) vrai c) vrai d) faux e) vrai

Les deux derniers cas sont problématiques. Il serait logique que l'avant-dernier soit vrai puisque « abc » commence bien par un argument vide. Pour le dernier au contraire, il faudrait qu'il soit faux, car un argument vide ne contient pas le motif « a ».

Pour faire en sorte qu'un ⟨*motif*⟩ vide donne toujours le code ⟨*vrai*⟩, il suffit à la ligne nº 4 de tester si #2 est vide et exécuter `\firstoftwo` si c'est le cas et appeler `\ifstart@i` sinon :

```
\ifempty{#2}\firstoftwo{\ifstart@i#1#2\@nil}
```

Si le ⟨*code*⟩ est vide, l'appel à `\ifstart@i` est `\ifstart@i#2\@nil` et donc, l'argument délimité `##1` de `\ifstart@i` est logiquement vide. On peut artificiellement le remplir en mettant un `\relax` juste avant le #2 de la ligne 4. Cela donne le code :

Code nº III-169

```
\catcode`\@11
\def\ifstart#1#2{%
  \def\ifstart@i##1#2##2\@nil{\ifempty{##1}}%
  \ifempty{#2}% si motif vide
    \firstoftwo% exécuter code "vrai"
    {\ifstart@i#1\relax#2\@nil}% sinon, aller à la macro auxiliaire
}
\catcode`\@12
a) \ifstart{abc}{bc}{vrai}{faux}\qquad
b) \ifstart{abc}{ab}{vrai}{faux}\qquad
c) \ifstart{ abc}{ }{vrai}{faux}\qquad
d) \ifstart{abc}{}{vrai}{faux}\qquad
e) \ifstart{}{a}{vrai}{faux}
```

a) faux b) vrai c) vrai d) vrai e) faux

■

5.2.3. Tester si un code se termine par un motif

Il est bien plus difficile de tester si un ⟨*code*⟩ se termine par un ⟨*motif*⟩. Cela est dû à la façon qu'ont de fonctionner les arguments délimités qui ne s'intéressent qu'à la *première* occurrence d'un délimiteur et non pas à la dernière. Si nous souhaitons écrire une macro `\ifend*` dont la syntaxe est

```
\ifend{⟨code⟩}{⟨motif⟩}{⟨vrai⟩}{⟨faux⟩}
```

nous allons devoir recourir à un algorithme récursif.

1) ajouter le ⟨*motif*⟩ au début du ⟨*code*⟩ et dans le résultat obtenu,
2) appeler ⟨*code*⟩ ce qui est après la première occurrence du ⟨*motif*⟩ ;
3) si ⟨*code*⟩ est vide exécuter `vrai` ;
4) sinon

a) si ⟨*code*⟩ contient le ⟨*motif*⟩, retourner en 2 ;
b) sinon, exécuter faux.

Le point n° 1 sera l'appel initial à une macro auxiliaire récursive `\ifend@i` à argument délimité. Cette macro occupera les points suivants (de 2 à 4).

Traduisons cet algorithme en code TEX et vérifions qu'il fonctionne sur quelques exemples. Le point 1 sera codé de cette façon :

```
\ifend@i#2#1\@nil
```

La macro auxiliaire `\ifend@i` va se charger, grâce à son argument délimité, d'isoler ce qui se trouve après la première occurrence de ⟨*motif*⟩. Voici ce que sera son texte de paramètre :

```
\def\if@end##1#2##2\@nil{...}
```

À l'intérieur de la macro `\if@end`, il suffira de tester si `##2` est vide auquel cas, `\firstoftwo` sera exécuté. Dans le cas contraire, il faudra tester avec `\ifin` si le `##2` contient le ⟨*motif*⟩ qui est `#2` et appeler récursivement `\ifend@i` dans l'affirmative et exécuter `\secondoftwo` sinon :

Code n° III-170

```
\catcode`\@11
\def\ifend#1#2{%
  \def\ifend@i##1#2##2\@nil{% ##2 = ce qui reste après le motif
    \ifempty{##2}% si ##2 est vide
      \firstoftwo% exécuter l'argument "vrai"
      {\ifin{##2}{#2}% sinon, si ##2 contient le <motif>
        {\ifend@i##2\@nil}% appeler \ifend@i avec ce qui reste
        \secondoftwo% sinon, exécuter l'argument "faux"
      }%
  }%
  \ifend@i#2#1\@nil% appelle la macro récursive
}
\catcode`\@12
1) \ifend{abc de}{de}{vrai}{faux}\qquad
2) \ifend{abd de }{de}{vrai}{faux}\qquad
3) \ifend{abc de }{ }{vrai}{faux}\qquad
4) \ifend{}{a}{vrai}{faux}\qquad
5) \ifend{abc de}{}{vrai}{faux}
```

1) vrai 2) faux 3) vrai 4) vrai 5) faux

■ Exercice 71

Expliquer pourquoi le code ci-dessus renvoie le contraire de ce que l'on attend aux cas n^{os} 4 et 5 et proposer des solutions pour y remédier.

□ Solution

Dans le cas où le ⟨*code*⟩ est vide (cas n° 4), l'appel de la ligne n° 11 devient

```
\ifend@i a\@nil
```

et l'argument `##2` de `\ifend@i` qui se trouve après la première occurrence de « a » est vide. Pour s'en prémunir, nous pouvons artificiellement meubler avec `\relax` ce qui se trouve après cette première occurrence. L'appel à la ligne n° 11 deviendrait donc :

```
\ifend@i#2\relax#1\@nil
```

Pour le cas n° 5, le ⟨*motif*⟩ est vide, l'appel de la ligne n° 11 est

```
\ifend@i abc de\@nil
```

et le texte de paramètre de `\ifend@i` est « `\def\ifend@i##1##2\@nil{...}` » ce qui signifie que l'argument `##1` n'est plus un argument délimité : il devient le premier argument (ici « a ») et `##2` est le reste jusqu'au `\@nil` (c'est-à-dire « bc de »). Continuons à faire fonctionner la macro mentalement : le test `\ifempty{##2}` est faux, le test suivant `\ifin{##2}{#2}` est faux aussi puisque `##2` est vide : la macro `\secondoftwo` va donc exécuter le code `faux`.

Le meilleur remède est sans doute de tester dès le premier appel (ligne n° 11) si `#2` est vide, cas où `\firstoftwo` doit être exécuté.

Code n° III-171

```
\catcode`\@11
\def\ifend#1#2{%
  \def\ifend@i##1#2##2\@nil{% ##2 = ce qui reste après le motif
    \ifempty{##2}% si ##2 est vide
      \firstoftwo% exécuter l'argument "vrai"
      {\ifin{##2}{#2}% sinon, si ##2 contient le <motif>
        {\ifend@i##2\@nil}% appeler \ifend@i avec ce qui reste
        \secondoftwo% sinon, exécuter l'argument "faux"
      }%
  }%
  \ifempty{#2}% si le motif est vide
    \firstoftwo% exécuter "vrai"
    {\ifend@i#2\relax#1\@nil}% sinon, appelle la macro récursive
}
\catcode`\@12
1) \ifend{abc de}{de}{vrai}{faux}\qquad
2) \ifend{abd de }{de}{vrai}{faux}\qquad
3) \ifend{abc de }{ }{vrai}{faux}\qquad
4) \ifend{}{a}{vrai}{faux}\qquad
5) \ifend{abc de}{}{vrai}{faux}
```

1) vrai 2) faux 3) vrai 4) faux 5) vrai

■

5.2.4. Remplacer un motif par un autre

Ns allons maintenant construire une macro `\substin`⋆, capable de remplacer un ⟨*motif1*⟩ par un ⟨*motif2*⟩ dans un ⟨*code*⟩ :

`\substin{`⟨*code*⟩`}{`⟨*motif1*⟩`}{`⟨*motif2*⟩`}`

La méthode va consister à parcourir le ⟨*code*⟩ un peu de la même façon que nous le faisions avec la macro `\ifend`. Voici l'algorithme :

1) si le ⟨*code*⟩ est vide, ne rien faire et fin du processus ;
2) sinon
   a) si le ⟨*code*⟩ contient le ⟨*motif1*⟩, aller au point n° 3 ;
   b) sinon afficher le ⟨*code*⟩ et fin du processus ;
3) afficher la partie du ⟨*code*⟩ qui se trouve avant la 1re occurrence de ⟨*motif1*⟩ ;
4) afficher ⟨*motif2*⟩ ;
5) retourner au point n° 1 en appelant ⟨*code*⟩ ce qui se trouve après ⟨*motif1*⟩.

Les points nos 1 et 2 vont être effectués par une macro auxiliaire dont le seul argument non délimité sera le ⟨*code*⟩ en cours d'examen. Les points nos 3 à 5 seront

dévolus à une autre macro auxiliaire qui elle, sera à argument délimité puisqu'il faut séparer ce qui se trouve avant et après le ⟨*motif1*⟩. Voici ce que cela donne en pseudocode :

```
———————— Remplacer toutes les occurrences d'un motif ————————

macro substin#1#2#3% #1=<code>, #2=<motif1>, #3=<motif2>
    macro substin@i##1
        si ##1 est vide
            relax% ne rien faire
        sinon
            si ##1 contient #3
                substin@ii##1@nil% appeler la macro à argument délimités
            sinon
                ##1% afficher le <code>
            finsi
        finsi
    fin

    macro substin@ii##1#2##2@nil% ##1 et ##2=ce qui est avant/après <motif1>
        afficher "##1#3"
        appeler substin@i(##2)% puis recommencer avec le code qui reste
    fin

    appeler substin@i(#1)% engage le processus
fin
```

La traduction de ce pseudocode en TEX ne pose pas de problème particulier :

Code nº III-172

```
\catcode`\@11
\def\substin#1#2#3{%
  \def\substin@i##1{%
    \ifempty{##1}% si le <code> est vide
      \relax% ne rien faire -> fin du processus
      {\ifin{##1}{#2}% sinon, si le <code> contient <motif1>
        {\substin@ii##1\@nil}% appeler la macro à argument délimités
        {##1}% sinon, afficher le <code>
      }%
  }%
  \def\substin@ii##1#2##2\@nil{%
    ##1#3% afficher ##1 et #3 (qui est <motif2>)
    \substin@i{##2}% et recommencer avec ce qui reste
  }%
  \substin@i{#1}%
}
\catcode`\@12
a) \substin{abracadabra}{a}{A}\qquad
b) \substin{abracadabra}{x}{W}\qquad
c) \substin{abracadabra}{br}{}\qquad
d) \substin{1\relax3}\relax{222}\qquad
```

a) AbrAcAdAbrA b) abracadabra c) aacadaa d) 12223

■ Exercice 72

Modifier cette macro en une macro `\substtocs*` pour qu'elle n'affiche pas le résultat, mais le mette dans le texte de remplacement d'une `\⟨macro⟩` dont l'utilisateur pourra choisir le nom :

`\substtocs\⟨macro⟩{⟨code⟩}{motif1}{motif2}`

□ **Solution**

Au lieu d'afficher des morceaux de code, il faudra les ajouter au texte de remplacement d'une macro chargée de collecter le code résultant de ces substitutions. Pour ajouter du code au texte de remplacement d'une macro, `\addtomacro`, programmée à la page 107 sera utile.

Code n° III-173

```
\catcode`\@11
\def\substtocs#1#2#3#4{%
  \def\substtocs@i##1{%
    \ifempty{##1}% si le <code> est vide
      \relax% ne rien faire -> fin du processus
      {\ifin{##1}{#3}% sinon, si le <code> contient <motif1>
        {\substtocs@ii##1\@nil}% appeler la macro à argument délimités
        {\addtomacro#1{##1}}% sinon, ajouter le <code>
      }%
  }%
  \def\substtocs@ii##1#3##2\@nil{%
    \addtomacro#1{##1#4}% ajouter ##1#4
    \substtocs@i{##2}% et recommencer avec ce qui reste
  }%
  \let#1=\empty% initialise la macro à vide
  \substtocs@i{#2}%
}
\catcode`\@12
\substtocs\foo{abracadabra}{a}{A}\meaning\foo\par
\substtocs\foo{abracadabra}{x}{W}\meaning\foo\par
\substtocs\foo{abracadabra}{br}{}\meaning\foo\par
\substtocs\foo{1\relax3}\relax{222}\meaning\foo\par
\substtocs\foo{\ifnum1=2 test vrai\fi}{2}{1}\meaning\foo
```

macro:->AbrAcAdAbrA
macro:->abracadabra
macro:->aacadaa
macro:->12223
macro:->\ifnum 1=1 test vrai\fi

■

■ **Exercice 73**

Écrire une macro `\majmot*{`⟨*phrase*⟩`}` qui met en majuscule tous les mots de la phrase.

□ **Solution**

Pour que `\majmot` effectue son travail, dans la ⟨*phrase*⟩, chaque espace doit être remplacé par « ␣`\majuscule` », où la macro `\majuscule*` lit un argument (qui sera la lettre du mot suivant) et la met en majuscule. En coulisses, nous utiliserons la primitive `\uppercase`.

Nous allons mettre `\substin` à contribution pour effectuer ces remplacements, en ayant pris soin de mettre `\majuscule` tout au début de la phrase afin de traiter la première lettre du premier mot.

Code n° III-174

```
\def\majuscule#1{\uppercase{#1}}
\def\majmot#1{\substin{\majuscule#1}{ }{ \majuscule}}
\majmot{un petit texte}\par
\majmot{Un grand texte sans importance}
```

Un petit texte
Un grand texte sans importance

Conclusion : ça ne fonctionne pas du tout !

Une petite analyse montre que tout vient de la façon dont est faite la substitution et il faut revenir au code de \substin :

```
1     \def\substin#1#2#3{%
2       \def\substin@i##1{%
2         \ifempty{##1}% si le code est vide
2           \relax% ne rien faire -> fin du processus
2           {\ifin{##1}{#2}% sinon, si le code contient motif1
2             {\substin@ii##1\@nil}% appeler la macro à argument délimités
2             {##1}% sinon, afficher le code
2           }%
2       }%
2       \def\substin@ii##1#2##2\@nil{%
2         ##1#3% afficher ##1 et #3 (qui est <motif2>)
2         \substin@i{##2}% et recommencer avec ce qui reste
2       }%
2       \substin@i{#1}%
2     }
```

Constatons à ligne nº 11 que le code #2 est substitué par le motif #3 donc, ##1#3 est exécuté par TEX dans la foulée. C'est de cette exécution *au fur et à mesure* que vient le problème. Ici en effet, le motif de substitution #3 est « ␣\majsucule ». Notre processus échoue parce que lors de l'exécution au fur et à mesure, la macro \majuscule va lire son argument qui est \substin@i de la ligne nº 12. Cet argument sera traité par \uppercase qui le laisse intact puisque \uppercase n'a aucun effet sur les séquences de contrôle. Le code ci-dessus ne produit donc aucun effet.

Nous allons nous en sortir en utilisant la macro \substtocs de façon à stocker le code obtenu dans une macro temporaire \temp que nous afficherons à la toute fin du processus, avant de sortir du groupe semi-simple dans lequel elle a été créée :

Code nº III-175

```
\def\majuscule#1{\uppercase{#1}}
\def\majmot#1{%
  \begingroup
    \substtocs\temp{\majuscule#1}{ }{ \majuscule}%
    \temp% exécute la macro temporaire
  \endgroup
}
\majmot{un petit texte}\par
\majmot{Un grand texte Sans importance}
```

Un Petit Texte
Un Grand Texte Sans Importance

La leçon à retenir ici est que faire exécuter par TEX de morceaux de code dans une macro récursive n'est pas équivalent à les mettre tous dans une séquence de contrôle que l'on exécute en une fois à la fin. ■

■ Exercice 74

Écrire une macro \cnttimes* {⟨*code*⟩}{⟨*motif*⟩} qui affiche combien de fois le ⟨*motif*⟩ est contenu dans le ⟨*code*⟩.

□ Solution

Nous allons procéder comme dans \substtocs. La macro chapeau initialisera un compteur à 0 et passera la main à une macro auxiliaire, chargée de l'incrémenter à chaque fois que le motif sera rencontré :

Code n° III-176

```
\catcode`\@11
\newcount\cnt@occ
\def\cnttimes#1#2{%
  \def\cnttimes@i##1{%
    \ifempty{##1}% si le <code> est vide
      {\number\cnt@occ}% afficher le nombre d'occurrences
      {\ifin{##1}{#2}% sinon, si le <code> contient <motif1>
        {\cnttimes@ii##1\@nil}% appeler la macro à argument délimités
        {\number\cnt@occ}% sinon, afficher le nombre d'occurrences
      }%
  }%
  \def\cnttimes@ii##1#2##2\@nil{%
    \advance\cnt@occ1
    \cnttimes@i{##2}% et recommencer avec ce qui reste
  }%
  \cnt@occ=0 % initialise le compteur
  \cnttimes@i{#1}%
}
\catcode`\@12
a) \cnttimes{abracadabra}{a}\qquad
b) \cnttimes{abracadabra}{ra}\qquad
c) \cnttimes{abracadabra}{w}\qquad
d) \cnttimes{abc def ghijk l }{ }\qquad
e) \cnttimes{\ifnum1=2 vrai\else faux\fi}{\ifnum}
```

a) 5 b) 2 c) 0 d) 4 e) 1

Juste pour le plaisir de se compliquer la vie, essayons de modifier la macro en `\cnttimestocs*` pour qu'elle réponde aux contraintes suivantes :

1. ne pas utiliser de compteur ;
2. n'écrire qu'une seule macro intérieure au lieu de deux ;
3. stocker le résultat dans une macro spécifiée par l'utilisateur.

Si nous nous interdisons l'utilisation d'un compteur, nous devrons recourir à la primitive `\numexpr` pour remplacer `\advance` afin d'incrémenter le nombre d'itérations. Il est évidemment possible de stocker le nombre d'itérations dans une séquence de contrôle, mais dans ce cas, se priver d'un compteur pour basculer vers une macro n'a pas vraiment de sens, autant conserver un compteur. Il est plus original de stocker le nombre d'itérations dans un argument qui s'ajoutera à ceux de la macro récursive `\cnttimes@i` vue dans le code précédent. Pour pouvoir facilement incrémenter cet argument (qui implique le 1-développer) avant de lancer l'appel récursif, il est judicieux de le placer comme premier argument. L'idée la plus pratique est donc d'ajouter cet argument (et de le délimiter par un `\@nil`) avant ceux de `\cnttimes@ii`. La macro récursive `\cnttimestocs@i` aura donc le texte de paramètre suivant :

```
\def\cnttimestocs@i##1\@nil##2#2##3\@nil{...}
```

Il faudra spécifier « `0` » comme argument `##1` lors du premier appel.

Ensuite, comme cette macro récursive à est à argument délimité par le ⟨*motif*⟩, elle *doit* recevoir un ⟨*code*⟩ qui contient ce ⟨*motif*⟩. Il est donc nécessaire de tester la présence de ce ⟨*motif*⟩ lors du premier appel et également à l'intérieur de la macro récursive avant qu'elle ne s'appelle elle-même avec le ⟨*code*⟩ restant.

Code n° III-177

```
\catcode`\@11
\long\def\cnttimestocs#1#2#3{% #3=macro recevant le résultat
  \long\def\cnttimestocs@i##1\@nil##2#2##3\@nil{%
```

```
    % ##1=nb d'occurrences rencontrées jusqu'alors
    % ##2 et ##3=ce qui est avant/après le <motif>
    \ifin{##3}{#2}% si le <motif> est dans ce qui reste
      {\expandafter\cnttimestocs@i% appeler la macro récursive
        \number\numexpr##1+1\relax\@nil% avec une occurrence de plus
        ##3\@nil% et ce qui reste
      }%
      {\edef#3{\number\numexpr##1+1\relax}}% sinon, stocker 1 occurrence de plus dans #3
  }%
  \ifin{#1}{#2}% si le <motif> est dans le <code>
    {\cnttimestocs@i 0\@nil#1\@nil}% appeler la macro récursive avec 0 occurrence
    {\def#3{0}}% sinon, mettre 0 dans #3
}
\catcode`\@12
a) \cnttimestocs{abracadabra}{a}\foo \meaning\foo\qquad
b) \cnttimestocs{abracadabra}{ra}\foo \meaning\foo\qquad
c) \cnttimestocs{abracadabra}{w}\foo \meaning\foo\qquad
d) \cnttimestocs{abc def ghijk l }{ }\foo \meaning\foo\qquad
e) \cnttimestocs{\ifnum1=2 vrai\else faux\fi}{\ifnum}\foo \meaning\foo
```

a) macro:->5 b) macro:->2 c) macro:->0 d) macro:->4 e) macro:->1

■

5.2.5. Substitutions successives

Nous allons maintenant écrire une macro `\multisubst*`, généralisation de la macro `\substin` où nous pourrons spécifier plusieurs substitutions successives :

`\multisubst{`⟨*code*⟩`}{{`⟨*motif1*⟩`}{`⟨*motif2*⟩`}{`⟨*motif3*⟩`}{`⟨*motif4*⟩`}{`⟨*motif5*⟩`}...}`

Cette macro substituera toutes les occurrences de ⟨*motif1*⟩ par ⟨*motif2*⟩ puis toutes celles de ⟨*motif3*⟩ par ⟨*motif4*⟩ et ainsi de suite jusqu'à ce que la liste des substitutions soit vide dans le deuxième argument. Les motifs utilisés dans la substitution seront donc lus par *paires*. L'idée est de se servir de la macro `\substin` précédemment écrite en lui envoyant comme second et troisième argument les motifs de substitution pris deux par deux dans la liste. Comme un espace non entouré d'accolades est ignoré en tant qu'argument, les motifs peuvent être groupés par paires *séparées par un espace* dans la liste des motifs afin d'en améliorer la lisibilité.

Code n° III-178

```
\catcode`\@11
\def\multisubst#1#2{%
  \def\subst@code{#1}% stocke le <code>
  \multisubst@i#2\@nil% appelle la macro récursive avec la liste de motifs
}

\def\multisubst@i#1#2#3\@nil{% #1 et #2=paire de motifs  #3=motifs restants
  \expandafter\substtocs\expandafter\subst@code\expandafter% 1-développe
    {\subst@code}{#1}{#2}% le <code> et effectue la substitution en cours
  \ifempty{#3}% si la liste des motifs est vide
    \subst@code% exécuter le <code> obtenu
    {\multisubst@i#3\@nil}% recommencer avec les motifs qui restent
}
\catcode`\@12
1) \multisubst{abracadabra}{aA rR}\par
2) \multisubst{Ce texte devenu \`a peine reconnaissable montre que le
```

```
17   r\'esultat contient des sonorit\'es catalanes, corses ou grecques
18   assez inattendues.}{a{AA} ya uy ou io ei {AA}e}
```

1) AbRAcAdAbRA
2) Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís cetelenis, cursis uy gricqyis essiz onettindyis.

La macro chapeau `\multisubst` de la ligne n° 2 se contente de lire le ⟨*code*⟩ et l'assigne à la macro `\subst@code` avant de passer la main à la véritable macro récursive `\multisubt@i`, à argument délimité. La liste des motifs de substitutions est passée comme argument pour s'étendre jusqu'au `\@nil`. Les arguments non délimités `#1` et `#2` deviennent les deux motifs de substitution de l'itération en cours tandis que `#3` est la liste des paires restantes.

Le cas n° 2 des lignes n°s 16 à 18 traite d'une façon différente le problème de la permutation circulaire des voyelles déjà résolu à la page 56. On ne touche plus à aucun code de catégorie, mais on effectue tour à tour les substitutions désirées. La méthode est cependant *lente*, car l'argument est ici un texte assez long et celui-ci sera parcouru par `\multisubst` autant de fois qu'il y a de substitutions à faire, c'est-à-dire 7 fois. La « propreté » que l'on gagne à ne pas modifier les codes de catégorie se paie en lenteur.

Au-delà ce ce défaut lié à la méthode utilisée, le code présente une autre redondance : chaque paire est lue *plusieurs fois*. Elle est lue une première fois par la macro chapeau `\multisubst` puisqu'elle lit toutes les paires. Par la suite, à chaque itération, la macro récursive `\multisubst@i` lit (et mange) la première paire de son argument, mais lit *aussi que la totalité des paires restantes*. La totalité des paires est donc lue deux fois : une fois par la macro chapeau et une fois par la macro récursive lors de la première itération (qui ensuite supprime la paire n° 1). À la 2e itération, la macro récursive lit la totalité des paires restantes et supprime la paire n° 2, etc. Ainsi, s'il y a n paires, la première est lue 2 fois, la 2e est lue 3 fois, et ainsi de suite jusqu'à la ne qui est lue $n + 1$ fois.

Il s'agit donc de changer de méthode afin de limiter le nombre de lectures des paires. Envisageons le scénario suivant :

- lire la liste de toutes les paires une fois afin de les transmettre à la macro récursive en les faisant suivre d'une paire spéciale, constituée de deux arguments spéciaux ;
- cette macro récursive, au lieu de lire tous les arguments, n'en lit que deux : pour savoir si elle doit les prendre comme une paire de motifs ou comme condition d'arrêt, il faut qu'elle les teste et les compare aux arguments spéciaux ;
- si le test est positif, cela signifie que la fin de la liste des couples de motifs a été complètement lue et que la fin du processus doit être exécutée ;
- dans le cas d'un test négatif, la substitution est exécutée et la macro s'appelle elle même pour une nouvelle itération.

Cette façon de procéder implique que chaque paire est lu exactement deux fois ; une fois par la macro chapeau et une fois par la macro récursive.

Un dernier détail doit être décidé : que sont des arguments *spéciaux* ? Ce sont des arguments qui ne sont pas susceptibles d'être un des deux motifs. Tout est imaginable, mais il semble assez sûr de prendre un quark tant ce genre de macro

est improbable dans une liste de motifs de substitution. Voici comment modifier le code :

Code n° III-179

```
\catcode`\@11
\def\quark{\quark}

\def\multisubst#1#2{% #1 = <code>   #2 = <liste des paires>
  \def\subst@code{#1}% stocke le <code>
  \multisubst@i#2\quark\quark% appelle la macro récursive avec
                             % 2 arguments spéciaux en fin de liste
}

\def\multisubst@i#1#2{% #1#2 = paire de motifs en cours
  \def\arg@a{#1}% stocke le <motif1> dans une macro
  \ifx\arg@a\quark% si le motif spécial est atteint
    \expandafter\subst@code% exécuter le code obtenu
  \else
    \expandafter\substtocs\expandafter\subst@code\expandafter% 1-développe
      {\subst@code}{#1}{#2}% le <code> et effectue la substitution en cours
    \expandafter\multisubst@i% puis lis la paire de motifs suivants
  \fi
}
\catcode`\@12
\multisubst{abracadabra}{aA rR}\par
\multisubst{Ce texte devenu \`a peine reconnaissable montre que le r\'esultat contient des
  sonorit\'es catalanes, corses ou grecques assez inattendues.}{a{AA} ya uy ou io ei {AA}e}
```

AbRAcAdAbRA
Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís cetelenis, cursis uy gricqyis essiz onettindyis.

62 - Méthode de programmation

Lorsqu'une liste d'arguments consécutifs doit être lue au fur et à mesure par une macro récursive, deux méthodes de lecture sont envisageables. Elles ont en commun une première lecture de tous les arguments afin de les passer à la macro récursive et c'est cette dernière qui a deux façons de fonctionner :

1. lire la totalité des arguments restants à chaque itération, utiliser ceux qui sont utiles à l'itération en cours et recommence1r avec tous les arguments restants. S'arrêter lorsque la liste d'arguments restants est vide ;
2. ne lire *que* les arguments utiles à l'itération en cours, les utiliser et recommencer. S'arrêter lorsque l'un des arguments lus est égal à un argument spécifique préalablement placé en fin de liste par la macro chapeau.

Il est inutile de dire qu'un programmeur rigoureux choisit de préférence la deuxième méthode même si souvent, le nombre d'arguments à lire est peu élevé auquel cas la première méthode est parfaitement acceptable.

5.3. Autres tests

5.3.1. Pseudotests et variables booléennes

TEX dispose des pseudotests \iftrue et \iffalse, le premier est toujours vrai et le second toujours faux. Bien qu'elles en partagent la syntaxe et les subtilités de développement, ces primitives n'effectuent pas de tests. Cependant, nous allons le voir un peu plus bas, elles n'en sont pas moins utiles.

Code nº III-180

```
\iftrue foobar\else ceci n'est jamais lu par \TeX\fi\par
\iffalse ceci n'est jamais lu par \TeX\else foobar\fi
```

foobar
foobar

On peut bien sûr rendre une macro \let-égale à \iftrue ou \iffalse pour, au choix, disposer d'une macro simulant un test toujours vrai ou toujours faux. C'est d'ailleurs exactement ce que fait la macro de plain-TEX \newif⟨*macro*⟩ qui construit 3 macros qui ont un comportement identique à celui de variables « booléennes » que l'on trouve dans d'autres langages.

Ainsi, si nous écrivons \newif\ifbar alors, 3 séquences de contrôle seront créées :

- \bartrue qui rend \ifbar \let-égale à \iftrue ;
- \barfalse qui rend \ifbar \let-égale à \iffalse ;
- \ifbar qui tient lieu de \iftrue ou \iffalse selon la « valeur booléenne » qui lui aura été donnée par l'une des deux premières macros.

Code nº III-181

```
\newif\ifhomme
\def\debutlettre{Ch\ifhomme er Monsieur\else ère Madame\fi}%

\hommetrue
\debutlettre
\medbreak

\hommefalse
\debutlettre
```

Cher Monsieur

Chère Madame

Le format plain-TEX fait en sorte que lorsqu'on écrit \newif\⟨*nom*⟩, le ⟨*nom*⟩ commencer *obligatoirement* par « if ». Il est intéressant de noter que LATEX réimplémente la macro \newif en autorisant le ⟨*nom*⟩ à commencer par deux caractères quelconques qui seront enlevés lorsque \⟨*nomtrue*⟩ ou \⟨*nomfalse*⟩ sont crées.

63 - Règle

Les pseudotests \iftrue et \iffalse n'effectuent aucun test et donc n'admettent aucun argument. Ils sont respectivement toujours vrai et toujours faux. En dépit de

ce caractère invariable, ils se comportent comme n'importe quel test, notamment en ce qui concerne l'éventuelle présence du \else et celle, obligatoire, du \fi.

La macro \newif, suivie d'une macro \if⟨*nom*⟩ crée 3 nouvelles macros dont le comportement est identique à celui des variables booléennes. Le test \if⟨*nom*⟩ sera vrai si la macro \⟨*nom*⟩true a été exécutée auparavant et sera faux si c'est la macro \⟨*nom*⟩false.

5.3.2. Le test \ifcat

Le test \ifcat⟨*token1*⟩⟨*token2*⟩ compare les codes de catégorie des deux tokens qui le suivent immédiatement. Ce test se comporte un peu comme \ifnum en ce sens qu'il lance un développement maximal *avant* de faire le test. Sans action visant à empêcher le développement (utilisation de \noexpand ou \unexpanded), le test compare donc les deux premiers tokens non développables résultant du développement maximal.

Il faut savoir que pour \ifcat, un token de type séquence de contrôle est vu comme ayant un code de catégorie égal à 16. Cette 17e catégorie, spécifique au test \ifcat, vient s'ajouter aux 16 autres catégories déjà existantes pour la primitive \catcode. En revanche, si une séquence de contrôle ou un caractère actif a été préalablement rendu \let-égal à un ⟨*token*⟩, \ifcat prend en compte le code de catégorie du ⟨*token*⟩.

Code n° III-182

```
\catcode`\*=13 \catcode`\+=13 % "*" et "+" sont actifs
1) \def*{xyz}\def+{abc}% définit les caractères actifs
   \ifcat *+vrai\else faux\fi\qquad
2) \ifcat \noexpand *\noexpand +vrai\else faux\fi\qquad
3) \def\foo{foobar}%
   \ifcat \noexpand\foo\relax vrai\else faux\fi\qquad% \foo et \relax sont vus égaux
4) \ifcat = vrai\else faux\fi\qquad% "=" et " " n'ont pas le même catcode
5) \ifcat _^vrai\else faux\fi\qquad% "_" et "^" n'ont pas le même catcode
6) \let\foo=&
   \ifcat &\foo vrai\else faux\fi% "&" et \foo (let-égal à "&") sont vus égaux
```

1) zabcvrai 2) vrai 3) vrai 4) faux 5) faux 6) vrai

Ce qui arrive lors du test n° 1 est facilement compréhensible : le caractère actif « * » est développé au maximum pour donner « xyz ». Le test \ifcat comparera donc les deux premiers tokens issus de ce développement, « x » et « y », et comme ils ont le même catcode, le test sera vrai et tout ce qui est avant le \else sera exécuté. Le « z » restant va être dirigé vers l'affichage. Ensuite, le caractère actif « + » va se développer et afficher « abc ». Enfin, « vrai » sera affiché. On obtient bien l'affichage « zabcvrai ».

Le \noexpand du test n° 3 bloque le développement de la macro \foo et la comparaison qui est faite est bien entre les catcodes de « \foo » et « \relax ».

64 - Règle

Le test \ifcat lance le développement maximal et dès qu'il obtient deux tokens non développables (ou dont le développement est bloqué par \noexpand ou \unexpanded), il effectue la comparaison des codes de catégorie.

Si un token testé par \ifcat est une séquence de contrôle ou un caractère actif rendu \let-égal à un ⟨*token*⟩, \ifcat prend en compte le code de catégorie du ⟨*token*⟩.
Si un token testé par \ifcat est une séquence de contrôle, son code de catégorie est vu égal à 16.

■ **Exercice 75**

Écrire une macro \ifcs⋆, de syntaxe

\ifcs⟨*token*⟩{⟨*vrai*⟩}{⟨*faux*⟩}

qui teste si le ⟨*token*⟩ est une séquence de contrôle au sens de \ifcat. Selon l'issue du test, ⟨*vrai*⟩ ou ⟨*faux*⟩ seront exécutés.

□ **Solution**

La comparaison sera faite entre le ⟨*token*⟩ dont on aura bloqué le développement et \relax. Voici le code :

Code n° III-183

```
\catcode`\@11
\def\ifcs#1{%
  \ifcat\noexpand#1\relax\expandafter\firstoftwo
  \else                  \expandafter\secondoftwo
  \fi
}
\catcode`\@12
1) \def\foo{bar}\ifcs\foo{vrai}{faux}\qquad
2) \ifcs\def{vrai}{faux}\qquad
3) \ifcs A{vrai}{faux}\qquad
\catcode`\*=13
4) \let*=W \ifcs*{vrai}{faux}\qquad
5) \let*=\relax \ifcs*{vrai}{faux}\qquad
6) \let\foo=A \ifcs\foo{vrai}{faux}\qquad
7) \ifcs\bgroup{vrai}{faux}
```

1) vrai 2) vrai 3) faux 4) faux 5) vrai 6) faux 7) faux

Remarquons aux cas n^os^ 5, 6 et 7 que si l'on s'en tient à la vraie définition d'une séquence de contrôle (suite de caractères qui commence par le caractère d'échappement), le test \ifcs que nous avons écrit n'est pas satisfaisant. ■

5.3.3. Le test \if

Le test \if est en tous points semblable au test \ifcat sauf que ce sont les codes de *caractère* des deux tokens qui sont comparés.

65 - Règle

Le test \if instaure un développement maximal et dès qu'il obtient deux tokens non développables (ou dont le développement est bloqué par \noexpand ou \unexpanded), il effectue la comparaison de leurs codes de caractère.
Si un token testé par \if est une séquence de contrôle ou un caractère actif rendu \let-égal à un ⟨*token*⟩, \if prend en compte le code de caractère du ⟨*token*⟩.

Si un token testé par \if est une séquence de contrôle, son code de caractère est vu égal à 256.

Code n° III-184

```
1) \def\foo{xxy}\def\bar{aab}
   \if\foo\bar vrai\else faux\fi\qquad
2) \if aA vrai\else faux\fi\qquad% "a" et "A" n'ont pas les mêmes charcode
3) \if\relax\noexpand\foo vrai\else faux\fi\qquad% \relax et \foo ont un charcode=256
4) \let\foo=&\if\foo &vrai\else faux\fi\qquad% \foo est vue comme "&"
5) \if\string~\noexpand~vrai\else faux\fi
```

1) yaabvrai 2) faux 3) vrai 4) vrai 5) vrai

■ Exercice 76

Utiliser le test \if pour que la macro \ifcs⟨*token*⟩{⟨*vrai*⟩}{⟨*faux*⟩} vue précédemment ne renvoie ⟨*vrai*⟩ que si le ⟨*token*⟩ est une séquence de contrôle, c'est-à-dire lorsqu'il commence par le caractère d'échappement.

□ Solution

Il est inévitable que nous allons devoir tester le caractère d'échappement. Pour cela, nous allons ouvrir un groupe semi-simple et imposer un caractère d'échappement avec \escapechar. Nous le prendrons égal à « @ » mais tout autre choix conviendrait ; l'essentiel est de se prémunir d'un régime spécial où l'\escapechar serait négatif ou nul. Il suffit ensuite de tester si les deux caractères qui se trouvent au début de « \string#1a » (le « a » protège d'un argument vide) et « \string \relax » sont les mêmes. Après avoir fermé le groupe, les classiques \firstoftwo ou \secondoftwo renvoient ⟨*vrai*⟩ ou ⟨*faux*⟩. Pour isoler le premier caractère d'une chaine arbitraire, la macro \firstto@nil sera utilisée.

Code n° III-185

```
\catcode`\@11
\def\ifcs#1{%
  \begingroup
    \escapechar=`\@ % prend "@" comme caractère d'échappement
    \if% les premiers caractères de
      \expandafter\firstto@nil\string#1a\@nil% "#1a"
      \expandafter\firstto@nil\string\relax\@nil% et "\relax" sont-ils égaux ?
      \endgroup\expandafter\firstoftwo% si oui, fermer le groupe et renvoyer "vrai"
    \else% sinon, fermer le groupe et renvoyer "faux"
      \endgroup\expandafter\secondoftwo
    \fi
}
\catcode`\@12
1) \def\foo{bar}\ifcs\foo{vrai}{faux}\qquad
2) \ifcs\def{vrai}{faux}\qquad
3) \ifcs A{vrai}{faux}\qquad
\catcode`\*=13
4) \let*=W \ifcs*{vrai}{faux}\qquad
5) \let*=\relax \ifcs*{vrai}{faux}\qquad
6) \let\foo=A \ifcs\foo{vrai}{faux}\qquad
7) \ifcs\bgroup{vrai}{faux}
```

1) vrai 2) vrai 3) faux 4) faux 5) faux 6) vrai 7) vrai

■

5.3.4. Test incomplets

Entrons à présent au plus profond de la mécanique TEXienne des tests et examinons les tests incomplets. Un test est dit incomplet lorsque les choses à comparer sont en nombre insuffisant et font intervenir les primitives du test lui-même, à savoir `\else` et `\fi`.

Demandons-nous quelle est la comparaison faite dans ce test :

```
\ifx a\fi
```

Si l'on s'en tient à la définition de `\ifx`, le « a » est comparé au `\fi` mais ce même `\fi`, capturé par la comparaison et dès lors indisponible, ne casse-t-il pas l'équilibrage entre le `\ifx` et le `\fi` ?

Dans le même ordre d'idée, que se passe-t-il lors de ce test ?

```
\ifnum1=2\else3\fi 4
```

Lorsque le 2 est lu, TEX est en train de lire un nombre et se trouve donc en phase de développement maximal. Va-t-il développer le `\else`, puis lire 3 puis poursuivre son développement et manger le `\fi` ? Ensuite, continue-t-il son développement plus loin avec le 4 et avec ce qui se trouve après ? Le `\else` et le `\fi`, mangés par la lecture du nombre, rompent-ils l'équilibrage entre le `\ifnum` et le `\fi` ?

66 - Règle

Lorsqu'un `\else` ou un `\fi` apparié avec un test est rencontré lors de l'évaluation de ce test, TEX insère un `\relax` spécial afin que le `\else` ou le `\fi` demeurent hors de l'évaluation faite par le test.

Ainsi, le test « `\ifx a\fi` » devient « `\ifx a\relax\fi` » où le `\relax` est un `\relax` spécial inséré par TEX et qui sera comparé avec « a ». De même, le test « `\ifnum1=2\else3\fi` » devient « `\ifnum1=2\relax\else3\fi` » où le `\relax` spécial interrompt la lecture du nombre 2.

Voici comment définir une macro `\specialrelax`⋆ dont le texte de remplacement est un `\relax` spécial :

```
\edef\specialrelax{\ifnum1=1\fi}
```

Ici, un `\relax` sera inséré avant le `\fi` et comme le test est vrai, ce `\relax` sera tout ce qui reste dans la branche du test.

Code nº III-186

```
1 \edef\specialrelax{\ifnum1=1\fi}
2 \meaning\specialrelax
```

```
macro:->\relax
```

Pourquoi dit-on `\relax` *spécial* et non pas `\relax` tout court ? Les `\relax` spéciaux insérés par TEX sont différents des `\relax` ordinaires lus dans le code source aux yeux de TEX lui-même. En voici la preuve :

Code n° III-187

```
\edef\specialrelax{\ifnum1=1\fi}% texte de remplacement = \relax spécial
\edef\normalrelax{\relax}% texte de remplacement = \relax normal
\meaning\specialrelax\par
\meaning\normalrelax\par
Les 2 \string\relax{} sont \ifx\specialrelax\normalrelax identiques\else différents\fi.
```

```
macro:->\relax
macro:->\relax
Les 2 \relax sont différents.
```

Ils sont différents car `\relax` n'est *pas* une primitive. C'est un token interne défini dans le programme `tex` lui-même qui se donne l'apparence de `\relax` mais qui est codé en dur et qui n'est pas modifiable par l'utilisateur. Autant on pourrait vivre dangereusement et redéfinir la primitive `\relax`, autant il est impossible de redéfinir `\relax` :

Code n° III-188

```
\edef\specialrelax{\ifnum1=1\fi}
\expandafter\def\specialrelax{foobar}% redéfinition un \relax spécial
```

```
Missing control sequence inserted
```

5.4. Tests successifs

5.4.1. Programmation du test `\ifnumcase`

Nous avons vu le test `\ifcase`⟨*nombre*⟩, cousin de `\ifnum`, qui permettait de décider que faire selon des valeurs entières du ⟨*nombre*⟩. L'inconvénient est que ces valeurs doivent commencer à 0 et se suivre de 1 en 1. Cette contrainte rend ce test inutilisable si l'on souhaite tester si le ⟨*nombre*⟩ est successivement égal à des valeurs non consécutives et distantes les unes des autres. L'idée est donc de créer un test qui décide que faire selon des valeurs entières précisées par l'utilisateur (par exemple −5, 14, −16, 20) et offrir une branche facultative `\elseif*` où un code alternatif sera exécuté si aucune égalité n'a été rencontrée.

Nous allons nous fixer la syntaxe suivante :

```
\ifnumcase{⟨nombre⟩}%
    {⟨valeur 1⟩}{⟨code 1⟩}
    {⟨valeur 2⟩}{⟨code 2⟩}
    ...
    {⟨valeur n⟩}{⟨code n⟩}
\elseif
    ⟨code alternatif⟩
\endif
```

Le `\elseif` et le ⟨*code alternatif*⟩ qui le suit sont facultatifs.

De plus, nous allons nous imposer une contrainte : lorsqu'un ⟨*code*⟩ est exécuté, il doit pouvoir accéder aux tokens qui suivent le `\endif`. Autrement dit, ce ⟨*code*⟩ doit être exécuté à la toute fin du traitement, juste avant que la macro ne rende la main.

L'idée directrice est de lire tout d'abord le ⟨*nombre*⟩ puis de passer la main à une macro récursive qui va lire ensuite *un seul* argument. Si cet argument est :

- une ⟨*valeur k*⟩, lire le ⟨*code k*⟩ et l'exécuter si le test est vrai après avoir tout mangé jusqu'au `\endif`. S'il est faux, recommencer ;
- `\elseif`, aucun test n'a été vrai et il faut exécuter le ⟨*code alternatif*⟩ après avoir tout mangé jusqu'au `\endif` ;
- `\endif`, cela signifie la fin du processus et correspond au cas où `\elseif` n'est pas présent et aucun test n'a été vrai.

Comme nous ne savons pas encore tester si un argument est une ⟨*valeur*⟩ (c'est-à-dire un entier), nous allons d'abord tester si l'argument est `\elseif` puis s'il est `\endif` et si les deux tests précédents sont négatifs, nous considèrerons que c'est un entier.

Afin d'être propre dans la programmation, l'ensemble du processus se déroulera dans un groupe semi-simple. Il conviendra de le fermer au bon moment. Pour faciliter la comparaison avec `\ifx`, les macros `\elseif` et `\endif` seront des quarks locaux à ce groupe.

1) ouvrir un groupe, lire le ⟨*nombre*⟩, définir les quarks `\elseif` et `\endif` ;
2) lire le premier argument x qui suit ;
   a) si x = `\elseif`, fermer le groupe et exécuter tout ce qui reste jusqu'à `\endif` ;
   b) sinon
      - si x = `\endif` fermer le groupe ;
      - sinon, cela signifie que x est une ⟨*valeur*⟩
        i) si x = ⟨*nombre*⟩, fermer le groupe, exécuter l'argument suivant après voir tout supprimé tout jusqu'à `\endif` ;
        ii) si $x \neq$ ⟨*nombre*⟩, manger l'argument suivant et retourner en 2.

Nous aurons besoin d'une macro auxiliaire récursive pour le point 2 ainsi que d'une macro auxiliaire qui s'occupe du point 2a que nous appellerons `\idto@endif` :

```
\def\idto@endif#1\endif{#1}
```

et d'une autre pour le point 2bi) que nous appellerons `\firstto@endif`. Elle sera chargée de lire l'argument suivant (argument non délimité `#1`) et de lire tout ce qui se trouve jusqu'au `\endif` (qui sera un argument délimité). Son texte de remplacement sera le premier argument :

```
\def\firstto@endif#1#2\endif{#1}
```

Voici le code :

Code n° III-189

```
\catcode`\@11
\def\ifnumcase#1{%
  \begingroup
  \def\elseif{\elseif}\def\endif{\endif}% définir deux "quarks" locaux
  \def\nombre@{#1}% stocker le nombre à comparer dans \nombre@
  \ifnumcase@i% appeller la macro récursive
}

\def\ifnumcase@i#1{%
  \def\nxt@arg{#1}% stocker l'argument suivant
  \ifx\nxt@arg\elseif% si c'est \elseif
    \def\next@todo{\endgroup\idto@endif}% fermer le groupe et aller à \idto@endif
  \else
    \ifx\nxt@arg\endif% si c'est \endif
```

```
      \let\next@todo\endgroup% fermer le groupe
    \else
      \ifnum\nombre@=\nxt@arg% s'il y a égalité
        \def\next@todo{\endgroup\firstto@endif}% fermer le groupe puis \firstto@endif
      \else% sinon
        \let\next@todo=\ifnumcase@ii% aller à \ifnumcase@ii
      \fi
    \fi
  \fi
  \next@todo% exécuter l'action décidée ci-dessus
}

\def\ifnumcase@ii#1{\ifnumcase@i}
\def\idto@endif#1\endif{#1}
\def\firstto@endif#1#2\endif{#1}
\catcode`\@12
\def\swaptwo#1#2{#2#1}
\def\testenombre#1{%
  \ifnumcase{#1}
    {1}{C'est "un" et je prends le premier argument: \firstoftwo}
    {3}{J'ai vu un "trois" et je mange les deux arguments : \gobtwo}
    {15}{L'argument est "15" et je prends le second argument : \secondoftwo}
  \elseif
    ni 1, ni 3, ni 5 et j'inverse les deux arguments : \swaptwo
  \endif
}
\testenombre{3}{foo}{bar}\par
\testenombre{16}{foo}{bar}\par
\testenombre{15}{foo}{bar}\par
\testenombre{1}{foo}{bar}
```

J'ai vu un "trois" et je mange les deux arguments :
ni 1, ni 3, ni 5 et j'inverse les deux arguments : barfoo
L'argument est "15" et je prends le second argument : bar
C'est "un" et je prends le premier argument : foo

Pour mettre en évidence que chaque ⟨*code*⟩ a accès à ce qui se trouve immédiatement après le `\endif`, les macros `\firstoftwo`, `\secondoftwo` sont appelées à la fin des ⟨*code*⟩ pour qu'elles agissent sur les arguments qui suivent le `\endif`. Une nouvelle macro `\swaptwo` qui inverse l'ordre des deux arguments suivants est également utilisée.

■ Exercice 77

En l'état, la macro `\ifnumcase` n'est pas purement développable, c'est-à-dire qu'on ne peut pas la mettre dans un `\edef`. Comment pourrait-on la modifier pour qu'elle le devienne (à condition bien sûr que les ⟨*code*⟩ le soient)?

□ Solution

Les primitives non développables doivent être supprimées : `\begingroup`, `\endgroup`, `\def` et `\let`. Il n'y aura plus de groupe semi-simple et les quarks seront donc définis de façon globale, à l'extérieur de la macro `\ifnumcase`.

Dans les branches des tests de `\ifnumcase@i`, nous pouvons facilement nous passer de la macro `\next@todo` à condition de recourir aux macros `\firstoftwo` et `\secondoftwo` qui rejettent hors des branches les codes exécutés selon l'issue du test. En revanche, il est plus difficile de se passer de `\nombre@` qui a stocké le ⟨*nombre*⟩ dans la macro chapeau. La seule alternative est de passer le ⟨*nombre*⟩ comme argument supplémentaire à la macro récursive.

Le code tend à se simplifier : l'essentiel est désormais fait par la macro récursive `\ifnumcase` qui à chaque fois, va lire le même premier argument (le ⟨*nombre*⟩) ainsi qu'un second argument qui devra être testé pour décider que faire. Dans le cas où ce second argument est une ⟨*valeur*⟩, il faudra aller lire le ⟨*code*⟩ correspondant, soit pour l'exécuter, soit pour le manger avant de repartir pour un tour.

Code n° III-190

```
\catcode`\@11
\def\endif{\endif}\def\elseif{\elseif}% définit les quarks

\def\ifnumcase#1#2{% #1=<nombre>   #2=prochain argument
  \ifx\elseif#2% si #2 est \elseif
  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    \idto@endif% aller à \idto@endif, sinon :
    {\ifx\endif#2% si #2 est \endif
    \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {}% ne rien faire. Sinon #2 est une <valeur i>:
      {\ifnum#1=#2 % s'il y a égalité entre <nombre> et <valeur i>
      \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
        \firstto@endif% aller à \firstto@endif
        {\ifnumcase@i{#1}}% sinon aller à \ifnumcase@i
      }%
    }%
}

\def\ifnumcase@i#1#2{% #1=<nombre>  #2=<code i> à suivre qui est mangé
  \ifnumcase{#1}% retourner à \ifnumcase en transmettant #1
}

\def\idto@endif#1\endif{#1}
\def\firstto@endif#1#2\endif{#1}
\catcode`\@12
\def\swaptwo#1#2{#2#1}
\def\testenombre#1{%
  \ifnumcase{#1}
    {1}{C'est "un" et je prends le premier argument: \firstoftwo}
    {3}{J'ai vu un "trois" et je mange les deux arguments : \gobtwo}
    {15}{L'argument est "15" et je prends le second argument : \secondoftwo}
  \elseif
    ni 1, ni 3, ni 5 et j'inverse les deux arguments : \swaptwo
  \endif
}
\testenombre{3}{foo}{bar}\par
\testenombre{16}{foo}{bar}\par
\testenombre{15}{foo}{bar}\par
\edef\macro{\testenombre{1}{foo}{bar}}\meaning\macro
```

J'ai vu un "trois" et je mange les deux arguments :
ni 1, ni 3, ni 5 et j'inverse les deux arguments : barfoo
L'argument est "15" et je prends le second argument : bar
macro:->C'est "un" et je prends le premier argument: foo

■

5.4.2. Programmation du test `\ifxcase`

Il ne nous reste plus qu'à faire la même chose avec le test `\ifx` en programmant une macro `\ifxcase` dont voici la syntaxe :

```
\ifxcase ⟨token⟩
    ⟨token 1⟩{⟨code 1⟩}
    ⟨token 2⟩{⟨code 2⟩}
    ...
    ⟨token n⟩{⟨code n⟩}
\elseif%
    ⟨code alternatif⟩
\endif
```

qui compare successivement ⟨*token*⟩ avec ⟨*token 1*⟩, ⟨*token 2*⟩, etc, où les comparaisons sont faites au sens de `\ifx`. Si un des tests est positif, le ⟨*code i*⟩ correspondant sera exécuté et sinon, ce qui se trouve après la macro facultative `\elseif` le sera.

Il n'y a rien de bien difficile, il suffit de recopier le code de `\ifnumcase` en remplaçant le test `\ifnum` par `\ifx` :

Code n° III-191

```
\catcode`\@11
\def\endif{\endif}\def\elseif{\elseif}%
\def\ifxcase#1#2{% #1=<token>   #2=prochain argument
  \ifx\elseif#2% si #2 est \elseif
  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    \idto@endif% aller à \idto@endif, sinon :
    {\ifx\endif#2% si #2 est \endif
    \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {}% ne rien faire. Sinon #2 est un <token i>:
      {\ifx#1#2% s'il y a égalité entre <token> et <token i>
      \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
        \firstto@endif% aller à \firstto@endif
        {\ifxcase@i{#1}}% sinon aller à \ifnumcase@i
      }%
    }%
}
\def\ifxcase@i#1#2{% #1=<token>  #2=<code i> à suivre (qui est mangé)
  \ifxcase{#1}% retourner à \ifxcase en transmettant #1
}
\def\idto@endif#1\endif{#1}
\def\firstto@endif#1#2\endif{#1}
\catcode`\@12
\def\testtoken#1{%
  \ifxcase{#1}
    a{Le token est "a"}
    +{J'ai vu un "+"}
    \relax{L'argument est "\string\relax"}
  \elseif
    Tous les tests sont négatifs%
  \endif
}
1) \testtoken a\par
2) \let\foo=a\testtoken\foo\par
3) \testtoken+\par
4) \testtoken\relax\par
5) \edef\foo{\testtoken\relax}\meaning\foo\par
6) \edef\foo{\testtoken W}\meaning\foo
```

1) Le token est "a"
2) Le token est "a"
3) J'ai vu un "+"
4) L'argument est "\relax"

```
5) macro:->L'argument est "\relax"
6) macro:->Tous les tests sont négatifs
```

On constate aux cas n^os^ 1 et 2 que le test est positif aussi bien lorsque le ⟨*token*⟩ est un « a » explicite que lorsqu'il est une macro `\let`-égale au caractère « a ».

Chapitre 6

Une boucle « loop repeat »

6.1. Implémentation de plain-TEX

Plain-TEX définit lui-même sa boucle « \loop ... \repeat » dont voici l'implémentation telle qu'elle a été écrite par D. Knuth :

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body \let\next\iterate \else\let\next\relax\fi \next}
\let\repeat=\fi
```

Il est clair que la macro \loop est à argument délimité (le délimiteur est \repeat) et stocke dans la macro \body tout ce qui se trouve entre \loop et \repeat (que l'on appelle « corps de la boucle ») avant d'appeler la macro \iterate. On remarque également qu'aucune macro n'est déclarée \long et donc il faudra proscrire la primitive \par entre \loop et \repeat. Ensuite, l'idée est que ce qui se trouve entre \loop et \repeat, stocké dans \body, *doit* contenir un test. Ce test, s'il est positif, autorisera la boucle exécuter une itération de plus tandis que s'il est négatif, il constitue la *condition d'arrêt* de la boucle qui lui commande de s'arrêter. Cela suppose que le corps de la boucle modifie un paramètre pris en compte par le test afin que ce dernier devienne à un moment négatif pour permettre la sortie de la boucle et éviter ainsi une boucle infinie.

Le mécanisme est assez facile à saisir : ce test sera exécuté lorsque \iterate puis \body seront développés. Si ce test est positif, \next est rendue \let-égale à \iterate et sinon, elle est rendue \let-égale à \relax . Cette macro \next est ensuite appelée après être sorti du test, ce qui rend la récursivité terminale.

La structure de la boucle est donc la suivante :

```
\loop
    ⟨code a⟩
    ⟨test⟩
    ⟨code b⟩
\repeat
```

où les ⟨*code a*⟩ et ⟨*code b*⟩ sont facultatifs, c'est-à-dire que l'un ou l'autre peuvent être réduit à 0 token. Il est en revanche important de comprendre que ⟨*code a*⟩ est toujours exécuté au moins une fois puisqu'il vient avant le test.

De par sa construction, il est également important de remarquer que ce qui se trouve entre `\loop` et `\repeat` ne doit pas contenir un test accompagné d'un `\else` puisqu'un `\else` est déjà écrit en dur dans la macro `\iterate`. Si l'utilisateur écrivait un `\else` par mégarde, cela provoquerait une erreur de type « `extra \else` » lorsque TEX atteindrait le second `\else`. Malgré cette petite contrainte, cette structure est très souple du fait que le test peut se trouver où l'on veut entre `\loop` et `\repeat`.

Enfin, on constate à la dernière ligne que `\repeat` est rendu `\let`-égal à `\fi` ce qui a pour conséquence que `\repeat` sera vu par TEX comme un `\fi` et donc pris en compte lors du comptage interne de TEX des tests, `\else` et des `\fi`. En d'autres termes, le `\repeat` sera bien apparié avec le ⟨*test*⟩ se trouvant entre `\loop` et `\repeat` et par conséquent, la boucle peut être mise dans les branches d'un test sans provoquer de mauvais appariements entre les tests et les `\fi`. Une structure de ce type est donc tout à fait envisageable :

```
⟨test⟩
    ⟨code 1⟩
    \loop
        ...
    \repeat
    ⟨code 2⟩
\else
    ⟨code 3⟩
    \loop
        ...
    \repeat
    ⟨code 4⟩
\fi
```

Voici à titre d'exemple comment on peut programmer la macro `\compte` vue dans un exercice du chapitre 3 :

Code n° III-192

```
\newcount\xx % définit un compteur utilisé dans la boucle
\def\compte#1{%
  \xx=0 % initialise le compteur
  \loop% début de la boucle
    \advance\xx1 % incrémente le compteur
    \ifnum\xx<#1 % s'il est inférieur à la borne
      \number\xx , % affiche le nombre et la virgule
  \repeat% et recommence
  \ifnum#1>0 \number\xx\relax\fi% si le nombre #1 est >0, afficher le dernier nombre
  }
a) \compte{1}.\qquad b) \compte{-3}.\qquad c) \compte{7}.
```

a) 1. b) . c) 1, 2, 3, 4, 5, 6, 7.

6.2. Implémentation de LaTeX

Le formatLaTeX redéfinit la boucle \loop … \repeat de cette façon :

```
\long\def\loop#1\repeat{%
  \def\iterate{#1\relax\expandafter\iterate\fi}%
  \iterate
  \let\iterate\relax}
\let\repeat=\fi
```

La macro \loop est déclarée \long et l'on fait l'économie de la macro \body en déclarant \iterate comme une « sous-macro » de \loop de telle sorte que l'argument #1 lui soit accessible. En considérant que #1 contient un test, on voit clairement que ce test peut être accompagné d'un \else puisque le code de la macro \iterate ne comporte pas cette primitive mais seulement un \fi. Le plus simple est de faire fonctionner la boucle dans les cas où son corps contient un \else ou pas :

Sans \else

```
\loop
    ⟨code a⟩
    ⟨test⟩
        ⟨code b⟩
\repeat
```

Avec \else

```
\loop
    ⟨code a⟩
    ⟨test⟩
        ⟨code b⟩
    \else
        ⟨code c⟩
\repeat
```

Voilà ce qui est stocké dans la macro \iterate dans chaque cas :

Sans \else

```
⟨code a⟩
⟨test⟩
    ⟨code b⟩\relax
    \expandafter\iterate
\fi
```

Avec \else

```
⟨code a⟩
⟨test⟩
    ⟨code b⟩
\else
    ⟨code c⟩\relax
    \expandafter\iterate
\fi
```

Il est maintenant facile de constater que si le corps de la boucle contient un \else, c'est la branche du test négatif (après le \else) qui maintiendra la récursivité. En revanche, s'il n'y a pas de \else, c'est la branche du test positif qui la maintient. Autrement dit, la condition d'arrêt est un test *négatif* s'il n'y a pas de \else alors que c'est un test *positif* s'il y en a.

Malgré ce petit manque de cohérence, on peut dire que l'implémentation LaTeX est supérieure à celle de plain-TeX puisque la souplesse de pouvoir mettre de \else dans la boucle rend parfois service, sans compter que l'on peut également faire appel à \unless pour échanger les branches du test.

6.3. Imbrications de boucles

Malgré leur côté pratique, ni l'implémentation TeX ni celle de LaTeX ne permettent l'imbrication de boucles \loop … \repeat, principalement à cause de deux choses :

– si deux boucles étaient imbriquées, le \repeat de la boucle intérieure serait pris comme fin de la boucle extérieure puisqu'avec les arguments délimités, la première occurrence du délimiteur est prise en compte ;
– la macro \iterate serait redéfinie par la boucle intérieure.

Avec la syntaxe actuelle, aucune implémentation facile à programmer et n'utilisant pas de groupe permet d'imbriquer des boucles \loop \repeat n'est faisable. La difficulté est de trouver ce qui se trouve entre un \loop et son \repeat apparié ! Nous allons donc tenter de bâtir une boucle \xloop*...\xrepeat*, fonctionnant de la même façon que \loop...\repeat sauf que l'imbrication des boucles sera possible.

Pour commencer, nous allons nous attacher à collecter le code compris entre un \xloop et son \xrepeat apparié. Ce code sera stocké dans la macro \xiterate et simplement *affiché* :

1) définir un compteur \cnt@repeat qui comptabilisera les occurrences de \xrepeat rencontrées, et l'initialiser à 0. Initialiser \xiterate à vide ;
2) ajouter à \xiterate tout le code qui se trouve jusqu'au prochain \xrepeat ;
3) stocker dans \cnt@loop le nombre de fois que \xloop est contenu dans le texte de remplacement de \xiterate ;
4) tester si \cnt@loop = \cnt@repeat ;
    a) si le test est positif, afficher le texte de remplacement de \xiterate à l'aide \detokenize ;
    b) sinon, incrémenter \cnt@repeat, ajouter \xrepeat à \xiterate et retourner en 2.

Voici un code TEX qui transcrit cet algorithme, où nous nous servons des macros \addtomacro et \ifin vues précédemment. Nous utiliserons aussi la macro \cnttimestocs vue dans la solution à la page 199 :

Code n° III-193

```
\catcode`\@11
\newcount\cnt@repeat % définit le compteur de \xrepeat

\def\xloop{%
  \def\xiterate{}% initialiser \xiterate à vide
  \cnt@repeat\z@% initialiser le compteur de \xrepeat
  \xloop@i% aller à la macro récursive
}

\long\def\xloop@i#1\xrepeat{%
  \addtomacro\xiterate{#1}% ajoute ce qui est avant le premier \xrepeat
  \exparg\cnttimestocs{\xiterate}\xloop\cnt@loop
  % combien de \xloop dans \xiterate
  \ifnum\cnt@loop=\cnt@repeat\relax
    \expandafter\firstoftwo\else\expandafter\secondoftwo
  \fi
    {% autant que de \xrepeat -> \detokenize pour afficher
    "\detokenize\expandafter{\xiterate}"%
    }
    {\addtomacro\xiterate\xrepeat% sinon, ajouter ce \xrepeat
    \advance\cnt@repeat by 1% incrémenter le compteutr de \xrepeat
    \xloop@i% et chercher le prochain \xrepeat
    }%
}
\let\xrepeat\fi
```

```
\catcode`\@12
\xloop a\xloop b\xloop 12\xrepeat c\xrepeat \xloop X\xrepeat\xrepeat
```

```
"a\xloop b\xloop 12\xrepeat c\xrepeat \xloop X\xrepeat "
```

Cela fonctionne bien : tout ce qui se trouve entre le premier `\xloop` et son `\xrepeat` apparié (qui est le dernier ici) est bien affiché. Maintenant, au lieu de simplement détokénizer le texte de remplacement de `\xiterate`, nous allons mimer avec cette macro ce que fait l'implémentation LaTeX. De plus, au lieu d'écrire `\xiterate`, il faudra écrire partout dans le code `\xiterate@`⟨*n*⟩ de telle sorte que les imbrications de boucles ne redéfinissent pas cette macro et aient chacune la leur. Le nombre ⟨*n*⟩ est le compteur d'imbrications stocké dans le compteur `\cnt@nested`. Celui-ci est initialisé à 0 en dehors de toute macro, incrémenté à chaque fois que `\xloop` est rencontré et décrémenté lors de la sortie de boucle.

Bien entendu, pour former la macro `\xiterate@`⟨*n*⟩, il va falloir faire appel à la paire `\csname` … `\endcsname` et donc, alourdir sensiblement le code :

Code n° III-194

```
\catcode`\@11
\newcount\cnt@repeat
\newcount\cnt@nested
\cnt@nested=0 % compteur d'imbrications
\def\xloop{%
  \global\advance\cnt@nested by 1 % augmente le compteur d'imbrications
  \expandafter\def\csname xiterate@\number\cnt@nested\endcsname{}%
  \cnt@repeat\z@% initialise le compteur de \xrepeat à 0
  \xloop@i% aller à la macro \xloop@i
}
\long\def\xloop@i#1\xrepeat{%
  \expandafter\addtomacro\csname xiterate@\number\cnt@nested\endcsname{#1}%
  \expandafter\expandafter\expandafter\cnttimestocs\expandafter\expandafter\expandafter
    {\csname xiterate@\number\cnt@nested\endcsname}\xloop\cnt@loop
  \ifnum\cnt@loop=\cnt@repeat\relax
    \expandafter\firstoftwo\else\expandafter\secondoftwo
  \fi
    {\expandafter\eaddtomacro\csname xiterate@\number\cnt@nested\expandafter\endcsname
      {\expandafter\expandafter\csname xiterate@\number\cnt@nested\endcsname\fi}%
    %\expandafter\show\csname xiterate@\number\cnt@nested\endcsname
    \csname xiterate@\number\cnt@nested\endcsname
    \letname{xiterate@\number\cnt@nested}\relax
    \global\advance\cnt@nested by -1
    }
    {\expandafter\addtomacro\csname xiterate@\number\cnt@nested\endcsname\xrepeat
    \advance\cnt@repeat by 1
    \xloop@i
    }%
}%
\let\xrepeat\fi
\catcode`\@12

\newcount\cntxx \cntxx=1 % compteur des lignes
\newcount\cntyy
\xloop
  \cntyy=1 % compteur des colonnes
  \xloop
    (\number\cntxx,\number\cntyy)% affiche "(ligne,colonne)"
    \ifnum\cntyy<5 % tant que colonne<5
```

```
40    \advance\cntyy1 % incrémente colonne
41    , % <- affiche ", "
42   \xrepeat% et recommence
43   \ifnum\cntxx<3 % tant que ligne<3
44   \advance\cntxx1 % incrémente ligne
45   \par % va à la ligne
46 \xrepeat% et recommence
```

(1,1), (1,2), (1,3), (1,4), (1,5)
(2,1), (2,2), (2,3), (2,4), (2,5)
(3,1), (3,2), (3,3), (3,4), (3,5)

L'utilisation de `\csname...\endcsname` rend la lecture du code bien plus difficile puisqu'elle implique le développement d'un pont d'`\expandafter` pour faire naitre une macro puis pour accéder à son texte de remplacement d'une macro (lignes n^{os} 12 à 12+3). Malgré cela, intrinsèquement, le code est identique à celui vu précédemment. La seule différence se trouve entre les lignes n^{os} 18 à n^{o} 23. En effet, aux lignes n^{os} 18-19, au lieu de détokénizer la macro `\xiterate@`⟨*n*⟩ pour simplement l'afficher comme on le faisait en analyse préalable, on ajoute au texte de remplacement de cette macro « `\expandafter` » suivi du nom de la macro, c'est-à-dire `\xiterate@`⟨*n*⟩ que l'on fait suivre d'un `\fi`. Ainsi, le texte de remplacement de `\xiterate@`⟨*n*⟩ est finalement :

⟨*ce qui est entre \xloop et \xrepeat*⟩ `\expandafter\xiterate@`⟨*n*⟩ `\fi`

On peut d'ailleurs prendre connaissance du texte de remplacement de la macro `\xiterate@`⟨*n*⟩ en dé-commentant la ligne n^{o} 20 pour laisser à la primitive `\show` l'écrire dans le fichier `log`. Pour `\xiterate@1`, on obtient :

```
\xiterate@1=macro:->
\cntyy =1
\xloop
    (\number \cntxx ,\number \cntyy )
    \ifnum \cntyy <5 ,
    \advance \cntyy 1
\xrepeat
\ifnum \cntxx <3
    \advance \cntxx 1
    \par
    \expandafter\xiterate@1
\fi.
```

et pour `\xiterate@2` :

```
\xiterate@2=macro:->
(\number \cntxx ,\number \cntyy )
\ifnum \cntyy <5 ,
\advance \cntyy 1
    \expandafter\xiterate@2
\fi.
```

C'est exactement ce que fait l'implémentation LaTeX de la boucle `\loop` … `\repeat` avec la macro `\iterate`. En poursuivant le parallèle avec la boucle LaTeX, il suffit ensuite de lancer la macro `\xiterate@`⟨*n*⟩ à la ligne n^{o} 21 et une fois la boucle terminée, on rend `\iterate@`⟨*n*⟩ `\let`-égale à `\relax` à la ligne n^{o} 22. Le tout se termine avec la décrémentation du compteur d'imbrication à la ligne n^{o} 23.

Chapitre 7

Une boucle « foreach in »

7.1. Une première approche

Nous allons construire une boucle où une « variable » – qui sera comme précédemment une séquence de contrôle – prendra successivement toutes les valeurs d'une liste de valeurs, pas nécessairement numériques, séparées par des virgules et à chaque itération, un code sera exécuté. La syntaxe de cette boucle, initiée par la macro `\doforeach`⋆, sera [1] :

`\doforeach\`⟨*macro*⟩`\in{`⟨*valeur 1*⟩`,`⟨*valeur 2*⟩`,...,`⟨*valeur n*⟩`}{`⟨*code*⟩`}`

où bien entendu, le ⟨*code*⟩ peut contenir la \⟨*macro*⟩.

La méthode, relativement simple, va être de lire toutes les valeurs une première fois, y ajouter à la fin une valeur reconnaissable (un quark) et par la suite, lire les valeurs une par une jusqu'à ce qu'apparaisse le quark. Nous mettons ici en œuvre la méthode recommandée pour lire une série d'arguments (voir page 203) :

1) lire la liste des valeurs et lui ajouter « `,\end@foreach,` », où `\end@foreach` est un quark.
   Sauvegarder le ⟨*code*⟩ dans une macro (que nous appelons `\loop@code`) ;
2) lire la valeur qui se trouve avant la virgule et l'assigner à \⟨*macro*⟩ ,
3) si \⟨*macro*⟩=`\end@foreach`, finir le processus ;
4) sinon exécuter le ⟨*code*⟩ et retourner en 2.

Le point 1 sera dévolu à une macro chapeau chargée de faire les assignations préalables. Les autres points seront contenus dans une macro récursive à argument

1. La commande `\foreach` est définie par le package « `pgf` » et pour éviter tout conflit, il vaut donc mieux choisir un autre nom.

délimité. Choisissons d'externaliser la macro récursive (c'est-à-dire de la définir en dehors du texte de remplacement de la macro chapeau). Assurons-nous que la macro récursive peut accéder à tous les arguments de la macro chapeau. Aucun problème pour la liste des valeurs qu'elle lira comme argument au fur et à mesure. Le ⟨*code*⟩ ne pose pas de problème puisqu'il est sauvegardé dans `\loop@code`. La variable `\⟨macro⟩` n'étant pas sauvegardée, prenons comme solution de la transmettre comme argument à la macro récursive : cette macro (qui n'est qu'un seul token) viendra comme premier argument non délimité, avant la liste des valeurs.

Code nº III-195

```
1  \catcode`\@11
2  \def\end@foreach{\end@foreach}% définit un quark
3  \long\def\doforeach#1\in#2#3{%
4    \def\loop@code{#3}% assigne le <code> à \loop@code
5    % appel à \doforeach@i : mettre la macro #1 en premier, puis la liste de valeurs #2
6    \doforeach@i#1#2,\end@foreach,% ajouter à la fin ",\end@foreach,"
7    % une fois les boucles finies, neutraliser les macros déjà définies
8    \let#1\relax \let\loop@code\relax
9  }
10
11 \long\def\doforeach@i#1#2,{% #1=\<macro>  #2=valeur courante
12   \def#1{#2}% stocke la valeur en cours dans la \<macro>
13   \unless\ifx\end@foreach#1% si \end@foreach n'est pas atteint
14     \antefi% amène le \fi ici
15     \loop@code% exécute le code
16     \doforeach@i#1% et recommencer
17   \fi
18 }
19 \catcode`\@12
20 \doforeach\x\in{a,bcd,{efg},hi}{\meaning\x.\par}
```

```
macro:->a.
macro:->bcd.
macro:->efg.
macro:->hi.
```

La ligne nº 11 montre que la macro `\doforeach@i` admet comme argument *non délimité* `#1` alors que la valeur en cours est l'argument `#2` délimité par la virgule. L'inconvénient, lié à la lecture d'argument, est que si une ⟨*valeur*⟩ est constituée d'un texte entre accolades, ce texte est dépouillé de ses accolades et assigné à la variable. Si l'on souhaitait éviter cet écueil, il faudrait mettre un token (par exemple un `\relax`) avant chaque ⟨*valeur*⟩ et faire en sorte de manger ce `\relax` avec `\gobone` avant d'assigner la ⟨*valeur*⟩ à la variable. Par souci de simplicité, cette manœuvre ne sera pas programmée.

■ Exercice 78

On pourrait vouloir modifier le séparateur de liste (qui est la virgule par défaut) pour mettre des nombres décimaux dans la liste d'arguments, par exemple.

Créer une macro `\defseplist*` dont l'argument est le séparateur voulu et modifier le code vu ci-dessus pour que la macro `\doforeach` tienne compte du séparateur ainsi défini.

□ Solution

L'idée est de faire de `\defseplist`⟨*séparateur*⟩ une macro « enveloppante » qui va définir, à chaque fois qu'elle sera appelée, les deux macros `\doforeach` et `\doforeach@i`, toutes les deux dépendantes de l'argument `#1` de `\defseplist`. Dans le code déjà vu, la virgule va

être replacée par #1 et les autres « # » qui s'appliquent aux arguments de \doforeach et \doforeach@i vont être doublés pour tenir compte de l'imbrication des macros :

Code n° III-196

```
\catcode`\@11
\def\end@foreach{\end@foreach}
\def\defseplist#1{%
  \long\def\doforeach##1\in##2##3{%
    \def\loop@code{##3}% assigne le <code> à \loop@code
    \doforeach@i##1##2#1\end@foreach#1% ajouter ",\end@foreach," et aller à \doforeach@i
    % après être sorti des boucles, neutraliser les macros déjà définies
    \let\loop@code\relax \let##1\relax
  }%
  \long\def\doforeach@i##1##2#1{%
    \def##1{##2}% stocke la valeur en cours dans la \<macro>
    \unless\ifx\end@foreach##1% si la fin n'est pas atteinte
      \antefi% amène le \fi ici
      \loop@code% exécute le code
      \doforeach@i##1% et recommencer
    \fi
  }%
}
\defseplist{,}% définit les macros avec le séparateur par défaut
\catcode`@12

\doforeach\x\in{a,bcd,efg}{Argument courant : "\x".\par}\medbreak

\defseplist{--}% séparateur "--"
\doforeach\nn\in{4,19--0,5--8,575}{Nombre lu : "\nn"\par}
```

Argument courant : "a".
Argument courant : "bcd".
Argument courant : "efg".

Nombre lu : "4,19"
Nombre lu : "0,5"
Nombre lu : "8,575"

■

7.2. Rendre possible l'imbrication

Il nous reste à rendre possible l'imbrication de boucles \doforeach et pour cela, comme précédemment avec \for et \xloop, nous allons définir un compteur qui sera chargé de compter le niveau d'imbrication et qui fera partie du nom de la macro chargée de stocker le ⟨*code*⟩. Nous prendrons « \loop@code@⟨*n*⟩ » :

Code n° III-197

```
\catcode`\@11
\newcount\cnt@nest \cnt@nest=0 % définit et initialise le compteur d'imbrication
\def\end@foreach{\end@foreach}

\def\defseplist#1{%
  \long\def\doforeach##1\in##2##3{%
    \global\advance\cnt@nest1 % entrée de boucle : incrémenter le compteur d'imbrication
    \defname{loop@code@\number\cnt@nest}{##3}% assigne le <code> à \loop@code@<n>
    \doforeach@i##1##2#1\end@foreach#1% ajouter ",\end@foreach," et aller à \doforeach@i
    % une fois fini :
```

```
    \let##1\empty% dans ce cas, neutraliser les macros déjà définies
    \letname{loop@code@\number\cnt@nest}\empty%
    \global\advance\cnt@nest-1 %sortie de boucle : décrémenter le compteur d'imbrication
  }%
  \long\def\doforeach@i##1##2#1{%
    \def##1{##2}% stocke la valeur en cours dans la \<macro>
    \unless\ifx\end@foreach##1% tant que la fin n'est pas atteinte
      \antefi% amène le \fi ici
      \csname loop@code@\number\cnt@nest\endcsname% exécute le code
      \doforeach@i##1% et recommencer
    \fi
  }%
}
\catcode`@12
\defseplist{,}

\doforeach\xxx\in{1,2,3}{%
  Ligne \xxx{} : \doforeach\yyy\in{a,b,c,d,e}{(\xxx,\yyy)}.\par
}
```

Ligne 1 : (1,a)(1,b)(1,c)(1,d)(1,e).
Ligne 2 : (2,a)(2,b)(2,c)(2,d)(2,e).
Ligne 3 : (3,a)(3,b)(3,c)(3,d)(3,e).

Profitons également de la simplicité relative de ce code pour implémenter une fonctionnalité qui pourrait être d'un grand secours. Si la « variable » est déjà définie lorsque la boucle est rencontrée, il serait judicieux de la sauvegarder pour la restaurer une fois la boucle terminée. Imaginons en effet qu'un utilisateur imprudent – ou inconscient – nomme sa variable `\par` par analogie avec le mot « paramètre » et construise une boucle de cette façon :

```
\doforeach\par\in{a,b,c,y,z}{$\par_i$}
```

La primitive `\par` serait silencieusement redéfinie à l'entrée de la boucle, pour être rendue `\let`-égale à `\relax` à la fin ce qui aurait pour effet de rendre TEX incapable de former un paragraphe. On imagine les dégâts, d'autant plus incompréhensibles que le tout se passe silencieusement. Il nous faut donc tester si la variable existe au début de la boucle et si tel est le cas, la sauvegarder avec `\let` pour la restaurer à la fin du processus.

Il existe un test `\ifdefined\`⟨*commande*⟩ de ε-TEX qui est vrai si la `\`⟨*commande*⟩ est définie et faux sinon. Ce test ne modifie pas la table de hashage [2]. Il existe aussi un test de ε-TEX un peu similaire `\ifcsname`⟨*nom d'une commande*⟩`\endcsname`, qui forme la ⟨*commande*⟩ comme le ferait `\csname` puis teste si elle existe, sans modifier la table de hashage.

La méthode sera ici de mettre à profit `\ifdefined` pour tester si la variable existe et la sauvegarder dans `\saved@var@`⟨*n*⟩ si le test est positif. À la sortie de

2. Les tests `\@ifundefined{`⟨*nom*⟩`}` et `\@ifdefinable\`⟨*nom*⟩ de LATEX forment la séquence de contrôle `\`⟨*nom*⟩ à l'aide de `\csname` et testent si elle est égale à `\relax` auquel cas, la séquence de contrôle est considérée non définie. Cette méthode présente deux gros inconvénients :

- toute macro existante et `\let`-égale à `\relax` est considérée par ces tests comme étant non définie ;
- par l'utilisation de `\csname`, une macro non définie devient définie et `\let`-égale à `\relax`. Un autre dégât collatéral est que la table de hashage est modifiée puisqu'une nouvelle macro est créée.

la boucle, il suffira de faire la manœuvre inverse pour restaurer la variable soit à \relax, soit à ce qu'elle était avant si elle existait auparavant.

Dans le code ci-dessous, la variable a été nommée « \par » ce qui est une chose à ne pas faire. Même si un dispositif a été prévu pour que \par soit restaurée à la fin du processus, redéfinir une primitive, surtout aussi primordiale que \par, *est un danger qu'il faut à tout prix éviter de courir.*

Code n° III-198

```
\catcode`\@11
\newcount\cnt@nest \cnt@nest=0 % définit et initialise le compteur d'imbrication
\def\end@foreach{\end@foreach}
\long\def\save@macro#1{% sauvegarde la macro #1
  \ifdefined#1% si la macro #1 est déjà définie...
    \letname{saved@var@\number\cnt@nest}#1% ...la sauvegarder
  \fi
}
\long\def\restore@macro#1{% restaure la macro #1
% le \csname donne \relax si la sauvegarde n'a pas été faite
  \expandafter\let\expandafter#1\csname saved@var@\number\cnt@nest\endcsname
}
\def\defseplist#1{%
  \long\def\doforeach##1\in##2##3{%
    \global\advance\cnt@nest1 % entrée de boucle : incrémenter le compteur d'imbrication
    \save@macro##1% sauvegarde la macro ##1
    \defname{loop@code@\number\cnt@nest}{##3}% assigne le <code> à \loop@code@<n>
    \doforeach@i##1##2#1\end@foreach#1% ajouter ",\end@foreach," et aller à \doforeach@i
    % une fois fini, neutraliser la macro contenant le <code>
    \letname{loop@code@\number\cnt@nest}\empty
    \restore@macro##1% restaurer la \<macro>
    \global\advance\cnt@nest-1 % et décrémenter le compteur d'imbrication
  }%
  \long\def\doforeach@i##1##2#1{%
    \def##1{##2}% stocke la valeur en cours dans la \<macro>
    \unless\ifx\end@foreach##1% si la fin n'est pas atteinte
      \antefi% amène le \fi ici
      \csname loop@code@\number\cnt@nest\endcsname% exécute le code
      \doforeach@i##1% et recommencer
    \fi%
  }%
}
\catcode`@12
\defseplist{,}
\doforeach\par\in{a,b,c,y,z}{$\par_i$}

\meaning\par% \par doit être redevenu une primitive

\doforeach\xxx\in{1,2,3}{%
  Ligne \xxx{} : \doforeach\yyy\in{a,b,c,d,e}{(\xxx,\yyy)}.\par
}
```

$a_i b_i c_i y_i z_i$
\par
Ligne 1 : (1,a)(1,b)(1,c)(1,d)(1,e).
Ligne 2 : (2,a)(2,b)(2,c)(2,d)(2,e).
Ligne 3 : (3,a)(3,b)(3,c)(3,d)(3,e).

7.3. Boucle à deux variables

Une amélioration de la macro \doforeach serait de détecter si la variable est seule, comme envisagé précédemment, ou si l'utilisateur souhaite avoir un *couple* de variables auquel cas, il écrirait :

\doforeach\⟨*macro1*⟩/\⟨*macro2*⟩\in{x1/y1,x2/y2,...,xn/yn}{⟨*code*⟩}

Chacune des variables \⟨*macro1*⟩ et \⟨*macro2*⟩ prendrait successivement les valeurs appariées spécifiées dans l'argument qui suit le \in.

L'idée va être de se servir de la macro \ifin vue précédemment pour tester si l'argument délimité par « \in » contient le caractère « / » et agir en conséquence. Si le test est positif, il faut s'orienter vers une macro à argument délimité du type #1/#2 qui sépare les deux variables. Nous allons donc, selon l'issue du test \ifin{#1}{/}, nous orienter vers la macro vue précédemment si le test est négatif ou vers une macro \doforeach@ii avec des arguments délimités appropriés. Nous allons profiter de cette nouvelle fonctionnalité pour améliorer et optimiser le code.

En premier lieu, la macro \save@macro effectuera *dans tous les cas* la sauvegarde de la (ou des) variables avec \let, que ces variables soient définies ou pas. La raison de ce choix, qui implique de se passer du test \ifdefined, réside dans le fait que la variable est rendue \let-égale à \relax à la fin du processus : cela sera *aussi* le cas si la variable est une macro indéfinie. En effet, si une macro x est rendue \let-égale à une macro y non définie, x devient également non définie. Par la suite, si y est rendue \let-égale à x où la macro x est construite via \csname, la macro y devient \let-égale à \relax

Code n° III-199

```
1 \let\foo=\djfhdsldv% \foo est rendue \let-égale à une macro indéfinie
2 a) \meaning\foo\par
3 % puis \djfhdsldv est rendue \let-égale à \foo (ou \foo est construit avec \csname)
4 \expandafter\let\expandafter\djfhdsldv\csname foo\endcsname
5 b) \meaning\djfhdsldv
```

a) undefined
b) \relax

L'optimisation que nous pouvons envisager est d'éviter que la macro \csname (qui est lente) ne se trouve dans une boucle comme c'était le cas à la ligne n° 28 du code précédent. Pour cela, nous allons donner le code à exécuter comme argument des macros récursives. L'appel initial aux macros récursives aura donc la structure suivante :

\doforeach@i{⟨*code*⟩}⟨*var*⟩x1,x2,...,xn,⟨*quark*⟩,

ou

\doforeach@ii{⟨*code*⟩}⟨*var1*⟩/⟨*var2*⟩x1/y1,x2/y2,...xn/yn,⟨*quark*⟩/⟨*quark*⟩,

Ces macros ne liront à chaque itération que le ⟨*code*⟩, la (ou le couple de) variable(s), ainsi que la valeur courante xi ou le couple de valeurs courant xi/yi. Par conséquent, l'appel récursif sera de la forme

\doforeach@i{#1}#2 ou \doforeach@i{#1}#2/#3

où #2 est la variable et #2/#3 est le couple de variables. Comme les récursivités sont terminales, les valeurs sont prises dans le code qui reste à lire.

Comme ces macros récursives effectuent un test \ifx à chaque itération pour déterminer si le quark marquant la fin des valeurs est atteint, nous devons nous garder de l'erreur qui consisterait à utiliser \antefi alors que le ⟨*code*⟩ se situe en \antefi et \fi. Il s'agirait d'une erreur, car si le ⟨*code*⟩ contient un test, l'\antefi (dont l'argument est délimité par \fi) prendrait le \fi du ⟨*code*⟩ comme délimiteur et romprait l'appariement correct entre \ifx et son \fi.

Enfin, il nous faut inventer un moyen de créer la macro \doforeachexit⋆, que l'utilisateur peut écrire dans le ⟨*code*⟩ et qui termine prématurément toutes les boucles \doforeach en cours d'exécution. Cette macro doit annuler l'appel récursif vu ci-dessus. L'idée directrice est de modifier ces appels récursifs de cette manière :

\allow@recurse{\doforeach@i{#1}#2}
ou
\allow@recurse{\doforeach@i{#1}#2/#3}

La macro \allow@recurse sera rendue \let-égale à \identity au début de la boucle pour permettre par défaut que l'appel récursif se fasse. Si l'utilisateur insère \doforeachexit dans le ⟨*code*⟩, \allow@recurse doit être modifiée pour que cette dernière mange tout ce qui reste. « Tout » signifie l'appel récursif *et* toutes les valeurs (ou couples de valeurs) restant à lire. Comme ce « tout » se termine forcément par le quark \end@foreach et le séparateur qui ont été insérés par l'appel initial, une macro \fordib@recurse à argument délimité effectuant cette tâche est facile à créer :

\long\def\fordib@recurse##1\end@foreach#1{}

Une fois tout ceci mis en place, la macro \doforeachexit n'a plus qu'à rendre \allow@recurse \let-égale à \fordib@recurse.

Code n° III-200

```
\catcode`\@11
\newcount\cnt@nest \cnt@nest=0 % définit et initialise le compteur d'imbrication
\def\end@foreach{\end@foreach}
\long\def\save@macro#1#2{\letname{saved@var@\number\cnt@nest#1}#2}
\long\def\save@macro@i#1/#2{\save@macro{a}#1\save@macro{b}#2}
\long\def\restore@macro#1#2{%
  \expandafter\let\expandafter#2\csname saved@var@\number\cnt@nest#1\endcsname
}
\long\def\restore@macro@i#1/#2{\restore@macro{a}#1\restore@macro{b}#2}
\def\defseplist#1{%
  \long\def\doforeach##1\in##2##3{%
    \global\advance\cnt@nest1 % entrée de boucle : incrémenter le compteur d'imbrication
    \global\let\allow@recurse\identity% permet l'appel récursif plus bas
    \ifin{##1}{/}% si ##1 contient "/"
      {\save@macro@i##1% sauvegarde les macros
      \doforeach@ii% appeler la macro récursive avec les arguments
        {##3}% 1) code à exécuter
        ##1%   2) variables sous la forme \<macro1>/\<macro2>
        ##2#1\end@foreach/\end@foreach#1% puis la liste ##2 suivie de
                                        % ",\end@foreach/\end@foreach,"
      \restore@macro@i##1% une fois sorti de toutes les boucles, restaurer les macros
      }% si ##1 ne contient pas "/"
      {\save@macro{}##1% sauvegarde la macro
      \doforeach@i% appeler la macro récursive
```

```
        {##3}% mettre en premier le <code>
        ##1% puis la variable ##1 en 2e position
        ##2#1\end@foreach#1% enfin la liste ##2 suivie de ",\end@foreach,"
      \restore@macro{}##1% une fois sorti de toutes les boucles, restaurer la macro
      }%
    \global\advance\cnt@nest-1 % décrémente le compteur d'imbrications
  }%
  % ##1 = code à exécuter, ##2= variable, ##3=valeur courante
  \long\def\doforeach@i##1##2##3#1{%
    \ifx\end@foreach##3% si la fin est atteinte
    \expandafter\gobone\else\expandafter\identity\fi% manger sinon exécuter:
      {\def##2{##3}% fait l'assignation à la variable
      ##1% le code puis
      \allow@recurse{\doforeach@i{##1}##2}% recommencer
      }%
  }%
  % ##1 = code à exécuter, ##2/##3= variables, ##4/##5=valeurs courantes
  \long\def\doforeach@ii##1##2/##3##4/##5#1{%
    \ifx\end@foreach##4% si la fin est atteinte
    \expandafter\gobone\else\expandafter\identity\fi% manger sinon exécuter:
      {\def##2{##4}\def##3{##5}% fait l'assignation des deux variables
      ##1% le code puis
      \allow@recurse{\doforeach@ii{##1}##2/##3}% recommencer
      }%
  }%
  % macro qui, si elle remplace \allow@recurse, annule l'appel récursif
  \long\def\forbid@recurse##1\end@foreach#1{}% tout manger jusqu'à "\end@foreach,"
}
\def\doforeachexit{\global\let\allow@recurse\forbid@recurse}
\catcode`\@12
\defseplist{,}
\doforeach\par\in{a,b,c,y,z}{$\par_i$}\medskip

\doforeach\sujet/\terminaison\in{Je/e,Tu/es,Il/e,Nous/ons,Vous/ez,Ils/ent}
  {\sujet\ programm\terminaison{} en \TeX\par}\medskip

Les voyelles lues sont :
\doforeach\ii\in{a,e,i,o,u,y}{\ii\if o\ii\doforeachexit\fi}.\medskip

\doforeach\xx\in{a,b,c,d,e,f}
  {\doforeach\ii\in{1,2,3,4}{\xx\ii{} \ifnum\ii=3 \doforeachexit\fi}\par}
```

$a_i b_i c_i y_i z_i$

Je programme en TEX
Tu programmes en TEX
Il programme en TEX
Nous programmons en TEX
Vous programmez en TEX
Ils programment en TEX

Les voyelles lues sont : aeio.

a1 a2 a3

■ Exercice 79

La dernière boucle imbriquée montre que `\doforeachexit` sort prématurément de toutes les boucles en cours.

Comment faudrait-il modifier le code pour que `\doforeachexit` ne sorte prématurément

que de la boucle dans laquelle elle a été appelée, sans modifier le déroulement des boucles de niveau supérieur ?

□ **Solution**

Il faut que `\forbid@recurse`, en plus de manger tout jusqu'à la fin de la liste, rende de nouveau possibles les appels récursifs à venir dans d'éventuelles boucles mères. Par conséquent, elle doit redonner à `\allow@recurse` l'équivalence avec `\identity` qu'elle a par défaut. La ligne n° 51 doit donc être

```
\long\def\forbid@recurse##1\end@foreach#1{\global\let\allow@recurse\identity}
```

■

Chapitre 8

Dimensions et boites

Avec TEX, tout ce qui touche de prés ou de loin à la typographie a une étroite proximité avec la géométrie et le placement des boites qui elles-mêmes, entretiennent de proches rapports avec les dimensions.

Le parti a été pris de présenter les dimensions (fixes et étirables) dans un premier temps, quitte à ce que cette théorie, dénuée des objets auxquels elle s'applique, soit un peu rébarbative. Une fois que ces outils et leurs propriétés ont été décrits, les boites et les réglures seront examinées dans un deuxième temps.

8.1. Les dimensions fixes

67 - Règle

Une dimension fixe est utilisée pour mesurer une longueur prise au sens géométrique du terme. TEX dispose de 256 registres de dimension (32 768 avec ε-TEX), chacun capable d'héberger une dimension. Pour faire référence à l'un de ces registres, on utilise la primitive `\dimen` suivie de son numéro (sous la forme d'un ⟨*entier*⟩).

Il est possible de demander l'allocation d'un registre inutilisé avec la macro de plain-TEX `\newdimen\`⟨*macro*⟩. La `\`⟨*macro*⟩, définie en coulisse par la primitive `\dimendef` avec

`\global\dimendef\`⟨*macro*⟩`=`⟨*nombre*⟩

sera par la suite équivalente à `\dimen`⟨*nombre*⟩ où le ⟨*nombre*⟩ est celui du prochain registre libre au moment de l'allocation.

Un ⟨*registre de dimension*⟩ est donc soit une \⟨*macro*⟩ préalablement définie à l'aide de \newdimen, soit « \dimen⟨*nombre*⟩ » où ⟨*nombre*⟩ est le numéro du registre de dimension auquel on souhaite faire référence.

Pour assigner une dimension à un registre de dimension, la syntaxe est

⟨*registre de dimension*⟩= ⟨*dimension*⟩

où le signe = et l'espace qui le suit sont facultatifs. Pour lire la ⟨*dimension*⟩, le développement maximal se met en marche et après développement, une ⟨*dimension*⟩ est :

- une dimension explicitement écrite, c'est-à-dire un ⟨*nombre*⟩ (entier) ou un nombre décimal suivi d'espaces facultatifs et de deux lettres de catcode 11 ou 12 représentant une des unités de dimensions qu'accepte TEX à savoir pt, pc, in, bp, cm, mm, dd, cc, sp, em et ex. Le séparateur décimal peut indifféremment être le point ou la virgule pourvu que son catcode soit 12. La lecture de la dimension s'arrête au premier token qui suit les deux caractères d'unité. Si ce token est un espace, celui est absorbé.
- un ⟨*registre de dimension*⟩. Si ce registre est précédé d'un nombre décimal signé, la dimension obtenue sera celle du registre multipliée par le nombre signé ;
- un ⟨*registre de ressort*⟩ (voir la définition à la section suivante) auquel cas, la dimension retenue est égale à la composante non étirable du registre de ressort. Si ce registre est précédé d'un nombre décimal signé, la dimension obtenue sera la composante fixe du ressort multipliée par le nombre signé.

La valeur absolue maximale d'une ⟨*dimension*⟩ est contenue dans le registre de dimension \maxdimen et vaut 16383.99999pt, ce qui représente 575.83199cm.

Pour ses dimensions, TEX n'est pas avare d'unités. Il en manipule 11 dont voici les définitions :

Unité	Abréviation	Définition	Exemple
point	pt		1 pt = ▮
pica	pc	1 pc = 12 pt	1 pc = ▭
pouce	in	1 in = 72,27 pt	1 in = ▭
gros point	bp	72 bp = 1 in	1 bp = ▮
centimètre	cm	2,54 cm = 1 in	1 cm = ▭
millimètre	mm	10 mm = 1 cm	1 mm = ▯
point didot	dd	1 157 dd = 1 238 pt	1 dd = ▮
cicéro	cc	1 cc = 12 dd	1 cc = ▭
point d'échelle	sp	65 536 sp = 1 pt	10^6 sp = ▭
cadrat	em		1em = ▭
hauteur d'œil	ex		1ex = ▯

Les deux dernières unités dépendent de la police utilisée et de sa taille. Historiquement, 1em est la largeur de la lettre « M », mais le créateur de la fonte peut en décider autrement et donner au em une valeur différente. Par exemple, dans la police en cours dans ce paragraphe, 1em vaut 2.8117mm alors que la largeur de la boite englobante de la lettre « M » vaut 2.94876mm.

La hauteur d'œil, ou ex est censée représenter la hauteur des lettres sans hampe, ni jambage ni signe diacritique. Dans la police en cours, 1ex vaut 1.49104mm et la

hauteur de la boite englobante de la lettre « x » vaut `1.52324mm`.

Les ⟨*registres de dimension*⟩ sont comme les autres registres que nous avons déjà vus (registres d'entiers ou de tokens) : ce ne sont que des sortes de pointeurs vers une zone de mémoire où sont écrits les octets capables, selon un procédé interne à TEX, de stocker le contenu du registre. Par conséquent, dans le code source, un ⟨*registre de dimension*⟩ ne peut être écrit seul *que* lorsque TEX s'apprête à lire une dimension ou lors d'une assignation. Dans les autres cas, la primitive développable `\the`, lorsqu'elle précède le registre, en donne le contenu sous forme de tokens de catcode 12. Lorsqu'on invoque `\the` pour faire remonter le contenu d'un registre de dimension des entrailles de TEX, on obtient toujours une dimension en points où les caractères d'unité « `pt` » ont un catcode de 12. La partie décimale est toujours affichée et éventuellement réduite à un 0, et le séparateur décimal est le point. Le nombre de chiffres est au maximum de 5 après la virgule :

Code n° III-201

```
\newdimen\dimA \newdimen\dimB% alloue deux registres de dimension
a) \dimA=59.5pt
   \the\dimA\qquad% doit afficher 59.5pt
b) \dimA=1.5cm
   \the\dimA\qquad% convertit à l'affichage 1.5cm en pt
c) \dimB=7pt % assigne 7pt à \dimB
   \dimA\dimB% rend \dimA égal à dimB
   \the\dimA\qquad
d) \dimA=6\dimB% rend \dimA égal à 6 fois \dimB
   \the\dimA
```

a) 59.5pt b) 42.67912pt c) 7.0pt d) 42.0pt

Bien qu'étant traduites en points, toutes ces dimensions ne sont que des multiples de l'unité de longueur que TEX manipule en interne, le « point d'échelle » (abréviation « `sp` » pour « scale point »), plus petite dimension que TEX peut manipuler et dont la longueur vaut approximativement $5{,}4 \times 10^{-9}$ mètre (soit 5,4 nanomètres). C'est extrêmement petit, de l'ordre de la dimension d'un petit virus. Cette dimension insécable, bien évidemment invisible à l'œil nu, donne aux calculs sur les longueurs de TEX une grande précision qui restera très inférieure à ce qui peut être décelé à l'œil nu, même après plusieurs arrondis.

8.1.1. Opérations sur les dimensions

Comme pour les entiers, des opérations sont disponibles pour les dimensions via les primitives `\advance` pour l'addition, `\multiply` et `\divide` pour la multiplication et la division par un entier.

Pour calculer $(25\texttt{pt} \div 4 + 1, 72\texttt{pt}) \times 3$, voici comment nous pourrions partir d'une dimension de 25pt, la diviser par 4, ajouter 1.72pt et multiplier le tout par 3 :

Code n° III-202

```
\newdimen\foo
\foo=25pt % assigne 25pt au registre \foo
\divide\foo 4 % le multiplie par 4
\advance\foo1.72pt % lui ajoute 1.72pt
\multiply\foo3 % le multiplie par 3
```

```
\the\foo% affiche la dimension obtenue
```

23.91pt

■ **Exercice 80**

Écrire une macro \removept* qui retire l'unité d'une dimension explicite obtenue avec \the. Si la partie décimale est nulle, elle ne sera pas affichée.

□ **Solution**

Le premier réflexe est d'écrire une macro à argument délimité par « pt » :

```
\def\removept#1.#2pt{#1\ifnum#2>0 .#2\fi}
```

Mais cela ne fonctionnera pas puisque les caractères « pt » du texte de paramètre n'ont pas le catcode 12. Il faut donc ruser un peu pour définir la macro. Nous pouvons par exemple utiliser une macro temporaire pour stocker « \def\removept#1.#2pt » où les délimiteurs « pt » ont un catcode égal à 12. Cette macro temporaire sera définie par un \edef pour développer « \string p » et « \string t » qui donnent les caractères « pt » de catcode 12 :

Code n° III-203

```
\begingroup% ouvrir un groupe
  \edef\temp{\endgroup\def\noexpand\removept##1.##2\string p\string t}%
\temp{#1\ifnum#2>0 .#2\fi}% et le fermer avant de définir \removept
\newdimen\foo
a) \foo=15pt \expandafter\removept\the\foo\qquad
b) \foo=3.14pt \expandafter\removept\the\foo
```

a) 15 b) 3.14

L'astuce consiste à inclure le \endgroup dans le texte de remplacement de \temp de telle sorte que lorsque \temp se développe puis s'exécute, ce \endgroup provoque la destruction de cette macro temporaire avant même d'avoir fini de lire son texte de remplacement.

Il est naturel de créer la macro \dimtodec* qui transforme un registre de dimension en nombre décimal :

Code n° III-204

```
\def\dimtodec{\expandafter\removept\the}
\newdimen\foo
a) \foo=15pt \dimtodec\foo \qquad
b) \foo=3.14pt \dimtodec\foo
```

a) 15 b) 3.14

■

8.1.2. La primitive \dimexpr

La primitive \dimexpr de ε-TEX est aux calculs sur dimensions ce que \numexpr est aux calculs sur les entiers. Elle rend possibles des calculs sur les dimensions de TEX sans passer par les registres de dimension et les primitives \advance, \multiply et \divide qui donnent un code assez fastidieux.

68 - Règle

La primitive \dimexpr de ε-TEX doit être suivie d'un enchainement d'opérations sur des dimensions qui est évalué selon les règles mathématiques habituelles : parenthèses, opérations (+, *, /), priorités. Cette primitive possède les propriétés suivantes :

- tout comme \numexpr, \dimexpr amorce un développement maximal pour évaluer l'⟨*expression*⟩ qui la suit. L'⟨*expression*⟩ prend fin au premier caractère qui ne peut faire partie d'un calcul sur les dimensions. La primitive \relax peut marquer la fin d'une ⟨*expression*⟩ et dans ce cas, elle est absorbée par \dimexpr ;
- les espaces dans l'⟨*expression*⟩ sont ignorés ;
- dans l'⟨*expression*⟩, une ⟨*dimension*⟩ ne peut être multipliée ou divisée (avec * ou /) que par un *entier*. Dans ce cas, l'entier *doit* être l'opérateur c'est-à-dire qu'il doit suivre la dimension. On doit donc écrire

  ⟨*dimension*⟩*⟨*entier*⟩ ou ⟨*dimension*⟩/⟨*entier*⟩

- une ⟨*expression*⟩ évaluée par \dimexpr est du même type qu'un registre de dimension, il s'agit donc d'une représentation *interne* d'une dimension. On peut convertir cette représentation interne en caractères de catcode 12 en faisant précéder \dimexpr de la primitive développable \the.

Voici quelques exemples où la primitive \dimexpr est mise à contribution :

Code n° III-205

```
a) \the\dimexpr 1cm + 0.5cm\relax \qquad
b) \the\dimexpr 1pt + 1pt\relax \qquad
c) \the\dimexpr 1pt + 2pt * 3\relax\qquad
d) \the\dimexpr (1pt + 2pt) * 3\relax\qquad
e) \the\dimexpr (1.2pt + 0.8pt) * 5\relax\qquad
f) \newdimen\foo \foo=15pt
   \the\dimexpr\foo-(\foo + 1pt) / 4\relax
```

a) 42.67911pt b) 2.0pt c) 7.0pt d) 9.0pt e) 10.0pt f) 11.0pt

8.1.3. Les limites du calcul sur les dimensions

Tout semble se passer pour le mieux et il serait tentant de penser qu'une fois libérée de son unité en pt, une dimension se comporte exactement comme un nombre décimal tandis que de son côté, \dimexpr joue le rôle de machine à calculer. La simple addition ci-dessous va tempérer ce jugement :

Code n° III-206

```
\the\dimexpr0,7pt + 0.4pt\relax
```

1.09999pt

Que se passe-t-il et pourquoi le simple calcul « `0,7pt + 0.4pt` » ne donne-t-il pas comme résultat `1.1pt` ? Les calculs de dimensions seraient-ils bogués ? La réponse est bien évidemment non, mais c'est le revers de la médaille : puisque toutes les dimensions sont converties en « sp » en interne pour effectuer les calculs

puis sont à nouveau converties en « pt » pour le résultat, des arrondis sont inévitablement faits lors de ces deux conversions. Il faut donc s'attendre à ce que ces arrondis viennent dans certains cas se cumuler défavorablement. Voici l'explication de l'erreur constatée :

- `0,7pt` vaut $0,7 \times 65\,536 = 45\,875{,}2$ sp qui est arrondi à 45 875 sp ;
- `0.4pt` vaut $0.4 \times 65\,536 = 26\,214{,}4$ sp qui est arrondi à 26 214 sp ;
- en tenant compte des arrondis, la somme vaut 45 875 + 26 214 = 72 089 sp ;
- convertie en pt, cette somme vaut $72\,089 \div 65536 = 1{,}099\,990\,844\,73$ et TEX donne 5 chiffres après la virgule ce qui fait bien 1,099 99 pt.

Le résultat attendu, `1.1pt` vaut $1{,}1 \times 65536 = 72\,089{,}6$ sp qui serait arrondi par TEX à 72 090 sp.

Si une dimension est considérée comme une mesure de ce qui doit être affiché, ces erreurs d'arrondis sont absolument insignifiantes puisqu'invisibles à l'œil nu ; insistons sur le fait qu'il ne s'agit ici que d'une minuscule erreur d'`1sp` c'est-à-dire de quelques nanomètres ! Mais pour le programmeur, c'est la douche froide et c'est l'assurance que les dimensions ne peuvent pas jouer le rôle de nombres décimaux pris en tant que nombres *mathématiques*, car une erreur sur une simple addition est inacceptable. On peut donc affirmer que TEX ne dispose de rien qui pourrait tenir lieu de nombres décimaux comme en disposent les autres langages. Bien évidemment, on peut *programmer* le fonctionnement de nombres décimaux (en virgule fixe ou flottante) avec les outils de base de TEX que sont les entiers et les 4 opérations. Construire des opérations sur ces nombres est extrêmement fastidieux [1]. Il faut mentionner la macro purement développable `\decadd`⋆, programmée en annexe (voir page 513), qui permet d'additionner de façon *exacte* deux nombres décimaux signés.

Il est primordial de comprendre qu'une dimension et un nombre entier ne sont que deux facettes d'une même chose : puisque tous les calculs se font en nombre entier de points d'échelle, une dimension en TEX n'est en réalité que le nombre *entier* de sp qu'elle représente !

69 - Règle

Toutes les dimensions fixes que TEX manipule sont converties en interne en le nombre *entier* de points d'échelle (sp) qu'elles représentent. Une dimension s'apparente donc à un entier.

Si TEX est à un endroit où il s'attend à lire un ⟨*nombre*⟩ entier et qu'il lit un ⟨*registre de dimension*⟩, il convertit la dimension contenue dans le registre en « entier contraint » qui est le nombre de sp qu'elle représente. Cet entier contraint est le ⟨*nombre*⟩ qui sera pris en compte.

1. L'extension « fp » (pour « fixed point ») a relevé ce défi. Cette extension est programmée en TEX et utilise les compteurs pour parvenir à ses fins. Elle définit et manipule des nombres en virgule *fixe* avec une partie entière et une partie décimale de 18 chiffres chacune et effectue toutes les opérations arithmétiques et scientifiques habituelles sur ces nombres en virgule fixe (puissances, exponentielles, logarithmes, lignes trigonométriques, tests, nombres pseudoaléatoires, etc.). Ainsi, il est très facile de calculer que

$$\frac{e^4 \ln\left(1+\cos\frac{\pi}{7}\right)}{1-3\arctan(0.35)} + 4\sqrt[3]{5+\sqrt{2}} \approx -3491.19785321755\mathbf{4299238}$$

Du fait d'arrondis successifs dans cette expression complexe, les décimales en gras sont fausses.

Constatons-le avec ce code où la primitive \number qui, attendant un nombre entier, force la conversion de la dimension en entier contraint :

Code n° III-207

```
a) \newdimen\foodim \foodim=1cm \number\foodim\relax\qquad
b) \number\dimexpr 0.7pt + 0.4pt\relax\qquad
c) \number\dimexpr 1.1pt\relax
```

a) 1864679 b) 72089 c) 72090

■ Exercice 81

La macro \for, programmée à la page 175, ne peut manipuler que des nombres *entiers*.

Sur le modèle de \for, créer une macro \FOR* où les nombres (borne mini, maxi et incrément) peuvent être décimaux.

On utilisera le test \ifdim, de syntaxe

```
\ifdim⟨dimension1⟩⟨signe de comparaison⟩⟨dimension2⟩
    ⟨code vrai⟩
\else
    ⟨code faux⟩
\fi
```

qui est aux ⟨*dimensions*⟩ ce que le test \ifnum est aux entiers.

□ Solution

Il y a peu de changements à faire. Il faudra remplacer les tests \ifnum par \ifdim et rajouter l'unité pt lorsque c'est nécessaire.

Code n° III-208

```
\catcode`\@11
\def\FOR#1=#2to#3\do#4#{%
  \ifempty{#4}
    {\let\FOR@increment\z@}
    {\edef\FOR@increment{\the\dimexpr#4pt\relax}}% lit et normalise l'argument optionnel
  \ifdim\FOR@increment=\z@% s'il est nul,
    \edef\FOR@increment{% le redéfinir à -1pt (si #3<#2) et 1pt sinon
      \ifdim\dimexpr#3pt-#2pt\relax<\z@ -1\else 1\fi pt
    }% \FOR@increment vaut donc 1 ou -1
  \fi
  \ifdim\dimtodec\dimexpr#3pt-#2pt\relax\dimexpr\FOR@increment\relax<\z@
    \expandafter\gobone% si argument optionnel incompatible, manger le {<code>}
  \else
    \edef#1{\dimtodec\dimexpr#2pt\relax}% initialise la \<macro>
    \edef\macro@args{% définit et développe les arguments à passer à \FOR@i
      %#1=nom de la macro récursive :
      \expandafter\noexpand\csname FOR@ii@\string#1\endcsname
      \ifdim\FOR@increment<\z@ <\else >\fi% #2=signe de comparaison
      {\FOR@increment}% #3=incrément
      \noexpand#1% #4=\<macro>
      {\the\dimexpr#3pt\relax}% #5=dimension n2
    }%
    \antefi% externalise la ligne ci-dessous de la portée du test
    \expandafter\FOR@i\macro@args% appelle \FOR@i avec les arguments définis ci-dessus
  \fi
}

% #1=nom de la macro récursive de type "\FOR@ii@\<macro>"
```

```
% #2=signe de comparaison  % #3=incrément
% #4=\<macro>  % #5=dimension n2  % #6=<code> à exécuter
\long\def\FOR@i#1#2#3#4#5#6{%
  \def#1{% définit la sous macro récursive
    \unless\ifdim#4pt#2#5\relax% tant que la \<macro> variable n'a pas dépassé n2
      \afterfi{% rendre la récursivité terminale
        #6% exécute le code
        \edef#4{\dimtodec\dimexpr#4pt+#3\relax}% incrémente la \<macro>
        #1% recommence
      }%
    \fi
  }%
  #1% appelle la sous-macro récursive
}%
\def\exitFOR#1{% #1=\<macro> correspondant à la boucle de laquelle on veut sortir
  \defname{FOR@ii@\string#1}{}%
}

