xstring

v1.4

Manuel de l'utilisateur

Christian Tellechea
unbonpetit@gmail.com

4 novembre 2008

Résumé

Cette extension, qui requiert Plain ε-TeX, regroupe un ensemble de macros manipulant des chaînes pouvant contenir des caractères, séquences de contrôle, groupes entre accolades, de façon à utiliser les macros aussi pour de la programmation TeX :

- ▷ 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 extractions de chaînes :
  - renvoi de ce qui se trouve avant (ou après) la n^{e} occurrence d'une sous-chaîne ;
  - renvoi de ce qui se trouve entre les occurrences de 2 sous-chaînes ;
  - sous-chaîne comprise entre 2 positions, etc.
- ▷ 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 du 1^{er} caractère différent entre 2 chaînes.

D'autres commandes permettent de gérer les caractères spéciaux normalement interdits dans les chaînes (`#` et `%`), ainsi que d'éventuelles différences entre catcodes de caractères, 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 caractères représentant du code TEX, un peu comme en disposent des langages dit « évolués ». On y trouve les opérations habituelles sur les chaînes de caractères, 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 ou de caractères, calculs de position d'une sous-chaîne, calculs du nombre d'occurrences, etc.

xstring lit les arguments qui lui sont transmis *unité syntaxique par unité syntaxique*[2], ce qui revient à les lire caractère par caractère lorsque ceux-ci contiennent des caractères « 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 caractères dont les catcodes sont moins inoffensifs. Voir le chapitre sur le mode de lecture et de développement des arguments (page 12), la commande `\verbtocs` (page 13), la commande `\scancs` (page 14).

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 pouvant éventuellement contenir des caractères, il peut arriver aux utilisateurs avancés de rencontrer des problèmes de « catcodes[3] » conduisant à des comportement inattendus. Ces effets indésirables peuvent être contrôlés. Consulter en particulier le chapitre sur les catcodes des arguments page 13.

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 de caractères. 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

2.1 Présentation des 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 caractères spéciaux et encore moins de séquence de contrôle dans les arguments !

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 ;
- 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

1. L'extension ne nécessite pas LATEXet 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 12.
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 13 pour les macros étoilées et page 11 pour l'argument optionnel en dernière position.
7. Pour une collection plus importante d'exemples, on peut aussi consulter le fichier de test.

quant à la présence d'espaces dans le résultat, celui-ci sera délimité par des « | », étant entendu qu'une chaîne vide est représentée par « || ».

2.2 Les tests

2.2.1 `\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*⟩.

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

2.2.2 `\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*⟩.

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

2.2.3 `\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*⟩.

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

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

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

2.2.6 \IfInteger

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

Teste si ⟨*nombre*⟩ est un nombre entier relatif, 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}  vrai
\IfInteger{-219}{vrai}{faux}  vrai
  \IfInteger{+9}{vrai}{faux}  vrai
\IfInteger{3.14}{vrai}{faux}  faux
   \IfInteger{0}{vrai}{faux}  vrai
 \IfInteger{49a}{vrai}{faux}  faux
   \IfInteger{+}{vrai}{faux}  faux
   \IfInteger{-}{vrai}{faux}  faux
\IfInteger{0000}{vrai}{faux}  vrai
```

2.2.7 \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 ce qui se trouve à droite du séparateur décimal s'il est présent est vide, le test est faux ;
▷ Si ce qui se trouve à gauche du séparateur décimal s'il est présent est vide, la partie entière est considérée comme étant 0.

```
\IfDecimal{3.14}{vrai}{faux}  vrai
\IfDecimal{3,14}{vrai}{faux}  vrai
\IfDecimal{-0.5}{vrai}{faux}  vrai
  \IfDecimal{.7}{vrai}{faux}  vrai
  \IfDecimal{,9}{vrai}{faux}  vrai
\IfDecimal{1..2}{vrai}{faux}  faux
  \IfDecimal{+6}{vrai}{faux}  vrai
 \IfDecimal{-15}{vrai}{faux}  vrai
  \IfDecimal{1.}{vrai}{faux}  faux
  \IfDecimal{2,}{vrai}{faux}  faux
```

```
\IfDecimal{.}{vrai}{faux}  faux
\IfDecimal{,}{vrai}{faux}  faux
\IfDecimal{+}{vrai}{faux}  faux
\IfDecimal{-}{vrai}{faux}  faux
```

2.2.8 \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 caractères dans le même ordre. Exécute ⟨*vrai*⟩ dans l'affirmative, et ⟨*faux*⟩ dans le cas contraire.

