xstring

v1.5b

Manuel de l'utilisateur

Christian Tellechea
unbonpetit@gmail.com

13 mars 2009

Résumé

Cette extension, qui requiert Plain ε-TEX, regroupe un ensemble de macros manipulant des chaînes de tokens (ou lexèmes en français). Les macros peuvent être utilisées de façon basique dans le traitement des chaînes de caractères mais peuvent également être utiles en programmation TEX pour la manipulation des tokens, c'est-à-dire du code TEX. Parmis les fonctionnalités, les principales sont :

- ▷ des tests :
  - une chaîne en contient elle une autre au moins n fois ?
  - une chaîne commence t-elle ou finit-elle par une autre ? etc.
  - une chaîne représente t-elle un entier relatif ? Un nombre décimal ?
  - deux chaînes sont-elles égales ?
- ▷ des recherches de chaînes :
  - recherche de ce qui se trouve avant (ou après) la n^{e} occurrence d'une sous-chaîne ;
  - recherche de ce qui se trouve entre les occurrences de 2 sous-chaînes ;
  - sous-chaîne comprise entre 2 positions ;
  - recherche d'un groupe entre accolades par son identifiant.
- ▷ le remplacement de toutes ou des n premières occurrences d'une sous-chaîne par une autre sous-chaîne ;
- ▷ des calculs de nombres :
  - longueur d'une chaîne ;
  - position de la la n^{e} occurrence d'une sous-chaîne ;
  - comptage du nombre d'occurrences d'une sous-chaîne dans une autre ;
  - position de la 1^{re} différence entre 2 chaînes ;
  - renvoi de l'identifiant du groupe dans lequel une recherche ou une coupure s'est faite.

D'autres commandes permettent de gérer les tokens spéciaux normalement interdits dans les chaînes (`#` et `%`), ainsi que d'éventuelles différences entre catcodes de tokens, ce qui devrait permettre de couvrir tous les besoins en matière de programmation.

Table des matières

1 Présentation

1.1 Description

Cette extension [1] regroupe des macros et des tests opérant sur des chaînes de tokens, un peu comme en disposent des langages dit « évolués ». On y trouve les opérations habituelles sur les chaînes, comme par exemple : test si une chaîne en contient une autre, commence ou finit par une autre, test si une chaîne est un nombre entier ou décimal, extractions de sous-chaînes, calculs de position d'une sous-chaîne, calculs du nombre d'occurrences, etc.

On appelle « chaîne de tokens » une suite de tokens quelconques, sachant qu'aucune supposition n'a été faite quant à leur nature, mis à part que dans les chaînes de tokens, les accolades doivent être équilibrées et que les tokens de catcode 6 et 14 (habituellement `%` et `#`) n'y sont pas admis. Tous les autres tokens sont *à priori* permis dans n'importe quel ordre, quelque soit le code qu'ils représentent.

Les arguments contenant des chaînes de tokens sont lus par xstring *unité syntaxique par unité syntaxique* [2], ce qui revient à les lire caractère par caractère lorsque ceux-ci contiennent des tokens « normaux », c'est-à-dire dont les catcodes sont 10, 11 et 12. On peut également utiliser xstring à des fins de programmation en utilisant des arguments contenant des séquences de contrôle et des tokens dont les catcodes sont moins inoffensifs. Voir le chapitre sur le mode de lecture et de développement des arguments (page 13), la commande `\verbtocs` (page 16), la commande `\scancs` (page 17).

Certes d'autres packages manipulant les chaînes de caractères existent (par exemple `substr` et `stringstrings`), mais outre des différences notables quant aux fonctionnalités, ils ne prennent pas en charge les occurrences des sous-chaînes et me semblent soit trop limités, soit trop difficiles à utiliser pour la programmation.

Comme les macros manipulent des chaînes de tokens, il peut arriver aux utilisateurs avancés de rencontrer des problèmes de « catcodes [3] » conduisant à des comportements inattendus. Ces effets indésirables peuvent être contrôlés. Consulter en particulier le chapitre sur les catcodes des arguments page 14.

1.2 Motivation

J'ai été conduit à écrire ce type de macros car je n'ai jamais vraiment trouvé de d'outils sous LaTeX adaptés à mes besoins concernant le traitement de chaînes. Alors, au fil des mois, et avec l'aide de contributeurs [4] de `fr.comp.text.tex`, j'ai écrit quelques macros qui me servaient ponctuellement ou régulièrement. Leur nombre s'étant accru, et celles-ci devenant un peu trop dispersées dans les répertoires de mon ordinateur, je les ai regroupées dans ce package.
Ainsi, le fait de donner corps à un ensemble cohérent de macros force à davantage de rigueur et induit naturellement de nécessaires améliorations, ce qui a pris la majeure partie du temps que j'ai consacré à ce package. Pour harmoniser le tout, mais à contre-cœur, j'ai fini par choisir des noms de macros à consonances anglo-saxonnes.

Ensuite, et cela a été ma principale motivation puisque j'ai découvert LaTeX récemment [5], l'écriture de xstring qui est mon premier package m'a surtout permis de beaucoup progresser en programmation pure, et aborder des méthodes propres à la programmation sous TeX.

2 Les macros

Pour bien comprendre les actions de chaque macro, envisageons tout d'abord le fonctionnement et la présentation des macros dans leur mode de fonctionnement le plus simple. Pas de problème de catcode ici, ni de tokens spéciaux et encore moins de séquence de contrôle dans les arguments : les arguments contiendront des caractères alphanumériques.

Dans ce chapitre, la totalité des macros est présentée selon ce plan :

- la syntaxe complète [6] ainsi que la valeur d'éventuels arguments optionnels ;
- une brève description du fonctionnement ;
- le fonctionnement sous certaines conditions particulières. Pour chaque conditions envisagée, le fonctionnement décrit est prioritaire sur celui (ceux) se trouvant au dessous de lui ;

1. L'extension ne nécessite pas LaTeX et peut être compilée sous Plain ε-TeX.
2. Sauf cas particulier, une unité syntaxique est un caractère lu dans le code à ces exceptions près : une séquence de contrôle est une unité syntaxique, un groupe entre accolades est aussi une unité syntaxique. Voir également page 13.
3. Codes de catégories, en français.
4. Je remercie chaleureusement Manuel alias « `mpg` » pour son aide précieuse, sa compétence et sa disponibilité.
5. En novembre 2007, je suis donc un « noob » pour longtemps encore !
6. L'étoile optionnelle après le nom de la macro, et l'argument optionnel entre crochet venant en dernier seront expliqués plus tard. Voir page 14 pour les macros étoilées et page 12 pour l'argument optionnel en dernière position.

– enfin, quelques exemples sont donnés. J'ai essayé de les trouver les plus facilement compréhensibles et les plus représentatifs des situations rencontrées dans une utilisation normale[7]. Si un doute est possible quant à la présence d'espaces dans le résultat, celui-ci sera délimité par des « | », étant entendu qu'une chaîne vide est représentée par « || ».

Important : dans les macros qui suivent, sauf cas spécifié, un ⟨*nombre*⟩ est un nombre entier, un compteur, ou le résultat d'une opération arithmétique effectuée à l'aide de la primitive `\numexpr`.

Dans le reste du texte, les macros de xstring sont affichées en rouge.

2.1 Les tests

2.1.1 `\IfSubStr`

`\IfSubStr`⟨`[*]`⟩`[`⟨*nombre*⟩`]{`⟨*chaine*⟩`}{`⟨*chaineA*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}`

L'argument optionnel ⟨*nombre*⟩ vaut 1 par défaut.

Teste si ⟨*chaine*⟩ contient au moins ⟨*nombre*⟩ fois ⟨*chaineA*⟩ et exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

▷ Si ⟨*nombre*⟩ ⩽ 0, exécute ⟨*faux*⟩ ;
▷ Si ⟨*chaine*⟩ ou ⟨*chaineA*⟩ est vide, exécute ⟨*faux*⟩.

```
1 \IfSubStr{xstring}{tri}{vrai}{faux}
2 \IfSubStr{xstring}{a}{vrai}{faux}
3 \IfSubStr{a bc def }{c d}{vrai}{faux}
4 \IfSubStr{a bc def }{cd}{vrai}{faux}
5 \IfSubStr[2]{1a2a3a}{a}{vrai}{faux}
6 \IfSubStr[3]{1a2a3a}{a}{vrai}{faux}
7 \IfSubStr[4]{1a2a3a}{a}{vrai}{faux}
```

vrai
faux
vrai
faux
vrai
vrai
faux

2.1.2 `\IfSubStrBefore`

`\IfSubStrBefore`⟨`[*]`⟩`[`⟨*nombre1*⟩`,`⟨*nombre2*⟩`]{`⟨*chaine*⟩`}{`⟨*chaineA*⟩`}{`⟨*chaineB*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}`

Les arguments optionnels ⟨*nombre1*⟩ et ⟨*nombre2*⟩ valent 1 par défaut.

Dans ⟨*chaine*⟩, la macro teste si l'occurrence n° ⟨*nombre1*⟩ de ⟨*chaineA*⟩ se trouve à gauche de l'occurrence n° ⟨*nombre2*⟩ de ⟨*chaineB*⟩. Exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

▷ Si l'une des occurrences n'est pas trouvée, exécute ⟨*faux*⟩ ;
▷ Si l'un des arguments ⟨*chaine*⟩, ⟨*chaineA*⟩ ou ⟨*chaineB*⟩ est vide, exécute ⟨*faux*⟩ ;
▷ Si l'un au moins des deux arguments optionnels est négatif ou nul, exécute ⟨*faux*⟩.

```
1 \IfSubStrBefore{xstring}{st}{in}{vrai}{faux}
2 \IfSubStrBefore{xstring}{ri}{s}{vrai}{faux}
3 \IfSubStrBefore{LaTeX}{LaT}{TeX}{vrai}{faux}
4 \IfSubStrBefore{a bc def }{ b}{ef}{vrai}{faux}
5 \IfSubStrBefore{a bc def }{ab}{ef}{vrai}{faux}
6 \IfSubStrBefore[2,1]{b1b2b3}{b}{2}{vrai}{faux}
7 \IfSubStrBefore[3,1]{b1b2b3}{b}{2}{vrai}{faux}
8 \IfSubStrBefore[2,2]{baobab}{a}{b}{vrai}{faux}
9 \IfSubStrBefore[2,3]{baobab}{a}{b}{vrai}{faux}
```

vrai
faux
faux
vrai
faux
vrai
faux
faux
vrai

2.1.3 `\IfSubStrBehind`

`\IfSubStrBehind`⟨`[*]`⟩`[`⟨*nombre1*⟩`,`⟨*nombre2*⟩`]{`⟨*chaine*⟩`}{`⟨*chaineA*⟩`}{`⟨*chaineB*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}`

Les arguments optionnels ⟨*nombre1*⟩ et ⟨*nombre2*⟩ valent 1 par défaut.