\def\ifexitFOR#1{% envoie vrai si on est prématurément sorti de la boucle de \<macro> #1
  % si la macro récursive est \empty
  \expandafter\ifx\csname FOR@ii@\string#1\endcsname\empty
    \expandafter\firstoftwo% c'est qu'on est sortir prématurément, renvoyer "vrai"
  \else
    \expandafter\secondoftwo% sinon, renvoyer "faux"
  \fi
}
\catcode`\@=12
a) \FOR\xx=1.5 to -5\do-1{"\xx" }\par

b) \FOR\ii=1 to 2.742\do.25{"\ii" }\par

c) \FOR\ii=0 to 1\do0.1{"\ii" \ifdim\ii pt>0.5pt \exitFOR\ii\fi}
```

a) "1.5" "0.5" "-0.5" "-1.5" "-2.5" "-3.5" "-4.5"
b) "1" "1.25" "1.5" "1.75" "2" "2.25" "2.5"
c) "0" "0.1" "0.20001" "0.30002" "0.40002" "0.50003"

La ligne n° 11 mérite tout de même quelques explications. Il s'agit de trouver le signe de

$$(\#3 - \#2) \times \texttt{\textbackslash for@increment}$$

Pour calculer cette expression sous forme de dimension, nous avons tout d'abord transformé `#3-#2` en un décimal sans unité avec la macro `\dimtodec`. La macro `\for@increment` est transformée en une dimension de type ⟨*registre*⟩ avec `\dimexpr`. Le test `\ifdim` voit donc ⟨*x*⟩⟨*registre de dimension*⟩ où ⟨*x*⟩ est un décimal signé. Comme l'explique la règle vue précédemment, le tout forme une ⟨*dimension*⟩ égale au produit de `#3-#2` par `\for@increment`.

Comme le montre le cas c, additionner à chaque itération l'incrément à la dimension précédente conduit parfois à des erreurs d'arrondi gênantes. Pour une addition décimale *exacte*, voir en annexe la macro `\decadd`, page 513. ■

8.1.4. Étendre les calculs sur les dimensions

Puisqu'il n'est pas question d'écrire une extension aussi complexe que `fp`, nous allons accepter les erreurs d'arrondis dues à la méthode de calcul sur les dimensions. Aussi, nous allons confondre dimension en `pt` sans unité et nombre décimal ; en

effet, passer de l'une à l'autre est facile. Il suffit d'appeler la macro \dimtodec pour le sens *dimension vers décimal* ou d'ajouter l'unité pt pour le sens inverse.

Nous allons essayer de tirer parti de \dimexpr et des règles que nous avons vues pour étendre un peu les possibilités de calcul de TEX sur les nombres décimaux. Si l'addition et la soustraction opèrent sur *deux* nombres décimaux, il n'en est pas de même pour la multiplication et la division qui ne sont capables de multiplier ou diviser par un *entier*.

Multiplier deux décimaux

Essayons tout d'abord de programmer une macro \decmul* qui permet de multiplier deux nombres *décimaux*, chacun se trouvant dans un argument.

Nous allons exploiter le fait que ⟨*décimal*⟩⟨*registre de dimension*⟩ est vue comme une dimension égale au produit des deux quantités.

Code n° III-209

```
\newdimen\foo
\foo=15.2pt \foo=1.5\foo \the\foo% vaut 15.2*1.5
```

22.79999pt

Aux erreurs d'arrondi près auxquelles nous nous sommes résignés, le résultat est correct. Voici donc comment programmer la macro \decmul :

Code n° III-210

```
\catcode`\@11
\newdimen\dim@a
\def\decmul#1#2{%
  \dim@a=#2pt    % range la dimension #2pt dans le registre de dimension
  \dim@a=#1\dim@a% multiplier le registre par le décimal #1
  \dimtodec\dim@a% convertir la dimension en décimal
}
\catcode`\@12
a) \decmul{15.2}{1.5}\qquad
b) \decmul{48.2}{.375}
```

a) 22.79999 b) 18.075

■ **Exercice 82**

La macro \decmul programmée ci-dessus n'est pas purement développable. Utiliser la primitive \dimexpr pour qu'elle le devienne.

□ **Solution**

La méthode va être de donner à \dimtodec non pas un vrai registre de dimension, mais une expression évaluée par \dimexpr, puisque cette primitive est du même type qu'un registre de dimension :

\dimtodec\dimexpr⟨*expression*⟩\relax

En ce qui concerne l'⟨*expression*⟩, nous allons utiliser la même méthode vue ci-dessus, à savoir #1⟨*registre*⟩ où le ⟨*registre*⟩ sera remplacé par \dimexpr#2\relax. Cela donne :

Code n° III-211

```
\def\decmul#1#2{\dimtodec\dimexpr#1\dimexpr#2pt\relax\relax}

a) \decmul{15.2}{1.5}\qquad
b) \edef\foo{\decmul{48.2}{.375}}\meaning\foo
```

a) 22.79999 b) macro:->18.075

■

Diviser par un décimal

Si nous souhaitons écrire une macro analogue `\decdiv*{`⟨*nba*⟩`}{`⟨*nbb*⟩`}` qui effectue la division du décimal ⟨*nba*⟩ par le décimal ⟨*nbb*⟩, l'astuce consiste à évaluer avec `\numexpr` l'entier suivant :

$$\frac{\langle nba\rangle\texttt{pt} \times \texttt{1pt}}{\langle nba\rangle\texttt{pt}}$$

Comme `\numexpr` s'attend à des opérations sur des entiers, chaque dimension, préalablement mise sous forme de registre avec `\dimexpr`, sera convertie en entier contraint. Le risque de dépassement de l'entier maximal au numérateur est improbable, car lorsque `\numexpr` (tout comme `\dimexpr`) effectue une multiplication immédiatement suivie d'une division, les calculs internes se font sur 64 bits et non sur 32.

Une fois cet entier calculé, nous lui donnerons l'unité `sp` et évaluerons le tout avec `\dimexpr`. Le résultat, expurgé de son unité avec `\dimtodec`, donnera le quotient cherché.

Pour éviter un `\dimexpr` inutile, la dimension `1pt`, qui vaut `65536sp`, a été directement convertie en l'entier contraint `65536` :

Code n° III-212

```
\def\decdiv#1#2{% divise le décimal #1 par le décimal #2
  \dimtodec
  \dimexpr
    \numexpr
      \dimexpr #1pt \relax * 65536 / \dimexpr #2pt \relax
    \relax
    sp
  \relax
}

1) \decdiv{4.5}{0.075}\qquad% doit donner 60
2) \decdiv{8}{0.1}\qquad% doit donner 80
3) \decdiv{3.14}{1.6}\qquad% doit donner 1.9625
4) \decdiv{687.59829}{5.29871}\qquad% doit donner 129.76706
4) \edef\foo{\decdiv{0.37}{2.5}}\foo% doit donner 0.148
```

1) 60.00244 2) 79.99512 3) 1.9625 4) 129.76721 4) 0.148

Ici encore, il ne s'agit que d'arithmétique TEXienne, c'est-à-dire que comme avec `\decmul`, les résultats sont entachés d'erreurs d'arrondis et donc, ces macros ne pourront être utilisées que pour des calculs de dimensions et non pas pour des calculs sur des nombres pris en tant qu'entités mathématiques.

■ **Exercice 83**

Écrire une macro \convertunit* de syntaxe

\convertunit{⟨*dimension*⟩}{⟨*unité*⟩}

qui convertit la ⟨*dimension*⟩ dans l'⟨*unité*⟩ spécifiée par les deux caractères caractérisant chaque unité, et d'afficher le résultat sans unité. La macro devra être purement développable.

□ **Solution**

Nous allons procéder comme nous l'avons fait avec la macro \decdiv : nous allons mettre \numexpr à contribution pour transformer en entier contraint toutes les dimensions et multiplier la ⟨*dimension*⟩ par 1pt et diviser le tout par 1⟨*unité*⟩ :

Code n° III-213

```
\def\convertunit#1#2{%
  \dimtodec
  \dimexpr
    \numexpr
      \dimexpr #1 \relax * 65536 / \dimexpr 1#2 \relax
    \relax
    sp
  \relax
}

a) \convertunit{15cm}{mm}\qquad
b) \convertunit{9,14in}{cm}\qquad
c) \convertunit{100000sp}{mm}\qquad
d) \convertunit{160.5pt}{cm}\qquad
e) \edef\foo{\convertunit{2,5cm}{cc}}\meaning\foo
```

a) 150.0008 b) 23.2156 c) 0.53629 d) 5.64093 e) macro:->5.53983

■

8.2. Les dimensions étirables ou ressorts

8.2.1. Qu'est-ce qu'un ressort ?

Intéressons-nous maintenant aux ressorts qui ont vocation à remplir des zones susceptibles d'avoir des tailles variables (souvent pour des raisons typographiques).

70 - Règle

Un ressort est une dimension assortie de composantes étirables optionnelles (éventuellement infinies) de sorte qu'il peut, selon le contexte et sa définition, se comprimer ou s'étirer. TEX offre 256 registres de ressorts et ce nombre monte à 32 768 avec ε-TEX. La primitive \skip suivie d'un ⟨*nombre*⟩ permet d'accéder à un registre spécifique par son numéro.

Plain-TEX fournit la macro \newskip\⟨*macro*⟩ qui met à profit la primitive \skipdef selon cette syntaxe

\skipdef\⟨*macro*⟩=⟨*nombre*⟩

pour rendre cette \⟨*macro*⟩ équivalente à \skip⟨*nombre*⟩, où le ⟨*nombre*⟩ est celui du prochain registre libre. Dès lors, un ⟨*registre de ressort*⟩ est soit \skip suivi d'un numéro, soit une \⟨*macro*⟩ préalablement définie par \newskip.

Pour assigner un ressort à un ⟨*registre de ressort*⟩, il faut écrire :

⟨*registre de ressort*⟩= ⟨*ressort*⟩

où le signe « = » et l'espace qui le suit sont optionnels. Pour lire le ⟨*ressort*⟩, le développement maximal se met en marche. Un ⟨*ressort*⟩ est une ⟨*dimension*⟩ éventuellement suivie d'une composante d'étirement optionnelle de la forme « `plus` ⟨*étirement*⟩ » puis d'une composante de compression optionnelle de la forme « `minus` ⟨*étirement*⟩ ».

Si elles sont présentes toutes les deux, ces deux composantes optionnelles doivent *nécessairement* être déclarées dans cet ordre. Si elles ne sont pas spécifiées, les composantes étirables valent `0pt` par défaut.

Un ⟨*étirement*⟩ est :

- soit une composante fixe qui est une ⟨*dimension*⟩ comme « `1.5pt` » ou « `0.25cm` » ;
- soit une composante infinie qui prend la forme d'un coefficient d'infini suivi de « `fil` », « `fill` » ou « `filll` », où ces 3 mots-clé représentent des infinis de *forces* différentes : « `filll` » est infiniment plus grand que « `fill` », quels que soient les coefficients qui les précèdent et de la même façon, « `fill` » est infiniment plus grand que « `fil` ».

  Le coefficient d'infini est un décimal signé dont la valeur absolue peut prendre toutes les valeurs entre 0 et `16383.99999`.

À la manière de `\dimexpr` qui fait des calculs sur les dimensions, la primitive `\glueexpr` effectue des calculs sur les ressorts :

Code n° III-214

```
\the\glueexpr 5pt plus 2pt minus 1.5pt + 7pt plus0.5pt minus 3pt\relax \par
\the\glueexpr 25pt plus2fil + 35pt plus 0.1fill\relax
```

12.0pt plus 2.5pt minus 4.5pt
60.0pt plus 0.1fill

Envisageons maintenant quelques exemples pour mieux comprendre comment les ressorts fonctionnent et quelles sont leurs « forces » relatives.

Intéressons-nous tout d'abord aux composantes d'étirement finies. Le ressort « `10pt plus1.5pt minus 2.5pt` » a une longueur naturelle de 10 pt mais peut s'allonger jusqu'à 11,5 pt et se comprimer jusqu'à 7,5 pt. De la même façon, le ressort « `0pt minus5pt` » a une longueur naturelle nulle, mais peut se comprimer jusqu'à une longueur négative de −5 pt.

Venons-en aux composantes d'étirement infinies. Supposons qu' un ressort R_1 soit égal à « `1.5cm plus2fill` ». Il a une longueur naturelle de 1,5 cm et peut s'allonger autant que nécessaire comme le spécifie sa composante étirable. Si un autre ressort R_2 est défini par « `2.5cm plus3fill` » et si nous demandons à ces deux ressorts de remplir un espace de 10 cm, la longueur à combler est de 6 cm puisque la somme des deux longueurs naturelles vaut 4 cm. Le ressort R_1 s'allongera de 2/5 la longueur à combler et R_2 de 3/5 de cette longueur, comme le spécifient les coefficients d'infini. R_1 mesurera $1.5 + 6 \times 2/5 = 3{,}9$ cm alors que R_2 mesurera $2.5 + 6 \times 3/5 = 6{,}1$ cm. En voici l'illustration ci-dessous où les dimensions naturelles

sont en traits pleins et les étirements en pointillés. Si aucune opération dans l'impression de ce livre n'a provoqué d'altération d'échelle, les dimensions données ci-dessus doivent exactement se retrouver dans ce schéma :

Enfin, si un ressort R_3 égal à « `3cm plus 20fil` » et un ressort R_4 égal à « `2cm plus 0.1fill` » doivent combler un espace de 20 cm, seul R_4 s'allongera pour mesurer 17 cm car son infini « `fill` » est infiniment plus grand que « `fil` ».

8.2.2. Insérer des espaces de dimensions choisies

Insertion d'espaces insécables

71 - Règle

Pour insérer une espace insécable, on utilise la primitive \kern suivie d'une ⟨*dimension*⟩ qui sera celle de l'espace insérée.

L'espace sera insérée dans le mode en cours : elle sera horizontale si TEX est en mode horizontal à ce moment-là et sera verticale s'il est en mode vertical.

L'espace ainsi créé est *insécable* car aucune coupure de ligne ou de page ne pourra s'y faire.

Code n° III-215

```
foo% les caractères font entrer TeX en mode horizontal
\kern0.5cm % espace insérée en mode horizontal
bar\par% \par fait passer en mode vertical
\kern0.5cm % espace insérée en mode vertical
boo
```

foo bar

boo

L'espace verticale entre la frontière basse de « bar » et la frontière haute de « boo » n'est pas `0.5cm`, car le ressort d'interligne vient s'ajouter à la dimension insécable insérée par \kern.

Insertion d'espaces sécables

À la différence des espaces insécables, un espace est dit « *sécable* » s'il peut céder sa place à une coupure (de ligne ou de page). Le verbe « céder » suggère que si une coupure est faite, l'espace est retiré de la liste en cours et est remplacé par une coupure.

72 - Règle

La primitive \hskip, suivie d'un ⟨*ressort*⟩ insère une espace (dont les dimensions sont celles permises par le ressort) dans la liste horizontale en cours. La primitive \vskip en est le pendant pour le mode vertical.

Si TEX rencontre une de ces deux primitives et ne se trouve pas dans le mode qu'elles exigent, alors le mode courant se termine et TEX passe dans le mode requis.
Les espaces ainsi insérées sont sécables, c'est-à-dire susceptibles d'être remplacées par des coupures de ligne ou de pages. De plus, elles sont ignorées lorsqu'elles se trouvent immédiatement avant la fin du paragraphe pour `\hskip` ou de la fin de la page pour `\vskip`.

Il est utile de savoir qu'en mode horizontal, le token espace (de catcode 10 et de code de caractère 32) se comporte comme un ressort : il a une dimension naturelle et peut, dans une certaine mesure, se comprimer ou se dilater (lire page 471).

Certaines primitives sont en réalité des ressorts. Il est important de distinguer deux types de « ressorts-primitives » :

- ceux qui sont *modifiables* et qui se comportent comme une macro définie avec `\newskip`. Ces ressorts-primitives sont susceptibles de recevoir un ⟨*ressort*⟩ par assignation. La plupart de ces ressorts-primitives concernent des réglages typographiques ;
- ceux, non modifiables, dont la valeur est prédéfinie et invariable.

Le tableau ci-dessous, sans être exhaustif, liste quelques ressorts utilisés par TEX, qu'ils soient de simples macros définies avec `\newskip` (partie du haut), des ressorts-primitives modifiables (écrits en gras dans la partie du milieu) ou ressorts-primitives non modifiables (précédés d'une « ∗ » dans la partie basse).

Dans chaque catégorie, la valeur par défaut assignée par plain-TEX ou le comportement (pour les ressorts-primitives non modifiables) est donné dans la colonne « Définition » tandis que le mode dans lequel le ressort est utilisé est donné dans la colonne de droite.

Nom	Définition	Mode
`\hideskip`	= `-1000pt plus 1fill`	h
`\centering`	= `0pt plus 1000pt minus 1000pt`	h
`\z@skip`	= `0pt plus0pt minus0pt`	h ou v
`\smallskipamount`	= `3pt plus 1pt minus 1pt`	v
`\medskipamount`	= `6pt plus 2pt minus 2pt`	v
`\bigskipamount`	= `12pt plus 4pt minus 4pt`	v
`\baselineskip`	= `12pt`	v
`\lineskip`	= `1pt`	v
`\leftskip`	= `0pt`	h
`\rightskip`	= `0pt`	h
`\parskip`	= `0pt plus 1pt`	v
`\parfillskip`	= `0pt plus 1fil`	h
*`\hfil`	≡ `\hskip 0pt plus 1fil`	h
*`\hfilneg`	≡ `\hskip 0pt plus -1fil`	h
*`\hfill`	≡ `\hskip 0pt plus 1fill`	h
*`\vfil`	≡ `\vskip 0pt plus 1fil`	v
*`\vfilneg`	≡ `\vskip 0pt plus -1fil`	v
*`\vfill`	≡ `\vskip 0pt plus 1fill`	v
*`\hss`	≡ `\hskip 0pt plus1fil minus1fil`	h
*`\vss`	≡ `\vskip 0pt plus1fil minus1fil`	v

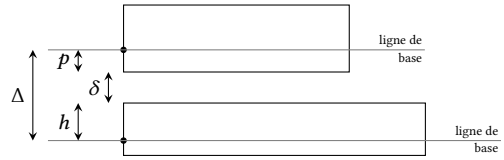
8.2.3. Les ressorts typographiques

Intéressons-nous aux ressorts-primitives écrits en gras dans le tableau précédent. Ces ressorts sont examinés lorsque le paragraphe est composé, c'est-à-dire lorsque \par est exécuté. Il est donc inutile de modifier plusieurs fois ces ressorts dans le même paragraphe puisque seule la dernière valeur sera prise en compte.

Cela implique en particulier que si un paragraphe est composé dans un groupe (semi-simple ou non) dans lequel certains de ces ressorts sont modifiés, il est nécessaire de composer le paragraphe *avant* de fermer le groupe. En écrivant \par avant la fin du groupe, on s'assure que les valeurs des ressorts, modifiées dans le groupe, seront bien prises en compte. En revanche, rejeter le \par hors du groupe constitue une erreur, car les valeurs, restaurées trop tôt à ce qu'elles étaient avant les modifications, rendent inutiles ces modifications internes au groupe.

Ressort d'interligne

Les boites contenant deux lignes consécutives d'un paragraphe ne sont généralement pas jointives. Le schéma ci-dessous illustre la situation : la distance entre les lignes de base des deux lignes consécutives est notée Δ tandis que la distance entre les frontières des deux boites est notée δ.



Plus généralement, si des boites sont empilées les unes au-dessous des autres en mode vertical, un ressort-primitive vertical « \baselineskip » est inséré entre elles. Ce ressort-primitive, appelé « ressort d'interligne », mesure l'espace insérée entre les *lignes de base* de ces boites.

Si la géométrie des boites (précisément la quantité $p + h$) fait que δ devient strictement inférieur à une certaine limite stockée dans la dimension-primitive \lineskiplimit, alors les deux boites sont empilées verticalement de telle sorte que δ soit égal au ressort-primitive \lineskip. Mathématiquement, si

$$\Delta - (p + h) < \texttt{\lineskiplimit}$$

alors l'insertion de \baselineskip entre les lignes de base est abandonnée et le ressort-primitive \lineskip est inséré *entre les frontières des boites*.

Plain-TEX effectue les initialisations suivantes :

```
\baselineskip=12pt    \lineskiplimit=0pt    \lineskip=1pt
```

La macro \nointerlineskip doit être appelée en mode vertical et annule l'insertion du prochain ressort d'interligne entre deux boites. Cette macro est à « un seul coup » et devra être appelée à nouveau si plus tard, on souhaite annuler l'insertion du ressort d'interligne entre deux autres boites.

Code n° III-216

```
1 Une première ligne\par
2 \nointerlineskip% n'insère pas de ressort d'interligne ici
3 Un second paragraphe constitué de plusieurs lignes.
```

```
Un second paragraphe constitué de plusieurs lignes.
Un second paragraphe constitué de plusieurs lignes.
\par% le ressort d'interligne sera inséré ici
Une dernière ligne
```

Une première ligne
Un second paragraphe constitué de plusieurs lignes. Un second paragraphe constitué de plusieurs lignes. Un second paragraphe constitué de plusieurs lignes.
Une dernière ligne

La macro `\offinterlineskip`, désactive durablement l'insertion du ressort d'interligne. Il est prudent de l'utiliser dans un groupe afin d'en limiter la portée.

Code n° III-217

```
\begingroup
\offinterlineskip
La macro \litterate-\offinterlineskip-, en modifiant de façon appropriée les trois
primitives \litterate-\baselineskip-, \litterate-\lineskip- et
\litterate-\lineskiplimit-, rend les boites consécutives jointives.

On peut constater dans ces deux paragraphes où \litterate-\offinterlineskip- a été
appelée, que les lignes sont placées verticalement au plus près les unes des autres ce
qui rend la lecture très pénible et démontre que le ressort d'interligne est une
nécessité typographique !\par
\endgroup
Désactiver le ressort d'interligne ne se justifie que lorsque l'on doit composer
des boites contenant autre chose que du texte, sauf à vouloir des effets
typographiques spéciaux.
```

La macro `\offinterlineskip`, en modifiant de façon appropriée les trois primitives `\baselineskip`, `\lineskip` et `\lineskiplimit`, rend les boites consécutives jointives.
On peut constater dans ces deux paragraphes où `\offinterlineskip` a été appelée, que les lignes sont placées verticalement au plus près les unes des autres ce qui rend la lecture très pénible et démontre que le ressort d'interligne est une nécessité typographique !
Désactiver le ressort d'interligne ne se justifie que lorsque l'on doit composer des boites contenant autre chose que du texte, sauf à vouloir des effets typographiques spéciaux.

Les deux cas les plus courants où la quantité $p + h$ est élevée sont :

1. la boite du haut est très profonde et p est grand (boite de type `\vtop`) ;
2. la boite du bas est très haute et h est grand (boite de type `\vbox`).

En voici l'illustration où nous allons construire une boite du haut profonde et où nous augmentons `\baselineskip` afin de mieux visualiser le problème :

Code n° III-218

```
\baselineskip=12pt
début \vtop{\hbox{ligne du haut ligne du haut ligne du haut}
\hbox{ligne du bas ligne du bas ligne du bas}
}
\par
Et la suite du texte suite du texte suite du texte
```

début ligne du haut ligne du haut ligne du haut
ligne du bas ligne du bas ligne du bas
Et la suite du texte suite du texte suite du texte

On constate que l'espace vertical entre la « ligne du bas » de la `\vtop` et la ligne suivante n'est pas `\baselineskip`. En effet, `\lineskip` a été inséré entre la frontière de la `\vtop` et la première ligne du paragraphe qui suit. Pour rétablir un

espacement correct, il faut mentir à TEX sur la profondeur de la boite précédente. En mode vertical, la primitive \prevdepth (qui mesure la longueur p du schéma), ayant le type d'une dimension, contient la profondeur de la précédente boite placée dans la liste verticale. Après être sorti de la \vtop, cette valeur est élevée du fait de la profondeur de la boite. L'astuce consiste à sauvegarder p dans une macro à la fin de la \vtop de telle sorte que cette macro contienne la profondeur *de la dernière ligne* de la boite. Comme on ne peut accéder à \prevdepth qu'en mode vertical, il suffit donc après le \par d'assigner à \prevdepth la valeur sauvegardée.

Code nº III-219

```
\baselineskip=12pt
début \vtop{\hbox{ligne du haut ligne du haut ligne du haut}
\hbox{ligne du bas ligne du bas ligne du bas}
\xdef\sprevdepth{\the\prevdepth}% sauvegarde la valeur de \prevdepth
}
\par\prevdepth=\sprevdepth\relax% ment sur la boite précédente
Et la suite du texte suite du texte suite du texte
```

début ligne du haut ligne du haut ligne du haut
ligne du bas ligne du bas ligne du bas
Et la suite du texte suite du texte suite du texte

Les ressorts de paragraphe

Avant de commencer un paragraphe, TEX ajoute à la liste verticale l'espace mesuré par le ressort-primitive \parskip (il n'est pas ajouté si la liste verticale est vide, c'est-à-dire au début d'une page ou d'une boite verticale). Ce ressort vient donc s'additionner au ressort d'interligne vu précédemment. Plain-TEX procède à l'assignation suivante :

```
\parskip= 0pt plus 1pt
```

Après la fin de la dernière ligne d'un paragraphe, TEX insère horizontalement le ressort-primitive \parfillskip. Plain-TEX demande à ce que

```
\parfillskip= 0pt plus1fil
```

ce qui signifie qu'un ressort d'étirement infini est inséré en fin de paragraphe. L'effet d'un tel ressort est de « pousser » le texte de la dernière ligne de façon à ce qu'il ne se justifie pas. On peut bien sûr modifier ce ressort pour obtenir des effets typographiques comme dans ce paragraphe où ce ressort a été rendu nul sans étirement afin que la dernière ligne soit justifiée. Cet effet doit être utilisé avec parcimonie, car les espaces intermots de cette dernière ligne, beaucoup trop larges ici, démontrent que cet effet peut parfois être très inesthétique.

Afin de bien voir la modification de la composition du paragraphe que cela entraine, voici le paragraphe précédent avec un ressort de fin de paragraphe ayant sa valeur par défaut :

ce qui signifie qu'un ressort d'étirement infini est inséré en fin de paragraphe. L'effet d'un tel ressort est de « pousser » le texte de la dernière ligne de façon à ce qu'il ne se justifie pas. On peut bien sûr modifier ce ressort pour obtenir des effets typographiques comme dans ce paragraphe où ce ressort a été rendu nul sans étirement afin que la dernière ligne soit justifiée. Cet effet doit être utilisé avec parcimonie, car

les espaces intermots de cette dernière ligne, beaucoup trop larges ici, démontrent que cet effet peut parfois être très inesthétique.

Ressorts d'extrémités de ligne

Les ressorts `\leftskip` et `\rightskip` sont insérés avant et après le contenu de chaque ligne (`\rightskip` est ajouté après `\parfillskip` à la dernière ligne du paragraphe). Lorsqu'ils sont égaux à `0pt` comme c'est le cas par défaut, le début et la fin de la ligne coïncident avec les limites de la zone de texte et le texte est dit « justifié » : ce sont les espaces entre les mots qui s'ajustent pour permettre cet effet typographique. La modification de ces ressorts d'extrémités de ligne permet des compositions qui contrarient cette justification.

Ainsi, donner une composante infinie à ces ressorts revient à les faire « pousser » avec une force infinie de telle sorte que les espaces entre les mots seront contraints à prendre leur dimension *naturelle* ce qui cassera la justification. Il est cependant plus judicieux de donner comme étirement une composante *finie* de telle sorte que celle-ci ne contraigne pas les espaces intermots. Il est utile de savoir qu'une composante *infinie* des ressorts d'extrémité de ligne décourage TEX d'effectuer une seconde passe lors de la composition du paragraphe. Seule la première passe est donc faite et les coupures de mots, envisagées à la seconde passe, ne seront donc jamais faites (lire page 484).

On voit que les ressorts d'extrémités de ligne entretiennent un rapport avec les espaces intermots dont il est temps de parler. Chaque police possède des registres internes définissant la largeur naturelle de l'espace intermot, son possible étirement et sa possible compression (lire page 471). Le ressort-primitive `\spaceskip` permet, lorsqu'il reçoit une assignation, d'écraser localement le contenu des registres de la police concernant les dimensions de l'espace intermot. Il est donc possible de supprimer la composante étirable de l'espace intermot afin que des espaces intermots trop longs et disgracieux soient évités dans une composition non justifiée.

Les effets typographiques les plus courants obtenus à l'aide des ressorts d'extrémité de ligne sont :

- composition au fer à gauche en donnant une composante étirable finie (par exemple 2 ou 3 `em`) à `\rightskip` et en laissant `\leftskip` nul ;
- composition au fer à droite en inversant les rôles des deux ressorts dans la manœuvre décrite au point précédent ;
- centrage du contenu des lignes en donnant la *même* composante étirable aux deux ressorts. Cette composante doit ici être assez grande pour permettre le centrage de lignes très courtes. Le choix de `1000pt` est fait par plain-TEX tandis que LATEX fait le choix de `0pt plus 1fil` pour sa macro `\centering`. On peut également choisir `0.5\hsize` où `\hsize` est la dimension-primitive qui contient la largeur de composition. Enfin, pour que la dernière ligne soit centrée, il faut également annuler `\parfillskip` qui sinon, deviendrait prépondérant par rapport à `\leftskip` et `\rightskip`.

Voici un exemple de méthode pour centrer deux paragraphes entiers :

Code n° III-220

```
\def\dummytext{Voici un texte qui ne revêt aucune importance
  et dont le seul but est de meubler artificiellement un paragraphe. }
```

```
\begingroup% à l'intérieur d'un groupe,
  \leftskip=0pt plus 0.5\hsize\relax
  \rightskip=\leftskip\relax % modifier les ressort d'extrémités
  \spaceskip=0.3em plus 0.05em\relax% dimension de l'espace intermot
  \parfillskip=0pt \relax% annuler le ressort de fin de paragraphe
  \dummytext\dummytext\dummytext% corps du paragraphe
  \par% compose la paragraphe courant
  Juste une phrase%
  \par% compose la paragraphe courant
\endgroup% avant de sortir du groupe
```

Voici un texte qui ne revêt aucune importance et dont le seul but est de meubler artificiellement un paragraphe. Voici un texte qui ne revêt aucune importance et dont le seul but est de meubler artificiellement un paragraphe. Voici un texte qui ne revêt aucune importance et dont le seul but est de meubler artificiellement un paragraphe.
Juste une phrase

En suivant cette méthode, il est facile de construire une macro `\narrowtext*` qui compose le paragraphe précédent et qui diminue désormais la largeur de composition de 20% de la largeur totale, sachant que cette largeur est contenue dans la dimension-primitive `\hsize`. Le « bloc » de composition ainsi constitué devra être centré par rapport aux frontières de la zone de texte. Il faut pour cela augmenter (via la primitive `\advance`) chacun des deux ressorts d'extrémités de ligne de `.1\hsize`. La macro `\endnarrowtext` signera la fin de la composition en largeur diminuée ; elle composera le paragraphe en cours avec `\par` et fermera le groupe semi-simple ouvert par `\narrowtext`.

Code n° III-221

```
\def\narrowtext{%
  \par
  \begingroup
    \advance\leftskip 0.1\hsize
    \advance\rightskip 0.1\hsize
}
\def\endnarrowtext{\par\endgroup}
\def\dummytext{Ceci est un texte sans
 importance destiné à meubler un paragraphe. }

\dummytext\dummytext\dummytext
\narrowtext
  \dummytext\dummytext\dummytext
 \narrowtext
    \dummytext\dummytext\dummytext
 \endnarrowtext
  \dummytext\dummytext\dummytext
\endnarrowtext
```

Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe.

Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe.

Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe.

Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans importance destiné à meubler un paragraphe. Ceci est un texte sans

importance destiné à meubler un paragraphe.

Comme aucune des macros \narrowtext ou \endnarrowtext n'admet d'argument, il n'y a aucun risque de mettre entre elles un contenu très long qui aurait pu inutilement surcharger la mémoire de TEX s'il avait été maladroitement incorporé dans un argument. L'autre avantage est que la lecture d'arguments est assortie du gel des catcodes qui n'existe pas ici.

8.2.4. Ressorts prédéfinis

Les ressorts-primitives modifiables – ceux qui étaient précédés d'une étoile dans le tableau de la page 244 – se comportent exactement comme s'ils étaient des macros définies avec \newskip. La seule différence est que leur valeur est figée. On remarque que \hfil et \vfil sont compensés (c'est-à-dire annulés) par \hfilneg et \vfilneg.

Les ressorts \hss et \vss sont eux encore plus curieux puisque, de dimension nulle, ils peuvent s'étirer ou se comprimer infiniment. Nous verrons leur intérêt plus loin.

Voici comment nous pouvons utiliser \hfill pour composer au fer à droite, au fer à gauche, centrer une ligne ou bien encore, régulièrement espacer zones de texte sur une ligne :

Code n° III-222

```
a) \hfill Composition au fer à droite\par
b) Composition au fer à gauche\hfill\kern0pt\par
c) \hfill Centrage\hfill\kern0pt\par
d) Des \hfill mots\hfill régulièrement\hfill espacés
```

a) Composition au fer à droite
b) Composition au fer à gauche
c) Centrage
d) Des mots régulièrement espacés

Comment se justifie la présence de ces « \kern0pt » après les \hfill en fin de ligne ? Tout vient de la règle précédente qui stipule qu'un ressort qui se situe juste avant la fin d'un paragraphe (autrement dit juste avant \par) est *ignoré*. Ce \kern0pt n'a d'autre but que de mettre quelque chose (on appelle cela un nœud) entre le ressort et \par. Bien évidemment, la dimension de ce nœud doit être nulle afin que son encombrement ne fausse pas la largeur des éléments pris en compte. Au lieu d'écrire \kern0pt, il est également courant d'employer la macro \null définie par plain-TEX dont le texte de remplacement est une boite horizontale vide, c'est-à-dire \hbox{}.

8.3. Les boites

Dans le monde de TEX, presque tout ce qui concerne la composition est lié aux boites. Les dimensions sont notamment utilisées pour mesurer les caractéristiques géométriques de ces boites. Les ressorts eux, sont chargés de remplir des zones dans ces boites.

Avant de commencer, il est important de comprendre qu'une boite reçoit du code source transmis sous la forme d'un argument, mais prend en compte le résultat typographique de ce code *une fois qu'il a été composé*. Les assignations sont mises à part pour être exécutés lorsque la boite est affichée. Ainsi, mettre quelque chose dans une boite implique donc au préalable le travail complet de composition.

8.3.1. Fabriquer des boites

Si TEX, de par son fonctionnement lors de la composition, fabrique automatiquement des boites pour les placer sur la page, des primitives permettent à l'utilisateur de fabriquer ses propres boites avec la possibilité d'y mettre un contenu arbitraire.

73 - Règle

TEX dispose de trois primitives pour enfermer un matériel dans une boite :

- `\hbox{`⟨*code*⟩`}` construit une boite *horizontale* contenant du matériel (qui est le résultat de la composition du ⟨*code*⟩) disposé en mode horizontal et compatible avec ce mode ;
- `\vbox{`⟨*code*⟩`}` et `\vtop{`⟨*code*⟩`}` bâtissent des boites *verticales*, susceptibles donc de recevoir du matériel disposé en mode vertical et compatible avec ce mode.

Dans les trois cas, l'intérieur d'une boite tient lieu de groupe, c'est-à-dire que les assignations faites dans le ⟨*code*⟩ restent locales à la boite.

Le ⟨*code*⟩ n'est pas lu en une fois comme s'il était l'argument d'une macro ; il est lu *au fur et à mesure* que la boite se remplit et donc au fil de cette lecture, des changements de catcode sont permis.

Une boite ainsi construite peut être :

- affichée lorsque `\hbox`, `\vbox` ou `\vtop` sont placés dans le flux d'entrée de TEX. Dans ce cas, la boite est ajoutée à la liste en cours selon le mode dans lequel TEX travaille à ce moment (horizontal ou vertical). Il est important de noter que ni `\hbox` ni `\vbox` ou `\vtop` ne provoquent de changement de mode, même si à l'intérieur de ces boites, un mode « interne » est imposé ;
- stockée dans un registre de boite (voir la section suivante).

Le code ci-dessous démontre que les boites s'affichent selon le mode en cours :

Code n° III-223

```
a% le caractère "a" fait passer en mode horizontal
\hbox{b}c\hbox{d}% les \hbox sont ajoutées en mode horizontal
\medbreak
...à comparer avec...
\medbreak
a \par% \par fait passer en mode vertical
\hbox{b}% \hbox est ajoutée à la liste verticale
c% "c" fait passer en mode horizontal
\hbox{d}% la \hbox est ajoutée en mode horizontal
```

abcd

...à comparer avec...

```
a
b
cd
```

Il est important de se souvenir de cette règle lorsqu'on construit une macro dont le premier acte est d'afficher une boite construite avec une des trois primitives vues ci-dessus. En effet, on souhaite souvent placer d'autres choses à droite de cette boite (en mode horizontal donc). Si la macro est appelée en mode vertical, le placement se fera *en dessous*, et non pas *après* comme attendu. Pour se prémunir de ce risque, il est *très sage* de mettre un `\leavevmode` avant la boite pour quitter le mode vertical. Ce faisant, la boite sera composée en mode horizontal et le comportement recherché sera obtenu.

Les boites construites avec `\vbox` ou `\vtop` empilent verticalement des éléments. Ces éléments sont susceptibles d'être mis dans une liste verticale car, à l'intérieur de `\vbox` ou `\vtop`, on est dans le mode vertical (plus exactement en mode vertical *interne*). Ce sont par exemple des paragraphes (qui ont une largeur égale à `\hsize`), des boites formées avec une des trois primitives, des ressorts verticaux insérés avec `\vskip`, des réglures (que nous verrons un peu plus loin), etc.

Autant la largeur d'une `\hbox` est clairement la largeur du matériel que l'on y a mis, autant celle d'une `\vbox` ou `\vtop` est moins évidente ; La largeur d'une boite verticale est celle de l'élément empilé verticalement qui a la plus grande largeur. Mais attention, si on écrit « `\vbox{foobar}` » ou « `\vtop{foobar}` », TEX composera un paragraphe pour ce seul mot et mettra cet élément dans la boite. Or, la largeur des paragraphes est égale à la dimension-primitive `\hsize` qui, dans le mode vertical principal, contient la largeur de la zone de texte. L'intérieur d'une boite jouant le rôle de groupe, on peut certes modifier `\hsize` dans une boite verticale pour imposer aux paragraphes une largeur choisie. Sans cette manœuvre pour modifier `\hsize`, la largeur de la boite sera la même que celles des paragraphes de la page, ce qui est peut-être un peu surdimensionné pour un paragraphe aussi court que « foobar » ! Au contraire, si l'on écrit « `\vbox{\hbox{foobar}}` » ou « `\vtop{\hbox{foobar}}` », l'élément qu'est la `\hbox` sera inséré (en mode vertical) et au final, la boite verticale aura la largeur de cet élément, c'est-à-dire la largeur du mot « foobar ».

74 - Règle

Les primitives `\vbox` et `\vtop` définissent des boites dans lesquelles des éléments sont empilés verticalement.

La différence entre `\vbox` et `\vtop` réside dans le point de référence de la boite finale. Dans les deux cas, les éléments sont empilés verticalement, le premier étant en haut et le dernier en bas. Avec `\vbox`, le point de référence de la boite est le point de référence du *dernier* élément (celui du bas) alors qu'avec `\vtop`, son point de référence est celui du *premier* élément (celui du haut).

Le point de référence d'une boite étant sur la ligne de base, on comprend donc que `\vbox` construit des empilements ayant des éléments au-dessus de cette ligne et donc produit en général des boites de grande hauteur et faible profondeur. À l'opposé, `\vtop` produit des empilements ayant des éléments sous cette ligne de base et donc des boites de petite hauteur et grande profondeur.

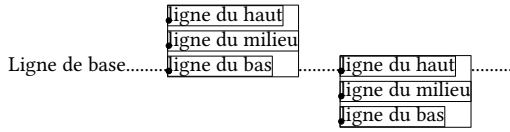
Dans l'exemple ci-dessous, on met dans une \vbox et une \vtop des éléments dont la largeur est contrôlé puisque ce sont des \hbox. On peut observer, grâce aux points qui représentent approximativement la ligne de base la différence fondamentale entre une \vbox et une \vtop :

Code n° III-224

```
Ligne de base : .........%
\vbox{\hbox{ligne du haut}\hbox{ligne du milieu}\hbox{ligne du bas}}%
.........%
\vtop{\hbox{ligne du haut}\hbox{ligne du milieu}\hbox{ligne du bas}}.........
```

```
                    ligne du haut
                    ligne du milieu
Ligne de base : .....ligne du bas  .....ligne du haut  .....
                                        ligne du milieu
                                        ligne du bas
```

Voici ce qui se passe en dessinant les boites et leur point de référence :



Nous allons maintenant faire de même avec un élément dont la largeur n'est pas contrôlée, c'est-à-dire un paragraphe entier composé de deux phrases identiques contenues dans la macro \dummytext. Afin que les deux boites tiennent horizontalement dans l'espace qui leur est réservé, la largeur des boites verticales sera imposée par \hsize à 4 cm :

Code n° III-225

```
\def\dummytext{Bla, bla, bla, un texte sans importance. }
Ligne de base.........%
\vbox{\hsize=4cm \dummytext\dummytext}%
.........%
\vtop{\hsize=4cm \dummytext\dummytext}.........
```

```
                    Bla, bla, bla, un texte sans impor-
                    tance. Bla, bla, bla, un texte sans im-
Ligne de base.....portance.                  .....Bla, bla, bla, un texte sans impor-.....
                                                  tance. Bla, bla, bla, un texte sans im-
                                                  portance.
```

8.3.2. Registres de boite

Une boite simplement affichée avec \hbox, \vbox ou \vtop ne permet pas de travailler sur cette boite. En revanche, une boite *stockée* dans un registre ouvre d'autres perspectives puisqu'elle est mémorisée par TEX. Dès lors, il devient possible d'effectuer des opérations sur ces boites.

Avant de poursuivre, il encore une fois est important de clarifier le sens du mot « stockage ». Autant les macros et les registres de tokens stockent du *code source tokénizé* c'est-à-dire du matériau brut initial, autant les registres de boites stockent d'un côté le résultat typographique d'un ⟨*code*⟩ une fois qu'il a été composé et de l'autre les assignations faites dans ce ⟨*code*⟩. Le stockage par macro et par boite sont deux stockages opposés en ce sens que chacun effectue une sauvegarde à l'un des deux bouts de la chaine qui mène du code source à l'affichage final.

Nous commençons à être familiers avec les registres puisque, hormis les macros, il en existe pratiquement pour chaque type de donnée que manipule TEX (tokens, entiers, dimensions, ressorts). Les boites ne font évidemment pas exception à la règle, même si la syntaxe des registres de boites est légèrement différente de celle des autres registres. La raison incombe au fait que TEX ne possède pas de primitive `\boxdef` alors qu'il possède `\toksdef`, `\countdef`, `\dimendef` et `\skipdef`.

75 - Règle

TEX dispose de 256 registres de boites (32 768 avec ε-TEX).

On demande à TEX d'allouer un numéro de registre de boite par l'intermédiaire de la macro `\newbox\`⟨*macro*⟩ et ce faisant, l'assignation suivante est faite

`\global\chardef\`⟨*macro*⟩`=`⟨*nombre*⟩

où le ⟨*nombre*⟩ est un numéro de registre de boite libre.

Ce faisant, la `\`⟨*macro*⟩ devient équivalente à `\char`⟨*nombre*⟩. Ainsi, `\`⟨*macro*⟩ produit le caractère de code ⟨*nombre*⟩, mais lorsque TEX s'attend à lire un nombre (comme c'est le cas lorsqu'il attend un numéro de boite), `\`⟨*macro*⟩ est lue comme l'entier ⟨*nombre*⟩. Du point de vue de la mécanique interne, un ⟨*registre*⟩ de boite est donc un ⟨*nombre*⟩.

On accède à la boite contenue dans un registre par `\box`⟨*registre*⟩.

Pour procéder à l'assignation, on écrit :

`\setbox`⟨*registre*⟩`=` ⟨*boite valide*⟩

où une ⟨*boite valide*⟩ est soit une boite contenue dans un registre et écrite sous la forme `\box`⟨*registre*⟩, soit une boite construite avec les primitives `\hbox`, `\vbox` ou `\vtop`. Le signe = et l'espace qui le suit sont facultatifs.

Pour afficher une boite stockée dans un ⟨*registre*⟩, on écrit `\box`⟨*registre*⟩. La boite est affichée selon le mode en cours (horizontal ou vertical) : le type de boite n'a pas d'influence sur le mode dans lequel elle est affichée.

Lorsqu'on utilise la primitive `\box` pour accéder à la boite stockée dans un registre, la boite contenue dans le registre est irrémédiablement perdue et le registre devient vide, c'est-à-dire qu'il ne contient aucune boite. Si on souhaite conserver la boite dans le registre, on doit utiliser la primitive `\copy` au lieu de `\box`.

76 - Règle

Le test

`\ifvoid`⟨*registre*⟩⟨*code vrai*⟩`\else`⟨*code faux*⟩`\fi`

se comporte comme les autres tests de TEX. Il revoit ⟨*code vrai*⟩ si le ⟨*registre*⟩ de boite est vide (c'est-à-dire s'il ne contient aucune boite).

Il est admis que la boite n° 0 est une boite « brouillon » qui peut être utilisée ponctuellement si l'on ne veut pas mobiliser un nouveau registre de boite.

L'exemple ci-dessous illustre ce que nous venons d'aborder. Nous allons tout d'abord allouer un nouveau registre `\foobox` par `\newbox` et nous y stockerons une

boite horizontale contenant le mot « foobar ». Ensuite, nous allons utiliser le test \ifvoid dans plusieurs cas de figure afin de montrer que le registre devient vide après avoir exécuté la primitive \box.

Code n° III-226

```
\newbox\foobox
a) \setbox\foobox=\hbox{foobar}% registre non vide
   Le registre est \ifvoid\foobox vide\else non vide\fi\par
b) \setbox\foobox=\hbox{foobar}
   "\box\foobox" et le registre est \ifvoid\foobox vide\else non vide\fi\par
c) \setbox\foobox=\hbox{foobar}
   "\copy\foobox" et le registre est \ifvoid\foobox vide\else non vide\fi\par
d) \setbox\foobox=\hbox{}
   Le registre est \ifvoid\foobox vide\else non vide\fi
```

a) Le registre est non vide
b) "foobar" et le registre est vide
c) "foobar" et le registre est non vide
d) Le registre est non vide

On remarque que \box vide le registre (cas b) tandis que \copy le préserve (cas c). Il est également important de noter que le registre n'est pas vide (cas d) s'il contient une boite vide.

Deux autres tests sur les registres de boite méritent d'être signalés :

\ifhbox⟨*registre*⟩ et \ifvbox⟨*registre*⟩

sont vrais si le ⟨*registre*⟩ contient respectivement une boite horizontale ou verticale.

8.3.3. Dimensions des boites

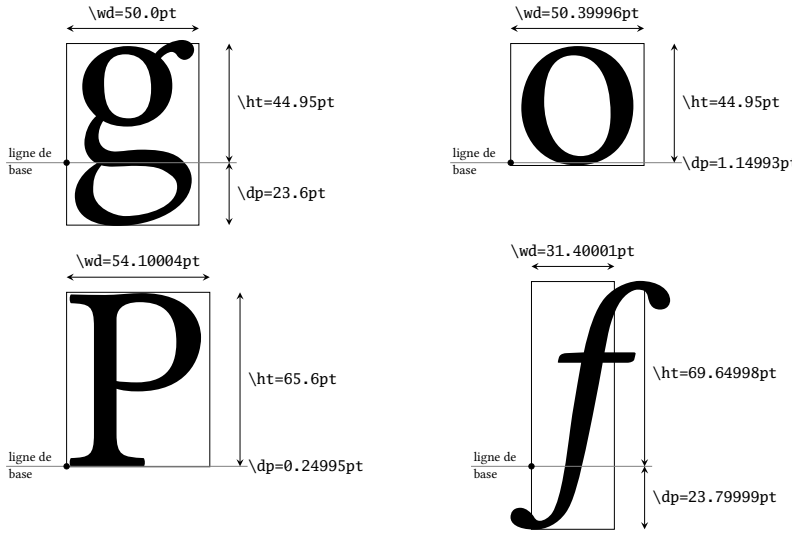
77 - Règle

Les dimensions d'une boite stockée dans un ⟨*registre*⟩ sont :

- sa longueur horizontale, accessible par \wd⟨*registre*⟩ ;
- sa hauteur, qui s'étend au-dessus de la ligne de base, accessible par \ht ⟨*registre*⟩ ;
- enfin, sa profondeur qui s'étend au-dessous de la ligne de base et qui est accessible par \dp⟨*registre*⟩.

Les primitives \wd, \ht et \dp suivies d'un ⟨*registre*⟩ se comportent comme des registres de dimension.

Pour fixer les idées, voici un schéma où les boites englobantes, rectangles qui représentent l'encombrement d'une boite tel qu'il est pris en compte par TEX, sont tracées et les trois dimensions sont écrites pour quatre lettres différentes :



On peut observer notamment avec la lettre « f » que la boite englobante ne correspond pas à la plus petite boite qui contient les contours de la lettre. Pour d'évidentes raisons de typographie, il est *nécessaire* que certaines parties de la lettre « f » dépassent de la boite englobante sinon, au lieu de « *foo* », nous aurions un horrible « $f\,oo$ » !

De plus, contrairement à ce que l'on pourrait croire, la profondeur de lettres comme « o » ou « P » n'est pas nulle ! Très faible certes, mais pas nulle… Le concepteur de la fonte « Libertine » avec laquelle est écrit ce livre en a décidé ainsi.

Passons à la pratique et mesurons maintenant les dimensions de boites construites successivement avec les trois primitives \hbox, \vbox et \vtop. Les boites verticales seront des empilements de \hbox, chacune contenant un mot de la phrase « Programmer est facile. »

Code n° III-227

```
\newbox\foobox
\setbox\foobox=\hbox{Programmer est facile.}
<<\copy\foobox>>
mesure \the\wd\foobox\ de long, \the\dp\foobox\ de profondeur et \the\ht\foobox\ de haut.
\medbreak

\setbox\foobox=\vbox{\hbox{Programmer}\hbox{est}\hbox{facile.}}
<<\copy\foobox>>
mesure \the\wd\foobox\ de long, \the\dp\foobox\ de profondeur et \the\ht\foobox\ de haut.
\medbreak

\setbox\foobox=\vtop{\hbox{Programmer}\hbox{est}\hbox{facile.}}
<<\copy\foobox>>
mesure \the\wd\foobox\ de long, \the\dp\foobox\ de profondeur et \the\ht\foobox\ de haut.
```

«Programmer est facile.» mesure 73.4159pt de long, 1.888pt de profondeur et 5.496pt de haut.

Programmer
est
«facile. » mesure 41.09595pt de long, 0.092pt de profondeur et 24.248pt de haut.

«Programmer» mesure 41.09595pt de long, 19.092pt de profondeur et 5.248pt de haut.
est
facile.

La faible profondeur de la \vbox provient une fois encore du design de la fonte Libertine qui donne une faible profondeur à certaines lettres qui composent le mot du bas « facile ». La verticalité *totale* des boites verticales (c'est-à-dire la somme de leur hauteur et de leur profondeur) \vbox et \vtop est la même dans les deux cas.

■ Exercice 84

Programmer une macro \vdim*{⟨*registre*⟩} qui calcule la verticalité de la boite contenue dans le ⟨*registre*⟩.

□ Solution

Il suffit de donner à \dimexpr l'addition de sa hauteur et sa profondeur :

Code n° III-228

```
\def\vdim#1{\dimexpr\ht#1+\dp#1\relax}
a) \setbox\foobox=\vbox{\hbox{Programmer}\hbox{est}\hbox{facile.}}
   Verticalité de la \litterate-\vbox- = \the\vdim\foobox\par
b) \setbox\foobox=\vtop{\hbox{Programmer}\hbox{est}\hbox{facile.}}
   Verticalité de la \litterate-\vtop- = \the\vdim\foobox
```

a) Verticalité de la \vbox = 24.34pt
b) Verticalité de la \vtop = 24.34pt

■

■ Exercice 85

Inventer un procédé qui permette de compter combien de caractères affichables contient un argument. On appellera la macro \countallchar*{⟨*texte*⟩} et on supposera que le ⟨*texte*⟩ est uniquement constitué de caractères affichables (catcodes 11 et 12) et d'espaces.

□ Solution

Au point où nous en sommes, la solution la plus immédiate est de mettre cet argument dans une boite \hbox où la fonte sera à chasse fixe. Il suffit de mesurer cette boite, de mesurer un caractère et d'effectuer la division entre les entiers contraints de ces deux dimensions pour afficher le nombre de caractères. Ici, nous allons même afficher la division conduisant au résultat :

Code n° III-229

```
\def\countallchar#1{%
  Il y a %
  \setbox0=\hbox{\tt#1}% met #1 dans la boite
  \edef\arglength{\number\wd0 }% stocke la largeur de la boite en sp
  \setbox0=\hbox{\tt A}% met "A" dans la boite
  \edef\charlength{\number\wd0 }% stocke la largeur d'un caractère
  $\number\arglength/\charlength % affiche la division
  =\number\numexpr\arglength/\charlength\relax$ % affiche le quotient
  caractères%
}
\countallchar{abcd efgh}\par
\countallchar{A5 xW5 64 a1}\par
\countallchar{affligeant}
```

Il y a 2265300/251700 = 9 caractères

```
Il y a 3020400/251700 = 12 caractères
Il y a 2517000/251700 = 10 caractères
```

Une méthode similaire peut être déployée pour compter combien de fois figure un caractère affichable α dans un argument. L'idée est de mesurer la boite contenant l'argument composé en fonte à chasse fixe et stocker cette longueur dans L. Ensuite, le caractère α est supprimé avec la macro \substin et la manœuvre est répétée : l est la nouvelle longueur. Le nombre d'occurrences de α est la différence $L - l$ divisée par la longueur de la boite contenant α :

Code n° III-230

```
\catcode`@11
\def\countchar#1#2{%
    \setbox\z@\hbox{\tt#2}% met #2 dans boite 0
    \edef\len@a{\number\wd\z@}% mesure la boite
    \setbox\z@\hbox{\tt\substin{#2}{#1}{}}% recommencer sans "#1"
    \edef\len@b{\number\wd\z@}% mesure la boite
    \setbox\z@\hbox{\tt A}% met "A" dans la boite
    \edef\charlength{\number\wd\z@}% stocke la largeur du caractère
    \number\numexpr(\len@a-\len@b)/\charlength% et affiche le quotient
}
\catcode`@12
a) \countchar{a}{abracadabra}\qquad
b) \countchar{b}{zigzag}\qquad
c) \countchar{ }{a bc de f ghi j k }
```

a) 5 b) 0 c) 7

■

8.3.4. Déplacer des boites

TEX met à disposition des primitives pour déplacer des boites dans la direction que l'on veut (haut, bas, gauche, droite).

78 - Règle

Les primitives \lower ou \raise doivent être suivies d'une ⟨*dimension*⟩, le tout devant précéder une boite horizontale, qu'elle soit écrite par l'intermédiaire de \hbox ou d'un registre. L'effet est d'abaisser pour \lower ou d'élever pour \raise la boite de la ⟨*dimension*⟩ spécifiée.

Les primitives \moveleft et \moveright partagent la même syntaxe, mais agissent sur des boites *verticales* en les déplaçant vers la gauche ou vers la droite.

Voici comment on peut enfermer des mots dans une \hbox et les élever ou les abaisser d'une dimension arbitraire :

Code n° III-231

```
Programmer \raise1ex\hbox{en} \lower1ex\hbox{\TeX} \lower2ex\hbox{est} facile.
```

Programmer $^{\text{en}}$ TEX $_{\text{est}}$ facile.

Il faut d'ailleurs noter que le logo « TEX », obtenu en exécutant la macro \TeX, est construit en utilisant \lower pour abaisser la lettre « E » après rebroussé chemin vers la gauche à l'aide de \kern. La lettre « X » est également déplacée vers la gauche avec un autre \kern. Voici comment est définie la macro TEX par plain-TEX :

```
\def\TeX{T\kern-.1667em\lower.5ex\hbox{E}\kern-.125emX}
```

Il peut sembler frustrant que l'on ne puisse pas « centrer » une boite contenant du matériel vertical par rapport à l'axe qu'est la ligne de base puisque \vbox et \vtop empilent ce matériel respectivement au-dessus ou au-dessous de cette ligne. L'idée serait d'abaisser ces boites avec \lower de la dimension égale à

$$\frac{\text{hauteur} - \text{profondeur}}{2}$$

étant entendu que si cette quantité est négative, abaisser d'une dimension négative revient à élever la boite. En coulisses, cette méthode nous contraint à mettre la boite dans un registre pour pouvoir la mesurer afin d'accéder à sa hauteur et sa profondeur. La macro \cbox⋆ va mettre en application cette méthode et sera amenée à recevoir du matériel vertical dans les mêmes conditions que \vbox afin de centrer la boite par rapport à la ligne de base :

Code n° III-232

```
\def\cbox#1{%
  \setbox0\vbox{#1}% met le contenu dans une \vbox
  \lower\dimexpr(\ht0-\dp0)/2\relax\box0 % l'abaisse
}

......\cbox{\hbox{$x^2$}}......\cbox{\hbox{foo}\hbox{et bar}}......%
\cbox{\hbox{Programmer}\hbox{en \TeX}\hbox{est facile}}.......
```

……x^2……foo et bar……Programmer en TEX est facile……

TEX dispose de la primitive \vcenter, qui fonctionne en mode mathématique, dont l'argument est constitué de matériel vertical (comme l'est celui de \vbox ou \vtop) et qui place la boite ainsi créé de telle sorte que ses frontières haute et basse soient équidistantes de « l'axe mathématique ». Deux précisions doivent être apportées :

- la primitive ne peut être appelée qu'en mode mathématique, mais le contenu de la boite n'hérite pas de ce mode, c'est-à-dire qu'il sera composé en mode texte, sauf si bien sûr, on demande explicitement à passer en mode mathématique à l'intérieur de la boite (les boites \hbox, \vbox et \vtop partagent aussi cette propriété) ;
- l'axe mathématique *n'est pas* à la même hauteur que la ligne de base ; il s'agit de deux axes différents. L'axe mathématique est horizontal et se trouve approximativement à équidistance des traits horizontaux du signe « = ». Il représente l'axe de symétrie horizontal de plusieurs signes mathématiques alors que la ligne de base est approximativement tangente au bas des lettres qui n'ont pas de jambage comme on le voit dans l'égalité ci-dessous :

ligne de base — $a(b+c) = ab + ac$ — axe math

Par conséquent, une boite centrée avec \vcenter ne sera pas exactement à la même hauteur que la même boite centrée avec la macro \cbox vue précédemment. La différence se voit sans peine à l'œil nu dans cet exemple :

Code n° III-233

```
1 ......\cbox{\hbox{foo}\hbox{et bar}}......$\vcenter{\hbox{foo}\hbox{et bar}}$......
```

.....foo et bar..... foo et bar.....

■ **Exercice 86**

Inventer un procédé pour que la macro `\htmath*` affiche à quelle distance de la ligne de base se trouve l'axe mathématique.

□ **Solution**

Si une \hbox contient une `\vcenter` vide, la hauteur de la \hbox sera précisément la distance cherchée.

Code n° III-234

```
1 \def\htmath{\begingroup
2   \setbox0=\hbox{$\vcenter{}$}\the\ht0
3 \endgroup
4 }
5 L'axe mathématique se trouve à \htmath{} de la ligne de base.
```

L'axe mathématique se trouve à 2.056pt de la ligne de base.

■

8.3.5. Choisir la dimension d'une boite

79 - Règle

On peut imposer une dimension aux boites en le spécifiant avec le mot-clé « `to` » suivi d'un espace optionnel puis d'une ⟨*dimension*⟩. Et donc, si l'on écrit

`\hbox to 3cm{`⟨*contenu*⟩`}`

cela créera une boite horizontale de longueur 3 cm, quel que soit le ⟨*contenu*⟩. Si le contenu ne mesure pas *exactement* 3 cm et ne contient aucun élément étirable, il y aura soit un débordement, soit un sous-remplissage de boite, chacun respectivement signalé par les messages d'avertissement « `Overfull \hbox` » ou « `Underfull \hbox` » dans le fichier `log`. La même syntaxe s'applique pour `\vbox` et `\vtop` sauf que la dimension est imposée dans le sens *vertical*.

On peut également écrire le mot-clé « `spread` » utilisé à la place de « `to` » si l'on souhaite construire une boite dont la dimension finale sera sa dimension naturelle augmentée de la ⟨*dimension*⟩. Si cette dimension est négative, la boite créée aura une dimension inférieure à sa dimension naturelle.

Voici un exemple qui montre comment une boite est affichée avec sa largeur naturelle puis étirée de `5pt` et `10pt` grâce à un ressort `\hfil` placé entre les syllabes « `foo` » et « `bar` » :

Code n° III-235

```
1 1) \hbox{foobar}\par
2 2) \hbox spread 5pt{foo\hfil bar}\par
```

```
3) \hbox spread10pt{foo\hfil bar}
```

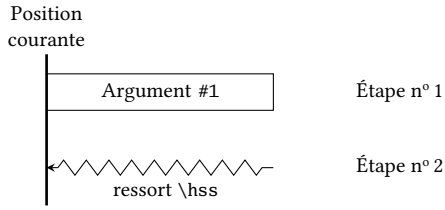
1) foobar
2) foo bar
3) foo bar

Que ce soit via « to » ou « spread », il est nécessaire que contenu de la boite ait un ou plusieurs ressorts. Ils devront avoir des composantes étirables convenablement choisies pour qu'en s'étirant ou en se comprimant, ils puissent compenser l'excès ou le manque de place pour le contenu. Un des ressorts les plus utilisés dans ce cas est \hss pour les boites horizontales et \vss pour les boites verticales. Rappelons qu'ils ont une longueur nulle et peuvent s'étirer ou se comprimer infiniment.

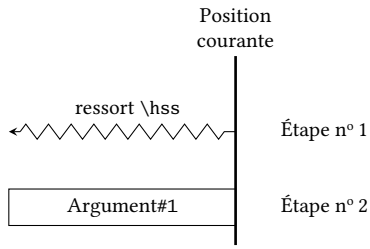
On peut facilement créer des macros qui affichent du texte *en surimpression* soit à droite, soit à gauche de la position courante. Ces deux macros, \rlap et \llap, dont le nom signifie « right overlap » (surimpression vers la droite) ou « left overlap » (surimpression vers la gauche) sont ainsi définies dans le format plain-TEX :

```
\def\rlap#1{\hbox to\z@{#1\hss}}
\def\llap#1{\hbox to\z@{\hss#1}}
```

Examinons tout d'abord \rlap. Son argument #1 est mis dans une boite horizontale de longueur nulle (\z@ représente la dimension 0pt). Un sur-remplissage de la boite semble inévitable, car le contenu #1 prend une certaine place horizontale. C'est là que le ressort \hss placé en fin de boite va intervenir : pour satisfaire la longueur de la boite nulle, il va se comprimer pour atteindre une longueur négative exactement égale à l'opposé de la longueur de l'argument #1. La somme des longueurs de #1 et de \hss satisfait bien la dimension nulle imposée.



Son homologue \llap fonctionne de la même façon sauf que le ressort \hss est placé en début de boite : il va donc se comprimer en premier ce qui se traduit à l'affichage par un recul vers la gauche de la longueur de #1. Ensuite, #1 est composé et rattrape exactement la longueur négative de \hss.



L'exemple ci-dessous montre comment ces deux macros peuvent utilisées pour

écrire soit à gauche, soit à droite de la position courante en surimpression. Dans les deux cas, le caractère « | » est utilisé pour symboliser la position courante lorsque TEX rencontre ces deux macros :

Code n° III-236

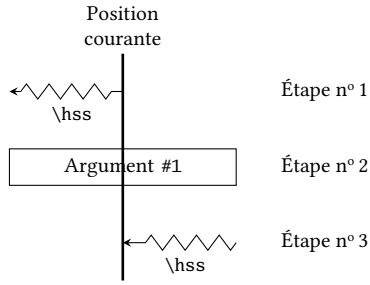
```
1 foobar|\rlap{/////}123456\qquad foobar|\llap{/////}123456
```

foobar|*/////*123456 foo*/////*bar|123456

Il est curieux que D. KNUTH n'ait pas implémenté la macro `\clap*` qui permettrait de centrer le contenu de son argument sur la position courante sans que l'encombrement horizontal ne soit pris en compte :

```
\def\clap#1{\hbox to0pt{\hss#1\hss}}
```

Bien sûr, le fonctionnement de cette macro est en tout point semblable à `\rlap` et `\llap` sauf que les deux ressorts se compriment de façon identique ce qui donne un centrage du contenu par rapport à la position courante :



Des effets d'empilement peuvent être créés avec les macros `\raise` et `\lower` qui déplacent verticalement la prochaine boite horizontale d'une dimension choisie :

Code n° III-237

```
1 \def\clap#1{\hbox to0pt{\hss#1\hss}}
2 a) avant la macro|\clap{SURIMPRESSION}après la macro\medbreak
3 b) avant la macro|\raise2.5ex\clap{Au-dessus}\lower2.5ex\clap{Au-dessous}%
4    après la macro\medbreak
5 c) avant la macro|\raise2.5ex\llap{Au-dessus avant}\lower2.5ex\rlap{Au-dessous après}%
6    après la macro
```

a) avant la macro|après la macro
SURIMPRESSION

Au-dessus
b) avant la macro|après la macro
Au-dessous

Au-dessus avant
c) avant la macro|après la macro
Au-dessous après

8.3.6. Redimensionner une boite

80 - Règle

Une fois qu'une boite est stockée dans un ⟨*registre*⟩, il est possible de modifier une ou plusieurs de ses dimensions. Il suffit d'écrire

$$\left.\begin{matrix}\texttt{\textbackslash wd}\\ \texttt{\textbackslash ht}\\ \texttt{\textbackslash dp}\end{matrix}\right\}\langle \textit{registre}\rangle = \langle \textit{dimension}\rangle$$

pour que la largeur, hauteur ou profondeur de la boite prenne la dimension indiquée, indépendamment de ce qu'est le contenu de la boite. Ces assignations sont toujours *globales*. En procédant de cette façon, le contenu reste intact, mais TEX est trompé et voit désormais la boite avec les nouvelles dimensions.

Un peu à la manière de `\rlap`, il devient ainsi possible de superposer une boite au texte qui la suit :

Code nº III-238

```
\setbox0=\hbox{\tt//////////}
\wd0=0pt % fait croire à TeX que la larguer de la boite est nulle
Voici \copy0 du texte partiellement barré...
```

Voici du texte partiellement barré...

Est-il possible de redonner au contenu d'une boite ainsi modifiée une nouvelle virginité quant à ses dimensions, et faire en sorte que ce contenu occupe à nouveau ses dimensions naturelles ?

81 - Règle

Si un ⟨*registre*⟩ de boite contient une boite horizontale ou verticale, il est possible d'extraire le contenu de la boite avec `\unhbox`⟨*registre*⟩ ou `\unvbox`⟨*registre*⟩. La liste horizontale ou verticale contenue dans la boite est alors ajoutée à la liste courante en cours. Après cette opération, le registre de boite devient vide, c'est à dire positif au sens de `\ifvoid`.

Pour ne pas vider le registre de boite, il faut employer `\unhcopy` ou `\unvcopy`.

L'important à retenir de cette règle est que la liste ainsi extraite retrouve ses dimensions naturelles, même si les dimensions de la boite avaient été modifiées après coup.

Code nº III-239

```
\def\printdim{largeur=\the\wd0 \qquad hauteur=\the\ht0 \qquad profondeur = \the\dp0 }
\setbox0=\hbox{Programmer en \TeX{} est facile}
a) \printdim\par
b) \wd0=0pt \ht0=0pt \dp0=0pt% rend toutes le dimensions nulles
   \printdim\par
c) \setbox0=\hbox{\unhbox0 }% reprend les dimensions d'origine
   \printdim
```

a) largeur=95.82626pt hauteur=5.496pt profondeur = 1.888pt
b) largeur=0.0pt hauteur=0.0pt profondeur = 0.0pt
c) largeur=95.82626pt hauteur=5.496pt profondeur = 1.888pt

Le point essentiel est que

\setbox⟨*registre*⟩=\hbox{\unhbox⟨*registre*⟩}

redonne au ⟨*registre*⟩ de boite les dimensions naturelles de la boite qu'il contient. La méthode donnée pour une boite horizontale est également valable pour une boite verticale en remplaçant \hbox par \vbox ou \vtop et \unhbox par \unvbox.

8.3.7. Tester si un registre de boite est vide

Nous avons vu que le test \ifvoid⟨*registre*⟩ est vrai si le registre de boite est vide. Il est tout de même très important d'insister sur la différence entre un registre vide (c'est à dire ne contenant aucune boite) et un registre contenant une boite vide.

Voici un exemple qui illustre le fait que si un registre contient une boite vide, il n'est pas vide !

Code n° III-240

```
\setbox0=\hbox{}% le registre 0 contient une boite vide
Le registre \ifvoid0 est vide\else n'est pas vide\fi
```

Le registre n'est pas vide

Dès lors se pose la question : « comment tester si un registre de boite est vide ? » La réponse dépend de ce qu'on entend par « vide ». Le parallèle devient évident avec un argument de macro qui n'était pas « vide » s'il contenait une macro vide telle que \empty (voir page 189). Pour répondre à la question « comment tester si un registre de boite est vide », nous allons construire un test purement développable \ifzerodimbox⋆ de syntaxe

\ifzerodimbox{⟨*nombre*⟩}{⟨*code vrai*⟩}{⟨*code faux*⟩}

qui renvoie ⟨*code vrai*⟩ si le registre est vide ou s'il contient une boite dont toutes les dimensions sont nulles, et faux dans tous les autres cas.

Code n° III-241

```
\catcode`\@11
\def\ifzerodimbox#1{% #1=registre de boite
% revoie vrai si le registre est vide ou contient une boite de dimensions nulles
  \csname% former la macro "\firstoftwo" ou "\secondoftwo"
  \ifvoid#1first%% si le registre est vide "first"
  \else% sinon
    \ifdim\wd#1=\z@% si la largeur
      \ifdim\ht#1=\z@% la hauteur
        \ifdim\dp#1=\z@ first% et la profondeur=0pt, "first"
        \else second% dans les autres cas "second"
        \fi
      \else second%
      \fi
    \else second%
    \fi
  \fi
  oftwo% compléter avec "oftwo"
  \endcsname
}
\catcode`\@12
a) \setbox0=\hbox{}\ifzerodimbox0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
b) \box0 % affiche la boite vide, le registre est maintenant "void"
```

```
   \ifzerodimbox0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
c) \setbox0=\hbox{x}\ifzerodimbox0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
d) \wd0=0pt \ht0=0pt \dp0=0pt % rend toutes les dimensions nulles
   \ifzerodimbox0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
e) \setbox0=\vbox{}\ifzerodimbox0{vrai}{faux} (et \ifvoid0 void\else non void\fi)
```

a) vrai (et non void)
b) vrai (et void)
c) faux (et non void)
d) vrai (et non void)
e) vrai (et non void)

Le test est négatif dans le cas c, seul cas où le registre contient une boite de dimensions non nulles. Il est positif ailleurs, que la boite soit vide (cas a et e), que la boite soit non vide mais redimensionnée (cas d) ou que le registre soit vide (cas b).

Le moteur ε-TEX nous donne une autre méthode, plus rapide et plus fiable pour tester si un registre de boite contient une boite *vide*. Nous nous mettrons à l'abri du faux positif de l'exemple précédent (cas d) d'une boite non vide, mais redimensionnée pour que toutes ses dimensions soient nulles.

La primitive `\lastnodetype` de ε-TEX est de type registre d'entier et à tout moment, selon sa valeur qui varie de -1 à 15, caractérise la nature du dernier nœud composé. Les différents types sont exposés dans le manuel d'ε-TEX. Le type qui nous intéresse ici est celui qui est qualifié par l'entier -1 et qui caractérise un nœud vide, c'est-à-dire résultant d'une liste vide.

L'idée est donc de composer le contenu de la boite, et de tester hors de cette boite si `\lastnodetype` est strictement négatif, seul cas où la boite sera vide. Nous allons mettre en œuvre cette méthode pour construire une macro *non développable* `\ifvoidorempty`* de syntaxe

`\ifvoidorempty`⟨*registre*⟩{⟨*code vrai*⟩}{⟨*code faux*⟩}

qui renverra ⟨*code vrai*⟩ si le ⟨*registre*⟩ est vide ou contient une boite vide :

Code n° III-242

```
\def\ifvoidorempty#1{% teste si le registre #1 est vide ou contient une boite vide
  \ifvoid#1\relax
    \expandafter\firstoftwo
  \else
    \begingroup% dans un groupe
      \setbox0=% affecter à la boite 0
        \ifhbox#1\hbox\bgroup\unhcopy% un boite horizontale
        \else    \vbox\bgroup\unvcopy% ou verticale
        \fi% dans laquelle on compose
        #1\relax% #1 en dimensions naturelles
        \expandafter\egroup% sauter la fin de la boite
        \expandafter% et le \endgroup
    \endgroup
    \ifnum\lastnodetype=-1 % et tester si le dernier noeud est vide
      \expandafter\expandafter\expandafter\firstoftwo
    \else
      \expandafter\expandafter\expandafter\secondoftwo
    \fi
  \fi
}
a) \setbox0=\hbox{}\ifvoidorempty0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
```

```
b) \box0 % affiche la boite vide, le registre est maintenant "void"
   \ifvoidorempty0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
c) \setbox0=\hbox{x}\ifvoidorempty0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
d) \wd0=0pt \ht0=0pt \dp0=0pt % rend toutes les dimensions nulles
   \ifvoidorempty0{vrai}{faux} (et \ifvoid0 void\else non void\fi)\par
e) \setbox0=\vbox{}\ifvoidorempty0{vrai}{faux} (et \ifvoid0 void\else non void\fi)
```

a) vrai (et non void)
b) vrai (et void)
c) faux (et non void)
d) faux (et non void)
e) vrai (et non void)

8.3.8. Mise en application

Comment pourrions-nous enfermer dans une boite `\vtop` des éléments mis dans des `\hbox` qui soient tous centrés horizontalement les uns par rapport aux autres ? Le but est d'écrire

```
\cvtop{Ligne du haut,Très grande ligne du milieu,Ligne du bas pour finir}
```

pour obtenir ceci

texte avant... Ligne du haut ...texte après
Très grande ligne de milieu
Ligne de bas pour finir

Si nous empilons des `\hbox` dans une `\vtop`, nous ne pouvons pas centrer les `\hbox`, car nous ne pouvons écrire ni `\hfill` ni `\hss` dans une `\vtop` puisque le matériel y est vertical et que ces primitive, exclusivement horizontales, y sont interdites. Agir sur les ressorts de paragraphe `\leftskip` et `\rightskip` est sans effet puisque dans cette boite, TEX ne compose pas de paragraphe, il ne fait qu'empiler des `\hbox` verticalement.

La solution va consister à trouver la longueur l de la plus longue `\hbox`, puis composer toutes les boites `\hbox` avec une longueur imposée l en ayant pris soin de centrer le contenu avec des ressorts `\hss` placés en début et en fin de boite. La macro `\doforeach` vue précédemment nous sera d'un grand secours. La variable, que nous appelons `\htext`, contiendra tout à tour à tour chaque élément de la liste, mais il faudra procéder en deux passes successives :

- la première trouvera, en les examinant tour à tour, la `\hbox` la plus longue et cette dimension maximale sera stockée dans le registre de dimension `\maxhsize` ;
- lors de la deuxième passe, nous écrirons à chaque itération

  ```
  \hbox to\maxhsize{\hss\htext\hss}
  ```

  pour enfermer l'élément courant dans une `\hbox` de longueur `\maxhsize` et centré dans celle-ci à l'aide des ressorts-primitives `\hss`.

Pour tester si la longueur de l'élément considéré est plus grande que celle du précédent, il faut d'abord initialiser `\hmaxsize` à la plus petite dimension permise qui est l'opposé de la plus grande dimension `\maxdimen`, registre de dimension défini dans plain-TEX et qui vaut `16383.99999pt`. Ensuite, si la dimension de l'élément est strictement supérieure à `\hmaxsize`, nous actualiserons `\hmaxsize` à la longueur de

l'élément examiné. En fin de boucle, nous serons assurés que \hmaxsize est la plus grande longueur.

Code n° III-243

```
\newdimen\hmaxsize
\def\cvtop#1{%
  \hmaxsize=-\maxdimen% initialise à la plus petite longueur
  \doforeach\htext\in{#1}% pour chaque élément :
    {\setbox0=\hbox{\htext}% stocker l'élément "\htext" dans une \hbox
    \ifdim\wd0 >\hmaxsize% si sa longueur est supérieure à \hmaxsize
      \hmaxsize=\wd0 % mettre à jour \hmaxsize
    \fi
    }%
  \vtop{% dans une \vtop...
    \doforeach\htext\in{#1}% pour chaque élément :
      {\hbox to\hmaxsize{\hss\htext\hss}% le centrer dans une \hbox de longueur \hmaxsize
      }%
  }%
}

texte avant...\cvtop{Ligne du haut,Très grande ligne du milieu,Ligne du bas pour finir}%
...texte après
```

texte avant... Ligne du haut ...texte après
Très grande ligne du milieu
Ligne du bas pour finir

■ Exercice 87

Proposer une autre façon de faire sans enfermer \htext dans une \hbox.

□ Solution

Chaque élément peut être composé comme un paragraphe. Pour cela, la longueur de la \vtop doit être égale à \hmaxsize, l'indentation doit être nulle et pour que les paragraphes soient centrés, \leftskip et \rightskip sont rendus égaux à `0pt plus 1fil`.

Cela donne :

Code n° III-244

```
\newdimen\hmaxsize
\def\cvtop#1{%
  \hmaxsize=-\maxdimen% initialise à la plus petite longueur
  \doforeach\htext\in{#1}% pour chaque élément :
    {\setbox0=\hbox{\htext}% stocker l'élément "\htext" dans une \hbox
    \ifdim\wd0 >\hmaxsize% si sa longueur est supérieure à \hmaxsize
      \hmaxsize=\wd0 % mettre à jour \hmaxsize
    \fi
    }%
  \vtop{% dans une \vtop...
    \hsize\hmaxsize % longueur de la \vtop = \maxhsize
    \parindent=0pt % pas d'indentation
    \leftskip=0pt plus1fil minus0pt \rightskip=\leftskip% ressorts de centrage
    \doforeach\htext\in{#1}% pour chaque élément :
      {\htext\par}% le composer et finir le paragraphe
  }%
}

texte avant...\cvtop{Ligne du haut,Très grande ligne du milieu,Ligne du bas pour finir}%
...texte après
```

```
texte avant...      Ligne du haut            ...texte après
                 Très grande ligne du milieu
                   Ligne du bas pour finir
```

■

Ces façons de faire sont un peu tirée par les cheveux. En effet, nous venons de programmer ce que fait (entre autres) la primitive de TEX « \halign{⟨*code*⟩} ». Le but ici n'est pas d'en décrire toutes ses fonctionnalités, car elles sont très nombreuses et pour certaines, très techniques [2].

Pour aller à l'essentiel, cette primitive opère en mode vertical et construit des alignements (autrement dit, des tableaux) constitués de lignes, elles-mêmes partagées en cellules. Ce qui est remarquable, c'est que lorsque \halign lit son argument, elle compose *tous* les contenus des cellules et par conséquent, fixe comme longueur d'une colonne la plus grande des longueurs rencontrées dans cette colonne.

Du côté de la syntaxe, chaque ligne s'achève par la primitive \cr et il est prudent de placer \crcr en fin de tableau. Au sein d'une ligne, le token « & » (de catcode 4) signifie « passer à la colonne suivante ». La syntaxe d'une ligne est donc :

⟨*cellule 1*⟩&⟨*cellule 2*⟩& ... &⟨*cellule n*⟩\cr

Raffinement supplémentaire, il est possible pour toutes les cellules d'une même colonne de définir un code qui sera inséré avant leur contenu et un code qui sera inséré après. Pour cela, la primitive \halign admet un *préambule* qui est placé avant le premier \cr. La syntaxe de ce préambule reprend la syntaxe d'une ligne, mais le ⟨*contenu*⟩ d'une cellule est symbolisé par le token « # ». Ainsi, pour une colonne (c'est-à-dire compris entre deux « & »), si on écrit dans le préambule

⟨*code avant*⟩#⟨*code après*⟩

alors, le texte ⟨*code avant*⟩ sera inséré avant le code contenu dans les cellules et ⟨*code après*⟩ sera inséré après.

Dans l'exemple ci-dessous, la colonne n° 1 est définie pour que « X » soit affiché avant chaque cellule et « **\ignorespaces » après, où \ignorespaces nous assure qu'en dehors du préambule, les espaces insérés avant le premier & ne compteront pas. Pour la deuxième colonne, # est seul et donc les cellules de cette colonne seront composées sans aucun ajout. Enfin, la colonne n° 3 est composée au fer à droite en insérant \hfil à gauche de # :

Code n° III-245

```
\halign{X#**\ignorespaces&#&\hfil #\cr % préambule
foo             &foo bar&123 456\cr % première ligne
deuxième ligne &1       &12\cr % deuxième ligne
a               &\TeX    &1 2 3 4 5 6\cr % troisième ligne
\crcr}
```

```
Xfoo                **foo bar    123 456
Xdeuxième ligne **1                   12
Xa                  **TEX     1 2 3 4 5 6
```

Pour tirer parti de \halign dans le code de \cvtop, nous allons définir un alignement à une colonne et nous spécifierons « \hfil#\hfil » en préambule afin

2. Pour en savoir plus, il est sage de se tourner vers le TEXbook, chapitre 22.

que les contenus soient centrés. À l'aide de \substtocs, nous stockerons dans une macro \temp l'argument de \cvtop où les virgules auront été remplacées par \cr. Il suffira ensuite de placer la macro \temp juste après le \cr qui marque la fin du préambule pour que TEX la développe (car le début de chaque cellule est développé) pour constituer le corps du tableau :

Code n° III-246

```
\def\cvtop#1{%
  \vtop{% \vtop assure que l'on est en mode vertical
    \substtocs\temp{#1}{,}{\cr}% dans \temp, remplacer les "," par "\cr"\substtocs
    \halign{%
      \hfil##\hfil\cr% préambule : centrer le contenu
      \temp\crcr}% mettre \temp dans l'alignement
  }% \temp est détruite en sortant de la boite
}

texte avant...\cvtop{Ligne du haut,Très grande ligne du milieu,Ligne du bas pour finir}%
...texte après
```

texte avant... Ligne du haut ...texte après
Très grande ligne du milieu
Ligne du bas pour finir

Ce code en suggère un autre, capable de calculer à l'aide de \halign, la plus grande longueur de textes. Pour cela, la macro \calcmaxdim*, de syntaxe

\calcmaxdim\⟨*macro*⟩{⟨*texte 1*⟩,texte 2>,...,⟨*texte n*⟩}

stockera dans la \⟨*macro*⟩ la plus grande longueur des ⟨*texte*⟩.

Code n° III-247

```
\def\calcmaxdim#1#2{%
  \setbox0=\vtop{% \vtop assure que l'on est en mode vertical (interne)
    \substtocs\temp{#2}{,}{\cr}% dans \temp, remplacer les "," par "\cr"
    \halign{##\hfil\cr% préambule : composer au fer à gauche
      \temp\crcr}% mettre \temp dans l'alignement
  }% \temp est détruite en sortant de la boite
  \edef#1{\the\wd0 }%
}
a) La plus grande longueur vaut
   \calcmaxdim\foo{Ligne du haut,Très grande ligne du milieu,Ligne du bas pour finir}\foo

b) Vérification : \setbox0=\hbox{Très grande ligne du milieu}\the\wd0
```

a) La plus grande longueur vaut 89.9599pt
b) Vérification : 89.9599pt

8.4. Les réglures

8.4.1. Tracer des lignes

Nous allons maintenant nous intéresser à des boites un peu particulières, les boites de réglure ou « réglures », c'est-à-dire des boites entièrement remplies de noir.

82 - Règle

Les primitives \hrule et \vrule, qui opèrent respectivement en mode *vertical* et *horizontal*, tracent des réglures. Par défaut, les dimensions de ces réglures sont :

	\hrule	\vrule
largeur (width)	*	0.4pt
hauteur (height)	0.4pt	*
profondeur (depth)	0pt	*

L'étoile signifie que la dimension est celle de la boite qui contient la réglure.

Si l'on souhaite passer outre les valeurs par défaut et imposer une ou plusieurs dimensions, on doit faire suivre ces deux primitives d'un (ou plusieurs) mots-clés parmi « `width` », « `height` » ou « `depth` » et en respectant cet ordre. S'ils sont présents, ces mots-clés doivent être suivis de la ⟨*dimension*⟩ que l'on souhaite imposer.

Si plusieurs spécifications contradictoires sur une dimension sont spécifiées, la dernière est prise en compte.

Voici par exemple ce que l'on obtient si on écrit « \vrule » dans le code source immédiatement après ces deux points :|. On constate que la réglure, qui opère en mode horizontal, a la largeur par défaut de `0.4pt` et a pour hauteur et profondeur celles de la boite dans laquelle elle est enfermée. Cette boite est la ligne en cours de composition.

Passons maintenant à \hrule et voici l'effet de « \hrule » juste après ces deux points :

La réglure prend la totalité de la largeur de la boite qui la contient. La boite dont il est question dans ce cas est la page elle-même, de largeur \hsize, dans laquelle les éléments sont empilés verticalement. On remarque que pour satisfaire son mode de fonctionnement, \hrule passe en mode vertical. Puisque nous étions à l'intérieur d'un paragraphe, celui-ci a été composé avant le passage en mode vertical. Par ailleurs, le ressort d'interligne n'est pas inséré, ni avant la réglure ni après. Une réglure \hrule est donc dessinée *au plus près* verticalement de ce qui la suit et de ce qui la précède.

L'exemple suivant montre que les primitives de réglures utilisées sans mot-clé adaptent leurs dimensions à celles de la boite qui les contient. Ici, on dessine des \hrule au sommet et au bas d'une \vbox dans la première ligne alors que l'on trace des \vrule avant et après une \vtop à la deuxième ligne :

Code n° III-248

```
a) .......\vbox{\hrule\hbox{foo}\hbox{ligne du bas}\hrule}.......\medbreak
b) .......\vrule\vtop{\hbox{foo}\hbox{ligne du bas}}\vrule.......
```

```
a) .....foo
       ligne du bas.....

b) .....|foo          |.....
       |ligne du bas|
```

On voit clairement au cas a que les \hrule à l'intérieur de \vbox prennent chacune la largeur totale de la boite. En ce qui concerne les \vrule qui se trouvent avant et après la \vtop au cas b, elles ont adapté leur profondeur et hauteur pour s'étendre sur la hauteur totale de la boite qui ici est la ligne en cours.

Voici maintenant des réglures dont les dimensions sont imposées par mots-clés :

Code n° III-249

```
Une réglure de 1.5cm :\hrule width1.5cm
foo\vrule width 2pt height .5cm depth .2cm bar
```

Une réglure de 1.5cm :

foo bar

Ici encore, la `\vrule` est en contact avec la `\hrule` puisque le ressort d'interligne n'est pas inséré.

■ **Exercice 88**

Écrire une macro `\showdim`*, opérant en mode horizontal, dont l'argument est une dimension et qui représente sous forme d'une réglure horizontale la dimension donnée.

Par exemple, « `\showdim{1.5cm}` » produit « ⌊——————⌋ » où les traits verticaux de début et de fin ont une hauteur de `1ex`. Toutes les réglures ont une épaisseur de `0.4pt`.

□ **Solution**

Voici le code de cette macro :

Code n° III-250

```
\def\showdim#1{%
  \vrule width 0.4pt height 1ex  depth 0pt % trait vertical gauche
  \vrule width #1    height0.4pt depth 0pt % réglure horizontale de longueur #1
  \vrule width 0.4pt height 1ex  depth 0pt % trait vertical droit
  \relax% stoppe la lecture de la précédente dimension
}

a) une longueur de 1 cm : \showdim{1cm}\par
b) une longueur de 137,4 pt : \showdim{137,4pt}\par
c) une longueur de 2 mm : \showdim{2mm}
```

a) une longueur de 1 cm : ⌊——⌋
b) une longueur de 137,4 pt : ⌊————————————————————————⌋
c) une longueur de 2 mm : ⌊⌋

Le code est très simple. En premier lieu (ligne n° 2), la primitive `\vrule` qui opère en mode horizontal trace une barre verticale d'`1ex` de hauteur et de `0.4pt` d'épaisseur. Ensuite, à la ligne n° 3, nous la faisons suivre d'une ligne horizontale dont la longueur a été forcée à la dimension contenue dans l'argument et dont la hauteur est `0.4pt`. Nous finissons à la ligne n° 4 avec la même barre verticale qu'au début. ■

8.4.2. Encadrer

On commence à deviner qu'il serait assez facile de programmer une macro `\FRbox`* qui encadre tout ce qui est dans son argument, argument devant être composé en mode horizontal :

`\FRbox{`⟨*contenu*⟩`}`

Les deux paramètres importants pour l'aspect visuel sont l'épaisseur des réglures et l'espacement entre elles et le contenu. Chacun de ces paramètres sera un

registre de dimension ; `\frboxrule*` pour l'épaisseur et `\frboxsep*` pour l'espacement [3].

En ce qui concerne la méthode pour encadrer un contenu arbitraire, il y a deux façons différentes de tracer les réglures :

⟨*contenu*⟩ ⟨*contenu*⟩

Décidons-nous par exemple pour la structure de droite. Pour arriver à nos fins, nous allons construire une `\vbox` dans laquelle nous allons insérer, de haut en bas :

- une réglure horizontale ;
- une espace verticale égale à `\frboxsep` ;
- le contenu (précédé et suivi d'espaces égales à `\frboxsep`) ;
- une espace verticale égale à `\frboxsep` ;
- une réglure horizontale.

Nous obtiendrons donc :

⟨*contenu*⟩

Qu'ils soient horizontaux ou verticaux, tous les espaces seront insérés avec la primitive `\kern` qui agit selon le mode en cours. Voici la construction obtenue pour un contenu égal au mot « Programmation » :

Code n° III-251

```
\newdimen\frboxrule \newdimen\frboxsep
\frboxsep=5pt \frboxrule=1pt
\leavevmode
...%
\vbox{%
  \hrule height\frboxrule% réglure supérieure
  \kern\frboxsep% espace verticale haute
  \hbox{\kern\frboxsep Programmation\kern\frboxsep}% contenu + espaces horizontales
  \kern\frboxsep% espace verticale basse
  \hrule height\frboxrule% réglure inférieure
  }%
...
```

Programmation

Il ne reste plus qu'à envelopper le tout dans une boite horizontale `\hbox`, après avoir fait précéder la `\vbox` de `\vrule` qui seront les réglures verticales à droite et à gauche de l'encadrement :

Code n° III-252

```
\newdimen\frboxrule \newdimen\frboxsep
\frboxsep=5pt \frboxrule=0.6pt
\def\FRbox#1{% /!\ ne change pas le mode H ou V en cours
  \hbox{% mettre à la suite horizontalement les 3 choses suivantes :
```

3. Ne pas confondre les noms de ces registres avec `\fboxrule` et `\fboxsep` qui sont les registres définis par LaTeX pour les boites encadrées avec la macro `\fbox`.

```
    \vrule width\frboxrule% 1) réglure gauche
    \vbox{%                 2) un empilement vertical comprenant
      \hrule height\frboxrule% a) réglure supérieure
      \kern\frboxsep%          b) espace verticale haute
      \hbox{%                  c) contenu + espaces en mode H
        \kern\frboxsep#1\kern\frboxsep
      }%
      \kern\frboxsep%          d) espace verticale basse
      \hrule height\frboxrule% e)réglure inférieure
      }%
    \vrule width\frboxrule% 3) réglure droite
  }%
}
Ligne de base : ...\FRbox{Programmation}...%
\frboxrule=2pt \FRbox{Programmation}...%
\frboxsep=0pt \FRbox{Programmation}...%
\frboxrule0.4pt \FRbox{Programmation}...
```

Ligne de base : ... Programmation Programmation Programmation Programmation ...

On constate que la ligne de base du contenu des boites n'est pas alignée avec la ligne de base de la ligne en cours. En fait, comme la \vbox empile les éléments au-dessus de la ligne de base du dernier élément, le point de référence de la \vbox se trouve sur la ligne de base de la réglure inférieure tracée à la ligne nº 13, c'est-à-dire au niveau du bord *inférieur* de cette réglure [4]. Le problème qui se pose est le suivant : comment construire une boite verticale où l'on place des listes de matériels verticaux

⟨*liste a*⟩
⟨*liste b*⟩

et faire en sorte que le point de référence de la boite ainsi construite soit sur la ligne de base du dernier élément vertical de ⟨*liste a*⟩ ? Faire appel à \vbox ou \vtop seules ne peut convenir. Il fait *combiner* ces deux boites. L'idée est d'enfermer ⟨*liste a*⟩ dans une \vbox et mettre cette \vbox comme premier élément vertical d'une \vtop qui englobe le tout. La boite ainsi construite a la structure suivante :

```
\vtop{%
    \vbox{⟨liste a⟩}
    ⟨liste b⟩
}
```

De par les définitions de \vbox et \vtop, le point de référence de la boite externe \vtop se trouvera sur la ligne de base du dernier élément vertical de ⟨*liste a*⟩. Voici un exemple qui illustre cette manipulation :

Code nº III-253

```
Persuadez-vous que :
\vtop{
  \vbox{
    \hbox{Programmer}
    \hbox{en}
    \hbox{\TeX}
```

4. Si nous avions écrit depth au lieu de height à la ligne nº 13, le point de référence aurait été au niveau du bord *supérieur* de la réglure du bas.

```
    \hbox{est}% <- ligne de base de la \vtop
  }
  \hbox{\it tout sauf}
  \hbox{facile.}
}
```

Persuadez-vous que : Programmer en TEX est *tout sauf* facile.

Nous allons appliquer cette structure pour créer une nouvelle macro \frbox afin que, contrairement à \FRbox, le point de référence du cadre soit au niveau de la ligne de base du contenu. Il faut donc rejeter hors de la \vbox les matériels verticaux des lignes nos 12 et 13 :

Code no III-254

```
%\newdimen\frboxrule \newdimen\frboxsep
\frboxrule=0.4pt \frboxsep=2pt
\def\frbox#1{% ne pas changer le mode H ou V en cours
  \hbox{% enferme dans une \hbox
    \vrule width\frboxrule% réglure gauche
    \vtop{%
      \vbox{% 1er élément de la \vtop
        \hrule height\frboxrule% réglure supérieure
        \kern\frboxsep% espace haut
        \hbox{%
          \kern\frboxsep% espace gauche
          #1% contenu
          \kern\frboxsep% espace droite
          }%
      }% puis autres éléments de la \vtop, sous la ligne de base
      \kern\frboxsep% espace bas
      \hrule height\frboxrule% réglure inférieure
    }%
    \vrule width\frboxrule% réglure droite
  }%
}
Ligne de base : ......\frbox{Programmation}......%
\frboxrule=2pt \frbox{Programmation}......%
\frboxsep=0pt \frbox{Programmation}......%
\frboxrule0.4pt \frbox{Programmation}......
```

Ligne de base : ..... Programmation ..... Programmation ..... Programmation ..... Programmation .....

■ Exercice 89

Créer une macro \centretitre*{⟨*texte*⟩} dont l'argument doit être composé dans un cadre (via \frbox) qui prend toute la largeur de composition, c'est-à-dire \hsize. L'argument doit être centré dans ce cadre.

□ Solution

Tout d'abord, nous allons placer en début de macro un \medbreak ce qui aura pour effet de composer le paragraphe en cours si la macro est appelée en mode horizontal et sautera une espace verticale. Ensuite, la primitive \noindent interdira le placement de la boite d'indentation en début de paragraphe, et provoquera le passage en mode horizontal.

Nous allons composer le ⟨*texte*⟩ dans une boite verticale \vbox. Sa dimension horizontale, spécifiée par \hsize, doit tenir compte des réglures droite et gauche (chacune de dimension \frboxrule) ainsi que les espaces à droite et à gauche entre le ⟨*texte*⟩ et les réglures (chacun de dimension \frboxsep). La dimension horizontale de la boite sera donc

$$\texttt{\hsize} - 2\texttt{\frboxrule} - 2\texttt{\frboxrule}$$

Pour que le centrage soit effectif, nous jouerons sur les ressorts \leftskip et \rightskip sans oublier d'annuler le ressort de fin de paragraphe \parfillskip.

Code n° III-255

```
\def\centretitre#1{%
  \medbreak% passe en mode v puis saute une espace verticale
  \noindent% pas d'indentation et passe en mode horizontal
  \frbox{% encadre
    \vbox{% une boite verticale
      \hsize=\dimexpr\hsize-2\frboxrule-2\frboxsep\relax
      \parindent=0pt % pas d'indentation
      \leftskip=0pt plus1fil \rightskip=\leftskip% ressorts de centrage
      \parfillskip=0pt % annule le ressort de fin de paragraphe
      #1% insère le titre
      \endgraf% et le compose
      }%
  }%
  \medbreak% passe en mode v puis saute une espace verticale
  \ignorespaces% mange les espaces situés après la macro \centretitre
}
\frboxrule=0.8pt \frboxsep=5pt
Voici un
\centretitre{Titre}
puis un
\centretitre{Texte très long pour composer un titre qui va prendre plusieurs
lignes et pour s'assurer que la composition s'effectue correctement}
```

Voici un

Titre

puis un

Texte très long pour composer un titre qui va prendre plusieurs lignes et pour s'assurer que la composition s'effectue correctement

■

■ Exercice 90

Programmer une macro \souligne*{⟨*texte*⟩} qui souligne d'un trait de 0.4pt d'épaisseur le ⟨*texte*⟩. Le trait de soulignement devra se situer à 1pt au-dessous du ⟨*texte*⟩ :

\souligne{Du texte}	affiche	Du texte
\souligne{Du texte profond}	affiche	Du texte profond

Proposer une variante \Souligne* où le trait devra se situer à 1pt au-dessous de la ligne de base, quelle que soit la profondeur du texte à souligner :

\Souligne{Du texte}	affiche	Du texte
\Souligne{Du texte profond}	affiche	Du texte profond

□ **Solution**

Nous devons tracer une réglure ayant comme dimension horizontale celle de la `\hbox` dans laquelle est placé l'argument. Afin de pouvoir mesurer cette boite, nous allons stocker `\hbox{#1}` dans le registre de boite n° 0.

Mais avant de poursuivre, regardons d'un peu plus près les dimensions qui suivent les mots-clés `height`, `width` et `depth`, et considérons-les comme des dimensions *signées*. Car c'est ainsi que TEX travaille ; rien ne dit qu'une dimension, fût-elle de réglure, doit être positive ! Par exemple, pour tracer une réglure de `0.4pt` d'épaisseur à `10pt` de hauteur, nous pouvons fixer sa hauteur à `10.4pt` et sa profondeur à `-10pt` :

Code n° III-256

```
Une réglure en hauteur : \vrule width 1cm height 10.4pt depth -10pt
```

Une réglure en hauteur :

De même, pour tracer une réglure d'épaisseur `0.4pt` à `2pt` sous la ligne de base, il suffit de fixer sa profondeur à `2.4pt` et sa hauteur à `-2pt` :

Code n° III-257

```
Une réglure en dessous: \vrule width 1cm depth 2.4pt height -2pt
```

Une réglure en dessous :

Revenons à notre problème… Nous stockerons la `\hbox` dans le registre de boite n° 0. Puis, nous allons appliquer l'astuce exposée ci-dessus pour tracer une réglure de `0.4pt` d'épaisseur à une distance de `\dp0+1pt` sous la ligne de base. Pour cela, nous allons stocker dans ce même registre n° 0 une `\vrule` dont la longueur est celle du texte à souligner, soit `\wd0`, de profondeur `\dp0+1.4pt` et de hauteur négative égale à `-\dp0-1pt`. Nous allons ensuite rendre nulles toutes les dimensions de la boite afin que son encombrement soit nul dans toutes les directions. Enfin, après être éventuellement sorti du mode vertical, nous affichons cette boite n° 0 avant le texte :

Code n° III-258

```
\def\souligne#1{%
  \setbox0=\hbox{#1}% stocke le contenu dans le registre no 0
  \setbox0=\hbox{% puis, dans une \hbox, construit une réglure
    \vrule width\wd0 % de la longueur du contenu
      depth\dimexpr\dp0 + 1.4pt\relax % dp = profondeur texte + 1.4pt
      height\dimexpr-\dp0 - 1pt\relax % ht = -profondeux texte - 1pt
  }%
  \wd0=0pt \dp0=0pt \ht0=0pt % annule toutes les dimensions
  \leavevmode \box0 % affiche la réglure
  #1% puis le contenu
}
Voici \souligne{du texte normal}.\par
Voici \souligne{du texte profond}.
```

Voici du texte normal.
Voici du texte profond.

Compte tenu des dimensions utilisées, le bord supérieur de la réglure est très exactement à `1pt` sous la plus basse partie du texte à souligner.

La variante `\Souligne` est plus simple : il ne faut pas tenir compte de la profondeur du registre n° 0.

Code n° III-259

```
\def\Souligne#1{%
  \setbox0=\hbox{#1}%
  \setbox0=\hbox{\vrule width\wd0 depth1.4pt height-1pt }%
  \wd0=0pt \dp0=0pt \ht0=0pt
  \leavevmode \box0 #1%
}
Voici \Souligne{du texte normal}.\par
Voici \Souligne{du texte profond}.
```

Voici du texte normal.
Voici du texte profond.

Les macros présentées ici ne permettent pas de souligner un texte trop long ou trop proche d'une fin de ligne, car une réglure, *a fortiori* enfermée dans une \hbox, ne peut franchir une coupure de ligne.

Une autre méthode, impliquant \lower, aurait également été envisageable :

Code n° III-260

```
\def\Souligne#1{%
  \setbox0=\hbox{#1}%
  \lower 1pt % abaisser à 1pt sous la ligne de base
    \rlap{% une \hbox en surimpression vers la droite
      \vrule width\wd0 height0pt depth0.4pt % contenant le soulignement
    }%
  #1% puis afficher le <texte>
}
Voici \Souligne{du texte normal}.\par
Voici \Souligne{du texte profond}.
```

Voici du texte normal.
Voici du texte profond.

La méthode n'est pas équivalente car dans ce dernier cas, la réglure qui tient lieu de soulignement est prise en compte dans la boite englobante de \Souligne⟨*texte*⟩, ce qui n'était pas le cas dans le premier code. ■

8.4.3. Empiler des boites

Essayons maintenant de jouer avec les boites et pour cela, créons une macro \stackbox⋆ capable d'empiler des boites encadrées, toutes identiques, aussi bien horizontalement que verticalement. Le fait qu'elles soient identiques implique évidemment que les dimensions de leurs contenus ne sont pas trop différentes. La syntaxe de la macro \stackbox pourrait être :

```
\stackbox{a,bc,,d\\e,foobar,g\\123,456,$\alpha$,$x+y$,}
```

Et le résultat serait :

a	bc		d	
	e	foobar	g	
123	456	α	$x+y$	

En ce qui concerne les dimensions des boites, on pourrait parcourir lors d'une première passe tous les arguments afin de trouver la plus grande des hauteurs,

largueurs et profondeurs et ensuite prendre ces maximums comme dimension de chaque boite, qui seraient alors composées lors d'une seconde passe. Afin de ne pas inutilement compliquer le code [5], nous allons plutôt décider que l'utilisateur va lui-même fixer la largueur *interne* des cadres à l'aide d'un registre de dimension `\stackwd` créé pour l'occasion : ceci suppose évidemment qu'aucune boite ne doit avoir un contenu dont la largeur excède `\stackwd`.

Parcourir l'ensemble des contenus va être facilitée avec la macro `\doforeach` à qui nous donnerons la liste d'éléments d'une même ligne. Tout cela se fait aisément avec une macro `\stackbox@i` à arguments délimités par « `\\` » dont le but est d'isoler et lire la ligne en cours. Son texte de paramètre sera :

```
\def\stackbox@i#1\\{...}
```

En premier lieu, la macro chapeau se contentera d'appeler la macro récursive `\stackbox@i` de cette façon :

```
\stackbox@i#1\\\quark\\
```

Ceci a deux avantages : c'est l'assurance que `\\` est présent dans l'argument, car après tout, on pourrait ne pas l'écrire dans l'argument de `\stackbox` si on ne voulait qu'une seule ligne. L'autre avantage est que la dernière ligne à lire est le quark « `\quark` ». Si la `\ifx`-égalité de l'argument `#1` avec `\quark` est vraie, nous serons alors surs que la ligne précédente était la dernière et donc, finir le processus. Nous aurons donc lu la totalité des lignes une première fois dans la macro chapeau puis chaque ligne une seule fois dans la macro récursive. Chaque ligne a été donc lue deux fois exactement. C'est la méthode préconisée par la règle de la page 203 pour lire une série d'arguments.

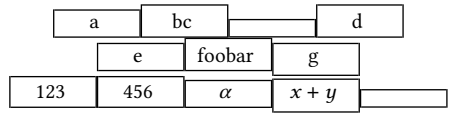
La macro `\nobreak`, insérée avant chaque ligne restante, interdit toute coupure de page entre deux lignes de boites consécutives. Cette macro est définie dans plain-TEX et son texte de remplacement est « `\penalty10000` ». Sachant que toute pénalité supérieure à 10 000 est comprise comme 10 000, il s'agit là de la plus forte pénalité possible ce qui explique qu'une coupure de page ne se fera jamais à cet endroit.

Code n° III-261

```
\newdimen\stackwd \stackwd=3em % dimension horizontale interne des cadres
\catcode`@11
\def\stackbox#1{%
  \par% termine le paragraphe en cours
  \noindent
  \stackbox@i#1\\\quark\\% ajoute "\\\quark\\" à la fin et appelle \stackbox@i
  \par
}

\def\stackbox@i#1\\{% #1=ligne courante
  \def\temp@{#1}% stocke la ligne courante
  \unless\ifx\quark\temp@% si ce n'est pas la fin
    \hfill % ressort infini de centrage (et fait passer en mode horizontal)
    \noindent
    \doforeach\current@item\in{#1}% pour chaque élément dans la ligne courante...
      {\frbox{% ...encadrer
        \hbox to\stackwd{% une \hbox de largeur \stackwd
          \hss% ressort de centrage
          \current@item% l'élément courant
```

5. La vraie et inavouable raison est que renoncer à faire deux passes va nous donner l'occasion de découvrir les « struts » !

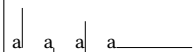
```
          \hss% ressort de centrage
          }
        }% fin de la \frbox
      }% fin \doforeach
    \hfill% ressort infini de centrage
    \null% assure que le dernier ressort est pris en compte
    \par% finir le paragraphe
    \nobreak% interdire une coupure de page
    \nointerlineskip% ne pas insérer le ressort d'interligne
    \expandafter\stackbox@i% et recommencer
  \fi
}
\catcode`@12

\frboxrule=0.5pt \frboxsep=3pt
\stackbox{a,bc,,d\\e,foobar,g\\123,456,$\alpha$,$x+y$,}
```



Le résultat est loin d'être celui escompté ! Quoique pas si loin que ça, après tout : les positions horizontales sont bonnes. Le bug vient du fait que nous n'avions pas prévu que les dimensions verticales (hauteur et profondeur) des éléments influaient naturellement sur la boite encadrée construite. On peut y remédier en insérant un « strut [6] » : c'est une réglure invisible, mais qui s'étend sur une dimension horizontale ou verticale. La question que l'on se pose immédiatement est « comment rendre une réglure invisible ? » Tout simplement en mettant sa largeur à `0pt`, auquel cas elle aura une dimension verticale ou bien en mettant sa hauteur et profondeur à `0pt` et dans ce cas, elle aura alors une dimension horizontale. Comme il est assez difficile de montrer des choses invisibles, voici une réglure de type `\vrule` rendue visible (son épaisseur est `0.2pt`) après la lettre « a » :

Code n° III-262

```
a\vrule width0.2pt height15pt depth0pt \quad
a\vrule width0.2pt height0pt depth5pt \quad
a\vrule width0.2pt height10pt depth10pt \quad
a\vrule width1cm height0.2pt depth0pt
```



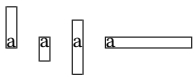
Il suffit d'imposer `0pt` comme dimension d'épaisseur pour rendre cette réglure invisible, mais bien réelle en ce qui concerne son encombrement. Voici comment la mettre en évidence en encadrant au plus proche l'ensemble formé par la lettre « a » et le strut :

Code n° III-263

```
\frboxsep0pt %encadrement au plus proche
\leavevmode
\frbox{a\vrule width0pt height15pt depth0pt }\quad
\frbox{a\vrule width0pt height0pt depth5pt }\quad
```

6. On dit « poutre » en français mais ce mot est assez peu usité.