```
\IfStrEq{a1b2c3}{a1b2c3}{vrai}{faux}  vrai
\IfStrEq{abcdef}{abcd}{vrai}{faux}  faux
\IfStrEq{abc}{abcdef}{vrai}{faux}  faux
\IfStrEq{3,14}{3,14}{vrai}{faux}  vrai
\IfStrEq{12.34}{12.340}{vrai}{faux}  faux
\IfStrEq{abc}{}{vrai}{faux}  faux
\IfStrEq{}{abc}{vrai}{faux}  faux
\IfStrEq{}{}{vrai}{faux}  vrai
```

2.2.9 \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 ;
- ▷ Il est possible d'évaluer des expressions algébriques en utilisant la primitive `\numexpr` de ε-TEX, en gardant à l'esprit qu'elle ne donne que des résultats entiers et qu'elle arrondit les résultats non entiers à l'entier le plus proche.

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

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

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

2.3 Les macros renvoyant une chaîne

2.3.1 \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}	|xs|
\StrBefore{LaTeX}{e}	|LaT|
\StrBefore{LaTeX}{p}	||
\StrBefore{LaTeX}{L}	||
\StrBefore{a bc def }{def}	|a bc |
\StrBefore{a bc def }{cd}	||
\StrBefore[1]{1b2b3}{b}	|1|
\StrBefore[2]{1b2b3}{b}	|1b2|

2.3.2 \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}	|ng|
\StrBehind{LaTeX}{e}	|X|
\StrBehind{LaTeX}{p}	||
\StrBehind{LaTeX}{X}	||
\StrBehind{a bc def }{bc}	| def |
\StrBehind{a bc def }{cd}	||
\StrBehind[1]{1b2b3}{b}	|2b3|
\StrBehind[2]{1b2b3}{b}	|3|
\StrBehind[3]{1b2b3}{b}	||

2.3.3 \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}`	|tri|
`\StrBetween{xstring}{i}{n}`	||
`\StrBetween{xstring}{a}{tring}`	||
`\StrBetween{a bc def }{a}{d}`	| bc |
`\StrBetween{a bc def }{a }{f}`	|bc de|
`\StrBetween{a1b1a2b2a3b3}{a}{b}`	|1|
`\StrBetween[2,3]{a1b1a2b2a3b3}{a}{b}`	|2b2a3|
`\StrBetween[1,3]{a1b1a2b2a3b3}{a}{b}`	|1b1a2b2a3|
`\StrBetween[3,1]{a1b1a2b2a3b3}{a}{b}`	||
`\StrBetween[3,2]{abracadabra}{a}{bra}`	|da|

2.3.4 \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.

`\StrSubstitute{xstring}{i}{a}`	xstrang
`\StrSubstitute{abracadabra}{a}{o}`	obrocodobro
`\StrSubstitute{abracadabra}{br}{TeX}`	aTeXacadaTeXa
`\StrSubstitute{LaTeX}{m}{n}`	LaTeX
`\StrSubstitute{a bc def }{ }{M}`	aMbcMdefM
`\StrSubstitute{a bc def }{ab}{AB}`	a bc def
`\StrSubstitute[1]{a1a2a3}{a}{B}`	B1a2a3
`\StrSubstitute[2]{a1a2a3}{a}{B}`	B1B2a3
`\StrSubstitute[3]{a1a2a3}{a}{B}`	B1B2B3
`\StrSubstitute[4]{a1a2a3}{a}{B}`	B1B2B3

2.3.5 \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 ;

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

```
\StrDel{abracadabra}{a}  brcdbr
\StrDel[1]{abracadabra}{a}  bracadabra
\StrDel[4]{abracadabra}{a}  brcdbra
\StrDel[9]{abracadabra}{a}  brcdbr
\StrDel{a bc def }{ }  abcdef
\StrDel{a bc def }{def}  |a bc  |
```

2.3.6 \StrSplit

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

La ⟨*chaine*⟩, est coupée en deux chaînes après le caractère 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 [9] 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*⟩.