Dans ⟨*chaine*⟩, la macro teste si l'occurrence n° ⟨*nombre1*⟩ de ⟨*chaineA*⟩ se trouve après l'occurrence n° ⟨*nombre2*⟩ de ⟨*chaineB*⟩. Exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

▷ Si l'une des occurrences n'est pas trouvée, exécute ⟨*faux*⟩ ;
▷ Si l'un des arguments ⟨*chaine*⟩, ⟨*chaineA*⟩ ou ⟨*chaineB*⟩ est vide, exécute ⟨*faux*⟩ ;
▷ Si l'un au moins des deux arguments optionnels est négatif ou nul, exécute ⟨*faux*⟩.

7. Pour une collection plus importante d'exemples, on peut aussi consulter le fichier de test.

```
\IfSubStrBehind{xstring}{ri}{xs}{vrai}{faux}
\IfSubStrBehind{xstring}{s}{i}{vrai}{faux}
\IfSubStrBehind{LaTeX}{TeX}{LaT}{vrai}{faux}
\IfSubStrBehind{a bc def }{ d}{a}{vrai}{faux}
\IfSubStrBehind{a bc def }{cd}{a b}{vrai}{faux}
\IfSubStrBehind[2,1]{b1b2b3}{b}{2}{vrai}{faux}
\IfSubStrBehind[3,1]{b1b2b3}{b}{2}{vrai}{faux}
\IfSubStrBehind[2,2]{baobab}{b}{a}{vrai}{faux}
\IfSubStrBehind[2,3]{baobab}{b}{a}{vrai}{faux}
```

faux
faux
faux
faux
faux
faux
vrai
faux
faux

2.1.4 \IfBeginWith

\IfBeginWith⟨[*]⟩{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

Teste si ⟨*chaine*⟩ commence par ⟨*chaineA*⟩, et exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

▷ Si ⟨*chaine*⟩ ou ⟨*chaineA*⟩ est vide, exécute ⟨*faux*⟩.

```
\IfBeginWith{xstring}{xst}{vrai}{faux}
\IfBeginWith{LaTeX}{a}{vrai}{faux}
\IfBeginWith{a bc def }{a b}{vrai}{faux}
\IfBeginWith{a bc def }{ab}{vrai}{faux}
```

vrai
faux
vrai
faux

2.1.5 \IfEndWith

\IfEndWith⟨[*]⟩{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

Teste si ⟨*chaine*⟩ se termine par ⟨*chaineA*⟩, et exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

▷ Si ⟨*chaine*⟩ ou ⟨*chaineA*⟩ est vide, exécute ⟨*faux*⟩.

```
\IfEndWith{xstring}{ring}{vrai}{faux}
\IfEndWith{LaTeX}{a}{vrai}{faux}
\IfEndWith{a bc def }{ef }{vrai}{faux}
\IfEndWith{a bc def }{ef}{vrai}{faux}
```

vrai
faux
vrai
faux

2.1.6 \IfInteger

\IfInteger{⟨*nombre*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

Teste si ⟨*nombre*⟩ est un nombre entier relatif (c'est-à-dire dont la partie décimale est absente ou constituée d'une suite de 0), et exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

Si le test est faux pour cause de caractères non autorisés, la séquence de contrôle `\@xs@afterinteger` contient la partie illégale de ⟨*nombre*⟩.

```
\IfInteger{13}{vrai}{faux}
\IfInteger{-219}{vrai}{faux}
\IfInteger{+9}{vrai}{faux}
\IfInteger{3.14}{vrai}{faux}
\IfInteger{8.0}{vrai}{faux}
\IfInteger{0}{vrai}{faux}
\IfInteger{49a}{vrai}{faux}
\IfInteger{+}{vrai}{faux}
\IfInteger{-}{vrai}{faux}
\IfInteger{0000}{vrai}{faux}
```

vrai
vrai
vrai
faux
vrai
vrai
faux
faux
faux
vrai

2.1.7 \IfDecimal

\IfDecimal{⟨*nombre*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}

Teste si ⟨*nombre*⟩ est un nombre décimal, et exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

Les compteurs `\integerpart` et `\decimalpart` contiennent les parties entières et décimales de ⟨*nombre*⟩.

Si le test est faux pour cause de caractères non autorisés, la séquence de contrôle `\@xs@afterdecimal` contient la partie illégale de ⟨*nombre*⟩, alors que si le test est faux parce que la partie décimale après le séparateur décimal est vide, elle contient « X ».

- ▷ Le séparateur décimal peut être un point ou une virgule ;
- ▷ Si le ⟨*nombre*⟩ commence par un séparateur décimal, le test est vrai (lignes 4 et 5) et la partie entière est considérée comme étant 0 ;
- ▷ Si le ⟨*nombre*⟩ se termine par un séparateur décimal, le test est faux (lignes 9 et 10).

```
1  \IfDecimal{3.14}{vrai}{faux}
2  \IfDecimal{3,14}{vrai}{faux}
3  \IfDecimal{-0.5}{vrai}{faux}
4  \IfDecimal{.7}{vrai}{faux}
5  \IfDecimal{,9}{vrai}{faux}
6  \IfDecimal{1..2}{vrai}{faux}
7  \IfDecimal{+6}{vrai}{faux}
8  \IfDecimal{-15}{vrai}{faux}
9  \IfDecimal{1.}{vrai}{faux}
10 \IfDecimal{2,}{vrai}{faux}
11 \IfDecimal{.}{vrai}{faux}
12 \IfDecimal{,}{vrai}{faux}
13 \IfDecimal{+}{vrai}{faux}
14 \IfDecimal{-}{vrai}{faux}
```

vrai
vrai
vrai
vrai
vrai
faux
vrai
vrai
faux
faux
faux
faux
faux
faux

2.1.8 \IfStrEq

`\IfStrEq`⟨`[*]`⟩`{`⟨*chaineA*⟩`}{`⟨*chaineB*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}`

Teste si les chaînes ⟨*chaineA*⟩ et ⟨*chaineB*⟩ sont égales, c'est-à-dire si elles contiennent successivement les mêmes unité syntaxique dans le même ordre. Exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

```
1 \IfStrEq{a1b2c3}{a1b2c3}{vrai}{faux}
2 \IfStrEq{abcdef}{abcd}{vrai}{faux}
3 \IfStrEq{abc}{abcdef}{vrai}{faux}
4 \IfStrEq{3,14}{3,14}{vrai}{faux}
5 \IfStrEq{12.34}{12.340}{vrai}{faux}
6 \IfStrEq{abc}{}{vrai}{faux}
7 \IfStrEq{}{abc}{vrai}{faux}
8 \IfStrEq{}{}{vrai}{faux}
```

vrai
faux
faux
vrai
faux
faux
faux
vrai

2.1.9 \IfEq

`\IfEq`⟨`[*]`⟩`{`⟨*chaineA*⟩`}{`⟨*chaineB*⟩`}{`⟨*vrai*⟩`}{`⟨*faux*⟩`}`

Teste si les chaînes ⟨*chaineA*⟩ et ⟨*chaineB*⟩ sont égales, *sauf* si ⟨*chaineA*⟩ et ⟨*chaineB*⟩ contiennent des nombres, auquel cas la macro teste si les nombres sont égaux. Exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

- ▷ La définition de *nombre* est celle évoquée dans la macro `\IfDecimal` (voir page 4), et donc :
- ▷ Les signes « + » sont facultatifs ;
- ▷ Le séparateur décimal peut être indifféremment la virgule ou le point.

```
1  \IfEq{a1b2c3}{a1b2c3}{vrai}{faux}
2  \IfEq{abcdef}{ab}{vrai}{faux}
3  \IfEq{ab}{abcdef}{vrai}{faux}
4  \IfEq{12.34}{12,34}{vrai}{faux}
5  \IfEq{+12.34}{12.340}{vrai}{faux}
6  \IfEq{10}{+10}{vrai}{faux}
7  \IfEq{-10}{10}{vrai}{faux}
8  \IfEq{+0,5}{,5}{vrai}{faux}
9  \IfEq{1.001}{1.01}{vrai}{faux}
10 \IfEq{3*4+2}{14}{vrai}{faux}
11 \IfEq{\number\numexpr3*4+2}{14}{vrai}{faux}
12 \IfEq{0}{-0.0}{vrai}{faux}
13 \IfEq{}{}{vrai}{faux}
```

vrai
faux
faux
vrai
vrai
vrai
faux
vrai
faux
faux
vrai
vrai
vrai

2.1.10 \IfStrEqCase

\IfStrEqCase⟨[*]⟩{⟨*chaine*⟩}{%
{⟨*chaine1*⟩}{⟨*code1*⟩}%
{⟨*chaine2*⟩}{⟨*code2*⟩}%
etc...
{⟨*chaineN*⟩}{⟨*codeN*⟩}}[⟨*code alternatif*⟩]

Teste successivement si ⟨*chaine*⟩ est égale à ⟨*chaine1*⟩, ⟨*chaine2*⟩, etc. La comparaison se fait au sens de \IfStrEq (voir paragraphes précédents). Si un test est positif, le code correspondant est exécuté et la macro se termine. Si tous les tests sont négatifs, le code optionnel ⟨*code alternatif*⟩ est exécuté s'il est présent.

```
\IfStrEqCase{b}{{a}{AA}{b}{BB}{c}{CC}}
|\IfStrEqCase{abc}{{a}{AA}{b}{BB}{c}{CC}}|
\IfStrEqCase{c}{{a}{AA}{b}{BB}{c}{CC}}[autre]
\IfStrEqCase{d}{{a}{AA}{b}{BB}{c}{CC}}[autre]
\IfStrEqCase{+3}{{1}{un}{2}{deux}{3}{trois}}[autre]
\IfStrEqCase{0.5}{{0}{zero}{.5}{demi}{1}{un}}[autre]
```

BB
||
CC
autre
autre
autre

2.1.11 \IfEqCase

\IfEqCase⟨[*]⟩{⟨*chaine*⟩}{%
{⟨*chaine1*⟩}{⟨*code1*⟩}%
{⟨*chaine2*⟩}{⟨*code2*⟩}%
etc...
{⟨*chaineN*⟩}{⟨*codeN*⟩}}[⟨*code alternatif*⟩]

Teste successivement si ⟨*chaine*⟩ est égale à ⟨*chaine1*⟩, ⟨*chaine2*⟩, etc. La comparaison se fait au sens de \IfEq (voir paragraphes précédents). Si un test est positif, le code correspondant est exécuté et la macro se termine. Si tous les tests sont négatifs, le code optionnel ⟨*code alternatif*⟩ est exécuté s'il est présent.

```
\IfEqCase{b}{{a}{AA}{b}{BB}{c}{CC}}
|\IfEqCase{abc}{{a}{AA}{b}{BB}{c}{CC}}|
\IfEqCase{c}{{a}{AA}{b}{BB}{c}{CC}}[autre]
\IfEqCase{d}{{a}{AA}{b}{BB}{c}{CC}}[autre]
\IfEqCase{+3}{{1}{un}{2}{deux}{3}{trois}}[autre]
\IfEqCase{0.5}{{0}{zero}{.5}{demi}{1}{un}}[autre]
```

BB
||
CC
autre
trois
demi

2.2 Les macros renvoyant une chaîne

2.2.1 \StrBefore

\StrBefore⟨[*]⟩[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]

L'argument optionnel ⟨*nombre*⟩ vaut 1 par défaut.