```
\frbox{a\vrule width0pt height10pt depth10pt }\quad
\frbox{a\vrule width1cm height0pt depth0pt }
```



■ Exercice 91

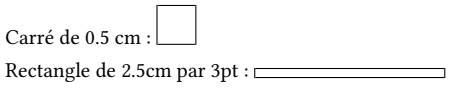
Construire une macro `\rectangle*{⟨x⟩}{⟨y⟩}` où ⟨x⟩ et ⟨y⟩ sont des dimensions et qui construit un rectangle dont les dimensions internes (c'est-à-dire de bord interne à bord interne) sont ⟨x⟩ et ⟨y⟩.

□ Solution

Il faut construire une `\frbox` dans laquelle nous allons mettre *deux* struts, l'un pour la dimension verticale et l'autre pour l'horizontale.

Code n° III-264

```
\def\rectangle#1#2{%
  \begingroup% dans un groupe
    \frboxsep = 0pt % encadrer au plus proche
    \frboxrule= 0.4pt % en traits assez fins
    \frbox{%
      \vrule width#1 height0pt depth0pt %strut horizontal
      \vrule width0pt height#2 depth0pt %strut vertical
    }%
  \endgroup% fermer le groupe
}
Carré de 0.5 cm : \rectangle{0.5cm}{0.5cm}\smallskip

Rectangle de 2.5cm par 3pt : \rectangle{2.5cm}{3pt}
```



■

Pour nos boites créées avec `\stackbox`, nous allons mettre dans chacune d'elles un strut dont les dimensions verticales sont les plus grandes que l'on peut rencontrer dans un texte « normal ». Nous prendrons les dimensions verticales d'une `\hbox` contenant « Àgjp ». Bien évidemment, si un élément contient un texte dont les dimensions verticales excèdent celles du strut, le problème réapparaitra.

Il y a un autre défaut, plus difficile à détecter dans notre macro `\stackbox` : les réglures des boites adjacentes ne se superposent pas, aussi bien horizontalement que verticalement, ce qui fait que leurs épaisseurs s'additionnent. Lorsque `\frboxrule` est faible (`0.5pt` ici), cela n'a pas d'influence visuelle notable, mais ce défaut serait beaucoup plus visible si `\frboxrule` valait 2pt par exemple. Nous allons donc insérer une espace négative avec `\kern-\frboxrule` entre chaque boite horizontalement et entre chaque ligne verticalement. Le dernier `\kern` horizontal, qui viendra après la dernière boite dans la boucle, sera annulé par la primitive `\unkern` que nous mettons après la boucle :

Code n° III-265

```
\newdimen\stackwd \stackwd=3em
\catcode`\@11
\def\stackbox#1{%
  \par% termine le paragraphe en cours
  \begingroup% dans un groupe semi-simple
    \parindent=0pt% pas d'indentation
    \parskip=0pt% annuler le \parskip
    \setbox0\hbox{Àgjp}% boite pour le strut
    \edef\stack@strut{\vrule width\z@ height\the\ht0 depth\the\dp0 }% définit le strut
    \stackbox@i#1\\\quark\\% ajoute "\\\quark\\" à la fin et appelle \stackbox@i
    \unkern% annule la dernière compensation verticale
    \par
  \endgroup
}
\def\stackbox@i#1\\{% #1=ligne courante
  \def\temp@{#1}% stocke la ligne courante
  \unless\ifx\quark\temp@% si ce n'est pas la fin
    \hfill % ressort infini de centrage (passe en mode horizontal)
    \doforeach\current@item\in{#1}% pour chaque élément dans la ligne courante...
      {\frbox{% ...encadrer
        \hbox to\stackwd{% une \hbox de largeur \stackwd contenant
          \hss%         1) ressort de centrage
          \stack@strut% 2) strut de dimension verticale
          \current@item%3) l'élement courant
          \hss}%        4)ressort de centrage
        }% fin de la \fbox
      \kern-\frboxrule% revenir en arrière pour superposer les réglures verticales
      }% fin de \doforeach
    \unkern% annuler la dernière compensation horizontale
    \hfill% ressort infini de centrage
    \null% fait prendre en compte le dernier ressort
    \par% termine le paragraphe
    \nobreak% interdit une coupure de page
    \nointerlineskip% sinon, ne pas ajouter le ressort d'interligne
    \kern-\frboxrule% superposer les réglures horizontales
    \expandafter\stackbox@i% et recommencer
  \fi
}
\frboxrule=0.5pt
\frboxsep=3pt
\stackbox{a,bc,,d\\e,foobar,g\\123,456,$\alpha$,$x+y$,}
```

a	bc		d	
	e	foobar	g	
123	456	α	$x+y$	

■ Exercice 92

Modifier la macro `\stackbox` en `\lineboxed*` de telle sorte que pour chaque ligne, les boites aient une largeur telle qu'elles occupent la totalité de la ligne.

□ Solution

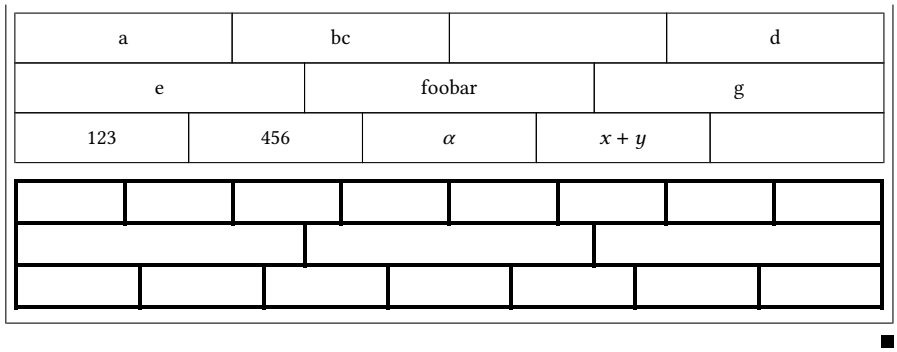
Pour que la largeur des boites d'une ligne soit telle que toutes ces boites remplissent la ligne, nous devons d'abord savoir combien il y a de boites dans chaque ligne. Pour cela, il faut compter combien de virgules il y a dans l'argument `#1` et ajouter 1 ou, méthode plus simple, rajouter une virgule à `#1` et effectuer le comptage. La macro `\cnttimestocs` se chargera de ce travail et assignera le nombre de boites *n* dans une macro.

La largeur disponible pour tous les contenus cumulés de toutes les boites de la ligne s'obtient en retirant de \hsize l'épaisseur \frboxrule des réglures verticales et les espaces \frboxsep entre ces réglures et les contenus des boites. S'il y a n boites, il y a $n+1$ réglures verticales et $2n$ espaces \frboxsep. La largeur interne d'une boite, stockée dans la macro \dim@box sera donc

$$\frac{\hsize - (n+1)\frboxrule - 2n\frboxsep}{n}$$

Code n° III-266

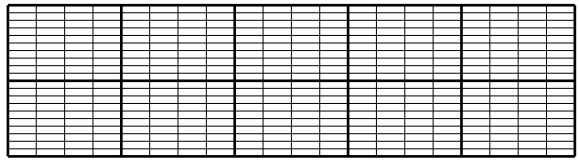
```
\catcode`\@11
\def\lineboxed#1{%
  \par% termine le paragraphe en cours
  \begingroup% dans un groupe semi-simple
    \parindent=0pt% pas d'indentation
    \parskip=0pt% annuler le \parskip
    \setbox0\hbox{Àgjp}% boite pour le strut
    \edef\stack@strut{\vrule width\z@ height\the\ht0 depth\the\dp0 }% définit le strut
    \lineboxed@i#1\\\quark\\% ajoute "\\\quark\\" à la fin et appelle la macro récursive
    \unkern% annule la dernière compensation verticale
    \par
  \endgroup
}
\def\lineboxed@i#1\\{% #1=ligne courante
  \def\temp@{#1}% stocke la ligne courante
  \unless\ifx\quark\temp@% si ce n'est pas la fin
    \cnttimestocs{#1,}{,}\nb@args% reçoit le nombre d'arguments dans la ligne courante
    \edef\dim@box{\the\dimexpr(\hsize-\frboxrule*(\nb@args+1)-
                  \frboxsep*2*\nb@args)/\nb@args}%
    \hbox{%
      \doforeach\current@item\in{#1}% pour chaque élément dans la ligne courante...
        {\frbox{% ...encadrer
          \hbox to\dim@box{% une \hbox de largeur \dim@box contenant
            \hss%          1) ressort de centrage
            \stack@strut% 2) strut de dimension verticale
            \current@item%3) l'élement courant
            \hss}%         4)ressort de centrage
          }% fin de la \fbox
        \kern-\frboxrule% revenir en arrière pour superposer les réglures verticales
        }% fin de \doforeach
      \unkern% annuler la dernière compensation horizontale
    }%
    \par% termine le paragraphe
    \nobreak% interdit une coupure de page
    \nointerlineskip% sinon, ne pas ajouter le ressort d'interligne
    \kern-\frboxrule% superposer les réglures horizontales
    \expandafter\lineboxed@i% et recommencer
  \fi
}
\catcode`\@12
\lineboxed{a,bc,,d\\e,foobar,g\\123,456,$\alpha$,$x+y$,}\medbreak

\frboxrule=1.5pt
\frboxsep=3pt
\lineboxed{,,,,,,,\\,,\\,,,,,,}
```



8.4.4. Dessiner des quadrillages

Comment parler de réglure sans parler de quadrillage ? Car s'il est bien un domaine où elles sont adaptées, c'est bien celui-là.

Mettons à profit ce que nous savons pour écrire une macro `\grid*`, capable de dessiner des quadrillages tels que celui-ci :



Avant d'entrer dans le vif du sujet, quel est le nombre de paramètres qu'il est nécessaire de spécifier ? L'observation de ce quadrillage permet de se faire une idée. Il faut définir la largeur et hauteur d'un carreau-unité que nous stockerons dans deux registres de dimension (`\xunit` et `\yunit`) ainsi que l'épaisseur des lignes principales et secondaires stockées dans deux autres registres de dimension (`\mainrule` et `\subrule`). Puis, il faut passer à la macro `\grid` le nombre de carreaux-unité que l'on veut horizontalement et verticalement ainsi que le nombre de subdivisions horizontales et verticales dans chaque carreau. Il semble donc naturel de construire une macro utilisant 4 registres de dimension et admettant 4 arguments.

Le quadrillage ci-dessus a été tracé en exécutant

```
\grid{5}{4}{2}{10}
```

pour demander qu'il comporte 5 carreaux-unité horizontalement, chacun divisé en 4 et 2 carreaux-unité verticalement, chacun divisé en 10. De plus, les registres de dimension ont été initialisés avec

```
\xunit=1.5cm \yunit=1cm \mainrule=1pt \subrule=0.2pt
```

Réglures horizontales

Construisons la macro `\grid` pas à pas et décidons que cette macro mettra des éléments dans une `\vbox`.Nous allons dans un premier temps empiler des éléments dans une liste verticale pour tracer les réglures horizontales. Il est bien entendu que l'épaisseur des réglures ne doit pas perturber les espacements verticaux, c'est pourquoi elles doivent être mises dans une sous-boite de hauteur nulle. Nous allons

donc définir l'analogue de \clap mais pour les \vbox. La macro suivante, \vlap* enferme son argument dans une boite verticale et rend la hauteur du tout égale à 0pt :

```
\def\vlap#1{\vbox to0pt{\vss#1\vss}}
```

Chaque réglure horizontale doit avoir comme largeur \xunit*#1, que nous stockerons dans une macro pour l'utiliser ensuite. Pour que les espacements verticaux soient rigoureusement exacts, le ressort inter-ligne doit être neutralisé et pour cela, \offinterlineskip sera appelé au début de la \vbox. Voici un premier essai où ne sont construites que les réglures horizontales principales. On prend les unités égales à 0,5 cm par souci de gain de place.

Code n° III-267

```
\xunit=0.5cm \yunit=0.5cm \mainrule=0.8pt \subrule=0.2pt
\def\vlap#1{\vbox to0pt{\vss#1\vss}}
\catcode`@11
\def\grid#1#2#3#4{%
  \vbox{% empiler les éléments verticalement
    \offinterlineskip% pas de ressort d'interligne
    \edef\total@wd{\the\dimexpr\xunit*#1\relax}% largeur totale ds réglures
    \for\ii = 1 to #3 \do1% pour chaque carreau vertical (\ii=variable muette)
      {\vlap{\hrule width\total@wd height\mainrule}% tracer la réglure horizontale
      \kern\yunit% insérer l'espace vertical
      }%
    \vlap{\hrule width\total@wd height\mainrule}% dernière réglure horizontale
  }%
}
\catcode`@12
\setbox0\hbox{\grid{4}{}{3}{}}% range la quadrillage dans le registre no 0
Essai \copy0{} qui a pour
largeur=\convertunit{\wd0}{cm} cm et pour hauteur=\convertunit{\ht0}{cm} cm
```

Essai _____ qui a pour largeur=2 cm et pour hauteur=1.5 cm

Pour s'assurer que les dimensions du quadrillage sont parfaitement correctes, le quadrillage a été placé dans une \hbox stockée dans le registre de boite n° 0. Ainsi, nous pouvons mesurer sa largeur et sa hauteur (converties en cm avec la macro \convertunit vue précédemment) : les dimensions obtenues sont bien 2 cm de largeur et 1,5 cm en hauteur qui correspondent à 4\xunit et à 3\yunit.

Venons-en aux lignes horizontales de subdivisions. Elles seront placées immédiatement *sous* la position courante de la réglure principale en cours, moyennant un ressort \vss inséré en fin de \vbox. Le code suivant illustre cette structure :

Code n° III-268

```
Début\vbox to0pt{\hbox{sous}\hbox{la ligne de base}\vss}suite
```

Début$_{\text{sous}}$ suite
la ligne de base

Dans le cas présent, nous placerons sous la position de la réglure principale en cours le matériel suivant, répété #4 − 1 fois :

– l'espace verticale de dimension \yunit/#4 ;
– la réglure secondaire, enfermée dans une \vlap, de façon à ce que son épaisseur n'entre pas en ligne de compte.

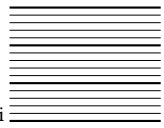
Remarquons qu'il n'y a pas lieu de tester si #4 > 1, car la boucle \for suivante

\for\jj = 2 to #4 \do 1 {⟨*code à exécuter*⟩}

ne s'exécutera que si le test est vrai.

Code n° III-269

```
\xunit=0.5cm \yunit=0.5cm \mainrule=0.8pt \subrule=0.2pt
\def\vlap#1{\vbox to0pt{\vss#1\vss}}
\catcode`@11
\def\grid#1#2#3#4{%
  \vbox{% empiler les éléments verticalement
    \offinterlineskip% pas de ressort d'interligne
    \edef\total@wd{\the\dimexpr\xunit*#1\relax}% largeur totale de la boite
    \edef\sub@unit{\the\dimexpr\yunit/#4\relax}% hauteur verticale de la subdivision
    \for\ii = 1 to #3 \do% pour chaque unité verticale en partant du haut, tracer :
      {\vlap{\hrule width\total@wd height\mainrule}% la réglure horizontale principale
      % et dessous, les réglures horizontales secondaires :
      \vbox to\z@{% dans une \vbox de hauteur nulle,
        \for\jj = 2 to #4 \do 1% insérer #4-1 fois sous la position courante :
          {\kern\sub@unit % l'espace verticale
          \vlap{\hrule width\total@wd height\subrule}% et la réglure secondaire
          }%
        \vss% ressort qui se comprime pour satisfaire la hauteur nulle
      }%
      \kern\yunit% insérer l'espace vertical entre réglures principales
      }%
    \vlap{\hrule width\total@wd height\mainrule}% dernière réglure principale du bas
  }%
}
\catcode`@12
\setbox0=\hbox{\grid{4}{}{3}{5}}
Essai \copy0{} qui a pour
largeur=\convertunit{\wd0}{cm} cm et pour hauteur=\convertunit{\ht0}{cm} cm
```



Essai qui a pour largeur=2 cm et pour hauteur=1.5 cm

Réglures verticales

Pour les réglures verticales, nous allons procéder exactement de la même façon qu'avec les réglures horizontales. Nous allons les tracer avant les réglures horizontales et le tout sera enfermé dans une \vbox chapeau contenant la totalité du quadrillage.

Afin que ces réglures verticales ne prennent aucun encombrement vertical, nous les enfermerons dans une \vbox de hauteur nulle et ferons en sorte que le contenu de cette boite se place au-dessous de la plus haute position de la \vbox chapeau en utilisant à nouveau cette structure :

\vbox to0pt{⟨*réglures verticales*⟩\vss}

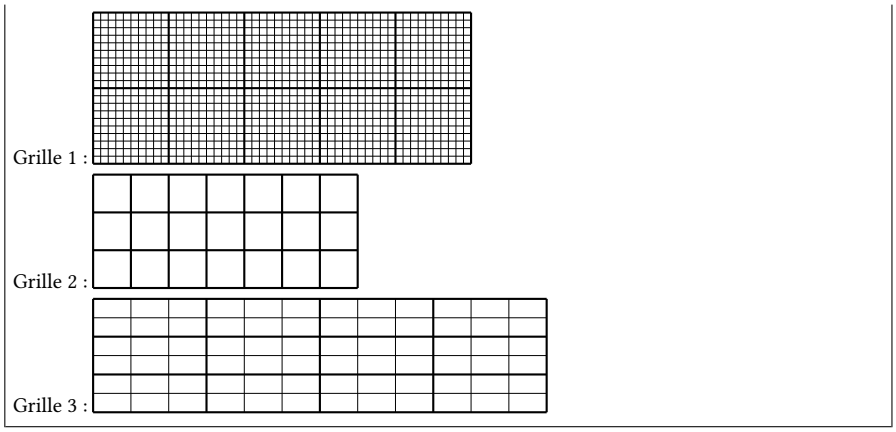
La totalité des réglures verticales sera mise dans une \rlap, c'est-à-dire en débordement à droite de la position courante :

\vbox to0pt{\rlap{⟨*réglures verticales*⟩}\vss}

Il est maintenant clair que, puisque la \rlap a une largeur nulle, cette \vbox aura toutes ses dimensions nulles. Cela signifie que toutes les réglures verticales seront dessinées sans que la position courante ne change.

Code nº III-270

```
\mainrule=0.8pt \subrule=0.2pt
\def\vlap#1{\vbox to0pt{\vss#1\vss}}
\catcode`@11
\def\grid#1#2#3#4{%
  \vbox{% empiler les éléments verticalement
    \offinterlineskip% pas de ressort d'interligne
    % ################### Tracé des réglures verticales ###################
    \vbox to\z@{% dans une \vbox de hauteur nulle
      \edef\total@ht{\the\dimexpr\yunit*#3\relax}% hauteur totale
      \edef\sub@unit{\the\dimexpr\xunit/#2\relax}% espace entre 2 subdivisions
      \rlap{% mettre à droite de la position sans bouger
        \for\ii = 1 to #1 \do 1% pour chaque unité horizontale
          {\clap{\vrule width\dimexpr\mainrule height\total@ht}% réglure principale
          \rlap{% mettre à droite de la position sans bouger
            \for\jj = 2 to #2 \do 1% insérer #2-1 fois
              {\kern\sub@unit % l'espace horizontal
              \clap{\vrule width\subrule height\total@ht}% et la réglure verticale
              }%
          }%
          \kern\xunit % insérer l'espace entre réglures horizontales
          }%
        \clap{\vrule width\mainrule height\total@ht}% dernière réglure principale
      }%
      \vss% compense la hauteur=0pt de la \vbox
    }%
    % ################### Tracé des réglures horizontales ###################
    \edef\total@wd{\the\dimexpr\xunit*#1\relax}% largeur totale de la boite
    \edef\sub@unit{\the\dimexpr\yunit/#4\relax}% espace entre 2 subdivisions
    \for\ii = 1 to #3 \do 1% pour chaque carreau vertical en partant du haut :
      {\vlap{\hrule width\total@wd height\mainrule}% réglure horizontale principale
      % et dessous, les réglures secondaires :
      \vbox to\z@{% dans une \vbox de hauteur nulle,
        \for\jj = 2 to #4 \do 1% insérer #4-1 fois sous la position courante :
          {\kern\sub@unit % l'espace vertical
          \vlap{\hrule width\total@wd height\subrule}% et la réglure secondaire
          }%
        \vss% ressort qui se comprime pour satisfaire la hauteur nulle
      }%
      \kern\yunit% insérer l'espace vertical entre réglures principales
      }%
    \vlap{\hrule width\total@wd height\mainrule}% dernière réglure horizontale
  }%
}
\catcode`@12
Grille 1 : \xunit=1cm \yunit=1cm \grid{5}{10}{2}{10}\smallskip

Grille 2 : \xunit=0.5cm \yunit=0.5cm \grid{7}{1}{3}{1}\smallskip

Grille 3 : \xunit=1.5cm \yunit=0.5cm \grid{4}{3}{3}{2}
```



8.5. Répétition de motifs

8.5.1. \leaders et ses complices

Venons-en maintenant à une primitive intéressante qui a la capacité de répéter une boite autant de fois que nécessaire pour remplir un espace donné, que ce soit en mode horizontal ou vertical. Cette primitive est \leaders et la syntaxe pour le mode horizontal est la suivante :

\leaders⟨*boite ou réglure*⟩\hskip⟨*ressort*⟩

La syntaxe est la même pour le mode vertical sauf que \vskip est utilisé au lieu de \hskip. En réalité, \leaders est juste une assignation pour le ⟨*ressort*⟩ qui spécifie tout simplement avec quel motif il doit être rempli. Car contrairement aux \kern, un ressort a vocation à être rempli avec quelque chose ! Si \leaders est absent comme cela a été le cas jusqu'à présent, ce motif est vide ce qui conduit à des ressorts qui laissent des espaces blancs.

Intéressons-nous au mode horizontal, celui en grande majorité pour lequel est utilisée \leaders. Supposons que la boite de 1,5 cm de long suivante

```
\hbox to1.5cm{\hss A\hss}
```

doive remplir un espace de 10 cm. La primitive \leaders va coller ces boites les unes aux autres de façon à remplir le plus grand espace possible. Ce collage se fera bord à bord en partant de la gauche et laissera, si la longueur à remplir n'est pas un multiple de la longueur du motif, une espace laissée vide à la fin. Dans le code ci-dessous, nous spécifions « \hbox to10cm » pour imposer que la dimension horizontale à remplir soit de 10 cm. Les \vrule placées de part et d'autre de cette \hbox permettent de visualiser le début et la fin de cette boite :

Code n° III-271

```
1 \vrule\hbox to10cm{\leaders\hbox to1.5cm{\hss A\hss}\hfill}\vrule
```

```
|   A       A       A       A       A       A       |
```

Comme on peut facilement s'en assurer par calcul mental, seules 6 copies de la

boites ont été collées les unes aux autres pour remplir un espace de 9 cm, laissant une espace vide de 1 cm à la fin.

Deux variantes de cette primitive existent et permettent de répartir l'éventuelle espace non remplie :

- avec `\cleaders`, elle est également répartie avant la première et après la dernière boite répétée ;
- avec `\xleaders`, elle est également répartie avant la première, après la dernière et entre chaque boite répétée.

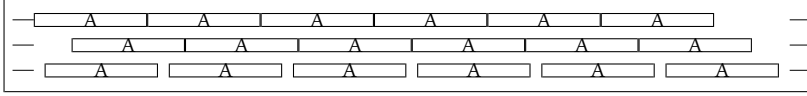
Le plus visuellement compréhensible est de définir cette boite « A », de 1,5 cm de long et encadrée au plus proche par ce code :

```
\frboxsep=-\frboxrule \frbox{\hbox to1.5cm{\hss A\hss}}
```

et regarder les différences de comportement entre les différentes variantes. Pour bien visualiser l'espace de 10 cm à remplir, le caractère actif « ~ » a été programmé pour imprimer une réglure horizontale de `1em` de longueur juste avant et juste après cette espace :

Code n° III-272

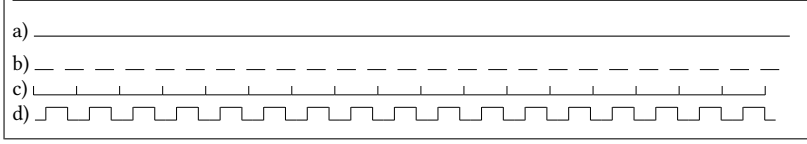
```
\frboxsep=-\frboxrule \def~{\leavevmode\raise.75ex\hbox{\vrule height.2pt width1em}}
~\hbox to10cm{\leaders\hbox{\frbox{\hbox to1.5cm{\hss A\hss}}}\hfill}~

~\hbox to10cm{\cleaders\hbox{\frbox{\hbox to1.5cm{\hss A\hss}}}\hfill}~

~\hbox to10cm{\xleaders\hbox{\frbox{\hbox to1.5cm{\hss A\hss}}}\hfill}~
```



Au lieu d'un encadrement contenant un texte, on peut aussi mettre une réglure ou même plusieurs dans une boite pour faire des dessins rudimentaires :

Code n° III-273

```
a) \leavevmode\hbox to10cm{\leaders\hrule\hfill}\smallskip

b) \hbox to10cm{\leaders\hbox{\vrule height0.2pt width2.5mm \kern1.5mm}\hfill}

c) \vrule width0.2pt height1ex depth0pt % première réglure verticale
     \hbox to10cm{% puis répétition de "_|"
     \leaders\hbox{\vrule width2em height0.2pt \vrule width0.2pt height1ex}\hfill}

d) \hbox to10cm{\leaders\hbox{%
    \vrule height.2pt width.5em% 1/2 palier bas
    \vrule height5pt width0.2pt% montée au palier haut
    \vrule height5pt depth-4.8pt width1em% palier haut
    \vrule height5pt width0.2pt% descente au palier bas
    \vrule height.2pt width.5em% 1/2 palier bas
    }\hfill}
```



La macro `\hrulefill`, qui remplit l'espace disponible avec une réglure horizontale, est programmée de la façon suivante dans plain-TEX :

```
\def\hrulefill{\leaders\hrule\hfill}
```

C'est d'ailleurs un cas où la primitive \hrule peut être appelée en mode horizontal pour produire une réglure horizontale alors que son mode de fonctionnement est le mode vertical.

8.5.2. Retour sur le quadrillage

On entrevoit que la primitive \leaders, de par sa propriété qui est d'effectuer une répétition, peut nous aider à tracer un quadrillage sans faire explicitement appel à une boucle ! Pour y arriver, nous allons tout d'abord nous employer à construire un carreau-unité avec les subdivisions adéquates que \leaders répètera horizontalement et verticalement pour créer le quadrillage.

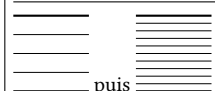
Commençons verticalement. S'il faut n subdivisions dans une espace verticale de \yunit, il faut empiler verticalement dans une boite de hauteur \yunit :

1. une réglure horizontale d'épaisseur \mainrule dont la dimension verticale sera forcée à 0pt ;
2. placer autant de fois qu'il est possible ce matériel vertical suivant (dont la dimension est \yunit/n) :
   a) une espace verticale de dimension \yunit/n ;
   b) une réglure horizontale d'épaisseur \subrule dont la dimension verticale sera forcée à 0pt ;
3. une espace verticale de dimension \yunit/n ;
4. une réglure horizontale d'épaisseur \mainrule dont la dimension verticale sera forcée à 0pt.

Comme les composants du point n° 2 prennent à chaque fois \yunit/n d'espace vertical et que les deux derniers composants prennent aussi cette espace, la répétition de l'élément 2 aura donc lieu $n - 1$ fois.

Code n° III-274

```
\mainrule=0.8pt \subrule=0.2pt
\def\carreau#1#2{% #1=nb subdiv H  #2=nb subdiv V
  \vbox to\yunit{% dans un \vbox de hauteur \yunit
    \offinterlineskip% pas de ressort d'interligne
    \vlap{\hrule height\mainrule width\xunit}% réglure principale du haut
    \leaders% répéter (ici ce sera donc #2-1 fois)
      \vbox to\dimexpr\yunit/#2\relax% une \vbox de hauteur \yunit/#2
        {\vss% ressort qui va s'étirer à \yunit/#2
        \vlap{\hrule height\subrule width\xunit}% réglure de subdiv de dim 0pt
        }\vfill
    \kern\dimexpr\yunit/#2\relax% dernière espace verticale
    \vlap{\hrule height\mainrule width\xunit}% réglure principale du bas
  }%
}
\yunit=1cm
\leavevmode \carreau{}{4} puis \carreau{}{10}
```

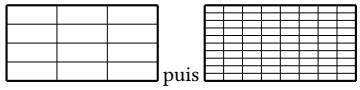


Pour les réglures horizontales, nous allons procéder de même après voir mis tout

le code précédent dans une \rlap de telle sorte que son encombrement horizontal soit nul.

Code n° III-275

```
\mainrule=0.8pt \subrule=0.2pt
\def\carreau#1#2{% #1=nb subdiv H  #2=nb subdiv V
  % ######## réglures horizontales ########
  \rlap{% mettre à droite de la position sans bouger
    \vbox to\yunit{% dans un \vbox de hauteur \yunit
      \offinterlineskip% pas de ressort d'interligne
      \vlap{\hrule height\mainrule width\xunit}% réglure principale du haut
      \leaders% répéter (ici ce sera #2-1 fois)
        \vbox to\dimexpr\yunit/#2\relax% une \vbox de hauteur \yunit/#2
          {\vss% ressort qui va s'étirer à \yunit/#2
          \vlap{\hrule height\subrule width\xunit}% réglure de subdiv de dim 0pt
          }\vfill% ressort de \leaders
      \kern\dimexpr\yunit/#2\relax% derniere espace verticale
      \vlap{\hrule height\mainrule width\xunit}% réglure principale du bas
    }%
  }%
  % ######## réglures verticales ########
  \hbox to\xunit{% dans une \hbox de longueur \xunit
    \clap{\vrule height\yunit width\mainrule}% réglure principale de gauche
    \leaders% répéter (ici ce sera #1-1 fois)
      \hbox to\dimexpr\xunit/#1\relax
        {\hss% ressort qui va s'étirer à \xunit/#1
        \clap{\vrule height\yunit width\subrule}% réglure H de dimension 0
        }\hfill% ressort de \leaders
      \kern\dimexpr\xunit/#1\relax% dernière espace H
      \clap{\vrule height\yunit width\mainrule}% réglure principale de droite
  }%
}
\yunit=1cm \xunit=2cm
\leavevmode \carreau{3}{4} puis \carreau{8}{10}
```

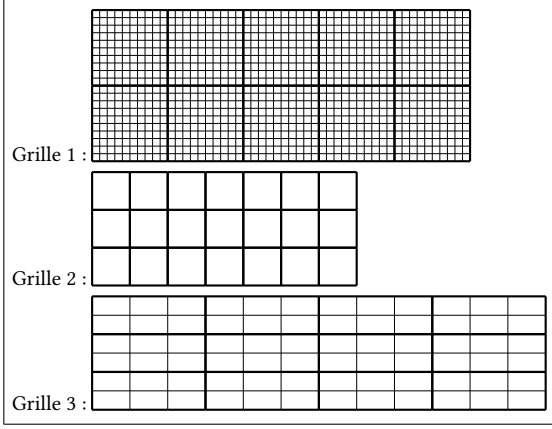


Il suffit maintenant de mettre le tout dans une \hbox. Cette boite a pour exactes dimensions \xunit et \yunit et il ne reste plus qu'à confier à \leaders le soin de la répéter autant de fois horizontalement et verticalement que l'on veut de carreaux dans ces deux directions :

Code n° III-276

```
\mainrule=0.8pt \subrule=0.2pt
\def\grid#1#2#3#4{%
  \vbox to#3\yunit{% dans une boite verticale de hauteur #3*\yunit
    \leaders% répéter verticalement
      \hbox to#1\xunit{% une boite de longueur #1*\xunit
        \leaders% dans laquelle se répète horizontalement
          \hbox{\carreau{#2}{#4}}% le carreau de largeur \xunit
        \hfill}%
    \vfill
  }%
}
Grille 1 : \xunit=1cm \yunit=1cm \grid{5}{10}{2}{10}\smallskip

Grille 2 : \xunit=0.5cm \yunit=0.5cm \grid{7}{1}{3}{1}\smallskip

```

```
16 Grille 3 : \xunit=1.5cm \yunit=0.5cm \grid{4}{3}{3}{2}
```



Bien que le résultat soit strictement identique, la façon de tracer le quadrillage avec `\leaders` comme ci-dessus est très différente de celle employée précédemment avec la boucle `\for`. En effet, la méthode avec `\leaders` est beaucoup plus « laborieuse » puisqu'il s'agit d'une bête répétition d'un élément. Le résultat est trompeur pour l'œil, car la répétition horizontale et verticale de cet élément donne l'illusion que les réglures horizontales ou verticales sont tracées en un coup alors qu'il n'en est rien. De plus, toutes les réglures principales communes à deux carreaux-unité adjacents sont même superposées et sont donc tracées deux fois. Personne ne tracerait un quadrillage à la main de cette façon sur une feuille de papier ! À l'inverse, la méthode avec `\for` bâtissait le quadrillage de façon plus globale puisque les réglures avaient la dimension maximale et étaient effectivement tracées en un coup, comme on le ferait spontanément à la règle et au crayon.

Chapitre 9

Fichiers : lecture et écriture

9.1. Lire un fichier

9.1.1. La primitive \input

TEX peut « lire un fichier », c'est-à-dire qu'il peut lire le contenu du fichier, exactement comme il le fait lorsqu'il lit le code source. Pour cela, on utilise la syntaxe [1] :

\input ⟨*nom d'un fichier*⟩

Lorsque TEX s'attend à lire le nom d'un fichier, le développement maximal se met en place. Le nom du fichier s'étend jusqu'au premier espace rencontré (et cet espace sera absorbé) ou jusqu'à la première primitive non développable : mettre un espace ou un \relax après la fin d'un nom de fichier est donc une bonne habitude à prendre.

La primitive \jobname se développe en le nom du fichier « maitre » en cours de compilation. Le fichier « maitre » est le fichier de plus haut niveau, celui qui n'a été appelé par aucun \input. Le nom créé par \jobname se compose de caractères de catcode 12 et ne comporte pas l'extension du fichier, c'est-à-dire le « . » et les caractères qui le suivent.

Le fichier lu par \input peut lui aussi contenir la primitive \input, et ainsi de suite récursivement dans des limites raisonnables. Une fois que la fin du fichier est

1. Il est peut-être utile de signaler que LATEX, aussi incroyable que cela puisse paraitre, franchit un interdit en redéfinissant la primitive \input pour en faire une macro (non développable). Fort heureusement, la primitive est sauvegardée par \let dans la séquence de contrôle \@@input.

atteinte ou que la primitive \endinput est rencontrée, TEX cesse la lecture du fichier et la reprend dans fichier parent où il s'était interrompu.

83 - Règle

Lorsque TEX lit le code source, on peut faire lire à TEX le contenu d'un fichier. La lecture se fera comme se fait celle du code source et tout se passe donc comme si l'intégralité du contenu du fichier était insérée dans le code source à l'endroit où est rencontré :

\input ⟨*nom de fichier*⟩

Si le ⟨*nom de fichier*⟩ ne spécifie pas d'extension, TEX rajoute « .tex ».

La lecture du fichier s'arrête lorsque la fin du fichier est atteinte ou lorsque la primitive \endinput est atteinte.

9.1.2. \input et ses secrets

Examinons maintenant les autres aspects bien plus TEXniques de la primitive \input. Cette section, assez ardue, peut être sautée lors d'une première lecture.

Tout d'abord, la primitive \input est développable lorsqu'on la 1-développe, tout se passe comme si TEX détournait sa tête de lecture pour aller lire la totalité du contenu du fichier. Il faut signaler que la totalité du fichier est lue selon le régime de catcode en vigueur au moment du 1-développement. De la même façon qu'il le fait lorsqu'il lit du code source, TEX insère le caractère de code \endlinechar à la fin de chaque ligne du fichier, y compris la dernière.

Plusieurs choses assez TEXniques se passent à la fin d'un fichier :

1. avant la fin du fichier, TEX insère le contenu de \everyeof. Cette primitive a un comportement identique à celui d'un registre de tokens et elle est vide par défaut ;
2. juste après le code inséré par \everyeof, TEX exécute une routine interne.

Cette routine interne est en réalité un test auquel procède TEX : il vérifie que l'endroit où il se trouve est compatible avec le statut \outer [2]. Avant de poursuivre, il faut définir ce qu'est ce statut : il concerne normalement les macros qui peuvent être définies comme étant « \outer ».

84 - Règle

Lorsque la primitive \outer précède la définition d'une macro, cette dernière ne pourra pas se situer dans les endroits où TEX lit du code « à grande vitesse », à savoir :

- le texte de remplacement ou le texte de paramètre lors de la définition d'une macro ;
- le code situé dans les branches d'un test ;
- le préambule d'un alignement initié par \halign ou \valign.

Si une telle macro se trouve dans ces endroits interdits, un message d'erreur sera émis et la compilation stoppée :

2. On peut le constater dans le code source « tex.web » du programme tex à la section nº 362, le test « check_outer_validity » est effectué.

Code n° III-277

```
\outer\def\foo{Bonjour}
\def\bar{\foo}
```

```
! Forbidden control sequence found while scanning definition of \bar.
```

Une action peu connue de \noexpand⟨*token*⟩, lorsque \noexpand est développé, est de désactiver pour le ⟨*token*⟩ le test qui vérifie si ce ⟨*token*⟩ est \outer [3]. On peut donc l'utiliser pour enfreindre la règle précédente et mettre une macro \outer dans le texte de remplacement d'une autre macro :

Code n° III-278

```
\outer\def\foo{Bonjour}
\expandafter\def\expandafter\bar\expandafter{\noexpand\foo}
\meaning\bar

\edef\baz{\noexpand\foo}
\meaning\baz
```

```
macro:->\foo
macro:->\foo
```

Revenons au test à la fin du fichier : comme il vérifie que l'endroit où il se trouve est compatible avec \outer, il communique à la fin du fichier le statut \outer. Pour désactiver le test, nous allons appliquer la même méthode qu'avec une macro \outer, c'est-à-dire le faire précéder de la primitive \noexpand.

85 - Règle

La fin d'un fichier a le statut \outer.

Pour permettre que la fin d'un fichier se situe dans un endroit normalement interdit à ce statut, il faut placer la primitive \noexpand juste avant la fin du fichier, soit en l'écrivant explicitement, soit via \everyeof avec l'assignation

\everyeof{\noexpand}

et faire en sorte de développer ce \noexpand.

Développer le \noexpand rend très contraignante et quasi impossible la définition d'une macro dont le texte de remplacement serait le contenu d'un fichier. En effet, pour forcer ce développement *avant* que la définition de la macro ne se fasse, il faudrait la définir avec \edef selon ce schéma :

```
\begingroup
\everyeof{\noexpand}
\global\edef\⟨macro⟩{\input⟨nom de fichier⟩ }
\endgroup
```

Cette définition est très peu satisfaisante, car la primitive \edef implique que le contenu du fichier est développé au maximum ce qui, sauf cas très particulier, n'est pas ce qui est recherché. De plus, elle rend la définition globale puisqu'un groupe semi-simple a été ouvert pour limiter la portée de la modification de \everyeof.

3. Cela est expliqué à la section n° 369 de tex.web par Knuth lui-même : « Comme des macros \outer peuvent se présenter ici, on doit aussi réinitialiser [à "normal"] le scanner_statut temporairement. »

Heureusement, nous pouvons bien mieux faire. L'idée est d'insérer un délimiteur (ici `\eof@nil`) en fin de fichier via `\everyeof`. Ce délimiteur sera capturé par une macro auxiliaire à argument délimité qui sera dès lors capable de savoir où se trouve la fin du fichier et affecter à une macro ce qui se trouve dans ce fichier avant ce délimiteur. Toute cette manœuvre suppose que le fichier ne contient pas `\eof@nil`.

Supposons par exemple que le fichier « `test.txt` » contient :

```
Programmer en \TeX{} est   facile

et utile
```

Voici comment programmer une macro

`\filedef\`⟨*macro*⟩`{`⟨*nom du fichier*⟩`}`

qui définit une `\`⟨*macro*⟩ dont le texte de remplacement est le contenu du fichier :

Code n° III-279

```
1  \catcode`\@11
2  \long\def\filedef#1#2{%
3    \begingroup
4      \let\input\@@input% <- utilisateurs de latex uniquement
5      \everyeof{\eof@nil\noexpand}% insère "\eof@nil\noexpand" à la fin du fichier
6      \expandafter\filedef@i\expandafter#1% développe \input #2
7        \expandafter\relax\input #2
8  }
9  \long\def\filedef@i#1#2\eof@nil{%
10   \endgroup
11   \expandafter\def\expandafter#1\expandafter{\gobone#2}% mange le \relax
12 }
13 \catcode`\@12
14 \filedef\foo{test.txt}
15 \meaning\foo
```

```
macro:->Programmer en \TeX {} est facile \par et utile
```

La ligne « `\let\input\@@input` » redonne à `\input` son statut de primitive et annule la redéfinition effectuée par LaTeX [4]. Il est bien évident que les utilisateurs d'autres formats ne doivent pas écrire cette ligne.

Ensuite, à la ligne n° 7, un `\relax` a été inséré avant le développement de `\input`. Ceci nous assure que l'argument délimité #2 de `\filedef@i` ne serait pas dépouillé de ses accolades si jamais le contenu du fichier était constitué d'un texte entre accolades. Ce `\relax` est mangé à la ligne n° 11 par le `\gobone` avant que la définition ne soit faite.

L'important est de comprendre qu'une fois le `\input` 1-développé par le pont d'`\expandafter` et en tenant compte du contenu de `\everyeof`, les lignes n°s 6 et 7 deviennent :

`\filedef@i#1\relax`⟨*contenu du fichier lu par \input*⟩`\eof@nil\noexpand`⟨*EOF*⟩

La macro `\filedef@i`, qui ne lit que ce qui se situe jusqu'au `\eof@nil`, ne verra jamais la fin du fichier ⟨*EOF*⟩. Le but recherché est atteint : le ⟨*EOF*⟩ est précédé d'un `\noexpand`, et ce `\noexpand`, en étant exécuté après que `\filedef@i` ait fait son travail, rendra ⟨*EOF*⟩ compatible avec le statut `\outer`.

4. Ce livre a été compilé en utilisant le format LaTeX.

9.2. Lire dans un fichier

« Lire un fichier » n'a pas la même signification que « lire *dans* un fichier » qui sous-entend que l'on peut lire, contrôler ce qu'on lit et donc reprendre la main. Pour fonctionner selon ce mode, TEX dispose de 16 canaux d'entrée, numérotés de 0 à 15, chacun correspondant à un flux de lecture. Comme pour les boites, les compteurs, les dimensions et les registres de tokens, plain-TEX fournit la macro \newread\⟨*macro*⟩ qui assigne [5] à la \⟨*macro*⟩ un entier correspondant à un canal de lecture disponible.

86 - Règle

TEX dispose de 15 canaux de lecture portant les numéros de 0 à 15.

On demande l'allocation d'un canal de lecture par

\newread\⟨*macro*⟩

et ce faisant, la \⟨*macro*⟩ devient équivalente à un ⟨*nombre*⟩, qui est un numéro de canal libre. Un ⟨*canal*⟩ est donc un ⟨*nombre*⟩ explicitement écrit ou une macro définie avec \newread.

Avant de lire dans un fichier, on doit d'abord lier un ⟨*canal*⟩ à un fichier avec :

\openin⟨*canal*⟩= ⟨*nom du fichier*⟩

Si aucune extension n'est précisée au ⟨*nom du fichier*⟩, TEX rajoute « .tex ». Le signe = et l'espace qui le suit sont optionnels. Lorsque TEX lit le ⟨*nom du fichier*⟩, il entre dans une phase de développement maximal et si ce nom est suivi d'un espace, cet espace est absorbé.

Tout canal ouvert doit être fermé et donc, après avoir fait les opérations de lecture, il convient d'exécuter \closein⟨*canal*⟩ pour désolidariser le ⟨*canal*⟩ du fichier qui lui avait été attaché avec \openin.

Le test \ifeof⟨*canal*⟩ teste si la lecture du fichier lié au ⟨*canal*⟩ est arrivée à la fin du fichier. Employé juste après avoir lié un flux à un fichier via \openin, le test \ifeof⟨*canal*⟩ sera vrai uniquement si le fichier spécifié n'existe pas. Voici donc comment bâtir une macro \iffileexists⋆ :

Code n° III-280

```
\newread\rtest % canal de lecture employé dans tout ce chapitre
\def\iffileexists#1#2{% #1=canal de lecture  #2=nom du fichier
  \openin#1=#2
  \ifeof#1% le fichier n'existe pas
    \closein#1
    \expandafter\secondoftwo% renvoyer faux
  \else
    \closein#1
    \expandafter\firstoftwo% sinon renvoyer vrai
  \fi
}
a) \iffileexists\rtest{test.txt}{vrai}{faux}\qquad
```

5. Cette macro fonctionne comme son homologue \newbox. Comme il n'existe pas de primitive \readdef, la macro est définie par la primitive \chardef.

```
13 b) \iffileexists\rtest{foobar.txt}{vrai}{faux}
```

a) vrai b) faux

Dans tout cet exemple, nous avons demandé l'allocation d'un ⟨*canal*⟩ avec « \newread\rtest » et désormais, nous utiliserons le canal de lecture n° \rtest dans tous les exemples.

Un fichier peut-être lu par plusieurs canaux simultanément. Cela signifie que si le besoin s'en fait sentir, on peut positionner des têtes de lecture dans un fichier à plusieurs endroits, chacune pouvant lire le fichier indépendamment des autres.

87 - Règle

Lorsqu'un ⟨*canal*⟩ de lecture est ouvert et lié à un fichier, TEX peut lire des *lignes* dans ce fichier les unes après les autres, et assigner la ligne lue à une macro. On utilise la syntaxe

\read\⟨*canal*⟩ to \⟨*macro*⟩

où le mot-clé « to » est obligatoire et éventuellement suivi d'espaces optionnels.

Ceci a pour effet de placer dans le texte de remplacement de la \⟨*macro*⟩ tout ce qu'il y a dans la ligne en cours :

- les accolades doivent être équilibrées dans la ligne en cours. Si elles ne le sont pas, TEX lit autant de lignes que nécessaire pour qu'elles le soient. Si la fin du fichier est atteinte sans que l'équilibrage des accolades ne soit réalisé, une erreur de compilation surviendra ;
- une fois l'équilibrage des accolades atteint, la primitive \read lit tout ce qui s'étend jusqu'à la « marque de fin de ligne » (voir page 29) ;
- les tokens dans le texte de remplacement de la \⟨*macro*⟩ tiennent compte du régime de catcode en vigueur lorsque \read est exécutée ; \read effectue donc une « tokénisation », c'est-à-dire la transformation d'octets bruts en tokens ;
- le caractère de code \endlinechar est inséré à la fin du texte de remplacement de la\⟨*macro*⟩ ;
- l'assignation est globale si \read est précédé de \global.

La primitive de ε-TEX\readline, dont la syntaxe est

\readline\⟨*canal*⟩ to \⟨*macro*⟩

agit comme \read sauf qu'elle insère dans le texte de remplacement de la \⟨*macro*⟩ tous les tokens lus dans la ligne (y compris le caractère de code \endlinechar). L'espace aura le catcode 10 et tous les autres caractères le catcode 12.

Pour fixer les idées, si le canal n° \rtest est lié à un fichier et que la ligne suivante à lire dans ce fichier est :

```
Programmer en \TeX{} est facile
```

Si l'on écrit « \read\rtest to \foo », tout se passe comme si on avait fait l'assignation d'une macro sans argument :

```
\def\foo{Programmer en \TeX{} est facile }
```

Il est important de noter que l'espace en fin de texte provient du caractère de code \endlinechar : celui-ci étant par défaut « ^^M » de catcode 5, il est lu comme un espace comme le stipule la règle page 29. Pour mettre ce comportement en évidence dans l'exemple ci-dessous, le fichier `filetest.txt` contiet :

```
Programmer en \TeX{} est   facile

et utile
```

Notons qu'il y a 3 espaces entre les mots « est » et « facile » et regardons ce que donne le code suivant :

Code n° III-281

```
\openin\rtest =filetest.txt
\read\rtest to \foo% lit la première ligne
1) Signification : \meaning\foo.\par% signification de ce qui a été lu
1) Exécution : \foo\par
\read\rtest to \foo% lit la deuxième ligne
2) Signification : \meaning\foo.\par
2) Exécution : \foo\par
\read\rtest to \foo% lit la dernière ligne
3) Signification : \meaning\foo.\par
3) Exécution : \foo
\closein\rtest
```

```
1) Signification : macro:->Programmer en \TeX {} est facile .
1) Exécution : Programmer en TEX est facile
2) Signification : macro:->\par .
2) Exécution :
3) Signification : macro:->et utile .
3) Exécution : et utile
```

Les points après chaque \meaning\foo sont ici destinés à mettre en évidence l'espace en fin de texte de remplacement.

Voici le même exemple, cette fois-ci avec la primitive \readline :

Code n° III-282

```
\openin\rtest =filetest.txt
\readline\rtest to \foo% lit la première ligne
1) Signification : \meaning\foo.\par% signification de ce qui a été lu
1) Exécution : \foo\par
\readline\rtest to \foo% lit la deuxième ligne
2) Signification : \meaning\foo.\par
2) Exécution : \foo\par
\readline\rtest to \foo% lit la dernière ligne
3) Signification : \meaning\foo.\par
3) Exécution : \foo
\closein\rtest
```

```
1) Signification : macro:->Programmer en \TeX{} est   facile,.
1) Exécution : Programmer en \TeX{} est   facile,
2) Signification : macro:->,.
2) Exécution : ,
3) Signification : macro:->et utile,.
3) Exécution : et utile,
```

Une première question se pose naturellement : d'où viennent les virgules que l'on observe à la fin du texte de remplacement de \foo ? Elles sont la preuve que le caractère de code \endlinechar, placé en fin de ligne, a été détokénizé par

\readline pour devenir un inoffensif caractère de catcode 12. On voit une virgule car \endlinechar a pour valeur 13 et que le caractère ayant ce code dans la fonte utilisée est la virgule :

Code n° III-283

```
Valeur de \string\endlinechar = \number\endlinechar\par
Caractère correspondant : << \char\endlinechar{} >>
```

Valeur de \endlinechar= 13
Caractère correspondant : « , »

Le second point qui peut surprendre est qu'au premier cas, le texte de remplacement de \foo contient *tous les espaces* (ils sont 3) entre les mots « est » et « facile », alors qu'un seul était lu par \read. La subtile différence entre \read et \readline est que cette dernière ignore les catcodes en cours et insère dans le texte de remplacement de la macro *tous* les caractères lus dans la ligne ; elle insère donc *tous* les espaces, en leur donnant un catcode de 10.

■ Exercice 93

Proposer une méthode pour que la macro \read ne lise que ce qui se trouve dans la ligne courante, *sans* y ajouter le caractère de code \endlinechar.

Appliquer cette méthode pour écrire une macro \xread* qui agit de la même façon que \read avec la même syntaxe :

\xread⟨*canal*⟩ to \⟨*macro*⟩

mais qui place dans le texte de remplacement de la \⟨*macro*⟩ le contenu de la ligne sans le caractère de code \endlinechar.

□ Solution

Il suffit d'opérer localement et définir le \endlinechar à −1 et utiliser un \global pour que la \⟨*macro*⟩ survive à la fermeture du groupe :

Code n° III-284

```
\openin\rtest =filetest.txt
Ligne 1 : {\endlinechar=-1 \global\read\rtest to \foo}% lit la première ligne
\meaning\foo.\par% donne ce qui a été lu
Ligne 2 : {\endlinechar=-1 \global\read\rtest to \foo}% lit la deuxième ligne
\meaning\foo.\par
Ligne 3 : {\endlinechar=-1 \global\read\rtest to \foo}% lit la dernière ligne
\meaning\foo.
\closein\rtest
```

Ligne 1 : macro:->Programmer en \TeX {} est facile.
Ligne 2 : macro:->.
Ligne 3 : macro:->et utile.

Contrairement à l'exemple précédent, comme le caractère de code \endlinechar n'est pas inséré à la fin des lignes, la seconde ligne est bien lue comme étant vide.

Pour la macro \xread, nous n'allons pas utiliser de \global car il est toujours plus sûr de se passer, lorsque c'est possible, d'assignations globales qui provoquent des effets de bord difficiles à maitriser. Nous allons tout d'abord sauvegarder dans la macro \restoreendlinechar le code qui permet de restaurer \endlinechar. Cette macro aura comme texte de remplacement

\endlinechar=⟨*valeur actuelle de* *endlinechar*⟩

Toute la difficulté vient ensuite des espaces optionnels qui se trouvent après le mot-clé « to ». En effet, si nous déclarons la macro \xread ainsi

\def\xread#1to#2{⟨*code à définir*⟩}

rien ne permet d'affirmer que l'argument #2 sera la \⟨*macro*⟩ : si des espaces optionnels sont écrits après « to », alors #2 sera un espace. Il va donc falloir tester l'argument #2 et ne le prendre en compte que lorsqu'il sera une séquence de contrôle, ce qui suppose la mise en place d'une macro récursive. Pour mener à bien le test, nous mettrons en œuvre la macro \ifcs programmée à la page 207.

Code n° III-285

```
\def\xread#1to#2{%
  \ifcs{#2}% si #2 est une séquence de contrôle
    {\edef\restoreendlinechar{\endlinechar=\the\endlinechar}%
    \endlinechar=-1 % supprime le caractère mis en fin de ligne
    \read#1to#2% lit la ligne et l'assigne à la macro #2
    \restoreendlinechar\relax% restaure le \endlinechar
    }% si #2 n'est pas une séquence de contrôle,
    {\xread#1to}% ignorer #2, et recommencer
}

\openin\rtest =filetest.txt
Ligne 1 : \xread\rtest to \foo% lit la première ligne
\meaning\foo.\par% donne ce qui a été lu
Ligne 2 : \xread\rtest to \foo% lit la deuxième ligne
\meaning\foo.\par
Ligne 3 : \xread\rtest to \foo% lit la dernière ligne
\meaning\foo.
\closein\rtest
```

Ligne 1 : macro:->Programmer en \TeX {} est facile.
Ligne 2 : macro:->.
Ligne 3 : macro:->et utile.

Une première remarque consiste à dire que la syntaxe de \xread n'est *pas exactement identique* à celle de \read. En effet, la macro récursive \xread ignore les arguments #2 tant qu'ils ne sont pas une séquence de contrôle. L'appel farfelu suivant est donc tout à fait valide :

\xread\rtest toW a aZcd 1 2345\foo

Mais surtout, il existe une méthode bien plus élégante pour programmer \xread, non récursive, ayant exactement la même syntaxe que \read, mais qui fait intervenir la primitive \afterassignment que nous verrons plus tard (voir page 354). Le code est donné à titre indicatif :

Code n° III-286

```
\def\xread{% doit être suivie de "<nombre> to \<macro>"
  \edef\restoreendlinechar{\endlinechar=\the\endlinechar}%
  \endlinechar=-1 % neutralise \endlinechar
  \afterassignment\restoreendlinechar% après l'assignation, restaurer \endlinechar
  \read% attend <nombre> to \<macro> pour effectuer l'assignation
}
\catcode`\@12

\openin\rtest =filetest.txt
Ligne 1 : \xread\rtest to \foo% lit la première ligne
\meaning\foo.\par% donne ce qui a été lu
Ligne 2 : \xread\rtest to \foo% lit la deuxième ligne
\meaning\foo.\par
```

```
Ligne 3 : \xread\rtest to \foo% lit la dernière ligne
\meaning\foo.
\closein\rtest
```

Ligne 1 : macro:->Programmer en \TeX {} est facile.
Ligne 2 : macro:->.
Ligne 3 : macro:->et utile.

■

9.2.1. Faire une recherche dans un fichier

Envisageons maintenant un cas pratique et imaginons qu'une liste de fournitures soit contenue dans un fichier « `founitures.txt` » dont le contenu est le suivant :

```
Ref,Item,Prix,Fournisseur
1201a,article 1,25.90,\TeX{} marchand
4562u,article 2,120,fournisseur 1
1721d,article 3,57.10,fournisseur 2
⟨etc⟩
```

On remarque que la première ligne du fichier n'est pas à proprement parler une ligne de données, elle indique le nom des champs qui composent les lignes suivantes.

Supposons que ce fichier soit dans le même répertoire [6] que le fichier maitre et que nous souhaitions chercher dans ce fichier une ligne par sa référence, c'est-à-dire par ce qui est contenu dans le premier champ. Le but est de mettre au point une macro `\searchitem*` appelée de cette façon

```
\searchitem⟨canal⟩{fournitures.txt}{1721d}\foo
```

de telle sorte que la ligne où se trouve le premier champ « `1721d` » soit trouvée. La macro purement développable `\foo{⟨champ⟩}` sera créée où son argument précisera de quel ⟨*champ*⟩ on souhaite obtenir la valeur. Par exemple,

`\foo{ref}`	affiche	1721d
`\foo{item}`	affiche	article 3
`\foo{prix}`	affiche	57.10
`\foo{fournisseur}`	affiche	fournisseur 2

La macro `\foo` est ici prise pour l'exemple, mais naturellement, le choix du nom de cette macro est à la convenance de l'utilisateur.

Fixons-nous trois contraintes supplémentaires :

- si le fichier n'existe pas ou si la référence cherchée n'existe pas, la macro `\foo` définie doit être `\let`-égale à `\gobone`. De cette façon, elle absorbera son argument et ne produira aucun affichage ;
- si un champ n'existe pas, rien ne doit être affiché. Par exemple, `\foo{truc}` ne doit produire aucun affichage ;
- lors de la recherche, toute ligne vide doit être ignorée.

La première chose à faire va être d'initialiser la macro `\foo` (argument #4) sera initialisée à `\gobone`. Ensuite, la première ligne du fichier sera lue, ses champs seront

6. Ou qu'un lien symbolique vers ce fichier soit présent dans le répertoire de compilation.

parcourus avec \doforeach afin de stocker leur nom écrit en lettres minuscules dans une macro \fieldname⟨*i*⟩ ou ⟨*i*⟩ est le numéro du champ.

Par la suite, nous pourrons commencer à lire les lignes de données : chaque ligne sera envoyée à une macro auxiliaire \test@field chargée d'examiner leurs champs. Le premier champ de la ligne en cours sera comparé à celui qui est recherché et si le test est positif, il incombera à la macro \test@filed de procéder aux assignations nécessaires des macros

- \foo.⟨*nom du champ 1*⟩
- \foo.⟨*nom du champ 2*⟩\foo.⟨*nom du champ 2*⟩
- etc

chacune \let-égale à \fieldname⟨*i*⟩.

Enfin, la macro purement développable \foo, prenant comme argument un ⟨*champ*⟩, sera créée. Elle testera avec \ifcsname si la macro \foo.⟨*champ*⟩ existe, auquel cas elle formera cette macro via \csname.

Code n° III-287

```
\def\macroname{% se développe en le nom de la macro qui suit sans
               % le caractère d'échappement
  \ifnum\escapechar>-1 % si le caractère d'échappement est positif
    \ifnum\escapechar<256 % et inférieur à 256, développer les 2 "\fi"
      \expandafter\expandafter\expandafter\expandafter% et le "\string", puis
      \expandafter\expandafter\expandafter\gobone% manger le "\" avec \gobone
    \fi
  \fi
  \string% doit être suivi d'une macro
}
\catcode`\@11
\newcount\field@cnt
\def\searchitem#1#2#3#4{% #1= canal  #2=nom fichier #3=référence  #4=macro à définir
  \let#4\gobone% pour l'instant, #4=\gobone
  \openin#1=#2\relax
  \unless\ifeof#1% si le fichier existe
    \lowercase{\def\sought@firstfield{#3}}% stocke le 1er champ à chercher
    \edef\macro@name{\macroname#4}% nom de la macro sans "\"
    \xread#1 to \current@line% lire la première ligne
    \field@cnt=0 % initialiser le compteur de champs
    % ################ sauvegarde du nom des champs ################
    \expsecond{\doforeach\current@field\in}\current@line% pour chaque champ
      {\advance\field@cnt1 % incrémenter le compteur de champs
      \lowercase\expandafter{% e texte de remplacement de \current@field en minuscule
        \expandafter\def\expandafter\current@field\expandafter{\current@field}%
      }%
      % sauvegarder chaque champ de la 1re ligne (qui sont les intitulés) dans une macro
      \letname{fieldname\number\field@cnt}=\current@field
      }%
    \edef\field@num{\number\field@cnt}% nombre de champs
    % ################ lecture des lignes de données ################
    \loop% tant que...
      \unless\ifeof#1\relax% ...la fin du fichier n'est pas atteinte
        \xread#1 to \current@line% lire une ligne
        \unless\ifx\current@line\empty% si elle n'est pas vide
          % examniner les champs qu'elle contient (aller à \test@field)
          \expsecond{\expandafter\test@field\current@line\@nil}\macro@name%
        \fi
    \repeat
  \fi
  \closein#1\relax
```

```
}

\def\test@field#1,#2\@nil#3{% #1=champ no 1 #2=autres champs #3=nom de la macro sans "\"
  \def\current@firstfield{#1}% stocke le premier champ de la ligne en cours
  \ifx\current@firstfield\sought@firstfield% s'il est égal à celui cherché
    \defname{#3.\csname fieldname1\endcsname}{#1}% définir la macros \<#3."champ 1">
    \field@cnt=1 % initialiser le compteur de champ
    \doforeach\current@field\in{#2}% puis, pour i>2, définir les macros \<#3."champ i">
      {\advance\field@cnt1 % incrémenter le compteur de champ
      \letname{#3.\csname fieldname\number\field@cnt\endcsname}=\current@field
      }%
    \defname{#3}##1{% et définir la macro \<#3>
      \ifcsname#3.##1\endcsname% si la macro \<#3."argument"> existe déjà
        \csname#3.##1\expandafter\endcsname% l'exécuter après avoir mangé le \fi
      \fi
    }%
  \fi
}
\catcode`\@12

\searchitem\rtest{fournitures.txt}{4562u}\monarticle
réf = \monarticle{ref},
dénomination = \monarticle{item},
prix = \monarticle{prix},
fournisseur = \monarticle{fournisseur},
champ non existant = \monarticle{foobar}.

\searchitem\rtest{fournitures.txt}{truc}\essai% référence "truc" n'existe pas
réf = \essai{ref},
dénomination = \essai{item},
prix = \essai{prix},
fournisseur = \essai{fournisseur}.
```

```
réf = 4562u, dénomination = article 2, prix = 120, fournisseur = fournisseur 1, champ non existant = .
réf = , dénomination = , prix = , fournisseur = .
```

La macro \macroname, donnée tout au début, doit être suivie d'une \⟨*macro*⟩ et se développe en le nom de cette \⟨*macro*⟩ sans l'éventuel caractère d'échappement.

■ Exercice 94

Programmer une amélioration de la macro \searchitem pour que la recherche puisse se faire sur n'importe quel champ. Les champs seront spécifiés par leur numéro et non plus par leur nom. La syntaxe sera :

\searchitem⟨*canal*⟩{⟨*nom fichier*⟩}{⟨*numéro champ*⟩=⟨*valeur*⟩}\⟨*macro*⟩

La modification de la syntaxe concerne le 3e argument et l'introduction du ⟨*numéro champ*⟩ que l'on souhaite chercher. Si ce ⟨*numéro champ*⟩ et le signe = sont absent, le ⟨*numéro champ*⟩ sera pris égal à 1.

On supposera que le fichier « basecourse.txt » recueille les données des participants à une course à pied et contient :

```
Nom,Prénom,Classement,Naissance,Licencié,Temps
Demay,Pierre,237,27/8/1986,oui,2h37
Leblanc,Nicolas,187,4/9/1978,non,2h05
Valet,Bruno,78,25/11/1989,oui,1h47
Hachel,André,283,2/3/1972,non,2h42
Jalabert,Jean,165,19/1/1982,Oui,2h01
```

Si l'on écrit

```
\searchitem\rtest{basecourse.txt}{3=283}\coureur
```

alors, `\searchitem` doit chercher la ligne où le 3e champ vaut 283. Par la suite, la macro `\coureur` doit admettre un argument *numérique* ⟨*i*⟩ et `\coureur{`⟨*i*⟩`}` doit renvoyer la valeur du champ no ⟨*i*⟩. Par exemple, `\coureur{2}` doit renvoyer « André ».

□ **Solution**

Les modifications à apporter ne sont pas très nombreuses.

Tout d'abord, il faut lire la première ligne et l'ignorer : elle n'est d'aucune utilité. Ensuite, il faut traiter correctement l'argument `#3` selon qu'il contient le signe = ou pas. À l'issue de ce traitement, on a stocké le numéro du champ et sa valeur dans 2 macros pour les utiliser plus tard.

Dans le code précédent, la boucle `\loop` était parcourue jusqu'à ce que `\ifeof` soit vrai. Une petite amélioration consiste à permettre une sortie prématurée si la ligne est trouvée. Pour cela, un booléen créé avec `\newif` sera initialisé à vrai et mis à faux s'il faut sortir de la boucle `\loop` :

- soit lorsque le test `\ifeof` est vrai ;
- soit parce que la ligne a été trouvée.

Code no III-288

```
\def\macroname{% se développe en le nom de la macro qui suit sans
               % le caractère d'échappement
  \ifnum\escapechar>-1 % si le caractère d'échappement est positif
    \ifnum\escapechar<256 % et inférieur à 256, développer les 2 "\fi"
      \expandafter\expandafter\expandafter\expandafter% et le "\string", puis
      \expandafter\expandafter\expandafter\gobone% manger le "\" avec \gobone
    \fi
  \fi
  \string% doit être suivi d'une macro
}
\catcode`\@11
\newcount\field@cnt
\newif\ifsearch@repeat
\def\assign@arg#1=#2\@nil{%
  \def\sought@fieldnumber{#1}% no du champ à chercher
  \def\sought@fielvalue{#2}% et sa valeur
}
\def\searchitem#1#2#3#4{% #1= canal  #2=nom fichier #3=champ cherché  #4=macro à définir
  \let#4\gobone% pour l'instant, #4=\gobone
  \openin#1=#2\relax%
  \unless\ifeof#1% si le fichier existe
    \edef\macro@name{\macroname#4}% nom de la macro sans "\"
    \xread#1 to \current@line% lire et ignorer la première ligne
    \ifin{#3}{=}% si #3 contient =
      {\assign@arg#3\@nil}% trouver le no de champ et sa valeur
      {\def\sought@fieldnumber{1}% sinon, no du champ = 1
      \def\sought@fielvalue{#3}% et sa valeur = #3
      }%
    % ################ lecture des lignes de données ################
    \search@repeattrue% poursuite de la boucle loop : vraie
    \loop% tant que...
      \ifeof#1\relax% ...la fin du fichier n'est pas atteinte
        \search@repeatfalse% sortir de la boucle loop
      \else
        \xread#1 to \current@line% lire une ligne
        \unless\ifx\current@line\empty% si elle n'est pas vide
          % examniner les champs qu'elle contient (aller à \test@field)
```

```
          \expsecond{\expandafter\test@field\current@line\@nil}\macro@name%
        \fi
      \fi
      \ifsearch@repeat% ne poursuivre que si le booléen en vrai
    \repeat
  \fi
  \closein#1\relax
}

\def\test@field#1\@nil#2{% #1=champs #2=nom de la macro sans "\"
  \field@cnt=0 % initialiser le compteur de champ
  \doforeach\current@field\in{#1}% parcourir les champs de la ligne en cours
    {\advance\field@cnt1 % incrémenter le compteur de champ
    \ifnum\field@cnt=\sought@fieldnumber\relax% si c'est le bon numéro de champ
      \ifx\current@field\sought@fielvalue% et si le champ correspond à celui cherché
        \search@repeatfalse% sortir de la boucle loop
        \doforeachexit% sortir de la boucle \doforeach en cours
      \fi
    \fi
    }%
  \unless\ifsearch@repeat% si la ligne a été trouvée
    \field@cnt=0 % initialiser le compteur de champ
    \doforeach\current@field\in{#1}% parcourir à nouveau les champs de la ligne
        {\advance\field@cnt1 % incrémenter le compteur de champ
        \letname{#2.\number\field@cnt}=\current@field% faire l'assignation
        }%
    \defname{#2}##1{% et définir la macro \<#2>
      \ifcsname#2.##1\endcsname% si la macro \<#2."argument"> existe déjà
        \csname#2.##1\expandafter\endcsname% l'exécuter après avoir mangé le \fi
      \fi
    }%
  \fi
}
\catcode`\@12
a) \searchitem\rtest{basecourse.txt}{3=283}\foo
   "\foo1", "\foo2", "\foo3", "\foo4", "\foo5", "\foo6", "\foo7"

b) \searchitem\rtest{basecourse.txt}{Valet}\bar
   "\bar1", "\bar2", "\bar3", "\bar4", "\bar5", "\bar6", "\bar{abcd}"
```

a) "Hachel", "André", "283", "2/3/1972", "non", "2h42", ""
b) "Valet", "Bruno", "78", "25/11/1989", "oui", "1h47", ""

■

9.2.2. Afficher le contenu exact d'un fichier

Nous allons maintenant programmer une macro `\showfilecontent*` de syntaxe

`\showfilecontent`⟨*canal*⟩{⟨*nom du fichier*⟩}

qui affiche le contenu du fichier passé en argument, en police à chasse fixe. Le but est d'imiter pour les fichiers la primitive `\meaning` qui donne le texte de remplacement d'une macro.

Il va falloir agir sur les codes de catégorie avec les macros `\do` et `\dospecials` de plain-TEX pour rendre tous les tokens inoffensifs, sauf l'espace qui sera rendu actif avec `\obeyspaces`.

Le fichier « `readtext.txt` » utilisé dans l'exemple contient 3 lignes :

```
Programmer en \TeX{} est   facile

et utile
```

Code n° III-289

```
\def\showfilecontent#1#2{% #1=canal de lecture #2=nom de fichier
  \begingroup
    \tt% sélectionner la fonte à chasse fixe
    \openin#1=#2\relax
    \ifeof#1% si la fin du fichier est déjà atteinte, il n'existe pas et
      Le fichier n'existe pas% afficher le message
    \else% le fichier existe
      \def\do##1{\catcode`##1=12 }%
      \dospecials% neutraliser tous les tokens spéciaux
      \obeyspaces% rendre l'espace actif
      \loop
        \xread#1 to \currline% lire une ligne
        \unless\ifeof#1% si la fin du fichier n'est pas atteinte
        \leavevmode\par% aller à la ligne
        \currline% afficher la ligne lue
      \repeat% recommencer
    \fi
    \closein#1\relax
  \endgroup
}

Contenu du fichier : "\showfilecontent\rtest{readtest.txt}", affiché tel quel
```

Contenu du fichier : "
`Programmer en \TeX{} est   facile`

`et utile`", affiché tel quel

Il est intéressant de noter qu'un `\obeyspaces` est écrit à la ligne n° 10 car

```
\letactive\ =\space
```

ne fonctionnerait pas puisque que `\letactive`, tout comme `\defactive`, *nécessite* que le caractère « ~ » soit actif. Or, ce n'est plus le cas ici puisque `\dospecials` l'a rendu de catcode 12.

En revanche, placer `\letactive\ =\space` avant le `\dospecials` aurait fonctionné.

■ Exercice 95

Expliquer pourquoi le guillemet initial ne se situe pas juste avant le mot « `Début` » et proposer une solution pour que cela soit le cas.

□ Solution

Chaque ligne affichée, contenue dans la macro `\currline`, est précédée de `\leavevmode \par` (ligne n° 14), y compris pour la première ligne lue. De par la structure de la macro `\showfilecontent`, un `\par` est donc toujours inséré entre le guillemet initial et le premier caractère du fichier.

Une solution originale est de garder « `\leavevmode\par` » sauf que l'on va mettre une macro « magique » `\magicpar*` en lieu et place de `\par`. Cette macro aux vertus spéciales doit n'avoir aucun effet pour l'affichage la première fois et exécuter un `\par` les autres fois. Pour répondre à ces contraintes, `\magicpar` doit se modifier elle-même en `\par` la première fois qu'elle sera exécutée. Voici comment la définir :

```
\def\magicpar{\let\magicpar=\par}
```

Avec une telle définition, \magicpar ne va rien afficher la première fois qu'elle sera exécutée, mais va silencieusement se redéfinir pour devenir égale à \par. Nous avons bien créé une macro locale au groupe semi-simple, qui est neutre pour l'affichage la première fois et équivalente à \par ensuite.

Code n° III-290

```
\def\showfilecontent#1#2{% #1=canal de lecture #2=nom de fichier
  \begingroup
    \tt% sélectionner la fonte à chasse fixe
    \openin#1=#2\relax
    \ifeof#1% si la fin du fichier est déjà atteinte, il n'existe pas et
      Le fichier n'existe pas% afficher le message
    \else% le fichier existe
      \def\do##1{\catcode`##1=12 }%
      \dospecials% neutraliser tous les tokens spéciaux
      \obeyspaces% rendre l'espace actif
      \def\magicpar{\let\magicpar=\par}%
      \loop
        \xread#1 to \currline% lire une ligne
        \unless\ifeof#1% si la fin du fichier n'est pas atteinte
        \leavevmode\magicpar% former le paragraphe (sauf à la 1er itération)
        \currline% afficher la ligne
      \repeat% recommencer
    \fi
    \closein#1\relax
  \endgroup
}

Contenu du fichier : "\showfilecontent\rtest{readtest.txt}", affiché tel quel
```

Contenu du fichier : "Programmer en \TeX{} est facile

et utile", affiché tel quel

9.3. Écrire dans un fichier

9.3.1. La primitive \write

Il est facile de deviner par symétrie que si TEX dispose de 16 canaux pour lire un fichier, il en dispose aussi de 16 pour écrire dans un fichier. Et le pendant de la macro \newread est évidemment \newwrite, qui en partage la syntaxe et qui alloue globalement un canal d'écriture libre.

88 - Règle

TEX dispose de 15 canaux d'écriture portant les numéros de 0 à 15.
On demande l'allocation d'un canal d'écriture par

\newwrite\⟨*macro*⟩

et ce faisant, la \⟨*macro*⟩ devient équivalente à un ⟨*nombre*⟩, qui est un numéro de canal libre. Un ⟨*canal*⟩ est donc un ⟨*nombre*⟩ explicitement écrit ou une macro définie avec \newwrite.

Avant d'écrire dans un fichier, on doit d'abord lier un ⟨*canal*⟩ à un fichier avec :

\openout⟨*canal*⟩= ⟨*nom du fichier*⟩

et lorsque les opérations d'écriture sont terminées, on désolidarise le ⟨*canal*⟩ du fichier par

\closeout⟨*canal*⟩= ⟨*nom du fichier*⟩

Une ligne vide est insérée en fin de fichier.
Pour écrire une ligne dans le fichier, on écrit :

\write⟨*canal*⟩{⟨*texte*⟩}

où le ⟨*texte*⟩ est un code dans lequel les accolades sont équilibrées. L'écriture dans le fichier n'a pas lieu lorsque la commande \write est rencontrée, mais lorsque la page courante est composée. Le ⟨*texte*⟩ est stocké dans la mémoire de TEX et est développé au maximum lorsque TEX écrit dans le fichier. Si l'utilisateur souhaite bloquer le développement, il lui appartient de le bloquer à l'aide des méthodes vues à la page 119.

Les opérations liées à l'écriture dans un fichier (\openout, \write et \closeout) ont la particularité d'avoir lieu lorsque la page est composée, c'est-à-dire plus tard dans le temps que lorsque les instructions sont rencontrées. Ce décalage temporel est bien utile lorsqu'on souhaite écrire un numéro de page puisque l'on s'assure que le bon numéro de page sera écrit, mais il est indésirable pour écrire des données non sensibles au positionnement dans le document. Pour s'affranchir de cette désynchronisation temporelle, il faut faire précéder \openout, \write et \closeout de la primitive « \immediate » qui force les opérations d'écriture à être faites immédiatement lorsqu'elles sont rencontrées.

Dans tout le reste du chapitre, nous utiliserons le canal d'écriture \wtest que nous allons allouer avec \newwrite\wtest.

Voici deux tentatives successives d'écriture dans un même fichier :

Code n° III-291

```
\newwrite\wtest% sera le canal d'écriture dans tout ce chapitre
\immediate\openout\wtest=writetest.txt % lie \wtest au fichier
\immediate\write\wtest{Programmer en \noexpand\TeX{} est   facile.}% écrit une ligne
\immediate\write\wtest{Et utile...}% puis une autre
\immediate\closeout\wtest% défait la liaison
a) Contenu du fichier :\par
"\showfilecontent\rtest{writetest.txt}"% affiche le contenu du fichier
\medbreak
% 2e tentative :
b) Contenu du fichier :\par
\immediate\openout\wtest=writetest.txt % lie \wtest au fichier
\immediate\write\wtest{Essai d'écriture}% écrit une ligne
\immediate\closeout\wtest% défait la liaison
"\showfilecontent\rtest{writetest.txt}"% affiche le contenu du fichier
```

a) Contenu du fichier :
```
"Programmer en \TeX {} est facile.
Et utile..."
```

b) Contenu du fichier :
"Essai d'écriture"

Remarquons à partir de la ligne n° 9, que si on lie un canal à un fichier déjà existant et que l'on écrit à nouveau dans ce fichier, le contenu précédent est effacé, c'est-à-dire que l'écriture recommence au début du fichier.

Remarquons également que l'argument de `\write` est lu par TEX avant d'être détokénizé et écrit dans le fichier. La perte d'informations lorsque TEX lit du code source a donc lieu. Ici en particulier, les espaces consécutifs dans « `est␣␣␣facile` » ont été lus comme un seul espace et écrits comme tel dans le fichier.

Enfin, il faut noter que `\write`, lorsqu'il détokénize le ⟨*texte*⟩, rajoute un espace après les séquences de contrôle.

9.3.2. Programmer des variantes de `\write`

La macro `\noexpwrite`

Si la primitive `\write` développe au maximum ce qu'elle va écrire, il est logique de construire un équivalent qui lui, ne procèdera à aucun développement. Parmi des méthodes à notre disposition pour bloquer le développement, la plus pratique est de recourir à `\unexpanded` :

Code n° III-292

```
1 \def\noexpwrite#1#2{% #1=numéro de canal  #2=texte à écrire
2   \write#1{\unexpanded{#2}}%
3 }
4 \immediate\openout\wtest=writetest.txt
5 \immediate\noexpwrite\wtest{Programmer en \TeX{} est   facile.}%
6 \immediate\noexpwrite\wtest{Et utile...}%
7 \immediate\closeout\wtest
8 Contenu du fichier :\par
9 \showfilecontent\rtest{writetest.txt}% affiche le contenu du fichier
```

Contenu du fichier :
```
Programmer en \TeX {} est facile.
Et utile...
```

Ici encore, comme l'argument #2 a été lu par la macro `\noexpwrite`, les mêmes pertes d'information que précédemment ont eu lieu. Le seul moyen d'écrire dans le fichier ce qu'il y a *exactement* dans le code source est de procéder comme avec la macro `\litterate` (voir page 81).

La macro `\exactwrite`

Nous allons créer l'équivalent de la macro `\litterate` pour l'écriture dans un fichier, c'est-à-dire une macro qui écrit dans un fichier *tous* les caractères qu'on lui donne à lire. Appelons cette macro `\exactwrite` et par commodité, sa syntaxe va ressembler à celle de `\litterate`. La primitive `\immediate` sera appelée et donc l'écriture sera synchrone. On suppose qu'un ⟨*canal*⟩ a déjà été lié au fichier dans lequel on veut écrire. La syntaxe sera :

`\exactwrite{`⟨*canal*⟩⟨*car*⟩⟨*texte*⟩⟨*car*⟩

Ici, ⟨*car*⟩ sera un caractère choisi par l'utilisateur qui servira de délimiteur au ⟨*texte*⟩ que l'on veut écrire dans le fichier, avec la limitation classique que ⟨*car*⟩ doit être choisi de telle sorte qu'il ne figure pas dans le ⟨*texte*⟩.

On va procéder comme avec `\litterate` en rendant inoffensifs des tokens en mettant leurs catcodes à 12, sauf qu'ici, cela va concerner *tous* les octets, de 0 à 255. C'est certes un remède de cheval, mais de cette façon, tous les caractères peuvent être écrits sur un fichier et ce quels que soient l'encodage du fichier source et le régime de catcode en cours.

La méthode va être la suivante :

1) lire le ⟨*canal*⟩ ;
2) ouvrir un groupe, mettre le catcode de tous les octets à 12 ;
3) lire le ⟨*car*⟩ (qui sera de catcode 12 à cause du point précédent) ;
4) isoler le ⟨*texte*⟩ qui se trouve jusqu'au prochain ⟨*car*⟩ ;
5) si ce ⟨*texte*⟩ contient le retour charriot « `^^M` » (de catcode 12) :
   a) écrire dans le fichier le texte se trouvant avant `^^M` ;
   b) retourner en 5) avec en prenant ⟨*texte*⟩ égal à ce qu'il y a près `^^M` ;
6) sinon écrire le ⟨*texte*⟩ dans le fichier et fermer le groupe.

Pour forcer les catcodes de tous les octets à 12, une simple boucle `\for` suffit :

```
\for\xx=0 to 255 \do{\catcode\xx=12 }
```

Ensuite, comme à partir du point n° 3, tout ce qui est lu est constitué de caractères de catcode 12, il va falloir définir un retour charriot de catcode 12 qui nous servira de délimiteur d'argument. On peut s'y prendre ainsi :

```
{\catcode`\^^M=12 \gdef\EOL@char{^^M}}
```

Dans les grandes lignes, la tâches vont se répartir de la façon suivante. Les points n^{os} 1 et 2 sont dévolus à la macro chapeau `\exactwrite`. Une macro auxiliaire `\exactwrite@i` va lire le ⟨*car*⟩, forcément de catcode 12, qui sera son unique argument. Une fois qu'elle s'est emparé de ce ⟨*car*⟩, elle peut définir une sous-macro `\exactwrite@ii` dont l'argument délimité par ce ⟨*car*⟩ sera le ⟨*texte*⟩ qu'elle transmettra à la macro récursive `\exactwrite@iii` qui se chargera des points n° 5 à 6.

Code n° III-293

```
\catcode`\@11
\def\exactwrite#1{% #1=numéro de canal
  \begingroup
    \def\canal@write{#1}% sauvegarde le numéro de canal
    \for\xx=0 to 255\do{\catcode\xx=12 }% donne à tous les octets le catcode 12
    \exactwrite@i% aller lire le <car>
}

\def\exactwrite@i#1{% #1 est le <car> de catcode 12
  \def\exactwrite@ii##1#1{% ##1 est le <texte> à envoyer vers le fichier
    \exactwrite@iii##1\@nil% envoyer <texte> à \exactwrite@iii
  }%
  \exactwrite@ii% va lire tout ce qui se trouve jusqu'au prochain <car>
}

{\catcode`\^^M 12 \gdef\EOL@char{^^M}}% définit le caractère <EOL> de catcode 12

\def\exactwrite@iii#1\@nil{% #1 = <texte> à écrire dans le fichier
  \expsecond{\ifin{#1}}\EOL@char% si #1 contient "retour charriot"
```

```
    {\write@line#1\@nil% écrire la première ligne de #1
    }
    {\immediate\write\canal@write{#1}% sinon : dernière ligne atteinte, l'écrire
    \endgroup% puis sortir du groupe
    }%
}

% les \expandafter provoquent le 1-développement de \EOL@char :
\expandafter\def\expandafter\write@line\expandafter#\expandafter1\EOL@char#2\@nil{%
  \immediate\write\canal@write{#1}% écrit la ligne (ce qui se trouve avant ^^M)
  \exactwrite@iii#2\@nil% recommencer avec ce qui se trouve après "^^M"
}
\catcode`\@12

\immediate\openout\wtest=writetest.txt % lie le canal \wtest au fichier
\exactwrite\wtest|Programmer en \TeX{} est   facile !

Et très utile.|
\immediate\closeout\wtest% et fermer le fichier

Le contenu du fichier "writetest.txt" est :\par
"\showfilecontent\rtest{writetest.txt}"
```

```
Le contenu du fichier "writetest.txt" est :
"Programmer en \TeX{} est   facile !

Et très utile."
```

Plus précisément, voici comment un éditeur de fichiers[7] voit le fichier généré :

```
0000 | 50 72 6F 67  72 61 6D 6D  65 72 20 65  6E 20 5C 54 | Programmer en \T
0010 | 65 58 7B 7D  20 65 73 74  20 20 20 66  61 63 69 6C | eX{} est   facil
0020 | 65 20 21 0A  0A 45 74 20  74 72 E8 73  20 75 74 69 | e !••Et très uti
0030 | 6C 65 2E 0A                                       | le.•
```

L'affichage se divise en trois parties, à gauche l'« offset » (en hexadécimal), c'est-à-dire le numéro d'ordre du premier octet de la ligne (le premier octet du fichier portant le numéro 0), au centre les octets contenus dans le fichier en notation hexadécimale et à droite les caractères correspondant aux octets (dans un codage sur un octet, le plus souvent `latin1`). Les caractères non affichables sont représentés par « • ». Constatons en premier lieu que le caractère « è » est codé sur *un seul* octet (qui est $E8_h$) puisque l'encodage utilisé dans le code source de ce livre est `latin1`[8]. Remarquons aussi que les seuls caractères non affichables portent le code de caractère $0A_h$ qui est 13 en décimal ; ce sont donc les retours charriot. Il est normal d'avoir un retour charriot en fin de fichier, car comme nous l'avons vu, TEX insère une ligne vide en fin de fichier.

9.3.3. Utilisation pratique

Mettons à profit ce que nous savons pour écrire une macro `\exo*`, utile pour rédiger des exercices et afficher leur barème. Supposons que « `\exo{3.5pt}` » soit exécutée. Elle devra afficher le texte suivant :

■ **Exercice 3** 3.5pt/15

7. Programme dont la principale fonctionnalité est de lire les octets d'un fichier un par un, de les afficher ainsi que les caractères ASCII correspondants s'ils sont affichables.

8. Si l'encodage avait été `UTF8`, le caractère « è » aurait été codé sur deux octets : $C3_h$ $A8_h$

où « 15 » est le total des points de toutes les macros \exo du document.

Remarquons que la macro \exo a été appelée 2 fois auparavant puisque cet appel génère le texte de l'exercice n° 3.

La principale difficulté est que le total n'est pas connu lorsque la macro \exo est exécutée puisque d'éventuels autres appels de cette macro qui sont à venir dans le code. C'est donc un cas où il va falloir se servir d'un fichier auxiliaire pour transmettre ce total entre deux compilations avec la contrepartie que lors d'une compilation :

- le fichier n'existe pas, ce qui signifierait qu'il s'agit de la première compilation du document ou que le fichier auxiliaire a été effacé. Dans ce cas, il faudra afficher un total qui représente cette situation particulière, par exemple « ## » ;
- le total est faux parce que depuis la dernière compilation, l'auteur du document a modifié le barème d'un ou plusieurs exercices ou a changé le nombre d'exercices.

Dans ces cas, il faudra lancer *deux* compilations successives pour obtenir le total correct.

Avant tout, appelons \total@points la macro dont le texte de remplacement sera soit « ## », soit le total des points. Il reste à prendre une décision ; comment pouvons-nous stocker dans un fichier le total des points au fur et à mesure que le document est compilé ? Nous pouvons procéder en 3 temps :

1. au début du document, appeler une macro \initexo qui va tester si le fichier auxiliaire existe.
   - dans l'affirmative, elle va exécuter le code qu'il contient avec \input. Ce code définira la macro \total@points avec le barème de la précédente compilation ;
   - sinon, elle va placer dans le texte de remplacement de \total@points le code « \char‘\#\char‘\# » qui affiche « ## ».
2. elle va ouvrir ce fichier en écriture y insérer les caractères « \exocalctotal ». Puis à chaque fois que la macro \exo{⟨*barème*⟩} est rencontrée, ajouter les caractères « +⟨*barème*⟩ » au fichier ;
3. à la fin du document, la macro \stopexo écrira « \relax\endtotal » dans le fichier et le fermera.

Admettons par exemple que « \exo{5pt} », « \exo{3.5pt} » et « \exo{3pt} » sont rencontrées dans le document. Alors, à la fin de la compilation (et au début de la suivante), le fichier auxiliaire contiendra :

```
\exocalctotal+5pt+3.5pt+3pt\relax\endtotal
```

Il nous reste à dire un mot sur la macro \exocalctotal. Elle sera définie pour que son argument soit délimité par \endtotal et son texte de remplacement placera dans la macro \total@points la somme de tous les points. Pour ce faire, \dimexpr et \dimtodec seront employées, avec les éventuels inconvénients concernant les erreurs d'arrondis qui peuvent survenir (voir page 235) :

```
\def\exocalctotal#1\endtotal{\edef\total@points{\dimtodec\dimexpr#1}}
```

Le fichier contenant les points sera nommé comme le fichier maitre (nom qui est le développement de \jobname), mais avec l'extension « .pts »

Code nº III-294

```
\catcode`\@11
\newcount\exo@number% compteur pour le numéro d'exercice

\def\exocalctotal#1\endtotal{\edef\total@points{\dimtodec\dimexpr#1}}

\def\initexo#1{%
  \def\exo@canal{#1}% sauvegarde le canal d'écriture
  \exo@number=0 % initialiser le compteur d'exo
  \iffileexists\exo@canal{\jobname.pts}% si le fichier .pts existe
    {\input \jobname.pts }% le lire et exécuter son contenu
    {\def\total@points{\char`\#\char`\#}}% sinon, définir un total alternatif
  \immediate\openout\exo@canal=\jobname.pts % ouvrir le fichier .pts
  \immediate\write\exo@canal{\noexpand\exocalctotal}% et commencer à y écrire dedans
}

\def\exo#1{% définti la macro qui affiche l'exercice
  \bigbreak% sauter une grande espace verticale
  \immediate\write\exo@canal{+#1}% écrire "+#1" dans le fichier .pts
  \advance\exo@number by 1 % incrémenter le numéro de l'exercice
  \noindent\vrule height1ex width1ex depth0pt % trace le carré
  \kern1ex% insérer une espace horizontale
  {\bf Exercice \number\exo@number}% afficher "Exercice <nombre>"
  \leaders\hbox to.5em{\hss.\hss}\hfill% afficher les pointillés
  #1/\total@points%  puis #1/<total>
  \smallbreak% composer la ligne précédente et sauter une espace verticale
}

\def\stopexo{%
  \immediate\write\exo@canal{\relax\noexpand\endtotal}%
  \immediate\closeout\exo@canal
}
\catcode`\@12
\initexo\wtest

\hfill{\bf Interrogation écrite. Sujet : \TeX{}}\hfill\null
\par
\leavevmode\hfill\vrule height.4pt width 2cm depth0pt\hfill\null
\exo{3pt}
Élaborer un test \litterate/\ifonebyte{<texte>}{<vrai>}{<faux>}/ qui teste, pour une
compilation avec un moteur 8 bits, si le \litterate/<texte>/ est codé sur un seul
octet. Ce test pourrait être utilisé pour déterminer si l'encodage d'un document
est à plusieurs octets (comme l'est UTF8) en prenant comme \litterate/<texte>/
les caractères <<~é~>>, <<~à~>>, etc.

\exo{5,5pt}
Si \verb-#1- est un nombre entier, quel est le test fait par ce code ?
\smallbreak
\hfill
\litterate/\if\string l\expandafter\firstto@nil\romannumeral#1\relax\@nil/
\hfill\null

\exo{4,5pt}
On a vu que pour provoquer un $n$-développement, les \litterate/\expandafter/se plaçaient
en nombre égal à $2^n-1$ devant chaque token précédant celui que l'on veut développer.
Or, ce nombre est {\it impair\/}. Trouver un exemple ou un cas particulier où il faut
placer un nombre {\it pair\/} d'\litterate/\expandafter/ devant un token (on pourra
envisager le cas de 2 \litterate/\expandafter/).
\stopexo
```

Interrogation écrite. Sujet : TEX

■ **Exercice 1** 3pt/13

Élaborer un test `\ifonebyte{<texte>}{<vrai>}{<faux>}` qui teste, pour une compilation avec un moteur 8 bits, si le `<texte>` est codé sur un seul octet. Ce test pourrait être utilisé pour déterminer si l'encodage d'un document est à plusieurs octets (comme l'est UTF8) en prenant comme `<texte>` les caractères « é », « à », etc.

■ **Exercice 2** 5,5pt/13

Si `#1` est un nombre entier, quel est le test fait par ce code ?

```
\if\string l\expandafter\firstto@nil\romannumeral#1\relax\@nil
```

■ **Exercice 3** 4,5pt/13

On a vu que pour provoquer un n-développement, les `\expandafter`se plaçaient en nombre égal à 2^n-1 devant chaque token précédant celui que l'on veut développer. Or, ce nombre est *impair*. Trouver un exemple ou un cas particulier où il faut placer un nombre *pair* d'`\expandafter` devant un token (on pourra envisager le cas de 2 `\expandafter`).

La macro `\dimtodec` est ici utilisée puisque le nombre de points n'est pas forcément entier. D'ailleurs, le côté pratique des « points » est que l'argument de `\exo` est considéré comme un barème par l'utilisateur, mais il est vu comme une vraie dimension par `\dimexpr` !

La manœuvre effectuée avec `\exo` est finalement très courante. Écrire dans un fichier lors d'une compilation pour lire des éléments dans ce fichier à la compilation suivante est utilisé dans beaucoup de contextes [9]. Pour afficher une table des matières en début de document, il faudrait, à l'aide d'une macro spéciale de sectionnement, d'abord stocker dans un fichier les titres assortis de leurs numéros de page, sachant que le numéro de page courant est contenu dans le compteur `\count0` pour plain-TEX et LATEX [10]. Dès lors, à la compilation suivante et sous réserve qu'aucune modification du fichier source n'ait été effectuée entre temps, les instructions écrites dans le fichier reflètent les bons numéros de page.

La construction d'un index repose sur le même stratagème.

Pour les lecteurs consciencieux, voici les réponses aux exercices donnés dans cet exemple :

□ **Solution exercice 1**

La solution apparait immédiatement en utilisant la macro `\ifempty` et `\gobone`, qui va enlever un octet à `#1` et que nous prenons soin de développer avant de tester :

```
\def\ifonebyte#1{\expandafter\ifempty\expandafter{\gobone#1}}
```

□ **Solution exercice 2**

Le test est positif si la lettre « `l` » est la lettre de gauche du nombre écrit en chiffres romains. Ce cas ne se présente que si l'entier est compris entre 50 et 89 (qui s'écrivent « `l` » et « `lxxxix` »). Ce test est donc positif pour les entiers allant de 50 à 89 compris et négatif pour tous les autres.

9. Le format LATEX écrit des informations de localisation de sectionnement et de références croisées dans le fichier « `.aux` » via le canal `\@auxout`.

10. Le nom qui lui est donné diffère selon le format : plain-TEX demande « `\countdef\pageno=0` » tandis que LATEX préfère « `\countdef\c@page=0` ».

□ **Solution exercice 3**
Le code \expandafter\expandafter⟨*token*⟩ procède au 1-développement de ⟨*token*⟩ puis au 1-développement du deuxième token du code qui résulte de ce développement.
Voici un cas où 2 \expandafter se justifient :

Code n° III-295

```
1 \def\identite{Foo Bar}% Prénom Nom
2 \def\beforespace#1 #2\nil{#1}
3 \def\afterspace#1 #2\nil{#2}
4 \def\prenom{\beforespace\identite\nil}
5 \def\nom{\afterspace\identite\nil}
6 Mon prénom : \expandafter\expandafter\prenom
7
8 Mon nom : \expandafter\expandafter\nom
```

Mon prénom : Foo
Mon nom : Bar

À la ligne n° 6, l'\expandafter de gauche effectue ce développement :

\prenom ⟶ \beforespace\identite\nil

et celui de droite 1-développe \itentite en « Foo Bar » de telle sorte que le code finalement obtenu est :

\beforespace Foo Bar\nil

9.4. \newlinechar et \endlinechar

L'entier \newlinechar représente le code du caractère qui ordonne à TEX de commencer une nouvelle ligne lors d'une opération avec \write. Dès lors, il devient possible d'écrire en une opération plusieurs lignes dans un fichier. Plain-TEX assigne −1 à cet entier ce qui signifie qu'aucun caractère ne revêt cette propriété. LATEX, en revanche, y range l'entier 10 qui est le code de caractère de LF (le caractère ^^J).

Code n° III-296

```
1 \newlinechar`\^^J
2 \immediate\openout\wtest=test1.txt
3 \immediate\write\wtest{Une première ligne^^JEt une seconde.}
4 \immediate\closeout\wtest
5 \showfilecontent\rtest{test1.txt}
```

```
Une première ligne
Et une seconde.
```

Quant à l'entier \endlinechar, il s'applique également lors des opérations de lecture, c'est-à-dire que le caractère dont le code est \endlinechar sera inséré à la fin de chaque ligne lue, qu'elle se trouve dans le fichier à lire ou dans le code source. Si ce caractère est modifié localement afin d'obtenir des effets spéciaux lors de la lecture d'un fichier, il faut penser à commenter les fins de ligne de la partie du code source concernée pour que ces effets ne s'y appliquent pas aussi.

Code n° III-297

```
1 \newlinechar`\^^J
2 \immediate\openout\wtest=test2.txt
3 \immediate\write\wtest{Une première ligne^^JEt une seconde.}
```

```
\immediate\write\wtest{Et la dernière.}
\immediate\closeout\wtest

{\endlinechar`\X % insère "X" à chaque fin de ligne
\openin\rtest=test2.txt % les fins de lignes sont commentées
\loop%                  pour éviter que \endlinechar
  \read\rtest to \foo%  ne soit inséré à chaque fin de
  \unless\ifeof\rtest%  ligne du code source
  \meaning\foo\par%     affiche le texte de remplacement de \foo
\repeat%
\closein\rtest}%
```

macro:->Une première ligneX
macro:->Et une seconde.X
macro:->Et la dernière.X

■ Exercice 96

Utiliser `\newlinechar` pour simplifier la macro `\exactwrite` vue à la page 310.

□ Solution

Rappelons-nous que le ⟨*texte*⟩, susceptible de contenir des retours à la ligne, était lu de telle sorte que tous les octets aient un catcode de 12. Au lieu d'écrire une macro récursive qui lit une seule ligne à chaque itération pour l'écrire dans le fichier, il est bien plus élégant et rapide d'assigner à `\newlinechar` l'entier `` `\^^M ``. Une fois ceci fait, il ne reste plus qu'à écrire en une seule fois le ⟨*texte*⟩ dans le fichier :

Code n° III-298

```
\catcode`\@11
\def\exactwrite#1{% #1=numéro de canal
  \begingroup
    \for\xx=0 to 255\do{\catcode\xx=12 }% donne à tous les octets le catcode 12
    \newlinechar=`\^^M % caractère de fin de ligne = retour charriot
    \exactwrite@i{#1}% donne le <canal> d'écriture comme premier argument
}

\def\exactwrite@i#1#2{% #2 est le caractère délimiteur de catcode 12
  \def\exactwrite@ii##1#2{% ##1 est le <texte> à envoyer vers le fichier
    \immediate\write#1{##1}% écrire le <texte> dans le fichier
    \endgroup% puis, sortir du groupe
  }%
  \exactwrite@ii% traiter tout ce qui se trouve jusqu'au prochain #2
}
\catcode`\@12

\immediate\openout\wtest=writetest.txt % lie le canal \wtest au fichier
\exactwrite\wtest|Programmer en \TeX{} est   facile !

Et très utile.|
\immediate\closeout\wtest% et fermer le fichier

Le contenu du fichier "writetest.txt" est :\par
"\showfilecontent\rtest{writetest.txt}"
```

Le contenu du fichier "writetest.txt" est :
"Programmer en \TeX{} est   facile !

Et très utile."

■

9.5. Autres canaux d'entrée-sortie

En plus des 16 canaux de lectures ou d'écriture, TEX dispose de canaux spéciaux. En ce qui concerne la lecture, si le ⟨*canal*⟩ qui suit `\read` n'est pas compris entre 0 et 15 ou si aucun fichier n'a pu être ouvert (notamment parce que le fichier n'existe pas) ou encore si la fin du fichier a été atteinte (et donc le test `\ifeof` est devenu vrai) alors, la lecture de la ligne courante se fait à partir du terminal : TEX attend que l'utilisateur tape un texte au clavier et finisse par la touche « entrée » (⏎) [11]. Un message d'invite est affiché sur le terminal sauf lorsque le ⟨*canal*⟩ est strictement négatif. Si on écrit

```
\read16 to \foo
```

alors, l'invite est « `\foo=` »

Du côté de l'écriture, la primitive `\message{`⟨*texte*⟩`}` a pour action d'écrire le ⟨*texte*⟩ sur le terminal et dans le fichier `log` après avoir développé au maximum ce ⟨*texte*⟩. Il convient de s'assurer que ce ⟨*texte*⟩ est purement développable et dans le cas contraire, bloquer le développement de certaines de ces parties avec `\noexpand` ou `\unexpanded`.

Si on utilise `\write`⟨*canal*⟩, si le ⟨*canal*⟩ n'est pas compris entre 0 et 15 ou si le flux spécifié par le ⟨*canal*⟩ n'a pas été lié à un fichier par `\openout`, l'écriture est dirigée vers le fichier `log` et vers le terminal, et uniquement vers le fichier `log` si le ⟨*canal*⟩ est strictement négatif.

Il est donc assez facile de converser avec l'utilisateur :

Code n° III-299

```
\newlinechar`\^^J
\message{^^JQuel est votre nom ? }
\xread-1 to \username
\message{Bonjour \username.^^J%
Depuis combien d'années pratiquez-vous TeX ? }
\read-1 to \yeartex
\message{%
  \ifnum\yeartex<5 Cher \username, vous êtes encore un débutant !
  \else\ifnum\yeartex<10 Bravo \username, vous êtes un TeXpert !
  \else\ifnum\yeartex<15 Félicitations \username, vous êtes un TeXgourou.
  \else Passez à autre chose, par exemple à Metafont et Metapost !
  \fi\fi\fi^^J}
```

```
Quel est votre nom ? Christian
Bonjour Christian.
Depuis combien d'années pratiquez-vous TeX ? 8
Bravo Christian, vous êtes un TeXpert !
```

Voici pour finir comment TEX pourrait, par dialogue avec l'utilisateur et en procédant par dichotomie, trouver un nombre entre 1 et 100 choisi au hasard par l'utilisateur :

Code n° III-300

```
\catcode`\@11
\def\answer@plus{+}\def\answer@minus{-}\def\answer@equal{=}%
```

11. Ce moyen de collecter interactivement des informations n'est possible que lorsque l'exécutable pfdTEX est appelé *sans* l'option « `-interaction=nonstopmode` » qui l'interdit !

```
\def\nb@found#1{% macro appelée lorsque le nombre (argument #1) est trouvé
  \message{^^JVotre nombre est #1.^^JMerci d'avoir joué avec moi.^^J}%
}

\def\nbguess#1#2{%
  \message{Choisissez un nombre entre #1 et #2.^^J
  Tapez entrée lorsque c'est fait.}%
  \read-1 to \tex@guess% attend que l'on tape sur "entrée"
  \nbguess@i{#1}{#2}%
}

\def\nbguess@i#1#2{%
  \ifnum#1<#2  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    % si #1<#2 (donc le nombre n'est pas trouvé),
    % mettre dans \tex@guess la troncature de la moyenne de #1 et #2
    {\edef\tex@guess{\number\truncdiv{\numexpr#1+#2\relax}2}%
    \message{Je propose \tex@guess.^^J% afficher sur le terminal
    Votre nombre est-il plus grand (+), plus petit (-) ou égal (=) : }%
    \read -1 to \user@answer% lire la réponse de l'utilisateur
    \edef\user@answer{%
      \expandafter\firstto@nil\user@answer\relax\@nil% ne garder que le 1er caractère
    }%
    \ifxcase\user@answer% envisager les cas "+", "-" et "="
      \answer@plus{\exparg\nbguess@i{\number\numexpr\tex@guess+1\relax}{#2}}%
      \answer@minus{\expsecond{\nbguess@i{#1}}{\number\numexpr\tex@guess-1\relax}}%
      \answer@equal{\nb@found\tex@guess}%
    \elseif% si la réponse ne commence pas par "+", "-" ou "="
      \message{Je n'ai pas compris votre réponse}% afficher message erreur
      \nbguess@i{#1}{#2}% et recommencer avec les mêmes nombres
    \endif
    }
    % si #1>=#2, le nombre est trouvé
    {\nb@found{#1}}%
}
\catcode`\@12

\nbguess{1}{100}
```

```
Choisissez un nombre entre 1 et 100.
Tapez entrée lorsque c'est fait.