```
\StrSplit{abcdef}{4}{\csA}{\csB}résultats : |\csA| et |\csB|  résultats : |abcd| et |ef|
\StrSplit{a b c }{2}{\csA}{\csB}résultats : |\csA| et |\csB|  résultats : |a | et |b c |
\StrSplit{abcdef}{1}{\csA}{\csB}résultats : |\csA| et |\csB|  résultats : |a| et |bcdef|
\StrSplit{abcdef}{5}{\csA}{\csB}résultats : |\csA| et |\csB|  résultats : |abcde| et |f|
\StrSplit{abcdef}{9}{\csA}{\csB}résultats : |\csA| et |\csB|  résultats : |abcdef| et ||
\StrSplit{abcdef}{-3}{\csA}{\csB}résultats : |\csA| et |\csB|  résultats : |abcdef| et ||
```

2.3.7 \StrGobbleLeft

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

Dans ⟨*chaine*⟩, enlève les ⟨*nombre*⟩ premiers caractères de gauche.

- ▷ Si ⟨*chaine*⟩ est vide, renvoie une chaîne vide ;
- ▷ Si ⟨*nombre*⟩ ⩽ 0, aucun caractère n'est supprimé.
- ▷ Si ⟨*nombre*⟩ ⩾ ⟨*longueurChaine*⟩, tous les caractères sont supprimés.

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

2.3.8 \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, aucun caractère n'est retourné.
- ▷ Si ⟨*nombre*⟩ ⩾ ⟨*longueurChaine*⟩, tous les caractères sont retournés.

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

2.3.9 \StrGobbleRight

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

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

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

9. Voir les explications sur l'argument venant en dernière position page 11.

2.3.10 \StrRight

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

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

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

2.3.11 \StrChar

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

Renvoie le caractère à la position ⟨*nombre*⟩ dans la chaîne ⟨*chaine*⟩.

- ▷ Si ⟨*chaine*⟩ est vide, aucun caractère n'est renvoyé ;
- ▷ Si ⟨*nombre*⟩ ⩽ 0 ou si ⟨*nombre*⟩ > ⟨*longueurChaine*⟩, aucun caractère n'est renvoyé.

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

2.3.12 \StrMid

\StrMid⟨[*]⟩{⟨*chaine*⟩}{⟨*nombreA*⟩}{⟨*nombreB*⟩}[⟨*nom*⟩]

Dans ⟨*chaine*⟩, renvoie la sous chaîne se trouvant entre[10] les positions ⟨*nombreA*⟩ et ⟨*nombreB*⟩.

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

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

2.4 Les macros renvoyant des nombres

2.4.1 \StrLen

\StrLen⟨[*]⟩{⟨*chaine*⟩}[⟨*nom*⟩]

Renvoie la longueur de ⟨*chaine*⟩.

```
\StrLen{xstring}  7
\StrLen{A}        1
\StrLen{a bc def } 9
```

2.4.2 \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}   5
\StrCount{abracadabra}{bra} 2
\StrCount{abracadabra}{tic} 0
\StrCount{aaaaaa}{aa}       3
```

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

2.4.3 \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}  4
\StrPosition[4]{abracadabra}{a}  8
\StrPosition[2]{abracadabra}{bra}  9
\StrPosition[9]{abracadabra}{a}  0
\StrPosition{abracadabra}{z}  0
\StrPosition{a bc def }{d}  6
\StrPosition[3]{aaaaaa}{aa}  5
```

2.4.4 \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 caractères 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 `\restoreecomparemode`.

Exemples en tolérance normale :

```
\StrCompare{abcd}{abcd}  0
\StrCompare{abcd}{abc}  0
\StrCompare{abc}{abcd}  0
\StrCompare{éùçà}{éùçà}  0
\StrCompare{a b c}{abc}  2
\StrCompare{aaa}{baaa}  1
\StrCompare{abc}{xyz}  1
\StrCompare{123456}{123457}  6
\StrCompare{abc}{}  0
```

Exemples en tolérance stricte :