Dans ⟨*chaine*⟩, renvoie ce qui se trouve avant l'occurrence n° ⟨*nombre*⟩ de ⟨*chaineA*⟩.

- ▷ Si ⟨*chaine*⟩ ou ⟨*chaineA*⟩ est vide, une chaîne vide est renvoyée ;
- ▷ Si ⟨*nombre*⟩ < 1 alors, la macro se comporte comme si ⟨*nombre*⟩ $= 1$;
- ▷ Si l'occurrence n'est pas trouvée, une chaîne vide est renvoyée.

```
\StrBefore{xstring}{tri}
\StrBefore{LaTeX}{e}
|\StrBefore{LaTeX}{p}|
|\StrBefore{LaTeX}{L}|
|\StrBefore{a bc def }{def}|
|\StrBefore{a bc def }{cd}|
\StrBefore[1]{1b2b3}{b}
\StrBefore[2]{1b2b3}{b}
```

xs
LaT
||
||
|a bc |
||
1
1b2

2.2.2 \StrBehind

\StrBehind⟨[*]⟩[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]

L'argument optionnel ⟨*nombre*⟩ vaut 1 par défaut.

Dans ⟨*chaine*⟩, renvoie ce qui se trouve après l'occurrence n° ⟨*nombre*⟩ de ⟨*chaineA*⟩.

- ▷ Si ⟨*chaine*⟩ ou ⟨*chaineA*⟩ est vide, une chaîne vide est renvoyée ;
- ▷ Si ⟨*nombre*⟩ < 1 alors, la macro se comporte comme si ⟨*nombre*⟩ = 1 ;
- ▷ Si l'occurrence n'est pas trouvée, une chaîne vide est renvoyée.

```
\StrBehind{xstring}{tri}
\StrBehind{LaTeX}{e}
|\StrBehind{LaTeX}{p}|
|\StrBehind{LaTeX}{X}|
|\StrBehind{a bc def }{bc}|
|\StrBehind{a bc def }{cd}|
\StrBehind[1]{1b2b3}{b}
\StrBehind[2]{1b2b3}{b}
|\StrBehind[3]{1b2b3}{b}|
```

```
ng
X
||
||
| def |
||
2b3
3
||
```

2.2.3 \StrBetween

\StrBetween⟨[*]⟩[⟨*nombre1*⟩,⟨*nombre2*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}[⟨*nom*⟩]

Les arguments optionnels ⟨*nombre1*⟩ et ⟨*nombre2*⟩ valent 1 par défaut.

Dans ⟨*chaine*⟩, renvoie ce qui se trouve entre[8] les occurrences n° ⟨*nombre1*⟩ de ⟨*chaineA*⟩ et n° ⟨*nombre2*⟩ de ⟨*chaineB*⟩.

- ▷ Si les occurrences ne sont pas dans l'ordre (⟨*chaineA*⟩ *puis* ⟨*chaineB*⟩) dans ⟨*chaine*⟩, une chaîne vide est renvoyée ;
- ▷ Si l'une des 2 occurrences n'existe pas dans ⟨*chaine*⟩, une chaîne vide est renvoyée ;
- ▷ Si l'un des arguments optionnels ⟨*nombre1*⟩ ou ⟨*nombre2*⟩ est négatif ou nul, une chaîne vide est renvoyée.

```
\StrBetween{xstring}{xs}{ng}
|\StrBetween{xstring}{i}{n}|
|\StrBetween{xstring}{a}{tring}|
|\StrBetween{a bc def }{a}{d}|
|\StrBetween{a bc def }{a }{f}|
\StrBetween{a1b1a2b2a3b3}{a}{b}
\StrBetween[2,3]{a1b1a2b2a3b3}{a}{b}
\StrBetween[1,3]{a1b1a2b2a3b3}{a}{b}
|\StrBetween[3,1]{a1b1a2b2a3b3}{a}{b}|
\StrBetween[3,2]{abracadabra}{a}{bra}
```

```
tri
||
||
| bc |
|bc de|
1
2b2a3
1b1a2b2a3
||
da
```

2.2.4 \StrSubstitute

\StrSubstitute[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}[⟨*nom*⟩]

L'argument optionnel ⟨*nombre*⟩ vaut 0 par défaut.

Dans ⟨*chaine*⟩, la macro remplace les ⟨*nombre*⟩ premières occurrences de ⟨*chaineA*⟩ par ⟨*chaineB*⟩, sauf si ⟨*nombre*⟩ = 0 auquel cas, *toutes* les occurrences sont remplacées.

- ▷ Si ⟨*chaine*⟩ est vide, une chaîne vide est renvoyée ;
- ▷ Si ⟨*chaineA*⟩ est vide ou n'existe pas dans ⟨*chaine*⟩, la macro est sans effet ;
- ▷ Si ⟨*nombre*⟩ est supérieur au nombre d'occurrences de ⟨*chaineA*⟩, alors toutes les occurrences sont remplacées ;
- ▷ Si ⟨*nombre*⟩ < 0 alors la macro se comporte comme si ⟨*nombre*⟩ = 0 ;
- ▷ Si ⟨*chaineB*⟩ est vide, alors les occurrences de ⟨*chaineA*⟩, si elles existent, sont supprimées.

8. Au sens strict, c'est-à-dire *sans* les chaînes frontière

```
\StrSubstitute{xstring}{i}{a}
\StrSubstitute{abracadabra}{a}{o}
\StrSubstitute{abracadabra}{br}{TeX}
\StrSubstitute{LaTeX}{m}{n}
\StrSubstitute{a bc def }{ }{M}
\StrSubstitute{a bc def }{ab}{AB}
\StrSubstitute[1]{a1a2a3}{a}{B}
\StrSubstitute[2]{a1a2a3}{a}{B}
\StrSubstitute[3]{a1a2a3}{a}{B}
\StrSubstitute[4]{a1a2a3}{a}{B}
```

```
xstrang
obrocodobro
aTeXacadaTeXa
LaTeX
aMbcMdefM
a bc def
B1a2a3
B1B2a3
B1B2B3
B1B2B3
```

2.2.5 \StrDel

\StrDel[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]

L'argument optionnel ⟨*nombre*⟩ vaut 0 par défaut.

Supprime les ⟨*nombre*⟩ premières occurrences de ⟨*chaineA*⟩ dans ⟨*chaine*⟩, sauf si ⟨*nombre*⟩ = 0 auquel cas, *toutes* les occurrences sont supprimées.

- ▷ Si ⟨*chaine*⟩ est vide, une chaîne vide est renvoyée ;
- ▷ Si ⟨*chaineA*⟩ est vide ou n'existe pas dans ⟨*chaine*⟩, la macro est sans effet ;
- ▷ Si ⟨*nombre*⟩ est supérieur au nombre d'occurrences de ⟨*chaineA*⟩, alors toutes les occurrences sont supprimées ;
- ▷ Si ⟨*nombre*⟩ < 0 alors la macro se comporte comme si ⟨*nombre*⟩ = 0 ;

```
\StrDel{abracadabra}{a}
\StrDel[1]{abracadabra}{a}
\StrDel[4]{abracadabra}{a}
\StrDel[9]{abracadabra}{a}
\StrDel{a bc def }{ }
|\StrDel{a bc def }{def}|
```

```
brcdbr
bracadabra
brcdbra
brcdbr
abcdef
|a bc  |
```

2.2.6 \StrGobbleLeft

\StrGobbleLeft{⟨*chaine*⟩}{⟨*nombre*⟩}[⟨*nom*⟩]

Dans ⟨*chaine*⟩, enlève les ⟨*nombre*⟩ premieres unités syntaxiques de gauche.

- ▷ Si ⟨*chaine*⟩ est vide, renvoie une chaîne vide ;
- ▷ Si ⟨*nombre*⟩ ⩽ 0, aucune unité syntaxique n'est supprimée ;
- ▷ Si ⟨*nombre*⟩ ⩾ ⟨*longueurChaine*⟩, toutes les unités syntaxiques sont supprimées.

```
\StrGobbleLeft{xstring}{2}
|\StrGobbleLeft{xstring}{9}|
\StrGobbleLeft{LaTeX}{4}
\StrGobbleLeft{LaTeX}{-2}
|\StrGobbleLeft{a bc def }{4}|
```

```
tring
||
X
LaTeX
| def |
```

2.2.7 \StrLeft

\StrLeft{⟨*chaine*⟩}{⟨*nombre*⟩}[⟨*nom*⟩]

Dans ⟨*chaine*⟩, renvoie la sous-chaîne de gauche de longueur ⟨*nombre*⟩.