Je propose 50.
Votre nombre est plus grand (+), plus petit (-) ou égal (=) : -

Je propose 25.
Votre nombre est plus grand (+), plus petit (-) ou égal (=) : +

Je propose 37.
Votre nombre est plus grand (+), plus petit (-) ou égal (=) : -

Je propose 31.
Votre nombre est plus grand (+), plus petit (-) ou égal (=) : -

Je propose 28.
Votre nombre est plus grand (+), plus petit (-) ou égal (=) : +

Je propose 29.
Votre nombre est plus grand (+), plus petit (-) ou égal (=) : +

Votre nombre est 30.
```

```
Merci d'avoir joué avec moi.
```

Chapitre 10

Autres algorithmes

Dans ce chapitre qui sera dépourvu d'exercice, nous allons nous attacher à écrire quelques algorithmes – des classiques de la programmation – en TEX.

10.1. Macros de calcul

10.1.1. La suite de Syracuse

Pour générer une suite de Syracuse, le procédé consiste à choisir un nombre entier strictement positif. S'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant l'opération, on génère une suite d'entiers appelée « suite de Syracuse ». Si à partir d'un certain rang, l'entier 1 est atteint, la suite de valeurs 1 ; 4 ; 2 se répète infiniment et ces trois valeurs constituent ce que l'on appelle le « cycle trivial ». La conjecture de Syracuse, non démontrée, affirme que, quel que soit l'entier choisi au départ, on arrive au bout d'un nombre fini d'itérations à la valeur 1 et donc au cycle trivial.

Nous allons programmer une macro `\syracuse*{`⟨*nombre*⟩`}`, purement développable qui va afficher toutes les valeurs générées selon le procédé décrit ci-dessus, et s'arrêter lorsque la valeur 1 est atteinte.

Nous aurons recours au test

`\ifodd`⟨*nombre*⟩⟨*code vrai*⟩`\else`⟨*code faux*⟩`\fi`

qui est vrai si le ⟨*nombre*⟩ est impair.

Code nº III-301

```
\def\syracuse#1{%
  #1% affiche le nombre
  \ifnum#1>1 % si le nombre est >1
    , % afficher une virgule+espace
    \ifodd#1 % s'il est pair
      \exparg\syracuse% appeler la macro \syracuse
      {\number\numexpr3*#1+1% avec 3*n+1
      \expandafter\expandafter\expandafter}% après avoir rendu la macro terminale
    \else % s'il est pair
      \expandafter\syracuse\expandafter% appeler la macro \syracuse
      {\number\numexpr#1/2% avec n/2
      \expandafter\expandafter\expandafter}% après avoir rendu la macro terminale
    \fi
  \else% si le nombre est 1
    .% afficher un point
  \fi
}
a) \syracuse{20}\par
b) \syracuse{14}\par
c) \syracuse{99}\par
d) \edef\foo{\syracuse{15}}\meaning\foo
```

a) 20, 10, 5, 16, 8, 4, 2, 1.
b) 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
c) 99, 298, 149, 448, 224, 112, 56, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
d) macro:->15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Les \expandafter qui se trouvent à la fin de l'argument de \syracuse lors de l'appel récursif sont développés par \numexpr et rendent la récursivité terminale. Il est cependant possible d'écrire un code comportant moins d'\expandafter en mettant le test \ifodd à l'intérieur du nombre évalué par \numexpr :

Code nº III-302

```
\def\syracuse#1{%
  #1% afficher le nombre
  \ifnum#1>1 % si le nombre est >1
    , % afficher une virgule+espace
    \exparg\syracuse{% appeler la macro \syracuse
      \number\numexpr% avec le nombre :
        \ifodd#1 3*#1+1% 3n+1 si #1 est impair
        \else #1/2%      n/2 sinon
        \fi%
      \expandafter}% avant de rendre la récursivité terminale
  \else
  .% si #1=1, afficher un point
  \fi
}
a) \syracuse{20}\par
b) \syracuse{14}\par
c) \syracuse{99}\par
d) \edef\foo{\syracuse{15}}\meaning\foo
```

a) 20, 10, 5, 16, 8, 4, 2, 1.
b) 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
c) 99, 298, 149, 448, 224, 112, 56, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
d) macro:->15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

10.1.2. Calculer une factorielle

La factorielle d'un nombre entier positif n, notée $n!$, est égale au produit des n premiers entiers strictement positifs. Par convention $0! = 1$.

L'algorithme récursif classique est le suivant où l'on considère que l'argument de la macro `\factorielle`⋆ est positif ou nul :

1) si `#1` = 0, `\factorielle{0}` = 1 ;
2) sinon, `\factorielle{#1}` = `#1` × `\factorielle{#1-1}`.

L'algorithme est d'une simplicité biblique et sa traduction en TEX est immédiate :

Code n° III-303

```
\def\factorielle#1{%
  \ifnum#1=0 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {1}% "1" si #1=0}
    {#1*\exparg\factorielle{\number\numexpr#1-1}}%
}
a) \factorielle{0}\qquad
b) \factorielle{3}\qquad
c) \edef\foo{\factorielle{8}}\meaning\foo
```

a) 1 b) 3*2*1*1 c) macro:->8*7*6*5*4*3*2*1*1

La première chose qui saute aux yeux est que l'expression est bien générée, mais qu'elle n'est pas évaluée par un `\numexpr`.

Pour y remédier, l'idée va être de partir d'une macro chapeau qui se chargera de lancer `\number\numexpr` afin de provoquer, par développement maximal, la génération de l'enchainement des multiplications obtenu précédemment et son calcul. Cette macro chapeau ajoutera également le `\relax` final pour stopper le calcul effectué par `\numexpr`.

Code n° III-304

```
\catcode`\@11
\def\factorielle#1{%
  \number\numexpr\factorielle@i{#1}\relax% appelle \factorielle@i
  % en lançant le développement maximal
}
\def\factorielle@i#1{%
  \ifnum#1=0 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {1}% "1" si #1=
    {#1*\exparg\factorielle@i{\number\numexpr#1-1}}%
}
\catcode`\@12
a) \factorielle{0}\qquad
b) \factorielle{3}\qquad
c) \edef\foo{\factorielle{8}}\meaning\foo
```

a) 1 b) 6 c) macro:->40320

10.1.3. Calculer un PGCD

Le célèbre algorithme d'Euclide permet de calculer le plus grand diviseur commun à deux entiers, voici comment procéder :

1) soit a le plus grand nombre et b le plus petit
2) effectuer la division euclidienne $a \div b$ et appeler r le reste ;

3) si $r = 0$, le PGCD est le diviseur précédent b ;

4) sinon, retourner en 2 en prenant $a \longleftarrow b$ et $b \longleftarrow r$

Cet algorithme est des plus faciles à traduire en TEX. La macro chapeau \PGCD⋆, purement développable, se chargera du point nº 1, une macro auxiliaire du point nº 2 et une autre macro auxiliaire des deux derniers points. La récursivité se jouera sur les deux macros auxiliaires :

Code nº III-305

```
\catcode`\@11
\def\PGCD#1#2{%
  \ifnum#1<#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\PGCD@i{#2}{#1}}% si #1<#2, mettre #2 (le  grand) dans le premier argument
    {\PGCD@i{#1}{#2}}%
}

\def\PGCD@i#1#2{% #1=a   #2=b avec a>b
  \exptwoargs\PGCD@ii% appeler la macro récursive avec
      {\number\numexpr#1-#2*\truncdiv{#1}{#2}}% le reste de a/b
      {#2}% et le divisieur b
}

\def\PGCD@ii#1#2{% #1=reste r   #2=diviseur b
  \ifnum#1=\z@\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {#2}% si le reste est nul, renvoyer b
    {\PGCD@i{#2}{#1}}% sinon, recommencer avec b et r
}
\catcode`\@12
a) \PGCD{120}{75}\qquad
b) \PGCD{64}{180}\qquad
c) \PGCD{145}{64}\qquad
d) \edef\foo{\PGCD{1612}{299}}\meaning\foo
```

a) 15 b) 4 c) 1 d) macro:->13

Il peut être utile d'écrire les divisions successives qui permettent d'arriver au résultat. Par exemple, si l'on écrit \calcPGCD{39}{15}, on aurait les étapes suivantes :

$39 = 15 \times 2 + 9$

$15 = 9 \times 1 + 6$

$9 = 6 \times 1 + 3$

$6 = 3 \times 2 + 0$

La macro \calcPGCD⋆ se chargera d'afficher les étapes ci-dessus. Comme il s'agit d'une macro qui produit un résultat typographique, il n'y a pas besoin de la rendre purement développable. Peu de choses changent ; un \edef stocke le quotient pour éviter de le calculer deux fois et pour chaque ligne de calcul, les nombres sont affichés lorsqu'ils sont disponibles. Le mode math commence donc dans la première macro auxiliaire et se termine dans la seconde, une fois que le reste a été calculé.

Code nº III-306

```
\catcode`\@11
\def\calcPGCD#1#2{%
  \ifhmode\par\fi% si en mode horizontal, former le paragraphe
  \ifnum#1<#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\calcPGCD@i{#2}{#1}}% si #1<#2, mettre #2 (le  grand) dans le premier argument
    {\calcPGCD@i{#1}{#2}}%
}

```

```
\def\calcPGCD@i#1#2{% #1=a   #2=b avec a>b
  \edef\calcPGCD@quotient{\number\truncdiv{#1}{#2}}% stocke le quotient
  $#1=\calcPGCD@quotient\times#2% en mode maths, afficher "a=q*b" (à suivre)
  \exptwoargs\calcPGCD@ii% appeler la macro récursive avec
      {\number\numexpr#1-#2*\calcPGCD@quotient}% le reste de a/b
      {#2}% et le divisieur b
}

\def\calcPGCD@ii#1#2{% #1=reste r   #2=diviseur b
  +#1$\par% (suite du mode math) afficher "+r", fermer le mode math et \par
  \ifnum#1=\z@\expandafter\firstoftwo\else\expandafter\secondoftwo\fi%
    {}% si le reste est nul, ne rien faire
    {\calcPGCD@i{#2}{#1}}% sinon, recommencer avec b et r
}
\catcode`\@12
\calcPGCD{39}{15}\medbreak
\calcPGCD{1612}{299}
```

$39 = 2 \times 15 + 9$
$15 = 1 \times 9 + 6$
$9 = 1 \times 6 + 3$
$6 = 2 \times 3 + 0$

$1612 = 5 \times 299 + 117$
$299 = 2 \times 117 + 65$
$117 = 1 \times 65 + 52$
$65 = 1 \times 52 + 13$
$52 = 4 \times 13 + 0$

Pour améliorer la lisibilité de ces lignes de calcul, un raffinement typographique supplémentaire pourrait être d'aligner sur une même verticale les signes « = », « × » et « + ». Comme ces signes sont toujours les mêmes, le meilleur moyen d'y parvenir va être de bâtir un alignement avec `\halign` où ces signes feront partie prenante du préambule de façon à ne pas être écrits à chaque ligne.

Examinons donc tout d'abord le préambule. Il y aura 4 colonnes, chacune étant réservée à un nombre comme dans

$$39 = 15 \times 2 + 9$$

Décidons que la première colonne est composée au fer à droite (le nombre sera au plus près du signe « = »), elle sera déclarée `$\hfil#$` dans le préambule.

Pour la seconde colonne, nous allons placer le signe « = » avant le contenu de la colonne. Il est important de comprendre qu'en mode mathématique, le signe « = » doit être précédé de quelque chose pour qu'une espace mathématique soit insérée avant lui [1] :

Code n° III-307

```
\frboxsep=0pt % encadrer au plus proche
\leavevmode\frbox{$=$} n'est pas identique à \frbox{${}={}$}
```

⊟ n'est pas identique à ⊏=⊐

1. Tous les composants d'une formule mathématique sont appelés *atomes*.
Pour de plus amples informations sur le type d'atomes mathématiques composant une formule (le signe = étant de type « *Rel* ») et les espaces qu'ils génèrent, voir le TEXbook pages 184-185, 200 ainsi que l'annexe G.

En effet, dans le premier cas, ce qui se trouve à gauche de l'atome de relation « = » est vide, il n'y a donc aucun espace mathématique qui est ajouté. En revanche, dans la seconde boite, ce qui se trouve à gauche de « = » est « } » qui est un atome « *Close* » et, en vertu des règles exposées dans le TEXbook, l'espace mathématique inséré à gauche de « = » devient le même qu'avec un atome « *Ord* » comme l'est un chiffre ou une lettre.

La conséquence de tout cela est que la deuxième colonne aura comme préambule

```
${}=\hfil#$
```

où le `\hfil` nous assure que le nombre, représenté par `#`, sera composé au fer à droite (il sera donc au plus près du signe « × » qui suit).

Puisque les atomes « × » et « + » étant de type *Rel*, le même raisonnement doit être suivi pour la 3e et 4e colonnes qui auront comme préambules

```
${}\times#\hfil$
```

et

```
${}+#\hfil$
```

Un deuxième problème est qu'une cellule d'un alignement est équivalente à un groupe. Par conséquent, pour pouvoir se transmettre dans l'alignement, l'assignation du quotient doit être globale. Enfin, le `\ifnum` et le `\fi` doivent se trouver dans la même cellule.

Code no III-308

```
\catcode`\@11
\def\calcPGCD#1#2{%
  \vtop{% mettre l'alignement dans une \vtop
    \halign{% les "#" doivent être doublés puisqu'à l'intérieur d'une macro
    $\hfil##$&${}=\hfil##$&${}\times##\hfil$&${}+##\hfil$% préambule
    \cr% fin du préambule et début de la première cellule
    \ifnum#1<#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {\calcPGCD@i{#2}{#1}}% si #1<#2, mettre #2 (le  grand) dans le premier argument
      {\calcPGCD@i{#1}{#2}}%
    \crcr% fin de l'alignement
    }%
  }%
}

\def\calcPGCD@i#1#2{% #1=a   #2=b avec a>b
  \xdef\calcPGCD@quotient{\number\truncdiv{#1}{#2}}% stocke le quotient
  #1 & \calcPGCD@quotient & #2 &% afficher "a=q*b" (à suivre)
  \exptwoargs\calcPGCD@ii% appeler la macro récursive avec
      {\number\numexpr#1-#2*\calcPGCD@quotient}% le reste de a/b
      {#2}% et le divisieur b
}

\def\calcPGCD@ii#1#2{% #1=reste r   #2=diviseur b
  #1% (suite de l'alignement) afficher "+r"
  \cr% et terminer la ligne en cours
  \ifnum#1=\z@\expandafter\firstoftwo\else\expandafter\secondoftwo\fi%
    {}% si le reste est nul, ne rien faire
    {\calcPGCD@i{#2}{#1}}% sinon, recommencer avec b et r
}
\catcode`\@12
a) \calcPGCD{39}{15}\medbreak
```

```
b) \calcPGCD{1612}{299}
```

a) $39 = 2 \times 15 + 9$
$15 = 1 \times 9 + 6$
$9 = 1 \times 6 + 3$
$6 = 2 \times 3 + 0$

b) $1612 = 5 \times 299 + 117$
$299 = 2 \times 117 + 65$
$117 = 1 \times 65 + 52$
$65 = 1 \times 52 + 13$
$52 = 4 \times 13 + 0$

10.1.4. Convertir un nombre en base 2

Pour convertir un entier positif n quelconque écrit en base 10 en base b, voici l'algorithme à utiliser :

1) effectuer la division de n par b. Appeler q est le quotient et r le reste ;
2) si $q \neq 0$, retourner en 1 avec $n \longleftarrow q$;
3) sinon, le nombre cherché est constitué des restes, lus du *dernier au premier*.

Voici les divisions euclidiennes qu'il faut poser pour convertir 43 en base 2 :

$$\begin{array}{rr|l}
43 & 2 & \\
\hline
\mathbf{1} & 21 & 2 \\
\hline
 & \mathbf{1} & 10 \quad 2 \\
\hline
 & & \mathbf{0} \quad 5 \quad 2 \\
\hline
 & & & \mathbf{1} \quad 2 \quad 2 \\
\hline
 & & & & \mathbf{0} \quad 1 \quad 2 \\
\hline
 & & & & & \mathbf{1} \quad 0
\end{array}$$

Et donc, en lisant les restes dans l'ordre inverse, le nombre 43 s'écrit 101011 en base 2.

Appelons `\baseconv*{`$\langle n \rangle$`}` la macro qui va convertir en base 2 le nombre $\langle n \rangle$ écrit en base 10. Donnons-nous une difficulté supplémentaire de taille : le code devra être purement développable. Ainsi, écrire `\edef\foo{\baseconv{43}}` met dans le texte de remplacement de `\foo` les caractères `101011`.

Dans un premier temps, nous allons écrire la macro `\baseconv` de façon à ce qu'elle écrive les restes dans l'ordre où ils sont trouvés. Afin que la macro soit purement développable, la macro `\truncdiv` sera mise à contribution.

Code n° III-309

```
\catcode`\@11
\def\baseconv#1{%
  \unless\ifnum#1=\z@ % si #1 est différent de 0
    \number\numexpr#1-2*\truncdiv{#1}2\relax% écrire le reste
    \exparg\baseconv{\number\truncdiv{#1}2\expandafter}% recommencer avec #1/2
  \fi% après que le \fi ait été lu
}
\catcode`\@12
a) \baseconv{43}\qquad
b) \baseconv{32}\qquad
c) \edef\foo{\baseconv{159}}\meaning\foo
```

a) 110101 b) 000001 c) macro:->11111001

Cela fonctionne, les restes sont bien écrits au fur et à mesure de leur calcul à la ligne nº 4. Certes, la macro pourrait être optimisée puisque `\truncdiv{#1}2` est calculé deux fois, mais laissons cela pour l'instant. La question qui se pose est « comment s'y prendre pour inverser l'ordre des restes ? »

L'idée est de copier la méthode des « réservoirs » mise en œuvre dans la macro `\reverse` programmée à la page 191. Pour ce faire, la macro `\baseconv` ne va devenir qu'une macro chapeau qui va transmettre à la vraie macro récursive `\baseconv@i` deux arguments non délimités : le premier, vide au début, contiendra la liste des restes écrits dans l'ordre désiré. Le second sera le nombre $\langle n \rangle$. Le fonctionnement de cette macro sera des plus simples.

Si $\langle n \rangle$ est nul, l'affichage de l'argument #1 sera fait. Sinon, la macro `\baseconv@i`

– ajoutera *avant* les autres restes le chiffre

```
\number\numexpr#2-2*\truncdiv{#2}2\relax
```

Notons que la primitive `\number` ne s'applique ici qu'à `\numexpr...\relax` qui a le statut de registre d'entier. Il n'y a donc aucun risque pour `\number` de lire un entier supérieur à $2^{31}-1$ si les autres restes qui sont à la suite sont en grand nombre ;

– remplacera l'argument #2 par `\number\truncdiv{#2}2` ;

– une fois ceci fait, elle s'appellera elle-même.

Code nº III-310

```
\catcode`\@11
\def\baseconv#1{%
  \baseconv@i{}{#1}%
}

\def\baseconv@i#1#2{% #1=restes   #2=n
  \ifnum#2=\z@\expandafter\firstoftwo\else\expandafter\secondoftwo\fi% si n=0
    {#1}% si <n>=0, afficher tous les restes
    {% sinon, recommencer en
    \exptwoargs\baseconv@i% ajoutant le reste courant avant #1
      {\number\numexpr#2-2*\truncdiv{#2}2\relax #1}
      {\number\truncdiv{#2}2}% et en prenant n:=n/2
    }%
}

\catcode`\@12
a) \baseconv{43}\qquad
b) \baseconv{32}\qquad
c) \edef\foo{\baseconv{159}}\meaning\foo
```

a) 101011 b) 100000 c) macro:->10011111

10.1.5. Aller vers des bases plus élevées

Pourquoi ne pas aller plus loin et convertir un nombre en une base arbitraire ? La limitation réside dans le fait que les symboles utilisés pour les chiffres ne sont pas extensibles à l'infini. Nous avons les 10 chiffres et les 26 lettres (que nous prendrons majuscules) ce qui nous fait 36 signes différents, que nous appellerons « ⟨*chiffre*⟩ ».

Nous allons donc d'emblée nous limiter à une conversion vers une base inférieure ou égale à 36.

Le premier travail est de construire une macro `\basedigit*`, purement développable, de syntaxe

`\basedigit{`⟨*nombre*⟩`}`

qui se développe en un ⟨*chiffre*⟩ selon le ⟨*nombre*⟩ inférieur à 36 passé en argument. La méthode bête et méchante du `\ifcase` et des 36 `\or` va être privilégiée ici.

Pour que le ⟨*chiffre*⟩ puisse être facilement accessible, l'ensemble

`\romannumeral\basedigit{`⟨*nombre*⟩`}`

doit donner le ⟨*chiffre*⟩ par 1-développement du `\romannumeral`. Cela implique de 1-développer les `\or` restants et le `\fi` avant que `\romannumeral` ait fini son travail :

Code n° III-311

```
\catcode`\@11
\def\z@@{\expandafter\z@\expandafter}
\def\basedigit#1{%
  \ifcase#1
    \z@@ 0%
    \or\z@@ 1\or\z@@ 2\or\z@@ 3\or\z@@ 4\or\z@@ 5\or\z@@ 6\or\z@@ 7%
    \or\z@@ 8\or\z@@ 9\or\z@@ A\or\z@@ B\or\z@@ C\or\z@@ D\or\z@@ E%
    \or\z@@ F\or\z@@ G\or\z@@ H\or\z@@ I\or\z@@ J\or\z@@ K\or\z@@ L%
    \or\z@@ M\or\z@@ N\or\z@@ O\or\z@@ P\or\z@@ Q\or\z@@ R\or\z@@ S%
    \or\z@@ T\or\z@@ U\or\z@@ V\or\z@@ W\or\z@@ X\or\z@@ Y\or\z@@ Z%
  \fi
}
\long\def\>#1<{\detokenize{#1}}
a) \expandafter\>\romannumeral\basedigit{23}<\quad
b) \expandafter\>\romannumeral\basedigit{6}<
\catcode`\@12
```

a) N b) 6

Le test `\ifcase`⟨*n*⟩, développé par `\romannumeral`, avale tout jusqu'au ⟨*n*⟩^{e} `\or`. Par exemple, si l'argument ⟨*n*⟩ est 3, `\romannumeral` va développer le test et trouver ceci :

`\z@@ C\or \z@@ D......\fi`

La macro `\z@@` sera 1-développée et `\romannumeral` va voir

`\expandafter\z@\expandafter C\or \z@@ D......\fi`

`\romannumeral` poursuit le développement puisque n'ayant toujours pas trouvé de nombre. Le pont d'`\expandafter` va 1-développer « `\or \z@@ D...\fi` », faisant disparaitre le tout. Il va rester

`\z@ C`

Ensuite, le `\z@` va stopper la lecture du nombre par `\romannumeral`, laissant « `C` ».

Pour que la macro `\baseconv` reste purement développable, il va falloir lui transmettre à chaque itération tous les arguments dont elle a besoin et en particulier la base, qui sera un argument figé et lu à chaque fois.

De plus, pour éviter que `\truncdiv{⟨n⟩}{⟨b⟩}` soit calculé deux fois, ce quotient, une fois calculé, sera transmis comme argument supplémentaire à une autre sous macro.

Voici le code complet :

Code n° III-312

```
\catcode`\@11
\def\baseconv#1#2{% #1=base  #2=nombre à convertir
  \ifnum#1<37 % base maxi = 36 (10 signes chiffres + 26 signes lettres)
    \antefi
    \baseconv@i{}{#2}{#1}%
  \fi
}
\def\baseconv@i#1#2#3{% #1=restes  #2=n  #3=base
  \ifnum#2=\z@ \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {#1}% si n=0, afficher tous les restes
    {% si non, transmettre en dernier argument
    \expsecond{\baseconv@ii{#1}{#2}{#3}}%
      {\number\truncdiv{#2}{#3}}% le quotient
    }%
}
\def\baseconv@ii#1#2#3#4{% #1=restes  #2=n  #3=base  #4=q
  \exparg\baseconv@i% recommencer, en ajoutant le <chiffre> avant les restes
    {\romannumeral\basedigit{\number\numexpr#2-#4*#3\relax}#1}%
    {#4}% et en remplaçant n par q
    {#3}%
}
\def\z@@{\expandafter\z@\expandafter}%
\def\basedigit#1{%
  \ifcase#1
    \z@@ 0%
    \or\z@@ 1\or\z@@ 2\or\z@@ 3\or\z@@ 4\or\z@@ 5\or\z@@ 6\or\z@@ 7%
    \or\z@@ 8\or\z@@ 9\or\z@@ A\or\z@@ B\or\z@@ C\or\z@@ D\or\z@@ E%
    \or\z@@ F\or\z@@ G\or\z@@ H\or\z@@ I\or\z@@ J\or\z@@ K\or\z@@ L%
    \or\z@@ M\or\z@@ N\or\z@@ O\or\z@@ P\or\z@@ Q\or\z@@ R\or\z@@ S%
    \or\z@@ T\or\z@@ U\or\z@@ V\or\z@@ W\or\z@@ X\or\z@@ Y\or\z@@ Z%
  \fi
}
\catcode`\@12
a) "\baseconv{20}{21587}"\qquad
b) "\baseconv{16}{32}"\qquad
c) \edef\foo{\baseconv{16}{159}}%
   "\meaning\foo"\qquad
d) "\baseconv{2}{43}"
```

a) "2DJ7" b) "20" c) "macro:->9F" d) "101011"

10.2. Macros manipulant les listes

Donnons-nous la définition suivante : une « liste » est une collection ordonnée d'« éléments » séparés les uns des autres par un « séparateur » que nous prendrons égal à la virgule. Un « élément » est un ensemble de tokens où les accolades éventuelles qui le composent sont équilibrées et qui ne contient pas le séparateur, sauf si celui-ci est entre accolades.

Le but est d'exécuter les actions suivantes sur une liste donnée : trouver un élément par son numéro, trouver le numéro d'un élément, supprimer un élément,

le déplacer, insérer un élément à une position, et « exécuter » la liste, c'est-à-dire que chaque élément deviendra l'argument d'une macro et le code obtenu sera lu par TEX.

Une liste est stockée par `\def` dans le texte de remplacement d'une macro. Une telle macro recevra le nom de `\⟨macrolist⟩`.

10.2.1. Trouver un élément d'une liste

Trouver le ⟨*n*⟩e élément se fera à l'aide de la macro `\finditem` selon cette syntaxe :

`\finditem\⟨macrolist⟩{⟨n⟩}`

Le ⟨*n*⟩e élément de la liste sera renvoyé. La macro sera purement développable, sous réserve que l'élément renvoyé le soit.

Du côté de la mise en œuvre, nous utiliserons la méthode classique : la macro `\finditem` sera une macro chapeau. Afin qu'elle soit purement développable, la macro récursive `\finditem@i` doit recevoir à chaque itération *tous* les paramètres dont elle a besoin sous forme d'arguments, c'est-à-dire

1. la position courante, initialisée à 1 par la macro chapeau et incrémentée à chaque itération ;
2. la position de l'élément à trouver ;
3. l'élément courant qui précède le reste de la liste, que la macro chapeau aura pris soin de terminer par une virgule et un quark.

Le travail de la macro consistera en deux tests imbriqués. Le premier qui regardera si la liste est arrivée à sa fin (en comparant l'élément courant au quark) et l'autre si la position courante correspond à la position cherchée.

Code n° III-313

```
\catcode`\@11
\def\quark@list{\quark@list}% quark de fin de liste
\def\finditem#1{% #1 = \<macrolist>, la position est lue plus tard par \finditem@i
  \exparg\finditem@i{#1}% 1-développe la \<macrolist>
}
\def\finditem@i#1#2{% #1 = liste   #2=position cherchée
  \finditem@ii{1}{#2}#1,\quark@list,% appelle la macro récursive
}
\def\finditem@ii#1#2#3,{% #1=position courante  #2=position cherchée  #3=élément courant
  \ifx\quark@list#3\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {}% si la fin de liste est atteinte, ne rien renvoyer
    {% sinon
    \ifnum#1=#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {% si la position est la bonne
      \finditem@iii{#3}% renvoyer #3 et manger les éléments restants
      }
      {% si la position n'est pas la bonne, recommencer en incrémentant #1
      \exparg\finditem@ii{\number\numexpr#1+1}{#2}%
      }%
    }%
}
\def\finditem@iii#1#2\quark@list,{% renvoyer #1 et manger le reste de la liste
  #1%
}
\catcode`\@12
\def\liste{a,bcd,{ef},g,hij,kl}
```

```
a) \edef\foo{\finditem\liste5}\meaning\foo\qquad
b) \edef\bar{\finditem\liste3}\meaning\bar
```

a) macro:->hij b) macro:->ef

L'élément « {ef} » est dépouillé de ses accolades lorsqu'il est lu par la macro récursive \finditem@ii. Pour nous prémunir de cette erreur, nous devons insérer un token (nous prendrons \relax) avant chaque élément, et supprimer ce \relax à chaque fois que figure l'élément #3 dans la macro \finditem@ii, ainsi qu'à la ligne nº 23 où il est l'argument #1 de la macro.

De plus, nous allons aussi définir une macro \finditemtocs⋆, qui assignera l'élément trouvé à une macro selon la syntaxe

\finditemtocs\⟨*macrolist*⟩{⟨*n*⟩}\⟨*macro*⟩

Comme cette variante n'a pas à être purement développable, il lui suffira de définir \finditemtocs@ii pour exécuter une l'assignation. Tout le reste sera identique à \finditem.

Code nº III-314

```
\catcode`\@11
\def\quark@list{\quark@list}% quark de fin de liste
\def\finditem#1{% #1 = \<macro>, la position est lue plus tard par \finditem@i
  \exparg\finditem@i{#1}% 1-développe la \<macro>
}
\def\finditem@i#1#2{% #1 = liste   #2=position cherchée
  \ifnum#2>\z@% ne faire quelque chose que si la position est >0
    \antefi
    \finditem@ii{1}{#2}\relax#1,\quark@list,% appelle la macro récursive
  \fi
}
\def\finditem@ii#1#2#3,{% #1=position courante  #2=position cherché  #3=élément courant
  \expandafter\ifx\expandafter\quark@list\gobone#3%
  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {}% si la fin de liste est atteinte, ne rien renvoyer
    {% sinon
    \ifnum#1=#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {% si la position est la bonne
      \finditem@iii{#3}% renvoyer #3 et manger les éléments restants
      }
      {% si la position n'est pas la bonne, recommencer en incrémentant #1
      \exparg\finditem@ii{\number\numexpr#1+1}{#2}\relax
      }%
    }%
}
\def\finditem@iii#1#2\quark@list,{% renvoyer #1 et manger le reste de la liste
  \gobone#1%
}
\def\finditemtocs#1#2#3{% #1 = \<macro>  #2=position  #3=macro à définir
  \def\finditemtocs@iii##1##2\quark@list,{% renvoyer #1 et manger le reste de la liste
    \expandafter\def\expandafter#3\expandafter{\gobone##1}%
  }%
  \let#3=\empty
  \exparg\finditemtocs@i{#1}{#2}% 1-développe la \<macro>
}
\def\finditemtocs@i#1#2{% #1 = liste   #2=position cherchée
  \ifnum#2>\z@% ne faire quelque chose que si la position est >0
    \antefi\finditemtocs@ii{1}{#2}\relax#1,\quark@list,% appelle la macro récursive
```

```
  \fi
}
\def\finditemtocs@ii#1#2#3,{%
% #1=position courante  #2=position cherché  #3=\relax + élément courant
  \expandafter\ifx\expandafter\quark@list\gobone#3%
  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {}% si fin de liste ne rien faire. Sinon, si position bonne
    {\ifnum#1=#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {\finditemtocs@iii{#3}% renvoyer #3 et manger les éléments restants
      }% si la position n'est pas la bonne, recommencer en incrémentant #1
      {\exparg\finditemtocs@ii{\number\numexpr#1+1}{#2}\relax%
      }%
    }%
}
\catcode`\@12
\def\liste{a,bcd,{ef},g,hij,kl}
a) "\finditem\liste5"\qquad
b) \edef\bar{\finditem\liste3}\meaning\bar\qquad
c) \finditemtocs\liste{3}\foo\meaning\foo
```

a) "hij" b) macro:->{ef} c) macro:->{ef}

10.2.2. Trouver la position d'un élément

Il faut annoncer la couleur d'entrée : la macro `\positem`⋆, chargée de trouver la position d'un élément et de syntaxe

`\positem\`⟨*macrolist*⟩`{`⟨*élément*⟩`}`

ne sera pas purement développable. En effet, il faudra comparer l'élément courant à l'élément cherché et pour ce faire, un test `\ifx` comparera deux macros contenant ces éléments. Définir ces macros rend donc la macro non développable.

Si l'élément cherché n'est pas dans la liste, la position renvoyée sera égale 0.

À nouveau, une variante `\positemtocs`⋆ sera programmée. Toutefois, nous allons essayer de ne pas procéder comme avec `\finditem` et ne pas réécrire presque complètement le code de la macro `\positem` en modifiant deux ou trois choses, ce qui est lourd et assez maladroit. Il est possible de faire mieux. Pour cela, il faut définir ce que l'on appelle en programmation un « *hook* ». Un hook est un appel à une macro que l'on met à certains endroits du code afin d'exécuter l'action faite par la macro. Pour modifier cette action, il suffit de reprogrammer le hook et non pas réécrire des portions entières du code initial.

Dans le cas présent, le hook sera nommé « `\positem@endprocess` ». Pour la macro `\positem`, se hook sera défini pour donner la position, préalablement calculée et stockée dans la macro `\pos@item`. Le hook sera donc défini par

```
\def\positem@endprocess{\pos@item}
```

Pour la macro `\positemtocs`, ce hook sera programmé pour rendre la macro à définir `\let`-égale à `\pos@item`.

Code n° III-315

```
\catcode`\@11
\def\quark@list{\quark@list}% quark de fin de liste
```

```
\def\positem#1{% #1 = \<macrolist>, la position est lue plus tard par \positem@i
  \def\positem@endprocess{\pos@item}% hook : afficher la position
  \exparg\positem@i{#1}% 1-développe la \<macrolist>
}
\def\positem@i#1#2{% #1 = liste   #2=élément cherché
  \def\sought@item{#2}% définir l'élément cherché
  \positem@ii{1}\relax#1,\quark@list,% appelle la macro récursive
}
\def\positem@ii#1#2,{% #1=position courante  #2=\relax + élément courant
  \expandafter\def\expandafter\current@item\expandafter{\gobone#2}%
  \ifx\current@item\quark@list\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\def\pos@item{0}% si la fin de liste est atteinte, renvoyer 0
    \positem@endprocess% et aller au hook
    }% sinon
    {\ifx\current@item\sought@item\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {\def\pos@item{#1}% si la position est la bonne, définir la position
      \positem@goblist% et manger les éléments restants
      }% si la position n'est pas la bonne, recommencer en incrémentant #1
      {\exparg\positem@ii{\number\numexpr#1+1}\relax
      }%
    }%
}
\def\positem@goblist#1\quark@list,{\positem@endprocess}% manger la liste et aller au hook

\def\positemtocs#1#2#3{% #1=\<macrolist>  #2=élément à chercher  #3=macro à définir
  \def\positem@endprocess{\let#3=\pos@item}% hook : mettre le résultat dans #3
  \exparg\positem@i{#1}{#2}% 1-développe la \<macrolist>
}

\catcode`\@12
\def\liste{a,bcd,{ef},g,hij,,kl}
a) \positem\liste{g}\qquad
b) \positem\liste{ef}\qquad
c) \positem\liste{{ef}}\qquad
d) \positem\liste{}\medbreak

\def\liste{a,bcd,{ef},g,hij,,kl}
a) \positemtocs\liste{g}\foo\meaning\foo\qquad
b) \positemtocs\liste{ef}\foo\meaning\foo\qquad
c) \positemtocs\liste{{ef}}\foo\meaning\foo\qquad
d) \positemtocs\liste{}\foo\meaning\foo\qquad
```

a) 4 b) 0 c) 3 d) 6

a) macro:->4 b) macro:->0 c) macro:->3 d) macro:->6

10.2.3. Insérer un élément dans une liste

Passons maintenant à la macro `\insitem` qui insère un élément dans une liste à une position spécifiée, position qui sera la position de l'élément dans la liste modifiée après l'insertion

`\insitem\`⟨*macrolist*⟩{⟨*position*⟩}{⟨*élément*⟩}

Si la ⟨*position*⟩ est inférieure ou égale à 1, l'élément devra être inséré à la première position. Si la ⟨*position*⟩ est strictement supérieure à la longueur de la liste, l'élément viendra en dernière position. Remarquons que rendre la macro `\insitem` purement développable serait un non-sens puisqu'elle n'a pas vocation à renvoyer

quelque chose, mais modifie la \⟨*macrolist*⟩ qui contient la liste d'éléments.

Nous allons procéder un peu comme avec \finditem, mais nous stockerons dans une macro temporaire \item@list les éléments mangés en y ajoutant l'élément à insérer lorsque la position voulue est atteinte. Par ailleurs, la macro récursive sera écrite comme une sous-macro de \insitem afin de pouvoir facilement accéder à ses arguments.

Code n° III-316

```
\catcode`\@11
\def\quark@list{\quark@list}% quark de fin de liste
\def\insitem#1#2#3{% #1 = macro   #2=position cherchée   #3=élément à insérer
  \let\item@list=\empty
  \ifnum#2<1 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    % si position < 1
    {\addtomacro\item@list{#3,}% ajouter l'élement à insérer en premier
    \eaddtomacro\item@list#1% puis la liste entière
    }
    % si la position  > 1
    {% définir la macro récursive
    \def\insitem@i##1##2,{% ##1 = position courante  ##2=\relax + élément courant
      \expandafter\ifx\expandafter\quark@list\gobone##2%
      \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
        {\addtomacro\item@list{#3}}% si fin de liste, ajouter l'élément en dernier
        {% sinon, si la position cherchée est atteinte
        \ifnum##1=#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
          {\addtomacro\item@list{#3,}% ajouter l'élement
          \add@remainlist##2,% et ##2 (en supprimant le \relax) et le reste
          }% si la position n'est pas atteinte
          {\eaddtomacro\item@list{\gobone##2,}% ajouter l'élément
          \exparg\insitem@i{\number\numexpr##1+1}\relax% et recommencer
          }%
        }%
      }%
    % appel de la macro récursive
    \expandafter\insitem@i\expandafter1\expandafter\relax#1,\quark@list,%
    }%
  \let#1=\item@list% rendre #1 égal au résultat
}

\def\add@remainlist#1,\quark@list,{%
  \eaddtomacro\item@list{\gobone#1}% ajouter #1 ainsi que les autres
}
\catcode`\@12
\def\liste{a,bra,{cA},da,brA}\insitem\liste3{XX}\meaning\liste\par
\def\liste{a,bra,{cA},da,brA}\insitem\liste0{XX}\meaning\liste\par
\def\liste{a,bra,{cA},da,brA}\insitem\liste9{XX}\meaning\liste
```

```
macro:->a,bra,XX,{cA},da,brA
macro:->XX,a,bra,{cA},da,brA
macro:->a,bra,{cA},da,brA,XX
```

10.2.4. Supprimer un élément

Pour supprimer un élément de la liste donné par son numéro avec \delitem*, nous allons agir comme ci-dessus avec la différence suivante : si la position est strictement inférieure à 1 ou strictement supérieure au numéro du dernier élément, ne rien faire.

Supprimer un élément va être un peu plus difficile qu'en insérer un. La difficulté vient du fait que lorsqu'on lit un élément, on lit aussi la virgule qui le suit puisqu'elle agit comme délimiteur de la macro récursive. Si l'élément à supprimer est le dernier de la liste, il restera donc à lire le quark et la virgule qui sont insérés à l'appel de la macro récursive. On ne peut donc pas définir la macro `\add@remainlist` avec un délimiteur « `,\quark@list,` » mais avec ce délimiteur-là : « `\quark@list,` ». Nous la nommerons `\add@reaminingitems`.

Code n° III-317

```
\catcode`\@11
\def\quark@list{\quark@list}% quark de fin de liste
\def\delitem#1#2{% #1 = macro   #2=position cherchée
  \ifnum#2>0 % ne faire quelque chose que si la position est >0
    \def\delitem@i##1##2,{% ##1 = position courante  ##2=\relax + élément courant
      \expandafter\ifx\expandafter\quark@list\gobone##2%
      \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
        {}% si fin de liste, ne rien faire
        {% sinon, si la position cherchée est atteinte
        \ifnum##1=#2 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
          {\add@reaminingitems% et ##2 (en supprimant le \relax) et le reste
          }% si la position n'est pas la bonne
          {\eaddtomacro\item@list{\gobone##2,}% ajouter l'élément
          \exparg\delitem@i{\number\numexpr##1+1}\relax% et recommencer
          }%
        }%
      }%
    \let\item@list=\empty% initialiser la macro tempporaire
    % appel de la macro récursive
    \expandafter\delitem@i\expandafter1\expandafter\relax#1,\quark@list,%
    \let#1=\item@list% rendre #1 égal au résultat
  \fi
}

\def\add@reaminingitems#1\quark@list,{%
  \eaddtomacro\item@list{#1}% ajouter tout jusqu'au quark
}
\catcode`\@12
\for\xx=0 to 8 \do{%
\def\liste{a,bcd,{ef},g,hij,kl}
position \xx{} : \expandafter\delitem\expandafter\liste\xx\meaning\liste.\par
}
```

```
position 0 : macro:->a,bcd,{ef},g,hij,kl.
position 1 : macro:->bcd,{ef},g,hij,kl,.
position 2 : macro:->a,{ef},g,hij,kl,.
position 3 : macro:->a,bcd,g,hij,kl,.
position 4 : macro:->a,bcd,{ef},hij,kl,.
position 5 : macro:->a,bcd,{ef},g,kl,.
position 6 : macro:->a,bcd,{ef},g,hij,.
position 7 : macro:->a,bcd,{ef},g,hij,kl,.
position 8 : macro:->a,bcd,{ef},g,hij,kl,.
```

Nous le constatons, mais il fallait s'y attendre : une virgule parasite se trouve à la fin de chaque liste traitée par la macro. Si l'élément à supprimer n'est pas le dernier, cette virgule provient de la ligne n° 20 où elle est insérée avant le quark et non mangée par la macro `\add@reaminingitems`. En revanche, si l'élément à supprimer est le dernier, cette virgule provient de la ligne n° 13 où elle a été insérée à l'itération précédente.

Dans tous les cas, il nous faut trouver un moyen de supprimer cette virgule. Le moyen le plus rapide est de définir une macro

```
\def\remove@lastcomma#1,\@nil{\def\item@list{#1}}
```

Il faut s'assurer que cette macro reçoit le texte de remplacement de `\item@list` (qui se finit par une virgule dont on veut se débarrasser) suivi d'un `\@nil`. Il faut donc insérer la ligne suivante juste après la ligne n° 20 de la macro `\delitem` :

```
\expandafter\remove@lastcomma\item@list\@nil
```

10.2.5. Déplacer un élément

Pour déplacer un item d'une position x à une position y, il faut déjà isoler cet item avec `\finditemtocs`, le supprimer avec `\delitem` et enfin l'insérer avec `\insitem` à la position y.

Code n° III-318

```
\catcode`\@11
\def\moveitem#1#2#3{% #1 = liste, #2=position départ, #3=position arrivée
  \ifnum#2>0 % ne faire quemque chose que si #2>0
    \finditemtocs#1{#2}\temp@item% sauvegarder l'élément
    \delitem#1{#2}% supprimer l'élément
    \expsecond{\insitem#1{#3}}\temp@item% insérer l'élément
  \fi
}
\catcode`\@12
% déplace "b" en 5e position
a) \def\liste{a,b,c,d,e,f} \moveitem\liste25 "\liste"\par
% déplace "d" en 1e position
b) \def\liste{a,b,c,d,e,f} \moveitem\liste41 "\liste"\par
% déplace "c" en 9e position
c) \def\liste{a,b,c,d,e,f} \moveitem\liste39 \liste"\par
% position départ=0 -> sans effet
d) \def\liste{a,b,c,d,e,f} \moveitem\liste02 "\liste"
```

a) "a,c,d,e,b,f"
b) "d,a,b,c,e,f"
c) a,b,d,e,f,c"
d) "a,b,c,d,e,f"

10.2.6. Exécuter une liste

La dernière chose qu'il nous reste à faire est d'exécuter une liste et pour ce faire, programmer une macro `\runlist*`. Pour cela, chacun des éléments de la liste sera placé dans l'argument d'une `\⟨macro⟩`. Donnons-nous cette syntaxe :

```
\runlist{\⟨macrolist⟩}\with\⟨macro⟩
```

Tous les éléments de la ⟨*liste*⟩ seront placés tour à tour dans l'argument d'une `\⟨macro⟩` acceptant un seul argument, et tout se passera comme si nous avions écrit

```
\⟨macro⟩{⟨élément 1⟩}\⟨macro⟩{⟨élément 2⟩}...\⟨macro⟩{⟨élément n⟩}
```

et que ce code soit exécuté par TEX.

La méthode consiste à manger chaque élément, l'écrire comme argument de la `\⟨macro⟩` et s'arrêter lorsque la fin de la liste est atteinte. Il va donc falloir stoker dans une macro auxiliaire nommée `\collect@run` les `\⟨macro⟩{⟨élément i⟩}`

au fur et à mesure que les éléments seront rencontrés et exécuter cette macro à la toute fin du processus. Pour être tout à fait propre en programmation, nous allons effectuer la collecte dans un groupe semi-simple de telle sorte que la macro `\collect@run` soit locale à ce groupe et n'existe plus ensuite, étant entendu que `\collect@run` doit être exécutée *hors* du groupe semi-simple.

Code n° III-319

```
1  \catcode`\@11
2  \def\runlist#1\with#2{% #1=liste #2=macro
3    \def\runlist@i##1,{%
4      \ifx\quark@list##1\relax\else% si la fin n'est pas atteinte
5        \addtomacro\collect@run{#2{##1}}% ajouter "\<macro>{<élément>}""
6        \expandafter\runlist@i% et recommencer en lisant l'élément suivant
7      \fi
8    }%
9    \begingroup% fait la collecte dans un groupe
10     \let\collect@run=\empty% initialiser la macro
11     \expandafter\runlist@i#1,\quark@list,% appeler \runlist@i
12   \expandafter\endgroup% ferme le groupe et détruit \collect@run
13   \collect@run% après l'avoir développé !
14 }
15 \catcode`\@12
16 \newcount\foocnt
17 \foocnt=0 % compteur utilisé pour numéroter les éléments dans la macro \foo
18 \def\foo#1{% la macro qui va exécuter chaque élément de la liste. #1 = l'élément
19   \advance\foocnt1 % incrémente le compteur
20   L'argument \number\foocnt\ est : {\bf #1}\par%
21 }
22 \def\liste{a,bra,ca,da,BRA}%
23 \runlist\liste\with\foo%
```

L'argument 1 est : **a**
L'argument 2 est : **bra**
L'argument 3 est : **ca**
L'argument 4 est : **da**
L'argument 5 est : **BRA**

Remarquons l'astuce à la ligne n° 12 qui consiste à « sauter » le `\endgroup` à l'aide d'un un `\expandafter` pour 1-développer la macro `\collect@run` avant que le `\endgroup` ne la détruise ! C'est une astuce assez commune que de sauter une fin de groupe avec un ou plusieurs `\expandafter` pour gagner le territoire se trouvant à l'extérieur du groupe tout en restant imprégné de ses propriétés. Cette méthode permet donc à peu de frais [2] d'exporter au dehors de son territoire les propriétés d'un groupe avant sa fermeture.

Ici, nous aurions pu procéder autrement et initialiser `\collet@run` de cette façon à la ligne n° 10

```
\def\collect@run{\endgroup}
```

et écrire en lieu et place des lignes n^os^ 12-13 un simple `\collect@run` qui, en se développant, aurait fait apparaitre le `\endgroup` au début de son texte de remplacement qui aurait provoqué sa destruction.

2. Tout au plus quelques `\expandafter` avant le `\endgroup` éventuellement relayés à l'extérieur du groupe par d'autres `\expandafter` ou tout autre procédé permettant de propager le développement jusqu'au lieu où l'on veut exporter les propriétés du groupe.

10.2.7. Enlever les espaces extrêmes

Il nous reste un dernier pas à accomplir, celui de se jouer des espaces laissés avant ou après les éléments d'une liste. En effet, il serait commode de saisir

```
\def\liste{ a, b c , def,   g h, ij }
```

et de disposer d'une macro `\sanitizelist*\liste` qui rendrait la liste égale à

```
a,b c,def,g h,ij
```

Cette macro `\sanitizelist` supposerait qu'il existe une macro capable de retirer *tous* les espaces qui se trouvent au début ou à la fin d'un argument, sans évidemment toucher aux espaces « intérieurs ».

Supprimer les espaces du début

La mise au point d'une macro `\removefirstspaces*` qui retire tous les espaces qui se trouvent au *début* de son argument demande tout d'abord de programmer un test `\ifspacefirst*{`⟨*texte*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}` qui est vrai si son premier argument commence par un espace. Pour éviter d'envisager le cas d'un ⟨*texte*⟩ vide, la lettre « W » sera ajoutée à la fin de cet argument.

Code n° III-320

```
\catcode`\@11
\def\ifspacefirst#1{%
  \expandafter\ifspacefirst@i\detokenize{#1W} \@nil% "W" se prémunit d'un argument vide
}
\def\ifspacefirst@i#1 #2\@nil{\ifempty{#1}}% renvoyer vrai s'il n'y a rien avant " "
\catcode`\@12
a) \ifspacefirst{a bc d}{vrai}{faux}\qquad
b) \ifspacefirst{ a bc d}{vrai}{faux}\qquad
c) \ifspacefirst{ }{vrai}{faux}\qquad
d) \ifspacefirst{}{vrai}{faux}\qquad
e) \ifspacefirst{{ } }{vrai}{faux}\qquad
f) \ifspacefirst{ {x} }{vrai}{faux}
```

a) faux b) vrai c) vrai d) faux e) faux f) vrai

Le côté pratique de `\detokenize` est qu'il permet des accolades dans le ⟨*texte*⟩ malgré le fait que ce ⟨*texte*⟩ soit l'argument d'une macro à argument délimité.

La méthode mise en œuvre sera simple : pour supprimer tous les espaces au début d'un argument, il faut tester s'il commence par un espace et dans l'affirmative, manger cet espace avec une macro `\gobspace*` puis recommencer avec ce qui a été obtenu. La macro `\gobspace` attendra donc un espace juste après elle et sera donc délimitée par un espace. Pour la définir, il n'est pas possible d'écrire

```
\def\gobspace {}
```

car dans le code source, un espace qui suit une séquence de contrôle est ignoré (voir règle page 42). Pour contourner cette règle, nous allons mettre en place un pont d'`\expandafter` pour 1-développer `\space` en ␣ avant que `\def` n'entre en action :

Code n° III-321

```
\catcode`\@11
\expandafter\def\expandafter\gobspace\space{}
```

```
\def\removefirstspaces#1{%
  \ifspacefirst{#1}% si #1 commence par un espace
    {\exparg\removefirstspaces{\gobspace#1}}% recommencer sans le 1er espace
    {#1}% sinon, renvoyer l'argument
}
\catcode`\@12
a) "\removefirstspaces{12 {\bf3}4 567}"\qquad
b) "\removefirstspaces{ 12 {\bf3}4 567}"\qquad
c) \edef\foo{\space\space\space 12 34 567}
   "\exparg\removefirstspaces\foo"
```

a) "12 **3**4 567" b) "12 **3**4 567" c) "12 34 567"

Le dernier cas permet de vérifier que la récursivité fonctionne lorsque le texte de remplacement de `\foo` commence par *trois* espaces, chacun issu du développement de `\space`.

Essayons d'aller un peu plus loin et faisons en sorte que cette macro se développe en un nombre connu de fois. Nous allons recourir à un `\romannumeral`, placé au tout début du texte de remplacement de `\removefirstspaces` afin de lancer le développement maximal. Il faudra le stopper en écrivant `{\z@#1}` à la ligne n° 7. On s'offre donc une macro qui effectue son travail en deux développements : le premier pour extraire son texte de remplacement et le deuxième qui lance celui de `\romannumeral`.

Code n° III-322

```
\catcode`\@11
\def\removefirstspaces{%
  \romannumeral% lance le développement maximal
  \removefirstspaces@i% et passe la main à la macro récursive
}

\def\removefirstspaces@i#1{%
  \ifspacefirst{#1}% si #1 commence par un espace
    {\exparg\removefirstspaces@i{\gobspace#1}}% recommencer sans le 1er espace
    {\z@#1}% sinon, renvoyer l'argument où \z@ stoppe l'action de \romannumeral
}
\catcode`\@12
\long\def\>#1<{"\detokenize{#1}"}

a) \expandafter\expandafter\expandafter\>\removefirstspaces{12 {\bf3}4 567}<\qquad
b) \expandafter\expandafter\expandafter\>\removefirstspaces{ 12 {\bf3}4 567}<\qquad
c) "\removefirstspaces{ 12 {\bf3}4 }"
```

a) "12 {\bf 3}4 567" b) "12 {\bf 3}4 567" c) "12 **3**4 "

Supprimer les espaces de fin

Programmons maintenant la macro `\removelastspaces*{`⟨*texte*⟩`}` qui supprime les espaces qui se trouvent à la fin de son argument. Bien que l'algorithme soit un peu plus compliqué, nous allons *aussi* utiliser des arguments délimités.

Pour décrire la méthode, prenons « `W` » comme délimiteur en supposant pour l'instant que cette lettre ne figure pas dans l'argument. L'idée se décompose en plusieurs étapes :

1. ajouter « `W␣W` » à la fin de l'argument ;
2. du résultat obtenu, prendre ce qui se trouve avant le premier « `␣W` » ;

3. ajouter « W » à la fin du résultat obtenu ;
4. du résultat précédent, prendre ce qui se trouve avant le premier « W ».

Énuméré comme cela, il est quasi impossible de se représenter les différentes étapes et comprendre comment évolue l'argument. Aussi, nous allons prendre deux exemples pour décrire les deux cas de figure pouvant se présenter, « a » qui ne finit pas par un espace et « b␣ » qui se trouve dans le cas opposé :

n° d'étape	decription	a	b␣
1	ajout de « W␣W »	aW␣W	b␣W␣W
2	ce qui est avant « ␣W »	aW	b
3	ajout de « W »	aWW	bW
4	ce qui est avant « W »	a	b

Le cheminement est désormais plus clair et il est plus facile de se persuader que la méthode fonctionne quels que soient « a » et « b ».

La méthode présente cependant un petit défaut ; elle indique ce qui se trouve *avant* les délimiteurs, mais ne dit pas un mot de ce qu'elle laisse *après*. Nous allons donc aussi récolter les reliquats qui sont laissés après les délimiteurs.

Imaginons qu'au lieu de W␣W, nous ajoutions en fin d'argument « W␣W\@nil » lors de la première étape et que l'on fasse fonctionner les étapes déjà décrites. Le fait nouveau sera qu'à la fin, on récolte les reliquats se trouvant avant le \@nil. Qu'y trouverait-on ? Tout ce qui a été laissé après les délimiteurs des lignes n° 2 et 4. Le tableau est repris sauf qu'ici, les lignes grises montrent ce qui est laissé après ces délimiteurs :

n° d'étape	description	a	b␣
1	ajout de « W␣W »	aW␣W	b␣W␣W
2	ce qui est avant « ␣W »	aW	b
	ce qui est après « ␣W »	⟨*vide*⟩	␣W
3	ajout de « W »	aWW	bW
4	ce qui est avant « W »	a	b
	ce qui est après « W »	W	⟨*vide*⟩

Avant le \@nil se trouve la concaténation des reliquats partiels, c'est-à-dire :

– « W » dans le cas « a » ;
– « ␣W » dans le cas « b␣ ».

Il est intéressant de remarquer que si le reliquat *commence* par un espace, cela signifie que l'argument se finissait par un espace. Dans le cas présent, puisque nous souhaitons supprimer *tous* les espaces à la fin de l'argument, il va donc falloir mettre en place une récursivité et recommencer dans le cas où le reliquat commence par un espace.

Réglons enfin le délimiteur « W » qui nous a servi pour exposer la méthode, mais qui ne peut être utilisé dans les vraies conditions, car trop probable dans un vrai argument. Remplaçons-le par un caractère bien plus improbable mais pas trop « exotique », par exemple le caractère de code 0 (qui s'écrit ^^00) et de catcode 12 [3].

Code n° III-323

```
\catcode`\@11
\edef\catcodezero@saved{\number\catcode0 }% stocke le catcode de ^^00
\catcode0=12 % le modifie à 12
\def\removelastspaces#1{%
```

3. Rien n'empêcherait de lui assigner un catcode de 3 pour le rendre plus exotique.

```
5   \romannumeral% lance le développement maximal
6   \removelastspaces@i\relax#1^^00 ^^00\@nil% mettre un \relax au début
7                                           % et passer la main à \removelastspaces@i
8 }
9 \def\removelastspaces@i#1 ^^00{% prendre ce qui est avant " W"
10  \removelastspaces@ii#1^^00% ajouter "W"
11 }
12 \def\removelastspaces@ii#1^^00#2\@nil{% #1=ce qui est avant "W"  #2=reliquat
13  \ifspacefirst{#2}% si le reliquat commence par un espace
14    {\removelastspaces@i#1^^00 ^^00\@nil}% recommencer sans passer par \removelastspaces
15    {\expandafter\z@\gobone#1}% sinon supprimer le \relax ajouté au début
16                              % et stopper l'action de \romannumeral avec \z@
17 }
18 \catcode0=\catcodezero@saved\relax% restaure le catcode de ^^00
19 \catcode`\@12
20 \long\def\>#1<{"\detokenize{#1}"}
21
22 a) \expandafter\expandafter\expandafter\>\removelastspaces{ 12 {\bf3}4 }<\qquad
23 b) \expandafter\expandafter\expandafter\>\removelastspaces{12 {\bf3}4}<\qquad
24 c) "\removelastspaces{ 12 {\bf3}4 }"
```

a) " 12 {\bf 3}4" b) "12 {\bf 3}4" c) " 12 **3**4"

La première remarque concerne le `\relax` mis avant `#1` tout au début afin d'éviter que `#1` ne soit dépouillé de ses accolades s'il s'agit d'un texte entre accolades. Ce `\relax` est retiré juste avant que `\z@` ne stoppe l'action de `\romannumeral`.

L'autre remarque, essentielle, est que l'appel récursif de la ligne n° 6 n'appelle pas la macro chapeau `\removelastspaces`. Cela évite de lancer à chaque itération un nouveau `\romannumeral` qui ne serait pas stoppé par la suite et provoquerait des erreurs de compilation.

Supprimer les espaces extrêmes

Tout est en place pour construire une macro `\removetrailspaces*`, chargée de supprimer les espaces en trop se trouvant au début et à la fin de son argument. Il suffit d'assembler correctement les macros vues précédemment. Cet assemblage mérite une explication… Si `#1` est l'argument de cette macro, on voudrait pouvoir écrire

`\removelastspaces{\removefirstspaces{`⟨*sp*⟩`#1`⟨*sp*⟩`}}`

où ⟨*sp*⟩ représente d'éventuels espaces extrêmes de `#1`. Mais n'oublions pas que TEX prend les arguments tels quels, sans les remplacer par ce qu'ils sont après développement.

Pour que `\removefirstspaces{`⟨*sp*⟩`#1`⟨*sp*⟩`}` soit développé, nous allons d'abord lancer le développement maximal avec `\romannumeral` en tout début de texte de remplacement de `\removetrailspaces`. Comme précédemment, cet artifice nous assurera que cette macro donne son résultat en deux développements.

Dans le code ci-dessus, convenons que « • » représente un `\expandafter`. Le pont d'`\expandafter`, lui-même développé par `\romannumeral`, va 2-développer « `\removefirstspaces{`⟨*sp*⟩`#1`⟨*sp*⟩`}` » :

`\romannumeral` •••`\removelastspaces` •••`{\removefirstspaces{`⟨*sp*⟩`#1`⟨*sp*⟩`}}`

Après que les `\expandafter` aient joué leur rôle, la macro `\removefirstspaces` a

effectué son travail et on obtient :

\removelastspaces{#1⟨*sp*⟩}

Enfin, comme \romannumeral n'a toujours pas trouve de nombre, le développement se poursuit, la macro \removelastspaces se 2-développe et nous obtenons :

#1

Mais à ce stade, \romannumeral est toujours en action et nous n'avons pas pensé à la stopper. Si #1 commence par un nombre au sens de TEX, ce nombre sera capturé par \romannumeral pour former les chiffres romains et sinon, TEX se plaindra d'une erreur de compilation du type « Missing number ».

Pour régler ce problème, il faut revenir en arrière et réfléchir où mettre \z@ pour qu'à la dernière étape ci-dessus, le #1 soit précédé de \z@. Le bon endroit est juste avant \removefirstspaces, mais surtout pas avant #1, car ce \z@ cacherait les éventuels espaces au début de #1. Remarquons au passage que ce token supplémentaire « \z@ » implique la prolongation du pont d'\expandafter. Voici les étapes du développement jusqu'au résultat final :

\romannumeral •••\removelastspaces •••{•••\z@\removefirstspaces{⟨*sp*⟩#1⟨*sp*⟩}}

\removelastspaces{\z@#1⟨*sp*⟩}

\z@#1

#1

Cette méthode donne le code suivant :

Code n° III-324

```
\catcode`\@11
\def\removetrailspaces#1{%
  \romannumeral% lance le développement maximal
    \expandafter\expandafter\expandafter% le pont d'\expandafter
  \removelastspaces
    \expandafter\expandafter\expandafter% fait agir \removefirstspaces en premier
  {%
    \expandafter\expandafter\expandafter
  \z@% stoppe le développement initié par \romannumeral
  \removefirstspaces{#1}%
  }%
}
\catcode`\@12
\long\def\>#1<{"\detokenize{#1}"}

a) \expandafter\expandafter\expandafter\>\removetrailspaces{ 12 {\bf3}4 }<\qquad
b) \expandafter\expandafter\expandafter\>\removetrailspaces{12 {\bf3}4}<\par
c) \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter
   \foo\expandafter\expandafter\expandafter{\removetrailspaces{ 12 {\bf3}4 }}%
   signification : \meaning\foo.\par
c) exécution : "\foo"\par
d) "\removetrailspaces{ 12 {\bf3}4 }"
```

a) "12 {\bf 3}4" b) "12 {\bf 3}4"
c) signification : macro:->12 {\bf 3}4.
c) exécution : "12 **3**4"
d) "12 **3**4"

La macro \sanitizelist

Le plus dur est fait ! Nous allons maintenant pouvoir finir en douceur en écrivant la macro \sanitizelist qui supprime tous les espaces extrêmes des éléments d'une liste. Le confort dont nous profitons ici est que \removetrailspaces ne demande que deux développements pour donner son résultat.

Au fur et à mesure de la lecture de la liste, nous allons collecter dans la macro \item@list les éléments purgés de leurs espaces indésirables. Ici encore, nous placerons un \relax avant chaque élément (argument #1), sous réserve de le supprimer ensuite avec \gobone. L'ajout de l'élément en cours #1 au collecteur \item@list se fera donc par :

```
\addtomacro\item@list{\removetrailspaces{\gobone#1},}
```

sauf qu'il faut développer les arguments avant que les macros ne s'en emparent. Ici, \gobone doit être 1-développé en premier puis \removetrailspaces doit être 2-développé. Cela va être le festival des \expandafter, ou plutôt l'invasion ! Un \expandafter est représenté par • ci-dessous.

```
•••••••\addtomacro•••••••\item@list
•••••••{•\removetrailspaces•{\gobone#1},}
```

Code n° III-325

```
\catcode`\@11
\def\sanitizelist#1{% #1 = liste
  \let\item@list\empty% initialise le réceptacle de la liste assainie
  \def\sanitizelist@endprocess{% définit le hook de fin
    \expandafter\remove@lastcomma\item@list\@nil% supprimer dernière virgule
    \let#1=\item@list% et assigner le résultat à #1
  }%
  \expandafter\sanitizelist@i\expandafter\relax#1,\quark@list,% aller à la macro récursive
}

\def\sanitizelist@i#1,{% #1="\relax" + élément courant
  \expandafter\ifx\expandafter\quark@list\gobone#1%
  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\sanitizelist@endprocess% si fin de liste, hook de fin
    }% si la fin de la liste n'est pas atteinte :
    {\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
      \addtomacro% 1-développer \gobone pour retirer \relax
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
      \item@list% puis 2-développer \removetrailspaces
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
      {\expandafter\removetrailspaces\expandafter{\gobone#1},}% "#1"
    \sanitizelist@i\relax% et continuer avec l'élément suivant
    }%
}

\catcode`\@12
\frboxsep=0pt % encadrer au plus proche
\def\foo#1{\frbox{\tt\strut#1} }% boite avec un strut et en fonte "tt" puis espace
\def\liste{ Programmer, en \TeX{} ,est   ,{\bf facile}, et utile }
a) \runlist\liste\with\foo% encadrer les items

b) \sanitizelist\liste% supprimer les espaces inutiles des items
   \runlist\liste\with\foo
```

a) Programmer | en TEX | est | **facile** | et utile

b) Programmer en TEX est **facile** et utile

Il est toujours possible d'éviter ces fastidieux ponts d'`\expandafter`. Nous aurions pu mettre à contribution nos macros `\expsecond` et `\eaddtomacro` pour alléger le code. Ces `\expandafter` ont été laissés en l'état pour montrer que la gestion du développement est parfois assez pénible.

★
★ ★

Après avoir compris dans cette partie comment construire des macros récursives et donc, comment s'y prendre pour bâtir des boucles, nous allons poursuivre notre chemin.

Il s'agit désormais de lire non pas des arguments comme le font les macros, mais descendre à un niveau inférieur, celui des tokens. Pour prendre une analogie chimique, les arguments seraient les molécules tandis que les tokens seraient les atomes ; l'un étant fait avec l'autre. Il nous faut apprendre à programmer des macros qui lisent des tokens, un peu comme le fait TEX lorsqu'il lit du code, mais en gardant la main après chaque token de façon à être capable de *contrôler* chaque token lu et d'agir en conséquence.

Quatrième partie

Au niveau des tokens

Sommaire

MAINTENANT que les arguments des macros n'ont plus de secrets, il faut apprendre à intervenir à un niveau plus bas, celui des tokens. En effet, pour certains besoins spécifiques, la lecture d'arguments par des macros est trop grossière et ne permet pas de les résoudre de façon satisfaisante.

Chapitre 1

Mise en évidence du problème

Supposons que nous souhaitions effectuer un traitement sur chaque caractère d'une phrase. Pour rendre les choses visuelles, choisissons d'encadrer chaque caractère. L'idée est de lire chaque caractère de la phrase par une macro à argument puis d'encadrer ce caractère à l'aide de la macro `\frbox` vue dans la partie précédente. La récursivité sera classique, une macro chapeau `\boxsentence*` se chargera de transmettre la phrase suivie d'un quark à une macro récursive. La macro récursive `\boxsentence@i` lira donc un argument, le stockera dans une macro brouillon pour le comparer au quark et tant que l'égalité n'aura pas lieu, bouclera sur elle-même pour lire les arguments les uns après les autres.

Code n° IV-326

```
\catcode`\@11
\def\boxsentence#1{%
  \leavevmode% se mettre en mode horizontal
  \boxsentence@i#1\quark% transmet "#1+\quark" à boxsentence@i
}
\def\boxsentence@i#1{% #1= argument lu
  \def\current@arg{#1}% stocke l'argument dans une macro temporaire
  \unless\ifx\quark\current@arg% si la fin n'est  pas atteinte
    \frbox{#1}% encadrer cet argument
    \expandafter\boxsentence@i% lire l'argument suivant
  \fi
}
\catcode`@12
\frboxsep=1pt \frboxrule=0.2pt
\boxsentence{Programmer en \TeX\ est facile}
```

Programmeren TEX est facile

Les espaces sont ignorés puisque la règle dit que lorsque les espaces ne sont pas enveloppés entre accolades, ils sont ignorés lorsque lus par une macro en tant qu'arguments. Si l'on avait voulu encadrer les espaces de cette phrase, il aurait fallu les écrire entre accolades ou bien utiliser la primitive `\ ` ou utiliser la macro `\space`. Ce sont des artifices que l'on n'utilise jamais dans une phrase tapée naturellement, sauf éventuellement après une séquence de contrôle, comme ici après `\TeX`.

Voici un autre exemple. On pourrait aussi vouloir écrire une macro et pouvoir l'appeler avec une *variante*. Le moyen le plus communément utilisé pour signifier que l'on souhaite la variante est d'écrire une étoile juste après le nom de la macro. À charge pour la macro de tester si l'étoile est présente ou pas et d'adopter deux comportements différents selon le cas. En l'état actuel de nos connaissances, nous sommes obligés de lire l'argument qui suit afin de le tester pour déterminer si cet argument est une étoile.

Appelons `\expo*` une macro sans argument qui, appelée seule, produit l'exposant « † » et, appelée étoilée (`\expo*`), affiche l'exposant « ‡ ». Le signe « † » s'obtient avec la macro `\dag` tandis que « ‡ » s'obtient avec `\ddag`, ces macros devant opérer en mode mathématique.

Nous allons donc définir `\expo` comme une macro admettant un argument :

- si c'est une étoile, afficher « `$^\dag$` » ;
- si ce n'est pas une étoile, afficher « `$^\ddag$` » et *réécrire l'argument lu.*

Code nº IV-327

```
\catcode`\@11
\edef\star@macro{\string *}% stocke une étoile de catcode 12
\def\expo#1{%
  \def\temp@arg{#1}% stocke l'argument lu
  \ifx\star@macro\temp@arg\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {% si l'argument est une étoile de catcode 12
    $^\dag$% affiche une dague
    }% sinon
    {$^\ddag$% affiche une double dague
    {#1}% puis ré-écrit l'argument {#1}
    }%
}
A\expo B\expo*C
```

A‡B†C

Bien que cela *semble* fonctionner, on va tout droit à la catastrophe si on écrit :

```
A\expo \def\foo{BAR}
```

En effet, à la ligne nº 10, en réécrivant l'argument `#1`, *on le met entre accolades.* Et donc ici, `\expo` prendrait `\def` comme argument. Ce `\def` se retrouverait *seul* entre accolades ce qui, lors de son exécution, déclencherait une erreur du type « `Missing control sequence.` ». Rappelons-nous que `\def` *doit* être suivie d'une séquence de contrôle ou d'un caractère actif. On pourrait penser s'en sortir en réécrivant `#1` à la ligne nº 10, c'est-à-dire sans entre accolades. Mais avec cette modification, si on écrit

```
A\expo {⟨code dans le groupe⟩}
```

alors, le ⟨*code dans le groupe*⟩ ne sera justement plus dans un groupe puisque dépouillé des accolades par la macro \expo lorsqu'elle le réécrit. Avec les conséquences que l'on sait : toutes les assignations que l'utilisateur pensait confinées dans le groupe seraient désormais de portée globale !

Toute la difficulté vient du fait qu'une macro qui lit un argument ne *peut pas savoir* si cet argument est enveloppé dans des accolades ou pas, notamment lorsque l'argument est un token unique. Ainsi, que l'on écrive « `\foo A` » ou « `\foo{A}` », la macro `\foo` lit dans les deux cas l'argument « `A` » et ne peut pas déterminer s'il était entre accolades ou pas.

Pour résumer, la lecture d'arguments est défectueuse sur deux points essentiels :

- les espaces ne sont pas vus comme des arguments et sont ignorés (sauf si l'utilisateur les a délibérément mis entre accolades) ;
- il est impossible pour une macro de faire la différence entre l'argument « `{x}` » et l'argument « x » où x est un token quelconque (de catcode civilisé tout de même) autre qu'un espace.

Ces deux défauts rendent bien trop imprécise la lecture d'un texte quelconque *argument par argument* en vue d'effectuer une action pour chaque argument lu. Heureusement, on peut lire du code *token par token*. Voyons comment…

Chapitre 2

Lire du code token par token

Un petit rappel s'impose concernant la primitive \let vue à partir de la page 45. Nous savons qu'elle agit au niveau des tokens. Sa syntaxe est la suivante :

\let\⟨*macro*⟩= ⟨*token*⟩

Et à la suite de cette assignation, la \⟨*macro*⟩ (qui peut être un caractère actif) devient un alias pour le ⟨*token*⟩. Rappelons également que le test \ifx se fait modulo la \let-égalité.

2.1. Reprendre la main après un \let

Comment construire une macro \readtok* qui lit le ⟨*token*⟩ qui la suit et qui l'assigne avec \let à \nxttok ?

La chose n'a rien de compliqué, il suffit de commencer l'assignation avec \let à l'intérieur du texte de remplacement de la macro pour que le ⟨*token*⟩ soit lu à l'extérieur. Il est important d'écrire la syntaxe *complète* de \let dans le texte de remplacement, c'est-à-dire faire suivre \let\nxttok de « =␣ ». Si par exemple, le ⟨*token*⟩ est « = », cette manœuvre nous assure que ce signe = n'est pas compris comme faisant partie prenante de la syntaxe de \let et soit perdu pour l'assignation. Voici comment on pourrait programmer la macro \readtok :

```
\def\readtok{\let\nxttok= }
```

En procédant de cette façon, même si la macro est suivie d'un signe égal comme dans « \readtok= », ce signe égal sera bien assigné à \nxttok. Le problème est cependant évident : la macro lit bien un ⟨*token*⟩ et l'assigne à \nxttok, mais *c'est tout* ! Après cette action, la macro s'achève. Pour reprendre la main après l'assi-

gnation afin d'appeler une autre macro \cmptok chargée d'analyser le ⟨*token*⟩ lu, nous sommes bloqués. En effet, si nous voulons appeler une macro \cmptok après l'assignation, il serait absurde d'écrire

```
\def\readtok{\let\nxttok= \cmptok}
```

car cela reviendrait à rendre \nxttok égal à \cmptok et pour le coup, la macro \readtok ne lirait rien du tout. On peut tourner le problème dans tous les sens, « \let\nxttok=␣ » *doit* se trouver à la toute fin du texte de remplacement de la macro.

2.1.1. La primitive \afterassignment

Ce cas de figure a été prévu et la primitive \afterassignment permet de s'en sortir.

89 - Règle

La primitive \afterassignment dont la syntaxe est

\afterassignment⟨*token*⟩

stocke le ⟨*token*⟩ dans un endroit spécial de la mémoire de TEX pour le placer sur la pile d'entrée immédiatement après la prochaine assignation. Cette assignation peut résulter de l'utilisation de primitives déjà vues \def, \gdef, \edef, \xdef, \let, \chardef, \mathchardef, ou bien d'une assignation à un registre (de boite, d'entier, de ressort, de dimension, de token), voire après une opération sur un registre d'entier ou de dimension par \advance, \multiply ou \divide.

Si plusieurs \afterassignment⟨*token*⟩ sont rencontrés avant une assignation, seul le dernier ⟨*token*⟩ est pris en compte.

Lorsque \afterassignment⟨*token*⟩ est appelé avant \setbox (c'est-à-dire avant l'assignation à un registre de boite), le ⟨*token*⟩ est placé en tout début de boite.

Tout ceci nous conduit donc à programmer la macro \readtok ainsi :

```
\def\readtok{\afterassignment\cmptok\let\nxttok= }
```

Dès lors, on peut programmer une macro \boxsentence qui va encadrer chaque token d'une phrase. Il suffira pour la macro \cmptok, appelée après chaque assignation, d'encadrer le token lu (dont l'alias est \nxttok) et de stopper la récursivité si elle lit un token spécial (nous prendrons ici \quark) que nous mettrons à la fin de l'argument de \boxsentence.

Code n° IV-328

```
\def\boxsentence#1{%
  \readtok#1\quark% met le token d'arrêt \quark à la fin de #1
}
\def\readtok{\afterassignment\cmptok\let\nxttok= }
\def\cmptok{%
  \unless\ifx\nxttok\quark% si la fin n'est pas atteinte
    \frbox{\nxttok}%  encadrer le token lu
    \expandafter\readtok% puis aller lire le suivant
  \fi
}
```

```
\frboxsep=1pt \frboxrule=0.2pt
\leavevmode\boxsentence{Programmer en \TeX\ est facile}
```

P r o g r a m m e r e n TEX e s t f a c i l e

Nous avons déjà levé un défaut qui existait avec la lecture d'arguments : les espaces ne sont pas ignorés lorsque lus après \let\⟨*macro*⟩=␣.

2.1.2. Application : retour sur la permutation circulaire des voyelles

Reprenons l'exercice de la deuxième partie où l'on effectuait une permutation circulaire sur les voyelles (voir page 56). Appelons \permutstart* la macro qui marque le début de la zone où cette permutation sera faite et \permutend* la macro qui marque la fin de cette zone.

Pour mener à bien cette permutation, il nous suffit de reprendre l'exemple précédent et tester dans \cmptok si le token lu est successivement chacune des voyelles et dans chaque cas, afficher la voyelle suivante. La macro \permutend sera un quark :

Code n° IV-329

```
\def\permutstart{\afterassignment\cmptok\let\nxttok= }
\def\permutend{\permutend}%
\def\cmptok{%
  \unless\ifx\permutend\nxttok% tant que la fin n'est pas atteinte :
    \ifxcase\nxttok
    ae% si le token lu est "a", afficher un "e"
    ei io ou uy ya% etc pour les autres lettres
    \elseif
      \nxttok% si ce n'est aucune voyelle, afficher le token
    \endif
    \expandafter\permutstart% aller lire le token suivant
  \fi
}
\permutstart Un "a" puis "e puis "i" ensuite, un "o", un "u" et "y".\permutend
```

Un "e" pyos "i pyos "o" insyoti, yn "u", yn "y" it "a".

Cela semble fonctionner comme nous l'attendons, mais ce n'est qu'une impression. L'algorithme présente un défaut majeur : à la ligne n° 9, il met le token lu tel quel dans le flux de lecture de TEX. Tout se passe bien tant qu'il ne s'agit que de tokens *indépendants* et n'ayant aucun besoin les uns des autres. Si l'on avait écrit « \'e », le token « \' » aurait été lu puis exécuté à la ligne n° 9. Comme ce token est une macro qui va lire l'argument suivant pour lui mettre un accent dessus, elle aurait donc capturé le premier \fi de la ligne n° 10 avec deux conséquences aussi graves l'une que l'autre :

- ce \fi capturé par \' n'aurait pas été disponible pour le \ifx apparié et donc, l'équilibrage entre les \ifx et les \fi aurait été rompu en faveur des \ifx qui auraient été excédentaires ;
- l'argument de \' ayant été \fi au lieu d'une simple lettre comme elle l'attend. Il est quasi certain que cet argument aurait provoqué une erreur de compilation puisque l'argument \fi est un argument « explosif » dès lors qu'il se retrouve dans un test (ce qui est le cas pour la macro \' avec le format LATEX.).

Comme on l'a déjà vu à l'exercice qui mettait les premières lettres de chaque mot en majuscule (lire page 198), il faudrait accumuler dans une variable les tokens au fur et à mesure de leur lecture. Il suffirait ensuite d'appeler cette variable en toute fin du processus. Hélas, une fois que \nxttok a été rendu \let-égal au token lu, on ne peut pas [1] savoir *a posteriori* à quel token il a été rendu \let-égal. Certes, on peut le tester, le faire lire par TEX pour l'afficher, mais on ne peut pas savoir ce à quoi il est égal ! L'affectation avec \let est à sens unique… C'est toute la différence entre une *macro* dont le texte de remplacement est facilement accessible (il suffit de la développer) et entre une séquence de contrôle assignée avec \let qui cache soigneusement ce à quoi elle est \let-égale. Impossible donc de faire comme avec une macro, de « développer » la séquence de contrôle \nxttok pour l'ajouter à une variable qui collecterait les tokens lus. Il faudra attendre le chapitre suivant pour apprendre comment, avec l'aide d'une autre primitive, procéder différemment.

2.1.3. Lecture d'accolades explicites

Essayons maintenant de corser la difficulté et de parcourir avec \let du code qui contient des accolades explicites (c'est-à-dire des tokens de catcode 1 et 2). Faisons l'expérience avec le code vu au-dessus, où chaque token est enfermé dans une boite encadrée par la macro \frbox programmée dans la partie précédente :

Code nº IV-330

```
\def\boxsentence#1{\readtok#1\boxsentence}
\def\readtok{\afterassignment\cmptok\let\nxttok= }
\def\cmptok{%
  %\show\nxttok% à décommenter pour débogage
  \unless\ifx\nxttok\boxsentence
    \frbox{\nxttok}%
    \expandafter\readtok
  \fi
}
\frboxsep=1pt \frboxrule=0.2pt
\leavevmode\boxsentence{Pro{gra}mmer en \TeX\ est facile}
```

```
! You can't use `\hrule' here except with leaders.
```

Le message d'erreur n'est pas vraiment explicite, mais il est manifeste que quelque chose a mal tourné. Pour avoir un début d'explication, on peut chercher à déboguer ce code. Un simple « \show\nxttok » placé tout au début de la macro \cmptok écrira dans le fichier log ce qu'est \nxttok à chaque itération. Voici ce que l'on obtient :

1. On pourrait éventuellement penser à \meaning mais les cas de figure à envisager sont tellement nombreux que cette manœuvre est irréaliste.

```
> \nxttok=the letter P.
> \nxttok=the letter r.
> \nxttok=the letter o.
> \nxttok=begin-group character {.
> \nxttok=the letter g.
> \nxttok=the letter r.
> \nxttok=the letter a.
> \nxttok=end-group character }.
! You can't use `\hrule' here except with leaders.
To put a horizontal rule in an hbox or an alignment,
you should use \leaders or \hrulefill (see The TeXbook).

! Missing number, treated as zero.
<to be read again>
```

Les messages d'erreurs (qui commencent par « ! ») continuent sur plusieurs lignes, seuls les deux premiers ont été retranscrits ici.

Il est donc intéressant de voir que \nxttok, accolade ouvrante devenue *implicite* (comme l'est \bgroup) par le truchement de \let, ne provoque pas d'erreur lorsqu'elle seule dans \frbox. Ainsi donc, \frbox{\bgroup} est valide… Mais ce n'est qu'apparent, car l'équilibrage des accolades a été rompu et TEX finirait tôt ou tard par se plaindre d'un « Missing } inserted ». Avant qu'il puisse en arriver là, l'erreur survient lorsque \let enferme dans \frbox l'accolade fermante explicite devenue implicite. Avant d'aller plus loin, il faut se souvenir que les boites de TEX ont comme propriété que leurs accolades peuvent *aussi* être implicites. À ce titre, les primitives de boites acceptent indifféremment accolades explicites et implicites qui sont donc interchangeables. Pour expliquer l'erreur constatée, voici la définition de \frbox où son argument #1 a été remplacé par un \egroup à la ligne nº 10 :

```
1   \def\frbox#1{%
2       \hbox{%
2           \vrule width\frboxrule
2           \vtop{%
2               \vbox{%
2                   \hrule height\frboxrule
2                   \kern\frboxsep
2                   \hbox{%
2                       \kern\frboxsep
2                       \egroup% est l'argument #1 de la macro
2                       \kern\frboxsep
2                       }%
2               }%
2               \kern\frboxsep
2               \hrule height\frboxrule
2           }%
2           \vrule width\frboxrule
2       }%
2   }
```

Le \egroup de la ligne nº 10 ferme prématurément la \hbox la plus intérieure (ligne nº 8), rejetant hors de celle-ci le \kern\frboxsep de la ligne nº 11. L'accolade explicite suivante (ligne nº 12) ferme la \vbox tandis que l'accolade explicite de la ligne nº 13 ferme la \vtop. Par conséquent, \kern\frboxsep et \hrule (lignes nº 14-15) se trouvent rejetés hors de la \vtop et prennent place dans la \hbox chapeau

de la ligne n° 2, en mode horizontal interne donc. C'est là que la première erreur survient car \hrule ne peut être employé en mode horizontal, sauf justement à l'utiliser avec \leaders comme le suggère le message d'erreur.

Pour en revenir à la lecture d'un code avec \let, on voit donc que les accolades explicites posent un vrai problème puisqu'elles sont derechef transformées par \let en leur « alias » \bgroup ou \egroup qui ne leur est pas équivalent. Là encore, patientons jusqu'au chapitre suivant où une nouvelle primitive nous permettra d'agir *avant* que le token ne soit lu.

2.1.4. Retour sur la macro \litterate

Forts de nos connaissances, nous sommes en mesure d'écrire une variante \Litterate* de la macro \litterate programmée à la page 81. Son action était de désactiver tous les tokens spéciaux de façon à écrire exactement le texte situé entre deux tokens identiques (par exemple « | ») servant de délimiteurs.

Le défi va être de fabriquer un équivalent de la macro \litterate mais sans modifier les catcodes. Pour y parvenir et pour rendre tous les tokens lus inoffensifs, il faudra faire en sorte que chaque token du texte lu soit précédé d'un \string.

La première chose à faire est de stocker le délimiteur qui suit \Litterate. Pour être surs que son catcode est 12, nous allons le faire précéder d'un \string que nous développerons : la séquence de contrôle \lim@tok recevra ce token. Par la suite, lorsque l'on parcourra le texte, si le token lu après avoir été traité par \string est égal à \lim@tok, cela signifiera que le deuxième token délimiteur est atteint et qu'il faut cesser la lecture.

Code n° IV-331

```
\catcode`\@11
\def\Litterate{%
  \begingroup% ouvrir un groupe
    \tt% et adopter une fonte à chasse fixe
    \afterassignment\Litterate@i% après l'assignation, aller à \Litterate@i
    \expandafter\let\expandafter\lim@tok\expandafter=\string% \lim@tok = token délimiteur
}
\def\Litterate@i{%
    \afterassignment\Litterate@ii%après avoir lu le prochain token, aller à \Litterate@ii
    \expandafter\let\expandafter\nxttok\expandafter=\string% lit le token suivant
}
\def\Litterate@ii{%
  \ifx\nxttok\lim@tok% si le token suivant="token délimiteur"
    \endgroup% fermer le groupe et finir
  \else
    \nxttok% sinon, afficher ce token
    \expandafter\Litterate@i% et lire le token suivant
  \fi
}
\catcode`\@12
\Litterate|Programmer     en \TeX {} est << facile >> !|
```

```
ProgrammerenTEX{}est<<facile>>!
```

Cela ne fonctionne pas du tout comme on l'attendait :

- les espaces sont ignorés ;
- la séquence de contrôle \TeX est exécutée.

En fait, les deux dysfonctionnements proviennent d'un même mal. L'erreur se trouve à la ligne nº 10 de la macro `\Litterate@i` qui lit le prochain token en développant au préalable `\string`. Si cette macro s'apprête à lire un espace, alors cet espace est précédé d'un `\string` et donc, après développement, cet espace reste inchangé. Il va être considéré par TEX comme l'espace facultatif qui vient après le signe « = » dans la syntaxe de `\let` ! Il sera donc ignoré et `\nxttok` sera rendu `\let`-égal au token suivant, mais sans que ce token ne soit soumis à `\string` puisque celui-ci a déjà été utilisé par l'espace. Voilà pourquoi la macro `\TeX`, qui se trouve juste après un espace, n'est pas passée par `\string` : `\nxttok` est donc devenu un alias pour `\TeX` et à la ligne nº 16, `\nxttok` est exécuté et provoque l'affichage du logo de TEX.

Pour corriger cette erreur de programmation, dans la macro `\Litterate@i`, il *faut* écrire l'espace après le signe « = » et sauter cet espace avec un `\expandafter` de plus. Mais on ne peut pas écrire :

```
\expandafter\let\expandafter\nxttok\expandafter=\expandafter \string
```

car l'espace serait ignoré puisque précédé par une séquence de contrôle. Au lieu de cet espace *explicite*, il va falloir écrire la macro `\space` (dont le texte de remplacement est un espace) que nous allons 1-développer après avoir 1-développé `\string`. Cela conduit à un pont d'`\expandafter`.

Code nº IV-332

```
\catcode`\@11
\def\Litterate{%
  \begingroup% ouvrir un groupe
    \tt% et adopter une fonte à chasse fixe
    \afterassignment\Litterate@i% après l'assignation, aller à \Litterate@i
    \expandafter\let\expandafter\lim@tok\expandafter=\string% \lim@tok = token délimiteur
}
\def\Litterate@i{%
  \afterassignment\Litterate@ii% après avoir lu un token, aller à \Litterate@ii
    \expandafter\expandafter\expandafter% 1-développer \string
  \let
    \expandafter\expandafter\expandafter% puis 1-développer \space en " "
  \nxttok
    \expandafter\expandafter\expandafter
  =\expandafter\space\string% et lire le token obtenu
}
\def\Litterate@ii{%
  \ifx\nxttok\lim@tok% si le token suivant="token délimiteur"
    \endgroup% fermer le groupe et finir
  \else
    \nxttok% sinon, afficher ce token
    \expandafter\Litterate@i% et lire le token suivant
  \fi
}
\catcode`\@12
\Litterate|Programmer     en \TeX {} est << facile >> : $~$^_#|
```

```
Programmer en \TeX{} est << facile >> : $~$^_#
```

Dernière observation : le résultat n'est pas *exactement* ce qui est entre les délimiteurs. Deux règles de lecture du code s'appliquent toujours :

- plusieurs espaces consécutifs sont lus comme un seul ;
- un espace qui suit une séquence de contrôle est ignoré. Ici, l'espace qui suit

la macro \TeX est ignoré.

Ces deux règles concernent les espaces, c'est-à-dire les tokens de catcode 10. Pour les contourner, nous allons devoir légèrement enfreindre la contrainte que nous nous étions donnée : changer le catcode de l'espace à 12 pour que les règles ne s'appliquent plus. Il suffit donc d'écrire

```
\catcode`\ =12
```

juste après le \begingroup pour que les espaces soient tous affichés.

La conclusion est que \Litterate fonctionne bien, mais de par l'assignation et le test qu'elle effectue pour chaque token, elle est *beaucoup plus lente* que son homologue \litterate. Cette dernière ne s'encombrait pas de tant précautions. Une fois les catcodes changés et les ligatures désactivées, elle laissait TEX librement lire le texte jusqu'à ce qu'il rencontre le délimiteur de fin qui, rendu actif et \let-égal à \endgroup, redonnait aux tokens spéciaux leur catcode normal et mettait fin au comportement initié par \litterate .

■ Exercice 97

Créer une macro \letterspace* de syntaxe

\letterspace{⟨*dimension de ressort*⟩}{⟨*texte*⟩}

où le ⟨*texte*⟩ est constitué de tokens de catcode 10, 11 ou 12. L'action de cette macro est d'afficher entre chaque caractère du ⟨*texte*⟩ un ressort dont la dimension est spécifiée dans le premier argument.

□ Solution

Puisque l'on sait lire un argument token par token, on va largement s'inspirer de ce qui a été fait avec la macro \Litterate.

Code n° IV-333

```
\catcode`@11
\newskip\ltr@spc
\def\letterspace#1#2{%
  \ltr@spc=#1\relax % assigne le ressort
  \letterspace@i#2\letterspace@i% appelle la macro \letterspace@i
}
\def\letterspace@i{%
  \afterassignment\letterspace@ii
  \let\nxttok=
}
\def\letterspace@ii{%
  \ifx\nxttok\letterspace@i% si la fin est atteinte
    \unskip% supprimer le dernier ressort qui est de trop
  \else
    \nxttok% afficher le token
    \hskip\ltr@spc\relax% insère le ressort
    \expandafter\letterspace@i% va lire le token suivant
  \fi
}
\catcode`@12
Voici "\letterspace{0.3em}{des lettres espacées}"
et voici "\letterspace{-0.075em}{de la compression}".
```

Voici "d e s l e t t r e s e s p a c é e s" et voici "de la compression".

Mais attention, si cela fonctionne ici, c'est parce que l'encodage du code source de ce livre est latin1 et donc, le caractère « é » est vu comme un seul token. Si l'encodage avait été UTF8

et le moteur 8 bits, le caractère « é », codé sur 2 octets, aurait été vu comme *deux* tokens. Ceux-ci sont inséparables pour former la lettre é (le premier est actif et prend le second comme argument). Comme l'algorithme les aurait séparés, on aurait certainement eu une erreur de compilation. Le même problème serait apparu si l'on avait écrit « \'e ». ■

■ Exercice 98

Inventer un procédé qui permet à une macro \ifinteger⋆ de tester si son argument est un nombre entier. La syntaxe sera :

\ifinterger{⟨*caractères*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

□ Solution

L'idée directrice est de définir un compteur, de faire précéder l'argument de cette macro d'un « 0 » et d'assigner 0#1 au compteur :

⟨*compteur*⟩=0#1\relax

Le « 0 » nous assure qu'au moins un caractère servira à former un ⟨*nombre*⟩. Après cette assignation, il faut aller voir ce qu'il reste jusqu'au \relax et qui ne sert pas à la formation du ⟨*nombre*⟩. S'il ne reste rien, alors la totalité de l'argument était constituée de caractères pouvant entrer dans la composition d'un ⟨*nombre*⟩ et peut donc être assimilée à un entier.

On va rencontrer un petit grain de sable lorsque #1 commence par le caractère « – » car

0–⟨*caractères*⟩

sera compris comme le nombre 0 suivi d'autres tokens ne pouvant entrer dans la composition d'un nombre à cause du « – ». Il faut donc prévoir de tester si #1 commence par « – » à l'aide de la macro \ifstart vue à la page 193 de la troisième partie. Si le test est positif, on se contentera de manger ce signe « – » et recommencer avec ce qui reste dans #1. Pour éviter qu'un argument vide donne un test \ifinteger positif, il est préférable de tester si #1 est vide auquel cas il faut lire l'argument ⟨*faux*⟩.

Code n° IV-334

```
\catcode`@11
\newcount\test@cnt
\def\ifinteger#1{%
  \ifstart{#1}{-}% si "-" est au début de #1
    {\exparg\ifinteger{\gobone#1}% l'enlever et recommencer
    }
    {\ifempty{#1}% sinon, si #1 est vide
      \secondoftwo% lire l'argument <faux>
      {\afterassignment\after@number% sinon, après l'assignation, aller à \after@number
      \test@cnt=0#1\relax% faire l'assignation
                        %(\relax termine le nombre et n'est pas mangé)
      }%
    }%
}
\def\after@number#1\relax{% #1 est ce qui reste après l'assignation
  \ifempty{#1}% teste si #1 est vide et lit l'argument <vrai> ou <faux> qui suit
}

\catcode`@12
1) \ifinteger{5644}{oui}{non}\qquad
2) \ifinteger{-987}{oui}{non}\qquad
3) \ifinteger{6a87}{oui}{non}\qquad
4) \ifinteger{abcd}{oui}{non}\qquad
5) \ifinteger{-a}{oui}{non}\qquad
```

```
6) \ifinteger{-}{oui}{non}\qquad
7) \ifinteger{3.14}{oui}{non}
```

1) oui 2) oui 3) non 4) non 5) non 6) non 7) non

Ajoutons que la macro \ifinteger n'est pas développable, car elle contient trois choses qui ne le sont pas :

1. la macro \ifstart ;
2. l'assignation à un compteur ;
3. la primitive \afterassignment.

■

2.1.5. En résumé...

Prenons le temps de faire le point sur la lecture token par token. Du côté des avantages :

– les espaces sont bien lus ;
– il est possible de faire un test sur chaque token après qu'il ait été lu.

Et du côté des défauts :

– lorsqu'un token a été lu et assigné à une séquence de contrôle par \let, on ne peut plus savoir quel était le token lu. Par conséquent, il est impossible d'ajouter ce token à une variable cumulative que l'on pourrait développer à la fin, hors de toute récursivité. Si l'on veut afficher le token lu, on doit mettre la séquence de contrôle qui lui est \let-égale dans le flux de lecture pendant la boucle récursive ce qui, on l'a vu, peut entrainer des problèmes lorsque des tokens sont des macros nécessitant un ou plusieurs arguments ;
– les tokens de catcode 1 et 2 (« { » et « } ») sont dénaturés par une assignation via \let et deviennent \bgroup et \egroup.

La conclusion est sans appel : la lecture token par token souffre de défauts qui la rendent bien peu commode. La suite va certainement adoucir cette conclusion…

2.2. Lire un token sans avancer la tête de lecture

Le titre ci-dessus a de quoi surprendre puisqu'on a considéré jusqu'à présent que toute lecture entrainait l'inévitable avancée de la tête de lecture de TEX ! Il y a eu là un petit mensonge par omission puisqu'il y a une seule exception à cette règle.

90 - Règle

La primitive \futurelet a la syntaxe suivante :

\futurelet\⟨*macro*⟩⟨*token1*⟩⟨*token2*⟩

L'effet est le suivant :

1. TEX effectue l'action « \let\⟨*macro*⟩=␣⟨*token2*⟩ » (en figeant le catcode de ⟨*token2*⟩) ;
2. puis la tête de lecture avance jusqu'à ⟨*token 1*⟩⟨*token2*⟩ qui restent intacts, comme s'ils n'avaient pas été lus.

Il y a comme un air de ressemblance avec \expandafter sauf qu'il n'est pas question de développement, mais d'assignation via \let : il y a bien le « saut » de ⟨*token1*⟩ pour faire une assignation sans que la tête de lecture ne bouge. Tout se passe comme si TEX allait lire le ⟨*token2*⟩ pour le *copier* juste après \⟨*macro*⟩ pour effectuer une assignation avec un \let. Et donc, le code

\futurelet\⟨*macro*⟩⟨*token1*⟩⟨*token2*⟩

est équivalent à

\let\⟨*macro*⟩= ⟨*token2*⟩⟨*token1*⟩⟨*token2*⟩

Assurément, \futurelet est une primitive formidable car elle va régler tous nos problèmes. En effet, si le texte de remplacement d'une \⟨*macro*⟩ se termine par

\futurelet\nxttok\testtok

alors, le token se trouvant juste après cette \⟨*macro*⟩ sera lu et stocké dans \nxttok, mais la tête de lecture n'aura pas bougé et ce token qui se trouve hors de la macro restera intact [2]. Il suffira que la macro \testtok teste ce token pour décider l'action à faire. Cette primitive permet donc de prévoir (d'un token) le futur, c'est sans doute pourquoi elle s'appelle \futurelet et non pas \letafter.

Voici comment programmer une macro \teststar⋆ dont le comportement va changer selon qu'elle est immédiatement suivie d'une étoile ou non : la version *normale* composera l'argument en italique et la version *étoilée* en gras. Clairement, un \futurelet sera utilisé pour savoir si le token qui suit immédiatement \teststar est une étoile ; la macro qui sera chargée de tester ce token est \teststar@i :

Code n° IV-335

```
\catcode`\@11
\def\teststar{%
  \futurelet\nxttok\teststar@i% % pioche le token suivant puis aller à \teststar@i
}
\def\teststar@i{%
  \ifx *\nxttok\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {% si le token suivant est une étoile
    \aftergroup\teststar@bf% après avoir lu "*", aller à \teststar@bf
    \let\nxttok= % lire l'étoile
    }% si le token n'est pas une étoile
    {\teststar@it% aller à \teststar@it
    }%
}
\def\teststar@bf#1{{\bf#1}}% lit l'argument et le compose en gras dans un groupe
\def\teststar@it#1{{\it#1\/}}% lit l'argument et le compose en italique
\catcode`\@12
Un essai \teststar{réussi} et un essai \teststar*{étoilé} réussi aussi.
```

Un essai *réussi* et un essai **étoilé** réussi aussi.

Il serait plus intéressant de programmer une macro à caractère généraliste « \ifnexttok⋆ » dont la syntaxe serait :

2. En particulier, les accolades explicites restent explicites et ne sont pas transformées en leurs alias \bgroup ou \egroup.

\ifnexttok⟨*token*⟩{⟨*code vrai*⟩}{⟨*code faux*⟩}

que l'on placerait à la fin du texte de remplacement d'une macro et qui irait tester le prochain token extérieur à la macro non égal à un espace. Le ⟨*code vrai*⟩ serait exécuté s'il est égal à ⟨*token*⟩ et ⟨*code faux*⟩ sinon.

Pour savoir si le token lu est un espace, nous le comparerons à \sptoken défini à la page 51. Profitons-en pour découvrir une autre façon (parmi beaucoup d'autres) de définir \sptoken avec une macro \deftok* :

Code n° IV-336

```
\catcode`@11
\def\deftok#1#2{\let#1= #2\empty}% définit le token #1 (\empty au cas où #2 est vide)
\deftok\sptoken{ }
\def\ifnexttok#1#2#3{% lit les 3 arguments : #1=token #2=code vrai #3=code faux
  \deftok\test@tok{#1}% stocke le token cherché
  \def\true@code{#2}\def\false@code{#3}% et les codes à exécuter
  \ifnexttok@i% aller à la macro récursive
}
\def\ifnexttok@i{%
  \futurelet\nxttok\ifnexttok@ii% piocher le token d'après et aller à \ifnexttok@ii
}
\def\ifnexttok@ii{%
  \ifx\nxttok\sptoken% si le prochain token est un espace
    \def\donext{%
      \afterassignment\ifnexttok@i% recommencer après
      \let\nxttok= % après avoir absorbé cet espace
    }%
  \else
    \ifx\nxttok\test@tok% si le prochain token est celui cherché
      \let\donext\true@code% exécuter le code vrai
    \else
      \let\donext\false@code% sinon code faux
    \fi
  \fi
  \donext% faire l'action décidée ci-dessus
}
\catcode`@12
\ifnexttok W{je vais lire un W : }{je ne vais pas lire un W : }W.\par
\ifnexttok W{je vais lire un W : }{je ne vais pas lire un W : }a.\par
\ifnexttok *{je vais lire une étoile : }{je ne vais pas lire une étoile : } *.\par
\ifnexttok *{je vais lire une étoile : }{je ne vais pas lire une étoile : }a.
```

```
je vais lire un W : W.
je ne vais pas lire un W : a.
je vais lire une étoile : *.
je ne vais pas lire une étoile : a.
```

Cela fonctionne bien comme on l'attend notamment à l'avant-dernière ligne où l'étoile est séparée de l'accolade fermante par un espace. Le test est pourtant positif ce qui montre que les espaces sont bien ignorés.

■ **Exercice 99**

Comment pourrait-on modifier la macro \ifnexttok pour lui donner une option afin de choisir d'ignorer les espaces ou pas ?

□ **Solution**

Le plus simple est de définir un booléen \iftestspace que l'on mettrait à faux par défaut et à vrai si l'on se trouve dans des conditions où l'on veut prendre en compte les espaces.

Code n° IV-337

```
\catcode`@11
\newif\iftestspace \testspacefalse
\def\deftok#1#2{\let#1= #2}\deftok\sptoken{ }
\def\ifnexttok#1#2#3{% #1=token #2=code vrai #3=code faux
  \let\test@tok= #1% stocke le token à tester
  \def\true@code{#2}\def\false@code{#3}% et les codes à exécuter
  \iftestspace \def\ifnexttok@i{\futurelet\nxttok\ifnexttok@ii}%
  \else        \def\ifnexttok@i{\futurelet\nxttok\ifnexttok@iii}%
  \fi% après avoir défini la macro récursive selon le booléen,
  \ifnexttok@i% l'exécuter
}
\def\ifnexttok@ii{% macro "normale" qui ne teste pas les espaces
  \ifx\nxttok\test@tok \expandafter\true@code% exécuter le code vrai
  \else                \expandafter\false@code% sinon code faux
  \fi
}
\def\ifnexttok@iii{% macro qui ignore les espaces
  \ifx\nxttok\sptoken% si le prochain token est un espace
    \def\donext{%
      \afterassignment\ifnexttok@i% lire le token d'après
      \let\nxttok= % après avoir absorbé l'espace
    }%
  \else
    \let\donext\ifnexttok@ii% sinon, faire le test "normal"
  \fi
  \donext% faire l'action décidée ci-dessus
}
\catcode`@12
\testspacefalse
\ifnexttok W{je vais lire un W : }{je ne vais pas lire un W : }W.\par
\ifnexttok W{je vais lire un W : }{je ne vais pas lire un W : } W.\medbreak
\testspacetrue
\ifnexttok W{je vais lire un W : }{je ne vais pas lire un W : }W.\par
\ifnexttok W{je vais lire un W : }{je ne vais pas lire un W : } W.
```

je vais lire un W : W.
je vais lire un W : W.

je vais lire un W : W.
je ne vais pas lire un W : W.

■

Élaborons à présent une macro `\ifstarred`* qui permettrait de tester, lorsque placée à la fin du texte de remplacement d'une macro, si cette dernière est étoilée ou pas et agir en conséquence. La syntaxe serait :

`\ifstarred{`⟨*code vrai*⟩`}{`⟨*code faux*⟩`}`

Gardons en mémoire qu'en cas de test positif avec `\ifnexttok`, l'étoile n'est pas mangée. Il faut donc trouver un moyen de manger cette étoile et le plus simple est d'utiliser `\firstoftwo` où le premier argument sera ⟨*code vrai*⟩ et le second l'étoile, non encore lue.

```
\def\ifstarred#1#2{% #1=code vrai  #2=code faux
    \ifnexttok *% si le prochain token est une étoile
        {\firstoftwo{#1}}% le 2e argument de \firstoftwo est "*"
        {#2}% sinon, exécuter le code faux
}
```

Comme l'argument {#2} figure une seule fois dans la macro en dernière position, nous pouvons nous permettre de ne pas l'écrire dans le texte de remplacement de \ifstarred. Au risque d'être bien moins compréhensible, le code peut donc se simplifier en :

```
\def\ifstarred#1{\ifnexttok*{\firstoftwo{#1}}}
```

Code n° IV-338

```
\def\ifstarred#1{\ifnexttok*{\firstoftwo{#1}}}
\catcode`\@11
\def\teststar{\ifstarred{\teststar@bf}{\teststar@it}}
\def\teststar@bf#1{{\bf#1}}
\def\teststar@it#1{{\it#1\/}}
\catcode`\@12

Un essai \teststar{réussi} et un essai \teststar*{étoilé} réussi aussi.
```

Un essai *réussi* et un essai **étoilé** réussi aussi.

2.3. Parser du code

2.3.1. La macro \parse

L'utilisation conjointe de \futurelet et de la lecture d'arguments est la méthode qu'il nous faut pour parcourir et analyser du code (l'anglicisme *parser du code* est souvent employé). Inventons une syntaxe simple :

\parse⟨*code à parser*⟩\parsestop

Le bond en avant que permet \futurelet est de savoir, avant de l'avoir absorbé, si le token qui suit dans le code est un token « à problème » comme le sont l'espace et l'accolade ouvrante. Le principe va être de parcourir le code et selon ce que l'on rencontre, ajouter des tokens au fur et à mesure dans un collecteur (qui sera un registre de tokens \code@toks).

Nous allons nous fixer comme objectif que si le ⟨*code à parser*⟩ contient un texte entre accolades, celui-ci sera ajouté tel quel au collecteur. En revanche, pour les autres tokens y compris l'espace, un test sera fait et selon le token rencontré, on pourra choisir ce que l'on ajoute au collecteur. Cela implique qu'un test sera effectué à chaque itération sur le prochain token non encore absorbé.

Voici l'algorithme qui va parser le code :

1) initialiser \code@toks à vide ;
2) lire le prochain token \nxttok avec \futurelet.
3) si \nxttok est \parsestop, manger \parsestop et afficher le collecteur \code@toks ;
4) si \nxttok est différent de \parsestop
   a) si \nxttok est un espace, manger l'espace et aller en 5 en transmettant « ␣ » ;
   b) si \nxttok est une accolade ouvrante (qui est vue comme un \bgroup par \futurelet), lire l'argument formé par le texte entre accolades et l'ajouter à \code@toks puis retourner en 2 ;
   c) dans les autres cas, aller en 5 pour lire ce prochain token ;
5) tester le token transmis en argument et, selon l'issue du test, ajouter ce que l'on souhaite à \code@toks puis, retourner en 2.

Du point de vue des macros à écrire, le point n° 1 est effectué par la macro chapeau \parse, le point n° 2 sera l'affaire d'une macro auxiliaire \parse@i et les points n^{os} 3 et 4 seront effectués par la macro \parse@ii. Le point n° 5 sera une macro *publique* « \testtoken » à un argument, écrite par l'utilisateur pour y effectuer les tests qu'il souhaite sur l'argument et selon l'issue de ceux-ci, ajouter ce qu'il souhaite au collecteur de tokens.

Code n° IV-339

```
\catcode`\@11
\def\parsestop{\parsestop}% définit la macro-quark se trouvant en fin de code
\newtoks\code@toks% registre contenant le code lu
\def\parseadd{\addtotoks\code@toks}
\def\parse{%
  \code@toks={}% initialise \code@toks à vide
  \parse@i% et passe la main à \parse@i
}
\def\parse@i{\futurelet\nxttok\parse@ii}% lit le prochain token et va à \parse@ii
\def\parse@ii{%
  \ifxcase\nxttok
    \parsestop\parsestop@i% si la fin va être atteinte, aller à \parsestop@i
    \sptoken\read@space% si un espace va être lu, aller à \read@space
    \bgroup\read@bracearg% si une accolade ouvrante  aller à \read@bracearg
  \elseif
    \testtoken% dans les autres cas, aller à \testtoken
  \endif
}

\def\parsestop@i\parsestop{% la fin est atteinte : manger \parsestop
  \the\code@toks% afficher le registre de tokens
}
\expandafter\def\expandafter\read@space\space{% manger un espace dans le code
  \testtoken{ }% et aller à \testtoken
}
\def\read@bracearg#1{% l'argument entre accolades est lu
  \parseadd{{#1}}% puis ajouté (entre accolades) tel quel à \code@toks
  \parse@i% ensuite, lire le prochain token
}
\catcode`@12

{\bf Exemple 1 :}
\catcode`@11
\def\testtoken#1{% macro qui teste le token
  \ifxcase{#1}
    a{\parseadd{e}}% remplacer a par e
    e{\parseadd{i}}% e par i
    i{\parseadd{o}}% i par o
    o{\parseadd{u}}% o par u
    u{\parseadd{y}}% u par y
    y{\parseadd{a}}% y par a
  \elseif
    \parseadd{#1}% sinon, ajouter le token tel quel
  \endif
  \parse@i% aller lire le token suivant
}%
\catcode`@12
\parse
Ce texte devenu \`a peine reconnaissable montre que le r\'esultat contient des sonorit\'es
{\bf catalanes, corses ou grecques} assez inattendues.
\parsestop\medbreak

```

```
{\bf Exemple 2 :}
\leavevmode \frboxsep=1pt
\catcode`@11
\def\testtoken#1{%
  \ifxcase{#1}% si #1 est un espace
    { }{\parseadd{\hskip 0.75em }}% ajouter un espace
    \ {\parseadd{\hskip 0.75em }}
  \elseif
    \parseadd{\frbox{#1}}% sinon, l'encadrer
  \endif
  \parse@i}%
\catcode`@12
\parse Programmer en \TeX\ est facile\parsestop\medbreak

{\bf Exemple 3 :}
\catcode`@11
\def\testtoken#1{%
  \ifx a#1\parseadd{X}\else\parseadd{#1}\fi% remplace les "a" par des "X"
  \parse@i
}
\catcode`@12
\parse a\bgroup\bf braca\egroup dabra\parsestop
```

Exemple 1 : Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís **catalanes, corses ou grecques** essiz onettindyis.

Exemple 2 : P r o g r a m m e r e n TEX e s t f a c i l e

Exemple 3 : X**brXcX**dXbrX

On le voit, le code entre accolades (ligne n° 50) n'a pas été modifié, c'est ce que l'on s'était fixé dans le cahier des charges.

■ Exercice 100

Une zone d'ombre subsiste quant à la séquence de contrôle `\bgroup`. En effet, une accolade ouvrante explicite sera vue par `\futurelet` comme `\bgroup`. Par conséquent, la macro `\parse` *ne sait pas distinguer* une accolade ouvrante de la séquence de contrôle `\bgroup`.

1. Décrire ce que contient le registre de tokens `\code@toks` lors de l'exemple n° 3.
2. Quelle difficulté se présente si on souhaite parser « `\hbox\bgroup XY\egroup` » ?
3. Comment peut-on résoudre cette difficulté ?

□ Solution

Lorsque le token qui doit être lu est `\bgroup`, la façon dont est programmée `\parse@ii` lui fait croire à tort que ce qui suit est une accolade ouvrante. La macro `\read@bracearg` lit donc l'argument (qui est ici `\bgroup`) et le met entre accolades. Le registre de tokens contient donc

```
X{\bgroup}\bf brXcX\egroup dXbrX
```

Pour la question n° 2, la réponse est donnée à la question précédente. En effet, si l'on parse

```
\hbox\bgroup XY\egroup
```

le registre de tokens va contenir

```
\hbox{\bgroup}XY\egroup
```

La `\hbox` va donc contenir un simple `\bgroup` au lieu du `XY`.

Répondre à la question n° 3 est difficile. Pour résoudre le problème, nous allons tout d'abord construire une macro `\ifbracefirst`⋆ de syntaxe

`\ifbracefirst{`⟨*argument*⟩`}{`⟨*code vrai*⟩`}{`⟨*code faux*⟩`}`

qui exécute ⟨*code vrai*⟩ si son ⟨*argument*⟩ commence par une accolade ouvrante (ou tout autre token de catcode 1) et ⟨*code faux*⟩ sinon.

Le simple fait que `\ifbracefirst` admette un argument et non un token implique de renoncer à la lecture *au fur et à mesure* du ⟨*texte*⟩ entre `\parse` et `\parsestop`. Nous sommes contraints de lire la *totalité* de ce qui reste avant `\parsestop` et de transmettre cet ⟨*argument*⟩ à `\ifbracefirst`.

Pour mener à bien sa mission, cette macro va détokéniser son ⟨*argument*⟩, n'en garder que le premier token et, avec `\catcode`, regarder s'il s'agit d'un token ayant un catcode égal à 1. Comme `\catcode` attend de lire un nombre, un développement maximal est lancé et un pont d'`\expandafter` détokénise d'abord l'argument `#1` puis en isole le premier token :

Code n° IV-340

```
\catcode`\@11
\def\ifbracefirst#1{%
  \ifnum\catcode\expandafter\expandafter\expandafter
    `\expandafter\firstto@nil\detokenize{#1W}\@nil=1 % tester si son catcode est 1
    \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
}
\catcode`\@12

a) \ifbracefirst{123 456}{vrai}{faux}\qquad
b) \ifbracefirst{\bgroup12\egroup3 456}{vrai}{faux}\qquad
c) \ifbracefirst{{12}3 456}{vrai}{faux}\qquad
d) \ifbracefirst{1{2}3 456}{vrai}{faux}\qquad
\begingroup
\catcode`[=1 \catcode`]=2 % les crochets deviennent des accolades
e) \ifbracefirst[[]123 456][vrai][faux]
\endgroup
```

a) faux b) faux c) vrai d) faux e) vrai

Le « `W` », écrit dans l'argument de `\detokenize`, nous met à l'abri d'une erreur de compilation en cas d'⟨*argument*⟩ vide. Par ailleurs, comme il est assez improbable que « `W` » ait un catcode 1 lorsque la macro est appelée, un ⟨*argument*⟩ vide donnera un test négatif.

Malgré cela, ce code comporte une erreur de programmation : si l'⟨*argument*⟩ commence par un espace, cet espace sera ignoré par `\firstto@nil` et donc, le test « `\ifbracefirst{␣{}}` » sera positif. Pour se prémunir de ce bug, il suffit au préalable d'effectuer le test `\ifspacefirst` et s'il est positif, agir en conséquence.

Code n° IV-341

```
\catcode`\@11
\def\ifbracefirst#1{% teste si #1 commence par un token de catcode 1
  \ifspacefirst{#1}% si #1 commence par un espace
    {\secondoftwo}% renvoyer faux
    {\ifnum\catcode\expandafter\expandafter\expandafter
      `\expandafter\firstto@nil\detokenize{#1W}\@nil=1 % tester si son catcode est 1
      \expandafter\firstoftwo
    \else
      \expandafter\secondoftwo
    \fi
    }%
}

```

```
\catcode`\@12
a) \ifbracefirst{}{vrai}{faux}\qquad b) \ifbracefirst{ }{vrai}{faux}\qquad
c) \ifbracefirst{ {}}{vrai}{faux}
```

a) faux b) faux c) faux

Revenons maintenant à la macro `\read@bracearg`. Voici son fonctionnement : tout d'abord, elle rajoute un token (ici `\relax`) avant ce qui reste à parser. Cette précaution évite que ce qui reste ne soit dépouillé des accolades s'il s'agit d'un unique texte entre accolades. Ensuite, elle passe la main à `\read@bracearg@i` dont l'argument, délimité par `\parsestop`, est tout ce qui reste à parser. Après avoir ôté le `\relax`, cet argument est testé avec `\ifbracefirst`. Remarquons que faire de `\parsestop` un délimiteur d'argument le rend *obligatoire* dans la syntaxe de `\parse` et introduit une contrainte assez forte. Par exemple, il ne sera pas possible de définir une autre séquence de contrôle `\let`-égale à `\parsestop` et utiliser cette macro en lieu et place de `\parsestop`.

La macro `\testtoken` est réduite à sa plus simple expression : elle ne fait que rajouter le token lu au collecteur de tokens `\code@toks` sans le modifier.

Pour s'assurer que le code est parsé correctement, au lieu d'écrire `\the\code@toks` à la fin du processus, on a écrit « `\detokenize\expandafter{\the\code@toks}` » de façon à afficher le contenu du registre de tokens. On aurait également pu écrire « `\showthe\code@toks` » pour écrire le contenu de ce registre dans le fichier `log`.

Le code ci-dessous ne montre que la macro `\read@bracearg` puisqu'elle seule a été modifiée.

Code n° IV-342

```
\catcode`\@11
\def\parsestop@i\parsestop{% la fin va être atteinte
  \detokenize\expandafter{\the\code@toks}% afficher le contenu du registre
}
\def\read@bracearg{%
  \read@bracearg@i\relax% ajoute un \relax avant de passer la main à \read@bracearg@i
}
\def\read@bracearg@i#1\parsestop{% #1 = tout jusqu'à \parsestop
  \exparg\ifbracefirst{\gobone#1}% retire le \relax et teste si #1 commence par "{"
    {\expandafter\read@bracearg@ii\gobone#1\parsestop}% si oui, aller à \readbrace@ii
    {\expandafter\testtoken\gobone#1\parsestop}% sinon, aller à \testtoken
}
\def\read@bracearg@ii#1{% lit l'argument entre accolades
  \parseadd{{#1}}% ajoute cet argument entre accolades
  \parse@i% aller lire le token suivant
}

\def\testtoken#1{%
  \parseadd{#1}% ajouter le token tel quel
  \parse@i% aller lire le token suivant
}
\catcode`\@12
\parse a\hbox\bgroup\bf braca\egroup {da}{}b{ra}\parsestop
```

a\hbox \bgroup \bf braca\egroup {da}{}b{ra}

Comme nous obtenons exactement ce qui est parsé, nous avons l'assurance que la macro `\read@brace` fait désormais correctement son travail. ■

2.3.2. Parser aussi l'intérieur des groupes

Le fait que la macro \parse n'aille pas voir à l'intérieur des accolades est certes une règle que l'on s'était fixée, mais on pourrait trouver utile d'ajouter une option pour que cette macro pénètre aussi dans les groupes entre accolades. En guise d'option, une étoile sera mise juste après la macro. Pour que l'option soit stockée quelque part, nous créerons un nouveau booléen \ifparse@groups que l'on mettra à vrai dans la version étoilée et à faux sinon. Ensuite, on sent bien que le nœud du problème se trouve dans la macro \read@bracearg, seul endroit où le parseur s'attend à lire un argument entre accolades.

La macro \read@bracearg doit donc être réécrite. Elle testera le booléen et s'il est faux, il faudra qu'elle agisse comme précédemment. S'il est vrai, \read@bracearg doit aller parser l'argument #1 qu'elle a lu. Pour ce faire, elle exécutera ce code *dans un groupe semi-simple* :

```
\parse*#1\parsestop
```

Une fois que le quark \parsestop est atteint, il ne faut pas afficher \code@toks comme le faisait \parsestop@i. La macro \parsestop@i doit donc être redéfinie localement afin de lui faire exécuter la chose suivante : ajouter le contenu de \code@toks collecté dans le groupe (sans oublier de l'envelopper d'accolades) au contenu de \code@toks tel qu'il était à l'extérieur du groupe. Le \endgroup fait donc partie prenant de \parsestop@i. Afin que cet ajout soit correctement mené à bien, un pont d'\expandafter sera mis en place pour retarder la sortie du groupe et développer au préalable « \the\code@toks » qui est l'argument de \parseadd :

```
\expandafter\endgroup% ne fermer le groupe qu'après avoir
\expandafter\parseadd% 1-développé à l'extérieur du groupe
\expandafter{\expandafter{\the\code@toks}}% "\the\code@toks"
```

Une fois cette action effectuée, il appartient à \parsestop@i de poursuivre le parsing en appelant la macro \parse@i qui lira le token suivant.

Voici le code complet :

Code n° IV-343

```
\catcode`\@11
\def\parsestop{\parsestop}% définit le quark se trouvant en fin de code
\newtoks\code@toks% alloue le registre contenant le code lu
\def\parseadd#1{\code@toks\expandafter{\the\code@toks#1}}
\newif\ifparse@group
\def\parse{%
  \code@toks{}% initialise le collecteur de tokens
  \ifstarred% teste si la macro est étoilée
    {\parse@grouptrue\parse@i}% mettre le booléen à vrai
    {\parse@groupfalse\parse@i}% sinon à faux
}
\def\parse@i{\futurelet\nxttok\parse@ii}% lit le prochain token
                                       % et va à \parse@ii
\def\parse@ii{%
  \ifxcase\nxttok
    \parsestop\parsestop@i% si la fin va être atteinte, aller à \parsestop@i
    \sptoken\read@space% si un espace va être lu, aller à \read@space
    \bgroup\read@bracearg% si une accolade ouvrante  aller à \read@bracearg
  \elseif
    \testtoken% dans les autres cas, aller à \testtoken
```