```
\StrCompare{abcd}{abcd}  0
\StrCompare{abcd}{abc}  4
\StrCompare{abc}{abcd}  4
\StrCompare{éùçà}{éùçà}  0
\StrCompare{a b c}{abc}  2
\StrCompare{aaa}{baaa}  1
\StrCompare{abc}{xyz}  1
\StrCompare{123456}{123457}  6
\StrCompare{abc}{}  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 tous les arguments 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 des arguments des macros (et ainsi revenir au comportement normal de TEX) en invoquant `\noexpandarg` ou `\normalexpandarg` qui sont synonymes.

Il existe enfin un autre mode de développement des arguments que l'on appelle avec `\expandarg`. Dans ce cas, le **premier lexème** de chaque argument est développé *une fois* avant que la macro ne soit appelée. Dans le cas où un argument contient plus d'un lexème, les lexèmes qui suivent le premier ne sont pas développés du tout (on peut contourner cette limitation et faire appel, au préalable, à la macro `\scancs*`, voir page 14).

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`.

3.1.2 Caractères autorisés dans les arguments

Tout d'abord, quelque soit le mode de développement choisi, **les lexèmes de catcode 6 et 14 (habituellement # et %) sont interdits dans tous les arguments** [11].

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 [12] ou non, les chiffres, les espaces [13] ainsi que tout autre lexème de catcode 10, 11 ou 12 (signes de ponctuation, signes opératoires mathématiques, parenthèses, crochets, etc) ;
- les lexèmes de catcode 1 à 4, qui sont habituellement : « `{` » « `}` » [14] « `$` » « `&` »
- les lexèmes de catcode 7 et 8, qui sont habituellement : « `^` » « `_` »
- toute séquence de contrôle si elle est purement développable [15] et dont le développement maximal donne des caractères autorisés ;
- un lexème de catcode 13 (caractère actif) s'il est purement développable.

En revanche, certains caractères [16] comme €, ¤, ¶, etc. généreront des erreurs.

Lorsque les arguments ne sont plus développés au maximum avec `\expandarg` ou `\noexpandarg`, les caractères autorisés sont ceux cités ci-dessus auxquels il faut rajouter toute séquence de contrôle, même non définie, ainsi que les caractères précédemment cités (€, ¤, ¶, etc.).

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, même en faisant appel à `\expandafter`.

C'est pour cela que les macros renvoyant un résultat, c'est-à-dire toutes sauf les tests et `\StrSplit`, 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.

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

11. Le lexème `#` sera peut-être autorisé dans une future version !
12. 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.
13. Selon la règle TEXiènne, plusieurs espaces consécutifs n'en font qu'un.
14. Attention : les accolades **doivent** être équilibrées dans les arguments !
15. C'est-à-dire qu'elle peut être mise à l'intérieur d'un `\edef`.
16. Ces caractères s'obtiennent avec la touche AltGr + lettre sous les distributions GNU/Linux.

```
\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 Lecture des arguments

3.3.1 Lecture à l'unité syntaxique prés

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

- une séquence de contrôle ;
- un groupe, c'est à dire une suite de caractères 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. 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 principal, 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 :

```
\noexpandarg
\exploregroups
\StrBefore[2]{\a1{\b1\a2}\a3}{\a}[\mycs]
\meaning\mycs
```