- ▷ Si ⟨*chaine*⟩ est vide, renvoie une chaîne vide ;
- ▷ Si ⟨*nombre*⟩ ⩽ 0, aucune unité syntaxique n'est retournée ;
- ▷ Si ⟨*nombre*⟩ ⩾ ⟨*longueurChaine*⟩, toutes les unités syntaxiques sont retournées.

```
\StrLeft{xstring}{2}
\StrLeft{xstring}{9}
\StrLeft{LaTeX}{4}
|\StrLeft{LaTeX}{-2}|
|\StrLeft{a bc def }{5}|
```

```
xs
xstring
LaTe
||
|a bc |
```

2.2.8 \StrGobbleRight

\StrGobbleRight{⟨*chaine*⟩}{⟨*nombre*⟩}[⟨*nom*⟩]

Agit comme \StrGobbleLeft, mais enlève les unités syntaxiques à droite de ⟨*chaine*⟩.

```
\StrGobbleRight{xstring}{2}
|\StrGobbleRight{xstring}{9}|
\StrGobbleRight{LaTeX}{4}
|\StrGobbleRight{LaTeX}{-2}|
|\StrGobbleRight{a bc def }{4}|
```

```
xstri
||
L
|LaTeX|
|a bc |
```

2.2.9 \StrRight

\StrRight{⟨*chaine*⟩}{⟨*nombre*⟩}[⟨*nom*⟩]

Agit comme \StrLeft, mais renvoie les unités syntaxiques à la droite de ⟨*chaine*⟩.

```
\StrRight{xstring}{2}
\StrRight{xstring}{9}
\StrRight{LaTeX}{4}
|\StrRight{LaTeX}{-2}|
\StrRight{a bc def }{5}
```

```
ng
xstring
aTeX
||
def
```

2.2.10 \StrChar

\StrChar{⟨*chaine*⟩}{⟨*nombre*⟩}[⟨*nom*⟩]

Renvoie l'unité syntaxique à la position ⟨*nombre*⟩ dans la chaîne ⟨*chaine*⟩.

- ▷ Si ⟨*chaine*⟩ est vide, aucune unité syntaxique n'est renvoyée ;
- ▷ Si ⟨*nombre*⟩ $\leqslant 0$ ou si ⟨*nombre*⟩ $>$ ⟨*longueurChaine*⟩, aucune unité syntaxique n'est renvoyée.

```
\StrChar{xstring}{4}
|\StrChar{xstring}{9}|
|\StrChar{xstring}{-5}|
\StrChar{a bc def }{6}
```

```
r
||
||
d
```

2.2.11 \StrMid

\StrMid{⟨*chaine*⟩}{⟨*nombre1*⟩}{⟨*nombre2*⟩}[⟨*nom*⟩]

Dans ⟨*chaine*⟩, renvoie la sous chaîne se trouvant entre[9] les positions ⟨*nombre1*⟩ et ⟨*nombre2*⟩.

- ▷ Si ⟨*chaine*⟩ est vide, une chaîne vide est renvoyée ;
- ▷ Si ⟨*nombre1*⟩ $>$ ⟨*nombre2*⟩, alors rien n'est renvoyé ;
- ▷ Si ⟨*nombre1*⟩ < 1 et ⟨*nombre2*⟩ < 1 alors rien n'est renvoyé ;
- ▷ Si ⟨*nombre1*⟩ $>$ ⟨*longueurChaine*⟩ et ⟨*nombre2*⟩ $>$ ⟨*longueurChaine*⟩, alors rien n'est renvoyé ;
- ▷ Si ⟨*nombre1*⟩ < 1, alors la macro se comporte comme si ⟨*nombre1*⟩ $= 1$;
- ▷ Si ⟨*nombre2*⟩ $>$ ⟨*longueurChaine*⟩, alors la macro se comporte comme si ⟨*nombre2*⟩ $=$ ⟨*longueurChaine*⟩.

```
\StrMid{xstring}{2}{5}
\StrMid{xstring}{-4}{2}
|\StrMid{xstring}{5}{1}|
\StrMid{xstring}{6}{15}
\StrMid{xstring}{3}{3}
|\StrMid{a bc def }{2}{7}|
```

```
stri
xs
||
ng
t
| bc de|
```

9. Au sens large, c'est-à-dire que les chaînes « frontière » sont renvoyés.

2.3 Les macros renvoyant des nombres

2.3.1 \StrLen

\StrLen{⟨*chaine*⟩}[⟨*nom*⟩]

Renvoie la longueur de ⟨*chaine*⟩.

```
\StrLen{xstring}
\StrLen{A}
\StrLen{a bc def }
```

7
1
9

2.3.2 \StrCount

\StrCount{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]

Compte combien de fois ⟨*chaineA*⟩ est contenue dans ⟨*chaine*⟩.

▷ Si l'un au moins des arguments ⟨*chaine*⟩ ou ⟨*chaineA*⟩ est vide, la macro renvoie 0.

```
\StrCount{abracadabra}{a}
\StrCount{abracadabra}{bra}
\StrCount{abracadabra}{tic}
\StrCount{aaaaaa}{aa}
```

5
2
0
3

2.3.3 \StrPosition

\StrPosition[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]

L'argument optionnel ⟨*nombre*⟩ vaut 1 par défaut.

Dans ⟨*chaine*⟩, renvoie la position de l'occurrence n° ⟨*nombre*⟩ de ⟨*chaineA*⟩.

▷ Si ⟨*nombre*⟩ est supérieur au nombre d'occurrences de ⟨*chaineA*⟩, alors la macro renvoie 0.
▷ Si ⟨*chaine*⟩ ne contient pas ⟨*chaineA*⟩, alors la macro renvoie 0.

```
\StrPosition{xstring}{ring}
\StrPosition[4]{abracadabra}{a}
\StrPosition[2]{abracadabra}{bra}
\StrPosition[9]{abracadabra}{a}
\StrPosition{abracadabra}{z}
\StrPosition{a bc def }{d}
\StrPosition[3]{aaaaaa}{aa}
```

4
8
9
0
0
6
5

2.3.4 \StrCompare

\StrCompare⟨[*]⟩{⟨*chaineA*⟩}{⟨*chaineB*⟩}[⟨*nom*⟩]

Cette macro peut fonctionner avec 2 tolérances : la tolérance « normale » sélectionnée par défaut, et la tolérance « stricte ».

– La tolérance normale, activée par la commande \comparenormal.
  La macro compare successivement les unité syntaxique de gauche à droite des chaînes ⟨*chaineA*⟩ et ⟨*chaineB*⟩ jusqu'à ce qu'une différence apparaisse ou que la fin de la plus courte chaîne soit atteinte. Si aucune différence n'est trouvée, la macro renvoie 0. Sinon, la position de la 1re différence est renvoyée.
– La tolérance stricte, activée par la commande \comparestrict.
  La macro compare les 2 chaînes. Si elles sont égales, elle renvoie 0 sinon la position de la 1re différence est renvoyée.

L'ordre des 2 chaînes n'a aucune influence sur le comportement de la macro.

On peut également mémoriser le mode de comparaison en cours avec \savecomparemode, le modifier par la suite et revenir à la situation lors de la sauvegarde avec \restorecomparemode.

Exemples en tolérance normale :

```
\StrCompare{abcd}{abcd}
\StrCompare{abcd}{abc}
\StrCompare{abc}{abcd}
\StrCompare{a b c}{abc}
\StrCompare{aaa}{baaa}
\StrCompare{abc}{xyz}
\StrCompare{123456}{123457}
\StrCompare{abc}{}
```

0
0
0
2
1
1
6
0

Exemples en tolérance stricte :

```
\StrCompare{abcd}{abcd}
\StrCompare{abcd}{abc}
\StrCompare{abc}{abcd}
\StrCompare{a b c}{abc}
\StrCompare{aaa}{baaa}
\StrCompare{abc}{xyz}
\StrCompare{123456}{123457}
\StrCompare{abc}{}
```

0
4
4
2
1
1
6
1

3 Modes de fonctionnement

3.1 Développement des arguments

3.1.1 Les macros \fullexpandarg, \expandarg et \noexpandarg

La macro \fullexpandarg est appelée par défaut, ce qui fait que certains arguments (en violet dans la liste ci dessous) transmis aux macros sont développés le plus possible (pour cela, un \edef est utilisé). Ce mode de développement maximal permet dans la plupart des cas d'éviter d'utiliser des chaînes d'\expandafter. Le code en est souvent allégé.
On peut interdire le développement de ces arguments (et ainsi revenir au comportement normal de TEX) en invoquant \noexpandarg ou \normalexpandarg qui sont synonymes.

Il existe enfin un autre mode de développement de ces arguments que l'on appelle avec \expandarg. Dans ce cas, le **premier token** de ces arguments est développé *une fois* avant que la macro ne soit appelée. Si l'argument contient plus d'un token, les tokens qui suivent le premier ne sont pas développés (on peut contourner cette volontaire limitation et utiliser la macro \StrExpand, voir page 17).

Les commandes \fullexpandarg, \noexpandarg, \normalexpandarg et \expandarg peuvent être appelées à tout moment dans le code et fonctionnent comme des bascules. On peut rendre leur portée locale dans un groupe.

On peut également mémoriser le mode de développement en cours avec \saveexpandmode, le modifier par la suite et revenir à la situation lors de la sauvegarde avec \restoreexpandmode.

Dans la liste ci-dessous, on peut voir en violet quels arguments sont soumis à l'éventuel développement pour chaque macro vue dans le chapitre précedent :

- \IfSubStr⟨[*]⟩[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfSubStrBefore⟨[*]⟩[⟨*nombre1*⟩,⟨*nombre2*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfSubStrBehind⟨[*]⟩[⟨*nombre1*⟩,⟨*nombre2*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfBeginWith⟨[*]⟩{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfEndWith⟨[*]⟩{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfInteger{⟨*nombre*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfDecimal{⟨*nombre*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfStrEq⟨[*]⟩{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfEq⟨[*]⟩{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfStrEqCase⟨[*]⟩{⟨*chaine*⟩}{{⟨*chaine1*⟩}{⟨*code1*⟩}
  {⟨*chaine2*⟩}{⟨*code2*⟩}
  ...
  {⟨*chaine n*⟩}{⟨*code n*⟩}}[⟨*code alternatif*⟩]
- \IfEqCase⟨[*]⟩{⟨*chaine*⟩}{{⟨*chaine1*⟩}{⟨*code1*⟩}
  {⟨*chaine2*⟩}{⟨*code2*⟩}
  ...
  {⟨*chaine n*⟩}{⟨*code n*⟩}}[⟨*code alternatif*⟩]

- \StrBefore⟨[*]⟩[⟨nombre⟩]{⟨chaine⟩}{⟨chaineA⟩}[⟨nom⟩]
- \StrBehind⟨[*]⟩[⟨nombre⟩]{⟨chaine⟩}{⟨chaineA⟩}[⟨nom⟩]
- \StrBetween⟨[*]⟩[⟨nombre1⟩,⟨nombre2⟩]{⟨chaine⟩}{⟨chaineA⟩}{⟨chaineB⟩}[⟨nom⟩]
- \StrSubstitute[⟨nombre⟩]{⟨chaine⟩}{⟨chaineA⟩}{⟨chaineB⟩}[⟨nom⟩]
- \StrDel[⟨nombre⟩]{⟨chaine⟩}{⟨chaineA⟩}[⟨nom⟩]
- \StrSplit{⟨chaine⟩}{⟨nombre⟩}{⟨chaineA⟩}{⟨chaineB⟩} (voir page 16 pour la macro StrSplit)
- \StrGobbleLeft{⟨chaine⟩}{⟨nombre⟩}[⟨nom⟩]
- \StrLeft{⟨chaine⟩}{⟨nombre⟩}[⟨nom⟩]
- \StrGobbleRight{⟨chaine⟩}{⟨nombre⟩}[⟨nom⟩]
- \StrRight{⟨chaine⟩}{⟨nombre⟩}[⟨nom⟩]
- \StrChar{⟨chaine⟩}{⟨nombre⟩}[⟨nom⟩]
- \StrMid{⟨chaine⟩}{⟨nombre1⟩}{⟨nombre2⟩}[⟨nom⟩]
- \StrLen{⟨chaine⟩}[⟨nom⟩]
- \StrCount{⟨chaine⟩}{⟨chaineA⟩}[⟨nom⟩]
- \StrPosition[⟨nombre⟩]{⟨chaine⟩}{⟨chaineA⟩}[⟨nom⟩]
- \StrCompare{⟨chaineA⟩}{⟨chaineB⟩}[⟨nom⟩]

3.1.2 Caractères et lexèmes autorisés dans les arguments

Tout d'abord, quelque soit le mode de développement choisi, **les tokens de catcode 6 et 14 (habituellement # et %) sont interdits dans tous les arguments** [10].