```
  \endif
}
\def\parsestop@i\parsestop{% la fin est atteinte
  \the\code@toks% afficher le registre de tokens
}
\expandafter\def\expandafter\read@space\space{% \read@space mange un espace dans le code
  \testtoken{ }% et va à \testtoken
}
\def\read@bracearg{%
  \read@bracearg@i\relax% ajoute un \relax avant de passer la main à \read@bracearg@i
}
\def\read@bracearg@i#1\parsestop{% l'argument tout jusqu'à \parsestop
  \expsecond\ifbracefirst{\gobone#1}% retire le \relax et teste si #1 commence par "{"
    {\expandafter\read@bracearg@ii\gobone#1\parsestop}% lire l'argument entre accolades
    {\expandafter\testtoken\gobone#1\parsestop}% sinon, tester le token
}
\def\read@bracearg@ii#1{% l'argument entre accolades est lu
  \ifparse@group\expandafter\firstoftwo\else\expandafter\secondoftwo\fi% si macro étoilée
    {\begingroup% ouvre un groupe pour parser l'intérieur de l'accolade
      \def\parsestop@i\parsestop{% redéfinir localement \parsestop@i pour
        \expandafter\endgroup% ne fermer le groupe qu'après avoir
        \expandafter\parseadd% 1-développé à l'extérieur du groupe
        \expandafter{\expandafter{\the\code@toks}}%
        \parse@i% puis va lire le token suivant
      }%
      \parse*#1\parsestop% <- le \parsestop@i fermera le groupe semi-simple
    }
    {\parseadd{{#1}}% macro non étoilée, on ajoute #1 tel quel entre accolades
    \parse@i% puis va lire le token suivant
    }%
}
\def\testtoken#1{% macro qui teste le token
  \ifxcase{#1}
    a{\parseadd{e}}
    e{\parseadd{i}}
    i{\parseadd{o}}
    o{\parseadd{u}}
    u{\parseadd{y}}
    y{\parseadd{a}}
  \elseif
    \parseadd{#1}%
  \endif
  \parse@i% aller lire le token suivant
}
\catcode`@12
\frboxsep=1pt
a) \parse
Ce texte devenu \`a peine reconnaissable montre que le r\'esultat contient des sonorit\'es
{\bf catalanes, \frbox{corses} ou grecques} assez inattendues.
\parsestop\medbreak

b) \parse*
Ce texte devenu \`a peine reconnaissable montre que le r\'esultat contient des sonorit\'es
{\bf catalanes, \frbox{corses} ou grecques} assez inattendues.
\parsestop
```

a) Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís **catalanes**, **corses** **ou grecques** essiz onettindyis.

b) Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís **cetelenis**, **cursis**

uy gricqyis essiz onettindyis.

■ Exercice 101

Expliquer d'où vient l'espace indésirable devant « Ci » que l'on constate lorsqu'on appelle la macro étoilée et proposer un remède pour le supprimer.

□ Solution

La présence de l'étoile fait que le retour à la ligne qui la suit est interprété comme un espace et n'est pas ignoré puisque dans ce cas, l'espace ne suit pas une séquence de contrôle.

Il va donc falloir tester si un espace suit l'étoile et dans l'affirmative, manger cet espace avant de commencer le processus. Un simple « `\ifnexttok{␣}` » tester si un espace suit l'étoile et dans l'affirmative, cet espace sera mangé en mettant à la fin du texte de remplacement de la macro :

```
\afterassignment\parse@i\let\nxttok=
```

Cette macro devient donc :

Code nº IV-344

```
\catcode`\@11
\def\parse{%
  \code@toks{}% initialise le collecteur de tokens
  \ifstarred
    {\parse@grouptrue
    \ifnexttok{ }% si un espace suit l'étoile
      {\afterassignment\parse@i% aller à \parse@i
      \let\nxttok= }% après l'avoir mangé
      {\parse@i}% sinon, aller à \parse@i
    }
    {\parse@groupfalse
    \parse@i
    }%
}
\def\testtoken#1{% macro qui teste le token
  \ifxcase{#1}
    a{\parseadd{e}}e{\parseadd{i}}i{\parseadd{o}}o{\parseadd{u}}
    u{\parseadd{y}}y{\parseadd{a}}
  \elseif
    \parseadd{#1}%
  \endif
  \parse@i% aller lire le token suivant
}
\catcode`\@12
\frboxsep=1pt
a) \parse
Ce texte devenu \`a peine reconnaissable montre que le r\'esultat contient des sonorit\'es
{\bf catalanes, \frbox{corses} ou grecques} assez inattendues.
\parsestop\medbreak
b) \parse*
Ce texte devenu \`a peine reconnaissable montre que le r\'esultat contient des sonorit\'es
{\bf catalanes, \frbox{corses} ou grecques} assez inattendues.
\parsestop
```

a) Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís **catalanes, corses ou grecques** essiz onettindyis.

b) Ci tixti diviny è pioni ricunneossebli muntri qyi li rísyltet cuntoint dis sunurotís **cetelenis, cursis uy gricqyis** essiz onettindyis.

■

2.4. Rechercher, remplacer un ensemble de tokens

Nous avions déjà écrit la macro \ifin à la page 192 qui permettait de savoir si un texte contenait un ensemble de tokens. Nous utilisions des arguments délimités et la limitation qui en découlait était que le motif cherché ne pouvait pas contenir d'accolades. Heureusement, nous allons pouvoir contourner cette difficulté en parcourant le code token par token.

Pour parvenir à nos fins, il nous suffit de programmer une macro capable de tester si un ⟨*texet*⟩ commence par un ⟨*motif*⟩.

2.4.1. Un ensemble de tokens commence-t-il par un motif ?

La fabrication de la macro \ifstartwith*, dont le but est de tester si un ⟨*texte*⟩ commence par un ⟨*motif*⟩ va donc devenir le seul enjeu pour l'instant. En voici la syntaxe :

\ifstartwith{⟨*texte*⟩}{⟨*motif*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

Avant d'en arriver à cette macro, la difficulté se concentre dans l'écriture d'une macro auxiliaire qui ampute le premier argument du texte du remplacement d'une ⟨*macro 1*⟩ et le place dans le texte de remplacement de ⟨*macro 2*⟩. Le mot « argument » doit ici être compris dans un sens *élargi* puisqu'il peut être un texte entre accolades, un unique token ou *un espace*.

Cette macro sera baptisée \grab@first et admettra deux arguments qui seront des séquences de contrôle

\grab@first\⟨*macro1*⟩\⟨*macro2*⟩

Le principe va consister à tester si le texte de remplacement de \⟨*macro1*⟩ commence par une accolade et sinon, utiliser \futurelet pour déterminer s'il commence par un espace ou par autre chose. Selon les trois cas envisagés, on s'orientera vers une macro qui lira convenablement le premier argument de ⟨*macro1*⟩ et le déplacera dans \⟨*macro2*⟩ tout en assignant ce qu'il reste à \⟨*macro1*⟩.

Code n° IV-345

```
\catcode`\@11
\def\grab@first#1#2{%
  \ifx#1\empty\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\let#2\empty% si #1 est vide, ne rien faire et assigner <vide> à #2
    }% si #1 n'est pas vide
    {\def\arg@b{#2}% stocke la macro #2 dans \arg@b
    \exparg\ifbracefirst#1% si le 1er token de #1 est "{"
      {\expandafter\grab@arg#1\@nil#1% aller lire l'argument avec \grab@arg
      }
      {% sinon, développer #1 avant de le regarder avec \futurelet :
      \expandafter\futurelet\expandafter\nxttok\expandafter\test@nxttok#1\@nil#1%
      % puis aller à \test@nxttok
      }%
    }%
}
\def\test@nxttok{% si le premier token de l'arg #1 de \grab@first est
  \ifx\nxttok\sptoken% un espace
    \expandafter\grab@spc% aller le lire avec \grab@spc
  \else
    \expandafter\grab@tok% sinon, lire le token avec \grab@tok
  \fi
```

```
}
\def\grab@arg#1{% assigne l'argument de \grab@first (mis entre accolades)
  \expandafter\def\arg@b{{#1}}% à #2
  \assign@tonil\relax% puis, assigne le reste à #1 de \grab@first
}
\expandafter\def\expandafter\grab@spc\space{%
  \expandafter\def\arg@b{ }% assigne un espace à #2 de \grab@first
  \assign@tonil\relax% puis, assigne le reste à #1 de \grab@first
}
\def\grab@tok#1{%% assigne le premier token de l'arg #1 de \grab@first
  \expandafter\def\arg@b{#1}% à la macro #2 de \grab@first
  \assign@tonil\relax% puis, assigne le reste à #1 de \grab@first
}
% assigne tout ce qui reste à lire (moins le "\relax") à la macro
% #1 de \grab@first
\def\assign@tonil#1\@nil#2{\expsecond{\def#2}{\gobone#1}}
\tt
a) \def\foo{1 {2 3} 4}\grab@first\foo\bar
   "\meaning\bar"\qquad"\meaning\foo"

b) \def\foo{ 1 {2 3} 4}\grab@first\foo\bar
   "\meaning\bar"\qquad"\meaning\foo""

c) \def\foo{{1 2} 3 4}\grab@first\foo\bar
   "\meaning\bar"\qquad"\meaning\foo"
```

```
a) "macro:->1"    "macro:-> {2 3} 4"
b) "macro:-> "    "macro:->1 {2 3} 4""
c) "macro:->{1 2}"    "macro:-> 3 4"
```

Revenons sur la macro `\grab@first` et ses macros auxiliaires… On constate dans son texte de remplacement que, en plus du cas trivial où #1 est vide, l'aiguillage se fait en deux temps selon ce par quoi commence le texte de remplacement de la macro #1. Le premier test fait concerne l'accolade ouvrante (ligne n° 7). Afin d'envisager tous les cas, supposons que le test est vrai : le texte de remplacement de #1 est de la forme « {⟨*code a*⟩}⟨*code b*⟩ ». L'appel suivant est alors effectué :

```
\expandafter\grab@arg#1\@nil#1
```

L'`\expandafter` fait apparaitre ce texte de remplacement puis `\grab@arg` en capture l'argument ⟨*code a*⟩ pour le mettre dans le texte de remplacement de la macro #2, après l'avoir rhabillé d'accolades. Une fois ceci fait, il reste à lire le code suivant :

⟨*code b*⟩`\@nil#1`

Rappelons-nous de ce code restant et passons maintenant à l'autre éventualité, celle où le texte de remplacement de #1 ne commence pas par une accolade. Le texte de remplacement de #1 est de la forme « ⟨*token a*⟩⟨*code b*⟩ ». Dans ce cas, on passe la main à cette ligne :

```
\expandafter\futurelet\expandafter\nxttok\expandafter\test@nxttok#1\@nil#1
```

Tout d'abord, un pont d'`\expandafter` développe la macro #1 pour faire apparaitre son texte de remplacement ce qui donne

`\futurelet\nxttok\test@nxttok`⟨*token a*⟩⟨*code b*⟩`\@nil#1`

Ensuite, le `\futurelet` va rendre `\nxttok` `\let`-égal à ⟨*token a*⟩. La macro `\test@nxttok` est appelée par le `\futurelet` pour tester ce qu'est ⟨*token a*⟩ et

selon qu'il est un espace ou pas, appelle \grab@spc ou \grab@tok. Ces deux macros capturent le ⟨*token a*⟩ pour le mettre dans le texte de remplacement de #2 et le code qui reste à lire après cette opération est ici encore :

⟨*code b*⟩\@nil#1

Que le texte de remplacement de #1 commence par un argument entre accolades, un espace ou un token, il reste dans tous les cas la même chose à lire

⟨*code b*⟩\@nil#1

Le code « \assign@tonil\relax » est exécuté à la toute fin de chacune des trois macros \grab@arg, \grab@spc et \grab@tok devant ce code restant de telle sorte que l'on a formé l'appel

\assign@tonil\relax⟨*code b*⟩\@nil#1

La macro \assign@tonil mange ce \relax avec \assign@tonil puis assigne à la \⟨*macro1*⟩ (qui est l'argument #1) le ⟨*code b*⟩ et ceci termine le processus. Ouf, nous avons fait le plus dur !

Ayant construit \grab@first, il suffit ensuite pour \ifstartwith d'extraire le premier argument élargi de ⟨*texte*⟩ et de ⟨*motif*⟩, de les comparer et renvoyer ⟨*faux*⟩ s'il ne sont pas égaux et dans le cas contraire, recommencer jusqu'à ce que le ⟨*motif*⟩ soit vide. À ce moment-là, comme aucune comparaison n'a été fausse, il faut renvoyer ⟨*vrai*⟩. Le cas où le ⟨*texte*⟩ devient vide *avant* le ⟨*motif*⟩ doit être envisagé, car cela signifie que le ⟨*texte*⟩ est plus court que le ⟨*motif*⟩ et dans cette éventualité, il faut renvoyer ⟨*faux*⟩.

Code nº IV-346

```
\catcode`\@11
\def\ifstartwith#1#2{% #1=<texte>  #2=<motif>
  \ifempty{#2}
    \firstoftwo% si <motif> est vide, renvoyer vrai
    {\ifempty{#1}% si <code> est vide et <motif> non vide
      \secondoftwo% renvoyer faux
      {\def\startwith@code{#1}\def\startwith@pattern{#2}%
      \ifstartwith@i% dans les autres cas, aller à \ifstartwith@i
      }%
    }%
}
\def\ifstartwith@i{%
  \grab@first\startwith@code\first@code% extrait le premier "argument" de <texte>
  \grab@first\startwith@pattern\first@pattern% et celui de <motif>
  \ifx\first@code\first@pattern\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {% si les deux arguments élargis sont égaux
    \exparg\ifempty\startwith@pattern
      \firstoftwo% et que <motif> ne contient plus rien => vrai
      {\exparg\ifempty\startwith@code
        \secondoftwo% si <texte> ne contient plus rien => faux
        \ifstartwith@i% sinon poursuivre les tests
      }%
    }
    \secondoftwo% si les deux argument élargis sont différents, renvoyer faux
}
\catcode`\@12

1) \ifstartwith{a b c}{a b}{oui}{non}\quad 2) \ifstartwith{a b}{a b c}{oui}{non}\quad
```

```
3) \ifstartwith{ 123 }{ }{oui}{non}\quad   4) \ifstartwith{ 123 }{1}{oui}{non}\quad
5) \ifstartwith{1{2} 3}{12}{oui}{non}\quad 6) \ifstartwith{1{2} 3}{1{2} }{oui}{non}\quad
7) \ifstartwith{{} {}}{ }{oui}{non}\quad   8) \ifstartwith{{} {}}{{} }{oui}{non}\quad
9) \ifstartwith{ {12} a}{ }{oui}{non}\quad10) \ifstartwith{ {12} a}{ {1}}{oui}{non}
```

1) oui 2) non 3) oui 4) non 5) non 6) oui 7) non 8) oui 9) oui 10) non

2.4.2. Un code contient-il un motif ?

Puisque nous avons à notre disposition la macro `\ifstartwith` qui sera la brique essentielle pour la suite, poursuivons avec la macro `\ifcontain`⋆ dont la syntaxe est

$$\texttt{\textbackslash ifcontain}\{\langle \textit{texte} \rangle\}\{\langle \textit{motif} \rangle\}\{\langle \textit{vrai} \rangle\}\{\langle \textit{faux} \rangle\}$$

et qui renverra ⟨*vrai*⟩ si le ⟨*texte*⟩ contient le ⟨*motif*⟩ et ⟨*faux*⟩ sinon.

L'algorithme le plus naïf, que nous allons utiliser, consiste à tester si ⟨*motif*⟩ est au début du ⟨*texte*⟩. Dans l'affirmative, il faut renvoyer ⟨*vrai*⟩ et dans le cas contraire, manger le premier « argument élargi » de ⟨*texte*⟩ et recommencer jusqu'à ce que ⟨*texte*⟩ soit vide auquel cas, on renvoie ⟨*faux*⟩.

Code n° IV-347

```
\catcode`\@11
\def\ifcontain#1#2{% #1 contient-il #2?
  \def\main@arg{#1}\def\pattern@arg{#2}% stocke le <code> et le <motif>
  \ifempty{#2}
    \firstoftwo% si #2 est vide => vrai
    {\ifempty{#1}
      \secondoftwo% si #1 est vide et pas #2 => faux
      \ifcontain@i% sinon, aller faire les tests
    }%
}
\def\ifcontain@i{%
  \exptwoargs\ifstartwith\main@arg\pattern@arg
    \firstoftwo% si motif est au début de code => vrai
    {\exparg\ifempty\main@arg
      \secondoftwo% sinon, si code est vide => faux
      {\grab@first\main@arg\aux@arg% autrement, manger le 1er "argument" de code
      \ifcontain@i% et recommencer
      }%
    }%
}
\catcode`\@12
1) \ifcontain{abc def}{c }{oui}{non}\quad  2) \ifcontain{abc def}{cd}{oui}{non}\quad
3) \ifcontain{12 34 5}{1 }{oui}{non}\quad  4) \ifcontain{12 34 5}{ }{oui}{non}\quad
5) \ifcontain{a{b c}d}{b c}{oui}{non}\quad 6) \ifcontain{a{b c}d}{{b c}}{oui}{non}\quad
7) \ifcontain{{} {}}{ }{oui}{non}\quad     8) \ifcontain{{} {}}{{}}{oui}{non}
```

1) oui 2) non 3) non 4) oui 5) non 6) oui 7) oui 8) oui

On le voit au cas n° 5, la macro `\ifcontain` n'inspecte pas l'intérieur des groupes et donc, elle considère que le motif « `b␣c` » n'est pas contenu dans « `a{b␣c}d` ». On pourrait souhaiter que la macro `\ifcontain` aille aussi voir à l'intérieur des groupes lorsqu'elle est étoilée. Pour ajouter cette fonctionnalité, il va falloir modifier la macro `\ifcontain@i` pour qu'elle teste si `\main@arg` commence par une accolade ouvrante et dans ce cas, agir différemment.

Voici l'algorithme que l'on pourrait suivre :

1) si ⟨*motif*⟩ est vide, renvoyer ⟨*vrai*⟩, sinon, si ⟨*texte*⟩ est vide, renvoyer ⟨*faux*⟩. Si aucun n'est vide, aller au point n° 2 ;
2) initialiser le booléen `\ifin@group` à faux ;
3) si ⟨*texte*⟩ commence par « { »
    a) enlever le premier argument de ⟨*texte*⟩ et le mettre dans une macro temporaire `\aux@arg` ;
    b) ouvrir un groupe semi-simple ;
    c) initialiser le booléen `\ifin@group` à `vrai` ;
    d) dans ce groupe, prendre ⟨*texte*⟩ égal à `\aux@arg` ;
    e) aller au point 3.
4) sinon
    a) si ⟨*motif*⟩ est au début de ⟨*texte*⟩ sortir de tous les groupes et renvoyer ⟨*vrai*⟩ ;
    b) sinon
        i) si ⟨*texte*⟩ est vide
            – si `\ifin@group` est `vrai` sortir du groupe semi-simple et aller en 3 ;
            – sinon renvoyer ⟨*faux*⟩ ;
        ii) sinon retirer le premier argument élargi de ⟨*texte*⟩ et aller en 3.

Distribuons les tâches. Les points 1 et 2 seront traités par une macro chapeau comme auparavant. Les points 3 et 4 seront écrits dans une macro récursive `\ifcontain@star` alors que la macro récursive `\ifcontain@nostar`, vue précédemment sous le nom de `\ifcontain@i` sera employée dans le cas ou `\ifcontain` n'est pas étoilé.

Pour le point 4.a, nous devons également écrire une macro `\return@true` qui ferme tous les groupes et qui renvoie ⟨*vrai*⟩. Le booléen `\ifin@group` rend la tâche facile puisqu'il est localement `vrai` dans tous les groupes et faux à l'extérieur. Une simple macro récursive qui ferme successivement les groupes jusqu'à ce que le booléen soit faux est suffisante :

```
\def\return@true{%
    \ifin@group% si on est dans un groupe
        \endgroup% le fermer
        \expandafter\return@true% et recommencer
    \else
        \expandafter\firstoftwo% sinon, renvoyer vrai
    \fi
}
```

Voici le code complet qui en résulte :

Code n° IV-348

```
\catcode`@11
\newif\ifin@group
\def\ifcontain{%
  \ifstarred
    {\in@groupfalse\ifcontain@i\ifcontain@star}%
    {\ifcontain@i\ifcontain@nostar}}

\def\ifcontain@i#1#2#3{% #1 = macro à appeler selon étoile ou pas. #2 = code. #3 = motif
```

```
  \def\main@arg{#2}\def\pattern@arg{#3}%
  \ifempty{#3}
    \firstoftwo
    {\ifempty{#2}
      \secondoftwo
      #1% aller à \ifcontain@star ou \ifcontain@nostar
    }%
}
\def\ifcontain@nostar{%
  \exptwoargs\ifstartwith\main@arg\pattern@arg
    \firstoftwo% si motif est au début de code => vrai
    {\exparg\ifempty\main@arg
      \secondoftwo% sinon, si code est vide => faux
      {\grab@first\main@arg\aux@arg% autrement, manger le 1er "argument" de code
      \ifcontain@nostar% et recommencer
      }%
    }%
}
\def\ifcontain@star{%
  \expandafter\ifbracefirst\expandafter{\main@arg}% si code commence par "{"
    {\grab@first\main@arg\aux@arg% enlever {<argument>} de main@arg
    \begingroup% ouvrir un groupe
      \in@grouptrue% mettre le booléen à vrai
      \expandafter\def\expandafter\main@arg\aux@arg% assigner "argument" à \main@arg
      \ifcontain@star% et recommencer avec ce nouveau \main@arg
    }% si code ne commence pas par "{"
    {\exptwoargs\ifstartwith\main@arg\pattern@arg% si motif est au début de code
      \return@true% renvoyer vrai
      {\expandafter\ifempty\expandafter{\main@arg}% si code est vide
        {\ifin@group% et que l'on est dans un groupe
          \endgroup \expandafter\ifcontain@star% en sortir et recommencer
        \else% si on n'est pas dans un groupe, le code a été parcouru
          \expandafter\secondoftwo% sans trouver <motif> => renvoyer faux
        \fi
        }% si code n'est pas vide
        {\grab@first\main@arg\startwith@code% manger le 1er "argument" de code
        \ifcontain@star% et recommencer
        }%
      }%
    }%
}
\def\return@true{%
  \ifin@group% tant qu'on est dans un groupe
    \endgroup \expandafter\return@true% en sortir et recommencer
  \else
    \expandafter\firstoftwo% sinon, renvoyer vrai
  \fi
}
\catcode`@12
1) \ifcontain{ab {c d}ef}{c}{oui}{non}\quad 2) \ifcontain*{ab {c d}ef}{c}{oui}{non}
```

1) non 2) oui

2.4.3. Remplacer un motif par un autre

Il nous reste une dernière étape, celle de remplacer dans ⟨*texte*⟩ un ⟨*motif a*⟩ par un ⟨*motif b*⟩. La syntaxe naturelle pour une telle macro est :

\substitute{⟨*texte*⟩}{⟨*motif a*⟩}{⟨*motif b*⟩}

La recette est connue ; il nous suffit de reprendre une méthode similaire à celle de la macro \ifcontain (en version étoilée ou pas) et si le ⟨*texte*⟩ restant à lire commence par le ⟨*motif*⟩, on ajoutera à un registre de tokens tenant lieu de collecteur le ⟨*motif b*⟩. Dans le cas contraire, il faudra ajouter à ce même registre le premier argument élargi du ⟨*texte*⟩.

Comme il n'est pas question de fermer tous les groupes d'un coup, nous allons nous passer du booléen \ifin@group en définissant une macro de fin de processus \substitute@end qui sera appelée lorsque ⟨*texte*⟩ sera vide. Ce ⟨*texte*⟩ est vide dans deux situations ; soit la totalité du texte initial a été parsée, soit nous sommes dans un groupe et le ⟨*texte*⟩ local est parsé. En dehors de tout groupe ou pour la macro non étoilée, la macro \substitute@end se contentera d'afficher le contenu du registre de tokens \subst@toks :

```
\def\substitute@end{\the\subst@toks}
```

En revanche, lorsque la macro est étoilée et que l'on inspecte un groupe, cette macro sera modifiée de la même façon qu'était modifiée la macro \parsestop@i : \substitute@end devra ajouter au collecteur \subst@toks hors du groupe ce qu'il était dans le groupe, après l'avoir enveloppé d'accolades.

Code n° IV-349

```
\catcode`@11
\newtoks\subst@toks% registre de tokens pour stocker le <texte> modifié
\def\substitute{%
  \def\substitute@end{\the\subst@toks}% macro exécutée à la fin
  \ifstarred
    {\let\recurse@macro\substitute@star  \substitute@i}%
    {\let\recurse@macro\substitute@nostar\substitute@i}%
}
\def\substitute@i#1#2#3#4{% #1=macro à appeler selon étoile ou pas.
                          % #2=<texte> #3=<motif>  #4=<motif de substi>
  \def\code@arg{#2}\def\pattern@arg{#3}\def\subst@arg{#4}%
  \subst@toks={}% initialiser le collecteur à vide
  #1% aller à \substitute@star ou \substitute@nostar
}
\def\substitute@nostar{%
  \exptwoargs\ifstartwith\code@arg\pattern@arg% si le <texte> commence par <motif>
    {\eaddtotoks\subst@toks\subst@arg% ajouter <motif subst>
    \grab@first\code@arg\aux@code% manger le 1er "argument" de <texte>
    \substitute@nostar% et recommencer
    }
    {\expandafter\ifempty\expandafter{\code@arg}%
      {\substitute@end% sinon, si <texte> est vide => afficher le registre de tokens
      }
      {\grab@first\code@arg\aux@arg% autrement, manger le 1er "argument" de <texte>
      \eaddtotoks\subst@toks\aux@arg% et l'ajouter au registre de tokens
      \substitute@nostar% et recommencer
      }%
    }%
}
\def\substitute@star{%
  \expandafter\ifbracefirst\expandafter{\code@arg}% si <texte> commence par "{"
    {\grab@first\code@arg\aux@arg% enlever {<argument>} de \code@arg
    \begingroup% ouvrir un groupe
      \def\substitute@end{% modifier localement la macro exécutée à la fin
        \expandafter\endgroup\expandafter% avant de fermer le groupe
        \addtotoks\expandafter\subst@toks\expandafter% ajouter au registre hors du groupe
          {\expandafter{\the\subst@toks}}% ce qui est collecté localement, mis entre {}
```

```
      \substitute@star% puis recommencer
      }%
      \subst@toks{}% initialiser à vide
      \expandafter\def\expandafter\code@arg\aux@arg% % assigner "argument" au <texte>
      \substitute@star% et recommencer avec ce nouveau \code@arg
    }% si code ne commence pas par "{"
    {\exptwoargs\ifstartwith\code@arg\pattern@arg% si <motif> est au début de <texte>
      {\eaddtotoks\subst@toks\subst@arg% ajouter <motif subst>
      \grab@first\code@arg\aux@code% manger le 1er "argument" de <texte>
      \substitute@star% et recommencer
      }
      {\expandafter\ifempty\expandafter{\code@arg}% si <texte> est vide
        {\substitute@end% aller à la macro de fin
        }% si <texte> n'est pas vide
        {\grab@first\code@arg\aux@code% manger le 1er "argument" de <texte>
        \eaddtotoks\subst@toks\aux@code% et l'ajouter au registre
        \substitute@star% et recommencer
        }%
      }%
    }%
}
\catcode`@12
1) \substitute{ab{\bf racada}bra}{a}{W}\qquad
2) \substitute*{ab{\bf racada}bra}{a}{W}\qquad
3) \substitute{12\frbox{\bf 34}56}{\bf}{\it}\qquad
4) \substitute*{12\frbox{\bf 34}56}{\bf}{\it}
```

1) ab**racada**bra2) ab**racada**bra3) 12 **34** 564) 12 **34** 56

■ Exercice 102

Dans le code proposé, les lignes n^{os} 16-28 sont identiques aux lignes n^{os} 44-56, exception faite des macros `\substitute@star` et `\substitute@nostar`. Proposer une amélioration afin de ne pas écrire deux fois cette portion de code.

Définir une macro `\substitutetocs`* de syntaxe

$$\texttt{\textbackslash substitutetocs\{}\langle texte\rangle\texttt{\}\{}\langle motif\ a\rangle\texttt{\}\{}\langle motif\ b\rangle\texttt{\}\textbackslash}\langle macro\rangle$$

qui assigne à $\langle macro\rangle$ le résultat de la substitution.

□ Solution

On peut définir une macro auxiliaire `\recurse@macro` qui, selon que la version est étoilée ou non, sera `\let`-égale à `\substitute@star` ou `\substitute@nostar`. Il suffit dès lors de remplacer les occurrences de `\substitute@star` et celles de `\substitute@nostar` par `\recurse@macro` et remplacer les lignes n^o 44-56 par `\substitute@nostar`.

Pour créer `\substitutetocs`, nous allons nous mettre à profit la macro `\substitute@end` qui est en réalité un hook. Il sera redéfini pour que la $\langle macro\rangle$ (argument #4) reçoive le contenu du registre de tokens `\subst@toks`. Un `\edef` peut être utilisé ici, car lorsque `\the`$\langle$*registre de tokens*$\rangle$ est placé dans un `\edef`, le contenu de ce registre *non développé* apparait dans le texte de remplacement de la macro ainsi définie (voir page 119).

Une fois le hook correctement défini, le déroulement de la macro `\subtitute` se poursuit.

Code n^o IV-350

```
\catcode`@11
\newtoks\subst@toks% registre de tokens pour stocker le <texte> modifié
```

```
\def\substitute{%
  \def\substitute@end{\the\subst@toks}% macro exécutée à la fin
  \ifstarred
    {\let\recurse@macro\substitute@star  \substitute@i}%
    {\let\recurse@macro\substitute@nostar\substitute@i}%
}
\def\substitute@i#1#2#3{% #1=<texte> #2=<motif> #3=<motif de substi>
  \def\code@arg{#1}\def\pattern@arg{#2}\def\subst@arg{#3}%
  \subst@toks={}% initialiser à vide
  \recurse@macro% aller à \substitute@star ou \substitute@nostar
}
\def\substitute@nostar{%
  \exptwoargs\ifstartwith\code@arg\pattern@arg% si le <texte> commence par <motif>
    {\eaddtotoks\subst@toks\subst@arg% ajouter <motif subst>\eaddtotoks
    \grab@first\code@arg\aux@code% manger le 1er "argument" de <texte>
    \recurse@macro% et recommencer
    }
    {\expandafter\ifempty\expandafter{\code@arg}%
      \substitute@end% sinon, si <texte> est vide => afficher le registre de tokens
      {\grab@first\code@arg\aux@arg% autrement, manger le 1er "argument" de <texte>
      \eaddtotoks\subst@toks\aux@arg% et l'ajouter au registre de tokens
      \recurse@macro% et recommencer
      }%
    }%
}
\def\substitute@star{%
  \expandafter\ifbracefirst\expandafter{\code@arg}% si <texte> commence par "{"
    {\grab@first\code@arg\aux@arg% enlever {<argument>} de \code@arg
    \begingroup% ouvrir un groupe
      \def\substitute@end{% modifier localement la macro exécutée à la fin
        \expandafter\endgroup\expandafter% avant de fermer le groupe
        \addtotoks\expandafter\subst@toks\expandafter% ajouter au registre hors du groupe
          {\expandafter{\the\subst@toks}}% ce qui est collecté localement, mis entre {}
        \recurse@macro% puis recommencer
      }%
      \subst@toks{}% initialiser à vide
      \expandafter\def\expandafter\code@arg\aux@arg% % assigner "argument" au <texte>
      \recurse@macro% et recommencer avec ce nouveau \code@arg
    }% si <texte> ne commence pas par "{" :
    \substitute@nostar% exécuter macro non étoilée pour une seule itération
}
\def\substitutetocs{%
  \ifstarred
    {\let\recurse@macro\substitute@star  \substitutetocs@i}%
    {\let\recurse@macro\substitute@nostar\substitutetocs@i}%
}
\def\substitutetocs@i#1#2#3#4{% #1=<texte> #2=<motif> #3=<motif de substi> #4=\<macro>
  \def\substitute@end{\edef#4{\the\subst@toks}}% macro exécutée à la fin
  \substitute@i{#1}{#2}{#3}% continuer comme avec \substitute
}
\catcode`@12
\frboxsep=1pt
1) \substitute{ab{\bf racada}bra}{a}{W}\qquad
2) \substitute*{ab{\bf racada}bra}{a}{W}\qquad
3) \substitute{12\frbox{\bf 34}56}{\bf}{\it}\qquad
4) \substitute*{12\frbox{\bf 34}56}{\bf}{\it}
\medbreak

1) \substitutetocs{ab{\bf racada}bra}{a}{W}\foo       \meaning\foo\par
2) \substitutetocs*{ab{\bf racada}bra}{a}{W}\foo      \meaning\foo\par
```

```
3) \substitutetocs{12\frbox{\bf 34}56}{\bf}{\it}\foo  \meaning\foo\par
4) \substitutetocs*{12\frbox{\bf 34}56}{\bf}{\it}\foo \meaning\foo
```

1) Wb**racada**brW 2) Wb**rWcWdW**brW 3) 12**34**56 4) 12*34*56

1) macro:->Wb{\bf racada}brW
2) macro:->Wb{\bf rWcWdW}brW
3) macro:->12\frbox {\bf 34}56
4) macro:->12\frbox {\it 34}56

■

Chapitre 3

Des macros sur mesure

3.1. Macros à arguments optionnels

L'utilisation de `\ifnexttok` va nous permettre de bâtir des macros acceptant des arguments *optionnels*. Les arguments optionnels de ces macros ont vocation à se trouver entre des délimiteurs que l'on peut choisir comme on le souhaite (parenthèses, crochets [1] ou autre), tout en rendant ces arguments et leurs délimiteurs *facultatifs*. En cas d'absence d'argument optionnel, un argument *par défaut*, fixé lors de la définition de la macro sera utilisé. Pour nous placer dans la continuité de ce qui se fait, nous prendrons les crochets pour délimiter les arguments optionnels.

La méthode consiste, pour une macro chapeau, à tester avec `\ifnexttok` si le prochain token est un délimiteur ouvrant. Si tel est le cas, il faut appeler une macro auxiliaire à argument délimité qui va lire l'argument optionnel entre délimiteurs. Dans le cas contraire, il faut transmettre à la macro à arguments délimités un argument optionnel entre délimiteurs que l'on choisit par défaut.

Imaginons une macro `\vecteur{`⟨*lettre*⟩`}` qui admet un argument obligatoire et qui se comporte de la façon suivante :

`\vecteur{n}` donne $(x_1, \ldots, x_n)$
`\vecteur{k}` donne $(x_1, \ldots, x_k)$
`\vecteur{i}` donne $(x_1, \ldots, x_i)$

L'argument obligatoire ⟨*lettre*⟩ est l'indice du dernier terme. On pourrait vouloir spécifier un argument *optionnel* entre crochets qui viendrait juste après la macro et qui serait l'indice du premier terme. La valeur par défaut de cet argument serait 1. Voici comment procéder :

1. C'est la choix fait par LaTeX et ConTeXt.

Code nº IV-351

```
\catcode`@11
\def\vecteur{%
  \ifnexttok[% si la macro est suivie d'un crochet
    \vecteur@i% aller à la macro à arguments délimités
    {\vecteur@i[1]}% sinon, ajouter l'argument optionnel par défaut entre crochets
}
\def\vecteur@i[#1]#2{% #1=arg optionnel  #2=arg obligatoire
  $(x_{#1},\ldots,x_{#2})$
}
\catcode`@12
1) \vecteur{n}\qquad 2) \vecteur[0]{k}\qquad 3) \vecteur[i]j \qquad4) \vecteur[]n
```

1) $(x_1, \ldots, x_n)$ 2) $(x_0, \ldots, x_k)$ 3) $(x_i, \ldots, x_j)$ 4) $(x, \ldots, x_n)$

Bien que fonctionnant bien, on peut imaginer ce qu'il va se passer si cette macro est appelée alors que l'on est *déjà* en mode mathématique : le premier $ rencontré dans la macro va faire quitter ce mode et provoquer des erreurs de compilation avec le token « _ » qui n'opère qu'en mode mathématique. Il faut donc programmer une macro `\forcemath`⋆ qui compose son argument en mode mathématique, quel que soit le mode en cours. Pour cela, on va utiliser le test `\ifmmode` qui est un test de TEX :

`\ifmmode`⟨*code vrai*⟩`\else`⟨*code faux*⟩`\fi`

Ce test est vrai si le mode mathématique est en vigueur. C'est lorsqu'il est faux que la macro `\forcemath` doit composer son argument entre délimiteurs mathématiques « $ ».

Une autre amélioration serait de programmer un deuxième argument optionnel pour spécifier la lettre des coordonnées du vecteur, en choisissant x par défaut. Bien qu'il nous soit possible de le mettre entre crochets, nous allons, par souci de changement, mettre cet argument optionnel entre parenthèses :

Code nº IV-352

```
\catcode`@11
\def\forcemath#1{% compose #1 en mode math, quel que soit le mode en cours
  \ifmmode\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {#1}{$#1$}%
}
\def\vecteur{%
  \ifnexttok[% si la macro est suivie d'un crochet
    \vecteur@i% aller à la macro à arguments délimités
    {\vecteur@i[1]}% sinon, ajouter l'argument optionnel par défaut entre crochets
}
\def\vecteur@i[#1]{%
  \ifnexttok(% si une parenthèse vient ensuite
    {\vecteur@ii[#1]}% aller à la macro à argument délimités
    {\vecteur@ii[#1](x)}% sinon, rajouter "x" comme argument optionnel
}
\def\vecteur@ii[#1](#2)#3{% #1 et #2=arg optionnel  #3=arg obligatoire
  \forcemath{({#2}_{#1},\ldots,{#2}_{#3})}%
}
\catcode`@12
1) \vecteur{n}\qquad 2) \vecteur[0]{k}\qquad
3) \vecteur[i](y)j \qquad4) \vecteur[](\alpha)n
```

1) $(x_1, \ldots, x_n)$ 2) $(x_0, \ldots, x_k)$ 3) $(y_i, \ldots, y_j)$ 4) $(\alpha, \ldots, \alpha_n)$

■ **Exercice 103**

La macro \vecteur ci-dessus admet deux arguments optionnels, mais ceux-ci doivent se trouver dans le bon ordre : celui entre crochets d'abord puis celui entre parenthèses et enfin l'argument obligatoire. On peut écrire \vecteur[i](y){j}, mais pas \vecteur(y)[i]{j}.

Comment faudrait-il modifier la macro pour que les arguments optionnels puissent être tapés dans n'importe quel ordre ?

□ **Solution**

Le plus simple est d'aiguiller l'algorithme vers deux « branches » selon que la macro \vecteur est suivie d'un crochet ou d'une parenthèse. Ces deux branches seront contenues dans les macros \vecteur@bracket et \vecteur@paren.

Code n° IV-353

```
\catcode`@11
\def\vecteur{%
  \ifnexttok[% si la macro est suivie d'un crochet
    \vecteur@bracket% lire cet argument entre crochet
    {\ifnexttok(% sinon, si elle est suivie d'une parenthèse
      \vecteur@paren % lire cet argument entre parenthèses
      {\vecteur@i[1](x)}% sinon, transmettre les arguments par défaut
    }%
}
\def\vecteur@bracket[#1]{%
  \ifnexttok(% s'il y a une parenthèse après
    {\vecteur@i[#1]}% lire la parenthèse
    {\vecteur@i[#1](x)}% sinon, donne "x" comme argument par défaut
}
\def\vecteur@paren(#1){%
  \ifnexttok[% si le caractère suivant est un crochet
    {\vecteur@ii(#1)}% lire l'argument entre crochets
    {\vecteur@ii(#1)[1]}% sinon, donner "1" comme argument par défaut
}
\def\vecteur@i[#1](#2)#3{\forcemath{({#2}_{#1},\ldots,{#2}_{#3})}}
\def\vecteur@ii(#1)[#2]{\vecteur@i[#2](#1)}% met les arg optionnels dans l'ordre
\catcode`@12
1) \vecteur{n}\qquad 2) \vecteur[i](y){j}\qquad 3) \vecteur(y)[i]{j}\qquad
4) \vecteur[0]{k}\qquad 5) \vecteur(\alpha){n}
```

1) $(x_1, \ldots, x_n)$ 2) $(y_i, \ldots, y_j)$ 3) $(y_i, \ldots, y_j)$ 4) $(x_0, \ldots, x_k)$ 5) $(\alpha_1, \ldots, \alpha_n)$

■

3.2. Définir une macro à plusieurs arguments optionnels

La tentation est grande de créer une macro qui étendrait les possibilités de la primitive \def et qui permettrait de définir des macros avec un ou plusieurs arguments optionnels [2]. Afin de rester simple, nous allons devoir faire quelques sacrifices : les macros ainsi définies ne pourront pas être à arguments délimités et les arguments optionnels seront entre crochets. Bien entendu, le nombre total d'arguments sera inférieur ou égal à 9.

Voici la syntaxe que nous pouvons nous fixer pour la macro \newmacro* :

2. Comme le fait la macro \newcommand de LaTeX, même si elle ne permet de définir qu'un seul argument optionnel.

\newmacro\⟨*macro*⟩⟨*paramètres*⟩{⟨*code*⟩}

Les ⟨*paramètres*⟩ sont composés soit de chiffres, soit de textes entre crochets. Les chiffres désignent le nombre d'arguments obligatoires requis à cette position et les crochets un argument optionnel dont la valeur par défaut est le code qui se trouve entre les crochets. Par exemple, si on écrit :

\newmacro\foo1[xxx]2[y]1{⟨*code*⟩}

alors, les paramètres « 1[xxx]2[y]1 » spécifient que la macro \foo admet 6 arguments qui seront dans l'ordre :

- un argument obligatoire ;
- un argument optionnel dont la valeur par défaut est « xxx » ;
- deux argument obligatoires ;
- un argument optionnel dont la valeur par défaut est « y » ;
- un argument obligatoire.

Le ⟨*code*⟩ de la macro pourra contenir les arguments sous la forme habituelle #1, #2, etc.

3.2.1. Cas particulier

Écrire d'emblée la macro \newmacro est un défi vraiment difficile. Essayons plutôt de construire à la main la macro \foo pour bien en comprendre le fonctionnement et ensuite, nous écrirons une macro généraliste \newmacro qui bâtira automatiquement des macros admettant plusieurs arguments optionnels.

Il est bien évident que notre macro \foo ne va pas tenir en une seule macro. En réalité, à chaque fois que nous devons lui faire lire un argument optionnel, il va falloir s'assurer que le token suivant est un crochet ou pas. Par conséquent, à ces occasions, la macro \foo va passer la main à d'autres macros auxiliaires \foo@i, \foo@ii, etc.

Voici le code qui permet de construire la macro \foo :

Code n° IV-354

```
\catcode`\@11
\def\foo#1{% #1 -> lit le premier argument obligatoire
  \ifnexttok[% si le token suivant est un crochet
    {\foo@i{#1}}% aller à \foo@i qui va lire cet argument
    {\foo@i{#1}[xxx]}% sinon, transmettre [xxx] à foo@i
}

\def\foo@i#1[#2]#3#4{% lit l'arg obligatoire + arg optionnel + 2 arg obligatoires
  \ifnexttok[
    {\foo@ii{#1}[#2]{#3}{#4}}%
    {\foo@ii{#1}[#2]{#3}{#4}[y]}%
}

\def\foo@ii#1[#2]#3#4[#5]#6{% lit tous les arguments
  1="#1" 2="#2" 3="#3" 4="#4" 5="#5" 6="#6"%
}

\catcode`\@12
1) \foo{arg1}{arg2}{arg3}{arg4}\par
2) \foo{arg1}[OPT\_A]{arg2}{arg3}{arg4}\par
3) \foo{arg1}{arg2}{arg3}[OPT\_B]{arg4}\par
```

```
4) \foo{arg1}[OPT\_A]{arg2}{arg3}[OPT\_B]{arg4}\medbreak
```

```
1) 1="arg1" 2="xxx" 3="arg2" 4="arg3" 5="y" 6="arg4"
2) 1="arg1" 2="OPT_A" 3="arg2" 4="arg3" 5="y" 6="arg4"
3) 1="arg1" 2="xxx" 3="arg2" 4="arg3" 5="OPT_B" 6="arg4"
4) 1="arg1" 2="OPT_A" 3="arg2" 4="arg3" 5="OPT_B" 6="arg4"
```

On voit donc qu'à chaque fois que l'on attend un argument optionnel, il s'agit de tester la présence d'un crochet. Pour lire l'argument optionnel entre crochet, une nouvelle macro auxiliaire `\foo@`⟨*n*⟩ à argument délimité doit être créée. On remarque aussi que lorsque cette macro auxiliaire est appelée (lignes n^{os} 4-5 et lignes n^{os} 10-11), la *totalité* des arguments lus jusqu'à présent lui est transmise. À charge pour cette macro de les lire, de lire aussi l'argument optionnel pour lequel elle est créée, ainsi que d'éventuels arguments obligatoires à suivre.

Il est également très important de remarquer que si une macro créée par le truchement de `\newmacro` admet un ou plusieurs arguments optionnels, non seulement elle ne peut pas être purement développable à cause de l'emploi de `\futurelet`, mais on ne peut pas faire apparaitre son texte de remplacement avec des `\expandafter`. C'est la contrepartie qu'il faut concéder pour profiter de la souplesse des arguments optionnels.

3.2.2. Cas général

Pour construire la macro `\newmacro`, nous allons d'abord nous mettre au clair avec les noms des macros auxiliaires. Dans le cas particulier précédent, la macro `\foo` avait deux macros auxiliaires `\foo@i` et `\foo@ii`. Ces macros, si elles sont automatiquement créées par `\newmacro`, posent un problème si l'utilisateur veut en disposer pour créer de lui même d'autres macros auxiliaires. Nous allons donc prendre le parti que si la macro à définir est \⟨*macro*⟩, alors les macros auxiliaires seront de la forme \⟨*macro*⟩`@[`⟨*chiffre romains*⟩`]`. Pour ce faire, une macro purement développable `\macro@name{`⟨*nombre*⟩`}` se chargera de donner sous forme de tokens de catcode 12, le nom de la macro (sans caractère d'échappement) sous la forme ⟨*macro*⟩`@[`⟨*chiffre romains*⟩`]`.

Venons-en à la méthode proprement dite. Elle va consister à définir la macro fille `\macro@name` puis à parser les ⟨*paramètres*⟩ jusqu'à rencontrer une accolade ouvrante. Comme on l'a vu avec la macro `\foo` précédemment, chaque macro auxiliaire doit relire la totalité des arguments rencontrés jusqu'à présent en y ajoutant d'autres arguments. Le texte de paramètre de ces macros va donc s'allonger au fur et à mesure que les ⟨*paramètres*⟩ seront parsés. Rappelons-nous qu'avec la macro `\foo`, ce texte de paramètre était « `#1` » pour la macro principale puis « `#1[#2]#3#4` » pour `\foo@i` et enfin « `#1[#2]#3#4[#5]#6` » pour la troisième macro `\foo@ii`. Dans le cas général, nous allons stocker ce texte de paramètre dans le registre de tokens `\param@text`.

Lorsque les ⟨*paramètres*⟩ sont parsés, trois cas sont possibles selon le prochain token vu par `\futurelet` :

1. si c'est une accolade ouvrante (ou plus exactement un `\bgroup` comme le voit `\futurelet`), alors, c'est que le texte de paramètre a été entièrement lu ;
2. si c'est un crochet, il faut créer une sous-macro supplémentaire et ajouter à

\param@text « [#⟨x⟩] » où ⟨x⟩ est le numéro du prochain paramètre ;

3. dans les autres cas, il s'agit donc d'un chiffre (un unique token de 1 à 9) et il faut ajouter à \param@text « #⟨x⟩ », « #⟨x+1⟩ », etc., autant de fois que le spécifie le chiffre lu.

La dernière difficulté est d'appeler les macros auxiliaires tout en leur transmettant les arguments déjà lus sous la forme « {#⟨x⟩} » si c'est un argument obligatoire et sous la forme « [#⟨x⟩] » si c'est un argument optionnel. Comme les arguments obligatoires doivent être transmis entre accolades, le registre de tokens \param@text ne peut pas être utilisé. Il faut donc utiliser un autre registre de tokens \arg@text, mis à jour en même temps que \param@text et qui enveloppera d'accolades les arguments obligatoires.

Voici donc le code, plutôt difficile, de la macro \newmacro :

Code n° IV-355

```
\catcode`\@11
\newcount\macro@cnt% numéro à mettre dans le nom des sous macros
\newcount\arg@cnt% compte le nombre d'arguments
\newtoks\param@text% texte de paramètre des macros sous forme "#x" et/ou "[#x]"
\newtoks\arg@text% arguments sous forme "{#x}" et/ou "[#x]"

\def\newmacro#1{%
  % \macro@name construit le <nom> de la macro et éventuellement "@[<chiffre romain>]"
  \def\macro@name##1{\expandafter\gobone\string#1\ifnum##1>0 @[\romannumeral##1]\fi}%
  \macro@cnt=0 \arg@cnt=0 % initialise les compteurs
  \param@text{}\arg@text{}% vide les registres de texte de paramètre et d'argument
  \newmacro@i% va voir le prochain token
}

\def\newmacro@i{\futurelet\nxttok\newmacro@ii}% met le prochain token dans \nxttok...
% ...puis va à la macro :
\def\newmacro@ii{%
  \ifxcase\nxttok
    [\newmacro@optarg% si le prochain token est un crochet aller à \newmacro@optarg
    \bgroup% si c'est un accolade ouvrante
      % le texte de paramètre est fini et il faut définir la macro
      {\defname{\macro@name\macro@cnt\expandafter}%
      \the\param@text}% <- le {<code>} est juste après, il n'est pas encore lu
  \elseif% sinon, c'est donc un chiffre
      \newmacro@arg% aller à \newmacro@arg
  \endif
}

\def\newmacro@optarg[#1]{% lit la valeur par défaut de l'argument optionnel
  % Définit la macro \<nom>@[<nbre>] qui lit tous les arguments (optionnels ou pas)
  % jusqu'alors définis à l'aide de \param@text. Puis, cette macro testera si le prochain
  % token est un crochet
  \expandafter\edef\csname\macro@name\macro@cnt\expandafter\endcsname\the\param@text{%
    \noexpand\ifnexttok[%
      % si oui : la macro \<nom>@<nbr+1> le lira
      {\expandafter\noexpand\csname\macro@name{\numexpr\macro@cnt+1}\expandafter\endcsname
      \the\arg@text}%
      % si non : transmettre à \<nom>@<nbr+1> l'argument optionnel par défaut lu
      {\expandafter\noexpand\csname\macro@name{\numexpr\macro@cnt+1}\expandafter\endcsname
      \the\arg@text[\unexpanded{#1}]}%
  }%
  \advance\arg@cnt 1 % incrémenter le numéro d'argument
  % pour ajouter "[#<x>]" à \param@text et à \arg@text
  \eaddtotoks\param@text{\expandafter[\expandafter##\number\arg@cnt]}%
```

```
  \eaddtotoks\arg@text  {\expandafter[\expandafter##\number\arg@cnt]}%
  \advance\macro@cnt 1 % incrémenter le numéro de nom de macro
  \newmacro@i% va voir le token suivant
}

\def\newmacro@arg#1{% #1=nombre d'arguments obligatoires à ajouter
  % boucle qui ajoute "#<x>#<x+1>etc" dans \param@text
  % et "{#<x>}{#<x+1>}etc" dans \arg@text
  \ifnum#1>\z@ % tant qu'on n'a pas ajouté le nombre de #x nécessaire
    \advance\arg@cnt 1 % incrémenter le numéro d'argument
    % pour ajouter #x à \param@text et {#x} à \arg@text
    \eaddtotoks\param@text{\expandafter##\number\arg@cnt}%
    \eaddtotoks\arg@text  {\expandafter{\expandafter##\number\arg@cnt}}%
    \expandafter\newmacro@arg\expandafter{\number\numexpr#1-1\expandafter}% boucler
  \else% si les arguments sont tous ajoutés
    \expandafter\newmacro@i% lire le token suivant
  \fi
}
\catcode`\@12
\newmacro\foo 1[xxx]2[y]1{1="#1" 2="#2" 3="#3" 4="#4" 5="#5" 6="#6"}
a) \foo{arg1}{arg2}{arg3}{arg4}\par
b) \foo{arg1}[OPT\_A]{arg2}{arg3}{arg4}\par
c) \foo{arg1}{arg2}{arg3}[OPT\_B]{arg4}\par
d) \foo{arg1}[OPT\_A]{arg2}{arg3}[OPT\_B]{arg4}\medbreak

\meaning\foo\par
\expandafter\meaning\csname foo@[i]\endcsname\par
\expandafter\meaning\csname foo@[ii]\endcsname
```

```
a) 1="arg1" 2="xxx" 3="arg2" 4="arg3" 5="y" 6="arg4"
b) 1="arg1" 2="OPT_A" 3="arg2" 4="arg3" 5="y" 6="arg4"
c) 1="arg1" 2="xxx" 3="arg2" 4="arg3" 5="OPT_B" 6="arg4"
d) 1="arg1" 2="OPT_A" 3="arg2" 4="arg3" 5="OPT_B" 6="arg4"

macro:#1->\ifnexttok [{\foo@[i] {#1}}{\foo@[i] {#1}[xxx]}
macro:#1[#2]#3#4->\ifnexttok [{\foo@[ii] {#1}[#2]{#3}{#4}}{\foo@[ii] {#1}[#2]{#3}{#4}[y]}
macro:#1[#2]#3#4[#5]#6->1="#1" 2="#2" 3="#3" 4="#4" 5="#5" 6="#6"
```

Il est peut être utile de donner quelques explications sur le code :

- au chapitre des variables, le compteur `\macro@cnt` sert à `\macro@name` pour nommer les macros auxiliaires via `\romannumeral`. Ces numéros entre crochets viennent après « `@` » et sont « `i` », « `ii` », etc.
  Ensuite, le compteur `\arg@cnt` sert à compter les arguments. Il est utilisé dans la macro `\newmacro@arg` pour insérer dans les registres à tokens `\param@text` et `arg@text`, soit `[#`⟨*i*⟩`]`, soit `#`⟨*i*⟩, soit `{#`⟨*i*⟩`}` ;
- la macro `\newmacro` n'est qu'une macro chapeau qui se charge des initialisations ;
- ensuite, `\newmacro@i` va voir le prochain token avec un `\futurelet` ;
- ce token est testé par la macro `\newmacro@ii` qui décide ce qu'il faut faire selon les cas envisagés plus haut :
  - si ce token est un crochet, `\newmacro@optarg` se charge de construire une macro auxiliaire supplémentaire avec un `\edef` en prenant soin de ne pas développer toutes les séquences de contrôle qui ne doivent pas l'être. La primitive `\unexpanded` est employée pour l'argument entre crochets puisqu'il est susceptible de contenir un nombre inconnu de

tokens ;
- si ce token est un chiffre, \newmacro@arg lit ce chiffre et met en place une récursivité pour ajouter {#⟨x⟩} et #⟨x⟩ autant de fois qu'il le faut aux registres de tokens \arg@text et \param@text ;
- enfin, si ce token est une accolade ouvrante, la dernière macro avec le texte de paramètre complet est définie avec le ⟨*code*⟩ qui suit les ⟨*paramètres*⟩.

Afin que les choses soient le plus claires possible et même si c'est fastidieux, essayons de faire mentalement fonctionner \newmacro pour l'exemple

```
\newmacro\foo 1[xxx]2[y]1
```

Par souci de clarté, les crochets autour du chiffre romain figurant dans les noms des macros auxiliaires seront omis :

1. le chiffre 1 est détecté. Il est lu par \newmacro@arg qui ajoute « {#1} » et « #1 » à \arg@text et \param@text qui, étant initialisés à vide par \newmacro contiennent donc « {#1} » et « #1 » ;
2. un crochet est détecté et l'argument « [xxx] » est lu par \newmacro@optarg :
   - elle définit la macro \foo de cette façon :
   ```
   \def\foo#1{%
       \ifnexttok[%
           {\foo@i{#1}}
           {\foo@i{#1}[xxx]}%
   }
   ```
   - les registres \arg@text et \param@text sont mis à jour et contiennent « {#1}[#2] » et « #1[#2] »
3. le chiffre 2 est détecté. Il est lu par \newmacro@arg qui ajoute 2 arguments « {#3}{#4} » et « #3#4 » à \arg@text et \param@text qui contiennent désormais « {#1}[#2]{#3}{#4} » et « #1[#2]#3#4 » ;
4. un crochet est détecté et l'argument « [y] » est lu par \newmacro@optarg :
   - elle définit la macro \foo@i de cette façon :
   ```
   \def\foo@i#1[#2]#3#4{%
       \ifnexttok[%
           {\foo@ii{#1}[#2]{#3}{#4}}
           {\foo@ii{#1}[#2]{#3}{#4}[y]}%
   }
   ```
   - les registres \arg@text et \param@text sont mis à jour et contiennent « {#1}[#2]{#3}{#4}[#5] » et « #1[#2]#3#4#5 »
5. le chiffre 1 est détecté. Il est lu par \newmacro@arg qui ajoute « {#6} » et « #6 » à \arg@text et \param@text ;
6. une accolade ouvrante est lue par \futurelet ce qui signe la fin du processus. Le code

   ```
   \defname{\macro@name\macro@cnt\expandafter}\the\param@text}
   ```

   se développe en

   ```
   \def\foo@ii#1[#2]#3#4#5#6
   ```

   Ceci est suivi du ⟨*code*⟩ entre accolades « {|#1|#2|#3|#4|#5|#6|} » toujours non lu ce qui signifie que la macro \foo@ii aura ce ⟨*code*⟩ comme

texte de remplacement.

Cet algorithme ne définit donc une macro *que* lorsqu'un crochet est détecté dans les ⟨*paramètres*⟩ ou lorsque l'accolade ouvrante marquant la fin des ⟨*paramètres*⟩ est atteinte.

■ Exercice 104

Créer une macro `\framebox`⋆ dont la syntaxe est

`\framebox[`⟨*lettres*⟩`]{<texte}`

et qui agit comme `\frbox` mais ne trace les réglures que selon les lettres qui figurent dans l'argument optionnel. La lettre « U » (comme « *up* ») demandera à ce que la réglure supérieure soit tracée. De la même façon « D », « L » et « R » spécifieront respectivement que les réglures inférieures, gauche et droite soient tracées. Par défaut, l'argument optionnel vaut « ULRD ».

□ Solution

Il faut reprendre la code de `\frbox` et tester avec `\ifin` s'il y a lieu de tracer une réglure et l'espace qui la sépare du ⟨*texte*⟩. Afin que les ⟨*lettres*⟩ de l'argument optionnel puisse être minuscules ou majuscules, la primitive `\uppercase` sera chargée de mettre en majuscule

`\ifin{#1}`⟨*lettre*⟩

Comme cette primitive est sans effet sur les séquences de contrôle, seuls l'argument optionnel `#1` et la ⟨*lettre*⟩ seront mis en majuscule.

Code n° IV-356

```
\newmacro\framebox[ULRD]1{% #1 = ULRD (Up, Down, Right, Left)
% ne pas changer le mode H ou V en cours
  \hbox{% enferme dans une \hbox
    \uppercase{\ifin{#1}L}{\vrule width\frboxrule}{}% réglure gauche
    \vtop{%
      \vbox{% 1er élément de la \vtop
        \uppercase{\ifin{#1}U}{% si la réglure sup doit être tracée
          \hrule height\frboxrule% réglure supérieure
          \kern\frboxsep% espace haut
          }
          {}%
        \hbox{%
          \uppercase{\ifin{#1}L}{\kern\frboxsep}{}% espace gauche
          #2% contenu
          \uppercase{\ifin{#1}R}{\kern\frboxsep}{}% espace droite
          }%
      }% puis autres éléments de la \vtop, sous la ligne de base
      \uppercase{\ifin{#1}D}{%
        \kern\frboxsep% espace bas
        \hrule height\frboxrule% réglure inférieure
        }%
        {}%
    }%
    \uppercase{\ifin{#1}R}{\vrule width\frboxrule}{}% réglure droite
  }%
}
\frboxsep=1pt
Boite \framebox{entière}, \framebox[ud]{Up down}, \framebox[LR]{Left Right},
\framebox[LU]{Left Up} et \framebox[rd]{Right Down}.
```

Boite entière, Up down, |Left Right|, Left Up et Right Down|.

■

3.3. Développer les arguments d'une macro

Pour rendre l'emploi des macros encore plus agréable et souple, nous devons maintenant chercher, lorsqu'elles sont exécutées, à développer comme on le souhaite leurs arguments. Nous nous limiterons aux *vrais* arguments et laisserons de côté les arguments optionnels.

Certes, nous avons déjà à notre disposition de `\exparg` (et `\expsecond` qui lui est `\let`-égale) ainsi que `\exptwoargs`, mais ces macros ne peut 1-développer qu'un seul ou deux arguments. Imaginons qu'une \⟨*macro*⟩ admette 5 arguments et qu'un appel à cette macro nécessite de lui passer le premier tel quel, de 1-développer le premier token du 2e argument, de 2-développer celui du 3e, de 3-développer celui du 4e et de développer au maximum le dernier argument. Nos macros `\exparg` et `\expsecond` seraient bien faibles pour y parvenir. Bâtissons une macro `\eargs`⋆, admettant un argument obligatoire entre crochets contenant, pour chaque argument, le nombre de développements souhaités. Si un développement maximal est voulu, un « + » sera spécifié. Pour notre macro \⟨*macro*⟩, voici la syntaxe que l'appel à cette macro prendrait :

`\eargs[0123+]\`⟨*macro*⟩`{`⟨*arg1*⟩`}{`⟨*arg2*⟩`}{`⟨*arg3*⟩`}{`⟨*arg4*⟩`}{`⟨*arg5*⟩`}`

Du côté de la TEXnique, `\eargs` lira son argument entre crochet et la \⟨*macro*⟩. Les autres argument seront lus un par un par une macro récursive. Cette macro se chargera de développer correctement l'argument courant et d'ajouter le résultat à un registre de tokens `\eargs@toks` qui jouera le rôle d'un accumulateur. Ce registre étant initialisé avec \⟨*macro*⟩ par `\eargs`, il contiendra à la fin cette \⟨*macro*⟩ et les arguments correctement développés.

Voici l'algorithme que nous mettrons en œuvre :

Macro `\eargs`

```
macro eargs[#1]#2% #1= liste des n-développements  #2=macro à exécuter
    eargs@toks ← #2% mettre la macro dans le collecteur de tokens
    eargs@i#1nil% appeler eargs@i avec la liste de n-développements
fin

macro eargs@i#1nil% #1=liste des n-développements
    si #1 est vide
        exécuter eargs@toks
    sinon
        eargs@ii#1nil% appeler eargs@ii avec la liste de n-développements
    finsi
fin

macro eargs@ii#1#2nil#3
%#1=n-développement actuel %#2=n-développements restants #3=argument courant
    si #1="+"% si un développement maximal est demandé
        macro@temp ← développement maximal de #3
    sinon
        macro@temp ← #3% mettre l'argument dans une macro
        pour i=1 to #1 faire% 1-développer #1 fois macro_temp
            macro@temp ← 1-développement du contenu de macro@temp
        finpour
    finsi
    eargs@toks ← eargs@toks+{macro@temp}% ajouter le résultat au registre
    eargs@i#2nil% recommencer avec la liste des n-développements restants
fin
```

Pour nous assurer que \eargs fonctionne bien, la macro \foo se contente ici d'afficher les arguments qu'elle reçoit tels quels avec \detokenize.

Code n° IV-357

```
\catcode`\@11
\newtoks\eargs@toks
\newtoks\eargs@temptoks
\def\eargs[#1]#2{% #1=liste des développements  #2=macro
  \eargs@toks{#2}% mettre la macro dans le collecteur de tokens
  \expandafter\eargs@i\detokenize{#1}\@nil% appeler \eargs@i avec
                                         % la liste des developpements
}

\def\eargs@i#1\@nil{% #1=liste des n-développements restant
  \ifempty{#1}% s'il n' y plus de n-développements
    {\the\eargs@toks}% exécuter la macro et ses arguments développés
    {\eargs@ii#1\@nil}% sinon appeller la macro qui lit un argument
}

% #1=n-développement actuel  #2=liste des n-développements restants   #3=argument lu
\def\eargs@ii#1#2\@nil#3{%
  \if+#1% si #1="+", un \edef est demandé pour cet argument
    \edef\eargs@tempmacro{{#3}}% le stocker dans une macro temporaire
  \else% sinon
    \eargs@temptoks={#3}% stocker l'argument dans un registre temporaire
    \for\eargs@loop = 1 to #1\do 1 % faire #1 fois :
      {\eargs@temptoks=% 1-développer le 1er token du registre temporaire
        \expandafter\expandafter\expandafter{\the\eargs@temptoks}%
      }% puis le stocker dans la macro temporaire
    \edef\eargs@tempmacro{{\the\eargs@temptoks}}%
  \fi
  \eaddtotoks\eargs@toks\eargs@tempmacro% ajouter le contenu de la macro au collecteur
  \eargs@i#2\@nil% appeler \eargs@i avec les n-développements restants
}
\catcode`\@12
\def\foo#1#2#3#4#5{\detokenize{1="#1" 2="#2" 3="#3" 4="#4" 5="#5"}}
\def\aaa{\bbb}\def\bbb{\ccc}\def\ccc{Bonjour}

\eargs[0123+]\foo{\aaa\bbb}{\aaa\bbb}{\aaa\bbb}{\aaa\bbb}{\aaa\bbb}.
```

1="\aaa \bbb " 2="\bbb \bbb " 3="\ccc \bbb " 4="Bonjour\bbb " 5="BonjourBonjour".

La ligne n° 26 rhabille d'accolades l'argument dument développé et, comme cela a été expliqué à la page 119, le \edef ne *développe pas* le contenu du registre.

Cinquième partie

Aller plus loin

Sommaire

Le moment est venu de mettre en application toutes les connaissances et les méthodes de programmation vues dans les parties précédentes. Un effort a été fait pour rendre certains de ces exemples aussi proches que possibles de ceux que l'on pourrait rencontrer dans une vie d'utilisateur de TEX, même si leur complexité peut pour certains sembler rebutante, mais il en est ainsi, plus on s'approche de *vrais* exemples, plus le niveau tend à s'élever.

Chapitre 1

Du nouveau dans les arguments des macros

1.1. Détecter des marqueurs dans l'argument d'une macro

1.1.1. Créer un effet entre deux marqueurs

Qui n'a jamais souhaité mettre des « marqueurs » dans l'argument d'une macro spécialement conçue, de telle sorte qu'entre deux de ces marqueurs, un effet spécial soit obtenu ? Si l'on appelle `\detectmark*` cette macro, on pourrait par exemple écrire

```
\detectmark{+}{Un +argument+ où les +marqueurs+ sont détectés}
```

pour que ce qui est entre deux « + » subisse un traitement que l'on peut choisir. On obtiendrait par exemple

Un **argument** où les **marqueurs** sont détectés

ou

Un [argument] où les [marqueurs] sont détectés

Le but est bien sûr de créer un raccourci pour passer de +⟨*code*⟩+ à {\bf⟨*code*⟩} ou à \frbox{⟨*code*⟩} ou à tout autre effet.

Pour programmer la macro `\detectmark`, l'idée est d'agir dans un groupe, de rendre actif son premier argument qui est le caractère « marqueur », et de programmer ce caractère pour obtenir l'effet souhaité. Nous utiliserons ici l'astuce du

\lccode et \lowercase, sous réserve que ~ soit actif au moment de la définition. Ceci fait, il suffira de faire lire le deuxième argument et ensuite, fermer le groupe.

Pour davantage de souplesse, la programmation de l'effet sera confiée à une macro externe \markeffect* qui recevra dans son argument ce qui se trouve entre les deux marqueurs.

Code n° V-358

```
\catcode`\@11
\def\detectmark#1{% #1 est le marqueur
  \begingroup
    \catcode`#1=13 % rendra #1 actif après la macro
    \begingroup% pour les besoins du \lccode
      \lccode`\~=`#1 % transforme "~" en " #1 actif"
      \lowercase{\endgroup\def~##1~}{\markeffect{##1}}%
    \detectmark@i
}
\def\detectmark@i#1{%
    #1% exécute le code
  \endgroup% ferme le groupe, le marqueur perd son catcode actif
}
\catcode`\@12
a) \def\markeffect#1{{\bf #1}}% met en gras
\detectmark+{Un +argument+ où les +marqueurs+ sont détectés}
\medskip

b) \def\markeffect#1{% met dans une boite
  \begingroup
    \frboxsep=1pt % modifie l'espacement entre texte et encadrement
    \frbox{\strut#1}% encadre
  \endgroup
}
\detectmark|{Un |argument| où les |marqueurs| sont détectés}
\medskip

c) \def\markeffect#1{$\vcenter{\hbox{#1}\hbox{#1}}$}% superpose 2 fois
\detectmark`{Un `argument` où les `marqueurs` sont détectés}
```

a) Un **argument** où les **marqueurs** sont détectés

b) Un argument où les marqueurs sont détectés

c) Un $\vcenter{\hbox{argument}\hbox{argument}}$ où les $\vcenter{\hbox{marqueurs}\hbox{marqueurs}}$ sont détectés

La macro \detectmark est certes bien pratique, mais elle doit cependant être utilisée avec précaution. Tout d'abord parce qu'en l'état, elle ne fonctionnera pas dans l'argument d'une macro :

Code n° V-359

```
\catcode`\@11
\def\detectmark#1{%
  \begingroup
    \catcode`#1=13 % rendra #1 actif après la macro
    \begingroup% pour les besoins du \lccode
      \lccode`\~=`#1 % transforme "~" en " #1 actif"
      \lowercase{\endgroup\def~##1~}{\markeffect{##1}}%   %
    \detectmark@i
}
\def\detectmark@i#1{%
```

```
    #1% lit le code
  \endgroup% ferme le groupe, le marqueur perd son catcode actif
}
\catcode`\@12
\def\markeffect#1{{\bf #1}}% met en gras
\frbox{\detectmark+{Un +argument+ où les +marqueurs+ sont détectés}}
```

Un +argument+ où les +marqueurs+ sont détectés

La raison est toujours la même, mais il faut la rappeler une fois de plus : la macro `\frbox` *lit* son argument (pour le traiter par la suite) et dès lors, les catcodes de tous les tokens qui le composent sont figés. Par conséquent, la macro interne `\detectmark`, qui agit *après* cette congélation des catcodes, ne peut plus modifier celui de + pour le rendre actif.

Mais il existe une autre limitation : nous aurions aussi pu rêver que la macro `\markeffect` agisse comme `\litterate` de la page 81 c'est-à-dire qu'elle affiche le code tel qu'il est écrit. Là aussi, l'échec est assuré, et toujours pour la même raison. En effet, le marqueur dans la macro `\detectmark`, après être rendu actif, a été programmé *lire* ce qui se trouve jusqu'au prochain marqueur. Ce marqueur transmet tout ce qu'il a lu à la macro `\markeffect` qui, à son tour, le *lit* avant de le traiter. Ce qui est entre deux marqueurs est donc lu *deux fois* avant d'être traité. La conséquence est la même que s'il y avait eu une seule lecture : les catcodes des tokens se trouvant entre deux marqueurs sont figés et il n'est plus possible de les neutraliser avec `\dospecials`.

1.1.2. Autoriser du verbatim dans l'argument d'une macro

Est-il possible de trouver une méthode permettant de faire lire une portion de texte en « verbatim » dans l'argument d'une macro ? C'est-à-dire que les tokens faisant partie de cette portion seraient lus par la macro comme étant tous de catcode 12. La réponse est oui, mais le procédé va être assez TEXnique et de toute façon, la solution aura des limitations.

Décidons pour cela de créer une macro `\alter*` dont la syntaxe serait la suivante :

`\alter`⟨*délimiteur*⟩`\`⟨*macro*⟩`{`⟨*argument*⟩`}`

La `\`⟨*macro*⟩ serait altérée en ce sens que si des ⟨*délimiteurs*⟩ sont présents par paires dans son ⟨*argument*⟩, alors tout ce qui est entre deux délimiteurs consécutifs serait compris comme étant en mode « verbatim ». Par exemple, écrire

```
\alter{|}\frbox{texte normal |$\ver {# batim$| texte normal}
```

produirait à l'affichage

texte normal $\ver {# batim$ texte normal

Pour parvenir à nos fins, le principe général est de collecter dans un registre de tokens `\alter@toks` tous les tokens de l'⟨*argument*⟩ en prenant soin de modifier à 12 le catcode de tous les tokens qui sont entre deux occurrences du ⟨*délimiteur*⟩. Une fois ceci fait, il ne reste plus qu'à donner à la `\`⟨*macro*⟩ le contenu du registre de tokens.

La macro \alter va effectuer les actions suivantes : elle va lire deux arguments qui sont le délimiteur et la \⟨*macro*⟩ à altérer et va les stocker pour les utiliser ultérieurement. C'est ensuite qu'elle va effectuer une action que nous n'avons jusqu'ici jamais faite : elle va manger l'accolade ouvrante de l'⟨*argument*⟩ avec un \let\next=␣ précédé d'un \afterassignment qui spécifiera où aller après avoir mangé cette accolade. Dès lors, le problème qui surgit est que l'accolade qui ferme l'⟨*argument*⟩ se retrouve orpheline et il faudra bien prendre en compte et gérer cette rupture d'équilibre à un moment. Mais poursuivons notre chemin : puisque \alter est entré à l'intérieur de l'⟨*argument*⟩, elle va le parcourir token par token (comme le faisait la macro \parse) et décider l'action à faire en fonction du prochain token à lire (obtenu avec \futurelet). Voici les cas à envisager :

- accolade fermante ;
- accolade ouvrante ;
- espace ;
- un ⟨*délimiteur*⟩ ;
- un autre token.

Précisons que la macro \ifbracefirst ne doit pas être utilisée pour tester si une accolade ouvrante est à lire, car cette macro lit la totalité de son argument pour savoir si celui-ci commence par une accolade. Cette lecture provoque des pertes d'informations (lire page 68) et le gel des catcodes de tous les tokens, ce qui l'exact contraire de ce qui est recherché ici.

Comme l'⟨*argument*⟩ est parcouru avec \futurelet, les limitations de la commande \alter apparaissent donc : il faut que l'⟨*argument*⟩ ne contienne ni \bgroup ni \egroup, car cela provoquerait des faux positifs par le test \ifx avec les accolades ouvrantes et fermantes.

La macro \alter va être construite sur le modèle de la macro \parse de la page 367 et, comme elle doit aussi effectuer son travail à l'intérieur de groupes entre accolades, elle va emprunter la méthode de la macro étoilée \parse* de la page 371. Une macro de fin de processus va être programmée et localement modifiée lorsqu'un groupe entre accolades est rencontré.

Ceci donne l'algorithme suivant :

1) définir la macro de fin \alter@stop
    a) lire l'accolade fermante orpheline « } » ;
    b) passer à l'argument de \⟨*macro*⟩ le contenu de \alter@toks
2) lire les deux arguments ⟨*délimiteur*⟩ et \⟨*macro*⟩
3) manger l'accolade ouvrante qui suit
4) initialiser \alter@toks à vide
5) lire avec \futurelet le prochain token \alter@nxttok
6) tester \alter@nxttok
    a) si c'est une accolade fermante, la fin de l'argument est donc atteinte, aller à la macro de fin \alter@stop
    b) si c'est un espace : absorber cet espace et ajouter un espace à \alter@toks
    c) si c'est une accolade ouvrante
        i) ouvrir un groupe
        ii) modifier localement la macro de fin \alter@stop pour qu'elle rajoute à \alter@toks le contenu local de \alter@toks enveloppé d'accolades puis qu'elle ferme le groupe et aille au point n° 5

iii) aller au point n° 3

d) si c'est ⟨*délimiteur*⟩ : ouvrir un groupe semi-simple, changer les catcodes de tous les octets pour 12, ajouter tous les tokens jusqu'à la prochaine occurrence de ce délimiteur après avoir fermé le groupe (toutes ces opérations sont faites par une macro à arguments délimités)

e) dans les autres cas, le token à lire ne demande aucune action particulière : le manger et l'ajouter à `\alter@toks`

7) dans tous les cas ci-dessous sauf le c, aller au point n° 5

Les cas n° b, c et e ont déjà été vus avec la macro `\parse`. Ici, il n'y a donc que deux cas supplémentaires à traiter, les cas n° a et d.

Code n° V-360

```
\catcode`\@11
\newtoks\alter@toks% collecteur de tokens

\def\alter#1#2{% #1= délimiteur  #2 = macro à altérer
  \let\alter@macro#2% sauvegarde la macro
  \edef\alter@restorecatcode{% restaurera le catcode de #1
    \catcode`\noexpand#1=\the\catcode`#1 }%
  \edef\alter@tmp{\let\noexpand\alter@markertoks= \string#1}%
  \alter@tmp% et sauvegarder le délimiteur après avoir mis son catcode à 12
  \edef\alter@tmp{\def\noexpand\alter@readlitterate@i\string#1####1\string#1}%
  % développe les \string#1 pour que les arguments délimités aient
  % des délimiteurs de catcode 12
  \alter@tmp{% <- comme si on écrivait "\def\alter@readlitterate@i#1##1#1"
    \endgroup% après avoir lu ##1 (tokens rendus inoffensifs), fermer le groupe
    \addtotoks\alter@toks{{\tt##1}}% ajouter ces tokens
    \alter@i% et aller lire le prochain token
  }%
  \alter@toks{}% initialise le collecteur de tokens
  \afterassignment\alter@i% aller lire le premier token après avoir
  \let\alter@tmptok= % mangé l'accolade ouvrante de l'<argument> qui suit
}

\def\alter@i{% lit le prochain token et va à \alter@ii
  \futurelet\alter@nxttok\alter@ii}%

\def\alter@ii{% teste le token qui doit être lu
  \ifxcase\alter@nxttok% si le token à lire est
    \egroup           \alter@stop% "}" : aller à \alterstop@i
    \sptoken          \alter@readspc% " " : aller à \alter@readspc
    \bgroup           \alter@readarg% "{" : aller à \alter@readarg
    \alter@markertoks \alter@readlitterate% "<delimiteur>" : aller à \alter@readlitterate
  \elseif
    \alter@readtok% dans les autres cas, aller à \alter@readtok
  \endif
}

\def\alter@readlitterate{% le prochain token est le délimiteur
  \begingroup% ouvrir un groupe
  \for\alter@tmp=0to255\do{\catcode\alter@tmp=12 }%
  % mettre tous les catcodes à 12
  \defactive{ }{\ }% sauf l'espace rendu actif
  \doforeach\alter@tmp\in{<,>,-,`,{,},'}% pour chaque motif de ligature
      {\unless\if\alter@tmp\alter@markertoks% s'il est différent du délimiteur
        % le rendre actif pour éviter la ligature
        \expandafter\alter@defligchar\alter@tmp
      \fi
```

```
      }%
  \alter@readlitterate@i% puis aller à \alter@readlitterate@i...
  % ...qui a été définie dans \alter
}

\def\alter@defligchar#1{% définit le caractère pour ne pas provoquer de ligature
  \defactive#1{\string#1{}}%
}

\expandafter\def\expandafter\alter@readspc\space{% mange un espace dans le code
  \addtotoks\alter@toks{ }% ajoute l'espace
  \alter@i% puis lire le token suivant
}

\def\alter@readarg{% le token qui suit est "{"
  \begingroup% ouvrir un groupe
  \def\alter@stop@ii{% et modifier localement la macro appelée à la toute fin,
  % après que l'accolade fermante ait été mangée (par \alterstop@i)
    \expandafter\endgroup% retarder la fermeture de groupe ouvert ci-dessus
    \expandafter\addtotoks\expandafter\alter@toks\expandafter
      {\expandafter{\the\alter@toks}}%
    % pour ajouter hors du groupe ce qui a été collecté à l'intérieur,
    % le tout mis entre accolades
    \alter@i% puis, lire le token suivant
  }%
  \alter@toks{}% au début du groupe, initialiser le collecteur
  \afterassignment\alter@i% aller lire le prochain token après
  \let\alter@tmptok= % avoir mangé l'accolade ouvrante
}

\def\alter@readtok#1{% le prochain token ne demande pas une action spéciale
  \addtotoks\alter@toks{#1}% l'ajouter au collecteur
  \alter@i% et aller lire le token suivant
}

\def\alter@stop{% le token à lire est "}"
  \afterassignment\alter@stop@ii% aller à \alter@stop@ii après
  \let\alter@tmptok= % avoir mangé l'accolade fermante
}

\def\alter@stop@ii{% donner à la \<macro> tout ce qui a été récolté
  \expandafter\alter@macro\expandafter{\the\alter@toks}%
  \alter@restorecatcode% puis restaure le catcode du délimiteur
}
\catcode`@12

\frboxsep=1pt
\alter|\frbox{Texte normal - |#& }| - texte normal - |_^ ##| - texte normal}
\alter=\frbox{La macro =\alter= autorise du verbatim dans
des commandes imbriquées \frbox{comme ici =\alter=}.}
```

Texte normal - #& } - texte normal - _^ ## - texte normal

La macro `\alter` autorise du verbatim dans des commandes imbriquées comme ici `\alter`.

Nous avons considéré que la \⟨*macro*⟩ à altérer n'avait qu'un seul argument. Or, il serait commode de pouvoir altérer les macros à plusieurs arguments de façon à mettre du verbatim dans n'importe lequel de leurs arguments. C'est ici que la macro `\identity` (voir page 71) peut résoudre ce problème. Il suffira de placer la

macro à altérer ainsi que ses arguments dans l'argument de `\identity` :

Code n° V-361

```
\def\hello#1#2{Bonjour #1 et  #2 !}
\hello{foo}{bar}\par
\alter|\identity{\hello{macro |{{{\foo|}{macro |\bar}|}}
```

Bonjour foo et bar !
Bonjour macro {{{\foo et macro \bar} !

1.2. Modifier les catcodes des arguments déjà lus

La règle de la page 28 était pourtant claire : dès que TEX lit un token, il lui affecte de façon inaltérable un code de catégorie. Il était même précisé que ce mécanisme est incontournable. Le titre de cette section pose donc des interrogations quant au moyen utilisé.

1.2.1. Technique de « écriture-lecture »

Ce moyen consiste à enregistrer des tokens dans un fichier auxiliaire puis lire ce fichier. En effet, lorsqu'on enregistre du code dans un fichier, celui-ci, après éventuel développement, est « détokénisé », c'est-à-dire que n'en subsistent que les caractères (au sens d'octet) qui le formaient. Les catcodes, caractéristiques de TEX, sont perdus à l'enregistrement. Par la suite, lors de la lecture d'un fichier, les octets qui le composent sont lus par TEX et transformés en tokens selon le régime de catcode en cours à ce moment.

Cette astuce redonne à n'importe quel ensemble de tokens une sorte de virginité puisqu'on peut les relire à n'importe quel moment – y compris dans le texte de remplacement d'une macro – pour qu'ils se colorent des catcodes en vigueur.

On peut l'observer en utilisant la macro `\frbox` en mettant dans son argument macro `\retokenize`⋆ qui enregistre les tokens de son argument (déjà lus par la macro `\frbox`) dans un fichier puis les relit :

Code n° V-362

```
\def\retokenize#1{%
  \immediate\openout\wtest=retokenize.tex % ouvre le fichier
  \immediate\write\wtest{\unexpanded{#1}}%  y écrit l'argument
  \immediate\closeout\wtest% ferme le fichier
  \input retokenize.tex % lit le fichier selon les catcodes en vigueur
  \unskip% mange l'espace précédemment ajouté qui provient de la fin du fichier
}
\frboxsep=1pt
1) \frbox{Programmer en \catcode`\~=12 \TeX{} est~facile et~utile.}\par
2) \frbox{Programmer en \retokenize{\catcode`\~=12 \TeX{} est~facile} et~utile.}\par
3) \frbox{Programmer en \catcode`\~=12 \retokenize{\TeX{} est~facile} et~utile.}
```

1) Programmer en TEX est facile et utile.
2) Programmer en TEX est~facile et utile.
3) Programmer en TEX est~facile et utile.

Le cas n° 1 montre bien que « ~ » ne change pas de catcode dans l'argument de la macro `\frbox` puisqu'il ne prend jamais le catcode 12 pour donner l'affichage

« ~ ». Il ne le prend pas non plus à l'extérieur du texte de remplacement de cette macro, car \frbox enveloppe son argument dans des boites qui jouent le rôle de groupe.

Au contraire, les cas n^os 2 et 3 mettent en évidence que la relecture du fichier « retokenize.tex » se fait avec les catcodes en cours. Le fichier est lu avec ~ de catcode 12 pour le cas n^o 2 tandis que pour le cas n^o 3, le changement de catcode est inclus dans le fichier ce qui ne change rien au résultat final.

Il faut bien avoir conscience que la relecture du fichier se fait *après* que l'argument de \frbox ait été lu. Cette lecture d'argument s'accompagne d'irrémédiables pertes d'information (par exemple des espaces consécutifs lus comme un seul espace). De plus, la lecture d'un fichier provoque une altération qui lui est propre : si ce n'est pas déjà le cas, un espace est inséré après toute séquence de contrôle lue dans le fichier. Cet espace ajouté n'a aucune incidence si le code est *exécuté* car les espaces sont ignorés lorsqu'ils suivent une séquence de contrôle. Le seul cas où cet espace peut devenir gênant est lorsqu'on compose du « verbatim » en provenance d'un fichier. L'exemple montre avec \litterate la double altération que subit le code : la première rend plusieurs espaces consécutifs égaux à un seul espace et l'autre insère un espace après la macro \TeX. Les deux arguments des cas n^o 1 et 2, bien que et présentant plusieurs différences, sont rendus de la même façon :

Code n^o V-363

```
\frboxsep=1pt
1) \frbox{Programmer \retokenize{\litterate|en \TeX {} est |}facile et utile.}\par
2) \frbox{Programmer \retokenize{\litterate|en    \TeX{} est |}facile et utile.}
```

1) Programmer en \TeX {} est facile et utile.
2) Programmer en \TeX {} est facile et utile.

1.2.2. La primitive \scantokens

Le moteur ε-TEX dispose de la primitive \scantokens{⟨*texte*⟩} qui agit comme la macro \retokenize avec quelques petites différences :

- l'écriture a lieu dans un fichier *virtuel*, c'est-à-dire qu'aucun fichier n'est créé sur le disque dur et en réalité, tout se passe dans la mémoire de ε-TEX ;
- la primitive \scantokens est développable et son 1-développement est l'ensemble des tokens qui sont dans son argument, remis au gout du régime de catcode en cours ;
- comme toutes les primitives devant être suivies d'une accolade, \scantokens procède à un développement maximal jusqu'à rencontrer l'accolade ouvrante.

Code n^o V-364

```
\frboxsep=1pt
1) \frbox{Programmer en \catcode`\~=12 \TeX{} est~facile et~utile.}\par
2) \frbox{Programmer en \scantokens{\catcode`\~=12 \TeX{} est~facile} et~utile.}\par
3) \frbox{Programmer en \catcode`\~=12 \scantokens{\TeX{} est~facile} et~utile.}\par
4) \frbox{Programmer \scantokens{\litterate|en \TeX {} est facile|} et utile.}\par
5) \frbox{Programmer \scantokens{\litterate|en    \TeX{} est facile|} et utile.}
```

1) Programmer en TEX est facile et utile.
2) Programmer en TEX est~facile et utile.
3) Programmer en TEX est~facile et utile.

4) Programmer en \TeX {} est facile et utile.
5) Programmer en \TeX {} est facile et utile.

■ **Exercice 105**

Expliquer d'où vient l'espace en trop entre les mots « facile » et « et » des cas n° 2-5 de l'exemple précédent ainsi que l'espace parasite entre lettres « a » et « b » ci-dessous :

Code n° V-365

```
\scantokens{a}b
```

a b

□ **Solution**

Il provient du caractère de fin de ligne qui est inséré à la fin de chaque ligne. Ici, le fichier virtuel est constitué d'une seule ligne contenant « a » et le caractère de code \endlinechar est inséré à la fin de cette ligne. Par défaut, ce caractère est « ^^M », de catcode 5, qui est lu comme un espace. On peut donc définir localement \endlinechar comme entier négatif pour le neutraliser :

Code n° V-366

```
\begingroup\endlinechar=-1 \scantokens{a}b\endgroup
```

ab

■

1.2.3. \scantokens : erreurs en vue

Le fonctionnement est clair, mais une erreur surprenante est émise lorsque l'on tente de développer \scantokens dans le texte de remplacement d'une macro :

Code n° V-367

```
\edef\foo{\scantokens{Bonjour le monde}}% produit une erreur
```

```
! File ended while scanning definition of \foo.
```

Si l'on réitère l'expérience en ne la développant qu'une seule fois, la même erreur survient :

Code n° V-368

```
\expandafter\def\expandafter\foo\expandafter
  {\scantokens{Bonjour le monde}}% produit une erreur
```

```
! File ended while scanning definition of \foo.
```

Il arriverait une erreur similaire si l'on écrivait :

```
\toks0=\expandafter{\scantokens{Bonjour le monde}}
```

et on obtiendrait « `! File ended while scanning text of \toks.` »

Comme \scantokens agit un peu comme \input, il en partage aussi les secrets (voir page 294). Le lecteur qui avait sauté la lecture de cette section ou qui n'en a plus qu'un vague souvenir est invité à s'y replonger. L'essentiel à retenir est que la

fin d'un fichier – virtuel ou pas – est « \outer ». Ici, TEX se plaint, car cette fin de fichier se trouve à des endroits qui lui sont interdits par son statut d'\outer.

Le remède pour la fin d'un fichier virtuel \outer est le même que pour la fin d'un fichier réel présent dans le texte de remplacement d'une macro : il suffit de placer un \noexpand juste avant la fin de ce fichier et de s'assurer qu'il sera développé *avant* que la définition de la macro ne se fasse. Ceci implique donc qu'il *faut* utiliser la macro \edef :

Code n° V-369

```
\edef\foo{\scantokens{Bonjour le monde\noexpand}}
Voici la macro \string\foo\ : \foo.
```

Voici la macro \foo : Bonjour le monde.

Utiliser \edef comporte l'inconvénient de développer la totalité de l'argument alors que finalement, seul le \noexpand final aurait besoin d'être développé. Comme nous l'avions fait avec \input, il est possible de construire une macro \scandef⋆ ayant la syntaxe suivante

\scandef\⟨*macro*⟩{⟨*texte*⟩}

et qui met le 1-développement de \scantokens{⟨*texte*⟩} dans le texte de remplacement de \⟨*macro*⟩. Nous allons utiliser le même artifice qu'avec \input (voir page 295) et nous servir de \everyeof pour placer un \@nil à la fin du fichier virtuel. Ce \@nil sera utilisé par une macro auxiliaire comme délimiteur d'argument :

Code n° V-370

```
\catcode`\@11
\def\scandef#1#2{% #1=\<macro>   #2=<texte>
  \begingroup
    \endlinechar=-1 % pas de caractère de fin de ligne
    \everyeof{\@nil#1\noexpand}% ajoute "\@nil\<macro>\noexpand" avant la fin du fichier
    \expandafter\scandef@i\expandafter\relax\scantokens{#2}%
}
\def\scandef@i#1\@nil#2{% "\@nil#2" ont été ajouté par \everyeof
  \endgroup% ferme le groupe
  \expandafter\def\expandafter#2\expandafter{\gobone#1}% et définit la \<macro>
}
\catcode`@12
\def\foo{%
  Dans tout l'argument de \string\foo, <<~>> est actif
  sauf
  \catcode`\~12
  \scandef\bar{dans celui de \string\bar : <<~>>}%
  \catcode`~13
  \bar
}
\foo
```

Dans tout l'argument de \foo, « » est actif sauf dans celui de \bar : «~»

Toute l'astuce réside dans « \everyeof{\@nil#1\noexpand} » qui demande que soit ajouté avant la fin du fichier un \@nil et la \⟨*macro*⟩ (qui est #1), le tout suivi de \noexpand. Par conséquent, la ligne

\expandafter\scandef@i\scantokens{⟨*texte*⟩}

se développe en

\scandef@i⟨*texte relu par \scantokens*⟩\@nil\⟨*macro*⟩\noexpand⟨*EOF*⟩

et la macro à argument délimité \scandef@i n'a plus qu'à saisir les arguments délimités pour, une fois sorti du groupe, définir la \⟨*macro*⟩ sans jamais avoir à lire la fin du fichier ⟨*EOF*⟩ qui de toute façon, est neutralisée par \noexpand.

La chronologie qui se met en place lorsque \scantokens{⟨*code*⟩} est exécuté est la suivante :

- insertion du caractère de code \endlinechar après tous les retours à la ligne, y compris après la dernière ;
- transformation du code qui en résulte en tokens ;
- insertion du code contenu dans \everyeof à la fin de ce qui va être lu par \scantokens.

On peut constater que le point n° 1 est bien celui qui est exécuté en premier avec ce code assez amusant où la lettre « d » de \noexpand est ajoutée par \endlinechar :

Code n° V-371

```
{%
\endlinechar=`\d% insère la lettre "d" à chaque fin de ligne
\edef\foo{\scantokens{Bonjour le monde\noexpan}}% le \noexpan(d) est incomplet
Voici la macro \string\foo\ : \foo.% fins de ligne...
}% ...commentées pour éviter le "d"
```

Voici la macro \foo : Bonjour le monde.

■ Exercice 106

Utiliser \scantokens pour programmer une macro \cprotect* (le « c » signifiant « catcode »), dont la syntaxe est

\cprotect\⟨*macro*⟩{⟨*argument*⟩}

et qui appelle la \⟨*macro*⟩ de cette façon :

\⟨*macro*⟩{\scantokens{⟨*argument*⟩\noexpand}}

Comme l'⟨*argument*⟩ est relu par \scantokens, l'intérêt de cette macro est qu'elle permet d'utiliser \litterate dans l'⟨*argument*⟩. De façon plus générale, \cprotect permet de changer des catcodes dans l'⟨*argument*⟩ de la \⟨*macro*⟩.

□ Solution

La primitive \scantokens va évidemment considérablement simplifier la méthode, mais va aussi changer les limites de la macro comme nous le verrons, de telle sorte que \cprotect ne peut pas être qualifié d'équivalent de la macro \alter vue précédemment.

Tout d'abord, nous allons lire l'⟨*argument*⟩ après avoir rendu les catcodes de tous les octets égaux à 12, sauf celui de l'accolade ouvrante et de l'accolade fermante qui gardent leurs catcodes de 1 et 2 afin justement de pouvoir délimiter l'argument et de rendre possible sa lecture. La suite est simple : l'argument ainsi lu sera retokénisé par \scantokens et donné comme argument à la \⟨*macro*⟩.

Code n° V-372

```
\catcode`\@11
\def\cprotect#1{% #1 est la \<macro>
  \def\cprotect@i##1{% ##1 est l'<argument>
    \endgroup% ferme le groupe précédemment ouvert
```

```
    #1{\scantokens{##1\noexpand}}%
  }%
  \begingroup% rend tous les octets de catcode12
    \for\cprotect@temp=0to255\do{\catcode\cprotect@temp=12 }%
    \catcode`\{=1 \catcode`\}=2 % sauf "{" et "}"
    \cprotect@i% puis, lit l'argument
}
\catcode`@12
\frboxsep=1.5pt
1) \cprotect\frbox{foo \litterate-&#  #^   ^_%- bar}\par
2) \cprotect\frbox{\catcode`\~=12 a~b~c~d}\par
3) \cprotect\frbox{foo \litterate-\bar- \cprotect\frbox{\litterate-&#   #-} fin}
```

1) foo &# #^ ^_% bar
2) a~b~c~d
3) foo \bar &# # fin

La macro \cprotect, malgré sa grande simplicité, présente trois limitations principales :

1. l'⟨*argument*⟩ ne doit pas contenir d'accolade ouvrante ou fermante non équilibrée, et donc le texte traité par \litterate doit également obéir à cette contrainte. Y contrevenir tromperait TEX sur la portée de l'⟨*argument*⟩ car lors de la lecture de cet ⟨*argument*⟩, les accolades ont leurs catcodes normaux.
2. \cprotect ne peut agir que sur une macro admettant un seul argument puisque \cprotect@i a été définie comme lisant et traitant un unique argument. Il n'est pas possible de contourner cette limitation comme on le faisait avec \identity pour la macro \alter. En revanche, il aurait été possible de programmer la macro \cprotect différemment afin de mettre entre les accolades la liste des arguments de la \⟨*macro*⟩ :

   \cprotect\⟨*macro*⟩{{argument 1}{argument 2}...}

3. on a considéré que l'⟨*argument*⟩ était entre accolades, sans envisager le cas d'autres caractères de catcode 1 ou 2. Pour bien faire, on n'aurait dû changer le catcode d'un caractère que s'il est différent de 1 ou 2. La boucle \for aurait donc dû être programmée ainsi :

```
\for\cprotect@temp=0to255\do{%
    \unless\ifnum\catcode\cprotect@temp=1
        \unless\ifnum\catcode\cprotect@temp=2
            \catcode\cprotect@temp=12
        \fi
    \fi}
```

Selon les besoins et les limitations dont on veut s'affranchir, il convient donc de s'orienter vers \cprotect ou \alter. ■

■ Exercice 107

Comment modifier le code de \cprotect pour permettre à la \⟨*macro*⟩ d'avoir plusieurs arguments et offrir la syntaxe qui est donnée au point n° 2 de la solution précédente ?

□ Solution

Appelons \Cprotect* cette macro. La stratégie consiste à parcourir la liste des arguments, et les stocker sous la forme

{\scantokens{⟨*argument*⟩\noexpand}}

dans un registre de tokens qui accumulera ces arguments de telle sorte qu'à la fin du processus, ce registre contienne

\⟨*macro*⟩{\scantokens{#1\noexpand}}{\scantokens{#2\noexpand}}etc...

Une fois tout ceci fait, le contenu du registre sera exécuté avec \the.

La principale difficulté est de parcourir tous les arguments et gérer d'une façon ou d'une autre la fin de la liste. Ici, le quark \quark est mis comme dernier argument. Une macro récursive lira un argument à la fois et avant d'agir, fera un test pour déterminer si tous les arguments ont été traités (cas où l'argument lu est \ifx-égal à \quark). Il faut également tenir compte du fait que chaque argument est dépouillé de ses éventuelles accolades lorsqu'il est lu par une macro.

Le registre de token n° 0 tiendra lieu d'accumulateur.

Code n° V-373

```
\catcode`@11
\def\Cprotect#1{% #1 est la \<macro>
  \def\Cprotect@i##1{% ##1 est la liste des arguments
    \endgroup% ferme le groupe précédemment ouvert
    \toks0={#1}% met la \<macro> dans le registre de tokens
    \Cprotect@ii##1\quark% \quark est mis à la fin des arguments
    \the\toks0 % exécute le registre
  }%
  \begingroup% rend tous les octets de catcode12
    \for\temp@arg= 0 to 255 \do{% changer à 12 tous les catcodes
      \unless\ifnum\catcode\temp@arg=1 % sauf si catcode=1
        \unless\ifnum\catcode\temp@arg=2 % ou 2
          \catcode\temp@arg=12
        \fi
      \fi}%
    \Cprotect@i% puis, lit l'argument
}
\def\Cprotect@ii#1{% #1 est l'argument courant (dépouillé de ses accolades)
  \def\temp@arg{#1}% stocke l'argument pour le tester ci dessous :
  \unless\ifx\quark\temp@arg% si la fin n'est pas atteinte
    % ajouter "{\scantokens{#1\noexpand}}" au registre
    \addtotoks{\toks0}{{\scantokens{#1\noexpand}}}%
    \expandafter\Cprotect@ii% et recommencer
  \fi
}
\catcode`@12

\def\test#1#2{Bonjour #1 et #2}
\Cprotect\test{{argument 1 : \litterate-\foo-}{argument 2 : \litterate-\bar-}}
```

Bonjour argument 1 : \foo et argument 2 : \bar

■

Chapitre 2

Aller plus loin avec des réglures

Les réglures, qui ne sont par définition que de simples rectangles noirs peuvent, moyennant un peu d'imagination, être utilisées pour donner lieu à des effets plutôt originaux…

2.1. Créer une police Impact

2.1.1. Analyse du problème

Le but ici est de créer de toutes pièces une « police » de type `Impact` comme il était courant d'en rencontrer du temps où les imprimantes à aiguilles (ou matricielles) étaient encore largement majoritaires. Leur fonctionnement est d'une grande simplicité : la tête d'écriture est constituée d'une ligne de fins picots – les aiguilles –, tous jointifs, alignés verticalement et actionnables indépendamment les uns des autres. Au fur et à mesure que la tête d'écriture se déplace par rapport à la page, les aiguilles sont brièvement actionnées, chacune imprimant un pixel lorsqu'elle exerce une pression sur le ruban encreur qui se trouve entre elle et la page. Voici l'effet que l'on obtient en rendant visibles les espaces entre les aiguilles :

La tête de lecture a deux caractéristiques qui lui sont données lors de sa fabrication : sa hauteur et le nombre d'aiguilles. Plus les aiguilles sont en grand nombre par unité de longueur, plus la résolution est importante et meilleure est la qualité d'impression.

Précisons tout de suite que nous allons *dessiner* des caractères de toutes pièces, mais que *stricto sensu*, nous n'allons pas créer une « police » de caractères. En effet, toutes les fonctionnalités avancées des *vraies* polices ne seront pas présentes ici :

- crénage [1] entre caractères ;
- ligatures ;
- coupures des mots, même si les coupures n'ont rien à voir avec les polices, mais sont faites par TEX en fonction de la position de certains groupes de lettres rencontrés dans le mot ;
- différentes formes (italique, petites majuscules, etc.), graisse.

Il s'agit donc d'un abus de langage, certes volontaire, mais pas réellement malhonnête puisqu'on pourra dessiner les caractères que l'on veut et écrire de vraies phrases avec les dessins créés.

2.1.2. Empilement de pixels

Avec TEX, ce qui va nous tenir lieu de point élémentaire et que nous dénommerons « *pixel*, est une simple réglure carrée. Contrairement aux imprimantes, nous n'imposerons aucune limite sur le nombre de pixels qui composent un caractère, pas plus en hauteur, profondeur qu'en largeur, la seule limite étant finalement la patience de l'utilisateur à créer un caractère fait d'un grand nombre de pixels. Pour dessiner un pixel et correctement le positionner par rapport aux autres, il nous faut définir deux dimensions, `\pixelsize`⋆ qui est le côté du carré et `\pixelsep`⋆ qui est l'espacement entre deux pixels adjacents, aussi bien horizontalement ou verticalement.

Autant commencer par le plus facile et définir la macro `\pixel` qui imprime un pixel carré en mode horizontal, de telle sorte que le bord inférieur du carré coïncide avec la ligne de base :

```
\def\pixel{\vrule height\pixelsize width\pixelsize depth0pt }
```

Pour corser un peu plus l'exemple qui va suivre, on peut essayer d'empiler verticalement deux lignes de pixels au-dessus de la ligne de base. Puisqu'on est au-dessus de cette ligne de base, on va naturellement le faire dans une `\vbox` et enfermer dans une `\hbox` chaque ligne de pixels après l'avoir convenablement séparée de la précédente. Pour respecter l'espacement vertical entre les deux lignes de pixels, il suffira d'annuler le ressort d'interligne avec `\offinterlineskip` puis de régler ce ressort d'interligne à `\pixelsep` :

Code n° V-374

```
\newdimen\pixelsize \newdimen\pixelsep
\def\pixel{\vrule height\pixelsize width\pixelsize depth0pt }
\def\vblankpixel{\vrule height\pixelsize width0pt depth0pt }
\def\blankpixel{\vblankpixel \vrule height0pt width\pixelsize depth0pt }
\def\gap{\kern\pixelsep}
\pixelsize=3pt \pixelsep=1pt
Essai :
\vbox{% aligne verticalement
```

1. C'est la possibilité de modifier (souvent réduire) l'espacement entre deux caractères en fonction de leur forme de telle sorte que l'ensemble soit plus esthétique. Ainsi « AVATAR » avec crénage est bien plus esthétique que « AVATAR » sans crénage. Le mot avec crénage a une longueur de 28.67998pt alors que le mot sans crénage mesure 31.368pt.