```
macro:->\b 1
```

L'exploration des groupes [17] peut ainsi changer le comportement de la plupart des macros de xstring, à l'exception de `\IfInteger`, `\IfDecimal`, `\IfStrEq`, `\StrEq` 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 des arguments, macros étoilées

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

Par exemple, ces 2 arguments :

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

ne se développent pas en 2 arguments égaux au 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 comme `\string`, `\detokenize`, `\meaning`, `\jobname`, `\fontname`, `\romannumeral`, etc.

Pour demander aux macros de ne pas tenir compte des catcodes, on peut utiliser les macros étoilées. Celles-ci convertissent tous leurs 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.

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.

Voici un exemple :

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

faux
vrai

Dans l'exemple ci-dessus, après développement (on suppose que `\fullexpandarg` est actif), le premier argument contient `{ab}` où, à cause de `\string`, les 2 caractères ont un catcode de 12. Le deuxième argument est `{ab}` où les caractères ont leurs catcodes naturels de 11. Les chaînes n'étant pas égales à cause des catcodes, le test est donc *négatif* dans la version non étoilée.

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. Ainsi après un « `\StrBefore*{a \b c d}{c}[\montexte]` », la séquence de contrôle `\montexte` se développera en « $\texttt{a}_{12\sqcup 10}\backslash_{12}\texttt{b}_{12\sqcup 10}$ ».

4 Autres macros pour une aide à 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 cela ne suffise pas. Ce chapitre présente d'autres macros qui permettent de contourner certaines limitations.

4.1 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 caractère délimitant le contenu verb est « | », étant entendu que ce caractère 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 caractère délimitant le contenu verb par la macro :

17. 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.

\setverbdelim{⟨*caractère*⟩}

Tout ⟨*caractère*⟩ ayant un catcode de 11 ou 12 peut être utilisé [18].

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 résultat :\par\resultat
```

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

4.2 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 lexèmes. Pour cela, on dispose de la macro :

\tokenize{⟨*nom*⟩}{⟨*séquences de contrôle*⟩}

⟨*séquences de contrôle*⟩ est développée le plus possible si l'on a invoqué \fullexpandarg ; elle n'est pas développée si l'on a invoqué \normalexpandarg ou \expandarg. Après développement éventuel, la suite de caractères est transformeé 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
résultat : \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.

4.3 Développement d'une séquence de contrôle avant une conversion en verb, la macro \scancs

On peut souhaiter développer une séquence de contrôle avant de convertir ce développement en texte. Pour cela existe la macro :

\scancs⟨[*]⟩[⟨*nombre*⟩]{⟨*nom*⟩}{⟨*séquence(s) de contrôle*⟩}

Le ⟨*nombre*⟩ vaut 1 par défaut et représente la profondeur à laquelle va être développée la ⟨*séquence de contrôle*⟩ avant d'être verbatimisée en caractères de catcode 12 ou 10 pour l'espace, puis assignée à la séquence de contrôle ⟨*nom*⟩.

Si cela s'avère nécessaire, on peut aussi contrôler la profondeur de développement avec l'argument optionnel. Si le n-développement donne une séquence de contrôle, alors cette séquence de contrôle est verbatimisée en caractères de catcode 12. L'exemple montre toutes les profondeurs de développement de la séquence de contrôle \c de la profondeur 0 jusqu'à la profondeur 3 :

```
\def\a{1 z 3}
\def\b{\a}
\def\c{\b}
\scancs[0]{\resultat}{\c}
développement 0 : \resultat\par
\scancs[1]{\resultat}{\c}
développement 1 : \resultat\par
\scancs[2]{\resultat}{\c}
développement 2 : \resultat\par
\scancs[3]{\resultat}{\c}
développement 3 : \resultat
```

développement 0 : \c
développement 1 : \b
développement 2 : \a
développement 3 : 1 z 3

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

Il est bien évident qu'il faut s'assurer que le développement à la profondeur souhaitée soit possible, sous peine d'erreur à la compilation.

En toute rigueur, le 3e argument ⟨*séquence de contrôle*⟩ (ou l'un de ses développements) doit contenir une seule et unique séquence de contrôle qui elle seule sera développée. Si ce 3e argument ou l'un ses développements contient plusieurs séquences de contrôle, la compilation s'arrête avec un message d'erreur invitant à utiliser la version étoilée. La macro étoilée, d'utilisation plus délicate et qu'il convient donc d'utiliser avec attention, permet en effet de développer *toutes* les séquences de contrôle contenues dans le 3e argument à la profondeur spécifiée dans le 1er argument. Il faut garder à l'esprit que si le n-développement contient un groupe entre accolades, ce groupe sera développé au développement $n+1$, et perdra ses accolades ! Il en est de même pour les espaces [19].
Voici un exemple :

```
\def\a{1 {2}}
\def\b{\a \a}
\scancs*[0]{\resultat}{{A}\b}
développement 0 : \resultat\par
\scancs*[1]{\resultat}{{A}\b}
développement 1 : \resultat\par
\scancs*[2]{\resultat}{{A}\b}
développement 2 : \resultat\par
\scancs*[3]{\resultat}{{A}\b}
développement 3 : \resultat
```

développement 0 : {A}\b
développement 1 : A\a \a
développement 2 : A1 {2}1 {2}
développement 3 : A1212

4.4 À l'intérieur d'une définition de macro

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 textuels 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 ;
– les caractères `#` sont doublés.