Lorsque le mode \fullexpandarg est activé, les arguments sont évalués à l'aide de la primitive \edef avant d'être transmis aux macros. Par conséquent, sont autorisés dans les arguments :

- les lettres, majuscules, minuscules, accentuées [11] ou non, les chiffres, les espaces [12] ainsi que tout autre token de catcode 10, 11 ou 12 (signes de ponctuation, signes opératoires mathématiques, parenthèses, crochets, etc) ;
- les tokens de catcode 1 à 4, qui sont habituellement : « { » « } » [13] « $ » « & »
- les tokens de catcode 7 et 8, qui sont habituellement : « ^ » « _ »
- toute séquence de contrôle si elle est purement développable [14] et dont le développement maximal donne des caractères autorisés ;
- un token de catcode 13 (caractère actif) s'il est purement développable.

Lorsque les arguments ne sont plus développés (utilisation de \noexpandarg), on peut aussi inclure dans les arguments n'importe quel token, quelque soit le code qui en résulte comme par exemple toute séquence de contrôle, même non définie ou tout token de catcode 13. On peut également inclure dans les arguments des tokens de test comme \if ou \ifx ou tout autre token de test, même sans leur \fi correspondant ; ou bien un \csname sans le \endcsname correspondant.
Dans l'exemple suivant, l'argument contient un \ifx *sans* le \fi, et l'on isole ce qui est entre le \ifx et le \else :

```
\noexpandarg
\StrBetween{\ifx ab faux \else vrai}{\ifx}{\else}
```

ab faux

Lorsqu'on utilise \expandarg, la précaution concerne le premier token qui est développé une fois et doit donc être défini. Les autres tokens sont laissés tel quels comme avec \noexpandarg.

3.2 Développement des macros, argument optionnel

Les macros de ce package ne sont pas purement développables et ne peuvent donc pas être mises dans l'argument d'un \edef. L'imbrication des macros de ce package n'est pas permise non plus.

C'est pour cela que les macros renvoyant un résultat, c'est-à-dire toutes sauf les tests, sont dotées d'un argument optionnel venant en dernière position. Cet argument prend la forme de [⟨nom⟩], où ⟨nom⟩ est une séquence de contrôle qui recevra (l'assignation se fait avec un \edef) le résultat de la macro, ce qui fait que ⟨nom⟩ est purement développable et peut donc se trouver dans l'argument d'un \edef. Dans le cas de la présence d'un argument optionnel en dernière position, aucun affichage n'aura lieu. Cela permet donc contourner les limitations évoquées dans les exemples ci dessus.

10. Le token # sera peut-être autorisé dans une future version !
11. Pour pouvoir utiliser des lettres accentuées de façon fiable, il est nécessaire de charger le packages \fontenc avec l'option [T1], ainsi que \inputenc avec l'option correspondant au codage du fichier tex.
12. Selon la règle TEXiènne, plusieurs espaces consécutifs n'en font qu'un.
13. Attention : les accolades **doivent** être équilibrées dans les arguments !
14. C'est-à-dire qu'elle peut être mise à l'intérieur d'un \edef.

Ainsi cette construction non permise censée assigner à `\Resultat` les 4 caractères de gauche de `xstring` :

```
\edef\Resultat{\StrLeft{xstring}{4}}
```

est équivalente à :

```
\StrLeft{xstring}{4}[\Resultat]
```

Et cette imbrication non permise censée enlever le premier et le dernier caractère de `xstring` :

```
\StrGobbleLeft{\StrGobbleRight{xstring}{1}}{1}
```

se programme ainsi :

```
\StrGobbleRight{xstring}{1}[\machaine]
\StrGobbleLeft{\machaine}{1}
```

3.3 Traitement des arguments

3.3.1 Traitement à l'unité syntaxique prés

Les macros de xstring traitent les arguments unité syntaxique par unité syntaxique. Dans le code TEX, une unité syntaxique[15] est soit :

- une séquence de contrôle ;
- un groupe, c'est à dire une suite de tokens située entre deux accolades équilibrées ;
- un caractère ne faisant pas partie des 2 espèces ci dessus.

Voyons ce qu'est la notion d'unité syntaxique sur un exemple. Prenons cet argument : « `ab\textbf{xyz}cd` »
Il contient 6 unités syntaxiques qui sont : « `a` », « `b` », « `\textbf` », « `{xyz}` », « `c` » et « `d` ».

Que va t-il arriver si l'on se place sous `\noexpandarg` et que l'on demande à xstring de trouver la longueur de cet argument et d'en trouver le 4e « caractère » ?

```
\noexpandarg
\StrLen{ab\textbf{xyz}cd}\par
\StrChar{ab\textbf{xyz}cd}{4}[\mychar]
\meaning\mychar
```

```
6
macro:->{xyz}
```

Il est nécessaire d'utiliser `\meaning` pour bien visualiser le véritable contenu de `\mychar` et non pas de simplement d'appeler cette séquence de contrôle, ce qui fait perdre des informations — les accolades ici. On voit qu'on n'obtient pas vraiment un « caractère », mais cela était prévisible : il s'agit d'une unité syntaxique.

3.3.2 Exploration des groupes

Par défaut, la commande `\noexploregroups` est appelée et donc dans l'argument à examiner qui contient la chaîne de tokens, xstring considère les groupes entre accolades comme unités syntaxiques fermées dans lesquelles xstring ne regarde pas.
Pour certains besoins spécifiques, il peut être nécessaire de modifier le mode de lecture des arguments et d'explorer l'intérieur des groupes entre accolades. Pour cela on peut invoquer `\exploregroups`.

Que va donner ce nouveau mode d'exploration sur l'exemple précédent ? xstring ne va plus compter le groupe comme une seule unité syntaxique mais va compter les unités syntaxiques se trouvant à l'intérieur, et ainsi de suite s'il y avait plusieurs niveaux d'imbrication de groupes :

```
\noexpandarg
\exploregroups
\StrLen{ab\textbf{xyz}cd}\par
\StrChar{ab\textbf{xyz}cd}{4}[\mychar]
\meaning\mychar
```

```
8
macro:->x
```

L'exploration des groupes peut se réveler utile pour le comptage, le calcul de position ou les tests, mais comporte une limitation lorsque l'on appelle des macros renvoyant des chaînes : lorsqu'un argument est coupé à l'intérieur d'un groupe, alors **le résultat ne tient pas compte de ce qui se trouve à l'extérieur de ce groupe**. Il faut donc utiliser ce mode en connaissance de cause lorsque l'on utilise les macros renvoyant des chaînes.
Voyons ce que cela signifie sur un exemple : mettons que l'on veuille renvoyer ce qui se trouve à droite de la 2e occurrence de `\a` dans l'argument `\a1{\b1\a2}\a3`. Comme l'on explore les groupes, cette occurrence se trouve à l'intérieur du groupe `{\b1\a2}`. Le résultat renvoyé sera donc : `\b1`. Vérifions-le :

15. Pour les utilisateurs familiers avec la programmation LATEX, une unité syntaxique est ce qui est supprimé par la macro `\@gobble` dont le code, d'une grande simplicité, est : `\def\@gobble#1{}`

L'exploration des groupes [16] peut ainsi changer le comportement de la plupart des macros de xstring, à l'exception de \IfInteger, \IfDecimal, \IfStrEq, \IfEq et \StrCompare qui sont insensibles au mode d'exploration en cours.
De plus, pour des raison d'équilibrage d'accolades, 2 macros n'opèrent qu'en mode \noexploregroups, quelque soit le mode d'exploration en cours : \StrBetween et \StrMid.

On peut mémoriser le mode de d'exploration en cours avec \saveexploremode, le modifier par la suite et revenir à la situation lors de la sauvegarde avec \restoreexploremode.

3.4 Catcodes et macros étoilées

Les macros de xstring tiennent compte des catcodes des tokens constituant les arguments. Il faut donc garder à l'esprit, particulièrement lors des tests, que les tokens *et leurs catcodes* sont examinés.

Par exemple, ces 2 arguments :

{\string a\string b} et {ab}

ne se développent pas en 2 arguments égaux aux yeux de xstring. Dans le premier cas, à cause de l'emploi de la primitive \string, les caractères « ab » ont un catcode de 12 alors que dans l'autre cas, ils ont leurs catcodes naturels de 11. Il convient donc d'être conscient de ces subtilités lorsque l'on emploie des primitives dont les résultats sont des chaînes de caractères ayant des catcodes de 12 et 10. Ces primitives sont par exemple : \string, \detokenize, \meaning, \jobname, \fontname, \romannumeral, etc.

Pour demander aux macros de ne pas tenir compte des catcodes, on peut utiliser les macros étoilées. Après l'éventuel développement des arguments en accord avec le mode de développement, celles-ci convertissent (à l'aide d'un \detokenize) leur arguments en chaînes de caractères dont les catcodes sont 12 et 10 pour l'espace, avant que la macro non étoilée travaille sur ces arguments ainsi modifiés. Il faut noter que les arguments optionnels ne sont pas concernés par ces modifications et gardent leur catcode.

Voici un exemple :

```
1 \IfStrEq{\string a\string b}{ab}{vrai}{faux}\par
2 \IfStrEq*{\string a\string b}{ab}{vrai}{faux}
```

faux
vrai

Les chaînes n'étant pas égales à cause des catcodes, le test est bien *négatif* dans la version non étoilée.