```
  \offinterlineskip% annule le ressort d'interligne
  \lineskip=\pixelsep\relax% pour le mettre à \pixelsep
  \hbox{\pixel\gap\blankpixel\gap\blankpixel\gap\blankpixel\gap\pixel}% 1re ligne
  \hbox{\pixel\gap\pixel     \gap\blankpixel\gap\pixel     \gap\pixel}% 2e ligne
}
```

Essai : 

Les deux autres macros \blankpixel et \vblankpixel nous seront utiles par la suite et laissent un pixel blanc pour la première et un strut de la hauteur d'un pixel pour la seconde. La macro \gap insère une espace insécable de la valeur de \pixelsep et a vocation à être insérée horizontalement entre deux pixels consécutifs, quelle que soit leurs couleurs.

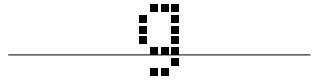
Le code est clair et on comprend vite le principe. On voit ici que les pixels dessinés pourraient former le haut de la lettre

Et l'on pourrait facilement écrire les lignes supplémentaires nécessaires pour afficher cette lettre en entier.

Remarquons que l'ordre d'affichage des pixels n'est pas celui des imprimantes à aiguilles : le caractère n'est pas imprimé par une succession de lignes de pixels verticales. Bien sûr, on pourrait procéder ainsi, mais les boites de TEX nous facilitent la tâche si l'on s'y prend autrement. On va plutôt imprimer de haut en bas toutes les lignes de pixels (elles-mêmes imprimées de gauche à droite) nécessaires à la formation d'un caractère.

2.1.3. Prise en compte de la ligne de base

Il faut maintenant s'intéresser au respect de la ligne de base si nous voulons imprimer un caractère comme : il faut que le bas des pixels formant la partie inférieure de la boucle de la lettre « » coïncide exactement avec la ligne de base :



Pour cela, la même méthode que celle de \frbox (voir page 273) va être déployée : une \vbox sera insérée au sommet d'une \vtop, ce qui revient à utiliser ce schéma :

```
\vtop{%
    \offinterlineskip \lineskip=\pixelsep
    \vbox{⟨lignes de pixels au-dessus de la ligne de base⟩}%
    ⟨lignes de pixels au-dessous de la ligne de base⟩%
    }
```

Voici un exemple où nous appliquons ce schéma pour construire la lettre « » :

Code n° V-375

```
\pixelsize=3pt \pixelsep=1pt
Essai :
\vtop{%
  \offinterlineskip \lineskip=\pixelsep\relax
  \vbox{%
```

```
    \hbox{\blankpixel\gap\pixel     \gap\pixel     \gap\pixel}%         ***
    \hbox{\pixel     \gap\blankpixel\gap\blankpixel\gap\pixel}%        *  *
    \hbox{\pixel     \gap\blankpixel\gap\blankpixel\gap\pixel}%        *  *
    \hbox{\pixel     \gap\blankpixel\gap\blankpixel\gap\pixel}%        *  *
    \hbox{\blankpixel\gap\pixel     \gap\pixel     \gap\pixel}%         ***
  }%                                                  ----ligne de base
  \hbox  {\blankpixel\gap\blankpixel\gap\blankpixel\gap\pixel}%           *
  \hbox  {\blankpixel\gap\pixel     \gap\pixel               }%         **
}
```

Essai : 

La constatation qui s'impose est que définir un caractère avec cette méthode est extrêmement fastidieux et il n'est pas acceptable de l'utiliser pour définir toutes les lettres de l'alphabet.

2.1.4. Créer une macro dont la syntaxe est facile

La macro \makecar

Syntaxiquement, l'idéal serait d'inventer un moyen pour écrire les lignes de pixels « visuellement » les unes à la suite des autres. Il faudrait donc bâtir une macro \makecar* qui ressemblerait à ceci

```
\makecar\⟨macro⟩{%
 ***
*  *
*  *
*  *
 ***_
   *
 **,}
```

et qui stockerait dans \⟨*macro*⟩ le code exposé dans l'exemple ci-dessus qui permet de tracer la lettre « ». Le caractère « _ » signifie, s'il est présent, que la ligne de pixels en cours s'achève et que les lignes suivantes doivent se trouver sous la ligne de base.

Pour s'offrir cette syntaxe, la première chose à faire est de forcer TEX à faire prendre en compte les espaces lorsqu'ils sont consécutifs ou placés en début de ligne. Ceci nous contraindra à changer localement le code de catégorie de l'espace à 12 pour qu'il échappe à la règle qui lui est fixée lorsque ce catcode est 10. Le retour charriot sera rendu actif et se développera en une virgule pour pouvoir utiliser par la suite la macro \doforeach.

Enfin, il faut déterminer si « _ » est présent ou pas et si c'est le cas, scinder l'argument #2 en deux parties, chacune représentant les lignes de pixels au-dessus et au-dessous de la ligne de base.

La macro \makecar@i

Une macro privée \makecar@i traitera une de ces parties en transformant « * » en \pixel, « ␣ » en \blankpixel* et en ajoutant \kern \pixelsep après chacune de

ces macros, tout en insérant à la toute fin \unkern pour annuler le dernier espace interpixel.

Par le truchement de \doforeach, cette macro \makecar@i appellera une autre macro auxiliaire \makecar@ii qui sera chargée de transformer chaque ligne de pixels (une suite de * et ␣) en une série de \pixel, \blankpixel et \gap*. Elle enfermera le tout dans une \hbox et ajoutera le code ainsi obtenu à une macro \pixabove qui jouera le rôle de collecteur et qui sera affichée dans la \vtop chapeau à la toute fin. L'écriture du code de cette macro privée ne présente pas de difficulté mais fait entrer en jeu quelques astuces.

Code n° V-376

```
\catcode`\@11
\begingroup% dans ce groupe :
\catcode`\ =12\relax% l'espace devient un "caractère autre"
\catcode`\^^M=13\relax% le retour à la ligne est actif
\edef^^M{\string,}% et se développe en une virgule (de catcode 12)
\global\deftok\otherspc{ }% définit un espace de catcode 12
\xdef\letter@code{% macro contenant le dessin de la lettre "e"
 **
*  *
***
*
 ***}%
\endgroup

\def\makecar@i#1{% #1 = dessin de la lettre avec les caractères "," "*" et " "
  \doforeach\current@line\in{#1}% pour chaque ligne dans #1 :
    {\ifx\empty\current@line% si la ligne est vide
      \addtomacro\pixabove{\hbox{\vblankpixel}}% ajouter une fausse ligne
    \else% sinon
      \let\pix@line\empty% initialiser le code de la ligne à vide
      \expandafter\makecar@ii\current@line\quark% et la construire
    \fi
    }%
}

\def\makecar@ii#1{% #1=caractère de dessin de la ligne en cours
  \ifxcase#1% si le caractère est
    *        {\addtomacro\pix@line\pixel}%
    \otherspc{\addtomacro\pix@line\blankpixel}%
  \endif
  \ifx#1\quark% si la fin est atteinte
    \addtomacro\pix@line\unkern% annuler le dernier espace interpixel
    \eaddtomacro\pixabove{% et encapsuler \pix@line dans une \hbox
    \expandafter\hbox\expandafter{\pix@line}}%
  \else% si la fin n'est pas atteinte, ajouter l'espace interpixel
    \addtomacro\pix@line\gap
    \expandafter\makecar@ii% recommencer avec le caractère suivant
  \fi
}
\pixelsize=3pt \pixelsep=1pt
\let\pixabove\empty% initialisation de la macro finale à vide
\exparg\makecar@i\letter@code% appelle la macro qui construit \pixabove
La lettre
\vtop{% enferme le tout dans une \vtop :
  \offinterlineskip\lineskip=\pixelsep% ajuste l'espace interligne
  \vbox{\pixabove}% affiche le caractère créé
}.
```

La lettre  .

La partie la plus intéressante se situe dans le groupe semi-simple au début (lignes n^os^ 2 à 13) où le code de catégorie de l'espace est changé à 12 et le celui du retour charriot est changé à 13 pour qu'il devienne actif. Cette altération de `^^M` suppose que toutes les fins de ligne où l'on ne souhaite pas voir ce `^^M` actif soient commentées, même celles que l'on ne commente habituellement pas (après une séquence de contrôle par exemple). Ces modifications locales faites, on peut donc tranquillement définir globalement la macro `\letter@code` qui contient le code qui dessine la lettre « e ». On profite de ce groupe pour définir avec `\deftok` la séquence de contrôle `\otherspc` qui sera `\let`-égale à un espace de catcode 12 et qui servira dans un test `\ifx` plus tard.

Retour sur la macro \makecar

On voit maintenant comment procéder lorsqu'il y a le marqueur « _ » qui indique que des pixels se trouvent au-dessous de la ligne de base. Il faut séparer le code définissant le caractère en deux parties et agir de la même façon avec `\makecar@i` sur chacune. Le code résultant de la première est voué à se retrouver dans la `\vbox` tandis que celui résultant de la seconde est simplement ajouté dans la `\vtop` chapeau.

Si une \⟨*macro de dessin*⟩ contient les éléments qui dessinent un caractère à l'aide de « * », « ␣ » et « ^^M » de catcode 12 et éventuellement « _ »), la macro `\makecar`, de syntaxe

\makecar\⟨*macro*⟩\⟨*macro de dessin*⟩

se chargera de stocker dans la \⟨*macro*⟩ le code résultant de l'analyse du texte de remplacement de \⟨*macro de dessin*⟩. Le texte de remplacement de \⟨*macro*⟩ sera constitué de `\pixel`, `\blankpixel` et `\gap` enfermés dans des `\hbox`, elles-mêmes incluses dans une `\vtop`.

Code n^o^ V-377

```
\catcode`\@11
\begingroup% dans ce groupe :
\catcode`\ =12\relax% l'espace devient un "caractère autre"
\catcode`\^^M=13\relax% le retour à la ligne est actif
\edef^^M{\string,}% et se développe en une virgule (de catcode 12)
\global\deftok\otherspc{ }% définit un espace de catcode 12
\xdef\letter@code{% macro contenant le dessin de la lettre "g"
 ***
*  *
*  *
*  *
 ***_
   *
 **}%
\endgroup

\def\makecar#1#2{% #1=nom recevant le code final  #2=macro contenant le dessin
  \let\pixabove\empty \let\pixbelow\empty \let\pix@line\empty% initialise à vide
  \exparg\ifin{#2}_% si le code contient _
    {\expandafter\makecar@iii#2\@nil}% aller à \makecar@iii
```

```
    {\exparg\makecar@i{#2}}% sinon, à \makecar@i
  \edef#1{% définit la macro #1 comme
    \vtop{% une \vtop contenant :
      \unexpanded{\offinterlineskip\lineskip\pixelsep}% réglage d'espace inter ligne
      \vbox{\unexpanded\expandafter{\pixabove}}% \vbox des pixels au-dessus
                                               % de la ligne de base
      \unless\ifx\pixbelow\empty% s'il y a des pixels au-dessous de la baseline
        \unexpanded\expandafter{\pixbelow}% les ajouter dans la \vtop
      \fi
      }%
    }%
}

\def\makecar@iii#1_,#2\@nil{%
  \makecar@i{#2}% construit la partie au-dessous de la baseline
  \let\pixbelow\pixabove% et affecte le code à \pixbelow
  \let\pixabove\empty \let\pix@line\empty% ré-initialise
  \makecar@i{#1}% construit la partie au-dessus de la baseline
}

\makecar\lettreg\letter@code
\catcode`\@12
\pixelsize=3pt \pixelsep=1pt

Essai : \lettreg
```

Essai : 

On voit clairement ce que fait la macro `\makecar@iii` ; elle construit séparément les lignes de pixels se trouvant au-dessous puis au-dessus de la ligne de base et stocke les codes obtenus dans les séquences de contrôle `\pixbelow` et `\pixabove`. La macro `\makecar` peut ensuite définir `#1` en incluant `\pixbelow` si elle n'est pas vide (cas où _ n'est pas présent). Cette définition se fait avec un `\edef` en prenant soin de protéger avec `\unexpanded` les endroits qui ne doivent pas être développés.

Définir toutes les lettres

L'étape suivante va être de finaliser le tout. Il faut d'abord trouver un moyen simple de définir toutes les lettres de l'alphabet. La macro `\impact@alphabet`, contenant le code de toutes les lettres, pourrait être définie ainsi :

```
\def\impact@alphabet{↵
a/↵
⟨dessin de la lettre a⟩,↵
b/↵
⟨dessin de la lettre b⟩,↵
etc↵
Z/↵
⟨dessin de la lettre Z⟩}
```

Pour que les choses soient parfaitement claires, les retours charriots ne faisant pas partie du dessin des lettres sont explicitement écrits « ↵ ». Il est important de noter que dans le texte de remplacement de `\impact@alphabet`, tous les `^^M` sont actifs, *mais non développés* en une virgule. Imaginons maintenant que l'on parcoure

ce texte de remplacement avec une boucle de type \doforeach :

\doforeach\letter@name/\letter@code\in
{⟨*texte de remplacement de \code@lettre*⟩}

À chaque itération, \letter@name va contenir

↵⟨*lettre*⟩

et \letter@code

↵⟨*dessin de la lettre*⟩

où les ↵ sont ceux qui ne font pas partie du dessin de la lettre. Si l'on développe au maximum \letter@name et \letter@code, tous les retours charriots vont céder leur place à une virgule, et les textes de remplacement de \letter@name et \letter@code seront :

,⟨*lettre*⟩ et ,⟨*dessin de la lettre*⟩

Il faudra donc tester si ces textes de remplacement commencent par une virgule et dans l'affirmative, la supprimer.

Code n° V-378

```
\catcode`\@11
\begingroup
\expandafter\gdef\csname impact@" "\endcsname{% définit la lettre "espace"
  \hskip 4\pixelsize plus.5\pixelsize minus.5\pixelsize\relax}%
\catcode`\^^M=13\relax% le retour à la ligne est actif
\edef^^M{\string,}% et se développe en une virgule (de catcode 12)
\catcode`\ =12\relax% l'espace devient un "caractère autre"
\gdef\impact@alphabet{
a/
***
   *
 ***
*  *
****,
b/
*
*
***
*  *
*  *
*  *
***,
% beaucoup de caractères omis
9/
 ***
*   *
*   *
 ****
    *
    *
 ***,
0/
 ***
*   *
*   *
* * *
*   *
```

```
*   *
 ***,
error/
* * *
 * *
* * *
 * *
* * *
 * *
* * *}% <- commenter la fin de ligne
\endgroup%
% On parcourt le texte de remplacement de \impact@alphabet
\edef\saved@crcatcode{\catcode13=\the\catcode13\relax}%
\catcode`\^^M=13\relax% le retour à la ligne est actif
\edef^^M{\string,}% et se développe en une virgule (de catcode 12)
\expsecond{\doforeach\letter@name/\letter@code\in}\impact@alphabet%
  {\edef\letter@name{\letter@name}% développe la lettre (^^M devient ",")
   \edef\letter@code{\letter@code}% développe le code (^^M devient ",")
   \exparg\ifstart\letter@name,% si la lettre commence par ","
     {\edef\letter@name{\expandafter\gobone\letter@name}}% la retirer
     {}%
   \exparg\ifstart\letter@code,% si le code commence par ","
     {\edef\letter@code{\expandafter\gobone\letter@code}}% la retirer
     {}%
   \expandafter\makecar\csname impact@"\letter@name"\endcsname\letter@code%
   }%
\saved@crcatcode% redonne le catcode de ^^M
% % définit l'espace
\pixelsize=3pt \pixelsep=1pt
% puis, on affiche ce qui a été défini :
a = \csname impact@"a"\endcsname\qquad b = \csname impact@"b"\endcsname\qquad
9 = \csname impact@"9"\endcsname \qquad 0 = \csname impact@"0"\endcsname\qquad
error = \csname impact@"error"\endcsname
```

a = b = 9 = 0 = error =

La méthode fonctionne correctement pour les caractères définis : bien évidemment, pour éviter un très long listing, seuls quelques uns sont définis ici.

Dans le code itéré de la boucle `\doforeach`, remarquons à la ligne n° 62 que `\makecar` est invoqué et la macro contenant le code qui dessine le caractère « ⟨*car*⟩ » est `\impact@"`⟨*car*⟩`"`. Notons également que l'espace est défini comme un ressort horizontal de dimension fixe égale à 4 fois celle de `\pixelsize` mais contenant aussi des composantes étirables égales à la moitié de `\pixelsize`. Celles-ci peuvent entrer en jeu pour justifier le texte à composer à la largeur disponible en comblant la différence entre la largeur des caractères affichés sur la ligne et la largeur de composition.

Un caractère « `error` » a également été dessiné au cas où l'on rencontrerait une lettre non définie.

La macro `\impact`

Tout cela va permettre de définir deux macros `\impact*` et `\impactend` qui vont composer en police `Impact` le ⟨*texte*⟩ se trouvant entre elles selon cette syntaxe :

```
\impact[1.2pt][0.2pt]Programmer en {TEX} est
facile et tr{e‘}s utile !\impactend
```

Tout d'abord, pour composer ce livre, un caractère de la police impact dont le nom est « TEX » a été défini et le fait qu'il soit écrit entre accolades assure que l'ensemble est bien lu comme un argument et compris comme le nom d'un caractère. Le même principe s'applique aux caractères accentués « `e‘` », « `e’` » et d'autres. Ensuite, la macro `\impact` a deux arguments optionnels, le premier étant la dimension qui sera assignée à `\pixelsize` et le second celle qui sera assignée à `\pixelsep`. La macro `\newmacro` que nous avons vue à la partie précédente sera utilisée afin que ces arguments optionnels soient facilement accessibles.

En reprenant la procédure mise en œuvre pour la macro `\parse`, nous allons parcourir ce qui se trouve après la macro `\impact` et, à l'aide de `\futurelet`, nous déciderons quelle action entreprendre selon le type de token à venir :

1. le quark `\impactend` ;
2. un espace ;
3. un argument commençant par une accolade.

Le premier cas stoppera la lecture du texte. Pour les deux derniers cas, on insèrera après chaque caractère une espace qui est en réalité le ressort `\letter@skip` défini par

```
\letter@skip=\pixelsep plus.1\pixelsize minus.1\pixelsize
```

Les composantes de compression et d'étirement de l'espace interlettre, cumulées avec celles du caractère intermot « `\letter@"␣"` », assurent que les avertissements de type `Overful/Underful hbox` qui caractérisent les lignes trop longues ou trop courtes seront le plus rare possible.

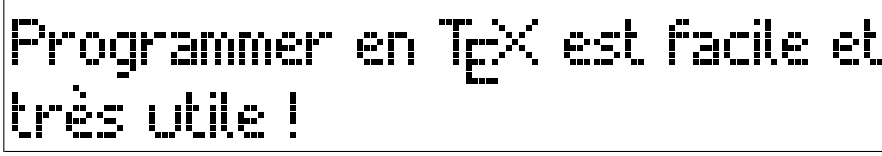
La façon de coder ce genre de macro a été vue à la partie précédente et il suffit de reproduire le schéma déjà étudié :

Code n° V-379

```
\catcode‘\@11
\newskip\letter@skip% ressort mis entre chaque lettre
\def\impactend{\impactend}% définit le quark de fin

\newmacro\impact[0.2ex][0pt]{% définit la macro à arguments optionnels
  \leavevmode% passer en mode horizontal
  \begingroup% dans un groupe semi-simple :
    \pixelsize=#1\relax \pixelsep=#2\relax% définir ces deux dimensions
    \letter@skip=#1 plus.1\pixelsize minus.1\pixelsize\relax% définir espace inter-lettre
    \baselineskip=% définir la distance entre les lignes de base
      \dimexpr12\pixelsize+7\pixelsep\relax% à 12\pixelsize+7\pixelsep
    \lineskiplimit=0pt % si lignes trop serrées
    \lineskip=2\pixelsize\relax% les espacer de 2\pixelsize
    \impact@i% puis aller voir le prochain token
}

% va voir le prochain token puis va le tester à \impact@ii
\def\impact@i{\futurelet\nxtletter\impact@ii}

\def\impact@ii{%
    \ifx\nxtletter\impactend% si le token est \let égal à \impactend
      \let\donext\impact@endprocess% aller à al macro de fin
    \else
      \ifx\nxtletter\sptoken% si c'est un espace
        \let\donext\impact@spc% aller à \impact@spc
```

```
      \else
        \let\donext\impact@arg% sinon, aller à \impact@arg
      \fi
    \fi
    \donext% faire l'action décidée ci-dessus
}

% mange un espace (argument délimité) et affiche "\letter@<spc>"
\expandafter\def\expandafter\impact@spc\space{%
    \csname impact@" "\endcsname
    \impact@i% puis va voir le prochain token
}
%
% lit l'argument suivant
\def\impact@arg#1{%
    \csname impact@% affiche
      \ifcsname impact@"#1"\endcsname\ifcsname
        "#1"% le caractère \impact@"#1" s'il est défini
      \else
        "error"% sinon \impact@"error"
      \fi
    \endcsname
    \hskip\letter@skip% insérer le ressort inter-lettre
    \impact@i% puis, aller voir le prochain token
}
\def\impact@endprocess\impactend{% macro exécuté lorsqque le quark \impactend va être lu
    \unskip% annuler le dernier ressort
    \par% composer le paragraphe pour prendre en compte
        % \baselineskip, \lineskip et \lineskiplimit
  \endgroup% et fermer le groupe
}
\catcode`\!=12 % rend le point d'exclamation "gentil"
\impact Programmer en {TEX} est facile et tr{e`}s utile !\impactend\par
\impact[2pt][0.5pt]Programmer en {TEX} est facile et tr{e`}s utile !\impactend
```

Programmer en TEX est facile et très utile !



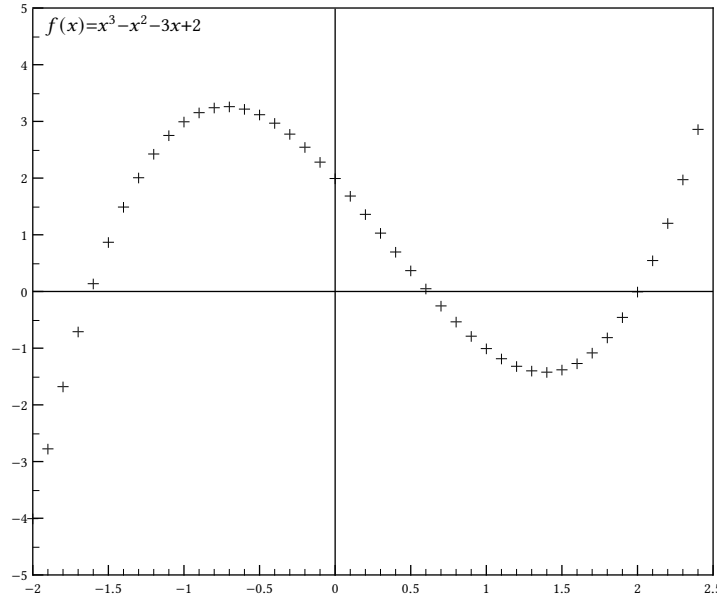
La valeur de \baselineskip s'explique par le fait que tels qu'ils sont dessinés, les caractères alphanumériques de la police impact ont au maximum 9 pixels de *hauteur totale*, où la hauteur totale est la somme de la hauteur et de la profondeur. L'encombrement vertical maximal d'un caractère est donc

$$9 \times \texttt{\pixelsize} + 7 \times \texttt{\pixelsep}$$

Pour augmenter un peu l'espacement entre les lignes, on rajoute à cette dimension $3 \times$ \pixelsize, ce qui donne la valeur de \baselineskip choisie pour composer le paragraphe composé en police impact.

2.2. Tracer des graphiques

Pourquoi vouloir tracer un graphique avec TEX alors que des extensions très puissantes [2] aux capacités quasi illimitées existent pour cela ? Il s'agit ici non pas de les concurrencer, mais de relever le défi et exploiter les maigres possibilités qu'offre TEX, c'est-à-dire les réglures. Le but est de comprendre comment les placer précisément dans une zone. Voici par exemple ce qu'il est possible de faire :

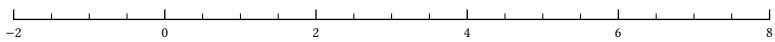


Le nombre de réglures est assez important et nous allons découper le problème en problèmes élémentaires avant de rassembler le tout à la fin.

La méthode qui va être notre fil directeur est de tracer *toutes* les réglures et *tous* les caractères après avoir rendu leurs dimensions nulles. Ainsi, pendant tout le tracé, le point de référence reste le point inférieur gauche qui a pour coordonnées dans cet exemple $(-2{,}5\ ;\ -5)$. C'est seulement à la fin que l'on forcera une boite à prendre celle du graphique afin que la boite englobante du tout coïncide avec le rectangle représentant la zone graphique.

2.2.1. Dessiner des axes gradués

Le premier problème est de tracer les axes gradués situés en bas et à gauche de la zone. Quels paramètres faut-il spécifier pour tracer un axe horizontal tel que celui-ci ?



Une simple observation permet de répondre. Il faut :

1. l'abscisse de départ et celle d'arrivée « `xmin` » et « `xmax` » (ici -2 et 8) ;
2. l'incrément « `inc` » entre 2 graduations principales (ici 2) ;
3. le nombre de subdivisions « `nb_subdiv` » (ici 4) ;
4. la distance « `dist` » entre deux graduations principales (ici 2 cm) ;

2. Les deux plus connues sont PStricks et Ti*k*z.

5. enfin, les dimensions des réglures proprement dites :
   - les dimensions verticales des traits représentant une division principale et une subdivision ;
   - les épaisseurs des traits représentant une division principale et une subdivision ;
   - l'épaisseur du trait horizontal représentant l'axe.

Nous allons décider que la macro `\xaxis*` va tracer un axe horizontal et que tous les arguments (sauf ceux du point n° 5) seront passés à la macro. Celle-ci aura 5 arguments (2 obligatoires et 3 optionnels) disposés selon cette syntaxe :

```
\xaxis[dist]{xmin}[inc]{xmax}[nb_subdiv]
```

Les 5 autres dimensions graphiques du point n° 5 seront contenues dans des registres de dimension :

- `\maingraddim` et `\subgraddim` pour les hauteurs des traits ;
- `\maingradwd` et `\subgradwd` pour leurs épaisseurs ;
- `\axiswd` pour l'épaisseur de l'axe.

Maintenant que la syntaxe est définie, passons au mécanisme de tracé proprement dit. Voyons d'abord comment tracer une graduation principale avec une abscisse située au-dessous. Pour le trait, nous utiliserons un `\rlap` pour que sa dimension soit nulle. Pour le nombre affiché au-dessous, il sera l'argument de la macro et sera également affiché avec la macro `\clap`, et abaissé de `1.5ex` à l'aide de la primitive `\lower` :

Code n° V-380

```
\newdimen\maingraddim \maingraddim=4pt % hauteur des graduations principales
\newdimen\maingradwd \maingradwd=0.5pt % épaisseur des graduations principales
\def\maingradx#1{%
  \lower1.5ex\clap{$\scriptscriptstyle#1$}% afficher l'argument au dessous
  \clap{\vrule height\maingraddim width\maingradwd depth0pt }% et la réglure
}
Début\maingradx{1}suite\maingradx{2}conclusion\maingradx{3}fin.
```

Début suite conclusion fin.
1 2 3

La longueur horizontale de l'axe sera

$$\texttt{dist} \times \frac{\texttt{xmax} - \texttt{xmin}}{\texttt{inc}}$$

Voici donc une ébauche de la macro `\xaxis` :

Code n° V-381

```
\maingraddim=4pt \maingradwd=0.5pt
\newdimen\axiswd \axiswd=0.5pt
\newmacro\xaxis[1cm]1[1]1[4]{%% #1= dist  #2=xmin  #3=inc  #4=xmax  #5=subdiv
  \hbox{% mettre le tout dans une \hbox
    \rlap{% en débordement à droite :
      \FOR\xx = #2 to #4 \do #3{% pour chaque graduation principale
        \maingradx{\xx}% tracer la graduation et écrire l'abscisse
        \kern#1\relax% puis se déplacer vers la droite
      }%
    }%
    \vrule% tracer l'axe des abscisses
      height\axiswd% d'epaisseur \axiswd, de longueur #1*(#4-#2)/#3
```

```
      width\dimexpr#1*\decdiv{\dimtodec\dimexpr#4pt-#2pt\relax}{#3}\relax
      depth 0pt\relax % et de profondeur nulle
  }%
}
a) \xaxis{-2}{5}suite

b) \frboxsep=0pt\frbox{\xaxis[1.25cm]{-1}[0.25]{1}}suite
```

a) −2 −1 0 1 2 3 4 5 suite

b) −1 −0.75 −0.5 −0.25 0 0.25 0.5 0.75 1 suite

Dans le deuxième cas, on constate en l'encadrant avec \frbox que l'encombrement de l'axe est bon. C'est ce qui était recherché puisque la seule chose tracée ayant une dimension horizontale non nulle est l'axe lui-même, c'est-à-dire la \vrule des lignes n^{os} 11-14.

Il nous reste à insérer dans chaque boucle les graduations secondaires qui sont au nombre de #5 − 1. Elles seront affichées en débordement à droite après chaque graduation principale, sauf pour la dernière. Préalablement à la boucle, elles seront stockées dans une \hbox via le registre de boite n^{o} 0. Dans cette \hbox, il doit y avoir #5 − 1 fois ceci :

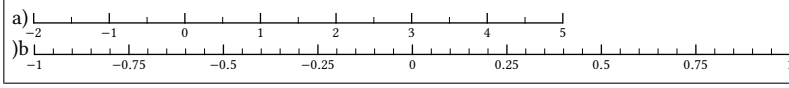
– une espace de dimension #1 ÷ #5 ;
– une réglure de dimension nulle correspondant à la marque de subdivision.

Nous avons le choix pour remplir cette \hbox avec ces #5 − 1 répétitions. Soit nous faisons appel à \leaders, soit à une boucle \for. C'est cette deuxième alternative qui sera choisie ici :

Code n^{o} V-382

```
\maingraddim=4pt \maingradwd=0.5pt \axiswd=0.5pt
\newdimen\subgraddim \subgraddim=2.5pt
\newdimen\subgradwd \subgradwd=0.2pt
% trace un trait de subdivision
\def\subgradx{\clap{\vrule height\subgraddim width\subgradwd depth0pt }}

\newmacro\xaxis[1cm]1[1]1[4]{% #1= dist  #2=xmin  #3=inc  #4=xmax  #5=subdiv
  \hbox{% tout mettre dans une \hbox
    \setbox0=\hbox{% stocke dans une \hbox les grad secondaires entre 2 unités
      \edef\dimsubgrad{\the\dimexpr#1/#5\relax}% dimension entre 2 subdivisions
      \for\xx=1 to #5-1 \do{% insérer #5-1 fois
        \kern\dimsubgrad% une espace secondaire
        \subgradx% une graduation secondaire
        }%
    }%
    \rlap{% en débordement à droite :
      \FOR\xx = #2 to #4 \do #3{% pour chaque graduation principale
        \maingradx{\xx}% imprimer l'abscisse
        \ifdim\xx pt<#4pt % et en débordement à droite,
          \rlap{\copy0 }% les réglures secondaires, sauf pour la dernière
        \fi
        \kern#1\relax% se déplacer vers la droite
      }%
    }%
    \vrule% tracer l'axe des abscisses
      height\axiswd% d'epaisseur \axiswd, de longueur #1*(#4-#2)/#3
      width\dimexpr#1*\decdiv{\dimtodec\dimexpr#4pt-#2pt\relax}{#3}\relax
      depth 0pt\relax % et de profondeur nulle
```

```
  }%
}
a) \xaxis{-2}{5}[2]

)b \xaxis[1.25cm]{-1}[0.25]{1}[5]
```



Retenons que la largeur de la \hbox chapeau est la largeur de la zone graphique. Notons également que l'on *ne peut pas* écrire

```
\setbox0\xaxis{-2}{5}[2]
```

car, contrairement aux apparences, la macro \xaxis ne se développe pas en une \hbox. Cela est dû à la façon dont \newmacro construit les macros qui ont des arguments optionnels : des sous-macros sont créées et l'emploi de \futurelet annule toute possibilité de développement maximal.

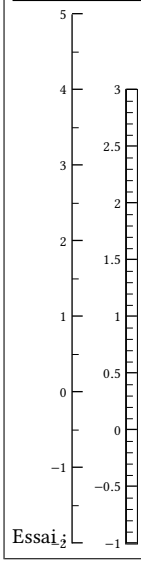
Passons maintenant aux axes verticaux. La méthode va être la même, sauf que toutes les primitives typées en mode horizontal passeront cette fois en mode vertical : nous mettrons le tout dans une \vbox et puisqu'elle sera remplie du haut vers le bas, il faudra commencer avec l'ordonnée de fin et construire une boucle \FOR avec un incrément négatif.

La méthode utilisée avec \yaxis* pour l'affichage des ordonnées ne sera pas identiques à celles des abscisses : les dimensions des nombres représentant les ordonnées seront rendues nulles grâce à une \llap. En effet, la dimension horizontale d'un nombre est variable alors que sa dimension verticale est constante. Réserver suffisamment d'espace à gauche de l'axe pour y placer les ordonnées aurait nécessité de calculer la plus grande dimension horizontale des ordonnées ce qui implique une boucle préalable. Il s'avère donc plus facile et plus économique d'ignorer cette dimension horizontale, quitte à rejeter hors de la boite englobante toutes les ordonnées.

Code n° V-383

```
\maingraddim=4pt \maingradwd=0.5pt \axiswd=0.5pt
\subgraddim=2.5pt \subgradwd=0.2pt
\def\maingrady#1{% affiche...
  \vlap{\llap{$\scriptscriptstyle#1$\kern2pt }}% l'ordonnée...
  \vbox to0pt{\vss\hrule height\maingradwd width\maingraddim depth0pt }% et la réglure
}

% affiche une subdiv
\def\subgrady{\vlap{\hrule height\subgradwd width\subgraddim depth0pt }}
\newmacro\yaxis[1cm]1[1]1[4]{%
% #1= dist  #2=ymin  #3=inc  #4=ymax  #5=subdiv
  \vbox{%
    \offinterlineskip% désactiver le ressort d'interligne
    \setbox0=\vbox{% stocke dans une \hbox les grad secondaires entre 2 unités
      \edef\dimsubgrad{\the\dimexpr#1/#5\relax}% dimension entre 2 subdivisions
      \for\xx=1 to #5-1 \do{% insérer #5-1 fois
        \kern\dimsubgrad% une espace secondaire
        \subgrady% une graduation secondaire
        }%
    }%
    \edef\dimsubgrad{\the\dimexpr#1/#5\relax}% distance entre 2 subdivisions
```

```
    \vbox to 0pt{% en débordement vers le bas
      \FOR\xx = #4to#2\do-#3{%
        \maingrady{\xx}% imprimer l'abscisse
        \ifdim\xx pt>#2pt % et en débordement à droite,
          \vbox to 0pt{\copy0 \vss}% les réglures secondaires, sauf pour la dernière
        \fi
        \kern#1\relax% se déplacer vers la droite
      }%
      \vss% assure le débordement vers le bas
    }%
    \clap{\vrule% tracer l'axe des ordonnées
      width\axiswd% d'épaisseur \axiwd, et de hauteur (#4-#2)/#3*#1
      height\decdiv{\dimtodec\dimexpr(#4pt-#2pt)\relax}{#3}\dimexpr#1\relax
      depth 0pt\relax % profondeur nulle
    }%
  }%
}

Essai : \yaxis{-2}{5}[2]\qquad \frboxsep=0pt \frbox{\yaxis[0.75cm]{-1}[0.5]{3}[5]}
```



2.2.2. Créer une zone pour le graphique

Tout est maintenant prêt pour tracer notre zone graphique. Deux registres de dimension supplémentaires vont être créés, `\xunit` et `\yunit`, chacun spécifiant la distance entre deux graduations principales sur chacun des axes. La syntaxe de la macro va être proche des syntaxes de `\xaxis` ou `\yaxis` :

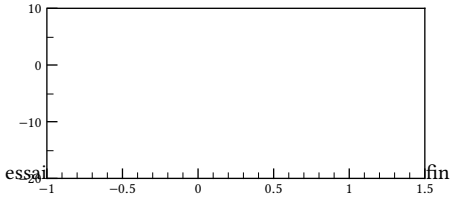
`\graphzone{xmin}[xinc]{xmax}[nbsubdivx]{ymin}[yinc]{ymax}[nbsubdivy]{`⟨*code*⟩`}`

où le ⟨*code*⟩ contient les instructions graphiques que nous verrons plus tard et qui permettent de dessiner dans la zone.

Code n° V-384

```
\newdimen\xunit \xunit=1cm
\newdimen\yunit \yunit=0.75cm
\maingraddim=4pt \maingradwd=0.5pt \axiswd=0.5pt
```

```
\subgraddim=2.5pt \subgradwd=0.2pt
\catcode`@11
\newmacro\graphzone1[1]1[4]1[1]1[4]{%
  \leavevmode% quitter le mode vertical
  \begingroup% travailler dans un groupe semi-simple
    \def\graphxmin{#1}\def\graphxmax{#3}% sauvegarder
    \def\graphymin{#5}\def\graphymax{#7}% les
    \def\xincrement{#2}\def\yincrement{#6}% arguments
    \setbox0\hbox{\yaxis[\yunit]{#5}[#6]{#7}[#8]}% axe "y" dans boite 0
    \setbox1\hbox{\xaxis[\xunit]{#1}[#2]{#3}[#4]}% axe "x" dans boite 1
    \edef\graphboxht{\the\ht0 }% \graphboxh est la hauteur de la zone
    \edef\graphboxwd{\the\wd1 }% \graphboxw est la largeur de la zone
    \rlap{% annuler la dimension horizontale, et...
      \box1 % ...afficher l'axe (Ox) puis
      % le trait vertical à l'extrême droite de la zone
      \clap{\vrule height\dimexpr\graphboxht+\axiswd\relax width\axiswd depth0pt }%
      }%
    \rlap{\box0 }% afficher l'axe (Oy) en annulant sa dimension horizontale
    \raise\graphboxht% puis monter tout en haut de la zone
      % pour tracer le trait horizontal en annulant sa longueur
      \rlap{\kern-0.5\axiswd% et en rattrapant le centrage de l'axe des ordonnées
        \vrule height\axiswd width\dimexpr\graphboxwd+\axiswd}%
    \begingroup% pour lire l'argument "<code>" :
      \catcode`\^^M=9\relax % ignorer les retours à la ligne
      \graphzone@i% appeler \graphzone@i
}
\def\graphzone@i#1{% lire le <code>...
    \endgroup% et fermer le groupe précédemment ouvert
    \setbox0\hbox{#1}% mettre les instructions graphique dans la boite 0
    \wd0=\dimexpr\graphboxwd+\axiswd\relax% forcer sa dim horizontale
    \ht0=\dimexpr\graphboxht+\axiswd\relax% et verticale
    % pour correspondre aux dimension de la zone graphique
    \box0 % l'afficher
  \endgroup% sortir du groupe initial
}
\catcode`@12
essai\graphzone{-1}[0.5]{1.5}[5]{-20}[10]{10}[2]{\relax}fin
```



Le code ne présente pas beaucoup de difficulté. Signalons tout de même les quelques points suivants :

- les réglures correspondant au trait vertical à droite et au trait horizontal en haut sont tracées à coup de `\vrule`. Celle de droite est tracée à la suite de `\box1` qui est l'axe horizontal et celle du haut en est correctement placée en s'aidant de `\raise` qui élève la boite `\rlap` de la hauteur de la zone graphique ;
- on a modifié le code de catégorie du retour à la ligne dans l'argument final ⟨*code*⟩ de `\graphzone` afin que les instructions graphiques puissent être entrées le plus « naturellement » possible, notamment sans avoir besoin de commenter les fins de ligne ;
- le point de référence demeure en bas à gauche de la zone graphique jusqu'au

\box0 de la macro \graphzone@i (ligne n° 36) puisque jusque là, on n'a affiché que des \rlap. C'est justement en forçant les dimensions de cette boite n° 0 aux dimensions de la zone graphique qu'en l'affichant, on obtient le bon encombrement final.

2.2.3. Remplir la zone graphique

Il nous reste à écrire des instructions graphiques pour remplir la zone… Tout d'abord, il va falloir construire une macro \putat* qui place n'importe quel contenu relativement au point de référence courant :

\putat{dimx}{dimy}{⟨*contenu*⟩}

Il est bien entendu que cette macro va placer le point de référence du ⟨*contenu*⟩ à des coordonnées relatives par rapport au point de référence courant, mais *sans changer ce point de référence courant*. Autrement dit, la boite englobante de cette macro a toutes ses dimensions nulles et donc, son exécution n'a aucune influence sur le point de référence courant.

En jouant sur les boites et imbriquant des boites horizontales et verticales en débordement, on peut facilement effectuer cette action :

Code n° V-385

```
\def\putat#1#2#3{%
  \leavevmode\rlap{\kern#1\vbox to0pt{\vss\hbox{#3}\kern#2}}%
}
Essai\putat{0.5cm}{0.3cm}{A}\putat{-0.2cm}{-0.1cm}{B}\putat{0pt}{0.2cm}{C}suite
```

Essai C A B suite

Cette macro \putat va nous permettre de placer n'importe quel matériel dans la zone graphique, mais l'idéal serait que le matériel placé le soit avec les coordonnées affichées sur les bords de la zone graphique (que nous appelons X et Y) et non pas les coordonnées par rapport au coin inférieur gauche (x et y) qui est le point de référence. Il est bien entendu facile de passer d'un système de coordonnées à l'autre avec une simple formule de changement de repère :

$$\begin{cases} X = (x - x_{\text{min}}) \div x_{\text{inc}} \\ Y = (y - y_{\text{min}}) \div y_{\text{inc}} \end{cases}$$

Pour éviter que les divisions ne soient faites à chaque placement d'un objet, nous diviserons une bonne fois pour toutes les dimensions \xunit et \yunit par les incréments x_{inc} et y_{inc} à l'intérieur de la zone graphique.

Pour placer les axes, il faut d'abord tester si le nombre 0 est contenu dans l'intervalle représenté sur l'axe horizontal dans la zone graphique et faire de même pour l'axe vertical. Le plus simple est de modifier la macro \ifinside, programmée à la page 155, mais qui avait le défaut de n'agir que sur les entiers :

Code n° V-386

```
\def\ifinside#1[#2,#3]{%
  \ifnum\sgn{\dimexpr#1pt-#2pt\relax}\sgn{\dimexpr#1pt-#3pt}1=1
    \expandafter\secondoftwo
  \else
```

```
    \expandafter\firstoftwo
  \fi
}
1) \ifinside3.5[0.5,4]{vrai}{faux}\qquad
2) \ifinside-0.2[-0.4,-0.3]{vrai}{faux}\qquad
3) \ifinside-0.999[-1,-0.5]{vrai}{faux}
```

1) vrai 2) faux 3) vrai

Puis, il faut définir la fonction dont on veut afficher la courbe représentative, sachant que les seules opérations dont nous disposons sont les quatre opérations arithmétiques sur des décimaux. On ne peut donc représenter que des fonctions rationnelles [3].

Supposons que l'on veuille représenter la fonction $f(x) = x^3 - x^2 - 3x + 2$. Nous allons devoir faire le petit effort de traduire cet enchainement d'opérations en instructions compréhensibles par ε-TEX muni des macros rendant possibles la multiplication et la division par un décimal que sont `\decmul` et `\decdiv`, vues à partir de la page 239 :

Code n° V-387

```
\def\fonction#1{\dimtodec\dimexpr
  \decmul{#1}{\decmul{#1}{#1}}pt% x^3
  -\decmul{#1}{#1}pt% -x^2
  -#1pt*3% -3x
  +2pt\relax}% +2
1) \fonction{-4}\qquad% doit afficher -66
2) \fonction{-1.7}\qquad %  doit afficher -0.703
3) \fonction{1}\qquad%  doit afficher -1
4) \fonction{1.35}%  doit afficher -1.412125
```

1) -66 2) -0.70296 3) -1 4) -1.41214

On voit ici que l'on a ajouté les dimensions en `pt` pour permettre à `\dimexpr` d'effectuer le calcul avant que `\dimtodec` n'enlève cette unité pour que cette macro, purement développable, renvoie un nombre décimal sans unité. Bien sûr, certaines erreurs d'arrondis rendent les résultats légèrement faux ce qui n'a aucune importance ici puisqu'ils seront des coordonnées de points dans un repère où l'erreur commise est insignifiante en regard de ce qui est visible à l'œil nu.

Il faut ensuite définir la forme du point qui sera affiché. Pourquoi pas une croix ? Il faut bien sûr qu'elle soit de dimensions nulles :

Code n° V-388

```
\newmacro\cross[2pt][0.2pt]{%
  % #1=dimensions de traits depuis le centre de la croix
  % #2=épaisseur des traits
  \leavevmode
  \vlap{%
    \clap{%
      \vrule height#2 depth0pt width#1 % 1/2 trait horizontal gauche
      \vrule height#1 depth#1 width#2  % trait vertical
      \vrule height#2 depth0pt width#1 % 1/2 trait horizontal droit
    }%
```

3. Il est bien évident qu'avec une extension spécialisée dans le calcul comme l'est « `fp` », toutes les fonctions, y compris transcendantes, sont très facilement accessibles.

```
  }%
}
Une croix : \cross{}puis une autre\cross[8pt][0.8pt]
```

Une croix : puis une autre

Voici maintenant le code reprenant toutes les macros vues précédemment où deux courbes représentatives sont construites :

Code n° V-389

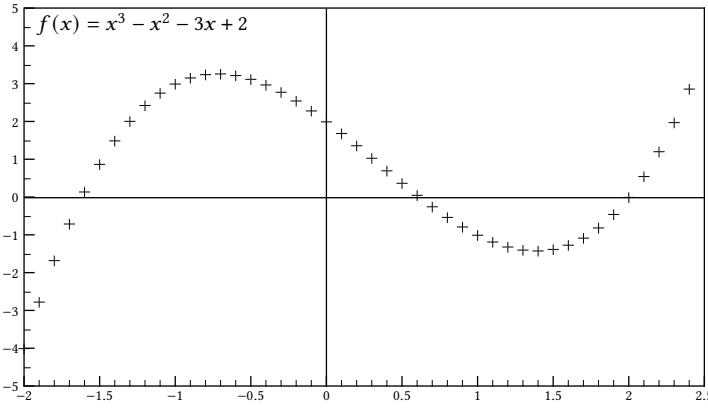
```
\maingraddim=4pt \maingradwd=0.5pt \axiswd=0.5pt
\subgraddim=2.5pt \subgradwd=0.2pt
\xunit=1.25cm \yunit=0.75cm

\def\maingradx#1{%
  \lower1.5ex\clap{$\scriptscriptstyle#1$}% afficher l'argument au dessous
  \clap{\vrule height\maingraddim width\maingradwd depth0pt }% et la réglure
}

\def\subgradx{\clap{\vrule height\subgraddim width\subgradwd depth0pt }}

\newmacro\xaxis[1cm]1[1]1[4]{%
  \hbox{% tout mettre dans une \hbox
    \setbox0=\hbox{% stocke dans une \hbox les grad secondaires entre 2 unités
      \edef\dimsubgrad{\the\dimexpr#1/#5\relax}% dimension entre 2 subdivisions
      \for\xx=1 to #5-1 \do{% insérer #5-1 fois
        \kern\dimsubgrad% une espace secondaire
        \subgradx% une graduation secondaire
        }%
    }%
    \rlap{% en débordement à droite :
      \FOR\xx = #2 to #4 \do #3{% pour chaque graduation principale
        \maingradx{\xx}% imprimer l'abscisse
        \ifdim\xx pt<#4pt % et en débordement à droite,
          \rlap{\copy0 }% les réglures secondaires, sauf pour la dernière
        \fi
        \kern#1\relax% se déplacer vers la droite
      }%
    }%
    \vrule% tracer l'axe des abscisses
      height\axiswd% d'epaisseur \axiswd, de longueur #1*(#4-#2)/#3
      width\dimexpr#1*\decdiv{\dimtodec\dimexpr#4pt-#2pt\relax}{#3}\relax
      depth 0pt\relax % et de profondeur nulle
  }%
}

\def\maingrady#1{% affiche...
  \vlap{\llap{$\scriptscriptstyle#1$\kern2pt }}% l'ordonnée...
  \vbox to0pt{\vss\hrule height\maingradwd width\maingraddim depth0pt }% et la réglure
}

% affiche une subdiv
\def\subgrady{\vlap{\hrule height\subgradwd width\subgraddim depth0pt }}
% #1= dim entre 2 grad principales #2=abscisse départ #3=incrément
% #4=abscisse arrivée #5=nb intervalles secondaires
\newmacro\yaxis[1cm]1[1]1[4]{%
  \vbox{%
    \offinterlineskip% désactiver le ressort d'interligne
    \setbox0=\vbox{% stocke dans une \hbox les grad secondaires entre 2 unités
```

```
      \edef\dimsubgrad{\the\dimexpr#1/#5\relax}% dimension entre 2 subdivisions
      \for\xx=1 to #5-1 \do{% insérer #5-1 fois
        \kern\dimsubgrad% une espace secondaire
        \subgrady% une graduation secondaire
        }%
    }%
    \edef\dimsubgrad{\the\dimexpr#1/#5\relax}% distance entre 2 subdivisions
    \vbox to 0pt{% en débordement vers le bas
      \FOR\xx = #4to#2\do-#3{%
        \maingrady{\xx}% imprimer l'abscisse
        \ifdim\xx pt>#2pt % et en débordement à droite,
          \vbox to 0pt{\copy0 \vss}% les réglures secondaires, sauf pour la dernière
        \fi
        \kern#1\relax% se déplacer vers la droite
      }%
      \vss% assure le débordement vers le bas
    }%
    \clap{\vrule% tracer l'axe des ordonnées
      width\axiswd% d'épaisseur \axiwd, et de hauteur (#4-#2)/#3*#1
      height\decdiv{\dimtodec\dimexpr(#4pt-#2pt)\relax}{#3}\dimexpr#1\relax
      depth 0pt\relax % profondeur nulle
    }%
  }%
}

\catcode`@11
\newmacro\graphzone1[1]1[4]1[1]1[4]{%
  \leavevmode% quitter le mode vertical
  \begingroup% travailler dans un groupe semi-simple
    \def\graphxmin{#1}\def\graphxmax{#3}% sauvegarder
    \def\graphymin{#5}\def\graphymax{#7}% les
    \def\xincrement{#2}\def\yincrement{#6}% arguments
    \setbox0\hbox{\yaxis[\yunit]{#5}[#6]{#7}[#8]}% axe "y" dans boite 0
    \setbox1\hbox{\xaxis[\xunit]{#1}[#2]{#3}[#4]}% axe "x" dans boite 1
    \edef\graphboxht{\the\ht0 }% \graphboxh est la hauteur de la zone
    \edef\graphboxwd{\the\wd1 }% \graphboxw est la largeur de la zone
    \rlap{% annuler la dimension horizontale, et...
      \box1 % ...afficher l'axe (Ox) puis
      % le trait vertical à l'extrême droite de la zone
      \clap{\vrule height\dimexpr\graphboxht+\axiswd\relax width\axiswd depth0pt }%
      }%
    \rlap{\box0 }% afficher l'axe (Oy) en annulant sa dimension horizontale
    \raise\graphboxht% puis monter tout en haut de la zone
      % pour tracer le trait horizontal en annulant sa longueur
      \rlap{\kern-0.5\axiswd% et en rattrapant le centrage de l'axe des ordonnées
        \vrule height\axiswd width\dimexpr\graphboxwd+\axiswd}%
    \begingroup% pour lire l'argument "<code>" :
      \catcode`\^^M=9\relax % ignorer les retours à la ligne
      \graphzone@i% appeler \graphzone@i
}
\def\graphzone@i#1{% lire le <code>...
    \endgroup% et fermer le groupe précédemment ouvert
    \xunit=\decdiv1\xincrement\xunit% divise les unités par l'incrément
    \yunit=\decdiv1\yincrement\yunit
    \setbox0\hbox{#1}% mettre les instructions graphique dans la boite 0
    \wd0=\dimexpr\graphboxwd+\axiswd\relax% forcer sa dim horizontale
    \ht0=\dimexpr\graphboxht+\axiswd\relax% et verticale
    % pour correspondre aux dimension de la zone graphique
    \box0 % l'afficher
  \endgroup% sortir du groupe initial
```

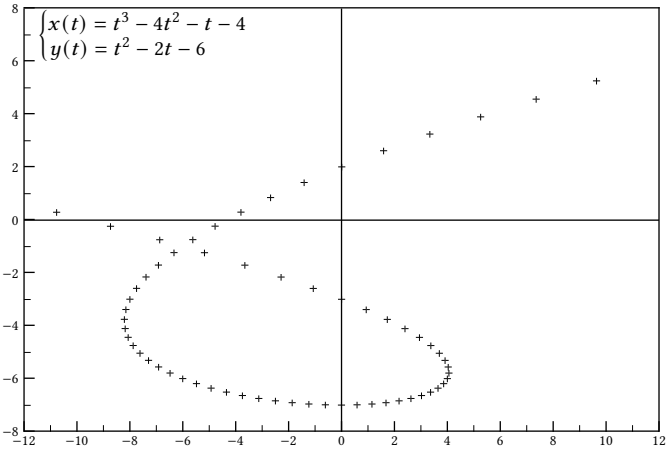
```
}

\def\putat#1#2#3{%
  \leavevmode\rlap{\kern#1\vbox to0pt{\vss\hbox{#3}\kern#2}}%
}

\newmacro\cross[2pt][0.2pt]{%
  \leavevmode
  \vlap{%
    \clap{%
      \vrule height#2 depth0pt width#1 % 1/2 terait horizontal gauche
      \vrule height#1 depth#1 width#2  % trait vertical
      \vrule height#2 depth0pt width#1 % 1/2 trait horizontal droit
    }%
  }%
}

\def\plot(#1,#2){% place "\plotstuff" aux coordonnées #1,#2
  \edef\x@plot{#1}\edef\y@plot{#2}% développer au cas où
  \ifinside\x@plot[\graphxmin,\graphxmax]% si #1 est dans les limites
    {\ifinside\y@plot[\graphymin,\graphymax]% et si #2 l'est aussi
      {\putat% placer aux coordonnées
        {\dimexpr\x@plot\xunit-\graphxmin\xunit\relax}% X=(x-xmin)/xinc
        {\dimexpr\y@plot\yunit-\graphymin\yunit\relax}% Y=(y-ymin)/yinc
        \plotstuff% le contenu de \plotstuff
      }%
      \relax
    }
    \relax
}

\def\showxaxis{% affiche l'axe (Ox)
  \ifinside0[\graphymin,\graphymax]%
    {\putat\z@{-\graphymin\yunit}{\vlap{\vrule width\graphboxwd height\axiswd}}}%
    \relax
}

\def\showyaxis{% affiche l'axe (Oy)
  \ifinside0[\graphxmin,\graphxmax]%
    {\putat{-\graphxmin\xunit}\z@{\clap{\vrule width\axiswd height\graphboxht}}}%
    \relax
}

\def\showaxis{\showxaxis\showyaxis}% affiche les deux axes
\catcode`@12

%%%%%%%%%%%%%% 1er exemple :
\xunit=1cm \yunit=0.5cm
\def\fonction#1{\dimtodec\dimexpr\decmul{#1}{\decmul{#1}{#1}}pt-%
  \decmul{#1}{#1}pt-#1pt*3+2pt\relax}
Exemple 1 :\qquad
\graphzone{-2}[0.5]{2.5}[5]{-5}{5}[2]{%
  \let\plotstuff\cross% définit ce qu'il faut afficher comme point
  \FOR\xx=-2 to 2.5 \do 0.1{%
    \plot(\xx,\fonction{\xx})% afficher les points de coordonnées (x ; f(x))
  }%
  \putat{5pt}{\dimexpr\graphboxht-10pt\relax}% afficher
    {$f(x)=x^3-x^2-3x+2$}% la fonction tracée
  \showaxis% et les axes
}\par\medskip
```

```
%%%%%%%%%%%% 2e exemple
\xunit=0.7cm \yunit=\xunit
\def\foncx#1{\dimtodec\dimexpr\decmul{#1}{\decmul{#1}{#1}}pt-%
  \decmul{4}{\decmul{#1}{#1}}pt-#1pt+4pt}
\def\foncy#1{\dimtodec\dimexpr\decmul{#1}{#1}pt-#1pt*2-6pt}
Exemple 2 :\qquad
\graphzone{-12}[2]{12}[2]{-8}[2]{8}[2]{%
  \def\plotstuff{\cross[1.25pt]}
  \FOR\tt = -5 to 5 \do 0.1 {%
    \plot(\foncx\tt,\foncy\tt)%
  }%
  \putat{5pt}{\dimexpr\graphboxht-20pt\relax}% afficher la fonction paramétrique tracée
    {$\left\{\vcenter{\hbox{$x(t)=t^3-4t^2-t-4$}\hbox{$y(t)=t^2-2t-6$}}\right.$}%
  \showaxis
}
```

Exemple 1 :



Exemple 2 :



2.2.4. Construire l'ensemble de Mandelbrot

L'ensemble de Mandelbrot [4], noté $\mathcal{M}$, est une figure fractale définie comme l'ensemble des points c du plan complexe pour lesquels la suite définie par récurrence par

4. Du nom de son inventeur, Benoît Mandelbrot (1924–2010), mathématicien franco-arménien.

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

est bornée en module. En posant $c = a + \mathrm{i}b$, on passe aux coordonnées cartésiennes et l'on obtient

$$\begin{cases} x_0 = 0 \\ x_{n+1} = x_n^2 - y_n^2 + a \end{cases} \qquad \begin{cases} y_0 = 0 \\ y_{n+1} = 2x_n y_n + b \end{cases}$$

Cet ensemble a été beaucoup étudié et on a démontré les choses suivantes qui vont nous servir pour le construire :

- $\mathcal{M}$ est symétrique par rapport à l'axe des abscisses et donc, on peut se limiter au calcul des points dont la partie imaginaire est positive et tracer leurs symétriques par rapport à (Ox) ;
- si à un certain indice i, on a $|z_i| > 2$, alors la suite diverge et donc, le point d'affixe c n'appartient pas à $\mathcal{M}$. Il est dès lors inutile de poursuivre les calculs aux indices supérieurs à i.

Pour dessiner l'ensemble, nous nous limiterons à la portion de plan complexe comprise entre -2 et 1 pour la partie réelle et entre -1 et 1 pour la partie imaginaire. L'essentiel est d'écrire une macro `\mandeltest*` qui admet deux arguments décimaux qui sont les coordonnées du point à tester. Cette macro mettra en place une boucle qui calculera les termes de la suite en fonction des deux arguments qui représentent $Re(c)$ et $Im(c)$. Bien évidemment, si la suite ne diverge pas, il n'est pas possible de calculer un nombre illimité de termes de la série. Nous nous limiterons à un entier, contenu dans la macro `\maxiter`.

L'algorithme de `\mandeltest` est très simple : les itérations doivent se poursuivre tant que le nombre d'itérations est inférieur à `\maxiter` et que le module du terme z_i actuellement calculé est inférieur à 2. Une fois le test terminé et l'issue connue, le test `\mandeltest` définira la macro `\mandelresult` : elle contiendra 1 si le test est positif et 0 sinon.

Pour chaque terme z_i, les macros \zx et \zy contiennent $Re(z_i)$ et $Im(z_i)$. Par souci d'optimisation et pour ne pas les calculer deux fois, nous stockerons les carrés de \zx et \zy dans \zxx et \zyy.

Tester si un point appartient à M

```
macro mandeltest #1#2 % #1 et #2 = coordonnées du point à tester
    mandelresult ← 1 % on considère a priori que le point apprtient à M
    zx ← 0          zy ← 0 % partie réelle et imaginaire de z0
    zxx ← 0         zyy ← 0 % les carrés de zx et zy
    pour i=1 to maxiter
        zy ← 2*zx*zy + #2 % y_{n+1} = 2x_n y_n + Im(c)
        zx ← zxx-zyy + #1 % x_{n+1} = x_n^2 - y_n^2 + Re(c)
        zxx ← zx*zx       zyy ← zy*zy % calcul des carrés
        si zxx + zyy > 4 % si |z_n|^2 > 4 (càd si le module est > à 2)
            mandelresult ← 0 % le point n'est pas dans M
            i ← maxiter % sortir de la boucle prématurément
        finsi
    finpour
finmacro
```

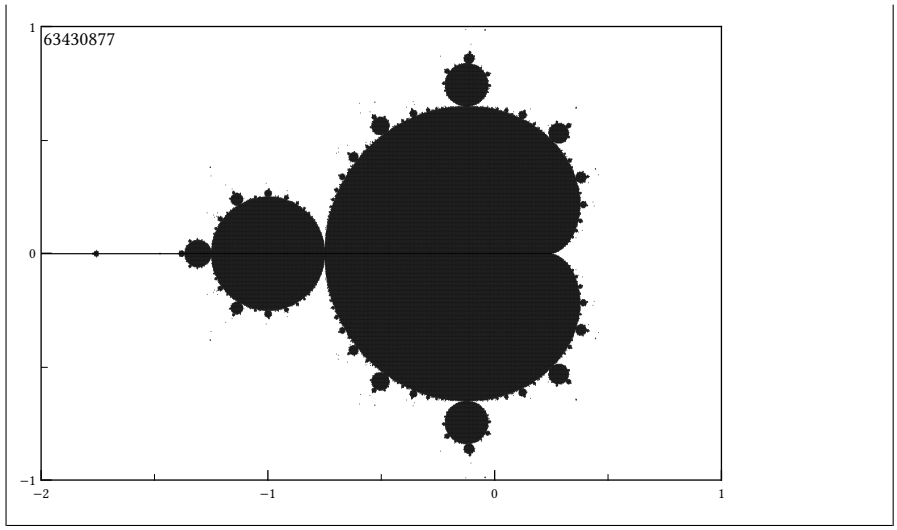
Dans le domaine $[-2\ ;\ 1] \times [-1\ ;\ 1]$ nous allons choisir la *résolution* du dessin, c'est-à-dire combien de pixels doivent être contenus dans une unité, aussi bien horizontalement que verticalement. Ce nombre sera le premier argument de la macro `\mandel` chargée de tracer $\mathcal{M}$. Le second argument sera le nombre maximal d'itérations.

Pour parcourir tous les points du rectangle, nous allons utiliser deux boucles `\FOR` imbriquées, l'une pilotant les abscisses `\xxx` et l'autre les ordonnées `\yyy`. L'incrément sera égal à $\frac{1}{\#1}$. Dans le cœur de la boucle, c'est-à-dire pour chaque point de coordonnées (`\xxx` ; `\yyy`), le test `\mandeltest` sera effectué et si la macro `\mandelresult` vaut 1, le pixel et son symétrique par rapport à l'axe (Ox) seront affichés.

Un pixel sera une réglure carrée ayant pour côté l'incrément multiplié par la longueur `\xunit`.

Code n° V-390

```
\def\mandeltest#1#2{%\mandeltest
  \def\zx{0}\def\zy{0}% zn=0 + i*0
  \def\zxx{0}\def\zyy{0}% carrés de \zx et \zy
  \def\mandelresult{1}% le point appartient à M a priori
  \for\ii=1to\maxiter\do1{%
    \advance\count255 by1
    \edef\zy{\dimtodec\dimexpr\decmul{\decmul2\zx}\zy pt+#2pt\relax}%
    \edef\zx{\dimtodec\dimexpr\zxx pt-\zyy pt+#1pt\relax}%%
    \edef\zxx{\decmul\zx\zx}%
    \edef\zyy{\decmul\zy\zy}%
    \ifdim\dimexpr\zxx pt+\zyy pt\relax>4pt
      \def\mandelresult{0}%
       \exitfor\ii
    \fi
  }%
}
\def\mandel#1#2{% #1=points par unité #2=nombre maximal d'itérations\mandel
  \graphzone{-2}[1]{1}[2]{-1}[1]{1}[2]{%
    \def\maxiter{#2}%
    \edef\plotstuff{\the\dimexpr\xunit/#1\relax}% taille d'un pixel
    \edef\plotstuff{\vrule height\plotstuff width\plotstuff}%
    \edef\increment{\decdiv{1}{#1}}% incrément
    \count255=0 % compteur des itérations
    \FOR\xxx = -2 to 1 \do \increment{% pour chaque
      \FOR\yyy = 0 to 1 \do \increment{% pixel du domaine
        \mandeltest\xxx\yyy% tester s'il est dans M
        \ifnum\mandelresult=1 % si oui,
          \plot(\xxx,\yyy)\plot(\xxx,-\yyy)% afficher les 2 points
        \fi
      }%
    }%
  \edef\plotstuff{$\scriptstyle\number\count255 $}% affiche la valeur du compteur
  \plot(-1.99,0.92)% aux coordonnées (-1.99 ; 0.92)
  }%
}
\xunit=3cm \yunit=3cm \mandel{400}{500}\mandel
```



Afin d'obtenir un tracé précis et assez fidèle à ce qu'est $\mathcal{M}$, nous avons demandé un grand nombre de pixels par unité (400) et un nombre élevé d'itérations (500). Nous avons donc 3×400 pixels horizontalement et 400 pixels verticalement à tester c'est-à-dire 480 000 tests à faire. Chacun exécute une boucle supplémentaire pouvant aller jusqu'à 500 itérations. Même si tous les tests ne procèdent pas à ces 500 itérations, il s'agit là sans nul doute du plus grand nombre d'itérations qu'il m'ait été donné de programmer en TEX : il y en a très exactement 63 430 877 ! Compte tenu de la grandeur des arguments de `\mandel` conduisant à cet incroyable nombre d'itérations, la compilation de ce seul code demande de longues minutes. En outre, le nombre de pixels (c'est-à-dire de réglures) à placer sur la page est extrêmement important ce qui requiert une augmentation de la mémoire de TEX. Pour ce faire, les paramètres suivants ont été modifiés :

```
main_memory = 10000000
extra_mem_bot = 20000000
```

Ce dernier code, en raison du temps de compilation qu'il requiert, déroge à la règle qui veut que chaque code génère l'affichage qui se trouve dans la partie basse du cadre. Ce code a donc été compilé à part avec une version modifiée de `tex` et le fichier pdf obtenu a été incorporé sous le code.

Chapitre 3

Aller plus loin dans la mise en forme

3.1. Formater un nombre

Quelle différence y a t-il entre « 9876543,21012 » et « 9 876 543,210 12 » ? Mathématiquement aucune, mais typographiquement, la différence est énorme : elle se situe au niveau de ces petites espaces insécables qui séparent les groupes de chiffres en les groupant trois par trois et qui font que, du premier coup d'œil, on sait que 9 est le chiffre des unités de millions. Il en va ainsi de la typographie française qui, par souci de lisibilité, fixe comme règle d'insérer une espace fine insécable entre chaque groupe de trois chiffres.

Le but que nous nous fixons e de pouvoir formater n'importe quel nombre, on ne va s'imposer aucune limite sur le nombre de chiffres, pas plus dans la partie entière que dans la partie décimale. Par conséquent, nous ne pourrons à aucun moment utiliser la primitive `\number` puisque l'entier qui la suit doit être inférieur à $2^{31} - 1$.

3.1.1. Partie décimale

La partie facile est de traiter la partie décimale puisqu'elle doit être parcourue de gauche à droite – le sens de lecture de TEX – afin d'insérer une espace fine (créée par la macro `\numsep`) chaque fois que l'on a parcouru 3 chiffres. Pour cela, nous allons reprendre la macro `\reverse` vue à la page 191 en la modifiant légèrement. Cette macro offre la particularité intéressante de fournir du matériel purement développable ne produisant aucune sortie vers l'affichage jusqu'à ce que l'argument

soit inversé en totalité, moment il est dirigé vers l'affichage.

Sur ce modèle, nous allons bâtir une macro récursive `\formatdecpart@i` qui admet trois arguments délimités. Le premier est l'entier qui représente le nombre de chiffres parcourus et qu'il faudra tester par rapport à 3. Les deux autres sont les « réservoirs » déjà vus avec la macro `\reverse` qui stockent les tokens et qui, en se passant de séquence de contrôle définies par `\def`, rendent la macro purement développable.

Code n° V-391

```
\catcode`\@11
\protected\def\numsep{\kern0.2em }% \numsep est le séparateur mis tous les 3 chiffres

\def\formatdecpart#1{% #1=série de chiffres
  \ifempty{#1}% si la partie décimale est vide
    {}% ne rien afficher
    {{,}\formatdecpart@i 1.#1..}% sinon, afficher la virgule et mettre en forme
}

% #1=compteur de caractères #2= chiffre courant
% #3= chiffres restants     #4 = chiffres déjà traités
\def\formatdecpart@i#1.#2#3.#4.{%
  \ifempty{#3}% si #2 est le dernier chiffre
    {#4#2}% le mettre en dernière position et tout afficher, sinon
    {\ifnum#1=3 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      % si 3 chiffres sont atteint, rendre #1 égal à 1 et
      {\formatdecpart@i 1.#3.#4#2\numsep.}% mettre #2\numsep en dernier puis recommencer
      % sinon, mettre #2 en dernière position et recommencer
      % tout en incrémentant #1 de 1
      {\expandafter\formatdecpart@i \number\numexpr#1+1.#3.#4#2.}%
    }%
}
a) \formatdecpart{1234567}\qquad
b) \formatdecpart{987125}\qquad
c) \formatdecpart{56}\qquad
d) \edef\foo{\formatdecpart{2014}}\meaning\foo
```

a) ,123 456 7 b) ,987 125 c) ,56 d) macro:->{,}201\numsep 4

La virgule, qui joue le rôle de séparateur décimal est insérée entre accolades pour que cette virgule ne soit pas prise comme un séparateur de liste si la macro est exécutée en mode mathématique car dans ce cas, elle serait suivie d'une espace.

3.1.2. Partie entière

La partie entière est plus difficile. La première étape sera de légèrement modifier la macro `\formatdecpart` en la macro `\formatintpart*` pour que cette dernière affiche le nombre qu'elle a traité, mais à l'envers. Pour cela, il suffit de changer l'ordre des arguments #2 et #4 dans le texte de remplacement de `\formatdecpart@i` :

Code n° V-392

```
\catcode`\@11
\protected\def\numsep{\kern0.2em }% \numsep est le séparateur mis tous les 3 chiffres
\def\formatintpart#1{% #1=série de chiffres
  \formatintpart@i 1.#1..% appelle la macro récursive
}
```

```
% #1=compteur de caractères #2= chiffre courant
% #3= chiffres restants     #4 = chiffres déjà traités
\def\formatintpart@i#1.#2#3.#4.{%
  \ifempty{#3}% si #2 est le dernier chiffre
    {#2#4}% le mettre en première position et tout afficher, sinon
    {\ifnum#1=3 \expandafter\firstoftwo\else\expandafter\secondoftwo
    \fi% si 3 chiffres sont atteint, rendre #1 égal à 1 et
      {\formatintpart@i 1.#3.\numsep#2#4.}% mettre "\numsep#2" en premier et recommencer
      % sinon, mettre #2 en première position et recommencer
      % tout en incrémentant #1 de 1
      {\expandafter\formatintpart@i \number\numexpr#1+1.#3.#2#4.}%
    }%
}
\catcode`\@12
a) \formatintpart{1234567}\qquad b) \formatintpart{987125}\qquad
c) \formatintpart{56}\qquad
d) \edef\foo{\formatintpart{2014}}\meaning\foo
```

a) 7 654 321 b) 521 789 c) 65 d) macro:->4\numsep 102

Comme deux inversions s'annulent, le résultat serait celui que l'on attend si au lieu du nombre passé comme argument à `\formatintpart`, on passait le nombre inversé. On va donc utiliser la macro `\reverse` et faire l'appel à la macro récursive de cette façon :

```
\expandafter\formatintpart@i\expandafter1\expandafter.\reverse{#1}..
```

Le seul ennui est que `\reverse` ne donne pas son résultat en un seul développement ! Mais, du fait de sa structure, l'affichage est fait à la toute fin, après des instructions purement développables ne donnant aucun affichage. La primitive `\romannumeral` va nous servir pour tout développer en un coup jusqu'à l'affichage final. Pour stopper `\romannumeral` *avant* qu'elle ne voit les chiffres et ne s'en empare pour former un nombre romain, nous mettrons `\z@` dans l'argument de `\reverse` telle sorte qu'il stoppe l'expansion et l'action de `\romannumeral`. Comme `\reverse` inverse l'ordre, il faut écrire `\z@` en dernier pour qu'il soit en premier à la fin du traitement :

Code n° V-393

```
\catcode`\@11
\def\formatintpart#1{% #1=série de chiffres
  \expandafter\formatintpart@i\expandafter1\expandafter.\romannumeral\reverse{#1\z@}..%
}

\catcode`\@12
a) \formatintpart{1234567}\qquad b) \formatintpart{987125}\qquad
c) \formatintpart{56}\qquad
d) \edef\foo{\formatintpart{2014}}\meaning\foo
```

a) 1 234 567 b) 987 125 c) 56 d) macro:->2\numsep 014

Mettre le tout en cohérence pour former une macro `\formatnum*` est maintenant aisé :

Code n° V-394

```
\catcode`\@11
\def\ifnodecpart#1{\if@nodecpart#1.\@nil}% teste si #1 est un entier
\def\if@nodecpart#1.#2\@nil{\ifempty{#2}}

```

```
\def\formatnum#1{%
  \ifnodecpart{#1}% s'il n'y a pas de partie décimale
    {\formatintpart{#1}}% formatter la partie entière
    {\formatnum@i#1\@nil}% sinon, formatter les deux parties
}

\def\formatnum@i#1.#2\@nil{%
  \formatintpart{#1}% formatte la partie entière
  \formatdecpart{#2}% et la partie décimale
}

\catcode`\@12
a) \formatnum{3.1415926}\qquad
b) \formatnum{1987654.12301}\qquad
c) \edef\foo{\formatnum{0987654.12300}}$\foo$
```

a) 3,141 592 6 b) 1 987 654,123 01 c) 0 987 654,123 00

Un défaut de notre macro saute aux yeux avec le dernier cas ; elle ne supprime pas les zéros inutiles. Et comme il est impossible d'utiliser `\number` qui l'aurait fait, il va donc falloir écrire deux macros, purement développables, qui suppriment les zéros inutiles d'un nombre entier lorsqu'ils sont au début pour l'une et à la fin pour l'autre.

3.1.3. Supprimer les 0 inutiles de gauche

Voici la macro `\removefirstzeros`⋆, purement développable, qui supprime les 0 au début de son argument :

Code nº V-395

```
\catcode`\@11
\def\removefirstzeros#1{%
  \removefirstzeros@i#1\quark% ajoute "\quark" en dernier
}
\def\removefirstzeros@i#1{% #1=chiffre courant
  \ifx\quark#1% fin atteinte donc nombre = 0
    \expandafter0% laisser un zéro
  \else
    \ifx0#1% si le chiffre lu est un 0
      \expandafter\expandafter\expandafter\removefirstzeros@i% recommencer
    \else% sinon remettre le chiffre #1 et tout afficher jusqu'à \removefirstzeros@i
      \expandafter\expandafter\expandafter\removefirstzeros@ii
      \expandafter\expandafter\expandafter#1%
    \fi
  \fi
}
\def\removefirstzeros@ii#1\quark{#1}
\catcode`\@12
a) \removefirstzeros{000325478}\qquad
b) \removefirstzeros{00000}\qquad
c) \edef\foo{\removefirstzeros{001000}}\meaning\foo\qquad
d) \long\def\>#1<{\detokenize{#1}}
   \expandafter\>\romannumeral\removefirstzeros{0123}<
```

a) 325478 b) 0 c) macro:->1000 d) cxxiii

La macro fonctionne bien sauf qu'au dernier cas, la forcer à donner le résul-

tat en un développement avec \romannumeral est un échec puisque justement, les caractères que laisse la macro sont des chiffres qui sont derechef convertis en un nombre romain. L'idéal serait de trouver une astuce pour rendre cette macro développable avec \romannumeral sans rien changer à la macro. La réponse, non encore vue jusqu'à présent tient dans le dernier point de la règle de la page 125.

Rappelons-nous que cette règle stipule que si TEX s'attend à lire un nombre, les caractères

```
-‘\@
```

sont lus comme la représentation interne de l'entier -64 car le code de caractère de @ est 64 [1]. Mais le point intéressant de cette règle est que TEX *continue* le développement maximal après ces caractères jusqu'à trouver un espace (qu'il absorbe) ou tout autre token (qu'il laisse tel quel). On comprend tout le bénéfice que l'on peut tirer de cette règle ici. En effet, \removefirstzeros ne dirige aucun token vers l'affichage tant qu'il n'a pas trouvé autre chose que 0. Par conséquent, si

```
\romannumeral-‘\@
```

est placé devant \removefirstzeros, le développement maximal sera en route jusqu'à ce que quelque chose soit affiché, c'est-à-dire après avoir mangé les 0 inutiles. De plus, \romannumeral évaluant le nombre négatif -‘\@, aucun chiffre romain ne sera dirigé vers l'affichage.

91 - Règle

Lorsqu'il est 1-développé, le code suivant

```
\romannumeral-‘\⟨caractère⟩
```

effectue l'action « tout développer au maximum jusqu'à trouver un token (absorbé si c'est un espace et laissé tel quel sinon) qui stoppera le développement ».

Code nº V-396

```
1 \long\def\>#1<{\detokenize{#1}}
2 \expandafter\>\romannumeral-‘\@\removefirstzeros{000123}<
```

```
123
```

3.1.4. Supprimer les 0 inutiles de droite

Maintenant, il nous faut écrire la partie plus difficile, celle d'enlever les zéros inutiles finaux. Pour cela, inutile de tout réinventer, nous avons déjà tout ce qu'il nous faut : il suffit d'inverser l'argument, d'enlever tous les zéros de gauche puis d'inverser encore le résultat obtenu pour arriver à nos fins. Rien n'empêche d'utiliser plusieurs \romannumeral imbriqués qui se passent la main afin de provoquer le développement maximal dans le bon ordre.

On souhaiterait écrire

```
\reverse{\removefirstzeros{\reverse{⟨nombre⟩}}}
```

1. On aurait tout aussi bien pu écrire « -‘\a » qui représente -97 ou encore tout nombre négatif ou nul de la forme « -‘\⟨*car*⟩ ».

mais ça serait oublier que TEX n'évalue pas ses arguments avant que la macro ne les lise. Il nous incombe donc de forcer le développement dans le bon ordre, c'est-à-dire développer `\reverse{⟨nombre⟩}` en premier, puis `\removefirstzeros` pour que le `\reverse` chapeau agisse sur le nombre purgé de ses 0 de gauche. De plus, `\removefirstzeros` doit être ré-écrit, car cette macro laissait un 0 si le nombre était nul, et ce comportement n'est pas souhaitable ici. La macro `\removelastzeros@i` rétablit le comportement souhaité.

Toute la TEXnicité réside dans la macro chapeau `\removelastzeros⋆` où, par le truchement de `\romannumeral`, les arguments sont développés dans le bon ordre.

Code n° V-397

```
\catcode`\@11
\def\removelastzeros#1{%
  \exparg\reverse% inverser après
    {\romannumeral-`\.% tout développer
      \expandafter\removelastzeros@i% enlever les 0 de gauche après
      \romannumeral\reverse{#1\z@}\quark% avoir inversé #1
    }%
}
\def\removelastzeros@i#1{% enlève tous les 0 de gauche
  \unless\ifx\quark#1% si la fin n'est pas atteinte
    \ifx0#1% si le chiffre lu est un 0
      \expandafter\expandafter\expandafter\removelastzeros@i% recommencer
    \else% sinon remettre le chiffre et tout afficher jusqu'à \removefirstzeros@i
      \expandafter\expandafter\expandafter\removelastzeros@ii
      \expandafter\expandafter\expandafter#1%
    \fi
  \fi
}
\def\removelastzeros@ii#1\quark{#1}
\catcode`\@12
a) \removelastzeros{0003254780}\qquad
b) \removelastzeros{00000}\qquad
c) \edef\foo{\removelastzeros{001000}}\foo\qquad
\long\def\>#1<{\detokenize{#1}}
d) \expandafter\>\romannumeral-`\.\removelastzeros{012300}<
```

a) 000325478 b) c) 001 d) 0123

3.1.5. Gérer le signe du nombre

Nous voyons bientôt le bout du tunnel, il nous reste à parler du signe du nombre. Pour rester cohérent avec TEX, nous allons admettre qu'un nombre peut commencer par autant de signes + et –, mais que le signe mathématique du nombre affiché sera négatif si les signes « – » sont en nombre impair. Pour cela, la macro `\formatnum` va devoir transmettre son argument à une macro `\formmatnum@i`, à arguments délimités de type « réservoirs communicants ». Ici, le délimiteur sera « `!` » puisque le « `.` » est susceptible d'être contenu dans le nombre. La macro chapeau effectuera cet appel initial à la macro récursive :

```
\def\formatnum#1{\formatnum@i 1!#1!}
```

Le premier argument délimité, qui tient lieu de réservoir n° 1, vaut 1 lors de l'appel initial. Ce réservoir accumulera en première position tous les signes rencontrés au début du réservoir n° 2 qui situé entre les deux points d'exclamation, dont

la valeur initiale est le ⟨*nombre*⟩ à formater. La macro `\formatnum@i` va donc avoir un texte de paramètre défini ainsi

```
\def\formatnum@i#1!#2#3!{...}
```

de façon à ce que #2 soit le premier caractère du nombre à examiner et #3 soit les caractères non encore traités. Il suffit de tester si ce caractère #2 est + ou – et dans ce cas, le mettre en première position dans le réservoir n° 1 et recommencer avec les arguments qui seront devenus « #2#1 » et #3 ». Si ce n'est pas le cas, cela veut dire que l'on a épuisé les signes situés au début du nombre. Il est alors temps de tester si le nombre dans le réservoir n° 1, constitué de tous les signes rencontrés suivis de 1, est lu par TEX comme égal à −1 et dans ce cas, afficher un « – » puis passer la main à la mise en forme du nombre proprement dite.

On remarquera que si le nombre n'est constitué que de signes + ou –, +1 ou −1 sera lu dans le premier réservoir selon la parité du nombre de signes –. Enfin, dernier raffinement, une macro se chargera de remplacer dans l'argument de `\formatnum` le séparateur décimaux « virgule » en « point » avant de traiter le nombre. Ainsi, la virgule ou le point peuvent indifféremment être utilisés comme séparateur décimal.

Voici le code complet :

Code n° V-398

```
\catcode`\@11
\protected\def\numsep{\kern0.2em }% \numsep est le séparateur mis tous les 3 chiffres

\def\formatdecpart#1{% #1=série de chiffres
  \ifempty{#1}% si la partie décimale est vide
    {}% ne rien afficher
    {{,}\formatdecpart@i 1.#1..}% sinon, afficher la virgule et mettre en forme
}

% #1=compteur de caractères #2= chiffre courant
% #3= chiffres restants     #4 = chiffres déjà traités
\def\formatdecpart@i#1.#2#3.#4.{%
  \ifempty{#3}% si #2 est le dernier chiffre
    {#4#2}% le mettre en dernière position et tout afficher, sinon
    {\ifnum#1=3 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      % si 3 chiffres sont atteint, rendre #1 égal à 1 et
      {\formatdecpart@i 1.#3.#4#2\numsep.}% mettre #2\numsep en dernier puis recommencer
      % sinon, mettre #2 en dernière position et recommencer
      % tout en incrémentant #1 de 1
      {\expandafter\formatdecpart@i \number\numexpr#1+1.#3.#4#2.}%
    }%
}

\def\formatintpart#1{% #1=série de chiffres
  \expandafter\formatintpart@i\expandafter1\expandafter.%
  \romannumeral\reverse{#1\z@}..% appelle la macro récursive
}

% #1=compteur de caractères #2= chiffre à déplacer
% #3= chiffres restants     #4 = chiffres déjà traités
\def\formatintpart@i#1.#2#3.#4.{%
  \ifempty{#3}% si #2 est le dernier chiffre à traiter
    {#2#4}% le mettre en première position et tout afficher, sinon
    {\ifnum#1=3 \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      % si 3 chiffres sont atteint, rendre #1 égal à 1 et
      {\formatintpart@i 1.#3.\numsep#2#4.}% mettre \numsep#2 en premier puis recommencer
      % sinon, mettre #2 en dernière position et recommencer
```

```
      % tout en incrémentant #1 de 1
      {\expandafter\formatintpart@i\number\numexpr#1+1.#3.#2#4.}%
    }%
}
\def\removefirstzeros#1{%
  \removefirstzeros@i#1\quark% ajoute "\quark" en dernier
}
\def\removefirstzeros@i#1{% #1=chiffre courant
  \ifx\quark#1% fin atteinte donc nombre = 0
    \expandafter0% laisser un zéro
  \else
    \ifx0#1% si le chiffre lu est un 0
      \expandafter\expandafter\expandafter\removefirstzeros@i% recommencer
    \else% sinon remettre le chiffre #1 et tout afficher jusqu'à \removefirstzeros@i
      \expandafter\expandafter\expandafter\removefirstzeros@ii
      \expandafter\expandafter\expandafter#1%
    \fi
  \fi
}
\def\removefirstzeros@ii#1\quark{#1}

\def\removelastzeros#1{%
  \exparg\reverse% inverser après
    {\romannumeral-`\.% tout développer
      \expandafter\removelastzeros@i% enlever les 0 de gauche après
      \romannumeral\reverse{#1\z@}\quark% avoir inversé #1
    }%
}
\def\removelastzeros@i#1{% enlève tous les 0 de gauche
  \unless\ifx\quark#1% si la fin n'est pas atteinte
    \ifx0#1% si le chiffre lu est un 0
      \expandafter\expandafter\expandafter\removelastzeros@i% recommencer
    \else% sinon remettre le chiffre et tout afficher jusqu'à \removefirstzeros@i
      \expandafter\expandafter\expandafter\removelastzeros@ii
      \expandafter\expandafter\expandafter#1%
    \fi
  \fi
}
\def\removelastzeros@ii#1\quark{#1}

% renvoie vrai s'il n'y a pas de partie décimale
\def\if@nodecpart#1.#2\@nil{\ifempty{#2}}

\def\formatnum#1{\formatnum@i1!#1!}

% #1 = <série de signes> suivie de "1"
% #2 = caractère courant
% #3 = caractères non traités
\def\formatnum@i#1!#2#3!{%
  \ifx+#2\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\ifempty{#3}% si #2=+ et #3 est vide
      {\number#1 }% afficher "+1" ou "-1" (il n'y avait que des signes)
      {\formatnum@i#2#1!#3!}% sinon, mettre le signe #2 devant #1
    }
    {\ifx-#2\expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {\ifempty{#3}% si #2=- et #3 est vide
        {\number#1 }% afficher "+1" ou "-1" (il n'y avait que des signes)
        {\formatnum@i#2#1!#3!}% sinon, mettre le signe #2 devant #1
      }% #2 est le dernier caractère qui n'est pas un signe + ou -
      {\ifnum#1<0 -\fi% si "<signes>1" est <0 afficher un "-"
```

```
      \formatnum@ii{#2#3}% formatter le nombre formé avec "#2#3""
      }%
    }%
}

\def\formatnum@ii#1{%
  \if@comma{#1}% si #1 comporte une virgule
    {\formatnum@iii#1\@nil}% la remplacer par un "." et recommencer
    {\if@nodecpart#1.\@nil% puis si les chiffres restants sont un entier
      {\formatintpart{#1}}% formatter l'entier
      {\formatnum@iv#1\@nil}% sinon, formatter le nombre
    }%
}

\def\formatnum@iii#1,#2\@nil{\formatnum@ii{#1.#2}}

\def\formatnum@iv#1.#2\@nil{% formatte le nombre décimal #1.#2
  \exparg\formatintpart{\romannumeral-`\.\removefirstzeros{#1}}%
  \exparg\formatdecpart{\romannumeral-`\.\removelastzeros{#2}}%
}

\def\if@comma#1{\if@comma@i#1,\@nil}% teste la présence d'un virgule dans #1
\def\if@comma@i#1,#2\@nil{\ifempty{#2}\secondoftwo\firstoftwo}
\catcode`\@12

a) \formatnum{3,14159265358979323846264338327950288419716939937510582097494459 2}\par
b) \formatnum{---+69874}\qquad
c) \formatnum{0032100,98000}\qquad
d) \formatnum{+++010.01100}\qquad
e) \formatnum{-+--+}\qquad
f) \formatnum{---00.0000}\qquad
g) \formatnum{+99,0000}\qquad
h) \formatnum{.123456}\par
i) \edef\foo{\formatnum{-+-+-010500,090900}}\meaning\foo
```

a) 3,141 592 653 589 793 238 462 643 383 279 502 884 197 169 399 375 105 820 974 944 592
b) -69 874 c) 32 100,98 d) 10,011 e) -1 f) -0 g) 99 h) 0,123 456
i) macro:->-10\numsep 500{,}090\numsep 9

3.2. Permettre à une boite encadrée de franchir des pages

3.2.1. Couper une boite verticale

Cela avait été clairement annoncé à la page 21 : toute boite construite avec les primitives \hbox, \vbox ou \vtop est *insécable*. C'est effectivement le cas à un petit mensonge près… Si une boite *verticale* est stockée dans un registre de boite, il est possible de couper la boite ainsi stockée en deux boites verticales. La primitive qui permet cette opération est `\vsplit` [2].

2. En revanche, les boites horizontales \hbox sont *vraiment* insécables, il n'existe hélas pas de primitive `\hsplit`.

92 - Règle

Si ⟨*x*⟩ est un registre de boite auquel on a assigné une boite verticale (construite avec `\vbox` ou `\vtop`) et si ⟨*y*⟩ est un numéro de registre de boite, alors

`\setbox` ⟨*y*⟩`=\vsplit` ⟨*x*⟩ `to` ⟨*dimension*⟩

va commander à TEX de parcourir la boite contenue dans le registre n° ⟨*x*⟩ et opérer la meilleure coupure possible pour générer une boite de hauteur ⟨*dimension*⟩. Le registre n° ⟨*y*⟩ contiendra cette nouvelle boite tandis que le registre ⟨*x*⟩ contiendra la boite initiale amputée d'autant. Si la ⟨*dimension*⟩ est suffisamment grande pour que la coupure englobe la totalité de la boite initiale ⟨*x*⟩, le registre ⟨*x*⟩ devient vide, c'est-à-dire positif au test `\ifvoid`.

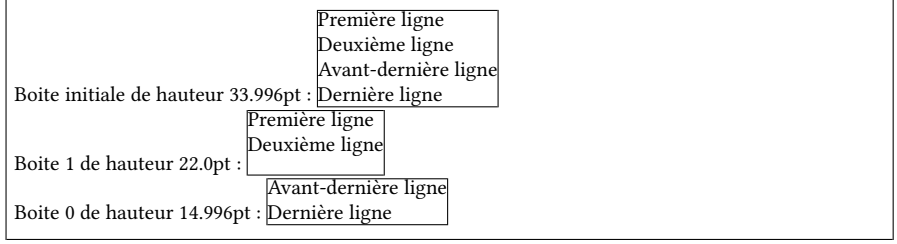
Le mécanisme de coupure de boite verticale est similaire à celui des coupures de pages. En particulier, les objets « volatils » sont supprimés à la coupure (pénalités et espaces verticaux de type `\kern` ou `\vskip`).

Enfin, un ressort appelé `\splittopskip` est inséré au sommet de la boite restante ⟨*x*⟩.

Le code ci-dessous illustre cette règle, où la boite initiale contient 4 lignes, chacune enfermée dans une `\hbox` :

Code n° V-399

```
\setbox0=\vbox{%
  \hbox{Première ligne}
  \hbox{Deuxième ligne}
  \hbox{Avant-dernière ligne}
  \hbox{Dernière ligne}
}

\frboxsep=0pt % aucun espace entre le contenu et l'encadrement
Boite initiale de hauteur \the\ht0 {} : \frbox{\copy0 }
\splittopskip0pt % ne rajouter aucun espace au sommet de la boite restante
\setbox1=\vsplit0 to 22pt % couper la boite à 22pt de hauteur

Boite 1 de hauteur \the\ht1 {} : \frbox{\box1 }

Boite 0 de hauteur \the\ht0 {} : \frbox{\box0 }
```



Il faut noter qu'un message d'avertissement est émis lorsque la coupure est faite : « `Underfull \vbox (badness 10000) detected` ». Ce message indique que la boite n° 1 est insuffisamment remplie, ce que l'on peut constater visuellement. L'explication tient au fait que seules deux lignes ont pu être logées verticalement dans 22 points. Mais ces deux lignes ne remplissent pas la totalité de ces 22 points qui sont imposés. Pour éviter de garder une boite insuffisamment pleine, on peut utiliser

```
\setbox1=\vbox{\unvbox1 }
```

afin que la boite n° 1 soit « extraite » de la \vbox de hauteur imposée et, ayant repris sa dimension naturelle, soit à nouveau enfermée dans une \vbox.

Pour empêcher TEX d'émettre un message d'avertissement lors de la coupe de la boite n° 0, il faut agir sur l'entier \vbadness qui représente le seuil de médiocrité au-delà duquel un message d'avertissement n'est plus émis concernant une boite verticale (\hbadness est le pendant pour les boites horizontales). Nous allons donc sauvegarder cet entier puis lui assigner l'entier 10 000 qui représente la médiocrité maximale, et après avoir coupé la boite avec \vsplit, restaurer \vbadness à la valeur initiale.

Code n° V-400

```
\setbox0=\vbox{%
  \hbox{Première ligne}
  \hbox{Deuxième ligne}
  \hbox{Avant-dernière ligne}
  \hbox{Dernière ligne}
}

\edef\restorevbadness{\vbadness=\the\vbadness\relax}% restaurera le \vbadness
\vbadness=10000 % plus d'avertissement pour boite verticale
\frboxsep=0pt % aucun espace entre le contenu et l'encadrement
Boite initiale de hauteur \the\ht0 {} : \frbox{\copy0 }
\splittopskip0pt % ne rajouter aucun espace au sommet de la boite restante
\setbox1=\vsplit0 to 22pt % couper la boite à 22pt de hauteur
\setbox1=\vbox{\unvbox1 }% la boite 1 prend la hauteur naturelle
\restorevbadness\relax% restaure le \vbadness

Boite 1 de hauteur \the\ht1 {} : \frbox{\box1 }

Boite 0 de hauteur \the\ht0 {} : \frbox{\box0 }
```

Boite initiale de hauteur 33.996pt : Première ligne / Deuxième ligne / Avant-dernière ligne / Dernière ligne

Boite 1 de hauteur 14.996pt : Première ligne / Deuxième ligne

Boite 0 de hauteur 14.996pt : Avant-dernière ligne / Dernière ligne

On constate bien que la boite n° 1 a repris la hauteur naturelle et devient donc la boite contenant le plus de lignes possible dans 22pt de hauteur.

Il est en revanche assez inexplicable que la somme des deux hauteurs ne soit pas égale à la hauteur de la boite initiale. Il manque

$$34.024\texttt{pt} - 2 \times 15.024\texttt{pt} \approx 4\texttt{pt}$$

Cette hauteur manquante est celle qui sépare la frontière inférieure de la boite n° 1 de la frontière supérieure de la boite n° 0. Ce ressort d'interligne a disparu à la coupure, comme le font tous les ressorts aux coupures de page ou de ligne.

3.2.2. Couper à la bonne hauteur

Avant de bâtir un algorithme permettant de composer un texte dans un cadre qui franchit des pages, il reste tout de même une question essentielle : comment

connaitre la hauteur qui reste à composer dans la page en cours ? Deux nouvelles primitives de TEX vont nous permettre de répondre à cette question :

- `\pagetotal` est la dimension accumulée verticalement jusqu'à la fin du précédent paragraphe entièrement composé ;
- `\pagegoal` est la dimension verticale du texte à composer dans la page en cours.

La dimension `\pagetotal` n'est pas actualisée en permanence, mais seulement après chaque paragraphe composé. Par ailleurs, au tout début de la construction d'une page, lorsqu'elle est vide, `\pagetotal` vaut 0 pt alors que `\pagegoal` vaut 16 383,999 99 pt [3], soit `\maxdimen`. Par conséquent, notre futur algorithme devra tenir compte de cette spécificité au début des pages. Afin que `\pagegoal` prenne une valeur reflétant la hauteur de la page, il faudra donc, avant de mesurer quoi que ce soit, faire en sorte que la nouvelle page ne soit pas vide . Insérer le matériel vertical « `\hbox{}\nointerlineskip` » tiendra lieu de remplissage de haut de page, tout en ne prenant aucune place verticalement. Ainsi, après cette opération, `\pagegoal` sera égal à la hauteur de la zone de texte en haut d'une page. Nous sommes maintenant assurés que, quelles que soient les circonstances, la différence entre `\pagegoal` et `\pagetotal` sera la hauteur de l'espace vertical disponible restant dans la page en cours.

Ces deux primitives étaient ce qui nous manquait pour construire l'algorithme permettent de couper n fois une boite verticale pour la loger sur $n + 1$ pages.

Nous allons d'abord décider que le contenu à afficher dans un cadre sera compris entre les macros `\breakpar*` et`\endbreakpar`. La première macro sera chargée d'assigner à un registre de boite verticale tout ce qui se trouvera jusqu'à `\endbreakpar`. Pour cela, cette macro utilisera `\bgroup` au lieu de « { » pour délimiter le début de la boite alors que `\endbreakpar` utilisera `\egroup` pour marquer la fin de la boite verticale. La macro `\breakpar` aura également pris soin, en début de boite, de diminuer la dimension horizontale `\hsize` de cette boite. En effet, la largeur du texte à encadrer doit tenir compte de l'épaisseur de la réglure d'encadrement `\frboxrule` et de l'espace entre le texte et cette réglure `\frboxsep`, chacune de ces dimensions devant être comptée deux fois, une fois pour la réglure de gauche et une fois pour celle de droite.

En ce qui concerne le cadre proprement dit, il faut bien garder à l'esprit que selon les cas, les encadrements pourront être de 4 types différents :

1. cadre plein (⬚) dans le cas où l'espace restant dans la page en cours permet de loger le cadre dans sa totalité ;

Si le cadre ne peut loger sur la page en cours, 3 autres types sont possibles :

2. cadre de début (⎾⏋) ;
3. cadre intermédiaire (⏐ ⏐) dans le cas où le texte à encadrer est tellement long qu'il faut une ou plusieurs pages intermédiaires entre le début du cadre et la fin ;
4. cadre de fin (⎿⏌).

Pour tracer ces cadres, la macro `\framebox` vue à la page 393 va nous être d'un grand secours, il suffira de spécifier le bon argument optionnel pour avoir le cadre du type souhaité.

Dans les grandes lignes, l'algorithme va fonctionner de cette manière :

3. Lire le TEXbook page 133 et plus généralement le chapitre 15 en entier.

1) capturer et composer dans une \vbox de largeur appropriée tout ce qui se trouve entre \breakpar et \endbreakpar, stocker cette boite dans le registre \remainbox et aller au point n° 2 ;
2) macro \splitbox*
   a) si \remainbox est vide : fin du processus
   b) sinon, aller au point n° 3.
3) macro \splitbox@i
   a) si \remainbox, une fois encadré, loge sur l'espace restant dans la page, l'encadrer avec le cadre n° 1 ou n° 4, et fin du processus ;
   b) sinon couper \remainbox à une hauteur correspondant à celle de l'espace restant sur la page ; encadrer le registre de boite obtenu (\partialbox) avec le cadre n° 2 ou n° 3, composer la page en cours et aller au point n° 2 avec la boite restante \remainbox.

Pour mener à bien cet algorithme, nous avons à écrire 5 macros différentes : \breakpar, \endbreakpar, \splitbox, \splitbox@i ainsi qu'une macro qui affiche la boite à encadrer \printpartialbox. Nous devrons correctement gérer l'argument optionnel de \framebox (où les lettres « UDRL » spécifient les réglures à afficher) pour choisir selon les cas, le bon encadrement. Pour ce faire, nous stockerons dans la macro \rule@arg les lettres devant être passées à cet argument optionnel.

Avant d'entrer dans le vif du sujet, il est nécessaire d'examiner ce qui se passe en haut d'une page : afin que dans la mesure du possible, les lignes de base des premières lignes de chaque page se situent toutes à la même position verticale, le ressort \topskip est inséré en haut de chaque page. Sa valeur est diminuée de la hauteur de la première ligne et si le résultat est négatif, le ressort 0pt est inséré. Plain-TEX prend \topskip égal à 10pt ce qui assure que toutes les premières lignes dont la hauteur est inférieure ou égale à 10pt commenceront à la même position verticale dans les pages.

Pour notre problème, si une boite doit être coupée, il ne faut pas qu'un ressort en haut de page soit inséré avant le cadre restant (| | ou ⌊⌋) : le haut des réglures du cadre doit exactement coïncider avec le bord supérieur de la zone de texte. Par conséquent, \topskip sera pris égal à 0pt. Avant de procéder à cette assignation, la valeur de \topskip sera copiée dans \splittopskip qui joue le même rôle que \topskip dans les boites résiduelles après une coupure par \vsplit. Ce faisant, à chaque début de page, on s'assure que la première ligne dans le cadre se situe à la même position verticale que les premières lignes des pages « normales ».

Venons-en au cœur du problème : à quelle hauteur allons-nous demander la coupure de la boite \remainbox ? Au premier abord, on pourrait être tenté de répondre \pagegoal-\pagetotal mais ce serait ignorer l'encombrement vertical de \frboxsep+\frboxrule, requis par les éventuelles réglures (supérieures ou inférieures) d'épaisseur \frboxrule et les espaces \frboxsep qui les suivent ou les précèdent. Il va de soi que selon les cas, ces réglures sont présentes soit toutes les deux (cas du cadre □), soit seulement une des deux (cadre ⌜⌝ ou ⌊⌋), soit aucune (cadre | |). Nous devons être rigoureux pour ne pas demander trop ou trop peu de hauteur de coupure.

Pour nous faciliter la tâche, nous allons stocker dans la macro \coeff@rule le nombre de réglures horizontales qui restent à afficher. Dans la macro chapeau, il

sera initialisé à 2 puisque les deux réglures sont encore à afficher. Ce coefficient sera ensuite pris égal à 1 dès que la première coupure sera faite puisque la réglure supérieure ne sera plus à afficher.

Du côté du calcul, il faut décider si une coupure de `\remainbox` doit avoir lieu ou pas. La réponse est positive si sa dimension verticale totale (hauteur + profondeur) est supérieure à l'espace disponible sur la page en cours auquel on a soustrait l'espace requis pour les réglures horizontales restantes. Mathématiquement, une coupure doit être faite si `\ht\remainbox+\dp\remainbox` est strictement supérieur à `\pagegoal − \pagetotal − (\frboxsep + \frboxrule) × \coeff@rule`. Nous appellerons D_v cette dernière quantité et la stockerons dans un registre de dimension appelé `\cut@ht`.

Si la coupure doit être faite, à quelle hauteur doit être coupée `\remainbox` ? Comme une coupure est faite, la réglure horizontale de fin de cadre ne se trouvera pas sur la page recevant la boite coupée. Par conséquent, la hauteur de coupure est D_v augmentée de `\frboxsep + \frboxrule`.

Voici comment va opérer la macro `\splitbox@i` :

- le registre de dimension `\cut@ht` reçoit la hauteur D_v ;
- si une coupure doit être faite, la dimension `\frboxsep + \frboxrule` est ajoutée à `\cut@ht` ;
- `\remainbox` est coupée à la hauteur `\cut@ht` et la boite obtenue est stockée dans le registre `\partialbox` ;
- on rend `\coeff@rule` égal à 1 puisque la réglure supérieure ne peut plus être présente ;
- si `\splitbox@i` détecte que la coupure n'est pas nécessaire, `\remainbox` sera copiée dans `\partialbox` et `\cut@ht` reçoit la hauteur verticale totale de `\partialbox` ;

Nous sommes bientôt au bout du tunnel, mais il y a un dernier raffinement auquel nous devons penser. Comment s'assurer que les réglures verticales des encadrements « ⌐¬ » et « | | » iront jusqu'en bas de la page ? En effet, si nous encadrons avec

```
\expandafter\framebox\expandafter[\rule@arg]{\vbox{\unvbox\partialbox}}
```

le « `\vbox{\unvbox\partialbox}` » va redonner à `\partialbox` sa hauteur naturelle (plus petite que `\cut@ht`) et l'encadrement n'atteindra pas exactement le bas de la page. Pour nous prémunir de cette petite perte de hauteur, nous devons imposer un cadre de hauteur `\cut@ht`, car ce registre contient la hauteur totale verticale du texte à encadrer. On écrira donc dans l'argument de `\framebox`

```
\vbox to\cut@ht{\unvbox\partialbox\vss}
```

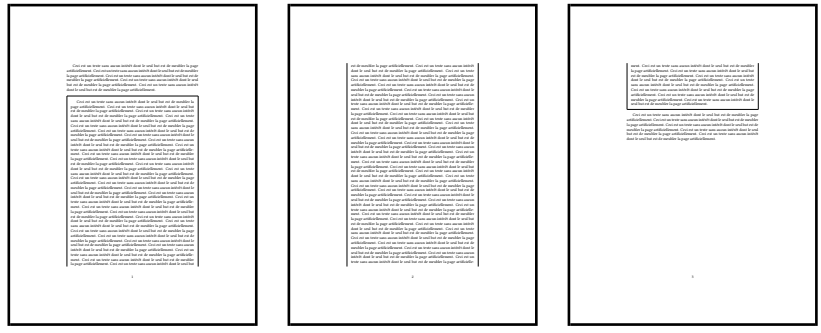
Le ressort `\vss` se charge de rattraper la différence entre `\cut@ht` et la hauteur naturelle de `\partialbox`.

Le code ci-dessous est un de ceux qui dérogent à ce qui est la règle dans ce livre. Le résultat du code n'est pas présenté celui qui découle de la compilation de ce code, car pour des raisons de place, les pages obtenues ont été réduites et présentées les unes à côté des autres.

Code nº V-401

```
\catcode‘@11
\newbox\remainbox
\newbox\partialbox
\newdimen\cut@ht

\def\breakpar{%
  \par\nointerlineskip% termine le paragraphe précédent
  \vskip\frboxsep\relax% et saute une petite espace verticale
  \begingroup
    \splittopskip\topskip% \topskip en haut des boites coupées
    \topskip=0pt % neutraliser le \topskip
    % nbre de réglures horizontales contribuant à l'encadrement restant (2 au début)
    \def\coeff@rule{2}%
    \setbox\remainbox=\vbox\bgroup% compose la boite après avoir...
      \advance\hsize by -2\dimexpr\frboxrule+\frboxsep\relax% ajusté sa largeur
}

\def\endbreakpar{%
    \egroup% fin de la composition de la boite
    \def\rule@arg{ULR}% prendre l\cut@htes réglures d'encadrement haute, gauche et droite
    \splitbox% puis, aller à l'algorithme de coupure
  \endgroup% une fois fini, sortir du groupe semi-simple
}

\def\splitbox{%
  \ifvoid\remainbox% si la boite est vide, c'est la fin du processus
    \par\nointerlineskip% termine le paragraphe précédent
    \vskip\frboxsep\relax% et saute un petit espace vertical
  \else% sinon
    \expandafter\splitbox@i% aller à \splitbox@i
  \fi
}

\def\splitbox@i{%
  \hbox{}% composer un noeud en mode vertical
  \nointerlineskip% pas de ressort d'interligne
  % calculer la dimension verticale disponible dans la page pour le texte de la boite
  \cut@ht=\dimexpr\pagegoal-\pagetotal-(\frboxsep+\frboxrule)*\coeff@rule\relax
  % si dimension totale du texte > dimension disponible pour le texte
  \ifdim\dimexpr\ht\remainbox+\dp\remainbox>\cut@ht% si une coupure doit être faite
      \advance\cut@ht\dimexpr%   augmenter Dv de l'espace verticale libérée
        +\frboxsep+\frboxrule% par la réglure inférieure qui n'est pas sur cette page
        \relax
    \edef\old@vbadness{\the\vbadness}% sauvegarder \badness
    \vbadness=10000 % désactive les avertissement lors de la coupure
    \def\coeff@rule{1}% ne prendre en compte que réglure D + espace D
    \setbox\partialbox=\vsplit\remainbox to\cut@ht% coupe à la hauteur calculée
    % \partialbox retrouve sa hauteur naturelle
    \setbox\partialbox=\vbox{\unvbox\partialbox}%
    \vbadness=\old@vbadness\relax% restaure \vbadness
    \printpartialbox% imprime la boite partielle
    \vfill\eject% et compose la page en cours
  \else% si une coupure n'est pas nécessaire :
    \setbox\remainbox\vbox{\unvbox\remainbox}% reprendre la hauteur naturelle
    \setbox\partialbox=\box\remainbox% \partialbox devient \remainbox
                                     % et cette dernière devient vide
    \cut@ht=\dimexpr\ht\partialbox+\dp\partialbox\relax% hauteur à encadrer
    \edef\rule@arg{\rule@arg D}% ajouter "D" aux réglures à tracer
    \printpartialbox% afficher la boite restante
  \fi
```

```
  \splitbox
}

\def\printpartialbox{% imprime \partialbox
  \expandafter\framebox\expandafter[\rule@arg]{%
    \vbox to\cut@ht{\unvbox\partialbox\vss}}%
  \def\rule@arg{LR}% ne mettre que les réglures d et g
}

\def\dummytext#1{%
  \for\xx=1to#1\do% composer #1 fois la phrase suivante :
    {Ceci est un texte sans aucun int\'er\^et dont le seul but est de meubler
    la page artificiellement.
    }%
}

\catcode`@12
\dummytext{5}
\frboxsep=5pt

\breakpar
  \dummytext{70}
\endbreakpar

\dummytext{5}
```



3.2.3. Couper un paragraphe en lignes

Maintenant que le mécanisme de coupure à l'aide de `\vsplit` est un peu maitrisé, nous allons le mettre à profit pour couper une boite verticale en lignes, chacune d'entre elles étant contenue dans une boite. L'énorme intérêt est qu'il devient alors possible d'effectuer une action pour chaque *ligne* d'un ou plusieurs paragraphes. On pourra par exemple les numéroter ou afficher quelque chose dans la marge de gauche ou de droite. Il deviendra même possible d'agir sur la boite elle-même, pour l'encadrer par exemple.

À quelle hauteur faudra-t-il couper la boite contenant le tout pour n'obtenir qu'une seule ligne ? La question mérite d'être posée car nous n'avons aucune indication sur la hauteur d'une ligne. La méthode consiste à demander une coupure à une hauteur nulle. Cela va évidemment provoquer une « `overful vbox` » puisque rien ne peut tenir dans `0pt` de hauteur. Il va donc falloir, le temps de la coupure, désactiver le mécanisme qui guette si une boite est trop remplie. TEX dispose de la primitive `\vfuzz` qui contient une dimension, seuil au-delà duquel est émis un

avertissement lorsqu'une boite verticale déborde (\hfuzz est l'équivalent pour les boites horizontales). Il suffira donc de dire que \vfuzz=\maxdimen pour s'assurer qu'aucun avertissement n'est émis lors du débordement d'une boite verticale, quelle que soit la valeur de ce débordement.

Code n° V-402

```
\setbox0=\vbox{%
  \hsize=5cm
  Ceci est un texte sans aucun intérêt dont le seul but est de meubler
  la page de façon artificielle.
}

Boite 0 : \copy0 % affiche la boite totale
\medbreak

\edef\restorevfuzz{\vfuzz=\the\vfuzz\relax}% appelée après la coupure
\vfuzz=\maxdimen% annule les avertissements pour débordement
\splittopskip=0pt % ne rajouter aucun espace au sommet de la boite restante
\setbox1=\vsplit0 to 0pt % couper la boite à 0pt de hauteur
\restorevfuzz% restaurer \vfuzz
\setbox1=\vbox{\unvbox1}% redonner à la boite sa hauteur d'origine

Boites 1+0 : \vbox{%
  \offinterlineskip% annule le ressort d'interligne
  \box1 % affiche la première ligne
  \box0 %affiche les lignes restantes
}
```

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon ar-
Boite 0 : tificielle.

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon ar-
Boites 1+0 : tificielle.

Comme on l'a déjà constaté, les ressorts disparaissent lors des coupures et donc ici, nous avons perdu le ressort d'interligne entre la 1re et la 2e ligne. Si l'on veut afficher les lignes une par une, telles qu'elles auraient été affichées lors d'une composition normale, on doit absolument récupérer les ressorts mangés lors des coupures pour les insérer entre chaque ligne. Si l'on appelle \htbefore la hauteur de la boite n° 0 avant la coupure, alors, la longueur verticale du matériel mangé à la coupure est

```
\htbefore-(\ht0+\dp0+\ht1+\dp1)
```

On peut le vérifier :

Code n° V-403

```
\def\vdim#1{\dimexpr\ht#1+\dp#1\relax}% hauteur totale de la boite #1
\setbox0=\vbox{%
  \hsize=5cm Ceci est un texte sans aucun intérêt dont le seul but est
  de meubler la page de façon artificielle.
}
\edef\htbefore{\the\vdim0}% hauteur de la boite 0

Boite 0 : \copy0 % affiche la boite totale
\medbreak

\edef\restoreparam{%
```

```
  \vfuzz=\the\vfuzz\relax% sauvegarde le \vfuzz
  \splittopskip=\the\splittopskip% et \splittopskip
}%
\vfuzz=\maxdimen% annule les avertissements pour débordement
\splittopskip=0pt % ne rajouter aucun espace au sommet de la boite restante
\setbox1=\vsplit0 to 0pt % couper la boite à 0pt de hauteur
\restoreparam
\setbox1=\vbox{\unvbox1}% redonner à la boite sa hauteur d'origine

\edef\intersplitspace{\the\dimexpr\htbefore-(\vdim0+\vdim1)\relax}%
Boites 1+0 : \vbox{%
  \offinterlineskip% annule le ressort d'interligne
  \box1 % affiche la première ligne
  \vskip\intersplitspace\relax% ajoute le ressort perdu à la coupure
  \box0 % affiche les lignes restantes
}
```

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon ar-
Boite 0 : tificielle.

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon ar-
Boites 1+0 : tificielle.

Ici encore, ε-TEX offre des primitives précieuses qui vont rendre la tâche précédente plus simple [4]. La primitive `\savingvdiscards` a vocation à contenir un entier qui, s'il est positif, autorise la sauvegarde des éléments ignorés lors d'une coupure verticale (les `\kern`, les ressorts verticaux et les pénalités), qu'elle soit une coupure de page ou une coupure de boite verticale par `\vsplit`. On peut ensuite insérer ces éléments dans une liste verticale avec `\pagediscards` pour les éléments ignorés lors d'une coupure de page et avec `\splitdiscards` pour ceux ignorés lors d'une coupure par `\vsplit`. Ces deux listes d'éléments sauvegardés sont vidées après la routine de sortie ou au début de l'opération `\vsplit`. Elles sont également vidées après avoir été utilisées.

Les macros `\totalht` et `\intersplitspace` du code précédent deviennent dès lors inutiles :

Code n° V-404

```
\setbox0=\vbox{%
  \hsize=5cm Ceci est un texte sans aucun intérêt dont le seul but est
  de meubler la page de façon artificielle.
}

Boite 0 : \copy0 % affiche la boite totale
\medbreak

\edef\restoreparam{%
  \vfuzz=\the\vfuzz\relax% sauvegarde le \vfuzz
  \splittopskip=\the\splittopskip\relax% , le \splittopskip
  \savingvdiscards=\the\savingvdiscards\relax% et le \savingvdiscards
  }%
\vfuzz=\maxdimen% annule les avertissements pour débordement
\splittopskip=0pt % ne rajouter aucun espace au sommet de la boite restante
\savingvdiscards=1 % autorise la sauvagarde des éléments supprimés
\setbox1=\vsplit0 to 0pt % couper la boite à 0pt de hauteur
```

4. Lire le manuel de ε-TEX, chapitre « 3.11 Discarded Items ».