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 16, à la fin de ce manuel.

Dans l'exemple artificiel [20] 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 }

19. Tout appel à la macro `\scancs` est accompagné d'un message d'attention mettant en grade sur ces inconvénients.
20. On peut agir beaucoup plus simplement en utilisant la commande `\detokenize`. Il suffit de définir la macro ainsi : `\newcommand\bracearg[1]{\detokenize{{#1}}}`

4.5 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 haut niveau avec `\noexploregroups`.

```
                                \noexploregroups
\StrRemoveBraces{a{b{c}d}e{f}g}[\mycs]
                                \meaning\mycs  macro:->ab{c}defg
                               \exploregroups
\StrRemoveBraces{a{b{c}d}e{f}g}[\mycs]
                                \meaning\mycs  macro:->abcdefg
```

4.6 Exemples d'utilisation en programmation

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

4.6.1 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 donne : **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 « vide » 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`.

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

4.6.2 Exemple 2

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 `\avant` et `\apres` et simplement appeler la macro `\frac` :

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

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

4.6.3 Exemple 3

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 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 lexème 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**

4.6.4 Exemple 4

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]{%
  \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}

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

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

BA*C*
A**B**123

21. En toute rigueur, il aurait fallu écrire :
`\StrSubstitute[1]{#1}{\mot}{\expandafter\textbf\expandafter{\mot}}`
De cette façons, 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.

4.6.5 Exemple 5

Soit une séquence de contrôle `\myCS` définie par un `\def` contenant des séquences de contrôles et des groupes entre accolades. Essayons de trouver le n^e groupe, c'est à dire ce qui se trouve entre la n^e paire d'accolades équilibrées. On prendra comme exemple une séquence de contrôle contenant :

```
\a{1\b{2}}\c{3}\d{4\e{5}\f{6{7}}}
```

```
\newcount\occurr \newcount\nbgroup
\newcommand\findgroup[2]{%
  \scancs[1]{\chaine}{#2}%
  \occurr=0
  \nbgroup=0
  \def\findthegroup{%
    \StrBehind{\chaine}{\Obr}[\remain]%
    \advance\occurr by 1% accolade fermante suivante
    \StrBefore[\the\occurr]{\remain}{\Cbr}[\group]%
    \StrCount{\group}{\Obr}[\nbA]%
    \StrCount{\group}{\Cbr}[\nbB]%
    \ifnum\nbA=\nbB% accolades équilibrées ?
      \advance\nbgroup by 1
      \ifnum\nbgroup<#1% si c'est pas le bon groupe
        \StrBehind{\chaine}{\group}[\chaine]%
        \occurr=0% on initialise \chaine et \occur
        \expandafter\expandafter\expandafter
         \findthegroup\expandafter% et on recommence
      \fi
    \else% accolades non équilibrées ?
      % cherche accolade fermante suivante
      \expandafter\findthegroup
    \fi}
  \findthegroup
  \group}

\verbtocs{\Obr}|{|
\verbtocs{\Cbr}|}|
\def\myCS{\a{1\b{2}}\c{3}\d{4\e{5}\f{6{7}}}}
groupe 1: \findgroup{1}{\myCS}\par
groupe 2: \findgroup{2}{\myCS}\par
groupe 3: \findgroup{3}{\myCS}
```

groupe 1 : 1\b {2}
groupe 2 : 3
groupe 3 : 4\e {5}\f {6{7}}

On peut observer qu'il faut deux compteurs, deux tests et une double récursivité pour trouver le bon groupe : un de chaque pour compter quelle accolade fermante va délimiter le groupe en cours, et l'autre pour compter quel groupe est examiné.

⋆
⋆ ⋆

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