Attention : utiliser une macro étoilée a des conséquenses ! Les arguments sont « détokénisés », il n'y a donc plus de séquence de contrôle, plus de goupes, ni aucun caractère de catcode spécial puisque tout est converti en caractères « inoffensifs » ayant le même catcode.

Ainsi, pour les macros renvoyant une chaîne, si on emploie les versions étoilées, le résultat sera une chaîne de caractères dont les catcodes sont 12, et 10 pour l'espace.
Et donc, après un « \StrBefore*{a \b c d}{c}[\montexte] », la séquence de contrôle \montexte se développera en « $a_{12}{}_{\sqcup 10}\backslash_{12}b_{12}{}_{\sqcup 10}$ ».

Les macros détokenisant leur arguments par l'utilisation de l'étoile sont présentées dans la liste ci-dessous. Pour chacune d'entre elles, on peut voir en violet quels arguments seront détokenisé lorsque l'étoile sera employée :

- \IfSubStr⟨[*]⟩[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfSubStrBefore⟨[*]⟩[⟨*nombre1*⟩,⟨*nombre2*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfSubStrBehind⟨[*]⟩[⟨*nombre1*⟩,⟨*nombre2*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfBeginWith⟨[*]⟩{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfEndWith⟨[*]⟩{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfStrEq⟨[*]⟩{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfEq⟨[*]⟩{⟨*chaineA*⟩}{⟨*chaineB*⟩}{⟨*vrai*⟩}{⟨*faux*⟩}
- \IfStrEqCase⟨[*]⟩{⟨*chaine*⟩}{{⟨*chaine1*⟩}{⟨*code1*⟩}
  {⟨*chaine2*⟩}{⟨*code2*⟩}
  ...
  {⟨*chaine n*⟩}{⟨*code n*⟩}}[⟨*code alternatif*⟩]
- \IfEqCase⟨[*]⟩{⟨*chaine*⟩}{{⟨*chaine1*⟩}{⟨*code1*⟩}
  {⟨*chaine2*⟩}{⟨*code2*⟩}
  ...
  {⟨*chaine n*⟩}{⟨*code n*⟩}}[⟨*code alternatif*⟩]
- \StrBefore⟨[*]⟩[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]
- \StrBehind⟨[*]⟩[⟨*nombre*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}[⟨*nom*⟩]

16. On peut consulter le fichier de test de xstring qui comporte de nombreux exemples et met en évidence les différences selon le mode d'exploration des groupes.

– \StrBetween⟨[*]⟩[⟨*nombre1*⟩,⟨*nombre2*⟩]{⟨*chaine*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}[⟨*nom*⟩]
– \StrCompare⟨[*]⟩{⟨*chaineA*⟩}{⟨*chaineB*⟩}[⟨*nom*⟩]

4 Macros avancées pour la programmation

Bien que xstring ait la possibilité de lire et traiter des arguments contenant du code TEX ou LATEX ce qui devrait couvrir la plupart des besoins en programmation, il peut arriver pour des besoins très spécifiques que les macros décritesz précédemment ne suffisent pas. Ce chapitre présente d'autres macros qui permettent d'aller plus loin ou de contourner certaines limitations.

4.1 Recherche d'un groupe, les macros \StrFindGroup et \groupID

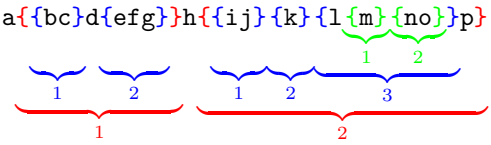
Lorsque le mode \exploregroups est actif, la macro \StrFindGroup permet de trouver un groupe entre accolades explicites en spécifiant son identifiant :

\StrFindGroup{⟨*argument*⟩}{⟨*identifiant*⟩}[⟨*nom*⟩]

Lorsque le groupe caractérisé par l'identifiant n'existe pas, une chaîne vide sera assignée à la séquence de contrôle ⟨*nom*⟩. Si le groupe existe, ce groupe *avec ses accolades* sera assigné à ⟨*nom*⟩.

Cet identifiant est une suite d'entiers séparés par des virgules caractérisant le groupe cherché dans l'argument. Le premier entier est le n^e groupe (d'imbrication 1) dans lequel est le groupe cherché. Puis, en se plaçant dans ce groupe, le 2^e entier est le n^e groupe dans lequel est le groupe cherché. Et ainsi de suite jusqu'à ce que l'imbrication du groupe soit atteinte.

Prenons par exemple l'argument suivant où l'on a 3 niveaux d'imbrication de groupes. Pour plus de clarté, les accolades délimitant les groupes sont colorées en rouge pour l'imbrication de niveau 1, en bleu pour le niveau 2 et en vert pour le niveau 3. Les groupes dans chaque imbrication sont ensuite numérotés selon la règle décrite ci-dessus :



Dans cet exemple :

– le groupe {{bc}d{efg}} a donc pour identifiant 1 ;
– le groupe {ij} a pour identifiant 2,1 ;
– le groupe {no} a pour identifiant 2,3,2 ;
– l'argument dans sa totalité « a{{bc}d{efg}}h{{ij}{k}{l{m}{no}}p} » a pour identifiant 0, seul cas où l'entier 0 est contenu dans l'identifiant d'un groupe.

Voici l'exemple complet :

```
\exploregroups
\expandarg
\def\chaine{a{{bc}d{efg}}h{{ij}{k}{l{m}{no}}p}}
\StrFindGroup{\chaine}{1}[\mongroupe]
\meaning\mongroupe\par
\StrFindGroup{\chaine}{2,1}[\mongroupe]
\meaning\mongroupe\par
\StrFindGroup{\chaine}{2,3,2}[\mongroupe]
\meaning\mongroupe\par
\StrFindGroup{\chaine}{2,5}[\mongroupe]
\meaning\mongroupe\par
```

```
macro:->{{bc}d{efg}}
macro:->{ij}
macro:->{no}
macro:->
```

Le processus inverse existe, et plusieurs macros de xstring donnent aussi comme information l'identifiant du groupe dans lequel elles ont fait une coupure ou trouvé une recherche. Ces macros sont : \IfSubStr, \StrBefore, \StrBehind, \StrSplit, \StrLeft, \StrGobbleLeft, \StrRight, \StrGobbleRight, \StrChar, \StrPosition.

Après l'appel à ces macros, la commande \groupID se développe en l'identifiant du groupe dans lequel la coupure s'est faite ou la recherche d'un argument a abouti. Lorsque la coupure ne peut avoir lieu ou que la recherche n'a pas abouti, \groupID est vide. Évidemment, l'utilisation de \groupID n'a de sens que lorsque le mode \exploregroups est actif, et quand les macros ne sont pas étoilées.

Voici quelques exemples avec la macro \StrChar :

```
\exploregroups
char 1 = \StrChar{a{b{cd}{e{f}g}h}ijkl}{1}\qquad
\string\groupID = \groupID\par
char 4 = \StrChar{a{b{cd}{e{f}g}h}ijkl}{4}\qquad
\string\groupID = \groupID\par
char 6 = \StrChar{a{b{cd}{e{f}g}h}ijkl}{6}\qquad
\string\groupID = \groupID\par
char 20 = \StrChar{a{b{cd}{e{f}g}h}ijkl}{20}\qquad
\string\groupID = \groupID
```

```
char 1 = a      \groupID= 0
char 4 = d      \groupID= 1,1
char 6 = f      \groupID= 1,2,1
char 20 =       \groupID=
```

4.2 Coupure d'une chaîne, la macro \StrSplit

Voici la syntaxe de cette macro :

\StrSplit{⟨*chaine*⟩}{⟨*nombre*⟩}{⟨*chaineA*⟩}{⟨*chaineB*⟩}

La ⟨*chaine*⟩, est coupée en deux chaînes juste après l'unité syntaxique se situant à la position ⟨*nombre*⟩. La partie gauche est assigné à la séquence de contrôle ⟨*chaineA*⟩ et la partie droite à ⟨*chaineB*⟩.
Cette macro renvoie *deux chaînes* et donc **n'affiche rien**. Par conséquent, elle ne dispose pas de l'argument optionnel en dernière position.

- ▷ Si ⟨*nombre*⟩ ⩽ 0, ⟨*chaineA*⟩ sera vide et ⟨*chaineB*⟩ contiendra la totalité de ⟨*chaine*⟩ ;
- ▷ Si ⟨*nombre*⟩ ⩾⟨*longueurChaine*⟩, ⟨*chaineA*⟩ contiendra la totalité de ⟨*chaine*⟩ et ⟨*chaineB*⟩ sera vide ;
- ▷ Si ⟨*chaine*⟩ est vide ⟨*chaineA*⟩ et ⟨*chaineB*⟩ seront vides, quelque soit l'entier ⟨*nombre*⟩.

```
    \def\seprouge{{\color{red}|}}
    \StrSplit{abcdef}{4}{\csA}{\csB}|\csA\seprouge\csB|\par
    \StrSplit{a b c }{2}{\csA}{\csB}|\csA\seprouge\csB|\par
    \StrSplit{abcdef}{1}{\csA}{\csB}|\csA\seprouge\csB|\par
    \StrSplit{abcdef}{5}{\csA}{\csB}|\csA\seprouge\csB|\par
    \StrSplit{abcdef}{9}{\csA}{\csB}|\csA\seprouge\csB|\par
    \StrSplit{abcdef}{-3}{\csA}{\csB}|\csA\seprouge\csB|
```

```
|abcd|ef|
|a |b c |
|a|bcdef|
|abcde|f|
|abcdef||
||abcdef|
```

Lorsque l'exploration des groupes est activée, et que l'on demande une coupure en fin de groupe, alors une chaîne contiendra la totalité du groupe tansque l'autre sera vide comme on le voit sur cet exemple :

```
\exploregroups
\StrSplit{ab{cd{ef}gh}ij}{6}\strA\strB
\meaning\strA\par
\meaning\strB
```

```
macro:->ef
macro:->
```

Une version étoilée de cette macro existe : dans ce cas, la coupure se fait juste avant la prochaine unité syntaxique qui suit l'unité syntaxique désirée. La version étoilée ne donne des résultats différents de la version normale que lorsque la n^{e} unité syntaxique est à la fin d'un groupe auquel cas, la coupure intervient non pas après cette unité syntaxique mais *avant* la prochaine unité syntaxique, que \StrSplit atteint en fermant autant de groupes que nécessaire.

```
\exploregroups
Utilisation sans \'etoile :\par
\StrSplit{ab{cd{ef}gh}ij}{6}\strA\strB
\meaning\strA\par
\meaning\strB\par
\string\groupID\ = \groupID\par\medskip
Utilisation avec \'etoile :\par
\StrSplit*{ab{cd{ef}gh}ij}{6}\strA\strB
\meaning\strA\par
\meaning\strB\par
\string\groupID\ = \groupID
```

```
Utilisation sans étoile :
macro:->ef
macro:->
\groupID = 1,1
Utilisation avec étoile :
macro:->cd{ef}
macro:->gh
\groupID = 1
```

4.3 Assigner un contenu verb, la macro \verbtocs

La macro \verbtocs permet le lire le contenu d'un « verb » qui peut contenir tous les caractères spéciaux : &, ~, \, {, }, _, #, $, ^ et %. Les caractères « normaux » gardent leur catcodes naturels, sauf les caractères spéciaux qui prennent un catcode de 12. Ensuite, ces caractères sont assignés à une séquence de contrôle. La syntaxe complète est :

\verbtocs{⟨*nom*⟩}|⟨*caractères*⟩|

⟨*nom*⟩ est le nom d'une séquence de contrôle qui recevra à l'aide d'un `\edef` les ⟨*caractères*⟩. ⟨*nom*⟩ contiendra donc des caractères de catcodes 12 ou 10 pour l'espace.