```
\setbox1=\vbox{\unvbox1}% redonner à la boite sa hauteur d'origine
\restoreparam

Boites 1+0 : \vbox{%
  \offinterlineskip% annule le ressort d'interligne
  \box1 % affiche la première ligne
  \splitdiscards% affiche les éléments ignorés
  \box0 % affiche les lignes restantes
}
```

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.
Boite 0 :

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.
Boites 1+0 :

Maintenant que la méthode pour couper *une* ligne a été exposée, il est facile de construire une macro récursive qui coupera successivement les lignes jusqu'à ce qu'il n'y ait plus rien à couper. Ceci sera fait par la macro `\numlines`⋆ :

`\numlines`⟨*texte à couper en lignes*⟩`\endnumlines`

Nous voulons manipuler chaque ligne ainsi coupée, c'est-à-dire avoir la possibilité d'écrire quelque chose à droite et à gauche de chaque ligne, tout en ayant également le contrôle sur la ligne elle-même. Nous allons donc écrire trois macros :

1. les macros `\leftline`⋆ et `\rightline`⋆, de syntaxe
   `\leftline[`⟨*dimension*⟩`]{`⟨*matériel*⟩`}`
   `\rightline[`⟨*dimension*⟩`]{`⟨*matériel*⟩`}`
   qui stockent dans des macros privées le ⟨*matériel*⟩ à écrire à gauche et à droite de chaque ligne. Ce ⟨*matériel*⟩ sera composé dans une `\hbox` de longueur ⟨*dimension*⟩ et dont le ressort `\hss` sera correctement positionné pour le contenu s'étende à gauche ou à droite en débordement ;
2. une macro `\formatline`⋆ dont l'argument `#1` sera la boite contenant la ligne en cours. Par défaut, `\formatline` sera rendue `\let`-égale à `\identity` pour afficher chaque ligne telle quelle. Il sera possible de personnaliser l'affichage en écrivant, par exemple, `\let\formatline=\frbox` pour encadrer chaque ligne.

Chaque ligne sera enfermée dans une `\hbox` afin que le mode vertical principal perdure. Après cette `\hbox`, `\splitdiscards` sera insérée. Ainsi, une ligne sera composée de la façon suivante :

```
\hbox{%
    \hbox to⟨dimension gauche⟩{\hss⟨matériel gauche⟩}%
    \formatline{⟨ligne courante⟩}%
    \hbox to ⟨dimension droite⟩{⟨matériel droit⟩\hss}%
}%
\splitdiscards
```

Insérer `\splitdiscards` dans la liste verticale principale nous empêche de savoir quelle est la dimension verticale totale de la ligne composée. Certes, nous aurions pu placer le tout dans une `\vbox` et stocker cette boite un registre pour accéder à sa dimension verticale. Mais agir ainsi et enfermer `\splitdiscards` aurait empêché le matériel contenu dans `\splitdiscards` de disparaitre aux coupures de

page.

Pourtant, accéder à la dimension verticale totale de la ligne composée peut s'avérer nécessaire pour correctement dimensionner les matériels affichés à droite et à gauche de la ligne courante pour obtenir certains effets. Nous allons donc agir comme au code de la page 455 et calculer `\htbefore` pour obtenir la hauteur de `\splitdiscards` dans `\intersplitspace`.

Comme le suggère son nom, la macro `\numlines` a pour vocation première à numéroter les lignes du ⟨*texte*⟩ qui s'étend jusqu'à `\endnumlines`. Pour ce faire, un compteur `\linecnt` est créé et incrémenté à chaque ligne. Il appartient à l'utilisateur d'afficher, s'il le souhaite, la valeur de ce compteur à l'aide de `\leftline` ou `\rightline`.

Code n° V-405

```
\catcode`@11
\newbox\remainbox% boite contenant le texte total
\newbox\currentline% boite contenant le ligne en cours
\newcount\linecnt% compteur pour numéroter les lignes

\def\vdim#1{\dimexpr\ht#1+\dp#1\relax}% hauteur totale de la boite #1

\newmacro\leftline[0pt]{% définit ce qui se trouve à gauche de chaque ligne
  \def\wd@left{#1}%
  \def\stuff@left
}

\newmacro\rightline[0pt]{% définit ce qui se trouve à droite de chaque ligne
  \def\wd@right{#1}%
  \def\stuff@right
}

\let\formatline=\identity% par défaut, afficher chaque ligne telle quelle

% Par défaut :
\leftline[11pt]{$\scriptscriptstyle\number\linecnt$\kern3pt }% numérotation à gauche
\rightline{}% rien à droite

\def\numlines{%
  \par\smallskip
  \begingroup% dans un groupe semi-simple
    \splittopskip=0pt % ne rajouter aucun espace au sommet de la boite restante
    \linecnt=0 % initialiser le compteur de lignes
    \savingvdiscards=1 % autorise la sauvagarde des éléments supprimés
    \setbox\remainbox=\vbox\bgroup% compose la boite...
      \advance\hsize by% diminuer la \hsize
      -\dimexpr\wd@left+\wd@right\relax% de la largeur des contenus
}

\def\endnumlines{%
  \egroup
  \offinterlineskip
  \split@line
}

\def\split@line{%
  \ifvoid\remainbox% si la boite est vide
    \par% fin du processus
    \endgroup% fermer le groupe ouvert au début
  \else% sinon
```

```
    \advance\linecnt 1 % incrémente le compteur de lignes
    \edef\htbefore{\the\vdim\remainbox}%
    \edef\restorevfuzz{\vfuzz=\the\vfuzz\relax}% sauvegarde le \vfuzz
    \vfuzz=\maxdimen% annule les avertissements pour débordement
    \setbox\currentline=\vsplit\remainbox to 0pt % couper la boite à 0pt de hauteur
    \restorevfuzz
    \setbox\currentline=\vbox{\unvbox\currentline}% redonner à la boite sa hauteur
    \edef\intersplitspace{% calcul de l'espace vertical perdu à la coupure
      \the\dimexpr\htbefore-(\vdim\remainbox+\vdim\currentline)\relax
    }%
    \hbox{% en mode vertical et dans une hbox, afficher :
      \hbox to\wd@left{\hss\stuff@left}%   1) ce qui est à gauche
      \formatline{\box\currentline}%       2) la ligne courante
      \hbox to\wd@right{\stuff@right\hss}% 3) ce qui est à droite
    }%
    \splitdiscards% affiche ce qui a été ignoré à la coupure
    \expandafter\split@line% recommencer
  \fi
}

\def\dummytext#1{%
  \for\xx=1to#1\do% composer #1 fois la phrase suivante :
    {Ceci est un texte sans aucun intérêt dont le seul but est de meubler
    la page de façon artificielle. }%
}

\catcode`@12
\parindent=2em

ESSAI 1 :
 \numlines
  \dummytext{2}\par% 2 phrases
  $$1+1=2$$\par% des maths
  \dummytext{1}\par% une phrase
  \hrulefill\par% un leaders
  \dummytext{2}% 2 phrases
  \hrule height2pt depth 2pt %une \hrule
  \vskip10pt % saute 10pt verticalement
  \dummytext{1}% une phrase
\endnumlines\medbreak

ESSAI 2 :
\leftline{}\rightline{}% rien à gauche et rien à droite
\frboxsep=-\frboxrule% encadrer vers "l'intérieur"
\let\formatline\frbox% lignes encadrées
 \numlines
  \dummytext{4}
\endnumlines\medbreak

ESSAI 3 :
\let\formatline\identity
\leftline[7pt]{%
  \for\xx= 1 to 2 \do{% insérer 2 fois
    \setbox0=\hbox{% % mettre dans une \hbox une \vrule de "bonnes dimensions"
      \vrule height\ht\currentline depth\dimexpr\dp\currentline+\intersplitspace
            width0.5pt }%
    \dp0=\dp\currentline% réajuster la profondeur (c-à-d enlever \intersplitspace)
    \box0 % afficher le boite
    \kern2pt % et insérer un espace horizontal de 2pt après chaque réglure verticale
    }%
```

```
  \kern2pt % ajouter 2pt de plus entre les lignes et le texte
}

\rightline[10pt]{\kern5pt $\scriptscriptstyle\number\linecnt$}% numéroter à droite

 \numlines
  \dummytext{2}

  $$a^2+b^2=c^2$$

  \dummytext{2}
\endnumlines
```

ESSAI 1 :

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

$$1 + 1 = 2$$

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

ESSAI 2 :

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

ESSAI 3 :

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

$$a^2 + b^2 = c^2$$

Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle. Ceci est un texte sans aucun intérêt dont le seul but est de meubler la page de façon artificielle.

À l'essai n° 3, une petite cuisine interne est faite pour tracer deux traits verticaux à gauche du texte. Elle consiste à afficher deux fois un trait vertical de bonnes dimensions. Pour ce faire, le registre de boite n° 0 reçoit une `\hbox` contenant une réglure dont la hauteur coïncide exactement avec celle de la ligne en cours et dont la profondeur couvre aussi l'espace additionnel `\intersplitspace` qui sépare deux lignes adjacentes. Cette `\hbox` est ensuite redimensionnée pour qu'elle ait la dimension verticale de la ligne à afficher. En procédant ainsi, les réglures seront jointives d'une ligne à l'autre.

Pour arriver au même résultat, nous aurions pu inclure une `\hrule` de même dimensions que la `\vrule` de l'essai n° 3 dans une `\vtop` dont la dimension verticale est imposée et égale à la hauteur totale de la ligne à afficher. Le débordement vers le bas, d'une valeur de `\intersplitspace` serait rattrapé par un `\vss`. Voici donc un code équivalent qui ne mobilise pas de registre de boite :

```
\leftline[7pt]{%
    \for\xx= 1 to 2 \do{% insérer 2 fois
        \vtop to\dimexpr\ht\currentline+\dp\currentline{%
            \hrule height\ht\currentline
                   depth\dimexpr\dp\currentline+\intersplitspace
                   width0.5pt
            \vss% rattrape la profondeur de trop \intersplitspace
            }%
        \kern2pt % et insérer un espace horizontal
                 % de 2pt après chaque réglure verticale
    }%
    \kern2pt % ajouter 2pt de plus entre les lignes et le texte
}
```

3.3. Étirer horizontalement du texte

Il tout d'abord faut être précis sur les termes. Étirer du texte va signifier ici que des espaces seront insérées entre chaque lettre de façon à ce que le texte ainsi modifié prenne davantage de place que s'il était composé normalement.

Il faut également savoir qu'étirer du texte est un effet typographiquement discutable qui donne parfois des résultats malheureux. Il est donc recommandé de ne pas en abuser et de limiter sa portée à des portions de texte *courtes*, par exemple pour mettre en évidence des titres, cas où son emploi est justifié.

3.3.1. Espace inter-lettre

La méthode qui semble être la plus naturelle est d'insérer un ressort prédéfini que nous appellerons « inter-lettre » après chaque caractère dont le catcode est 11 ou 12 et en remplaçant l'espace par un ressort « inter-mot » dont la dimension fixe sera plus importante que celles des ressorts inter-lettres. La macro `\spreadtxt*{⟨texte⟩}` s'acquittera de cette tâche.

La macro `\parse` écrite à la page 367 permet de lire le texte token par token et d'agir en conséquence via la macro `\testtoken`. Nous nous contenterons donc de modifier localement `\testtoken` pour lui faire insérer les espaces voulues. La macro `\spreadtxt` se chargera de lire son argument, d'ouvrir un groupe semi-simple, de correctement définir `\testtoken` et appellera `\parse` :

Code n° V-406

```
% définition des ressorts "inter-lettre" et "inter-mot"
\newskip\interletterskip \interletterskip=0.25em plus0.05em minus0.05em
\newskip\interwordskip   \interwordskip=3\interletterskip\catcode`\@11
\catcode`@11
\def\spreadtxt@testtoken#1{% macro qui teste le token
  \ifcat\noexpand#1\sptoken% si le token est un espace
    \parseadd{%
      \unskip% retirer le précédent ressort
      \hskip\interwordskip}% et ajouter le ressort inter-mot
  \else
    \ifcat\noexpand#1a% si le token est une lettre
      \parseadd{#1\hskip\interletterskip}% ajouter le ressort inter-lettre
    \else
      \ifcat\noexpand#1.% si le token est "autre", comme le "."
        \parseadd{#1\hskip\interletterskip}% ajouter le ressort inter-lettre
```

```
16      \else% sinon
17        \parseadd{#1}% ajouter le token lu
18      \fi
19    \fi
20  \fi
21  \parse@i
22 }
23 \def\spreadtxt{%
24  \ifstarred% si étoilée
25    {\spreadtxt@i{\parse*}}% appeler \parse*
26    {\spreadtxt@i{\parse}}% sinon, appeler \parse
27 }
28 \def\spreadtxt@i#1#2{% #1= appel "\parse*" ou "\parse"   #2 = texte à espacer
29  \begingroup% dans un groupe
30    \let\testtoken=\spreadtxt@testtoken% modifier \testtoken
31    #1#2\parsestop% et appeler \parse
32  \endgroup
33 }
34 \catcode`@12
35
36 \spreadtxt{Comme on le voit sur cet exemple, une espace est insérée après
37 {\it chaque} caractère de catcode 10, 11 ou 12.}
38 \medbreak
39
40 \spreadtxt*{Comme on le voit sur cet exemple, une espace est insérée après
41 {\it chaque} caractère de catcode 10, 11 ou 12.}
```

Comme on le voit sur cet exemple, une espace est insérée après *chaque* caractère de catcode 10, 11 ou 12.

Comme on le voit sur cet exemple, une espace est insérée après *chaque* caractère de catcode 10, 11 ou 12.

Il y a un petit dysfonctionnement tout de même : les caractères accentués ne sont pas suivis d'une espace. Ceci s'explique par le fait que ce livre a été composé avec LaTeX et l'extension `inputenc` chargée avec l'option `latin1`. La combinaison des deux rend les caractères accentués actifs et donc les tests des lignes nos 6, 11 et 14 sont faux. Le token est donc simplement rajouté au registre de tokens à la ligne no 17 sans le ressort inter-lettre. Il y a plusieurs façons de se sortir de ce mauvais pas. On pourrait imaginer un test supplémentaire où l'on traiterait un caractère actif comme un caractère normal et insèrerait le ressort inter-lettre après lui. Mais cette méthode échouerait avec `UTF8` et un moteur 8 bits, car si l'on rencontre un caractère codé sur plusieurs octets, le premier octet (actif) sera séparé des autres octets participant à la construction du caractère final.

On touche du doigt une difficulté majeure, celle de lire un texte caractère par caractère, où le mot « caractère » est pris au sens d'entité typographique. La difficulté dépend de la combinaison entre l'encodage du code source et le type de moteur utilisé :

cas favorables Les associations entre un moteur et un encodage sur un nombre d'octets correspondant à ce que lit le moteur ne présentent aucune difficulté puisque les caractères typographiques sont les entités lues par le moteur.

- moteur 8 bits (comme `tex`, `etex` et `pdftex`) avec un encodage 8 bits (comme `latin1`) : les caractères lus sont des octets ;

– moteur UTF8 (comme xetex et luatex) avec un encodage UTF8 : les caractères lus sont des caractères UTF8 ;

cas défavorables Les combinaisons entre un moteur 8 bits et un encodage multi octets comme UTF8 va faire surgir de grandes difficultés pour lire le code source caractère UTF8 par caractère UTF8.

3.3.2. Liste de motifs insécables

Comment s'en sortir sans laisser de côté le cas le plus défavorable qui est encore très utilisé [5] ? L'idée est de créer une liste de motifs indivisibles qui seraient laissés tels quels : il faudrait donc tester à chaque itération si les prochains caractères à lire ne commencent pas par un des motifs. Dans l'affirmative, il faudrait laisser ce motif et insérer le ressort inter-lettre après ce motif. Si le test s'avérait négatif, il faudrait agir comme précédemment.

Le problème est que pour déterminer si ce qui reste à lire commence par un motif, il faut lire la totalité du texte à espacer au préalable. La méthode change donc radicalement. Il ne s'agit plus de laisser \parse faire le travail. Nous allons créer une nouvelle macro \spacetxt* qui reprendra en grande partie l'algorithme vu pour la macro \substitute (voir page 379). Le principe consiste à stocker la totalité du texte initial dans la macro \spacetxt@code et le texte espacé dans le registre de tokens \spacetxt@toks. Au fur et à mesure des itérations, \spacetxt@code sera peu à peu amputée des caractères lus et \spacetxt@toks collectera le code produisant le texte espacé.

La liste des motifs indivisibles sera stockée dans la macro \indivilist, chacun étant séparé du suivant par une virgule. Pour tenir compte de ces motifs, il faudra à chaque itération et avant toute chose, tester si le texte de remplacement de \spacetxt@code commence par un des motifs déclarés dans \indivilist. Cette série de tests sera faite à l'aide d'une boucle de type \doforeach. Il est clair que tous ces tests supplémentaires vont être exécutés à chaque itération et cela consommera du temps ; ce n'est donc pas une panacée du point de vue de l'efficacité du programme. Heureusement, comme le texte à espacer est censé être assez court et le nombre de motifs ne doit normalement pas excéder une dizaine d'éléments, la perte de temps n'est pas très importante. Nous utiliserons la macro \doforeachexit pour sortir prématurément de la série de tests si l'un est positif et le booléen \if@indivifound sera pris égal à vrai pour indiquer qu'un motif a été trouvé. Dans ce cas, nous retirerons ce motif à \spacetxt@code pour l'ajouter à \spacetxt@toks en le faisant suivre de l'espace inter-lettre.

La fin du processus se produit lorsqu'il n'y a plus de texte à espacer, c'est-à-dire lorsque \spacetxt@code est vide.

Pour espacer du texte entre accolades, la macro étoilée \spacetxt* devra être appelée. Les mêmes astuces de programmation que pour la macro \substitute ont été employées.

5. L'association de pdfTEX (ou pdfLATEX) avec un encodage UTF8 est en effet très répandu. Cette combinaison ouvre la possibilité d'écrire dans le code source beaucoup plus de caractères que ceux accessibles avec un encodage 8 bits et fonctionne très bien sauf lorsqu'on souhaite élaborer des *programmes* devant lire le code source caractère par caractère.

Code nº V-407

```
\newskip\interletterskip
\newskip\interwordskip
\catcode`\@11
\newtoks\spacetxt@toks%  le registre qui contient le texte final

\def\spacetxt{%
  \let\spacetxt@endprocess\spacetxt@endnormal
  % définit la macro appelée en fin de processus -> a priori : fin normale
  \ifstarred% si la macro est étoilée
    {\let\spacetxt@recurse\spacetxt@star% définir la macro récursive
    \spacetxt@i% et aller à \spacetxt@i
    }% sinon
    {\let\spacetxt@recurse\spacetxt@nostar% définir la macro récursive
    \spacetxt@i% et aller à \spacetxt@i
    }%
}

\newmacro\spacetxt@i[0.3em plus0.07em minus.07em][3\interletterskip]1{%
% arg optionnel #1 et #2 = ressorts inter-lettre et inter--mot
% #3 = texte à espacer
  \interletterskip=#1\relax
  \interwordskip=#2\relax
  \def\spacetxt@code{#3}% met le texte à espacer dans \spacetxt@code
  \spacetxt@toks{}% initialiser le registre contenant le texte final
  \spacetxt@recurse% aller à la macro récursive précédemment définie
}

\newif\if@indivifound% booléen qui sera vrai si un motif spécial est rencontré

\def\rightofsc#1#2{%
  \exparg\ifin{#1}{#2}% si #1 contient le #2
    {\def\right@of##1#2##2\@nil{\def#1{##2}}%
    \expandafter\right@of#1\@nil% appelle la macro auxiliaire
    }%
    {\let#1=\empty}% sinon, #1 est vide
}

\def\spacetxt@nostar{%
  \exparg\ifempty{\spacetxt@code}% si texte restant est vide
    \spacetxt@endprocess% aller à la fin du processus
    {\@indivifoundfalse% sinon, a priori, les motifs non réguliers ne sont pas trouvés
    % pour chaque \indivi@tmp dans \indivilist
    \expsecond{\doforeach\indivi@tmp\in}{\indivilist}% pour chaque motif indivisible
      {% si le code commence par le motif courant
      \exptwoargs\ifstartwith\spacetxt@code\indivi@tmp
        {% l'ajouter dans le registre ainsi que l'espace inter-lettre
        \eaddtotoks\spacetxt@toks{\indivi@tmp\hskip\interletterskip}%
        % et enlever le motif du texte restant à lire
        \expsecond{\rightofsc\spacetxt@code}{\indivi@tmp}%
        \@indivifoundtrue% marquer qu'un motif a été trouvé
        \doforeachexit% et sortir prématurément de la boucle
        }%
        \relax% si le code ne commence pas le motif courant -> ne rien faire
      }%
    \unless\if@indivifound% si aucun motif n'a été trouvé
      \grab@first\spacetxt@code\spacetxt@temp% retirer le 1er caractère du texte
      \ifx\spacetxt@temp\space% si le 1er caractère est un espace
        \addtotoks\spacetxt@toks{%\addtotoks
          \unskip% annuler le précédent ressort
          \hskip\interwordskip}% ajouter l'espace inter-mot au registre de token
```

```
      \else% si le 1er caractère n'est pas un espace
        % ajouter ce caractère et l'espace inter-lettre au registre de token
        \eaddtotoks\spacetxt@toks{\spacetxt@temp\hskip\interletterskip}%
      \fi
    \fi
    \spacetxt@recurse% enfin, continuer le processus
    }%
}

\def\spacetxt@star{%
  \exparg\ifempty{\spacetxt@code}% si texte restant est vide
    \spacetxt@endprocess% aller à la fin du processus
    {% sinon, si texte commence par "{"
    \exparg\ifbracefirst{\spacetxt@code}%
      {\grab@first\spacetxt@code\spacetxt@temp
      % mettre {<argument} dans \spacetxt@temp
      \begingroup% ouvrir un groupe
      % mettre le contenu de l'argument dans \spacetxt@code
      \expandafter\def\expandafter\spacetxt@code\spacetxt@temp
      \let\spacetxt@endprocess\spacetxt@endingroup% changer le processus de fin
      \spacetxt@toks{}% initialiser
      \spacetxt@recurse% exécuter le processus avec ce nouveau texte
      }% si le code ne commence pas par "{", aller à \spacetxt@nostar mais comme
      \spacetxt@nostar% \spacetxt@recurse vaut \spacetxt@star, n'y faire qu'1 boucle
    }%
}

\def\spacetxt@endnormal{% fin de processus normal
  \the\spacetxt@toks% afficher le registre à token
  \unskip% et supprimer le dernier ressort
}

\def\spacetxt@endingroup{% fin du processus dans un groupe :
  \expandafter\endgroup\expandafter% avant de fermer le groupe
  \addtotoks\expandafter\spacetxt@toks\expandafter% ajouter au registre hors du groupe
    {\expandafter{\the\spacetxt@toks}}% ce qui est collecté localement mis entre {}
  \spacetxt@recurse% puis aller à la macro récursive
}

\catcode`\@12
\def\indivilist{é,è}% liste des motifs spéciaux
\spacetxt*{Comme on le voit sur cet exemple, une espace est insérée après
{\it chaque} caractère.}
\medbreak

\def\indivilist{é,es,au,<<,>>}
\spacetxt[4pt plus.7pt minus.7pt][12pt plus2pt minus2pt]{Ici, les motifs <<es>> et <<au>>
restent indivisibles et la ligature des guillemets devient possible en déclarant "<<" et
">>" comme motifs.}
```

Comme on le voit sur cet exemple, une espace est insérée après *chaque* caractère.

Ici, les motifs « es » et « au » restent indivisibles et la ligature des guillemets devient possible en déclarant "«" et "»" comme motifs.

Nous avons utilisé les arguments optionnels de la macro `\spacetxt@i` pour définir à l'intérieur de celle-ci les ressorts inter-lettre et inter-mot. Cette méthode est préférable à celle de `\spreadtxt` où l'on avait défini ces registres de ressort

en dehors de toute macro. En effet, si l'on définit ces ressorts avec des unités qui varient selon le contexte (comme l'ex ou l'em), les dimensions des ressorts ainsi définis ne dépendront que de la valeur de ces unités *lors de l'assignation.*

Il est important de comprendre que l'assignation d'une dimension à un registre de ressort (ou de dimension) *évalue* cette dimension pour la stocker dans le registre. La différence est fondamentale entre

⟨*ressort*⟩=0.3em

et

\def\⟨*macro*⟩{0.3em}

suivi plus tard de

⟨*ressort*⟩=\⟨*macro*⟩

En effet, lorsqu'on écrit « ⟨*ressort*⟩=0.3em », la dimension 3em est évaluée et le résultat est stocké dans le registre qui dès lors, contient une dimension qui dépend de la valeur de l'em lors de l'assignation.

Au contraire, écrire « \def\⟨*macro*⟩{0.3em} » stocke simplement dans le texte de remplacement de \⟨*macro*⟩ les caractères « 0.3em ». C'est lors de l'assignation de ressort « ⟨*ressort*⟩=\⟨*macro*⟩ » que l'évaluation de 0.3em est faite. Ce passage par une macro auxiliaire de stockage permet donc de garder le caractère variable de l'unité em et choisir le moment où l'on évalue cette dimension.

3.3.3. Une alternative à \litterate

La méthode précédente peut être adaptée pour insérer un court passage en fonte à chasse fixe à l'intérieur d'un paragraphe, pour écrire une adresse URL, par exemple. Nous allons écrire une macro \ttcode⋆, de syntaxe

\ttcode⟨*délimiteur*⟩⟨*texte*⟩⟨*délimiteur*⟩

qui compose le ⟨*texte*⟩ en police à chasse fixe où les tokens de catcode différents de 10, 11 et 12 sont neutralisés avec \dospecials. En invoquant \spacetxt, nous allons insérer un ressort de faible dimension après chaque token du ⟨*texte*⟩ et donc, contrairement à \litterate, \ttcode permet qu'une coupure se produise après chaque token du ⟨*texte*⟩.

Code nº V-408

```
\catcode`\@11
\def\ttcode#1{% lit #1, le <délimiteur> de début
  \def\ttcode@i##1#1{% ##1 = <texte> entre délimiteurs
    \tt% passe en fonte à chasse fixe
    \setbox0=\hbox{ }%
    \edef\spc@wd{\the\wd0 }% longueur d'un espace
    \spacetxt
      [.1pt plus0pt minus.1pt]% espace inter-lettre
      [\glueexpr\wd0+.3pt plus.1pt minus.1pt\relax]% espace inter-mot
      {##1}% le <texte> est composé par \spacetxt
    \endgroup
  }%
  \begingroup
    \def\indivilist{<<,>>,{,,},--}% (rajouter à, é, è, etc. en codage UTF8)
    \def\do##1{\catcode`##1=12 }%
    \dospecials% rend inoffensifs tous les tokens spéciaux
```

```
    \letactive\ =\space % rend l'espace actif
    \ttcode@i% va lire le <texte> et le délimiteur de fin
}
\catcode`\@12

\hfill\vrule\vbox{%
  \hsize=.75\hsize
  Avec \ttcode/\ttcode/, on peut insérer une adresse internet <<
  \ttcode-http://www.gutenberg.eu.org/Typographie- >> puis repasser en fonte normale
  puis \ttcode+même composer un court passage en fonte à chasse fixe -- même si les
  coupures de mots se font n'importe où -- et aussi afficher tous les caractères
  spéciaux <<{$^ _$#}&+>>, et finir en fonte normale\ldots
}\vrule\hfill\null
```

Avec \ttcode, on peut insérer une adresse internet « http://www.gutenberg.eu.org/Typographie » puis repasser en fonte normale puis même composer un court passage en fonte à chasse fixe – même si les coupures de mots se font n'importe où – et aussi afficher tous les caractères spéciaux «{$^ _$#}&», et finir en fonte normale…

3.4. Composition en fonte à chasse fixe

3.4.1. Mise en évidence du problème

Une fonte[6] est dite à « chasse fixe » lorsque tous ses caractères ont la même largeur, y compris l'espace qui est non étirable et non comprimable. La composition en fonte de ce type peut poser des problèmes, car à cause de cette particularité géométrique, il devient impossible de respecter *en même temps* les trois contraintes suivantes :

1. caractères parfaitement les uns au-dessous des autres d'une ligne à l'autre ;
2. composition justifiée, c'est-à-dire que la distance entre le bord gauche du premier caractère et le bord droit du dernier est toujours la même ;
3. coupures des mots respectées ou interdites.

Il faut savoir que le passage en fonte à chasse fixe avec LaTeX à l'aide des macros `\texttt` ou `\ttfamily` désactive la coupure des mots (nous verrons comment). Dès lors, il suffit qu'un mot assez long se trouve en fin de ligne pour que le dépassement dans la marge soit tellement important que parfois, les caractères se trouvent en dehors de la page physique. Les utilisateurs de LaTeX savent certainement de quoi je parle…

À titre d'exemple, voici un court paragraphe composé en fonte à chasse fixe (avec la macro `\ttfamily`) comme le verrait un utilisateur de LaTeX (alors que pour un utilisateur de TeX avec la macro `\tt`, les coupures de mots seraient effectuées) :

6. On peut souvent confondre police et fonte, mais une différence sémantique existe. Une *police* est un ensemble de glyphes qu'un auteur (bien souvent, c'est le même auteur qui dessine une police entière) a dessiné dans un même esprit en respectant le même style. Une *fonte* est un sous-ensemble d'une police ayant des caractéristiques bien déterminées : taille, graisse (léger, normal, demi-gras, gras, etc.), forme (italique, penché, petite capitale, etc.)

Par exemple, le texte de ce livre est écrit avec la *police* Libertine, mais **ceci** et *ceci* sont écrits en deux *fontes* différentes de cette police.

Code n° V-409

```
\def\ttwide{0.7}% coefficient pour la largeur de composition
\def\ttindent{5}% nombre de caractères d'indentation
\hfill\vrule
\vbox{%
  \ttfamily% en TeX, on écrirait "\tt"
  \setbox0\hbox{0}% \wd0 est donc la largeur d'un caractère
  \parindent=\ttindent\wd0 % réglage de l'indentation
  \hsize=\ttwide\hsize % compose sur 70% de la largeur
  Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauches
  et droites du texte ont été tracées, on constate que les caractères sont
  exactement les uns au-dessous d'une ligne à l'autre. Des débordements dans
  la marge deviennent alors inévitables, car la largeur de composition (ici
  \the\hsize) n'est pas un multiple de la largeur d'un caractère (\the\wd0 ),
  le quotient des deux valant environ
  \xdef\ttratio{\decdiv{\dimtodec\hsize}{\dimtodec\wd0 }}\ttratio.
}%
\vrule\hfill\null
```

```
    Dans ce paragraphe, composé en fonte à chasse fixe et où
les limites gauches et droites du texte ont été tracées, on constate
que les caractères sont exactement les uns au-dessous d'une ligne
à l'autre. Des débordements dans la marge deviennent alors inévitables,
car la largeur de composition (ici 229.26292pt) n'est pas un
multiple de la largeur d'un caractère (3.84064pt), le quotient
des deux valant environ 59.69398.
```

Bien sûr, le fichier `log` contient 6 avertissements, chacun informant l'utilisateur que TEX a dû composer une ligne trop longue :

```
Overfull \hbox (4.97778pt too wide) in paragraph at lines 9--16
Overfull \hbox (31.85786pt too wide) in paragraph at lines 9--16
Overfull \hbox (20.33783pt too wide) in paragraph at lines 9--16
Overfull \hbox (43.3779pt too wide) in paragraph at lines 9--16
Overfull \hbox (1.13777pt too wide) in paragraph at lines 9--16
Overfull \hbox (8.8178pt too wide) in paragraph at lines 9--16
```

Ici en effet, sur chaque ligne, il entre 59 caractères complets. Après ce 59e caractère, comme il reste de la place, TEX place le caractère suivant entrant dès lors dans un dépassement dans la marge. Les coupures de mots étant interdites, il continuera à placer des caractères dans la marge jusqu'au prochain espace qui provoquera, mais trop tard, une coupure de ligne.

Livrons-nous à un petit calcul pour comprendre la valeur `8.8178pt` donnée pour le dépassement de la sixième ligne. Cette ligne comporte 62 caractères : les 59 complets plus les 3 dernières lettres du mot « `quotient` ». Le dépassement dans la marge vaut

$$62 \times 3.84001 - 229.26292$$

c'est-à-dire `8.8177pt`. C'est bien, à une erreur d'arrondi près, la valeur qui est donnée par TEX dans le message d'avertissement du fichier `log`.

3.4.2. Quelques notions sur les fontes

Avant d'envisager toute solution concernant les fontes de caractères, il nous faut entrer un peu plus profondément dans les entrailles de TEX afin d'examiner

quelques commandes spécifiques. Nous nous en tiendrons au strict minimum, car la manipulation des fontes en TEX est un monde long et complexe à explorer.

Les primitives \font et \fontname

93 - Règle

La primitive \fontname permet d'accéder au nom externe d'un fichier de fonte selon la syntaxe

\fontname⟨*fonte*⟩

où ⟨*fonte*⟩ est soit une séquence de contrôle définie avec la primitive \font, soit la primitive \font elle-même auquel cas on fait référence à la fonte en cours d'utilisation. Le tout se développe en le nom du fichier externe correspondant à la fonte spécifiée.

Ainsi, ce paragraphe est écrit avec une des fontes de la police « Libertine » dont le nom du fichier externe est « LinLibertineT-tlf-t1 ». Lorsqu'on passe en fonte grasse, le nom devient « LinLibertineTB-tlf-t1 » tandis qu'avec la fonte italique, le nom est « LinLibertineTI-tlf-t1 ».

Lorsqu'une fonte est utilisée dans une autre taille que celle pour laquelle elle a été dessinée, le mot-clé « at » suivi de la taille d'utilisation est ajouté après le nom obtenu :

Code nº V-410

```
Fonte normale : \fontname\font\par
{\bf Fonte grasse : \fontname\font\par}
{\it Fonte italique : \fontname\font\par}
{\sc Fonte petites majuscules : \fontname\font}
```

Fonte normale : LinLibertineT-tlf-t1 at 8.0pt
Fonte grasse : LinLibertineTB-tlf-t1 at 8.0pt
Fonte italique : LinLibertineTI-tlf-t1 at 8.0pt
Fonte petites majuscules : LinLibertineT-tlf-sc-t1 at 8.0pt

Le « at 8.0pt » exprime que les rendus des codes de ce livre sont composés avec une taille de 8pt imposée, ce qui revient à dire que les dessins des caractères ainsi que toutes leurs dimensions géométriques ont été redimensionnés pour atteindre cette taille.

94 - Règle

La primitive \font permet de créer une séquence de contrôle qui, lorsqu'elle sera exécutée, effectuera un changement de fonte. On utilise la syntaxe

\font\⟨*macro*⟩=⟨*nom de fonte*⟩ at ⟨*dimension*⟩

où « at ⟨*dimension*⟩ » est facultatif.

Voici comment créer des macros (\gras, \ital et \itgras) qui, lorsqu'elles sont appelées, sélectionnent respectivement la fonte de taille 8pt, en gras, italique et gras italique :

Code n° V-411

```
\font\gras=LinLibertineTB-tlf-t1 at 8pt
\font\ital= LinLibertineTI-tlf-t1 at 8pt
\font\itgras=LinLibertineTBI-tlf-t1 at 8pt
Du texte normal {\gras puis en gras, \ital en italique,
\itgras en italique gras} et retour à la normale.
```

Du texte normal **puis en gras,** *en italique,* ***en italique gras*** et retour à la normale.

Caractère de coupure de mots

Chaque fonte se voit assigner un caractère de coupure qui sera inséré à la fin d'une ligne lorsqu'un mot est coupé. Par défaut, ce caractère est le tiret « – ».

95 - Règle

Le caractère inséré aux coupures de mots est stocké dans un registre interne de type entier appelé \hyphenchar. Ce registre *doit* être suivi de la fonte à laquelle on souhaite se référer :

\hyphenchar⟨*fonte*⟩

On peut à tout moment choisir un autre caractère de coupure par son code de caractère en modifiant \hyphenchar :

\hyphenchar⟨*fonte*⟩=⟨*code de caractère*⟩

Si le ⟨*code de caractère*⟩ contenu dans \hyphenchar est négatif ou est supérieur à 255, aucun caractère n'est sélectionné et cela désactive les coupures de mots.

L'assignation d'un entier à \hyphenchar est toujours globale.

Code n° V-412

```
Code du caractère de coupure = \number\hyphenchar\font\par
Caractère de coupure : "\char\hyphenchar\font"
```

Code du caractère de coupure = 45
Caractère de coupure : "-"

Voici comment modifier le \hyphenchar :

Code n° V-413

```
\def\longtext{Voici une phrase écrite avec des mots insignifiants mais terriblement,
  épouvantablement, horriblement et indéniablement longs.}
% créér une macro restaurant le \hyphenchar
\edef\restorehyphenchar{\hyphenchar\font=\number\hyphenchar\font}%

%comportement normal, caractère de coupure "-"
1) \vrule\vbox{\hsize=5cm \longtext}\vrule\hfill
% modification du caractère de coupure "W"
2) \vrule\vbox{\hsize=5cm \hyphenchar\font=`W \longtext}\vrule
\medbreak
% interdiction des coupures de mots
3) \vrule\vbox{\hsize=5cm \hyphenchar\font=-1 \longtext}\vrule
\restorehyphenchar
```

1) Voici une phrase écrite avec des mots insignifiants mais terriblement, épouvantablement, horriblement et indéniablement longs.

2) Voici une phrase écrite avec des mots insiW gnifiants mais terriblement, épouvantableW ment, horriblement et indéniablement longs.

3) Voici une phrase écrite avec des mots insignifiants mais terriblement, épouvantablement, horriblement et indéniablement longs.

Lorsque la largeur de la composition est faible, interdire les coupures peut conduire TEX à composer des lignes trop ou pas assez remplies. La macro de LATEX `\sloppy` relâche certains paramètres de composition de TEX de telle sorte que les débordements dans les marges sont presque toujours évités, au prix d'espaces inter-mots parfois très (trop !) larges. C'est pourquoi elle ne doit être utilisée qu'en ultime recours.

Code n° V-414

```
\def\longtext{Voici une phrase écrite avec des mots insignifiants mais terriblement,
  épouvantablement, horriblement et indéniablement longs.}

\edef\restorehyphenchar{\hyphenchar\font=\number\hyphenchar\font}%
\vrule\vbox{\hsize=5cm \hyphenchar\font=-1 \sloppy \longtext}\vrule
\restorehyphenchar
```

Voici une phrase écrite avec des mots insignifiants mais terriblement, épouvantablement, horriblement et indéniablement longs.

Les utilisateurs de LATEX, à qui s'adresse le code ci-dessous, doivent donc savoir que lorsqu'ils passent en fonte à chasse fixe (avec la macro `\ttfamily` ou `\texttt`), le code du caractère de coupure vaut −1. Les coupures de mots sont dès lors interdites.

Code n° V-415

```
a) \texttt{\number\hyphenchar\font}\qquad
b) {\ttfamily\number\hyphenchar\font}
```

a) –1 b) –1

Dimensions de fonte

96 - Règle

Pour composer du texte, chaque fonte dispose de sept dimensions propres contenues dans des registres de dimension spéciaux accessibles via la primitive `\fontdimen`. Chacun de ces registres est accessible par

`\fontdimen`⟨*entier*⟩⟨*fonte*⟩

où ⟨*entier*⟩ est le numéro de la dimension (compris entre 1 et 7) et dont la signification est la suivante :

- n° 1 pente par point (dimension utilisée pour placer les accents) ;
- n° 2 espace inter-mot (dimension naturelle de l'espace) ;
- n° 3 étirement inter-mot ;

nº 4 compression inter-mot ;
nº 5 x-height (valeur de 1ex) ;
nº 6 cadrat (valeur de 1em) ;
nº 7 espace supplémentaire (espace ajouté en fin des phrases).

Les plus intéressants ici sont les nos 2, 3 et 4 qui permettent de régler la façon dont un espace se comporte lors de la composition. Pour les fontes à chasse fixe, les dimensions nos 3 et 4 sont nulles et la dimension nº 2 est égale à largeur de tous les autres caractères.

Code nº V-416

```
\leavevmode
\vbox{%
  \hsize=.4\hsize
  \Souligne{Police à chasse variable}%
  \vskip5pt
  espace inter-mot = \the\fontdimen2\font\par
  étirement inter-mot = \the\fontdimen3\font\par
  compression inter-mot = \the\fontdimen4\font
}\hfill
\vbox{%
  \tt
  \hsize=.4\hsize
  \Souligne{Police à chasse fixe}%
  \vskip5pt
  espace inter-mot = \the\fontdimen2\font\par
  étirement inter-mot = \the\fontdimen3\font\par
  compression inter-mot = \the\fontdimen4\font
}
```

Police à chasse variable

espace inter-mot = 2.0pt
étirement inter-mot = 1.0pt
compression inter-mot = 0.6664pt

Police à chasse fixe

espace inter-mot = 3.84064pt
étirement inter-mot = 0.0pt
compression inter-mot = 0.0pt

Coupures de mots

Si l'on travaille avec LaTeX, la première idée est de réactiver les coupures de mots pour composer un texte en fonte à chasse fixe.

Code nº V-417

```
\def\ttwide{0.7}\def\ttindent{5}%
\hfill\vrule
\vbox{%
  \ttfamily
  \hyphenchar\font='\- % change le caractère de coupure de la fonte en cours
  \setbox0\hbox{0}\parindent=\ttindent\wd0
  \hsize=\ttwide\hsize % compose sur 70% de la largeur
  Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
  et droite du texte ont été tracées, on constate que les caractères sont
  exactement les uns au-dessous des autres d'une ligne à l'autre. Des débordements
  dans la marge deviennent inévitables même si les coupures des mots sont
  à nouveau rendues possibles.%
}%
\vrule\hfill\null
```

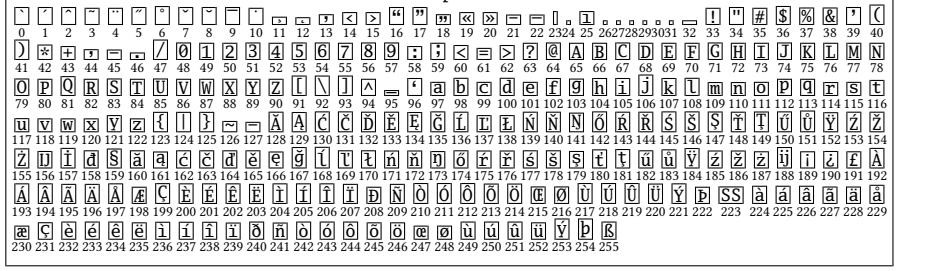
```
    Dans ce paragraphe, composé en fonte à chasse fixe et où
les limites gauche et droite du texte ont été tracées, on con-
state que les caractères sont exactement les uns au-dessous des
autres d'une ligne à l'autre. Des débordements dans la marge
deviennent inévitables même si les coupures des mots sont à nou-
veau rendues possibles.
```

Peut-on effectuer les coupures, mais ne pas insérer de caractère de coupure ? La réponse est non, mais il est parfois possible, selon la fonte utilisée, de trouver un caractère de dimension horizontale nulle et ne se traduisant par aucun dessin. On peut le chercher en construisant une macro `\printallchars` moyennant une boucle `\for` :

Code nº V-418

```
\frboxsep=0.5pt
\def\printallchars{%
  \leavevmode
  \for\xx=0 to255\do{%
    \vbox{% empiler verticalement
      \offinterlineskip% en désactivant le ressort d'interligne
      \setbox0\hbox{\frbox{\char\xx}}%
      \copy0 % la boite contenant le caractère encadré
      \kern1pt% saute 1pt vertical
      \hbox to\wd0{\hss$\scriptscriptstyle\xx$\hss}% le numéro
    }\hskip0.5em plus1pt minus1pt % saute 0.5em horizontalement
  }%
}
\tt
Nom de la fonte = \fontname\font\par
\printallchars
```

Nom de la fonte = GoMono-tlf-t1 at 6.40002pt



Par chance, il semblerait que dans la fonte à chasse fixe utilisée dans ce livre (qui est « GoMono »), le caractère nº 23 de cette fonte n'ait aucune dimension horizontale. Vérifions-le :

Code nº V-419

```
\setbox0=\hbox{\tt\char23}
Largeur = \the\wd0 \qquad Hauteur = \the\ht0 \qquad Profondeur = \the\dp0
```

Largeur = 0.0pt Hauteur = 3.46721pt Profondeur = 0.0pt

Il est utile de signaler que ε-TEX offre des primitives bien pratiques pour mesurer les dimensions d'un caractère dans une fonte donnée : `\fontcharht`, `\fontcharwd`, `\fontchardp` et `\fontcharic`, suivies d'une spécification de fonte et d'un code de caractères donnent des dimensions égales à la hauteur, largeur, profondeur et correc-

tion d'italique du caractère concerné. La dimension `0pt` est renvoyée si le caractère n'existe pas. Le code précédent peut donc s'écrire :

Code n° V-420

```
{\tt\xdef\nfont{\the\font}}
Largeur = \the\fontcharwd\nfont23 \qquad Hauteur = \the\fontcharht\nfont23
\qquad Profondeur = \the\fontchardp\nfont23
```

Largeur = 0.0pt Hauteur = 3.46721pt Profondeur = 0.0pt

Concernant ce caractère n° 23, il importe peu que la hauteur ne soit pas nulle puisque seule la largeur nulle revêt une importance ici. Elle signifie que si l'on choisit ce caractère comme caractère de coupure avec

```
\hyphenchar\font=23
```

tout se passera comme si aucun caractère de coupure n'était sélectionné. Cette manœuvre suppose bien entendu que la fonte s'y prête et dispose d'un tel caractère.

Nous avons progressé, mais pas résolu le problème que pose la composition sur plusieurs lignes de texte en fonte à chasse fixe : les débordements sont toujours bien là, même s'ils sont moins importants.

3.4.3. Justifier et couper

Nous allons maintenant supprimer la première contrainte vue à la page 467 et nos contraintes seront donc :

2. composition justifiée, c'est-à-dire que la distance entre le bord gauche du premier caractère et le bord droit du dernier est toujours la même ;
3. coupures des mots respectées ou carrément interdites.

Respecter ces contraintes revient à « casser » l'alignement vertical des caractères entre les lignes et en tirer parti pour supprimer – ou au moins réduire au minimum – les débordements dans la marge. La méthode consiste à donner à l'espace le caractère étirable qu'il a dans les autres fontes, c'est-à-dire le rendre compressible et extensible dans des limites raisonnables à fixer, bien entendu. Pour cela, il faut rendre non nuls les `\fontdimen` n^os^ 3 et 4 de la fonte à chasse fixe. Ici, nous allons donner une composante extensible égale à 30% de l'espace inter-mot et une composante compressible égale à 20% de cet espace. Un « environnement » sera créé pour l'occasion et s'étendra entre la macro `\ttstart*` et `\ttstop`.

Nous devons prendre nos précautions pour que les modifications des paramètres de la fonte ne restent pas globales [7]. Pour cela, la macro `\restaurefontsettings` sera chargée de sauvegarder les paramètres de fonte avant les modifications. Elle sera appelée par `\ttstop`, juste avant la fermeture du groupe.

Code n° V-421

```
\newmacro\ttstart[5]{%
  \begingroup
  \tt
  \edef\restorefontsettings{% stocke les paramètres de fonte
    \hyphenchar\font=\the\hyphenchar\font\relax% le \hyphenchar
    \fontdimen2\font=\the\fontdimen2\font\relax% et les paramètres d'espacement
```

7. Les assignations `\fontdimen` et`\hyphenchar` sont toujours globales, lire le TEXbook page 322.

```
    \fontdimen3\font=\the\fontdimen3\font\relax
    \fontdimen4\font=\the\fontdimen4\font\relax
  }%
  \fontdimen3\font=0.30\fontdimen2\font% composante + = 30% de la dimension naturelle
  \fontdimen4\font=0.20\fontdimen2\font% composante - = 20% de la dimension naturelle
  \hyphenchar\font=`\- % on autorise la coupure des mots (au cas où on utilise latex)
  \setbox0\hbox{0}% largeur d'un caractère
  \parindent=#1\wd0 % indentation (en nombre de caractères)
  \ignorespaces
}
\def\ttstop{%
  \restorefontsettings% restaure les paramètres de fonte
  \endgroup% et ferme le groupe
}
\hfill\vrule
\def\ttwide{0.70}%
\vbox{%
  \hsize=\ttwide\hsize % compose sur 70% de la largeur
  \ttstart[5]
  Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
  et droite du texte ont été tracées, on constate que les caractères ne sont pas
  exactement les uns au-dessous des autres entre les lignes du paragraphe.
  Puisque les espaces sont redevenus étirables, les débordements dans la marge
  (même s'ils restent possibles), sont bien plus rares et ce d'autant plus que
  le nombre d'espaces dans une ligne est grand.%
  \ttstop
}%
\vrule\hfill\null
```

```
    Dans ce paragraphe, composé en fonte à chasse fixe et
où les limites gauche et droite du texte ont été tracées, on
constate que les caractères ne sont pas exactement les uns
au-dessous des autres entre les lignes du paragraphe. Puisque
les espaces sont redevenus étirables, les débordements dans
la marge (même s'ils restent possibles), sont bien plus rares
et ce d'autant plus que le nombre d'espaces dans une ligne
est grand.
```

Le but est atteint, aucun débordement dans la marge n'a lieu, mais comme attendu, on perd l'alignement vertical des lettres entre les lignes. Remarquons que les valeurs 30% et 20% devront être augmentées si le nombre de caractères par ligne diminue et réduites dans le cas contraire.

Une autre alternative aux `\fontdimen` existe pour modifier l'espace inter-mot. Il s'agit du ressort-primitive `\spaceskip` qui, s'il n'est pas nul, écrase les spécifications contenues dans `\fontdimen` n^{os} 2, 3 et 4. L'avantage est que sa modification tient compte des groupes et y reste locale, contrairement aux modifications de `\fontdimen`. Ainsi, les lignes

```
\fontdimen3\font=0.30\fontdimen2\font
\fontdimen4\font=0.20\fontdimen2\font
```

peuvent être remplacées par celles-ci

```
\spaceskip=\fontdimen2\font
    plus0.3\fontdimen2\font
    minus0.2\fontdimen2\font
```

3.4.4. Justifier et aligner

Le but ici est de supprimer la 3e contrainte de la page 467 ce qui revient à se fixer les contraintes suivantes :

1. caractères parfaitement les uns au-dessous des autres d'une ligne à l'autre ;
2. composition justifiée, c'est-à-dire que la distance entre le bord gauche du premier caractère et le bord droit du dernier est toujours la même.

Ces deux contraintes impliquent que ces compromis doivent être concédés :

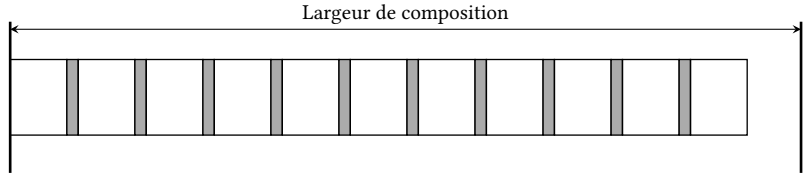
- une espace inter-lettre de dimension invariable doit être insérée entre tous les caractères d'une même ligne ;
- les coupures peuvent se faire n'importe où, même entre deux caractères où elles n'auraient pas eu lieu si elles avaient été faites par l'algorithme de coupure [8] de TEX.

Puisqu'une espace inter-lettre est insérée entre chaque caractère, nous allons prendre le parti de laisser à l'utilisateur la possibilité de choisir cette espace. Pour ce faire, nous allons écrire une macro `\breaktt*` qui va insérer un *faible* ressort fixe entre chaque lettre (rappelons-nous qu'un ressort est susceptible d'être un point de coupure). La syntaxe de cette macro sera :

`\breaktt[`⟨*dimension inter-lettre*⟩`][`⟨*largeur de composition*⟩`]{`⟨*texte*⟩`}`

Les deux arguments optionnels permettent de spécifier la dimension de l'espace inter-lettre et la largeur de composition. La dimension inter-lettre sera insérée *après* chaque caractère, sauf le dernier de chaque ligne.

Si l'on veut que la largeur du texte soit égale à la largeur de composition passée en argument, la dimension inter-lettre doit être choisie avec soin. Elle dépend mathématiquement de la largeur des caractères et de la largeur de composition. Si l'utilisateur demande une dimension inter-lettre sans précaution, la largeur d'une ligne sera inférieure à la largeur de composition. Voici le schéma représentant la situation d'une ligne :



Les boites englobantes des caractères sont représentées par des rectangles vides et les espaces inter-lettres demandées par l'utilisateur par des zones grisées.

Pour éviter que l'utilisateur ne calcule lui-même la valeur du premier argument, il devient évident qu'il doit être automatiquement ajusté de telle sorte que l'espace disponible à droite du dernier caractère soit également réparti dans chaque zone grisée. Par conséquent, la ⟨*dimension inter-lettre*⟩ donnée en argument n° 1 ne sera pas *exactement* celle insérée entre chaque caractère, mais devra être légèrement augmentée.

Cherchons maintenant à formaliser le tout avec les notations suivantes :

- L est la largeur de composition (argument optionnel #2) ;
- l est la largeur d'un caractère ;

8. Cet algorithme est décrit à l'annexe H du TEXbook.

– Δ la dimension inter-lettre demandée (argument optionnel #1).

Compte tenu du fait que le dernier caractère de la ligne n'est pas suivi d'une espace inter-lettre, nous allons dans un premier temps ignorer ce caractère et soustraire l à la largeur de composition L pour obtenir la place disponible pour les paires ⟨*caractère*⟩ + ⟨*dimension inter-lettre*⟩. Pour calculer le nombre de ces paires (de largeur $l + \Delta$), il faut prendre la partie entière du quotient

$$\frac{L - l}{l + \Delta}$$

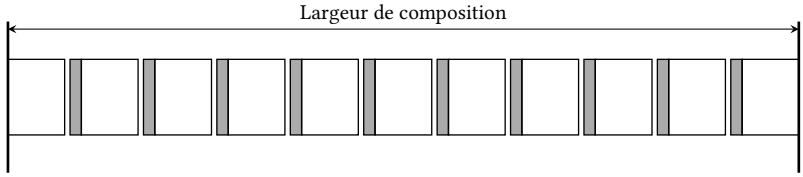
En ajoutant le dernier caractère de la ligne que nous avions ignoré, nous obtenons le nombre maximal de caractères n qu'il est possible de loger sur une ligne. Cela donne (E représente l'opérateur « partie entière ») :

$$n = \mathrm{E}\left(\frac{L - l}{l + \Delta}\right) + 1$$

Il est immédiat que le nombre de dimensions inter-lettre est égal à $n - 1$ et donc, pour justifier le texte, il faut répartir également l'espace non occupé par les caractères entre chaque caractère. Par conséquent, il faut insérer après chaque caractère, non pas l'espace inter-lettre Δ, mais

$$\Delta' = \frac{L - nl}{n - 1}$$

Voici ce que devient le schéma précédent :



L'espace grisé Δ doit être augmenté d'une petite quantité pour que la justification se fasse. Comme l'argument optionnel #1 n'est *pas* exactement l'espace qui sera inséré entre chaque caractère, il doit être compris comme une dimension *minimale* susceptible d'être un peu allongée pour justifier le texte à la dimension spécifiée.

Entrons dans le vif du sujet et définissons les variables principales dont nous avons besoin :

– un compteur `\brktt@cnt` qui va compter, pour chaque ligne, combien de caractères ont déjà été placés ;
– une macro `\maxchar@num` qui contiendra n, le nombre de caractères par ligne ;
– un registre de dimension inter-lettre `\brktt@interletter` pour recevoir Δ' ;
– une macro `\tt@remaintext` dont le texte de remplacement contient ce qu'il reste du texte à composer ;
– une liste de ligatures `\liglist` contenant des suites de caractères susceptibles de former un caractère *unique*, soit par ligature (par exemple « << », « -- », etc.) soit parce que ces caractères sont codés sur plusieurs octets en UTF8 avec un moteur 8 bits.

Le processus va se décomposer en deux. Tout d'abord, la macro \breaktt va ouvrir un groupe semi-simple qui sera fermé à la fin du processus et passera en mode vertical. Elle va également procéder aux initialisations et aux calculs préalables (nombre de caractères par ligne n, espace inter-lettre Δ') ainsi qu'au passage en fonte à chasse fixe. Cette macro chapeau appellera la macro récursive \breaktt@i qui imprimera, en mode vertical et dans une \hbox une ligne entière. Ensuite, si le texte restant est vide, cela signe la fin du processus. Dans le cas contraire, la \hbox doit être fermée et une nouvelle doit être ouverte pour accueillir la ligne suivante qui sera imprimée par l'appel récursif \breaktt@i.

Le mécanisme de fermeture/ouverture de \hbox sera effectué par la macro \restart@hbox dont l'argument est le texte restant à composer. Après avoir fermé la \hbox précédente avec \egroup et avoir ouvert une nouvelle \hbox avec \bgroup, cette macro initialise le compteur de caractères \brktt@cnt à 0 et, à l'aide de \removefirstspaces, purge son argument des éventuels espaces qui se trouvent en première position. Le résultat de cette opération est stocké dans \tt@remaintext : ceci évite que la prochaine ligne ne commence par une espace.

Le nœud du processus sera une macro auxiliaire \print@nchar{⟨n⟩} qui est chargée d'afficher ⟨n⟩ caractères pris dans \tt@remaintext. Une boucle de type \for sera mise en œuvre à cet effet. Au cours de la boucle :

- si \tt@remaintext est vide, une sortie prématurée de boucle doit être faite et la macro doit rendre la main ;
- si \tt@remaintext commence par un des motifs figurant dans la liste des ligatures, le caractère courant à afficher sera pris égal à ce motif et dans le cas contraire au premier caractère de \tt@remaintext ;
- l'espace inter-lettre ne doit pas être inséré après le dernier caractère de la ligne. Autrement dit, il ne faut insérer cette espace que si \brktt@cnt < \maxchar@num ;
- si \brktt@cnt > \maxchar@num, une sortie prématurée de boucle doit être faite et la macro doit rendre la main.

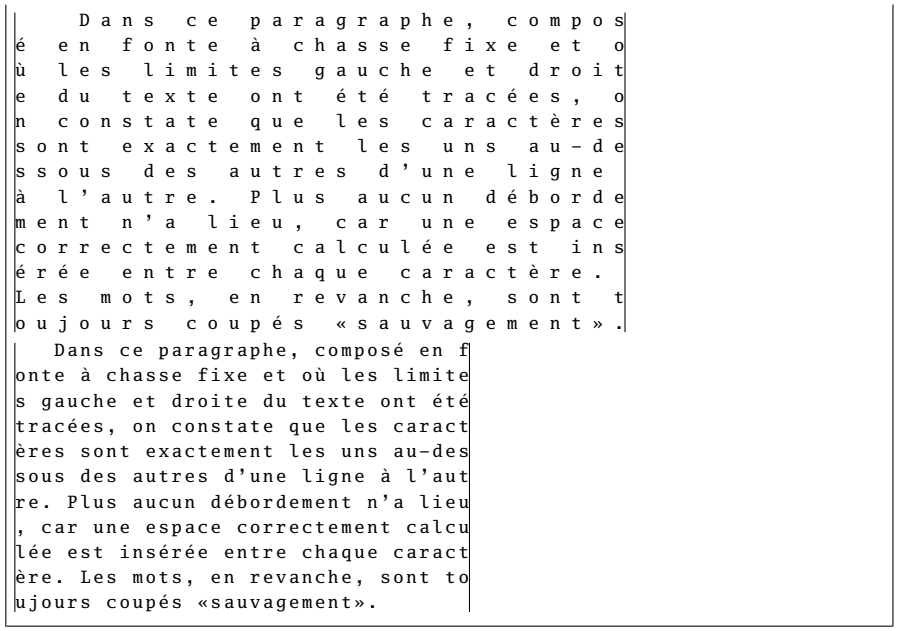
Le booléen \ifline@start, inutile ici, nous servira dans d'autres versions de la macro \breaktt.

Code n° V-422

```
\catcode`\@11
\newcount\brktt@cnt
\newdimen\brktt@interletter
\newif\ifline@start% booléen vrai lorsqu'aucun caractère n'est encore affiché
\newif\if@indivifound% booléen qui sera vrai si un motif spécial est rencontré

\def\insert@blankchar{%
  \ifstarred\insert@blankchar@ii\insert@blankchar@i
}

\def\insert@blankchar@i#1{% insère une espace de largeur #1 caractères complets
  \ifnum\numexpr#1\relax>0
    \kern\numexpr#1\relax\dimexpr\ttchar@width+\brktt@interletter\relax
  \fi
}

\def\insert@blankchar@ii#1{% insère #1-1 caractères complets + 1 largeur de caractère
  \ifnum\numexpr#1\relax>0
    \insert@blankchar@i{#1-1}\kern\ttchar@width
```

```
  \fi
}

\def\restart@hbox#1{%
  \egroup% feerme la \hbox précédente
  \hbox\bgroup% ouvre la suivante
  \expsecond{\def\tt@remaintext}
    {\romannumeral\removefirstspaces@i{#1}}% initialiser le code à composer
  \let\previous@char\space% initialise le caractère précédent
  \line@starttrue% aucun caractère n'a encore été imprimé
  \brktt@cnt=0\relax% remettre le compteur à 0
}

\newmacro\breaktt[0.3em][\hsize]1{%
% arg optionnel #1 et #2 = ressorts inter-lettre et dimension horizontale texte
% #3 = texte à espacer
  \begingroup% ouvrir un groupe et le fermer à la toute fin
  \par% commencer un nouveau paragraphe -> passage en mode vertical
  \parindent=0pt% empêche l'indentation
  \tt% passer en fonte à chasse fixe
  \setbox0 = \hbox{M}% la boite 0 contient un caractère
  \edef\ttchar@width{\the\wd0 }% largeur de chaque caractère en fonte \tt
  \edef\text@width{\the\dimexpr#2\relax}% largeur de composition
  % les 2 lignes suivantes rendent le compteur égal à E((L-l)/(l+Delta))
  \brktt@cnt=\numexpr\dimexpr#2-\wd0 \relax\relax% largeur diminuée du 1er caractère
  \divide\brktt@cnt by \numexpr\dimexpr\wd0 + #1 \relax\relax
  % le nombre de caractère par ligne est égal à 1 de plus :
  \edef\maxchar@num{\number\numexpr\brktt@cnt+1\relax}%
  % calcul de la dimension inter-lettre
  \brktt@interletter=\dimexpr(\text@width-\ttchar@width*\maxchar@num)/\brktt@cnt\relax
  % stocke le texte après avoir enlevé les éventuels espaces extremes :
  \expsecond{\expsecond{\def\tt@remaintext}}{\removetrailspaces{#3}}%
  \unless\ifx\tt@remaintext\empty% si le texte à composer n'est pas vide
    \hbox\bgroup% démarrer la boite horizontale contenant la première ligne
    \insert@blankchar\ttindent% insérer une espace d'indentation
    \brktt@cnt=\ttindent\relax% tenir compte du nombre de caractères indentés
    \line@starttrue% il s'agit du début d'une ligne
    \expandafter\breaktt@i% aller à la macro récursive
  \fi
}

\def\breaktt@i{%
  \print@nchar\maxchar@num% afficher \maxchar@num caractères
  \ifx\tt@remaintext\empty% si texte restant est vide
    \egroup% fermer la hbox
    \par% aller à la ligne
    \endgroup% fermer le groupe ouvert au début
  \else
    \unless\ifnum\brktt@cnt<\maxchar@num\relax% si la ligne est remplie
      \exparg\restart@hbox{\tt@remaintext}% ferme la hbox et en ré-ouvre une
    \fi
    \expandafter\breaktt@i% enfin, continuer le processus
   \fi
}

\def\print@nchar#1{% affiche #1 caractères pris dans \tt@remaintext
  \for\xxx= 1 to #1 \do 1{%
    \ifx\tt@remaintext\empty% si le code restant à composer est vide
      \exitfor\xxx%sortir de la boucle prématurément
    \else
```

```
      \@indivifoundfalse% sinon, a priori, les motifs de ligature ne sont pas trouvés
      % pour chaque \indivi@tmp dans la liste de ligatures
      \expsecond{\doforeach\indivi@tmp\in}{\liglist}%
        {% si le code commence par la ligature courante
        \exptwoargs\ifstartwith\tt@remaintext\indivi@tmp
          {\let\previous@char\indivi@tmp% prendre le motif pour caractère courant,
          \expsecond{\rightofsc\tt@remaintext}{\indivi@tmp}% l'enlever du texte restant
          \@indivifoundtrue% marquer qu'un motif a été trouvé
          \doforeachexit% et sortir prématurément de la boucle
          }%
          {}% si le code ne commence pas le motif courant -> ne rien faire
        }%
      \unless\if@indivifound% si aucun motif trouvé,
        \grab@first\tt@remaintext\previous@char%  lire le premier caractère
      \fi
      \advance\brktt@cnt by 1 % incrémenter le compteur de caractères
      \hbox to\ttchar@width{\hss\previous@char\hss}% afficher le caractère lu
      \line@startfalse% nous ne sommes plus au début d'une ligne
      \ifnum\brktt@cnt<\maxchar@num\relax% si la ligne n'est pas encore remplie
        \kern\brktt@interletter% insérer le ressort inter-lettre
      \else
        \exitfor\xxx% sinon, sortir de la boucle prématurément
      \fi
    \fi
  }%
}
\catcode`\@12

\def\liglist{<<,>>}% liste des motifs de ligature
                   % ajouter à, é, etc si codage UTF8 + moteur 8 bits
\def\ttindent{3}% valeur de l'indentation'
\vrule\vbox{\breaktt[4pt][.7\hsize]{%
Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
et droite du texte ont été tracées, on constate que les caractères sont
exactement les uns au-dessous des autres d'une ligne à l'autre. Plus aucun
débordement n'a lieu, car une espace correctement calculée est insérée entre
chaque caractère. Les mots, en revanche, sont toujours coupés <<sauvagement>>.%
}%
}\vrule\smallbreak

\vrule\vbox{\breaktt[1pt][6cm]{%
Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
et droite du texte ont été tracées, on constate que les caractères sont
exactement les uns au-dessous des autres d'une ligne à l'autre. Plus aucun
débordement n'a lieu, car une espace correctement calculée est insérée entre
chaque caractère. Les mots, en revanche, sont toujours coupés <<sauvagement>>.%
}%
}\vrule
```



3.4.5. Aligner et ne pas couper

Pour éviter que les mots ne soient coupés n'importe comment, la solution radicale consiste à interdire toute coupure. L'inévitable conséquence est que la contrainte n° 2 de la page 467 (composition justifiée) sera impossible à satisfaire.

Dans l'algorithme précédent, la macro `\print@nchar` de la ligne n° 62 effectuait la composition aveugle de `\maxchar@num` caractères à chaque ligne. Cette manière de faire est trop grossière et doit être affinée : au fur et à mesure de l'avancement, nous allons calculer la largeur du prochain mot `\next@len` par l'intermédiaire de la macro `\len@tonextcut`. Il faudra ensuite tester si ce mot loge en entier dans ce qu'il reste de la ligne en cours de composition. Dans l'affirmative, nous composerons ce mot avec la macro `\print@nchar` et sinon, il faudra finir la ligne en cours pour en commencer une nouvelle.

Le raffinement supplémentaire va être de tenir compte des tirets « – » éventuellement contenus dans les mots et rendre ces tirets comme possibles caractères de fin de ligne. Nous allons donc considérer qu'un mot est ce qui se trouve avant le plus proche des caractères « ␣ » ou « – ». Il faudra tenir compte du fait que le tiret *ne disparait pas* en fin de ligne contrairement à l'espace. Pour ce faire, une macro `\extra@char` contenant 0 ou 1 nous permettra de savoir par la suite si, en plus du mot courant, il faut loger un caractère de plus (le tiret) sur la ligne ou pas.

Nous allons baptiser `\breakttA*` cette variante qui compose du texte sans couper les mots. Les macros `\insert@blankchar`, `\restart@hbox` et `\print@nchar` de la précédente variante `\breaktt` seront réutilisées telles quelles. Elles ne sont donc pas réécrites dans le code ci-dessous. La macro `\breakttA@i` ne fait plus appel à une boucle `\for`, mais contient des appels récursifs jusqu'à ce que `\tt@remaintext` ne contienne plus de texte à composer.

Code nº V-423

```
\catcode`\@11
\newcount\brktt@cnt
\newdimen\brktt@interletter
\newif\ifline@start% booléen vrai lorsqu'aucun caractère n'est encore affiché
\newif\if@indivifound% booléen qui sera vrai si un motif spécial est rencontré

\newmacro\breakttA[0.3em][\hsize]1{%
% arg optionnel #1 et #2 = ressorts inter-lettre et dimension horizontale texte
% #3 = texte à espacer
  \begingroup% ouvrir un groupe et le fermer à la toute fin
  \par% commencer un nouveau paragraphe -> passage en mode vertical
  \parindent=0pt% empêche l'indentation
  \tt% passer en fonte à chasse fixe
  \setbox0 = \hbox{M}% la boite 0 contient un caractère
  \edef\ttchar@width{\the\wd0 }% largeur de chaque caractère en fonte \tt
  \edef\text@width{\the\dimexpr#2\relax}% largeur de composition
  % les 2 lignes suivantes rendent le compteur égal à E((L-l)/(l+Delta))
  \brktt@cnt=\numexpr\dimexpr#2-\wd0 \relax\relax% largeur diminuée du 1er caractère
  \divide\brktt@cnt by \numexpr\dimexpr\wd0 + #1 \relax\relax
  % le nombre de caractères par ligne est égal à 1 de plus :
  \edef\maxchar@num{\number\numexpr\brktt@cnt+1\relax}%
  % calcul de la dimension inter-lettre
  \brktt@interletter=\dimexpr(\text@width-\ttchar@width*\maxchar@num)/\brktt@cnt\relax
  % stocke le texte après avoir enlevé les éventuels espaces extremes :
  \expsecond{\expsecond{\def\tt@remaintext}}{\removetrailspaces{#3}}%
  \unless\ifx\tt@remaintext\empty% si le texte à composer n'est pas vide
    \hbox\bgroup% démarrer la boite horizontale contenant la première ligne
    \insert@blankchar\ttindent% insérer une espace d'indentation
    \brktt@cnt=\ttindent\relax% tenir compte du nombre de caractères indentés
    \line@starttrue% il s'agit du début d'une ligne
    \expandafter\breakttA@i% aller à la macro récursive
  \fi
}

\def\breakttA@i{%
  \edef\remaining@chars{% calculer le nombre de caractères restant à placer sur la ligne
    \numexpr\maxchar@num-\brktt@cnt\relax}%
  \len@tonextword% \next@len contient le nombre de caractères du prochain mot
  % si le mot + l'eventuel "-" qui le suit ne peut pas loger sur la ligne en cours
  \ifnum\numexpr\next@len+\extra@char\relax>\remaining@chars\relax
    \ifline@start% et si c'est le début d'une ligne
      % avertir l'utilisateur
      \message{Largeur de composition trop faible pour
        \unexpanded\expandafter{\next@word}^^J}%  ^^J
      % et composer tout de même "à la sauvage" jusqu'à la fin de la ligne
      \exparg\print@nchar{\number\numexpr\maxchar@num-\brktt@cnt}%
    \else% si la ligne en cours n'est pas au début
      \insert@blankchar*{\maxchar@num-\brktt@cnt}% remplir la ligne d'espaces
    \fi
    \exparg\restart@hbox{\tt@remaintext}% commencer une nouvelle ligne
    \expandafter\breakttA@i% et poursuivre le processus
  % s'il y a assez de place pour accueillir ce qui est avant la prochaine coupure
  \else
    \print@nchar\next@len% afficher le mot
    \ifx\tt@remaintext\empty% si texte restant est vide
      \insert@blankchar*{\maxchar@num-\brktt@cnt}% remplir la ligne
      \egroup% fermer la hbox en cours
      \par% aller à la ligne
      \endgroup% fermer le groupe ouvert au début. Fin du processus
    \else% si le texte restant n'est pas vide
```

```
      \ifnum\brktt@cnt<\maxchar@num\relax% si la ligne n'est pas remplie
          \print@nchar{1}% afficher le caractère qui suit le mot (" " ou "-")
      \else% si la ligne est remplie
        \exparg\restart@hbox{\tt@remaintext}% ferme la hbox et en ré-ouvre une
      \fi
      \expandafter\expandafter\expandafter\breakttA@i% enfin, continuer le processus
    \fi
  \fi
}

\def\leftofsc#1#2{% dans la macro #1, garde ce qui est à gauche de #2
  \def\leftofsc@i##1#2##2\@nil{\def#1{##1}}%
  \expandafter\leftofsc@i#1#2\@nil
}

\def\len@tonextword{% stocke dans \next@len le nombre de caractères avant
                    % le prochain point de coupure dans \tt@remaintext
  \let\next@word\tt@remaintext% copie \tt@remaintext dans la macro temporaire \next@word
  \leftofsc\next@word{ }% ne prend que ce qui est avant le prochain espace
  \exparg\ifin\next@word{-}% si le mot contient un tiret
    {\leftofsc\next@word{-}% prendre ce qui est à gauche de ce tiret
    \def\extra@char{1}% il y a un caractère de plus à loger après le mot
    }% sinon, le caractère après le mot est un espace
    {\def\extra@char{0}% qu'il ne faut pas compter
    }%
  \setbox0=\hbox{\next@word}% enfermer le mot dans une boite
  % et en calculer le nombre de caractères = dim(boite)/dim(caractère)
  \edef\next@len{\number\numexpr\dimexpr\wd0 \relax/\dimexpr\ttchar@width\relax\relax}%
}
\catcode`\@12

\def\liglist{<<,>>}% liste des motifs de ligature (mettre à, é, etc si codage UTF8)
\def\ttindent{3}

\vrule\vbox{\breakttA[3pt][8cm]{%
Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
et droite du texte ont été tracées, on constate que les caractères sont
exactement les uns au-dessous d'une ligne à l'autre. Plus aucun débordement
n'a lieu, car une espace correctement calculée est insérée entre chaque
caractère. Dorénavant, les mots ne sont plus coupés <<~sauvagement~>>.
}}\vrule\medbreak

\leavevmode\vrule\vbox{\breakttA[1pt][5cm]{%
Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
et droite du texte ont été tracées, on constate que les caractères sont
exactement les uns au-dessous d'une ligne à l'autre. Plus aucun débordement
n'a lieu, car une espace correctement calculée est insérée entre chaque
caractère. Dorénavant, les mots ne sont plus coupés <<~sauvagement~>>.
}}\vrule\qquad\def\ttindent{0}%
\vrule\vbox{\breakttA[0.6pt][2cm]{mot-composé mot-clé passe-droit au-dessus
là-bas remonte-pente vingt-huit Notre-Dame-de-Lourdes Mont-Blanc
Saint-Jean-de-Luz}}\vrule
```

```
    Dans ce paragraphe, composé en
fonte à chasse fixe et où les
limites gauche et droite du texte
ont été tracées, on constate que
les caractères sont exactement
les uns au-dessous d'une ligne à
l'autre. Plus aucun débordement
n'a lieu, car une espace
correctement calculée est insérée
entre chaque caractère.
Dorénavant, les mots ne sont plus
coupés « sauvagement ».
```

```
   Dans ce paragraphe,
composé en fonte à chasse
fixe et où les limites gauche
et droite du texte ont été
tracées, on constate que les
caractères sont exactement
les uns au-dessous d'une
ligne à l'autre. Plus aucun
débordement n'a lieu, car une
espace correctement calculée
est insérée entre chaque
caractère. Dorénavant, les
mots ne sont plus coupés
« sauvagement ».
```

```
mot-composé
mot-clé
passe-droit
au-dessus
là-bas
remonte-
pente vingt-
huit Notre-
Dame-de-
Lourdes
Mont-Blanc
Saint-Jean-
de-Luz
```

3.4.6. Aligner et couper

Entrons maintenant dans la phase la plus difficile… Nous allons essayer de composer du texte en fonte à chasse fixe tout en alignant les caractères les uns au-dessous des autres et en permettant que les coupures de mots se fassent.

Voici dans les grandes lignes la démarche que nous allons suivre lorsque le prochain mot ne loge pas sur l'espace restant de la ligne en cours :

1. pour le prochain mot du texte à composer (un « mot » étant ce qui se trouve avant le plus proche espace ou tiret), nous allons trouver toutes les coupures possibles ;
2. nous allons mesurer les longueurs des syllabes ainsi obtenues et cumuler leurs longueurs de façon à construire la liste des nombres croissants exprimant les longueurs (en caractères) jusqu'aux coupures possibles du mot ;
3. enfin, nous parcourons cette liste et effectuerons, si c'est possible, la coupure la plus longue possible tout en ne dépassant pas le nombre de caractères qu'il reste à composer sur la ligne en cours. Ceci fait, il nous restera à enfermer la ligne en cours dans une `\hbox` et en commencer une nouvelle.

Comme plusieurs difficultés se dressent devant nous, nous allons procéder pas à pas.

Provoquer toutes les coupures

Tout d'abord, comment forcer TEX à effectuer toutes les coupures possibles pour un texte donné ? Pour l'obliger à procéder de la sorte, nous allons composer ce texte dans une boite verticale de largeur *nulle*. Bien sûr, des débordements auront lieu, mais nous neutraliserons leur signalement en réglant `\hfuzz` à `\maxdimen`.

Avant de commencer à composer du texte en largeur nulle, certains réglages de composition doivent être faits. Intéressons-nous à la façon dont TEX fabrique un paragraphe et comment nous pouvons l'influencer. Il faut savoir que *plusieurs* passes peuvent être faites par TEX pour composer un paragraphe [9]. La première est tentée si l'entier `\pretolerance` est positif et a lieu sans faire de coupures de mots. Ceci fait, si aucune ligne du paragraphe obtenu n'a de médiocrité [10] supérieure à l'entier `\pretolerance`, le paragraphe ainsi obtenu sera retenu. Dans le cas contraire, une seconde passe, où les coupures de mots sont permises, est effectuée lors de laquelle, si c'est possible, TEX s'efforce de fabriquer des lignes de telle sorte qu'aucune n'ait de médiocrité supérieure à l'entier `\tolerance`.

Comme nous cherchons à provoquer les coupures de mots, la première passe ne nous intéresse pas et nous règlerons donc `\pretolerance` à −1. Par ailleurs, nous fixerons `\tolerance` à 10000 de telle sorte que toutes les lignes construites à la deuxième passe soient acceptables [11] aux yeux de TEX.

Ensuite, il nous faut modifier localement les réglages de composition en vue du but que nous nous sommes fixé. Comme la boite verticale de longueur nulle que nous obtiendrons a vocation à être coupée en lignes avec `\vsplit`, nous devons nous assurer qu'aucune pénalité additionnelle entre les lignes n'interdit cette coupure. Voici donc les réglages à effectuer :

- l'entier `\hyphenpenalty`, qui caractérise la pénalité d'une coupure de mots sera pris égal à −10000 et donc les coupures, extrêmement favorisées par cette pénalité négative, seront toujours effectuées. Par ailleurs, `\exhyphenpenalty` est la pénalité d'une coupure sur un caractère explicite (le tiret). Notre algorithme ne donnera jamais à composer un mot contenant un tiret puisqu'un « mot » s'étend jusqu'au prochain espace ou tiret. Par conséquent, `\exhyphenpenalty`, rendu égal à 0, ne jouera aucun rôle.
- l'entier `\clubpenalty` est ajouté à la pénalité de coupure de ligne pour la première ligne du paragraphe. Nous prendrons 0 comme valeur de cet entier. L'entier `\widowpenalty` agit de même pour la dernière ligne : il sera réglé à 0 ;
- l'entier `\interlinepenalty`, ajouté à la pénalité d'une coupure de ligne, sera pris égal à 0 ;
- les entiers `\lefthyphenmin` et `\righthyphenmin`, qui caractérisent la longueur minimale du fragment laissé au début ou à la fin d'un mot coupé, seront réglés à leurs valeurs par défaut, soit 2 et 3 ;
- nous réinitialiserons `\rightskip` et `\leftskip` à `0pt` au cas où ces dimensions ne seraient pas nulles au moment de la composition. Le ressort `\parfillskip` sera réinitialisé à « `0pt plus 1fil` » qui est sa valeur par défaut. Nous viderons également `\everypar` qui contient la liste de tokens exécutée au début de chaque paragraphe ;
- l'entier `\hyphenchar` sera sauvegardé puis pris égal à `'\-` et enfin restauré en fin de composition ;
- l'indentation sera désactivée avec `\noindent` tout en provoquant le passage en mode horizontal ;

9. Pour en savoir plus, lire le TEXbook pages 112–113.
10. Lire le TEXbook page 144 notamment pour comprendre comment est calculée la médiocrité d'une ligne.
11. Toute valeur supérieure à 10000 est comprise comme étant égale à 10000.

– un ressort de longueur nulle sera inséré avant le texte à composer, car une coupure ne peut intervenir sur un mot si ce mot commence le paragraphe. Ce ressort tiendra lieu de remplissage factice et autorisera la coupure du premier mot ;
– la largeur de composition \hsize sera rendue égale à 0pt.

Forts de tous ces réglages, nous allons bâtir la macro \zerocompose* :

\zerocompose[⟨*code*⟩]{⟨*registre de boite*⟩}{⟨*texte*⟩}

où ⟨*code*⟩ est un code optionnel qui sera exécuté avant que la composition ne commence, ⟨*registre de boite*⟩ est le numéro du registre de la boite verticale \vbox recevant le résultat de la composition du ⟨*texte*⟩.

Code n° V-424

```
\newmacro\zerocompose[]2{%
% #1=code à exécuter avant la composition
% #2=registre de boite recevant le résultat
% #3= texte à composer en largeur 0pt
  \setbox#2=\vbox{%
    #1% code a exécuter (changement de fonte par exemple)
    \hfuzz=\maxdimen% annule les avertissements pour débordement horizontaux
    \hbadness=10000 % annule les avertissements pour mauvaise boite horizontale
    \pretolerance=-1 % désactive la première passe (celle sans coupures)
    \tolerance=10000 % passe avec coupures acceptée
    \hyphenpenalty=-10000 % favorise fortement les copures de mots
    \lefthyphenmin=2 \righthyphenmin=3 % longueur mini des fragments de début et fin
    \clubpenalty=0 % pas de pénalité supplémentaire après la première ligne
    \interlinepenalty=0 % pas de pénalité inter-ligne
    \widowpenalty=0 % pas de pénalité supplémentaire avant la dernière ligne
    \exhyphenpenalty=0 % ne pas pénaliser une coupure explicite
    \leftskip=0pt \rightskip=0pt % désactive les éventuels ressorts latéraux
    \everypar={}% désactive l'éventuel \everypar
    \parfillskip=0pt plus1fil % règle le \parfillskip par défaut
    \hsize=0pt % largeur de composition = 0pt
    \edef\restorehyphenchar{\hyphenchar\font=\number\hyphenchar\font}%
    \hyphenchar\font=`\- % impose "-" comme caractère de coupure
    \noindent % pas d'indentation + passage en mode horizontal
    \hskip0pt \relax% premier noeud horizontal pour permettre la coupure de la suite
    #3\par% compose #3
    \restorehyphenchar% restaure le caractère de coupure
    }%
}
\zerocompose{0}{Programmation}%
\leavevmode\box0
\hskip 5cm
\zerocompose{0}{Composer en largeur nulle}%
\box0
```

Programmation Composer en largeur nulle

Malgré les apparences, les boites n° 0 affichées dans l'exemple ci-dessus ont bel et bien une largeur *nulle*. C'est \hskip 5cm qui évite qu'elles ne se chevauchent.

Nous allons appliquer cette méthode à des mots seuls et utiliser \vsplit pour

vider peu à peu (par le haut) la boite verticale résultante afin d'obtenir à chaque itération une boite horizontale contenant une ligne de la boite initiale. Le problème est que la boite récupérée sera de largeur nulle puisque la \vbox initiale avait cette largeur. Dès lors, il sera impossible de la mesurer pour trouver le nombre de caractères qui la composent !

Nous allons donc faire appel à une primitive, non encore vue jusqu'alors qui va nous permettre de redonner à la boite contenant la première ligne sa largeur naturelle.

La primitive \lastbox

97 - Règle

La primitive \lastbox a un comportement très particulier :

- non seulement elle contient la dernière boite composée dans la liste en cours, mais utiliser \lastbox *retire* cette boite de la liste dans laquelle elle était placée ;
- si le dernier élément n'est pas une boite, \lastbox est vide ;
- si l'on souhaite accéder à la dernière boite composée, \lastbox s'utilise seule. Il est incorrect de la faire précéder de \box.

L'opération \lastbox n'est pas permise dans le mode mathématique ni dans le mode vertical *principal* : elle ne peut donc pas servir à retirer une boite de la page courante en cours de composition. En revanche, elle peut très bien être utilisée dans le mode vertical interne.

La boite \lastbox hérite de la nature – horizontale ou verticale – de la dernière boite composée.

Si un registre de boite est vidé par \lastbox, le registre ne devient pas vide, mais contient une boite vide (le test \ifvoid est donc inadapté pour tester si un registre est vidé par \lastbox).

La façon habituelle de se servir de \lastbox est de capturer la dernière boite pour l'assigner à un registre de boite normal que l'utilisateur peut ensuite manier à sa guise. On rencontre donc la syntaxe

\setbox⟨*registre de boite*⟩=\lastbox

Par exemple, à l'intérieur d'un groupe et avec l'aide de la boite brouillon n° 0, \lastbox permet d'effacer la dernière boite composée :

```
{\setbox0 =\lastbox}
```

On peut également s'en servir pour capturer la dernière ligne d'un paragraphe composé dans une boite verticale (dans ce cas, \lastbox sera une boite horizontale, celle qui contient la dernière ligne) :

Code n° V-425

```
\def\dummytext{Ceci est un texte sans aucun intérêt dont le seul but est de meubler
   la page de façon artificielle. }
\setbox0=\vtop{% définit la boite 0
  \hsize=8cm
  \dummytext\dummytext
}
```

```
7
8 a) Boite pleine : \copy0
9
10 b) \setbox0=\vtop{%
11   \unvbox0 % boite précédemment composée
12   \global\setbox1=\lastbox% et capture la dernière ligne dans la boite 1
13   }%
14   Dernière ligne = |\hbox{\unhbox1 }|\par% redonne à box1 sa largeur naturelle
15   Boite restante = \box0
```

```
a) Boite pleine : Ceci est un texte sans aucun intérêt dont le seul but est de meubler la
                  page de façon artificielle. Ceci est un texte sans aucun intérêt dont le
                  seul but est de meubler la page de façon artificielle.
b) Dernière ligne = |seul but est de meubler la page de façon artificielle.|
Boite restante = Ceci est un texte sans aucun intérêt dont le seul but est de meubler la
                  page de façon artificielle. Ceci est un texte sans aucun intérêt dont le
```

Cet exemple illustre comment `\lastbox` permet de « couper » une boite verticale par le bas. La similarité devient évidente entre `\lastbox` qui coupe par le bas et `\vsplit` qui coupe par le haut. En répétant la manœuvre plusieurs fois, il deviendrait possible de vider peu à peu une boite verticale en enlevant à chaque itération la ligne du bas :

Code nº V-426

```
\def\dummytext{Ceci est un texte sans aucun intérêt dont le seul but est de meubler
   la page de façon artificielle. }
\setbox0=\vtop{% définit la boite 0
  \hsize=8cm
  \dummytext\dummytext
}

\setbox0=\vtop{\unvbox0 \global\setbox1=\lastbox}
\setbox1=\hbox{\unhbox1 }
1) |\box1|

\setbox0=\vtop{\unvbox0 \global\setbox1=\lastbox}
\setbox1=\hbox{\unhbox1 }
2) |\box1|

\setbox0=\vtop{\unvbox0 \global\setbox1=\lastbox}
\setbox1=\hbox{\unhbox1 }
3) |\box1|
```

```
1) |seul but est de meubler la page de façon artificielle.|
2) ||
3) ||
```

Pourquoi `\lastbox` échoue à capturer la dernière ligne dans les deux derniers cas ? C'est que `\lastbox` est plus contraignante que `\vsplit` et la réponse demande de connaitre un peu la façon dont est construit un paragraphe par TEX : le ressort d'interligne, qui est inséré entre chaque boite horizontale contenant une ligne, est assorti d'une pénalité qui le précède.

Pour s'en persuader, il est possible de faire remonter des entrailles de TEX vers le fichier `log` les composants élémentaires des boites qu'il construit. Pour ce faire, plusieurs réglages sont à faire :

– pour que les informations ne soient écrites que dans le fichier `log` (et non

pas également sur le terminal), il faut que la primitive \tracingonline soit strictement positive ;
- l'entier \showboxdepth spécifie le nombre de niveaux d'imbrication de boites qui doivent être montrés ;
- l'entier \showboxbreadth spécifie le nombre d'éléments montrés dans chaque niveau ;
- la primitive \showbox , suivie d'un registre de boite donne l'ordre à TEX d'examiner la boite contenue dans le registre et de délivrer les informations selon les réglages spécifiés.

Code n° V-427

```
\setbox0=\vbox{\hsize=2.5cm Programmer en TeX est facile}%
\showboxbreadth=5 \showboxdepth=3 \tracingonline=1
\showbox0
```

```
> \box0=
\vbox(14.77199+0.092)x71.13188
.\hbox(5.27199+1.888)x71.13188, glue set - 1.0
..\hbox(0.0+0.0)x0.0
..\T1/LinuxLibertineT-TLF/m/n/8 P
..\T1/LinuxLibertineT-TLF/m/n/8 r
..\kern-0.06401
..\T1/LinuxLibertineT-TLF/m/n/8 o
..etc.
.\penalty300
.\glue(\baselineskip) 2.08801
.\hbox(5.52399+0.092)x71.13188, glue set 53.71591fil
..\T1/LinuxLibertineT-TLF/m/n/8 f
..\T1/LinuxLibertineT-TLF/m/n/8 a
..\discretionary
...\T1/LinuxLibertineT-TLF/m/n/8 -
..\T1/LinuxLibertineT-TLF/m/n/8 c
..\T1/LinuxLibertineT-TLF/m/n/8 i
..etc.
```

On constate que la pénalité (ici de 300) est bien insérée *avant* le ressort d'interligne (\baselineskip). Notons par ailleurs que les « .\hbox » représentent les boites horizontales contenant les lignes (le nombre de « . » indique le niveau d'imbrication des boites, étant entendu qu'une absence de point indique la boite mère). Il faut également remarquer, juste après le « r », que « \kern-0.06401 » règle le crénage (kerning) entre les caractères « r » et « o » dans la fonte « LinuxLibertine-TLF » et montre donc que pour des raisons esthétiques, ces caractères sont légèrement rapprochés.

Pour en revenir à \lastbox, lorsqu'elle a capturé la dernière ligne, elle laisse en bas de la boite restante une pénalité et un ressort vertical. Par conséquent, la \lastbox suivante, ne pouvant capturer une boite, sera vide. Pour que la méthode fonctionne, il faut supprimer *dans cet ordre* le ressort avec \unskip puis la pénalité avec \unpenalty avant d'extraire la prochaine \lastbox. Construire une macro \reversebox qui vide une boite verticale par le bas devient une simple routine, à condition de ne pas effectuer le test \ifvoid, inadapté à un registre vidé par \lastbox, mais les tests \ifzerodimbox ou \ifvoidorempty que nous avons programmés (voir page 264 et suivantes) :

Code n° V-428

```
\newbox\tempbox
\def\reversebox#1{% affiche le contenu de la boite verticale #1
  \setbox#1=\vbox{%
    \unvbox#1% composer la boite #1
    \unskip\unpenalty% retirer ce qui n'est pas capturable par \lastbox
    \global\setbox0=\lastbox% la boite 1 devient la dernière ligne
  }%
  |\hbox{\unhbox0 }|\par% afficher la boite 1 dans sa largeur d'origine
  \ifvoidorempty{#1}% si le registre #1 contient une boite vide
    \relax% ne rien faire
    {\reversebox{#1}}% sinon, rencommencer
}
\def\dummytext{Ceci est un texte sans aucun intérêt dont le seul but est de meubler
    la page de façon artificielle. }
\setbox\tempbox=\vbox{% définit la boite 0
  \hsize=8cm \dummytext\dummytext
}

\reversebox\tempbox
```

```
|seul but est de meubler la page de façon artificielle.|
|page de façon artificielle. Ceci est un texte sans aucun intérêt dont le|
|Ceci est un texte sans aucun intérêt dont le seul but est de meubler la|
```

Mettons tout ceci en application. La macro `\zerocompose` va nous permettre, conjointement avec `\lastbox` et `\vsplit`, de récupérer une boite horizontale de largeur naturelle contenant la première ligne.

Code n° V-429

```
% compose "Programmation" en largeur 0 dans la boite 0 :
\zerocompose{0}{Programmation}%
Boite initiale : \copy0 \par% compose la boite résultante
\vfuzz=\maxdimen% annule les avertissements pour débordements
\splittopskip=0pt % ne rajouter aucun espace au sommet de la boite restante
\setbox1 =\vsplit0 to 0pt % coupe la boite 0 à 0pt
{% dans un groupe (où la boite 0 sert de brouillon)
  \setbox0 =\vbox{% affecter à la boite brouillon
    \unvbox1 % la boite 1 composée dans sa largeur naturelle
    \unskip\unpenalty% annule ce qui n'est pas une boite
    \global\setbox2 =\lastbox% affecte globalement la dernière (et unique) ligne
                             % à la boite 1
  }%
}% ferme le groupe
\setbox2=\hbox{\unhbox2}% rend à la boite 2 sa largeur naturelle
Première ligne = "\box2 "% compose la boite 2
```

```
                Pro-
                gram-
                ma-
Boite initiale : tion
Première ligne = "Pro-"
```

Mesurer toutes les coupures

Le but est de construire une macro contenant les longueurs (en caractères) avant les possibles coupures du mot, en tenant compte du caractère de coupure. Par exemple, le mot « programmation » aura 3 coupures possibles auxquelles correspondent

les longueurs suivantes :

Caractères avant la coupure	Longueur
pro–	4
program–	8
programma–	10

Nous allons donc construire une macro

\hyphlengths{⟨*mot*⟩}{\⟨*macro*⟩}

qui assigne à la \⟨*macro*⟩ la liste de toutes les longueurs avant les coupures possibles. Par exemple, si l'on écrit « \hyphlengths{programmation}\foo », la macro \foo a comme texte de remplacement « 4,8,10 ».

La méthode pour y parvenir fera intervenir

\vsplit⟨*boite*⟩ to 0pt

pour couper la boite résultante de la macro \zerocompose et en obtenir la première ligne dans une boite verticale. Ensuite, en composant cette boite dans une boite verticale, en capturant la dernière boite horizontale composée avec \lastbox puis en appliquant \unhbox, cette ligne se retrouvera dans une boite horizontale ayant sa largeur naturelle. Dès lors qu'on peut la mesurer, il sera facile de trouver le nombre de caractères sur chaque ligne (4, 5 et 3 pour « pro– », « gram– » et « ma– ») : la simple division de la largeur de la boite par la largeur d'un caractère donne le quotient cherché. Il faudra lui soustraire 1 pour écarter le caractère de coupure. En cumulant ces nombres et en ajoutant 1 à chaque cumul pour prendre en compte le caractère de coupure, on arrive bien à la liste « 4,8,10 ».

Code n° V-430

```
\catcode`@11
\def\hyphlengths#1#2{%#2 = macro contenant les longueurs de coupures du mot #1
  \begingroup
    \zerocompose
      [\tt% passer en fonte à chasse fixe
      \setbox\z@\hbox{M}\xdef\ttwidth{\the\wd\z@}% mesurer la largeur des caractères
      ]\z@{#1}% compose en 0pt dans la boite 0
    \let#2 = \empty% initialise la macro #2
    \def\cumul@length{0}% le cumul des longueurs initialisé à 0
    \vfuzz=\maxdimen% annule les avertissements pour débordement
    \splittopskip=\z@ % ne rajouter aucun espace au sommet de la boite restante
    \loop
      \setbox1=\vsplit\z@ to \z@% couper la boite à 0pt de hauteur
      {\setbox\z@=\vbox{\unvbox1 \unskip\unpenalty\global\setbox1=\lastbox}}%
      \setbox1=\hbox{\unhbox1 }%
      \unless\ifvoid\z@% si la boite 0 n'est pas encore vide
        \edef\cumul@length{% mettre à jour \cumul@length
          \number\numexpr
            \cumul@length
            +% ajouter le quotient "largeur syllabe/largeur d'1 caractère"
            \wd1/\dimexpr\ttwidth\relax
            -1% et soustraire 1 (le caractère "-")
            \relax
        }%
        % ajouter à #2 la virgule et le cumul actuel +1
        \edef#2{% définir la macro #2 :
          #2% reprendre le contenu de #2
          \ifx#2\empty\else,\fi% ajouter "," si #2 non vide
```

```
          \number\numexpr\cumul@length+1\relax% et le cumul
          }%
    \repeat% et recommencer
    \expsecond{% avant de fermer le groupe
  \endgroup
  \def#2}{#2}% définit #2 hors du groupe
}
\catcode`\@12
a) \hyphlengths{programmation}\foo liste = "\foo"\par
b) \hyphlengths{typographie}\foo liste ="\foo"\par
c) \hyphlengths{j'entendrai}\foo liste ="\foo"\par
d) \hyphlengths{vite}\foo liste ="\foo"
```

a) liste = "4,8,10"
b) liste ="3,5,8"
c) liste ="5,8"
d) liste =""

Une solution

Il est temps de mettre en œuvre toute cette théorie. Curieusement, l'algorithme vu précédemment avec `\breakttA` n'a pas à subir beaucoup de changements pour autoriser les coupures de mots.

Tout ce qui est nouveau va être contenu dans la macro récursive `\breakttA@i`. En premier lieu, après avoir calculé la longueur du prochain mot (qui s'étend jusqu'à un espace ou un tiret), nous devons tester si ce mot (ainsi que l'éventuel tiret qui le suit) peut loger sur l'espace restant de la ligne en cours. Ce n'est qu'en cas de réponse négative que nous devons entreprendre la coupure du mot. À l'aide de `\hyphlengths` et en parcourant la liste de longueurs cumulées qu'elle génère, nous pouvons trouver combien mesure la plus longue coupure possible de ce mot compatible avec la place restante sur la ligne. La macro `\nexthyph@len` sera cette longueur. À l'issue de ce processus, si cet entier est nul, soit aucune coupure n'était nécessaire, soit aucune coupure n'a été trouvée. Dans le cas où il n'est pas nul, il sera copié dans `\next@len`.

Si une coupure doit être faite, le test suivant, qui s'assure que `\next@len` est inférieur au nombre de caractères restant à placer, est toujours vrai. Nous devons alors afficher toutes les lettres de la syllabe retenue (c'est-à-dire `\next@len`–1 caractères), puis afficher un tiret, mission qu'exécutera la macro `\print@hyphchar`. Il reste ensuite à remplir la ligne avec des caractères blancs, fermer et ouvrir une nouvelle boite et appeler la macro `\breakttA@i`.

Seules les macros `\breakttA@i` et `\print@hyphchar` ont été écrite dans cet exemple puisque toutes les autres macros déjà vues sont strictement identiques.

Code n° V-431

```
\catcode`\@11
\newcount\brktt@cnt
\newdimen\brktt@interletter
\newif\ifline@start% booléen vrai lorsqu'aucun caractère n'est encore affiché
\newif\if@indivifound% booléen qui sera vrai si un motif spécial est rencontré

\let\breakttB=\breakttA

\def\breakttA@i{%
```

```
  \edef\remaining@chars{% calculer le nombre de caractères restant à placer sur la ligne
    \numexpr\maxchar@num-\brktt@cnt\relax}%
  \len@tonextword% calculer dans \next@len le nombre de caractères avant le prochain mot
  \def\next@hyphlen{0}% initialiser la longueur de coupure possible du mot
  % si le mot + l'eventuel "-" qui le suit ne rentre pas sur ce qui reste de la ligne
  \ifnum\numexpr\next@len+\extra@char\relax>\remaining@chars\relax
    \hyphlengths\next@word\list@hyphlengths% bâtir la liste des longueurs de coupures
    \unless\ifx\list@hyphlengths\empty% si une coupure est possible
      \expsecond{\doforeach\xx\in}{\list@hyphlengths}% les examiner toutes
        {% si la coupure examinée peut loger sur la ligne
        \unless\ifnum\xx>\remaining@chars\relax%
          \let\next@hyphlen\xx% la stocker
        \fi% pour que \next@hyphlen soit maximal
        }%
    \fi
  \fi
  \ifnum\next@hyphlen>0 % si une coupure est nécessaire et a été trouvée
    \let\next@len\next@hyphlen% mettre à jour la longueur du mot à placer
  \fi
  % si le mot + l'eventuel "-" qui le suit ne peut pas loger sur la ligne en cours
  \ifnum\numexpr\next@len+\extra@char\relax>\remaining@chars\relax
    \ifline@start% si c'est le début d'une ligne
      % avertir l'utilisateur
      \message{Largeur de composition trop faible pour
         \unexpanded\expandafter{\next@word}^^J}% ^^J
      % et composer tout de même jusqu'à la fin de la ligne
      \exparg\print@nchar{\number\numexpr\maxchar@num-\brktt@cnt}%
    \else% sinon
      \insert@blankchar*{\maxchar@num-\brktt@cnt}% remplir la ligne
    \fi
    \exparg\restart@hbox{\tt@remaintext}% ré-ouvrir une boite
    \expandafter\breakttA@i% et poursuivre le processus
  % s'il y a la place pour accueillir ce qui est avant la prochaine coupure
  \else
    \ifnum\next@hyphlen>0 % si une coupure de mot doit être faite
      \exparg\print@nchar{\number\numexpr\next@len-1}% afficher les lettres de la syllabe
      \print@hyphchar% et le caractère de coupure
      \insert@blankchar*{\maxchar@num-\brktt@cnt}% remplir la ligne
      \exparg\restart@hbox{\tt@remaintext}% ré-ouvrir une boite
      \expandafter\expandafter\expandafter\breakttA@i% et continuer le processus
    \else% si pas de coupure dans le mot
      \print@nchar\next@len% afficher le mot
      \ifx\tt@remaintext\tt@emptytext% si texte restant est vide
        \insert@blankchar*{\maxchar@num-\brktt@cnt}% remplir la ligne
        \egroup% fermer la hbox
        \par% aller à la ligne
        \endgroup% fermer le groupe ouvert au début. Fin du processus
      \else% si le texte restant n'est pas vide
        \ifnum\brktt@cnt<\maxchar@num\relax% si la ligne n'est pas remplie
            \print@nchar{1}% afficher le caractère qui suit le mot
        \else% si la ligne est remplie
          \exparg\restart@hbox{\tt@remaintext}% ferme la hbox et en ré-ouvre une
        \fi
        \expandafter\expandafter\expandafter\breakttA@i% enfin, continuer le processus
      \fi
    \fi
  \fi
}

\def\print@hyphchar{%
```

```
  \advance\brktt@cnt by 1 % augmenter le compteur de caractères
  \hbox to\ttchar@width{\hss-\hss}% afficher "-"
  \ifnum\brktt@cnt<\maxchar@num\relax% si la ligne n'est pas encore remplie
    \kern\brktt@interletter\relax% insérer le ressort inter-lettre
  \fi
}

\catcode`\@12

\def\liglist{<<,>>}% liste des motifs de ligature (mettre à, é, etc si codage UTF8)
\def\ttindent{3}

\vrule\vbox{\breakttB[3pt][8cm]{%
Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
et droite du texte ont été tracées, on constate que les caractères sont
exactement les uns au-dessous d'une ligne à l'autre. Plus aucun débordement
n'a lieu, car une espace correctement calculée est insérée entre chaque
caractère. Dorénavant, les mots ne sont plus coupés <<sauvagement>>.
}%
}\vrule

\vrule\vbox{\breakttB[1pt][5cm]{%
Dans ce paragraphe, composé en fonte à chasse fixe et où les limites gauche
et droite du texte ont été tracées, on constate que les caractères sont
exactement les uns au-dessous d'une ligne à l'autre. Plus aucun débordement
n'a lieu, car une espace correctement calculée est insérée entre chaque
caractère. Dorénavant, les mots ne sont plus coupés <<sauvagement>>.
}%
}\vrule
```

```
    Dans ce paragraphe, composé en
fonte à chasse fixe et où les li-
mites gauche et droite du texte
ont été tracées, on constate que
les caractères sont exactement
les uns au-dessous d'une ligne à
l'autre. Plus aucun débordement
n'a lieu, car une espace correc-
tement calculée est insérée entre
chaque caractère. Dorénavant, les
mots ne sont plus coupés «sauva-
gement».
   Dans ce paragraphe, com-
posé en fonte à chasse fixe
et où les limites gauche et
droite du texte ont été tra-
cées, on constate que les ca-
ractères sont exactement les
uns au-dessous d'une ligne à
l'autre. Plus aucun déborde-
ment n'a lieu, car une espace
correctement calculée est in-
sérée entre chaque caractère.
Dorénavant, les mots ne sont
plus coupés «sauvagement».
```

La macro `\breakttA@i` est la même que vue précédemment sauf que les lignes n^{os} 13–28 et 44–50 sont ajoutées pour permettre les coupures de mots.

Concluons ce difficile défi en remarquant qu'en nous aidant de celui de TEX, nous avons *programmé* un algorithme de coupures, mais il n'a pas toutes les fonc-

tionnalités de l'original. En effet, les coupures se feront *toujours* lorsqu'elles sont possibles. Il n'y a par exemple aucun moyen de décourager des coupures, que ce soit sur des lignes adjacentes comme l'augmentation de `\doublehyphendemerits` le permet ni sur l'avant-dernière ligne d'un paragraphe avec `\finalhyphendemerits`. Nous avons cependant gagné sur un point : les coupures se feront sur des mots comprenant un trait d'union, aussi bien sur ce trait que sur le mot qui le précède ou le suit, alors que TEX n'autorise une coupure que sur le trait d'union.

3.5. Afficher des algorithmes

Bien que la programmation qui entre en jeu ici est *linéaire* (c'est-à-dire sans boucle), il est intéressant de décortiquer la méthode à déployer pour afficher des algorithmes sous forme de pseudocode comme cela est parfois fait dans ce livre (voir par exemple ceux des pages 162, 197 ou 436).

La syntaxe et les fonctionnalités

Commençons par la syntaxe finale. La macro `\algorithm*` admettra un argument optionnel qui est la liste des tokens dont le catcode sera rendu égal à 12 pendant le pseudocode. Elle admettra aussi un argument obligatoire qui est le titre de l'algorithme (éventuellement vide). Le pseudocode proprement dit est contenu entre deux tokens identiques, à la mode de `\litterate` :

`\algorithm[`⟨*liste de tokens*⟩`]{`⟨*titre*⟩`}`⟨*délimiteur*⟩⟨*texte*⟩⟨*délimiteur*⟩

Pour que la saisie soit la plus « naturelle » possible, voici les principales fonctionnalités à mettre en œuvre :

1. chaque caractère de tabulation `^^I^^I` sera traduit par une espace horizontale. La valeur de cette espace est contenue dans une macro `\algoindent` ;
2. les espaces perdent leur comportement naturel et plusieurs espaces consécutifs seront affichés ;
3. les retours à la ligne `^^M` perdent également leur comportement naturel et sont reprogrammés pour provoquer un retour à la ligne ;
4. le caractère ~ reste actif, mais est reprogrammé pour que le ⟨*texte*⟩ se trouvant entre deux ~ soit affiché en caractères gras ;
5. les caractères « `:=` », utilisés pour une assignation, sont traduits à l'affichage par « ← », provoqué par la macro de plain-TEX `\leftarrow` ;
6. le caractère « _ » garde son comportement normal en mode mathématique mais est affiché tel quel sinon ;
7. le caractère « `%` » met en italique et en gris tout ce qui reste sur la ligne courante ;
8. les lignes sont numérotées dans la marge de gauche, mais il est possible de désactiver cette fonctionnalité.

Voici ce que nous devons pouvoir écrire (où « ⟶ » représente le caractère de tabulation) :

Code n° V-432

```
Voici un algorithme élémentaire :
\algorithm[\#]{Algorithme {\bf MinMax}}|
macro ~Min_Max~ (#1) % afficher la valeur max et min d'une liste de valeurs #1
⟶ $V_{max}:=-\infty$ % $-\infty$ ou la plus petite valeur possible
⟶ $V_{min}:=+\infty$ % $+\infty$ ou la plus grande valeur possible
⟶ ~Pour~ chaque $x$ dans (#1)
⟶ ⟶ ~Si~ $x>V_{max}$
⟶ ⟶ ⟶ ~alors~ $V_{max}:=x$
⟶ ⟶ ~FinSi~
⟶ ⟶ ~Si~ $x<V_{min}$
⟶ ⟶ ⟶ ~alors~ $V_{min}:=x$
⟶ ⟶ ~FinSi~
⟶ ~FinPour~
⟶ Afficher "$V_{min}$ et $V_{max}$"
~FinMin_Max~|
Suite du texte...
```

Voici un algorithme élémentaire :

Algorithme **MinMax**

```
1  macro Min_Max (#1) % afficher la valeur max et min d'une liste de valeurs #1
2      V_max ← −∞ % −∞ ou la plus petite valeur possible
3      V_min ← +∞ % +∞ ou la plus grande valeur possible
4      Pour chaque x dans (#1)
5          Si x > V_max
6              alors V_max ← x
7          FinSi
8          Si x < V_min
9              alors V_min ← x
10         FinSi
11     FinPour
12     Afficher "V_min et V_max"
13 FinMin_Max
```

Suite du texte...

Le problème étant posé, seuls les points n^{os} 1 et 8 vont faire intervenir des méthodes nouvelles.

Rendre actif le caractère de tabulation

Pour que le caractère de tabulation ait le comportement souhaité, la première idée serait de le rendre actif et de lui faire insérer horizontalement l'espace dont la mesure est contenue dans `\alogoindent` :

Code n° V-433

```
\def\algoindent{1cm}%
\begingroup
\parindent=0pt % pas d'indentation
\defactive\^^M{\leavevmode\par}% rend le retour charriot actif
\defactive\^^I{\hskip\algoindent\relax}% tabulation active ^^I
Pas d'indentation ici\ldots
⟶ Voici une première ligne
⟶ ⟶ une seconde plus indentée
⟶ retour à l'indentation normale
⟶ ⟶ \for\xx=1to10\do{un long texte }
⟶ pour terminer, la dernière ligne
```

```
\endgroup
```

Pas d'indentation ici…
Voici une première ligne
une seconde plus indentée
retour à l'indentation normale
un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte
pour terminer, la dernière ligne

Le comportement est celui souhaité sauf lorsque le texte est trop long pour loger sur la ligne entière, comme celui écrit par la boucle `\for`. Pour garder une cohérence dans l'alignement vertical, il faut que l'indentation soit rémanente. La méthode la plus immédiate est d'incrémenter le ressort `\leftskip` de la quantité `\algoindent` à chaque tabulation rencontrée et remettre ce ressort à `0pt` à chaque nouvelle ligne. Dans tout l'algorithme, nous assignerons un ressort infini à `\rightskip`.

Code n° V-434

```
\def\algoindent{1cm}%
\begingroup
\leftskip=0pt \rightskip=0pt plus1fil \relax % initialise les ressorts latéraux
\parindent=0pt % pas d'indentation
\defactive\^^M{\leavevmode\par\leftskip=0pt }% rend le retour charriot actif
\defactive\^^I{\advance\leftskip by \algoindent\relax}% tabulation active ^^I
Pas d'indentation ici\ldots
 ⟶ Voici une première ligne
 ⟶  ⟶ une seconde plus indentée
 ⟶ retour à l'indentation normale
 ⟶  ⟶ \for\xx=1to10\do{un long texte }
 ⟶ pour terminer, la dernière ligne
\endgroup
```

Pas d'indentation ici…
Voici une première ligne
une seconde plus indentée
retour à l'indentation normale
un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte un long texte
pour terminer, la dernière ligne

Numérotation des lignes

Passons maintenant au point n° 8 qui consiste à numéroter chaque ligne de l'algorithme. Une « ligne » de l'algorithme est ce qui se trouve entre deux frappes de la touche entrée ce qui revient à considérer ici qu'une ligne est donc un paragraphe pour TEX. La primitive `\everypar`, qui se comporte comme un registre de tokens, permet de définir un ensemble de tokens qui seront exécutés (en mode horizontal) lorsqu'un paragraphe débute :

`\everypar= {`⟨*texte*⟩`}`

Nous allons mettre cette primitive à contribution pour numéroter chaque paragraphe. Pour que le numéro soit dans la marge de gauche, nous allons le placer dans une boite en débordement à gauche de type `\llap` et laisser une espace de `4pt` entre ce numéro et le début de la ligne. Afin que les numéros soient correctement placés,

nous devons tenir compte du ressort \leftskip. Il faut donc laisser une espace de 4pt + \leftskip entre le numéro et le début de la ligne.

Code n° V-435

```
\newcount\algocnt % compteur de lignes pour l'algorithme
\def\algoindent{1cm}%
\begingroup
\algocnt=1 % initialise le compteur à 1
\leftskip=0pt \rightskip=0pt plus1fil\relax% initialise les ressorts à 0pt
\parindent=0pt % pas d'indentation
\everypar={\llap{$\scriptstyle\number\algocnt$\kern\dimexpr4pt+\leftskip\relax}}%

\defactive\^^M{\leavevmode\par\advance\algocnt by1 \leftskip=0pt }% retour charriot actif
\defactive\^^I{\advance\leftskip by \algoindent\relax}% tabulation active ^^I
Pas d'indentation ici\ldots
 ⟶ Voici une première ligne
 ⟶  ⟶ une seconde plus indentée
 ⟶ retour à l'indentation normale
 ⟶  ⟶ \for\xx=1to10\do{un long texte }
 ⟶ pour terminer, la dernière ligne
\endgroup
```

```
1 Pas d'indentation ici…
2         Voici une première ligne
3                 une seconde plus indentée
4         retour à l'indentation normale
5                 un long texte un long texte un long texte un long texte un long texte un long texte
                  un long texte un long texte un long texte un long texte
6         pour terminer, la dernière ligne
```

Ignorer les indésirables en début d'algorithme

Il nous reste une astuce à prévoir pour que la macro \algorithm soit la plus agréable à utiliser. Il s'agit de ce qui se trouve au début de l'algorithme, juste après le ⟨*token*⟩ qui délimite le texte de l'algorithme. Pour que les choses soient plus claires, supposons que ce ⟨*token*⟩ est « | ». Il faudrait que l'affichage soit le même que l'on écrive

1. `\algorithm{`⟨*titre*⟩`}|début de l'algorithme...|`
2. `\algorithm{`⟨*titre*⟩`}|`
   `début de l'algorithme...|`

Il faut donc que l'on supprime tous les retours charriots (actifs) et tous les espaces (actifs) qui pourraient se trouver au tout début du corps de l'algorithme. Cela suppose une macro non linéaire \sanitizealgo, la seule de cette section. En voici l'algorithme :

Manger tous les ^^M et

macro **lire_prochain_token**
    lire le token suivant x à l'aide de \futurelet
    aller à **tester_prochain_token**
fin

macro **tester_prochain_token**
    si (x = espace actif) **ou** (x = ^^M actif)
        manger x
        aller à **lire_prochain_token**
    finsi

fin

Nous appellerons cette macro *en dernier* puis, une fois son travail effectué, nous laisserons TEX composer le texte constituant l'algorithme. On ne pourra reprendre la main que lorsqu'il exécutera le ⟨*délimiteur*⟩ final rendu actif qui fermera le groupe ouvert tout au début, tracera la ligne horizontale de fin d'algorithme et sautera une petite espace verticale à l'aide de `\smallbreak`.

Commentaires grisés avec le package `color`

Enfin, pour que les commentaires soient en italique et en gris, nous allons rendre le caractère « % » actif de façon à ce qu'il sélectionne la fonte italique (avec `\it`) et colore le texte à venir en gris (avec `\color{gray}`). Cette dernière action suppose que la couleur « `gray` » a été définie et que le package « `color` » a préalablement été chargé moyennant :

- `\usepackage{color}` pour le format LATEX ;
- `\input color` pour le format plain-TEX ;
- `\setupcolor[state=start]` pour le format ConTEXt.

Pour que la typographie normale (caractères non italiques et couleur noire) soit à nouveau en vigueur à la ligne suivante, nous chargerons le caractère actif `^^M` d'effectuer les actions « `\rm` » pour un retour en fonte droite et « `\color{black}` » pour restaurer la couleur noire.

Le code complet

Voici le code complet commenté qui reprend des méthodes déjà vues plusieurs fois (programmation de caractères actifs notamment) :

Code n° V-436

```
\newif\ifnumalgo
\newcount\algocnt
\numalgotrue

\def\algoindent{2em}
\def\algorule{.4pt}% épaisseur des réglures de début et de fin

\def\algohrulefill{% remplit avec une ligne horizontale
  \leavevmode
  \leaders\hrule height\algorule\hfill
  \kern0pt % rajoute un noeud au cas où la commande est en fin de ligne
}

\newmacro\algorithm[\\,\#,\{,\}]2{%
  \medbreak
  \begingroup% ouvre un groupe (sera fermé à la fin de l'algorithme)
    \algocnt=1 % initialise le compteur de lignes
    \leftskip=0pt \rightskip=0pt plus1fil\relax% initialise les ressorts latéraux
    \parindent=0pt % pas d'indentation
    %%%%%%%%%%% affichage du titre %%%%%%%
    \algohrulefill% remplir avec une ligne horizontale
    \ifempty{#2}% si #2 est vide
      {}% ne rien insérer
      {% sinon
      \lower.5ex\hbox{ #2 }% insérer le titre abaissé de 0.5ex
```

```
      \algohrulefill% insérer une ligne horizontale
      }%
    \par% aller à la ligne
    \nointerlineskip% ne pas insérer le ressort d'interligne
    \kern7pt % et sauter 7pt verticalement
    \nobreak% empêcher une coupure de page
    %%%%%%%%%%%%%% fin du titre %%%%%%%%%%
    %
    %%%%%%%%%%%%%% rend les caractères actifs %%%%%%%%%%%%%%
    \def~##1~{\begingroup\bf##1\endgroup}%
    \defactive:{% rend ":" actif
      \futurelet\nxttok\algoassign% \nxttok = token suivant
    }%
    \def\algoassign{% suite du code de ":"
      \ifx=\nxttok% si ":" est suivi de "="
        \ifmmode% si mode math
          \leftarrow% afficher "\leftarrow"
        \else% si mode texte
          ${}\leftarrow{}$% passer en mode math pour "\leftarrow"
        \fi
        \expandafter\gobone% manger le signe "="
      \else% si ":" n'est pas suivi de "="'
        \string:% afficher ":"
      \fi
    }%
    \ifnumalgo% si la numérotation est demandée,
      \everypar={% définir le \everypar
        \llap{$\scriptstyle\number\algocnt$\kern\dimexpr4pt+\leftskip\relax}%
      }%
    \fi
    \doforeach\currentchar\in{#1}{\catcode\expandafter`\currentchar=12 }%
    \def\algocr{% définit ce que fait ^^M
      \color{black}% repasse en couleur noire
      \rm% fonte droite
      \leavevmode\par% termine le paragraphe
      \advance\algocnt by1 % incrémente le compteur de lignes
      \leftskip=0pt % initialise le ressort gauche
    }
    \letactive\^^M=\algocr%
    \defactive\^^I{\advance\leftskip by \algoindent\relax}% ^^I
    \defactive\ {\hskip1.25\fontdimen2\font\relax}% espace = 25% de + que
                                                  % la largeur naturelle
    \defactive\%{\it\color{gray}\char`\%}% passe en italique et gris puis affiche "%"
    \defactive_{\ifmmode_\else\string_\fi}%
    \defactive#3{% rend le token #3 actif (il sera rencontré à la fin de l'algorithme)
      \everypar{}% neutralise le \everypar
      \par% aller à la ligne
      \nointerlineskip% ne pas insérer le ressort d'interligne
      \kern7pt % et sauter 7pt verticalement
      \nobreak% empêcher une coupure de page
      \algohrulefill% tracer la ligne de fin
  \endgroup% ferme le groupe ouvert au début de l'algorithme
  \smallbreak% saute un petit espace vertical
    }%
  %%%%%%%%%%%%%%%% fin des caractères actifs %%%%%%%%%%%%%%%%
  \sanitizealgo% va manger les espaces et les ^^M au début de l'algo
}

\def\sanitizealgo{\futurelet\nxttok\checkfirsttok}% récupère le prochain token

```

```
\def\checkfirsttok{% teste le prochaun token
  \def\nextaction{% a priori, on considère que la suite est " " ou "^^M" donc
    \afterassignment\sanitizealgo% aller à \sanitizealgo
    \let\nexttok= % après avoir mangé ce "^^M" ou cet espace
  }%
  \unless\ifx\nxttok\algocr% si le prochain token n'est pas un ^^M
    \unless\ifx\space\nxttok% et si le prochain token n'est pas un espace
      \let\nextaction\relax% ne rien faire ensuite
    \fi
  \fi
  \nextaction% faire l'action décidée ci-dessus
}

Voici un algorithme élémentaire
\algorithm[\#]{Algorithme {\bf MinMax}}|
macro ~Min_Max~ (#1) % afficher la valeur max et min d'une liste de valeurs #1
 ⟶ $V_{max}:=-\infty$ % $-\infty$ ou la plus petite valeur possible
 ⟶ $V_{min}:=+\infty$ % $+\infty$ ou la plus grande valeur possible
 ⟶ ~Pour~ chaque $x$ dans (#1)
 ⟶  ⟶ ~Si~ $x>V_{max}$
 ⟶  ⟶  ⟶ ~alors~ $V_{max}:=x$
 ⟶  ⟶ ~FinSi~
 ⟶  ⟶ ~Si~ $x<V_{min}$
 ⟶  ⟶  ⟶ ~alors~ $V_{min}:=x$
 ⟶  ⟶ ~FinSi~
 ⟶ ~FinPour~
 ⟶ Afficher "$V_{min}$ et $V_{max}$"
~FinMin_Max~|
Suite du texte...
```

Voici un algorithme élémentaire

Algorithme **MinMax**

```
1  macro Min_Max (#1) % afficher la valeur max et min d'une liste de valeurs #1
2      V_max ← −∞ % −∞ ou la plus petite valeur possible
3      V_min ← +∞ % +∞ ou la plus grande valeur possible
4      Pour chaque x dans (#1)
5          Si x > V_max
6              alors V_max ← x
7          FinSi
8          Si x < V_min
9              alors V_min ← x
10         FinSi
11     FinPour
12     Afficher "V_min et V_max"
13 FinMin_Max
```

Suite du texte...

L'algorithme obtenu peut franchir des sauts de page et donc s'étendre sur plusieurs pages. Ce comportement peut être facilement interdit en englobant la totalité de l'algorithme dans une boite verticale de type `\vbox` ou `\vtop`. On peut par exemple remplacer le `\begingroup` de la ligne n° 16 par `\vbox \bgroup` qui marquera le début de la boite et en profiter, si on le souhaite pour ajuster la largeur `\hsize`. Il faudra également remplacer le `\endgroup` de la ligne n° 77 par `\egroup` qui fermera la boite.

Conclusion

Nous voilà, cher lecteur, arrivés à la fin de notre périple. Nous en savons beaucoup plus qu'au début, mais il reste tant à découvrir, tant à expérimenter… Le maitre mot est d'essayer de coder par soi même ; c'est un poncif tant c'est répété, mais c'est en programmant qu'on devient programmeur. Pas en lisant un livre sur le sujet.

Si les notions abordées dans ce livre ont balayé le plus large champ possible pour permettre un début d'autonomie, elles ne sont pas exhaustives. Par exemple, nous n'avons jamais parlé de la routine de sortie qui occupe à elle seule un chapitre entier (le n° 23) du TEXbook. De la même façon, pas un mot n'a été dit sur les insertions, mécanisme souple et puissant utilisé dans les notes de bas de page ou les notes marginales. Nous avons à peine effleuré les alignements et les tableaux de TEX alors que ce sujet est d'une grande richesse. Il y reste donc des pans entiers à découvrir…

Bonne chance donc et avant tout, cher lecteur, prenez du plaisir avec TEX !

Sixième partie

Annexes

Sommaire

Chapitre 1

Débogage

Le débogage est un art difficile qui consiste à trouver pour quelle(s) raison(s) un programme ne fonctionne pas comme on s'y attend. Certes, la relecture du code accompagnée de la simulation mentale de son fonctionnement permet parfois de trouver où se trouve l'erreur. Bien souvent hélas, cet exercice est insuffisant pour débusquer l'endroit où se cache le « *bug* » . Il faut alors s'en remettre à des instructions de débogage.

1.1. Délivrer les informations stockées

Nous allons d'abord examiner les instructions qui font remonter des entrailles de TEX ce qui est contenu dans les différentes structures de données permettant de stocker de l'information (primitives, macros ou registres). Nous avons déjà vu que `\meaning`⟨*token*⟩ se développait en tokens de catcode 12 qui, selon ce qu'est le ⟨*token*⟩ :

- est le nom de la primitive si ce ⟨*token*⟩ est une primitive ;
- est le texte de paramètre et le texte de remplacement dans le cas d'une macro ;
- est le ⟨*token*⟩ lui-même accompagné de sa nature si c'est un caractère.

Code n° VI-437

```
a) \meaning a\par
b) \meaning {\par
c) \meaning _\par
d) \meaning\def\par % une primitive
e) \meaning\baselineskip\par% une primitive
f) \long\def\foo#1#2.#3{#1 puis #2 et #3 !}\meaning\foo
```

```
a) the letter a
b) begin-group character {
c) subscript character _
d) \def
e) \baselineskip
f) \long macro:#1#2.#3->#1 puis #2 et #3 !
```

La primitive \show a un comportement identique sauf que ces informations sont écrites dans le fichier log et sur le terminal. Cela présente l'avantage de ne pas surcharger l'affichage comme peut le faire \meaning, à condition d'aller lire le fichier log après la compilation.

Pour connaitre l'argument d'une macro, \show manque de souplesse puisqu'il faut définir une macro auxiliaire dont le texte de remplacement est cet argument et enfin, donner cette macro à \show :

Code n° VI-438

```
\def\foo#1#2{%
  \def\argA{#1}\show\argA% débogage
  Bonjour #1 et #2}
\foo{{\bf X}}{Y}
```

```
> \argA=macro:
->{\bf X}.
```

Le moteur ε-TEX fournit la primitive

\showtokens{⟨*ensemble de tokens*⟩}

et écrit sur le terminal et dans le fichier log cet ⟨*ensemble de tokens*⟩. De plus, cette primitive devant être suivie d'une accolade ouvrante, elle obéit à la règle que partagent les primitives possédant cette propriété : elle développe tout jusqu'à trouver cette accolade ouvrante.

Code n° VI-439

```
\def\foo#1#2{%
  \showtokens{#1}% débogage
  Bonjour #1 et #2}
\foo{{\bf X}}{Y}
```

```
>{\bf X}.
```

La primitive \show reste pourtant inopérante pour afficher quel est le contenu d'un registre, qu'il soit primitive (comme \baselineskip) ou défini par l'utilisateur (comme un registre d'entier par exemple).

Code n° VI-440

```
\newtoks\footoks
\footoks={foo bar}
\show\footoks
```

```
> \footoks=\toks62.
```

Pour extraire les valeurs contenues dans des registres de dimension, de ressort, d'entier ou de tokens, la primitive \showthe doit être utilisée :

Code nº VI-441

```
\showthe\baselineskip% registre-primitive
\newtoks\footoks \footoks={foo bar} \showthe\footoks
\newcount\foocount \foocount=987 \showthe\foocount
\newdimen\foodimen \foodimen=3.14pt \showthe\foodimen
\newskip\fooskip \fooskip=10 pt plus 1pt minus 1fil \showthe\fooskip
\newbox\foobox \foobox\hbox{foo} \showthe\foobox
```

```
> 12.0pt.
> foo bar.
> 987.
> 3.14pt.
> 10.0pt plus 1.0pt minus 1.0fil.
> 101.
```

Le dernier registre, celui de boite, échappe à \showthe. Cela est dû au fait que \newbox définit \foobox à l'aide de \chardef puisqu'il n'existe pas de primitive \boxdef. Nous avons vu que pour demander à TEX de donner la composition d'une boite, il faut passer par la primitive \showbox (voir page 489).

Correctement placer les instructions de débogage telles que \show, \showthe et \showtokens afin de se donner le maximum de chances de trouver le dysfonctionnement n'est pas trivial et demande souvent plusieurs essais-erreurs. Bien souvent, l'enquête consiste à suivre l'évolution du texte de remplacement d'une macro (avec \show), comprendre comment se transmet un argument entre plusieurs macros (avec \showtokens) ou « *tracer* » la valeur d'un registre (avec \showthe).

1.2. Délivrer des informations sur l'exécution

TEX fournit également des primitives qui permettent de contrôler quelles informations seront affichées dans le fichier log lors de l'exécution du code. Une fois la compilation faite, l'examen de ces informations retrace, selon la précision demandée, ce que TEX a fait.

La primitive \tracingmacros se comporte comme un registre d'entier. Lorsqu'elle reçoit un entier strictement positif, TEX donnera dans le fichier log des informations sur chaque macro développée : son nom, son texte de paramètre, son texte de remplacement et si elle admet des arguments, la valeur des arguments lus au moment du développement.

Code nº VI-442

```
\def\foo#1,#2\endfoo{Bonjour #1 et #2}
\tracingmacros=1
\foo moi,toi\endfoo
\tracingmacros=0
```

```
\foo #1,#2\endfoo ->Bonjour #1 et #2
#1<-moi
#2<-toi
```

Une autre aide au diagnostic est fournie par la primitive \tracingcommands, qui est également de type entier. En assignant un nombre strictement positif à cette primitive, TEX écrira dans le fichier log toutes les commandes qu'il exécute. Si l'entier vaut 2 ou plus, les tests et leurs issues seront également montrés.

Code nº VI-443

```
\tracingcommands=2
Foo \hbox{et \ifnum2>1 bar\else toi\fi}\par
Suite
\tracingcommands=0
```

```
{vertical mode: the letter F}
{horizontal mode: the letter F}
{blank space }
{\hbox}
{restricted horizontal mode: the letter e}
{blank space }
{\ifnum}
{true}
{the letter b}
{\else}
{end-group character }}
{horizontal mode: \par}
{vertical mode: the letter S}
{horizontal mode: the letter S}
{blank space }
{\tracingcommands}
```

Les espaces parasites générés par des fins de ligne non commentées peuvent facilement être débusqués en cherchant « `blank space` » dans le fichier `log`.

Pour déboguer des programmes, la macro `\tracingrestores` peut également s'avérer utile. Elle est également de type entier et indique dans le fichier `log` ce que TEX garde en mémoire lorsque des assignations sont locales. Ces instructions placées dans la *pile de sauvegarde* sont dépilées à la sortie du groupe pour restaurer la variable à son état antérieur. Si une assignation est globale, rien n'est placé dans la pile et la dernière valeur globale assignée sera celle en vigueur à la sortie du groupe

Prenons un exemple inspiré de celui du TEXbook. Afin d'éviter de définir un compteur, utilisons celui qui porte le nº 255 et définissons la macro `\i` qui incrémente ce compteur.

Code nº VI-444

```
\def\i{\advance\count255 1 }
\tracingrestores=1
\count255=2 {\i\i\i}
\tracingrestores=0
```

```
{restoring \count255=2}
```

L'instruction « `{restoring \count255=2}` » reflète ce qui exécuté à la fin du groupe. L'instruction « `\count255=2` » a été placée sur la pile de sauvegarde lorsque le premier `\i` a été exécuté.

Lorsqu'une assignation est globale, la dernière valeur globale assignée devient celle qui sera en vigueur à la sortie du groupe. Dès lors, les précédentes instructions de sauvegarde placées sur la pile de sauvegarde deviennent inutiles, mais elles y restent et seront ignorées à la sortie du groupe. Mettons ceci en évidence avec la macro `\gi` qui incrémente globalement le compteur nº 255 :

Code n° VI-445

```
\def\i{\advance\count255 1 } \def\gi{\global\i}
\tracingrestores=1
\count255=2 {\i\i\gi\i\gi\i}
\tracingrestores=0
```

```
{restoring \count255=7}
{retaining \count255=7}
{retaining \count255=7}
```

Les empilements sur la pile de sauvegarde ont lieu :

1. lorsque le premier `\i` est exécuté ;
2. au troisième `\i`, qui modifie localement une valeur rendue globale par le `\gi` précédent ;
3. au dernier `\i` qui à nouveau, modifie localement une valeur globale.

Au dépilement qui se fait dans l'*ordre inverse* de l'empilement, seule la dernière valeur de sauvegarde 7 est prise en compte. Les deux derniers ordres de sauvegarde sont ignorés et la valeur 7 est retenue. Cela se traduit dans le fichier `log` par lignes contenant le mot « `retaining` » et non pas « `restoring` ».

Il faut donc être conscient qu'effectuer localement des assignations locales *et* globales peut charger la pile de sauvegarde inutilement. Dans les faits, une instruction de sauvegarde est empilée chaque fois qu'une assignation locale suit une assignation globale. La pile de sauvegarde n'est évidemment pas infinie et un dépassement de sa capacité se traduit par une erreur à la compilation de type « `save size capacity exceeded.` »

Chapitre 2

Programmer l'addition décimale

Comme nous l'avons vu et expliqué à la page 235, l'addition sur les dimensions pose des problèmes d'arrondis conduisant à des résultats faux. Ces erreurs, insignifiantes pour une dimension, sont inacceptables lorsque ces nombres sont considérés comme entités mathématiques. Il est donc utile de programmer une macro purement développable `\decadd{`⟨*décimal 1*⟩`}{`⟨*décimal 2*⟩`}` qui ne présente pas les défauts de l'addition sur les dimensions.

Ayant à notre disposition l'addition des entiers relatifs, programmer une simple addition de deux nombres décimaux relatifs semble trivial. Pourtant…

2.1. Procédure mathématique

Appelons a et b les deux nombres à additionner. Si les deux nombres sont entiers, le problème est résolu : il suffit de les additionner avec `\numexpr` qui ne procède à aucun arrondi.

Si l'un d'entre eux est entier, il faut d'abord le transformer en nombre décimal en ajoutant « `.0` ». Ceci fait, les deux nombres a et b sont de la forme « $x.y$ » où x est un entier relatif et y un entier naturel. Ensuite, nous allons transmettre le signe de x à y de telle sorte que l'entier représentant la partie entière et celui représentant la partie décimale aient le même signe. Par exemple, si $a = -3.14$, alors $x_a = -3$ et $y_a = -14$.

L'addition se fait donc entre deux nombres de la forme $x_a.y_a$ et $x_b.y_b$ où x_a, x_b, y_a et y_b sont des entiers *relatifs*. Nous allons transformer les parties décimales pour qu'elles aient le même nombre de chiffres en rajoutant à l'une d'entre-elles, si c'est

nécessaire, des 0 inutiles à sa droite. Une fois ceci effectué, on définit 10^n comme la puissance de 10 immédiatement supérieure à $|y_a|$ ou $|y_b|$ que l'on nomme « seuil de retenue ».

La méthode consiste ensuite à ajouter indépendamment les parties entières et les parties décimales. Pour que les choses soient plus claires, les parties entières et décimales sont séparées par une ligne verticale :

$$\begin{array}{rcl} & x_a & y_a \\ + & x_b & y_b \\ \hline & x & y \end{array}$$

On examine ensuite les nombres relatifs x et y obtenus. Si l'on note σ_x le signe de x (ou celui de y si x est nul) et σ_y celui de y, alors, le cas est favorable se produit lorsque $|y| < 10^n$ (pas de retenue) et $\sigma_x = \sigma_y$. Le résultat est alors le nombre décimal

$$\begin{cases} x.|y| & \text{si } x \neq 0 \\ \sigma_y 0.|y| & \text{si } x = 0 \end{cases}$$

Deux cas défavorables peuvent se présenter :

1. x et y sont de signes contraires. Ces additions en sont des illustrations :

$$\begin{array}{rcl} & 7 & 20 \\ + & -4 & -45 \\ \hline & 3 & -25 \end{array} \qquad \begin{array}{rcl} & 2 & 70 \\ + & -4 & -15 \\ \hline & -2 & 55 \end{array}$$

   Dans ce cas, afin que x et y satisfassent les conditions du cas favorable, les modifications suivantes doivent y être apportées :

$$\begin{cases} x \longleftarrow x - (\sigma_x 1) \\ y \longleftarrow y - (\sigma_y 10^n) \end{cases}$$

   Les additions précédentes deviennent :

$$\begin{array}{rcl} & 7 & 20 \\ + & -4 & -45 \\ \hline & 3 & -25 \\ & 3-(1) & -25-(-100) \\ = & 2 & 75 \end{array} \qquad \begin{array}{rcl} & 2 & 70 \\ + & -4 & -15 \\ \hline & -2 & 55 \\ & -2-(-1) & 55-(100) \\ = & -1 & -45 \end{array}$$

2. x et y sont de même signe *et* $|y| \geqslant 10^n$, comme dans ces additions :

$$\begin{array}{rcl} & 7 & 20 \\ + & 4 & 95 \\ \hline & 11 & 115 \end{array} \qquad \begin{array}{rcl} & -2 & -70 \\ + & -4 & -65 \\ \hline & -6 & -135 \end{array}$$

   Pour que x et y vérifient les conditions du cas favorable, leurs valeurs doivent être modifiées selon la règle suivante :

$$\begin{cases} x \longleftarrow x + (\sigma_x 1) \\ y \longleftarrow y - (\sigma_y 10^n) \end{cases}$$

   Les deux additions précédentes deviennent alors :

	7	20			−2	−70
+	4	95		+	−4	−65
	11	115			−6	−135
	11 + (1)	115 − (100)			−6 + (−1)	−135 − (−100)
=	12	15		=	−7	−35

2.2. Mise en œuvre

2.2.1. Rendre les nombres décimaux

Dans un premier temps, la macro \decadd doit s'assurer que les parties décimales sont bien présentes et le cas échéant, ajouter « .0 » comme partie décimale. Une fois ceci fait, elle passera la main à une macro auxiliaire \decadd@i, à arguments délimités. La macro \ifnodecpart, utilisée par \formatnum, sera chargée de tester si le point est présent ou pas.

Code n° VI-446

```
\catcode`@11
\def\ifnodecpart#1{\ifnodecpart@i#1.\@nil}
\def\ifnodecpart@i#1.#2\@nil{\ifempty{#2}}
\def\decadd#1#2{% #1 et #2=nombre à additionner
  \ifnodecpart{#1}% si #1 est un entier
    {\ifnodecpart{#2}% et #2 aussi, les additionner avec \numexpr
      {\number\numexpr#1+#2\relax}%
      {\decadd@i#1.0\@nil#2\@nil}% sinon, ajouter ".0" après #1
    }
    {\ifnodecpart{#2}% si #1 a une partie entière mais pas #2
      {\decadd@i#1\@nil#2.0\@nil}% ajouter ".0" à #2
      {\decadd@i#1\@nil#2\@nil}% sinon, les 2 parties entières sont présentes
    }%
}
\def\decadd@i#1.#2\@nil#3.#4\@nil{% affiche les parties entières et décimales reçues
  $x_a=#1\quad y_a=#2\qquad x_b=#3\quad y_b=#4$
}
\catcode`@12
a) \decadd{5}{9.4}\par
b) \decadd{-3.198}{-6.02}\par
c) \decadd{0.123}{123}
```

a) $x_a = 5 \quad y_a = 0 \qquad x_b = 9 \quad y_b = 4$
b) $x_a = -3 \quad y_a = 198 \qquad x_b = -6 \quad y_b = 02$
c) $x_a = 0 \quad y_a = 123 \qquad x_b = 123 \quad y_b = 0$

2.2.2. Rendre les parties décimales de même longueur

Venons-en maintenant à la partie la plus complexe. Nous allons programmer la \addzeros* qui va se charger d'ajouter des 0 inutiles à la fin d'un des deux nombres #2 ou #4 reçus par \decadd@i pour que ces parties décimales aient le même nombre de chiffres. Afin de rester purement développable, cette macro va utiliser le système des « arguments réservoirs ». Son texte de paramètre sera

```
#1#2/#3.#4#5/#6.#7/
```

où :

– #1 est le premier chiffre de y_a et #2 les chiffres restants ;

– #3 est le réservoir contenant les chiffres de y_a déjà examinés ;
– #4, #5 et #6 jouent le même rôle pour y_b que #1, #2 et #3 pour y_a ;
– #7 est un réservoir recevant le seuil de retenue : il contient 1 au début et un « 0 » est ajouté à chaque itération.

L'idée directrice est que le point d'arrêt se produit lorsque les deux arguments délimités #2 et #5 sont vides. Dans le cas contraire, à chaque itération, #1 passe en fin de #3, #4 passe en fin de #6 et un 0 est ajouté à #7. Si un des arguments #2 ou #5 devient vide, il faut alimenter l'argument non délimité #1 ou #4 qui le précède avec 0.

Pour se représenter la situation, imaginons que les nombres $y_a = 457$ et $y_b = 689714$ soient traités par \addzeros et que l'appel suivant ait été fait :

```
\addzeros457/.689714/.1
```

Le chiffre 4 et le chiffre 6 vont être transférés dans les réservoirs et un 0 sera ajouté après le 1. L'appel récursif après la première itération sera donc

```
\addzeros57/4.89714/6.10
```

Puis le processus identique sera effectué et l'appel suivant sera

```
\addzeros7/45.9714/68.100
```

Comme il ne reste plus aucun chiffre après le 7 (ce qui signifie que l'argument délimité #2 est vide), le 7 sera transféré dans le réservoir, mais un 0 sera mis à sa place pour les appels suivants. Ces appels seront donc :

```
\addzeros0/457.714/689.1000
\addzeros0/4570.14/6897.10000
\addzeros0/45700.4/68971.100000
```

Enfin, comme il ne reste plus de chiffre ni après y_a ni après y_b, le point d'arrêt est atteint : #2 et #5 sont vides. La macro transfère une dernière fois un 0 en fin de réservoir #3, transfère aussi le 4 en fin de réservoir #5 et ajoute un dernier 0 au dernier argument pour mettre dans le flux de lecture de TEX les trois arguments suivants :

```
{457000}{689714}{1000000}
```

Pour comprendre le cheminement des arguments, la macro affiche ici ce qu'elle fait :

Code nº VI-447

```
\def\addzeros#1#2/#3.#4#5/#6.#7/{%
  Arguments reçus : #1#2/#3.#4#5/#6.#7/\par% afficher ce que la macro reçoit
  \ifempty{#2}% si #1 est le dernier chiffre de y
    {\ifempty{#5}% et si #4 est le dernier chiffre de y2
      {Résultat final : \detokenize{{#3#1}{#6#4}{#70}}}% afficher le résultat final
      {\addzeros0/#3#1.#5/#6#4.#70/}% sinon alimenter #1 avec un 0
    }
    {\ifempty{#5}% si #4 est le dernier chiffre de y2
      {\addzeros#2/#3#1.0/#6#4.#70/}% alimenter #4 avec un 0
      {\addzeros#2/#3#1.#5/#6#4.#70/}% #2 et #5 non vides
    }%
}
\addzeros457/.689714/.1/
```

```
Arguments reçus : 457/.689714/.1/
Arguments reçus : 57/4.89714/6.10/
Arguments reçus : 7/45.9714/68.100/
Arguments reçus : 0/457.714/689.1000/
Arguments reçus : 0/4570.14/6897.10000/
Arguments reçus : 0/45700.4/68971.100000/
Résultat final : {457000}{689714}{1000000}
```

2.2.3. Additionner séparément

Intéressons-nous tout d'abord à la macro qui va donner le signe du nombre décimal. Si x est sa partie entière, la macro `\sgn`, vue précédemment ne va pas convenir. Voici quel était son code :

```
\def\sgn#1{\ifnum#1<0 -\fi}
```

Par exemple, si le décimal à examiner vaut -0.123, alors, la partie entière $x = -0$ et `\sgn{-0}` ne revoit pas un signe négatif. Pour se prémunir de cette erreur, la macro `\true@sgn`* que nous allons employer renverra un signe « – » lorsque son argument commence par « – ». Pour ce faire, elle se contente de tester le signe de son argument une fois lui avoir ajouté le chiffre 1 en dernière position :

```
\def\true@sgn#1{\ifnum#11<\z@-\fi}
```

Il faut rester conscient qu'ajouter le chiffre 1 au nombre `#1` et évaluer le tout avec `\ifnum` n'est pas anodin. Le résultat évalué par `\ifnum`, qui est `#1` $\times\, 10 \pm 1$, doit être dans l'intervalle $I = [-2^{31} + 1\ ;\ 2^{31} - 1]$. Par conséquent, l'intervalle dans lequel doit se trouver `#1` est plus étroit que I. Même si cet écueil n'est pas vraiment gênant, on aurait pu s'en prémunir en testant si `#1` commence par « – ».

La macro `\decadd@i` va utiliser le fait que `\romannumeral-'\@`, placé avant la macro `\addzeros`, lance le développement maximal pour obtenir les trois arguments finaux. Elle appellera donc une nouvelle macro `\decadd@ii`, admettant ces trois arguments auxquels nous rajouterons, en 4$^{\text{e}}$ et 5$^{\text{e}}$ arguments, les parties entières. La macro `\decadd@ii`, qui reçoit ces 5 arguments, se contentera d'additionner les parties décimales entre elles (après avoir leur avoir ajouté le signe des parties entières) ainsi que les parties entières entre elles et transmettra le tout à la macro suivante `\decadd@iii` :

Code n° VI-448

```
\catcode`\@11
\def\addzeros#1#2/#3.#4#5/#6.#7/{%
  \ifempty{#2}% si #1 est le dernier chiffre de y1
    {\ifempty{#5}% et si #4 est le dernier chiffre de y2
      {{#3#1}{#6#4}{#70}}% redonner les 3 arguments
      {\addzeros0/#3#1.#5/#6#4.#70/}% sinon alimenter #1 avec un 0
    }
    {\ifempty{#5}% si #4 est le dernier chiffre de y2
      {\addzeros#2/#3#1.0/#6#4.#70/}% alimenter #4 avec un 0
      {\addzeros#2/#3#1.#5/#6#4.#70/}% #2 et #5 non vides
    }%
}
\def\decadd#1#2{% #1 et #2=nombre à additionner
  \ifnodecpart{#1}
    {\ifnodecpart{#2}{\number\numexpr#1+#2\relax}{\decadd@i#1.0\@nil#2\@nil}}
    {\ifnodecpart{#2}{\decadd@i#1\@nil#2.0\@nil}{\decadd@i#1\@nil#2\@nil}}%
```

```
}
\def\decadd@i#1.#2\@nil#3.#4\@nil{%
  \expandafter\decadd@ii
      \romannumeral-`\0\addzeros#2/.#4/.1/% se développe en 3 arguments
      {#1}
      {#3}%
}
\def\decadd@ii#1#2#3#4#5{%
% #1 et #2=parties décimales (mêmes longueurs);
% #3=seuil de retenue; #4 et #5=parties entières
  \exptwoargs{\decadd@iii{#3}}% envoyer le seuil de retenue tel quel
    % sommer les parties décimales signées
    {\number\numexpr\true@sgn{#4}#1+\true@sgn{#5}#2\relax}%
    % et les parties entières
    {\number\numexpr#4+#5\relax}%
}
\def\decadd@iii#1#2#3{%
  seuil de retenue = #1 \qquad nombre = "#3"."#2"%
}
\catcode`\@12
a) \decadd{6.7}{3.498}\par
b) \decadd{1.67}{-4.9}\par
c) \decadd{3.95}{2.0005}\par
d) \decadd{1.007}{2.008}\par
e) \decadd{-7.123}{3.523}
```

a) seuil de retenue = 1000 nombre = "9"."1198"
b) seuil de retenue = 100 nombre = "-3"."-23"
c) seuil de retenue = 10000 nombre = "5"."9505"
d) seuil de retenue = 1000 nombre = "3"."15"
e) seuil de retenue = 1000 nombre = "-4"."400"

2.2.4. Correctement gérer les 0

Les deux derniers cas montrent que les 0 au début de la partie décimale sont supprimés par le `\number` de la ligne n° 29 tandis que les 0 inutiles de la fin sont conservés. C'est exactement le comportement inverse de celui qui est souhaité ! Afin de rétablir une partie décimale ayant des 0 là où il faut, nous devons écrire une macro

`\format@dec{`⟨*partie décimale*⟩`}{`⟨*seuil de retenue*⟩`}`

qui est purement développable et qui supprime les 0 de droite tout en ajoutant les 0 de gauche.

Si par exemple, on obtient une partie décimale égale à 710 avec un seuil de retenue de 100000, la partie décimale avait 5 chiffres, c'est-à-dire 00710 qui se doit être transformée en 0071. Pour parvenir à ses fins, la macro `\format@dec` va d'abord ajouter le seuil de retenue à la valeur absolue de la partie décimale :

$$710 + 100000 = 100710$$

Elle va ensuite inverser le résultat avec `\reverse`

017001

Puis soumettre le nombre obtenu à \number ce qui aura pour effet de supprimer les 0 de début (qui étaient ceux de fin)

17001

Ensuite, il faut à nouveau inverser le résultat pour rétablir l'ordre de chiffres

10071

Enfin, il reste à manger le "1" en première position avec \gobone pour obtenir 0071. L'ordre des opérations est critique :

Code n° VI-449

```
\catcode`\@11
\def\format@dec#1#2{% #1=partie décimale #2=seuil de retenue
  \expandafter\gobone% le \gobone agira en dernier,
    \romannumeral-`\0% mais avant, tout développer
    \expandafter\reverse\expandafter% et retarder le \reverse de fin
      % pour développer son argument qui
      {\number% développe tout et évalue avec \number
        \expandafter\reverse\expandafter% l'inversion de
        {\number\numexpr\abs{#1}+#2\relax}% |#1|+#2
      }%
}
a) \format@dec{710}{100000}\par
b) \format@dec{6}{100}\par
c) \format@dec{-12300}{1000000}
\catcode`\@12
```

a) 0071
b) 06
c) 0123

2.2.5. Calculer le résultat

Reprenons notre chemin dans le texte de remplacement de \decadd@iii. Elle doit effectuer les éventuelles modifications sur les parties entières et décimales selon que x et y sont de signes contraires et sinon, effectuer d'autres modifications si $|y| \geqslant 10^n$. Ces tests ne présentent pas de difficulté et seront faits avec \ifnum. À la toute fin, la macro \decadd@iv reçoit 3 arguments de \decadd@iii, la partie entière, la partie décimale et le seuil de retenue. Cette macro se chargera d'afficher #1, et si #2 est différent de 0, le point décimal ainsi que la partie décimale formatée avec \format@dec. Pour que le 2-développement de \decadd donne le résultat, le développement maximal sera engagé via \romannumeral début de la macro. Comme rien n'est dirigé vers l'affichage jusqu'à \decadd@iv, ce développement sera en vigueur à l'entrée de cette macro. Il faut ensuite le propager pour qu'il atteigne \format@dec avant que la partie entière ne soit affichée.

Voici le code complet :

Code n° VI-450

```
\catcode`@11
\def\true@sgn#1{\ifnum#11<\z@-\fi}
\def\decadd#1#2{% #1 et #2=nombre à additionner
   \romannumeral-`\.% tout développer jusqu'à l'affichage du nombre (\decadd@iv)
  \ifnodecpart{#1}% si #1 est un entier\ifnodecpart
```

```
    {\ifnodecpart{#2}% et #2 aussi, les additionner avec \numexpr
      {\numexpr#1+#2\relax}%
      {\decadd@i#1.0\@nil#2\@nil}% sinon, ajouter ".0" après #1
    }
    {\ifnodecpart{#2}% si #1 a une partie entière mais pas #2
      {\decadd@i#1\@nil#2.0\@nil}% ajouter ".0" à #2
      {\decadd@i#1\@nil#2\@nil}% sinon, les 2 parties entières sont présentes
    }%
}
\def\decadd@i#1.#2\@nil#3.#4\@nil{%
  \expandafter\decadd@ii
      \romannumeral-`\0\addzeros#2/.#4/.10/% se développe en 3 arguments
      {#1}
      {#3}%
}
\def\decadd@ii#1#2#3#4#5{%
% #1 et #2=parties décimales (mêmes longueurs)
% #3=seuil de retenue; #4 et #5=parties entières
  \exptwoargs{\decadd@iii{#3}}% envoyer le seuil de retenue tel quel
    % sommer les parties décimales signées
    {\number\numexpr\true@sgn{#4}#1+\true@sgn{#5}#2\relax}%
    % et les parties entières
    {\number\numexpr#4+#5\relax}%
}
\def\decadd@iii#1#2#3{% #1=seuil de retenue #2=partie décimale #3= partie entière
  \ifnum\true@sgn{#2}\true@sgn{\ifnum#3=\z@#2\else#3\fi}1=-1 % si les signes sont
                                                            % différents
  \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
    {\exptwoargs\decadd@iv% transmettre les arguments modifiés :
      {\number\numexpr#3-\true@sgn{#3}1}% #3:=#3-sgn(#3)1
      {\number\numexpr#2-\true@sgn{#2}#1}% #2:=#2-sgn(#2)10^n
      {#1}%
    }% si les signes sont égaux
    {\ifnum\abs{#2}<#1 % et si abs(y)<10^n
    \expandafter\firstoftwo\else\expandafter\secondoftwo\fi
      {\decadd@iv{#3}{#2}{#1}% tranmettre les arguments tels quels
      }
      {\exptwoargs\decadd@iv% sinon
        {\number\numexpr#3+\true@sgn{#3}1}% #3:=#3+sgn(#3)1
        {\number\numexpr#2-\true@sgn{#2}#1}% #2:=#2-sgn(#2)10^n
        {#1}%
      }%
    }%
}
\def\decadd@iv#1#2#3{% affiche le décimal "#1.#2"
  % le développement maximal initié par le \romannumeral de \decadd est actif
  \ifnum#1=\z@\ifnum#2<\z@% si #1=0 et #2<0
    \expandafter\expandafter\expandafter% transmettre le développement à \number
    -% puis afficher le signe "-"
  \fi\fi
  \number#1% affiche #1 qui est la somme des parties entières
  % poursuivre le développement initié par \number
  \unless\ifnum#2=\z@% si la partie décimale est différente de 0
    \antefi% se débarrasser de \fi
    \expandafter.% afficher le "." décimal après avoir
    \romannumeral-`\0\format@dec{#2}{#3}% correctement géré les 0 de #2
  \fi
}
\def\addzeros#1#2/#3.#4#5/#6.#7/{%
  \ifempty{#2}% si #1 est le dernier chiffre de y1
```

```
    {\ifempty{#5}% et si #4 est le dernier chiffre de y2
      {{#3#1}{#6#4}{#7}}% redonner les 3 arguments
      {\addzeros0/#3#1.#5/#6#4.#70/}% sinon alimenter #1 avec un 0
    }
    {\ifempty{#5}% si #4 est le dernier chiffre de y2
      {\addzeros#2/#3#1.0/#6#4.#70/}% alimenter #4 avec un 0
      {\addzeros#2/#3#1.#5/#6#4.#70/}% #2 et #5 non vides
    }%
}
\def\format@dec#1#2{% #1=partie décimale #2=seuil de retenue
  \expandafter\gobone% le \gobone agira en dernier,
    \romannumeral-`\0% mais avant, tout développer
    \expandafter\reverse\expandafter% et retarder le \reverse de fin
      % pour développer son argument qui
      {\number% développe tout et évalue avec \number
        \expandafter\reverse\expandafter% l'inversion de
        {\number\numexpr\abs{#1}+#2\relax}% abs(#1)+#2
      }%
}
\catcode`@12
a) $-3.78+1.6987=\decadd{-3.78}{1.6987}$\par
b) $3.56-3.06=\decadd{3.56}{-3.06}$\par
c) $4.125+13.49=\decadd{4.125}{13.49}$\par
d) $-0.99+1.005=\decadd{-0.99}{1.005}$\par
e) $16.6-19.879=\decadd{16.6}{-19.879}$\par
f) $5.789-0.698=\decadd{5.789}{-0.698}$\par
g) $0.123-0.123=\decadd{0.123}{-0.123}$\par
h) $3.14-16.4912=\decadd{3.14}{-16.4912}$\par
i) $0.1-0.98=\decadd{0.1}{-0.98}$\par
j) $2.43+7.57=\decadd{2.43}{7.57}$\par
h) \edef\foo{\decadd{1.23}{9.78}}\meaning\foo\par
j) \detokenize\expandafter\expandafter\expandafter{\decadd{3.14}{-8.544}}
```

a) $-3.78 + 1.6987 = -2.0813$
b) $3.56 - 3.06 = 0.5$
c) $4.125 + 13.49 = 17.615$
d) $-0.99 + 1.005 = 0.015$
e) $16.6 - 19.879 = -3.279$
f) $5.789 - 0.698 = 5.091$
g) $0.123 - 0.123 = 0$
h) $3.14 - 16.4912 = -13.3512$
i) $0.1 - 0.98 = -0.88$
j) $2.43 + 7.57 = 10$
h) macro:->11.01
j) -5.404

Chapitre 3

Primitives spécifiques à un moteur

Les nouvelles primitives introduites par ε-TEX font désormais partie du vocabulaire reconnaissable par tous les moteurs (pdfTEX, XƎTEX et luaTEX).

Cette compatibilité est malheureusement cassée par des primitives *spécifiques* aux moteurs, c'est-à-dire prises en charge par lui seul. On franchit là une ligne de démarcation importante pour la compatibilité puisqu'utiliser ces primitives rend obligatoire la compilation avec un moteur spécifique. Beaucoup de ces primitives spécifiques sont certes *très* pratiques, mais il est utile de se demander si les fonctionnalités additionnelles qu'elles apportent valent une rupture de compatibilité.

Il n'est évidemment pas question de décrire ici toutes les primitives spécifiques à tous les moteurs. Cela prendrait beaucoup de place (pdfTEX en a plus de 130 !) pour un intérêt limité, car la plupart sont correspondent à un besoin très précis et sont d'une utilisation assez pointue. Les documentations des moteurs en dressent la liste exhaustive, en précisent la syntaxe et décrivent les fonctionnalités offertes. Nous nous contenterons de présenter deux primitives de pdfTEX, moteur utilisé pour compiler le code source de ce livre.

Le lecteur pourra, s'il le souhaite, se plonger dans la documentation de son moteur préféré, explorer les primitives qu'il offre et pourquoi pas, les utiliser pour ses besoins propres. Il faut cependant garder à l'esprit que ces primitives imposant un moteur sont déconseillées si le code source a vocation à être diffusé et compilé par d'autres personnes non avisées de la rupture de compatibilité.

3.1. Comparer deux textes avec \pdfstrcmp

La primitive purement développable

\pdfstrcmp{⟨*texte 1*⟩}{⟨*texte 2*⟩}

permet de comparer les deux textes. Avant qu'elle n'entre en jeu, cette primitive développe au maximum ses deux arguments et les détokenize. *Elle ne tient donc pas compte des catcodes* contrairement au test \ifx :

\def\foo{⟨*texte 1*⟩} \def\bar{⟨*texte 2*⟩}
\ifx\foo\bar⟨*code vrai*⟩\else⟨*code faux*⟩\fi

La primitive \pdfstrcmp se développe en « 0 » si tes textes sont égaux, en « −1 » si le ⟨*texte 1*⟩ vient avant le ⟨*texte 2*⟩ et en « 1 » s'il vient après. Le classement des textes se fait en fonction des codes ASCII des caractères, pris successivement de gauche à droite jusqu'à ce qu'une différence apparaisse.

Code n° VI-451

```
a) \pdfstrcmp{foo}{bar}\qquad
b) \pdfstrcmp{bar}{foo}\qquad
c) \def\foo{ABC}\pdfstrcmp{\foo}{ABC}\qquad
d) \edef\foo{\string_}\pdfstrcmp{1_2}{1\foo2}\qquad
e) \pdfstrcmp{\string\relax}{\relax}
```

a) 1 b) -1 c) 0 d) 0 e) -1

La dernière comparaison est fausse, car « \detokenize{\relax} » se développe en \relax␣ alors que \string n'introduit pas d'espace après les séquences de contrôle. Les textes « \relax␣ » et « \relax » ne sont pas égaux.

Cette primitive dispose d'un équivalent pour le moteur XeTeX, mais elle ne porte pas le même nom : \strcmp.

3.2. Mesurer le temps

Voici un deuxième exemple de primitive spécifique à pdfTeX, \pdfelapsedtime qui permet de mesurer le temps de compilation. Cette primitive est de type entier et contient le nombre de secondes *d'échelle* écoulées depuis le début de la compilation : 1 seconde d'échelle vaut $\frac{1}{65536}$ de seconde tout comme un point d'échelle vaut $\frac{1}{65536}$ de point. Convertir les secondes d'échelle en secondes revient à convertir les sp en pt, ce que permet très facilement la macro \convertunit.

Code n° VI-452

```
\edef\tempcompil{\number\pdfelapsedtime}%
Depuis le début de la compilation, il s'est écoulé \tempcompil{} secondes d'échelle,
soit \convertunit{\tempcompil sp}{pt} secondes.
```

Depuis le début de la compilation, il s'est écoulé 2554949 secondes d'échelle, soit 38.98543 secondes.

Quelle que soit la durée de compilation, l'entier \pdfelapsedtime ne peut excéder $2^{31} - 1$ qui sera sa limite supérieure et qui représente 32 768 secondes.

La primitive \pdfresettimer, qui remet à 0 le compteur interne de secondes d'échelle, ouvre des perspectives intéressantes. Couplée à \pdfelapsedtime, elle

permet de mesurer avec précision la durée qui s'écoule entre deux moments arbitraires de la compilation. Nous allons ici mettre à profit cette possibilité pour mesurer la vitesse relative entre la boucle \loop...\repeat (celle de TEX puis celle de LATEX) et la boucle \for, dont la programmation a été abordée à partir de la page nº 175. Chaque boucle sera parcourue 100000 fois, mais, afin de mesurer la vitesse de la boucle elle-même, aucun code ne sera exécuté à chaque itération :

Code nº VI-453

```
%%%%%%%%%% definition de \loop...\repeat comme plain-TeX %%%%%%%%%%
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body \let\next\iterate \else\let\next\relax\fi \next}
\let\repeat=\fi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\newcount\testcnt
\pdfresettimer% remet le compteur à 0
  \testcnt=0
  \loop % Test no 1
    \ifnum\testcnt<100000 \advance\testcnt 1
  \repeat
Temps 1 = \convertunit{\pdfelapsedtime sp}{pt} s (boucle loop de \TeX)\par
%%%%%%%%%%% definition de \loop...\repeat comme LaTeX %%%%%%%%%%%
\def\loop#1\repeat{\def\iterate{#1\relax\expandafter\iterate\fi}%
  \iterate \let\iterate\relax}
\let\repeat\fi
\pdfresettimer% remet le compteur à 0
  \testcnt=0
  \loop % Test no 2
    \ifnum\testcnt<100000
    \advance\testcnt 1
  \repeat
Temps 2 = \convertunit{\pdfelapsedtime sp}{pt} s (boucle loop de \LaTeX)\par
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\pdfresettimer% remet le compteur à 0
  \for\ii=1 to 100000\do{}% Test no 3
Temps 3 = \convertunit{\pdfelapsedtime sp}{pt} s (boucle for)
```

Temps 1 = 0.05315 s (boucle loop de TEX)
Temps 2 = 0.04533 s (boucle loop de LATEX)
Temps 3 = 0.18024 s (boucle for)

Notre boucle \for semble donc environ 3 fois plus lente que la boucle \loop... \repeat de LATEX qui est la plus rapide. Il faut dire « *semble* », car l'incrémentation ne se fait pas sur le même type de donnée. La boucle \for incrémente le *texte de remplacement* d'une macro tandis que dans le code ci-dessus, un *compteur* est incrémenté dans la boucle \loop...\repeat. Pour être strictement équitable et pour être sûr de comparer les codes des deux boucles, la boucle \loop...\repeat doit imiter la boucle \for et incrémenter le texte de remplacement d'une macro :

Code nº VI-454

```
%%%%%%%%%%% definition de \loop...\repeat comme LaTeX %%%%%%%%%%%
\def\loop#1\repeat{\def\iterate{#1\relax\expandafter\iterate\fi}%
  \iterate \let\iterate\relax}
\let\repeat\fi
\pdfresettimer% remet le compteur à 0
  \def\ii{0}%
  \loop % Test no 1
    \ifnum\ii<100000
```

```
    \edef\ii{\number\numexpr\ii+1\relax}%
  \repeat
Temps 1 = \convertunit{\pdfelapsedtime sp}{pt} s (boucle loop de \LaTeX)\par
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\pdfresettimer% remet le compteur à 0
  \for\ii=1 to 100000\do{}% Test no 2
Temps 2 = \convertunit{\pdfelapsedtime sp}{pt} s (boucle for)
```

Temps 1 = 0.13908 s (boucle loop de LaTeX)
Temps 2 = 0.14131 s (boucle for)

La différence de vitesse est maintenant plus ténue, de l'ordre de 30%, ce qui semble une contrepartie acceptable en regard des fonctionnalités d'imbrication qu'offre `\for` par rapport à `\loop...\repeat`. On en tire par ailleurs l'enseignement qu'incrémenter un compteur est *plus rapide* que de recourir à `\numexpr` et `\edef` pour incrémenter un entier contenu dans le texte de remplacement d'une macro. En voici la démonstration :

Code n° VI-455

```
\newcount\testcnt
\pdfresettimer
\testcnt=0
\for\ii=1 to 100000\do{\advance\testcnt1 }
Temps 1 : \convertunit{\pdfelapsedtime sp}{pt} s (incrémentation compteur)

\pdfresettimer
\def\foo{0}%
\for\ii=1 to 100000\do{\edef\foo{\number\numexpr\foo+1\relax}}%
Temps 2 : \convertunit{\pdfelapsedtime sp}{pt} s (incrémentation du texte de remplacement)
```

Temps 1 : 0.15959 s (incrémentation compteur)
Temps 2 : 0.2155 s (incrémentation du texte de remplacement)

On peut ainsi comparer les vitesses entre les variantes d'un algorithme afin de trouver celle qui optimise la vitesse d'exécution.

Chapitre 4

Recueil des règles

Les règles se trouvant dans les cadres grisés ont été assez nombreuses (il y en a exactement 97). Les voici rassemblées ici.

1. Si un code quelconque, notons-le <*x*>, produit un affichage <*y*> alors, sauf cas très particulier, il n'est pas équivalent d'écrire <*x*> ou <*y*> dans l'argument d'une macro.

2. Lorsque TEX est en mode vertical, la commande `\par` est sans effet.
Par conséquent, si l'on se trouve en mode horizontal et si plusieurs commandes `\par` se suivent, seule la première est utilisée pour composer le paragraphe en cours et les autres sont ignorées.

3. Lorsque plusieurs espaces (ou plus généralement « caractères de catcode 10 ») se suivent dans le code source, TEX ne prend en compte que le premier d'entre eux et ignore les autres.
Si une ligne de code source commence par des espaces, ceux-ci sont ignorés.
Plain-TEX et LATEX assignent au caractère de tabulation (*HT*) le catcode de 10 ce qui signifie que ce caractère revêt toutes les propriétés de l'espace.

4. Dès que TEX lit un caractère, il lui affecte de façon inaltérable un code de catégorie.

5. Lorsqu'on écrit « ^^⟨*car*⟩ » dans le code source, tout se passe donc comme si TEX voyait le caractère se trouvant 4 lignes plus haut ou plus bas dans la table ASCII, selon que ⟨*car*⟩ se trouve dans la partie basse ou haute de cette table.
Si les *deux* caractères qui suivent ^^ forment un nombre hexadécimal à deux chiffres minuscules pris parmi « `0123456789abcdef` », alors TEX remplace ces 4 caractères par le caractère ayant le code hexadécimal spécifié. Avec cette méthode, on peut donc facilement accéder à n'importe quel caractère. Le caractère « z » peut aussi s'écrire « `^^7a` ».

6. Dans la ligne courante, si un caractère de catcode 14 (caractère de commentaire, généralement `%`) ou 5 (retour charriot, généralement `^^M`) apparait, alors, ce caractère ainsi que tout ce qui va jusqu'à la fin de la ligne est ignoré, y compris le caractère de code `\endlinechar` inséré à la fin de la ligne.

7. Un caractère de catcode 5 est interprété comme un espace.
Deux caractères de catcode 5 consécutifs sont transformés en la séquence de contrôle `\par` .

Cela signifie qu'un retour charriot est vu comme un espace, sauf s'il est précédé du caractère `%` et que deux retours charriots consécutifs (symbolisés dans le code source par une ligne vide) seront équivalents à `\par`.

8. Si `\endlinechar` est négatif ou supérieur à 255, aucun caractère n'est inséré aux fins de lignes.

9. Une séquence de contrôle commence par le caractère d'échappement de catcode 0 qui est habituellement « \ » et se termine à la fin de la plus longue série de lettres (caractères de catcode 11) qui le suit.

10. Voici des règles concernant les espaces et les séquences de contrôle :

1. dans le code qui est tapé, tout espace qui suit une unité lexicale de type séquence de contrôle est ignoré ;
2. si plusieurs espaces se suivent *dans le code*, seul le premier est pris en compte et les autres sont ignorés (ceci est un rappel de la règle page 27) ;
3. il découle des deux premiers points que, quel que soit le nombre d'espaces qui suivent une séquence de contrôle, ceux-ci sont ignorés.

11. Lorsque TEX assigne un texte de remplacement à une macro avec `\def`, ce texte de remplacement n'est pas exécuté, il est juste converti en tokens et rangé quelque part dans la mémoire de TEX pour être ressorti plus tard pour venir en replacement de la macro, et éventuellement être exécuté à ce moment.
Le texte de remplacement est très peu *analysé* lorsqu'il est stocké avec `\def`, ce qui veut dire que si une erreur est contenue dans le texte de remplacement, elle ne sera pas détectée au moment où `\def` agit mais plus tard lors du remplacement et de l'exécution proprement dite.
Une des rares choses que TEX vérifie dans le texte de remplacement d'une macro est qu'un caractère de catcode 15 n'y figure pas (nous verrons plus loin les autres vérifications que TEX effectue).

12. On peut créer des zones où les modifications faites aux macros et autres paramètres de TEX sont locales. Ces zones portent le nom de groupes.
Un groupe est délimité :

- soit par une accolade ouvrante et une accolade fermante auquel cas le groupe est qualifié de « simple » ;
- soit par les primitives `\begingroup` et `\endgroup` et dans ce cas, le groupe est dit « semi-simple ».

Il est entendu qu'un groupe ouvert avec une accolade ne peut être fermé qu'avec une accolade et il en est de même avec `\begingroup` et `\endgroup`.
À l'intérieur d'un groupe, les assignations sont locales à ce groupe et sont restaurées à leur état antérieur lors de la fermeture du groupe. Pour qu'une assignation soit *globale*, c'est-à-dire pour qu'elle survive à la fermeture du groupe, il faut faire précéder la commande d'assignation de la primitive `\global`.

Les groupes délimités par accolades et les groupes semi-simples peuvent être emboités, mais *ne doivent pas* se chevaucher.

13. La primitive `\aftergroup`⟨*token*⟩ permet de stocker un ⟨*token*⟩ dans une pile spéciale de la mémoire de TEXpour que ce token soit lu et exécuté juste après être sorti du groupe courant, que ce groupe soit simple ou semi-simple.
Si plusieurs commandes `\aftergroup` sont rencontrées dans un même groupe, les tokens mis en mémoire seront lus dans l'ordre où ils ont été définis. Ainsi, écrire

`\aftergroup`⟨*x*⟩`\aftergroup`⟨*y*⟩

se traduira par « ⟨*x*⟩⟨*y*⟩ » après la fermeture du groupe.

14. Lorsque le caractère d'échappement « \ » est suivi d'un caractère dont le catcode n'est pas 11, celui des lettres, seul ce caractère est pris en compte et TEX forme ce que l'on appelle un « caractère de contrôle ».

15. Contrairement aux séquences de contrôle dont le nom est formé de lettres, les espaces qui suivent les « caractères de contrôle » ne sont pas ignorés. La seule exception est la primitive `\ `, qui ajoute un espace à la liste courante. En effet, si cette dernière est suivie d'un espace, alors deux espaces se suivent dans le code et une autre règle stipule que le second est ignoré.

16. Un caractère est *actif* lorsque son code de catégorie vaut 13. Dans ce cas, on peut lui donner un texte de remplacement comme on le fait pour une macro.

17. Un espace après un caractère actif est pris en compte.

18. La primitive `\show` écrit dans le fichier `log` la « signification » du token qui la suit :

1. si ce token est une macro (ou un caractère actif), le texte « `macro->` » suivi du texte de remplacement de cette macro est écrit ;
2. si ce token est une primitive, qui par définition n'a pas de texte de remplacement, le nom de la primitive est écrit ;
3. sinon, TEX écrit un texte bref (caractérisant le catcode du token) suivi du token.

La primitive `\meaning` écrit les mêmes choses que `\show` sauf qu'elle les écrits dans le flux de lecture de TEX, dans la pile d'entrée.

19. Un argument d'une macro est :

- soit un token c'est-à-dire un caractère (un octet) ou une séquence de contrôle ;
- soit le code qui se trouve entre deux accolades, étant entendu que ce code est un ensemble de tokens équilibrés en accolades ouvrantes et fermantes.

20. Un espace *non entouré d'accolades* est ignoré en tant qu'argument.

21. Lorsque des accolades délimitent un argument d'une macro, elles ne jouent pas le rôle de délimiteur de *groupe*.

22. Une macro peut accepter de 0 à 9 arguments.
Pour le spécifier, on indique juste après `\def`, dans le « texte de paramètre », les arguments les uns à la suite des autres sous la forme « `#1` », « `#2` », et ainsi de suite jusqu'au nombre d'arguments que l'on veut donner à la macro.
Le texte de paramètre est strict sur l'ordre des arguments, mais en revanche, il n'y a aucune contrainte sur l'ordre et l'existence des arguments dans le texte de remplacement de la macro. On peut donc les y écrire dans l'ordre que l'on veut, autant de fois que l'on souhaite, voire ne pas en écrire certains.

23. Si l'argument ⟨*i*⟩ d'une macro est vide, cela équivaudra dans le texte de remplacement de la macro à ce que `#`⟨*i*⟩ n'existe pas.

24. Une macro ne peut admettre la primitive `\par` dans aucun de ses arguments. Elle admet en revanche toute séquence de contrôle `\let`-équivalente à `\par`, par exemple `\endgraf`, définie par plain-TEX.
Pour définir une macro capable d'accepter `\par` dans ses arguments, on doit faire précéder `\def` de la primitive `\long`.

25. Lorsqu'une macro est *définie*, les arguments figurant dans le texte de remplacement sous la forme `#`⟨*chiffre*⟩ (où « # » symbolise un token ayant le catcode 6 au moment de la définition) sont des endroits où TEX mettra les arguments lus plus tard lorsque la macro sera appelée.
Lors de sa lecture à grande vitesse, le reste du texte de remplacement est « tokénizé » c'est-à-dire que les caractères composant le code source sont transformés en tokens et les catcodes qui leur sont assignés obéissent au régime en cours lorsque la définition a lieu. Ces tokens sont stockés dans la mémoire de TEX.

Lorsqu'une macro est *appelée* et qu'elle *lit* ses arguments, ceux-ci sont lus à grande vitesse et sont tokénizés selon le régime de catcode en vigueur cette macro est appelée (et qui peut donc différer de celui en cours lors de sa définition). Ils sont ensuite insérés aux endroits où `#`⟨*chiffre*⟩ était écrit dans le texte de remplacement. Les tokens du texte de remplacement ne sont pas modifiés et restent ce qu'ils étaient lors de la définition.

26. Lorsque TEX lit du code source, d'irréversibles pertes d'informations ont lieu lorsque des *caractères* sont transformés en *tokens*.

27. Lorsqu'on définit une macro « fille » dans le texte de remplacement d'une macro « mère », les # concernant les arguments de la macro fille sont doublés, aussi bien dans le texte de paramètre que dans le texte de remplacement.
Ils seront doublés à nouveau à chaque imbrication supplémentaire pour éviter de confondre les arguments des macros imbriquées.

28. La primitive `\lowercase` doit être suivie de son argument entre accolades et cet argument, constitué d'un texte dont les accolades sont équilibrées, est lu de la façon suivante :

- les séquences de contrôles restent inchangées ;
- pour chaque caractère, le code de caractère est remplacé par le code minuscule, étant entendu que le code de catégorie *ne change pas*.
  Si le code minuscule d'un caractère est nul, ce caractère n'est pas modifié par `\lowercase`.

29. Si l'on écrit « \lccode⟨*entier1*⟩=⟨*entier2*⟩ » alors, dans l'argument de \lowercase, tous les caractères de code ⟨*entier1*⟩ seront lus comme des caractères de code ⟨*entier2*⟩ tout en gardant leur catcode originel.
En pratique, si l'on écrit

\lccode`\⟨*car1*⟩=`\⟨*car2*⟩

puis

\lowercase{⟨*texte*⟩}

alors, dans le ⟨*texte*⟩, tous les ⟨*car1*⟩ sont lus comme ⟨*car2*⟩, mais en conservant le catcode de ⟨*car1*⟩.

30. Un argument #<*i*> est dit « délimité » si, dans le texte de paramètre, il est suivi par autre chose que #<*i*+1> ou que l'accolade « { » qui marque le début du texte de remplacement.

31. Lorsqu'une macro est exécutée et qu'elle doit lire une suite de n arguments dont le dernier est délimité, les $n-1$ premiers arguments sont assignés comme des arguments non délimités. Le dernier argument reçoit ce qui reste jusqu'au délimiteur, ceci étant éventuellement vide.

32. Dans tous les cas, qu'il soit délimité ou pas, si un argument est de la forme {⟨*tokens*⟩}, où ⟨*tokens*⟩ est un ensemble d'unités lexicales où les accolades sont équilibrées, alors TEX *supprime les accolades extérieures* et l'argument lu est « ⟨*tokens*⟩ ».

33. Un argument délimité est vide si l'ensemble des tokens qui lui correspond est vide ou « {} ».

34. Pour spécifier que le dernier argument ⟨*i*⟩ d'une macro doit être délimité par une accolade ouvrante, il faut écrire dans le texte de paramètre

#⟨*i*⟩#{

L'accolade qui suit le # sert de délimiteur et marque aussi le début du texte de remplacement. Lorsque la macro sera exécutée, l'argument #⟨*i*⟩ s'étendra jusqu'à la prochaine accolade ouvrante.

35. Lorsqu'un délimiteur d'argument est une séquence de contrôle, TEX ne prend pas en compte la signification de cette séquence de contrôle : seuls les caractères qui composent son nom sont examinés.

36. Les délimiteurs figurant dans le texte de paramètre d'une macro ne peuvent pas contenir de tokens de catcode 1 ou 2 équilibrés (accolade ouvrante et fermante) car celles-ci seraient interprétées non pas comme faisant partie d'un délimiteur, mais comme marquant les frontières du texte de remplacement.

La conséquence est donc que lorsqu'une macro définit une macro fille à argument délimité où les délimiteurs sont les arguments de la macro mère, ceux-ci ne peuvent pas contenir un groupe entre accolades.

37. TEX est un langage de macros, c'est-à-dire qu'il agit comme un outil qui *remplace* du code par du code. Ce remplacement, aussi appelé développement, a lieu dans une zone de mémoire nommée « pile d'entrée ».
Ainsi, lorsque TEX exécute une macro, elle (ainsi que ses éventuels arguments) est d'abord remplacée par le code qui lui a été donné lorsqu'on l'a définie avec `\def`. Ce code est le « texte de remplacement ». Une fois que ce développement a été fait, TEX continue sa route en tenant compte du remplacement effectué.

38. Une séquence de contrôle est développable si TEX peut la remplacer par un code différent.

39. La lecture du code se fait séquentiellement de gauche à droite et TEX ne cherche pas à lire plus de tokens que ce dont il a besoin.

Lorsque TEX rencontre une séquence de contrôle, il lit aussi ses éventuels arguments pour procéder à son développement qui consiste à remplacer le tout par du code. Si la pile ne contient pas assez d'arguments, TEX va chercher ces arguments manquants juste après la pile, c'est-à-dire dans le code source.

40. Si $\langle x\rangle$ et $\langle y\rangle$ sont deux *tokens*, écrire

$$\texttt{\textbackslash expandafter}\langle x\rangle\langle y\rangle$$

a pour effet, lorsque TEX développe `\expandafter`, de 1-développer $\langle y\rangle$ sans pour autant que la tête de lecture ne bouge de place (et donc reste devant $\langle x\rangle$ qu'elle n'a toujours pas lu).
Le `\expandafter` disparait après s'être développé et on obtient donc

$$\langle x\rangle\langle *y\rangle$$

où « * » indique le 1-développement du token $\langle y\rangle$.

41. Lorsqu'on stocke des tokens dans une $\backslash\langle$*macroA*$\rangle$ (comme cela arrive très souvent), si l'on veut mettre ces tokens dans l'argument d'une autre macro $\backslash\langle$*macroB*$\rangle$, il *faut* développer $\backslash\langle$*macroA*$\rangle$ avant que $\backslash\langle$*macroB*$\rangle$ n'agisse.

42. Pour 1-développer un token précédé de n tokens, il faut mettre un `\expandafter` devant chacun de ces n tokens.

43. Les primitives qui doivent immédiatement être suivies d'une accolade ouvrante ont la particularité suivante : elles développent au maximum jusqu'à trouver l'accolade ouvrante.
Nous connaissons :

- `\lowercase` et `\uppercase` ;
- `\detokenize` ;
- `\toks`$\langle$*nombre*$\rangle$=␣ (ou $\backslash\langle$*nom*$\rangle$=␣ si on est passé par `\newtoks`$\backslash\langle$*nom*$\rangle$), sachant que le signe = et l'espace qui le suit sont facultatifs.

La conséquence est que l'on peut placer un (ou plusieurs) `\expandafter` entre ces primitives et l'accolade ouvrante pour développer leur argument avant qu'elles n'agissent.

44. Pour provoquer le k-développement d'un token $\langle y\rangle$ précédé par n autres tokens, il faut placer 2^k-1 `\expandafter` devant chacun des n tokens qui le précèdent.

45. La primitive \edef a la même syntaxe et les mêmes actions que \def, c'est-à-dire qu'elle stocke un texte de remplacement associé à une séquence de contrôle. La différence réside dans le fait que la totalité du texte de remplacement est développée *au maximum* avant que l'assignation soit faite.
Le développement s'applique au texte de remplacement *hors arguments* symbolisés par #⟨*chiffre*⟩. Ceux-ci, qu'ils soient délimités ou pas, seront insérés tels qu'ils seront lus dans le texte de remplacement (qui lui aura été préalablement développé).
La primitive \xdef effectue les mêmes actions que \edef sauf que la définition faite est globale.

46. La primitive développable \noexpand bloque le développement du token x qui la suit. Elle a pour effet, lorsqu'elle se 1-développe, d'être le token x lui-même. Si x est une séquence de contrôle qui aurait été développée, x est rendu temporairement égal à \relax.

47. Il existe 4 moyens de bloquer le développement maximal dans les arguments de certaines primitives :

1. mettre un \noexpand devant chaque token dont on veut bloquer le développement ;
2. placer les tokens dont on veut bloquer le développement dans l'argument de la primitive \unexpanded de ε-TEX ;
3. stocker au préalable tous les tokens dont on veut bloquer le développement dans un ⟨*registre de token*⟩ et écrire dans l'argument de la primitive \the⟨*registre de token*⟩ ;
4. lorsqu'on définit une macro, il est possible de faire précéder le \def de la primitive \protected de ε-TEX pour empêcher le développement maximal de cette macro.

48. Un code est purement développable si son développement maximal se réduit aux caractères qui auraient été affichés s'il avait été exécuté par TEX.
Dans la majorité des cas, un tel code ne doit être constitué que caractères (ceux que l'on souhaite dans l'affichage final) ainsi que de séquences de contrôle ou caractères actifs *développables*. Citons :

- du côté des primitives : \expandafter, \number, \the, \string ainsi que les primitives de ε-TEX \detokenize et \unexpanded. Tous les tests de TEX que nous verrons plus tard sont développables. Il y a aussi deux primitives de ε-TEX, \numexpr et \dimexpr qui seront étudiées dans la partie suivante ;
- les macros définies par l'utilisateur, qu'elles soient à arguments délimités ou pas, pourvu que leur développement maximal soit constitué des caractères que l'on souhaite dans l'affichage final.

49. Un « nombre » est un entier relatif compris entre $-2^{31}+1$ et $2^{31}-1$, c'est-à-dire entre −2 147 483 647 et 2 147 483 647. Tout nombre hors de cet intervalle provoque une erreur de compilation.
Lorsque TEX s'attend à lire un nombre, il entre dans une phase de développement maximal. Dans les résidus de ce développement, il cherche tout d'abord une série de signes optionnels de catcode 12 : « + » et « − ». Le signe du nombre final dépendra de la parité du nombre de signes « − » lus au total. Ensuite, plusieurs cas peuvent se présenter :

1. le nombre est explicitement écrit en
   - base 8 sous la forme '⟨*signes*⟩ ;
   - base 10 sous la forme ⟨*signes*⟩ ;
   - base 16 sous la forme "⟨*signes*⟩,

où les ⟨*signes*⟩ sont une suite de chiffres de catcode 12 allant de `0` à `7` pour la base 8, de `0` à `9` pour la base 10 et pour la base 16, de `0` à `9` auxquels s'ajoutent les lettres *majuscules* de catcode 11 ou 12 allant de `A` à `F`.

TEX stoppe la lecture du nombre au premier token qui n'est pas un ⟨*signe*⟩ valide. Si ce token est un espace, il sera absorbé. Le nombre lu comprend donc la plus longue série de ⟨*signes*⟩ valides possible ;

2. un « compteur » se présente (nous verrons les compteurs plus loin). Le nombre qu'il contient est lu et la lecture du nombre est stoppée ;
3. un « registre de dimension » se présente. Dans ce cas, la dimension est convertie en entier (nous verrons comment plus loin) et la lecture du nombre s'achève ;
4. une séquence de contrôle définie avec `\chardef` pour les nombres compris entre 0 à 255 ou avec `\mathchardef` pour ceux compris entre 0 à 32767 se présente, le nombre correspondant est lu et la lecture du nombre est stoppée ;
5. un nombre écrit sous la forme `‘`⟨*car*⟩ ou `‘\`⟨*car*⟩ se présente, où « `‘` » est l'apostrophe inverse. Le nombre lu est le code de caractère de ⟨*car*⟩. Le *développement maximal se poursuit* jusqu'au premier token non développable qui signe la fin de la lecture du nombre. *Si ce token est un espace, il est absorbé.*

50. La primitive `\romannumeral` doit être suivie d'un ⟨*nombre*⟩ valide.
Son 1-développement est

- l'écriture en chiffres romains de catcode 12 si le ⟨*nombre*⟩ est strictement positif ;
- vide si le nombre est négatif ou nul.

Si le ⟨*nombre*⟩ est supérieur à 1000, la lettre « `m` » de catcode 12 sera produite autant de fois qu'il y a de milliers dans le nombre.

51. Un compteur a vocation à contenir un nombre entier de $-2^{31}+1$ à $2^{31}-1$. On y fait référence par

`\count`⟨*nombre*⟩

Il est aussi possible d'utiliser la commande

`\newcount\`⟨*macro*⟩

qui fait le nécessaire pour lier la `\`⟨*macro*⟩ au prochain numéro de compteur non utilisé. Par la suite, la `\`⟨*macro*⟩ se comporte exactement comme `\count`⟨*nombre*⟩. Comme les registres de tokens, les ⟨*compteurs*⟩ peuvent donc indifféremment être désignés selon ces deux possibilités.

Pour assigner un ⟨*entier*⟩ (voir la définition d'un entier à la page 125) à un compteur, il faut écrire

⟨*compteur*⟩`=` ⟨*entier*⟩

où le signe = ainsi que l'espace qui le suit sont facultatifs.

52. Dans les lignes ci-dessous, ⟨*n*⟩ est un nombre entier au sens de TEX :

- `\advance`⟨*compteur*⟩ `by` ⟨*n*⟩ : ajoute ⟨*n*⟩ au ⟨*compteur*⟩ ;
- `\multiply`⟨*compteur*⟩ `by` ⟨*n*⟩ : multiplie le ⟨*compteur*⟩ par ⟨*n*⟩ ;
- `\divide`⟨*compteur*⟩ `by` ⟨*n*⟩ : divise le ⟨*compteur*⟩ par ⟨*n*⟩ en lui assignant la troncature à l'unité si le résultat de la division n'est pas entier.

Dans tous les cas, le mot « `by` » et l'espace qui le suit sont facultatifs.

53. La primitive \numexpr permet de calculer des enchainements d'opérations sur les entiers, pris au sens de la définition de la page 125. Elle doit être suivie d'une expression arithmétique où les espaces sont ignorés, les signes opératoires sont +, –, *, / et où les parenthèses sont admises .
Elle cherche à évaluer ce qui la suit en amorçant un développement maximal jusqu'à rencontrer un token qui ne peut figurer dans une expression arithmétique. Si l'expression arithmétique est suivie d'un \relax, alors ce \relax est mangé par \numexpr. Ceci constitue une spécificité de la primitive \numexpr puisqu'avec TEX, seuls les *espaces* sont mangés après les nombres. C'est donc une excellente habitude de faire suivre une expression arithmétique évaluée par \numexpr d'un \relax.
La structure

\numexpr⟨*expression arithmétique*⟩\relax>

est un entier au sens de TEX, mais un peu comme l'est un compteur, c'est une représentation *interne* d'un entier et donc, pour l'afficher, il faut recourir à une primitive qui transforme la représentation interne en caractères affichables telle que \the, \number ou \romannumeral.

54. Les tests de TEX sont exécutés par des primitives qui obéissent aux contraintes suivantes :

1. le nom des primitives exécutant un test commence par les lettres « if » suivies d'autres lettres notées ⟨*test*⟩ déterminant de quel test il s'agit ;
2. si \if⟨*test*⟩ est une primitive exécutant un test sur ses ⟨*arguments*⟩, sa structure est la suivante :

```
\if⟨test⟩⟨arguments⟩
    ⟨code exécuté si le test est vrai⟩
\else
    ⟨code exécuté si le test est faux⟩
\fi
```

   La branche entre \else et \fi est facultative et donc, un test peut également avoir la structure suivante :

```
\if⟨test⟩⟨arguments⟩
    ⟨code exécuté si le test est vrai⟩
\fi
```

Au lieu de \else et \fi, on peut mettre toute séquence de contrôle rendue \let-égale à \else ou \fi.
Les tests et les primitives \else et \fi sont développables.

55. Lorsqu'un test est exécuté par une primitive de test \if⟨*test*⟩, les primitives \else et \fi ne sont *pas* supprimées lorsque le test est fait. Elles restent en place et lorsqu'elles seront rencontrées plus tard, elles s'autodétruiront par un simple développement.

56. La primitive \ifnum effectue une comparaison entre deux entiers. La syntaxe est de la forme

```
\ifnum⟨entier1⟩⟨signe⟩⟨entier2⟩
    ⟨code exécuté si le test est vrai⟩
\else
    ⟨code exécuté si le test est faux⟩
\fi
```

où le ⟨*signe*⟩, de catcode 12, est soit « = » si l'on veut tester l'égalité entre les deux entiers, soit « < » ou « > » s'il l'on cherche à tester une inégalité stricte.

57. Le test \ifcase teste si un entier est successivement égal à 0, 1, 2, ... selon la syntaxe suivante :

```
\or
  \ifcase⟨nombre⟩
      ⟨code exécuté⟩ si ⟨nombre⟩ = 0
  \or
      ⟨code exécuté⟩ si ⟨nombre⟩ = 1
  \or
      ⟨code exécuté⟩ si ⟨nombre⟩ = 2
  etc...
  \else
      ⟨code alternatif⟩% exécuté si aucune égalité précédente n'est vérifiée
  \fi
```

Les entiers auxquels le ⟨*nombre*⟩ est comparé commencent nécessairement à 0, sont consécutifs et vont aussi loin qu'il y a de branches \or. Il est donc nécessaire d'écrire autant de \or que l'on veut envisager de cas.
La branche \else⟨*code alternatif*⟩ est facultative.

58. Une récursivité est terminale lorsque rien n'est laissé sur la pile lors de l'appel récursif. Pratiquement, la récursivité est terminale si l'appel récursif intervient *en dernier* dans le code de la macro qui s'appelle elle-même ou intervient après avoir supprimé par développement tout ce qui se trouve après cet appel.

59. Le test \ifx compare les significations des deux *tokens* qui le suivent. Il a la syntaxe suivante :

```
\ifx⟨token1⟩⟨token2⟩
    ⟨code vrai⟩
\else
    ⟨code faux⟩
\fi
```

Il est vrai si la signification des deux tokens est la même. Parmi toutes les possibilités qui se présentent selon la nature des deux tokens, les trois cas suivants sont les plus courants :

1. si les deux tokens sont des caractères, alors le test est positif si les caractères sont identiques, c'est-à-dire si leur code de caractère *et* leur catcode sont égaux ;
2. si les deux tokens sont des primitives, le test est positif si les primitives sont les mêmes ;
3. si les deux tokens sont des macros ou des caractères actifs, le test \ifx compare leurs textes de remplacement : le test est positif si les textes de remplacement sont identiques, c'est-à-dire constitués des mêmes tokens avec les mêmes catcodes. La comparaison tient également compte des propriétés de ces macros, notamment si elles sont \long ou \outer.
   Deux macros non définies sont égales pour \ifx, quels que soient leurs noms.

60. Le test \ifx ne distingue pas deux tokens \let-égaux.

61. Un quark est une macro dont le texte de remplacement est constitué de la macro elle-même.
Un quark ne doit *jamais* être exécuté, car étant invariant par développement, son traitement par TEX engendrerait une boucle infinie.

Si \⟨*quark*⟩ est un quark et si \⟨*macro*⟩ est définie par

\def\⟨*macro*⟩{\⟨*quark*⟩} ou par \let\⟨*macro*⟩ = \⟨*quark*⟩

alors, le test \ifx\⟨*macro*⟩\⟨*quark*⟩ sera vrai dans les deux cas.

62. Lorsqu'une liste d'arguments consécutifs doit être lue au fur et à mesure par une macro récursive, deux méthodes de lecture sont envisageables. Elles ont en commun une première lecture de tous les arguments afin de les passer à la macro récursive et c'est cette dernière qui a deux façons de fonctionner :

1. lire la totalité des arguments restants à chaque itération, utiliser ceux qui sont utiles à l'itération en cours et recommence1r avec tous les arguments restants. S'arrêter lorsque la liste d'arguments restants est vide ;
2. ne lire *que* les arguments utiles à l'itération en cours, les utiliser et recommencer. S'arrêter lorsque l'un des arguments lus est égal à un argument spécifique préalablement placé en fin de liste par la macro chapeau.

63. Les pseudotests \iftrue et \iffalse n'effectuent aucun test et donc n'admettent aucun argument. Ils sont respectivement toujours vrai et toujours faux. En dépit de ce caractère invariable, ils se comportent comme n'importe quel test, notamment en ce qui concerne l'éventuelle présence du \else et celle, obligatoire, du \fi.

La macro \newif, suivie d'une macro \if⟨*nom*⟩ crée 3 nouvelles macros dont le comportement est identique à celui des variables booléennes. Le test \if⟨*nom*⟩ sera vrai si la macro \⟨*nom*⟩true a été exécutée auparavant et sera faux si c'est la macro \⟨*nom*⟩false.

64. Le test \ifcat lance le développement maximal et dès qu'il obtient deux tokens non développables (ou dont le développement est bloqué par \noexpand ou \unexpanded), il effectue la comparaison des codes de catégorie.

Si un token testé par \ifcat est une séquence de contrôle ou un caractère actif rendu \let-égal à un ⟨*token*⟩, \ifcat prend en compte le code de catégorie du ⟨*token*⟩.

Si un token testé par \ifcat est une séquence de contrôle, son code de catégorie est vu égal à 16.

65. Le test \if instaure un développement maximal et dès qu'il obtient deux tokens non développables (ou dont le développement est bloqué par \noexpand ou \unexpanded), il effectue la comparaison de leurs codes de caractère.

Si un token testé par \if est une séquence de contrôle ou un caractère actif rendu \let-égal à un ⟨*token*⟩, \if prend en compte le code de caractère du ⟨*token*⟩.

Si un token testé par \if est une séquence de contrôle, son code de caractère est vu égal à 256.

66. Lorsqu'un \else ou un \fi apparié avec un test est rencontré lors de l'évaluation de ce test, TEX insère un \relax spécial afin que le \else ou le \fi demeurent hors de l'évaluation faite par le test.

67. Une dimension fixe est utilisée pour mesurer une longueur prise au sens géométrique du terme. TEX dispose de 256 registres de dimension (32 768 avec ε-TEX), chacun capable d'héberger une dimension. Pour faire référence à l'un de ces registres, on utilise la primitive \dimen suivie de son numéro (sous la forme d'un ⟨*entier*⟩).

Il est possible de demander l'allocation d'un registre inutilisé avec la macro de plain-TEX \newdimen\⟨*macro*⟩. La \⟨*macro*⟩, définie en coulisse par la primitive \dimendef avec

\global\dimendef\⟨*macro*⟩=⟨*nombre*⟩

sera par la suite équivalente à \dimen⟨*nombre*⟩ où le ⟨*nombre*⟩ est celui du prochain registre libre au moment de l'allocation.
Un ⟨*registre de dimension*⟩ est donc soit une \⟨*macro*⟩ préalablement définie à l'aide de \newdimen, soit « \dimen⟨*nombre*⟩ » où ⟨*nombre*⟩ est le numéro du registre de dimension auquel on souhaite faire référence.
Pour assigner une dimension à un registre de dimension, la syntaxe est

⟨*registre de dimension*⟩= ⟨*dimension*⟩

où le signe = et l'espace qui le suit sont facultatifs. Pour lire la ⟨*dimension*⟩, le développement maximal se met en marche et après développement, une ⟨*dimension*⟩ est :

- une dimension explicitement écrite, c'est-à-dire un ⟨*nombre*⟩ (entier) ou un nombre décimal suivi d'espaces facultatifs et de deux lettres de catcode 11 ou 12 représentant une des unités de dimensions qu'accepte TEX à savoir pt, pc, in, bp, cm, mm, dd, cc, sp, em et ex. Le séparateur décimal peut indifféremment être le point ou la virgule pourvu que son catcode soit 12. La lecture de la dimension s'arrête au premier token qui suit les deux caractères d'unité. Si ce token est un espace, celui est absorbé.
- un ⟨*registre de dimension*⟩. Si ce registre est précédé d'un nombre décimal signé, la dimension obtenue sera celle du registre multipliée par le nombre signé ;
- un ⟨*registre de ressort*⟩ (voir la définition à la section suivante) auquel cas, la dimension retenue est égale à la composante non étirable du registre de ressort. Si ce registre est précédé d'un nombre décimal signé, la dimension obtenue sera la composante fixe du ressort multipliée par le nombre signé.

La valeur absolue maximale d'une ⟨*dimension*⟩ est contenue dans le registre de dimension \maxdimen et vaut 16383.99999pt, ce qui représente 575.83199cm.

68. La primitive \dimexpr de ε-TEX doit être suivie d'un enchainement d'opérations sur des dimensions qui est évalué selon les règles mathématiques habituelles : parenthèses, opérations (+, *, /), priorités. Cette primitive possède les propriétés suivantes :

- tout comme \numexpr, \dimexpr amorce un développement maximal pour évaluer l'⟨*expression*⟩ qui la suit. L'⟨*expression*⟩ prend fin au premier caractère qui ne peut faire partie d'un calcul sur les dimensions. La primitive \relax peut marquer la fin d'une ⟨*expression*⟩ et dans ce cas, elle est absorbée par \dimexpr ;
- les espaces dans l'⟨*expression*⟩ sont ignorés ;
- dans l'⟨*expression*⟩, une ⟨*dimension*⟩ ne peut être multipliée ou divisée (avec * ou /) que par un *entier*. Dans ce cas, l'entier *doit* être l'opérateur c'est-à-dire qu'il doit suivre la dimension. On doit donc écrire

  ⟨*dimension*⟩*⟨*entier*⟩ ou ⟨*dimension*⟩/⟨*entier*⟩

- une ⟨*expression*⟩ évaluée par \dimexpr est du même type qu'un registre de dimension, il s'agit donc d'une représentation *interne* d'une dimension. On peut convertir cette représentation interne en caractères de catcode 12 en faisant précéder \dimexpr de la primitive développable \the.

69. Toutes les dimensions fixes que TEX manipule sont converties en interne en le nombre *entier* de points d'échelle (sp) qu'elles représentent. Une dimension s'apparente donc à un entier.
Si TEX est à un endroit où il s'attend à lire un ⟨*nombre*⟩ entier et qu'il lit un ⟨*registre de dimension*⟩, il convertit la dimension contenue dans le registre en « entier contraint » qui est le nombre de sp qu'elle représente. Cet entier contraint est le ⟨*nombre*⟩ qui sera pris en compte.

70. Un ressort est une dimension assortie de composantes étirables optionnelles (éventuellement infinies) de sorte qu'il peut, selon le contexte et sa définition, se comprimer ou s'étirer. TEX offre 256 registres de ressorts et ce nombre monte à 32 768 avec ε-TEX. La primitive `\skip` suivie d'un ⟨*nombre*⟩ permet d'accéder à un registre spécifique par son numéro.
Plain-TEX fournit la macro `\newskip\`⟨*macro*⟩ qui met à profit la primitive `\skipdef` selon cette syntaxe

`\skipdef\`⟨*macro*⟩`=`⟨*nombre*⟩

pour rendre cette `\`⟨*macro*⟩ équivalente à `\skip`⟨*nombre*⟩, où le ⟨*nombre*⟩ est celui du prochain registre libre. Dès lors, un ⟨*registre de ressort*⟩ est soit `\skip` suivi d'un numéro, soit une `\`⟨*macro*⟩ préalablement définie par `\newskip`.
Pour assigner un ressort à un ⟨*registre de ressort*⟩, il faut écrire :

⟨*registre de ressort*⟩`=` ⟨*ressort*⟩

où le signe « = » et l'espace qui le suit sont optionnels. Pour lire le ⟨*ressort*⟩, le développement maximal se met en marche. Un ⟨*ressort*⟩ est une ⟨*dimension*⟩ éventuellement suivie d'une composante d'étirement optionnelle de la forme « `plus` ⟨*étirement*⟩ » puis d'une composante de compression optionnelle de la forme « `minus` ⟨*étirement*⟩ ».
Si elles sont présentes toutes les deux, ces deux composantes optionnelles doivent *nécessairement* être déclarées dans cet ordre. Si elles ne sont pas spécifiées, les composantes étirables valent `0pt` par défaut.
Un ⟨*étirement*⟩ est :

- soit une composante fixe qui est une ⟨*dimension*⟩ comme « `1.5pt` » ou « `0.25cm` » ;
- soit une composante infinie qui prend la forme d'un coefficient d'infini suivi de « `fil` », « `fill` » ou « `filll` », où ces 3 mots-clé représentent des infinis de *forces* différentes : « `filll` » est infiniment plus grand que « `fill` », quels que soient les coefficients qui les précèdent et de la même façon, « `fill` » est infiniment plus grand que « `fil` ».
  Le coefficient d'infini est un décimal signé dont la valeur absolue peut prendre toutes les valeurs entre `0` et `16383.99999`.

71. Pour insérer une espace insécable, on utilise la primitive `\kern` suivie d'une ⟨*dimension*⟩ qui sera celle de l'espace insérée.
L'espace sera insérée dans le mode en cours : elle sera horizontale si TEX est en mode horizontal à ce moment-là et sera verticale s'il est en mode vertical.
L'espace ainsi créé est *insécable* car aucune coupure de ligne ou de page ne pourra s'y faire.

72. La primitive `\hskip`, suivie d'un ⟨*ressort*⟩ insère une espace (dont les dimensions sont celles permises par le ressort) dans la liste horizontale en cours. La primitive `\vskip` en est le pendant pour le mode vertical.
Si TEX rencontre une de ces deux primitives et ne se trouve pas dans le mode qu'elles exigent, alors le mode courant se termine et TEX passe dans le mode requis.
Les espaces ainsi insérées sont sécables, c'est-à-dire susceptibles d'être remplacées par des coupures de ligne ou de pages. De plus, elles sont ignorées lorsqu'elles se trouvent immédiatement avant la fin du paragraphe pour `\hskip` ou de la fin de la page pour `\vskip`.

73. TEX dispose de trois primitives pour enfermer un matériel dans une boite :

- `\hbox{`⟨*code*⟩`}` construit une boite *horizontale* contenant du matériel (qui est le résultat de la composition du ⟨*code*⟩) disposé en mode horizontal et compatible avec ce mode ;

– \vbox{⟨*code*⟩} et \vtop{⟨*code*⟩} bâtissent des boites *verticales*, susceptibles donc de recevoir du matériel disposé en mode vertical et compatible avec ce mode.

Dans les trois cas, l'intérieur d'une boite tient lieu de groupe, c'est-à-dire que les assignations faites dans le ⟨*code*⟩ restent locales à la boite.
Le ⟨*code*⟩ n'est pas lu en une fois comme s'il était l'argument d'une macro ; il est lu *au fur et à mesure* que la boite se remplit et donc au fil de cette lecture, des changements de catcode sont permis.
Une boite ainsi construite peut être :

– affichée lorsque \hbox, \vbox ou \vtop sont placés dans le flux d'entrée de TEX. Dans ce cas, la boite est ajoutée à la liste en cours selon le mode dans lequel TEX travaille à ce moment (horizontal ou vertical). Il est important de noter que ni \hbox ni \vbox ou \vtop ne provoquent de changement de mode, même si à l'intérieur de ces boites, un mode « interne » est imposé ;
– stockée dans un registre de boite (voir la section suivante).

74. Les primitives \vbox et \vtop définissent des boites dans lesquelles des éléments sont empilés verticalement.
La différence entre \vbox et \vtop réside dans le point de référence de la boite finale. Dans les deux cas, les éléments sont empilés verticalement, le premier étant en haut et le dernier en bas. Avec \vbox, le point de référence de la boite est le point de référence du *dernier* élément (celui du bas) alors qu'avec \vtop, son point de référence est celui du *premier* élément (celui du haut).

75. TEX dispose de 256 registres de boites (32 768 avec ε-TEX).
On demande à TEX d'allouer un numéro de registre de boite par l'intermédiaire de la macro \newbox\⟨*macro*⟩ et ce faisant, l'assignation suivante est faite

\global\chardef\⟨*macro*⟩=⟨*nombre*⟩

où le ⟨*nombre*⟩ est un numéro de registre de boite libre.
Ce faisant, la \⟨*macro*⟩ devient équivalente à \char⟨*nombre*⟩. Ainsi, \⟨*macro*⟩ produit le caractère de code ⟨*nombre*⟩, mais lorsque TEX s'attend à lire un nombre (comme c'est le cas lorsqu'il attend un numéro de boite), \⟨*macro*⟩ est lue comme l'entier ⟨*nombre*⟩. Du point de vue de la mécanique interne, un ⟨*registre*⟩ de boite est donc un ⟨*nombre*⟩.
On accède à la boite contenue dans un registre par \box⟨*registre*⟩.
Pour procéder à l'assignation, on écrit :

\setbox⟨*registre*⟩= ⟨*boite valide*⟩

où une ⟨*boite valide*⟩ est soit une boite contenue dans un registre et écrite sous la forme \box⟨*registre*⟩, soit une boite construite avec les primitives \hbox, \vbox ou \vtop. Le signe = et l'espace qui le suit sont facultatifs.
Pour afficher une boite stockée dans un ⟨*registre*⟩, on écrit \box⟨*registre*⟩. La boite est affichée selon le mode en cours (horizontal ou vertical) : le type de boite n'a pas d'influence sur le mode dans lequel elle est affichée.
Lorsqu'on utilise la primitive \box pour accéder à la boite stockée dans un registre, la boite contenue dans le registre est irrémédiablement perdue et le registre devient vide, c'est-à-dire qu'il ne contient aucune boite. Si on souhaite conserver la boite dans le registre, on doit utiliser la primitive \copy au lieu de \box.

76. Le test

\ifvoid⟨*registre*⟩⟨*code vrai*⟩\else⟨*code faux*⟩\fi

se comporte comme les autres tests de TEX. Il revoit ⟨*code vrai*⟩ si le ⟨*registre*⟩ de boite est vide (c'est-à-dire s'il ne contient aucune boite).

77. Les dimensions d'une boite stockée dans un ⟨*registre*⟩ sont :

- sa longueur horizontale, accessible par \wd⟨*registre*⟩ ;
- sa hauteur, qui s'étend au-dessus de la ligne de base, accessible par \ht⟨*registre*⟩ ;
- enfin, sa profondeur qui s'étend au-dessous de la ligne de base et qui est accessible par \dp⟨*registre*⟩.

Les primitives \wd, \ht et \dp suivies d'un ⟨*registre*⟩ se comportent comme des registres de dimension.

78. Les primitives \lower ou \raise doivent être suivies d'une ⟨*dimension*⟩, le tout devant précéder une boite horizontale, qu'elle soit écrite par l'intermédiaire de \hbox ou d'un registre. L'effet est d'abaisser pour \lower ou d'élever pour \raise la boite de la ⟨*dimension*⟩ spécifiée.

Les primitives \moveleft et \moveright partagent la même syntaxe, mais agissent sur des boites *verticales* en les déplaçant vers la gauche ou vers la droite.

79. On peut imposer une dimension aux boites en le spécifiant avec le mot-clé « to » suivi d'un espace optionnel puis d'une ⟨*dimension*⟩. Et donc, si l'on écrit

\hbox to 3cm{⟨*contenu*⟩}

cela créera une boite horizontale de longueur 3 cm, quel que soit le ⟨*contenu*⟩. Si le contenu ne mesure pas *exactement* 3 cm et ne contient aucun élément étirable, il y aura soit un débordement, soit un sous-remplissage de boite, chacun respectivement signalé par les messages d'avertissement « Overfull \hbox » ou « Underfull \hbox » dans le fichier log. La même syntaxe s'applique pour \vbox et \vtop sauf que la dimension est imposée dans le sens *vertical*.

On peut également écrire le mot-clé « spread » utilisé à la place de « to » si l'on souhaite construire une boite dont la dimension finale sera sa dimension naturelle augmentée de la ⟨*dimension*⟩. Si cette dimension est négative, la boite créée aura une dimension inférieure à sa dimension naturelle.

80. Une fois qu'une boite est stockée dans un ⟨*registre*⟩, il est possible de modifier une ou plusieurs de ses dimensions. Il suffit d'écrire

$$\left.\begin{matrix}\texttt{\wd}\\ \texttt{\ht}\\ \texttt{\dp}\end{matrix}\right\}\langle\textit{registre}\rangle\texttt{=}\ \langle\textit{dimension}\rangle$$

pour que la largeur, hauteur ou profondeur de la boite prenne la dimension indiquée, indépendamment de ce qu'est le contenu de la boite. Ces assignations sont toujours *globales*. En procédant de cette façon, le contenu reste intact, mais TEX est trompé et voit désormais la boite avec les nouvelles dimensions.

81. Si un ⟨*registre*⟩ de boite contient une boite horizontale ou verticale, il est possible d'extraire le contenu de la boite avec \unhbox⟨*registre*⟩ ou \unvbox⟨*registre*⟩. La liste horizontale ou verticale contenue dans la boite est alors ajoutée à la liste courante en cours. Après cette opération, le registre de boite devient vide, c'est à dire positif au sens de \ifvoid.

Pour ne pas vider le registre de boite, il faut employer \unhcopy ou \unvcopy.

82. Les primitives \hrule et \vrule, qui opèrent respectivement en mode *vertical* et *horizontal*, tracent des réglures. Par défaut, les dimensions de ces réglures sont :

	\hrule	\vrule
largeur (width)	*	0.4pt
hauteur (height)	0.4pt	*
profondeur (depth)	0pt	*

L'étoile signifie que la dimension est celle de la boite qui contient la réglure.
Si l'on souhaite passer outre les valeurs par défaut et imposer une ou plusieurs dimensions, on doit faire suivre ces deux primitives d'un (ou plusieurs) mots-clés parmi « width », « height » ou « depth » et en respectant cet ordre. S'ils sont présents, ces mots-clés doivent être suivis de la ⟨*dimension*⟩ que l'on souhaite imposer.
Si plusieurs spécifications contradictoires sur une dimension sont spécifiées, la dernière est prise en compte.

83. Lorsque TEX lit le code source, on peut faire lire à TEX le contenu d'un fichier. La lecture se fera comme se fait celle du code source et tout se passe donc comme si l'intégralité du contenu du fichier était insérée dans le code source à l'endroit où est rencontré :

\input ⟨*nom de fichier*⟩

Si le ⟨*nom de fichier*⟩ ne spécifie pas d'extension, TEX rajoute « .tex ».
La lecture du fichier s'arrête lorsque la fin du fichier est atteinte ou lorsque la primitive \endinput est atteinte.

84. Lorsque la primitive \outer précède la définition d'une macro, cette dernière ne pourra pas se situer dans les endroits où TEX lit du code « à grande vitesse », à savoir :
- le texte de remplacement ou le texte de paramètre lors de la définition d'une macro ;
- le code situé dans les branches d'un test ;
- le préambule d'un alignement initié par \halign ou \valign.

85. La fin d'un fichier a le statut \outer.
Pour permettre que la fin d'un fichier se situe dans un endroit normalement interdit à ce statut, il faut placer la primitive \noexpand juste avant la fin du fichier, soit en l'écrivant explicitement, soit via \everyeof avec l'assignation

\everyeof{\noexpand}

et faire en sorte de développer ce \noexpand.

86. TEX dispose de 15 canaux de lecture portant les numéros de 0 à 15.
On demande l'allocation d'un canal de lecture par

\newread\⟨*macro*⟩

et ce faisant, la \⟨*macro*⟩ devient équivalente à un ⟨*nombre*⟩, qui est un numéro de canal libre. Un ⟨*canal*⟩ est donc un ⟨*nombre*⟩ explicitement écrit ou une macro définie avec \newread.

Avant de lire dans un fichier, on doit d'abord lier un ⟨*canal*⟩ à un fichier avec :

\openin⟨*canal*⟩= ⟨*nom du fichier*⟩

Si aucune extension n'est précisée au ⟨*nom du fichier*⟩, TEX rajoute « .tex ». Le signe = et l'espace qui le suit sont optionnels. Lorsque TEX lit le ⟨*nom du fichier*⟩, il entre dans une phase de développement maximal et si ce nom est suivi d'un espace, cet espace est absorbé.

Tout canal ouvert doit être fermé et donc, après avoir fait les opérations de lecture, il convient d'exécuter \closein⟨*canal*⟩ pour désolidariser le ⟨*canal*⟩ du fichier qui lui avait été attaché avec \openin.

87. Lorsqu'un ⟨*canal*⟩ de lecture est ouvert et lié à un fichier, TEX peut lire des *lignes* dans ce fichier les unes après les autres, et assigner la ligne lue à une macro. On utilise la syntaxe

\read\⟨*canal*⟩ to \⟨*macro*⟩

où le mot-clé « to » est obligatoire et éventuellement suivi d'espaces optionnels.
Ceci a pour effet de placer dans le texte de remplacement de la \⟨*macro*⟩ tout ce qu'il y a dans la ligne en cours :

- les accolades doivent être équilibrées dans la ligne en cours. Si elles ne le sont pas, TEX lit autant de lignes que nécessaire pour qu'elles le soient. Si la fin du fichier est atteinte sans que l'équilibrage des accolades ne soit réalisé, une erreur de compilation surviendra ;
- une fois l'équilibrage des accolades atteint, la primitive \read lit tout ce qui s'étend jusqu'à la « marque de fin de ligne » (voir page 29) ;
- les tokens dans le texte de remplacement de la \⟨*macro*⟩ tiennent compte du régime de catcode en vigueur lorsque \read est exécutée ; \read effectue donc une « tokénisation », c'est-à-dire la transformation d'octets bruts en tokens ;
- le caractère de code \endlinechar est inséré à la fin du texte de remplacement de la\⟨*macro*⟩ ;
- l'assignation est globale si \read est précédé de \global.

La primitive de ε-TEX\readline, dont la syntaxe est

\readline\⟨*canal*⟩ to \⟨*macro*⟩

agit comme \read sauf qu'elle insère dans le texte de remplacement de la \⟨*macro*⟩ tous les tokens lus dans la ligne (y compris le caractère de code \endlinechar). L'espace aura le catcode 10 et tous les autres caractères le catcode 12.

88. TEX dispose de 15 canaux d'écriture portant les numéros de 0 à 15.
On demande l'allocation d'un canal d'écriture par

\newwrite\⟨*macro*⟩

et ce faisant, la \⟨*macro*⟩ devient équivalente à un ⟨*nombre*⟩, qui est un numéro de canal libre. Un ⟨*canal*⟩ est donc un ⟨*nombre*⟩ explicitement écrit ou une macro définie avec \newwrite.

Avant d'écrire dans un fichier, on doit d'abord lier un ⟨*canal*⟩ à un fichier avec :

\openout⟨*canal*⟩= ⟨*nom du fichier*⟩

et lorsque les opérations d'écriture sont terminées, on désolidarise le ⟨*canal*⟩ du fichier par

\closeout⟨*canal*⟩= ⟨*nom du fichier*⟩

Une ligne vide est insérée en fin de fichier.
Pour écrire une ligne dans le fichier, on écrit :

\write⟨*canal*⟩{⟨*texte*⟩}

où le ⟨*texte*⟩ est un code dans lequel les accolades sont équilibrées. L'écriture dans le fichier n'a pas lieu lorsque la commande \write est rencontrée, mais lorsque la page courante est composée. Le ⟨*texte*⟩ est stocké dans la mémoire de TEX et est développé au maximum lorsque TEX écrit dans le fichier. Si l'utilisateur souhaite bloquer le développement, il lui appartient de le bloquer à l'aide des méthodes vues à la page 119.

Les opérations liées à l'écriture dans un fichier (\openout, \write et \closeout) ont la particularité d'avoir lieu lorsque la page est composée, c'est-à-dire plus tard dans le temps que lorsque les instructions sont rencontrées. Ce décalage temporel est bien utile lorsqu'on souhaite écrire un numéro de page puisque l'on s'assure que le bon numéro de page sera écrit, mais il est indésirable pour écrire des données non sensibles au positionnement dans le document. Pour s'affranchir de cette désynchronisation temporelle, il faut faire précéder \openout, \write et \closeout de la primitive « \immediate » qui force les opérations d'écriture à être faites immédiatement lorsqu'elles sont rencontrées.

89. La primitive \afterassignment dont la syntaxe est

\afterassignment⟨*token*⟩

stocke le ⟨*token*⟩ dans un endroit spécial de la mémoire de TEX pour le placer sur la pile d'entrée immédiatement après la prochaine assignation. Cette assignation peut résulter de l'utilisation de primitives déjà vues \def, \gdef, \edef, \xdef, \let, \chardef, \mathchardef, ou bien d'une assignation à un registre (de boite, d'entier, de ressort, de dimension, de token), voire après une opération sur un registre d'entier ou de dimension par \advance, \multiply ou \divide.
Si plusieurs \afterassignment⟨*token*⟩ sont rencontrés avant une assignation, seul le dernier ⟨*token*⟩ est pris en compte.
Lorsque \afterassignment⟨*token*⟩ est appelé avant \setbox (c'est-à-dire avant l'assignation à un registre de boite), le ⟨*token*⟩ est placé en tout début de boite.

90. La primitive \futurelet a la syntaxe suivante :

\futurelet\⟨*macro*⟩⟨*token1*⟩⟨*token2*⟩

L'effet est le suivant :

1. TEX effectue l'action « \let\⟨*macro*⟩=␣⟨*token2*⟩ » (en figeant le catcode de ⟨*token2*⟩) ;
2. puis la tête de lecture avance jusqu'à ⟨*token 1*⟩⟨*token2*⟩ qui restent intacts, comme s'ils n'avaient pas été lus.

Il y a comme un air de ressemblance avec \expandafter sauf qu'il n'est pas question de développement, mais d'assignation via \let : il y a bien le « saut » de ⟨*token1*⟩ pour faire une assignation sans que la tête de lecture ne bouge. Tout se passe comme si TEX allait lire le ⟨*token2*⟩ pour le *copier* juste après \⟨*macro*⟩ pour effectuer une assignation avec un \let. Et donc, le code

\futurelet\⟨*macro*⟩⟨*token1*⟩⟨*token2*⟩

est équivalent à

\let\⟨*macro*⟩= ⟨*token2*⟩⟨*token1*⟩⟨*token2*⟩

91. Lorsqu'il est 1-développé, le code suivant

\romannumeral-‘\⟨*caractère*⟩

effectue l'action « tout développer au maximum jusqu'à trouver un token (absorbé si c'est un espace et laissé tel quel sinon) qui stoppera le développement ».

92. Si ⟨*x*⟩ est un registre de boite auquel on a assigné une boite verticale (construite avec \vbox ou \vtop) et si ⟨*y*⟩ est un numéro de registre de boite, alors

\setbox ⟨*y*⟩=\vsplit ⟨*x*⟩ to ⟨*dimension*⟩

va commander à TEX de parcourir la boite contenue dans le registre n° ⟨*x*⟩ et opérer la meilleure coupure possible pour générer une boite de hauteur ⟨*dimension*⟩. Le registre n° ⟨*y*⟩ contiendra cette nouvelle boite tandis que le registre ⟨*x*⟩ contiendra la boite initiale amputée d'autant. Si la ⟨*dimension*⟩ est suffisamment grande pour que la coupure englobe la totalité de la boite initiale ⟨*x*⟩, le registre ⟨*x*⟩ devient vide, c'est-à-dire positif au test \ifvoid.
Le mécanisme de coupure de boite verticale est similaire à celui des coupures de pages. En particulier, les objets « volatils » sont supprimés à la coupure (pénalités et espaces verticaux de type \kern ou \vskip).
Enfin, un ressort appelé \splittopskip est inséré au sommet de la boite restante ⟨*x*⟩.

93. La primitive \fontname permet d'accéder au nom externe d'un fichier de fonte selon la syntaxe

\fontname⟨*fonte*⟩

où ⟨*fonte*⟩ est soit une séquence de contrôle définie avec la primitive \font, soit la primitive \font elle-même auquel cas on fait référence à la fonte en cours d'utilisation. Le tout se développe en le nom du fichier externe correspondant à la fonte spécifiée.

94. La primitive \font permet de créer une séquence de contrôle qui, lorsqu'elle sera exécutée, effectuera un changement de fonte. On utilise la syntaxe

\font\⟨*macro*⟩=⟨*nom de fonte*⟩ at ⟨*dimension*⟩

où « at ⟨*dimension*⟩ » est facultatif.

95. Le caractère inséré aux coupures de mots est stocké dans un registre interne de type entier appelé \hyphenchar. Ce registre *doit* être suivi de la fonte à laquelle on souhaite se référer :

\hyphenchar⟨*fonte*⟩

On peut à tout moment choisir un autre caractère de coupure par son code de caractère en modifiant \hyphenchar :

\hyphenchar⟨*fonte*⟩=⟨*code de caractère*⟩

Si le ⟨*code de caractère*⟩ contenu dans \hyphenchar est négatif ou est supérieur à 255, aucun caractère n'est sélectionné et cela désactive les coupures de mots.
L'assignation d'un entier à \hyphenchar est toujours globale.

96. Pour composer du texte, chaque fonte dispose de sept dimensions propres contenues dans des registres de dimension spéciaux accessibles via la primitive \fontdimen. Chacun de ces registres est accessible par

\fontdimen⟨*entier*⟩⟨*fonte*⟩

où ⟨*entier*⟩ est le numéro de la dimension (compris entre 1 et 7) et dont la signification est la suivante :

- n° 1 pente par point (dimension utilisée pour placer les accents) ;
- n° 2 espace inter-mot (dimension naturelle de l'espace) ;
- n° 3 étirement inter-mot ;
- n° 4 compression inter-mot ;
- n° 5 x-height (valeur de 1ex) ;
- n° 6 cadrat (valeur de 1em) ;
- n° 7 espace supplémentaire (espace ajouté en fin des phrases).

97. La primitive \lastbox a un comportement très particulier :

- non seulement elle contient la dernière boite composée dans la liste en cours, mais utiliser \lastbox *retire* cette boite de la liste dans laquelle elle était placée ;
- si le dernier élément n'est pas une boite, \lastbox est vide ;
- si l'on souhaite accéder à la dernière boite composée, \lastbox s'utilise seule. Il est incorrect de la faire précéder de \box.

L'opération \lastbox n'est pas permise dans le mode mathématique ni dans le mode vertical *principal* : elle ne peut donc pas servir à retirer une boite de la page courante en cours de composition. En revanche, elle peut très bien être utilisée dans le mode vertical interne.

La boite \lastbox hérite de la nature – horizontale ou verticale – de la dernière boite composée.

Si un registre de boite est vidé par \lastbox, le registre ne devient pas vide, mais contient une boite vide (le test \ifvoid est donc inadapté pour tester si un registre est vidé par \lastbox).

Bibliographie

1. D.E. Knuth. Le TEXbook, Vuibert, 2003.
   http://www.vuibert.fr/ouvrage-9782711748198-le-texbook.html
2. D.E. Knuth. TEX The Program.
   http://tug.org/texlive/devsrc/Build/source/texk/web2c/tex.web
3. V. Eijkhout. TEX by Topic, Addison-Wesley, 1992.
   http://www.eijkhout.net/texbytopic/texbytopic.html
4. P.W. Abrahams & K. Hargreaves & K. Berry. TEX pour l'impatient.
   http://ctan.mirrorcatalogs.com/info/impatient/fr/fbook.pdf
5. J. Braams & D. Carlisle & A. Jeffrey & L. Lamport & F.Mittelbach & C. Rowley & R. Schöpf. The LATEX 2_ε sources
   http://www.tug.org/texlive/Contents/live/texmf-dist/doc/latex/base/source2e.pdf
6. E. Gregorio. Appunti di programmazione in LATEX e TEX.
   http://profs.scienze.univr.it/~gregorio/introtex.pdf

Index général

L'index ci-dessous liste les commandes et les notions abordées dans le livre. Les commandes, qu'elles soient primitives ou macros, sont classées alphabétiquement sans tenir compte du caractère d'échappement « \ ». Les commandes précédées d'une étoile sont des primitives.

Certaines séquences de contrôle ont été très fréquemment employées dans les codes exposés dans ce livre. On trouve par exemple `\def`, `\expandafter`, `\catcode`, etc. Afin de ne pas inutilement surcharger l'index, seuls les numéros de pages des premières apparitions ont été retenus. Ils sont suivis par « … » qui symbolise les nombreux autres numéros de page qui ont été omis.

Lorsqu'un renvoi « *voir* » à une macro définie dans ce livre figure dans cet index, cette macro est suivie de « ⋆ ». L'index des macros définies dans ce livre se trouve à la page 559.

F

G

H

Index des macros définies dans ce livre

L'index ci-dessous liste les macros définies dans ce livre. Toutes n'y figurent pas. À de rares exceptions près, seules les macros *publiques*, c'est-à-dire sans « @ », y figurent.

E

F

G

H

I

Christian Tellechea
—
Dépôt légal : septembre 2014