Par défaut, le token délimitant le contenu verb est « | », étant entendu que ce token ne peut être à la fois le délimiteur et être contenu dans ce qu'il délimite. Au cas où on voudrait lire un contenu verb contenant « | », on peut changer à tout moment le token délimitant le contenu verb par la macro :

`\setverbdelim{`⟨*token*⟩`}`

Tout ⟨*token*⟩ de catcode 12 peut être utilisé [17].

Concernant ces arguments verb, il faut tenir compte des deux points suivants :
– tous les caractères se trouvant avant |⟨*caractères*⟩| seront ignorés ;
– à l'intérieur des délimiteurs, tous les espaces sont comptabilisés même s'ils sont consécutifs.

Exemple :

```
\verbtocs{\resultat}  |a   & b{ c% d$ e \f|
J'affiche le r\'esultat :\par\resultat
```

J'affiche le résultat :
a & b{ c% d$ e \f

4.4 Tokenisation d'un texte vers une séquence de contrôle, la macro `\tokenize`

Le processus inverse de ce qui a été vu au dessus consiste à interpréter une suite de caractères en tokens. Pour cela, on dispose de la macro :

`\tokenize{`⟨*nom*⟩`}{`⟨*texte*⟩`}`

⟨*texte*⟩ est développé le plus possible si l'on a invoqué `\fullexpandarg` ; il n'est pas développé si l'on a invoqué `\noexpandarg` ou `\expandarg`. Après développement éventuel, le ⟨*texte*⟩ est transformé en tokens puis assigné à l'aide d'un `\def` à la séquence de contrôle ⟨*nom*⟩.

Voici un exemple où l'on détokenise un argument, on affiche le texte obtenu, puis on transforme ce texte en ce que l'argument était au début ; enfin, on affiche le résultat de la tokenisation :

```
\verbtocs{\text}|\textbf{a} $\frac{1}{2}$|
texte : \text
\tokenize{\resultat}{\text}\par
r\'esultat : \resultat
```

texte : \textbf{a} $\frac{1}{2}$
résultat : **a** $\frac{1}{2}$

Il est bien évident à la dernière ligne, que l'appel à la séquence de contrôle `\resultat` est ici possible puisque les séquences de contrôle qu'elle contient sont définies.

4.5 Développement contrôlé, les macros `\StrExpand` et `\scancs`

La macro `\StrExpand` développe les tokens de la chaîne passée en argument. Voici sa syntaxe :

`\StrExpand[`⟨*nombre*⟩`]{`⟨*chaine*⟩`}{`⟨*nom*⟩`}`

Le ⟨*nombre*⟩ vaut 1 par défaut et représente le nombre de développement(s) que doit subir la ⟨*chaine*⟩ de tokens. Le ⟨*nom*⟩ est le nom d'une séquence de contrôle à laquelle est assigné le résultat, une fois que tous les tokens aient été développé le nombre de fois demandé.

La macro opère séquentiellement et par passe : chaque token est remplacé par son 1-dévelopement, et le token suivant subit le même traitement jusqu'à ce que la chaîne ait été parcourue. Ensuite, s'il reste des niveaux de développement à faire, une nouvelle passe est initiée, et ainsi de suite jusqu'à ce que le nombre de développements voulu aient été exécutés.

17. Plusieurs tokens peuvent être utilisés au risque d'alourdir la syntaxe de `\verbtocs` ! Pour cette raison, avertissement sera émis si l'argument de `\setverbdelim` contient plusieurs tokens.

Voici un exemple :

```
\def\csA{1 2}
\def\csB{a \csA}
\def\csC{\csB\space}
\def\csD{x{\csA y}\csB{\csC z}}
D\'eveloppement de \string\csD\ au\par
\StrExpand[0]{\csD}{\csE} niveau 0 :
\detokenize\expandafter{\csE}\par
\StrExpand[1]{\csD}{\csE} niveau 1 :
\detokenize\expandafter{\csE}\par
\StrExpand[2]{\csD}{\csE} niveau 2 :
\detokenize\expandafter{\csE}\par
\StrExpand[3]{\csD}{\csE} niveau 3 :
\detokenize\expandafter{\csE}\par
\StrExpand[4]{\csD}{\csE} niveau 4 :
\detokenize\expandafter{\csE}
```

Développement de \csD au
niveau 0 : \csD
niveau 1 : x{\csA y}\csB {\csC z}
niveau 2 : x{1 2y}a \csA {\csB \space z}
niveau 3 : x{1 2y}a 1 2{a \csA z}
niveau 4 : x{1 2y}a 1 2{a 1 2 z}

La macro opère séquentiellement et chaque token est développé isolément sans tenir compte de ce qui suit. On ne peut donc développer que des tokens se suffisant à eux même et n'en nécessitant aucun autre. Ainsi, « \iftrue A\else B\fi », bien que se développant en « A » ne peut être mis dans l'argument de \StrExpand, et l'instruction :

\StrExpand{\iftrue A\else B\fi}\resultat

fera échouer la compilation puisque le premier token « \iftrue » sera développé *seul*, c'est-à-dire sans son \fi correspondant, ce qui fâchera TEX !

Les règles habituelles de lecture des arguments sont en vigueur, à savoir qu'un espace suivant une séquence de contrôle est ignoré, et plusieurs espaces consécutifs n'en font qu'un. Ces règles ne s'appliquent pas pour des espaces explicitement demandés avec \space ou \␣ [18].

Le développement de ce qui se trouve à l'intérieur des groupes est *indépendant* du mode d'exploration des groupes : cette macro possède ses propres commandes pour développer ce qui est dans les groupes ou pas. Par défaut, les tokens se trouvant à l'intérieur des groupes sont développés, mais on peut demander à ce que ce développement ne se fasse pas en invoquant \noexpandingroups et revenir au comportement par defaut par \expandingroups.

On peut détokeniser le résultat obtenu par la macro \StrExpand avec la macro \scancs dont la syntaxe est :

\scancs[⟨*nombre*⟩]{⟨*nom*⟩}{⟨*chaine*⟩}

Le ⟨*nombre*⟩ vaut 1 par défaut et représente le nombre de développement(s) que doit subir chaque token constituant la ⟨*chaine*⟩. Le ⟨*nom*⟩ est le nom d'une séquence de contrôle à laquelle est assigné le résultat, une fois que tous les tokens aient été développés le nombre de fois demandé et ensuite détokénisés.

\scancs a été conservée pour garder une compatibilité avec des précédentes versions de xstring. Pour les mêmes raisons, sa syntaxe n'est pas cohérente avec la syntaxe des autres macros. Cette macro, devenue triviale, prend le résultat de \StrExpand et lui applique simplement un \detokenize.

4.6 À l'intérieur d'une définition de macro

Avec le verbatim, certaines difficultés surviennent lorsque l'on se trouve à l'intérieur de la définition d'une macro, c'est-à-dire entre les accolades suivant un \def\macro ou un \newcommand\macro.

Pour les mêmes raison qu'il est interdit d'employer la commande \verb à l'intérieur de la définition d'une macro, les arguments verb du type |⟨*caractères*⟩| sont également interdits, ce qui disqualifie la macro \verbtocs. Il faut donc observer la règle suivante :

Ne pas utiliser la macro \verbtocs à l'intérieur de la définition d'une macro.

Mais alors, comment faire pour manipuler des arguments textuels verb et « verbatimiser » dans les définitions de macro ?

Il y a la primitive \detokenize de ε-TEX, mais elle comporte des restrictions, entre autres :
– les accolades doivent être équilibrées ;
– les espaces consécutifs sont ignorés ;
– les signes % sont interdits ;
– une espace est ajoutée après chaque séquence de contrôle.

18. À ce propos, \space n'a pas la même signification que \ . La première séquence de contrôle se *développe* en un espace tandis que la deuxième est une primitive TEX qui *affiche* une espace. Comme toutes les primitives, cette dernière est son propre développement.

Il est préférable d'utiliser la macro \scancs, et définir avec \verbtocs à l'extérieur des définitions de macros, des séquences de contrôle contenant des caractères spéciaux détokénisés. On pourra aussi utiliser la macro \tokenize pour transformer le résultat final (qui est une chaîne de caractères) en une séquence de contrôle. On peut voir des exemples utilisant ces macros page 19, à la fin de ce manuel.

Dans l'exemple artificiel [19] qui suit, on écrit une macro qui met son argument entre accolades. Pour cela, on définit en dehors de la définition de la macro 2 séquences de contrôles \Ob et \Cb contenant une accolade ouvrante et une accolade fermante de catcodes 12. Ces séquences de contrôle sont ensuite développées et utilisées à l'intérieur de la macro pour obtenir le résultat voulu :

```
\verbtocs{\Ob}|{|
\verbtocs{\Cb}|}|
\newcommand\bracearg[1]{%
    \def\text{#1}%
    \scancs{\result}{\Ob\text\Cb}%
    \result}

\bracearg{xstring}\par
\bracearg{\a}
```

{xstring}
{\a }

4.7 La macro \StrRemoveBraces

Pour des utilisations spéciales, on peut désirer retirer les accolades délimitant les groupes dans un argument. On peut utiliser la macro \StrRemoveBraces dont voici la syntaxe :

\StrRemoveBraces{⟨*chaine*⟩}[⟨*nom*⟩]

Cette macro est sensible au mode d'exploration, et retirera *toutes* les accolades avec \exploregroups alors qu'elle ne retirera que les accolades des groupes de plus bas niveau avec \noexploregroups.

```
\noexploregroups
\StrRemoveBraces{a{b{c}d}e{f}g}[\mycs]
\meaning\mycs

\exploregroups
\StrRemoveBraces{a{b{c}d}e{f}g}[\mycs]
\meaning\mycs
```

macro:->ab{c}defg
macro:->abcdefg

4.8 Exemples d'utilisation en programmation

Voici quelques exemples très simples d'utilisation des macros comme on pourrait en rencontrer en programmation.

4.8.1 Exemple 1

On cherche à remplacer les deux premiers \textit par \textbf dans la séquence de contrôle \myCS qui contient :

\textit{A}\textit{B}\textit{C}

On cherche évidemment à obtenir \textbf{A}\textbf{B}\textit{C} qui affiche : **AB***C*

Pour cela, on va développer les arguments des macros une fois avant qu'elles les traitent, en invoquant la commande \expandarg.
Ensuite, on définit \pattern qui est le motif à remplacer, et \replace qui est le motif de substitution. On travaille token par token puisque \expandarg a été appelé, il suffit d'invoquer \StrSubstitute pour faire les 2 substitutions.

```
\expandarg
\def\myCS{\textit{A}\textit{B}\textit{C}}
\def\pattern{\textit}
\def\replace{\textbf}
\StrSubstitute[2]{\myCS}{\pattern}{\replace}
```

AB*C*

Pour éviter de définir les séquences de contrôle \pattern et \replace, on aurait pu utiliser un leurre comme par exemple une séquence de contrôle qui se développe en « rien » comme \empty, et coder de cette façon :

\StrSubstitute[2]{\myCS}{\empty\textit}{\empty\textbf}

Ainsi, \empty est développée en « rien » et il reste dans les 2 derniers arguments les séquences de contrôles significatives \textit et textbf.

19. On peut agir beaucoup plus simplement en utilisant la commande \detokenize. Il suffit de définir la macro ainsi : \newcommand\bracearg[1]{\detokenize{{#1}}}

La séquence de contrôle \empty est donc un « hack » pour \expandarg : elle permet de bloquer le développement du 1er token ! On aurait d'ailleurs pu utiliser \noexpand au lieu de \empty pour obtenir le même résultat.

4.8.2 Exemple 2

On cherche ici à écrire une commande qui efface n unités syntaxiques dans une chaîne à partir d'une position donnée, et affecte le résultat dans une séquence de contrôle dont on peut choisir le nom.
On va appeler cette macro StringDel et lui donner la syntaxe :

```
\StringDel{chaine}{position}{n}{\nom_resultat}
```

On peut procéder ainsi : prendre la chaîne se trouvant juste avant la position, la sauvegarder. Ensuite enlever n + position unités syntaxiques à la chaîne initiale, et concaténer ce résultat à ce qui a été sauvegardé auparavant. Cela donne le code suivant :

```
1  \newcommand\StringDel[4]{%
2      \begingroup
3      \expandarg% portee locale au groupe
4      \StrLeft{\empty#1}{\number\numexpr#2-1}[#4]%
5      \StrGobbleLeft{\empty#1}{\numexpr#2+#3-1}[\StrA]%
6      \expandafter\expandafter\expandafter\endgroup
7      \expandafter\expandafter\expandafter\def
8      \expandafter\expandafter\expandafter#4%
9      \expandafter\expandafter\expandafter
10     {\expandafter#4\StrA}%
11 }
12
13 \noexploregroups
14 \StringDel{abcdefgh}{2}{3}{\cmd}
15 \meaning\cmd
16
17 \StringDel{a\textbf{1}b\textbf{2c}3d}{3}{4}{\cmd}
18 \meaning\cmd
```

```
macro:->aefgh
macro:->a\textbf 3d
```

Pour la concaténation, on aurait pu procéder différemment en utilisant la macro \g@addto@macro de LaTeX. Cela évite aussi ces laborieux « ponts » d'\expandafter. Il suffit alors de remplacer l'assignation et la sortie du groupe se trouvant entre les lignes 6 à 10 par [20] :

```
\expandafter\g@addto@macro\expandafter#4\expandafter{\StrA}\endgroup
```

4.8.3 Exemple 3

Cherchons à écrire une macro \tofrac qui transforme une écriture du type « a/b » par « $\frac{a}{b}$ ».

Tout d'abord, annulons le développement des arguments avec \noexpandarg : nous n'avons pas besoin de développement ici. Il suffit d'isoler ce qui se trouve avant et après la 1re occurrence de « / » (on suppose qu'il n'y a qu'une seule occurrence), le mettre dans les séquences de contrôle \num et \den et simplement appeler la macro TeX \frac :

```
1  \noexpandarg
2  \newcommand\tofrac[1]{%
3      \StrBefore{#1}{/}[\num]%
4      \StrBehind{#1}{/}[\den]%
5      $\frac{\num}{\den}$%
6  }
7  \tofrac{15/9}
8  \tofrac{u_{n+1}/u_n}
9  \tofrac{a^m/a^n}
10 \tofrac{x+\sqrt{x}/\sqrt{x^2+x+1}}
```

$\frac{15}{9}$ $\frac{u_{n+1}}{u_n}$ $\frac{a^m}{a^n}$ $\frac{x+\sqrt{x}}{\sqrt{x^2+x+1}}$

4.8.4 Exemple 4

Soit une phrase composée de texte. Dans cette phrase, essayons construire une macro qui mette en gras le 1er mot qui suit un mot donné. On entend par mot une suite de caractère ne commençant ni ne finissant par un espace. Si le mot n'existe pas dans la phrase, rien n'est fait.

On va écrire une macro \grasapres qui effectue ce travail. On appelle \StrBehind qui assigne à \mot ce qui se trouve après la 1re occurrence du mot (précédé et suivi de son espace). Ensuite, le mot à mettre en gras est

20. À condition d'avoir provisoirement changé le code de catégorie de « @ » en écrivant la macro entre les commandes \makeatletter et \makeatother

ce qui se trouve avant le 1er espace dans la séquence de contrôle \mot. Remarquons que ceci reste vrai même si le mot à mettre en gras est le dernier de l'argument car un espace a été rajouté à la fin de l'argument par {#1 } lors de l'appel à \StrBehind. Remarquons aussi que \expandarg a été appelé et donc, le premier token de l'argument \textbf{\mot} est développé 1 fois, *lui aussi* ! Cela est possible (heureusement sinon, il aurait fallu faire autrement et utiliser le hack de l'exemple précédent) puisque le 1-développement de cette macro de LaTeX est « \protect\textbf » [21].

```
\newcommand\grasapres[2]{%
    \noexpandarg
    \StrBehind[1]{#1 }{ #2 }[\mot]%
    \expandarg
    \StrBefore{\mot}{ }[\mot]%
    \StrSubstitute[1]{#1}{\mot}{\textbf{\mot}}%
}

\grasapres{Le package xstring est nouveau}{package}

\grasapres{Le package xstring est nouveau}{ckage}

\grasapres{Le package xstring est nouveau}{est}
```

Le package **xstring** est nouveau
Le package xstring est nouveau
Le package xstring est **nouveau**

4.8.5 Exemple 5

Soit un argument commençant par au moins 3 séquences de contrôles avec leurs éventuels arguments. Comment intervertir les 2 premières séquences de contrôle de telle sorte qu'elles gardent leurs arguments ? On va pour cela écrire une macro \swaptwofirst.
Cette fois ci, on ne peut pas chercher le seul caractère « \ » (de catcode 0) dans un argument. Nous serons obligé de détokeniser l'argument, c'est ce que fait \scancs[0]\chaine{#1} qui met le résultat dans \chaine. Ensuite, on cherchera dans cette séquence de contrôle les occurrences de \antislash qui contient le caractère « \ » de catcode 12, assigné avec un \verbtocs écrit *en dehors* [22] du corps de la macro. La macro se termine par une retokenisation, une fois que les chaînes \avant et \apres aient été échangées.

```
\verbtocs{\antislash}|\|
\newcommand\swaptwofirst[1]{%
    \begingroup
    \fullexpandarg
    \scancs[0]\chaine{#1}%
    \StrBefore[3]{\chaine}{\antislash}[\firsttwo]%
    \StrBehind{\chaine}{\firsttwo}[\others]
    \StrBefore[2]{\firsttwo}{\antislash}[\avant]
    \StrBehind{\firsttwo}{\avant}[\apres]%
    \tokenize\myCS{\apres\avant\others}%
    \myCS
    \endgroup
}

\swaptwofirst{\underline{A}\textbf{B}\textit{C}}

\swaptwofirst{\Large\underline{A}\textbf{B}123}
```

BAC
AB123

4.8.6 Exemple 6

Dans une chaîne, on cherche ici à isoler le ne mot se trouvant entre 2 délimiteurs précis. Pour cela on écrira une macro \findword admettant comme argument optionnel le délimiteur de mot (l'espace par défaut), 1 argument contenant la chaîne, 1 argument contenant le nombre correspondant au ne mot cheché.

La macro \findword utilise \StrBetween et \numexpr de façon astucieuse, et profite du fait que \StrBetween

21. En toute rigueur, il aurait fallu écrire :
    \StrSubstitute[1]{#1}{\mot}{\expandafter\textbf\expandafter{\mot}}
    De cette façon, dans {\expandafter\textbf\expandafter{\mot}}, la séquence de contrôle \mot est développée **avant** que l'appel à la macro ne se fasse. Cela est dû à l'\expandafter placé en début d'argument qui est développé à cause de \expandarg et grace à l'autre \expandafter, provoque le développement de \mot
22. En effet, la macro \verbtocs et son argument verb est interdite à l'intérieur de la définition d'une macro.

n'explore pas les groupes :

```
\newcommand\findword[3][ ]{%
    \StrBetween[#3,\numexpr#3+1]{#1#2#1}{#1}{#1}%
}
\noexpandarg
|\findword{a bc d\textbf{e f} gh}{3}|

|\findword[\nil]{1 \nil 2 3 \nil4\nil5}{2}|
```

```
|de f|
|2 3 |
```

On peut observer que le résultat de la ligne 7 qui est « 2␣3␣ » ne commence *pas* par une espace puisque dans le code, cet espace suit une séquence de contrôle — ici `\nil`.

⋆
⋆ ⋆

C'est tout, j'espère que ce package vous sera utile !
Merci de me signaler par email tout bug ou toute proposition d'amélioration...

Christian Tellechea