xstring

v1.0

Manuel de l'utilisateur

Christian Tellechea
unbonpetit@gmail.com

5 juillet 2008

Résumé

Cette extension regroupe un ensemble de macros manipulant des chaînes de caractères :

- ▷ 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.
- ▷ 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.

D'autres commandes permettent de traiter les caractères spéciaux (`&`, `~`, `\`, `{`, `}`, `_`, `#`, `$`, `^` et `%`) de façon à les utiliser dans les arguments des macros de ce package, ce qui permet d'utiliser ces macros à fins de programmation.

Table des matières

1 Présentation

1.1 Description

Cette extension regroupe des macros et des tests opérant sur des chaînes de caractères, 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, extractions de sous-chaînes ou de caractères, calculs de position d'une sous-chaîne, calculs du nombre d'occurrences, etc.

Certes d'autres packages 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.

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 derniers mois, et avec l'aide de contributeurs[1] 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 ça a été ma principale motivation puisque j'ai découvert LaTeX il y a moins d'un an, 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.

1.3 Fonctionnement

Dans la suite, on écrira « `texte`$_{10,11,12}$ » une suite de caractères dont les catcodes sont 10, 11 ou 12.

1.3.1 Développement des arguments

Tous les arguments des macros[2] manipulant des chaînes de caractères sont susceptibles de donner, à une certaine profondeur de développement du `texte`$_{10,11,12}$. Ainsi, *par défaut*, pour éviter des chaines d'`\expandafter` et faciliter l'usage des macros de ce package, tous les arguments sont développés au maximum avant d'être pris en compte par la macro appelée : pour cela, la commande `\fullexpandarg` est appelée par défaut.

Par exemple, si `\macro` est une macro de ce package nécessitant un argument texte et un argument nombre obligatoires, les constructions suivantes sont équivalentes :

Construction avec `\fullexpandarg`	Construction habituelle sous LaTeX ou avec `\normalexpandarg`
`\def\aa{du texte}`	`\def\aa{du texte}`
`\def\nn{2}`	`\def\nn{2}`
`\macro{\aa}{\nn}`	`\expandafter\expandafter\expandafter\macro`
	`\expandafter\expandafter\expandafter`
	`{\expandafter\aa\expandafter}\expandafter{\nn}`

La construction de gauche permet de s'affranchir de l'ordre de développement des arguments et épargne à l'utilisateur des chaînes d'`\expandafter`. En revanche, les arguments doivent être purement développables, et donner à leur développement maximum du `texte`$_{10,11,12}$ contenant ce qui est attendu (nombre ou chaîne de caractères).

On peut cependant revenir à tout moment à l'ordre habituel de développement avec la commande `\normalexpandarg`, et utiliser quand on le souhaite `\fullexpandarg` qui est appelée par défaut pour basculer à nouveau vers le développement maximum des arguments.

1. Je remercie chaleureusement Manuel alias « `mpg` » pour son aide précieuse, sa compétence et sa disponibilité.
2. À l'exeption des 2 derniers arguments des macros tests qui contiennent des séquences de contrôle à exécuter selon le résultat du test, ainsi qu'à l'exception de l'argument optionnel venant en dernière position.

1.3.2 Arguments textuels

Les macros de packages manipulant des chaînes de caractères, elles nécessitent un ou plusieurs arguments contenant du `texte`$_{10,11,12}$ (voir 1.3), en respectant la classique syntaxe entre accolades `{texte`$_{10,11,12}$`}` ou entre crochet pour les arguments optionnels `[texte`$_{10,11,12}$`]`.

Il convient de respecter les règles suivantes pour le développement des arguments textuels :
- ils peuvent contenir des lettres (majuscules ou minuscules, accentuées [3] ou non), des chiffres et des espaces, ainsi que tout caractère de catcode 10, 11 ou 12 (signes de ponctuation, signes opératoires mathématiques, parenthèses, crochets, etc). Le signe €, par contre, génèrera des erreurs.
- les espaces sont gérés et comptabilisés comme des caractères à part entière, à la condition qu'il ne se suivent pas, auquel cas, la règle LaTeXienne prévaut et plusieurs espaces consécutifs n'en font qu'un seul ;
- aucun caractère spécial n'est accepté, c'est-à-dire que les 10 caractères suivants sont strictement interdits : `&`, `~`, `\`, `{`, `}`, `_`, `#`, `$`, `^` et `%`.

Pour contourner certaines de ces règles et aller plus loin dans les possibilités des macros pour les utiliser à des fins de programmation, il existe des macros spéciales qui permettent d'inclure dans des arguments textuels n'importe quel caractère spécial, c'est-à-dire un caractère dont le catcode est différent de 10, 11 ou 12, ainsi que des séquences de contrôle en contenant. Voir la description détaillée de ce mode de fonctionnement au chapitre 3, page 9.

1.3.3 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`. Par conséquent, certaines constructions ne sont pas acceptées et conduisent à des erreurs lors de la compilation. Si par exemple, `\commande{argument}` est une macro de ce package avec son argument, supposée renvoyer une chaîne de caractères, alors les constructions suivantes ne sont pas permises :

```
\edef\Resultat{\commande{argument}}
```

pas plus que

```
\commandeA{\commandeB{\commandeC{argument}}}
```

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 le nom d'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. Cet argument optionnel, évidemment, n'est pas développé contrairement aux autres arguments. Il permet ainsi contourner les limitations évoquées dans les exemples ci dessus.

Ainsi cette construction non permise :

```
\edef\Resultat{\commande{arguments}}
```

est équivalente à

```
\commande{argument}[\Resultat]
```

Et cet autre construction non permise :

```
\commandeA{\commandeB{\commandeC{arguments}}}
```

se comporte de même que

```
\commandeC{arguments}[\MaChaine]
\commandeB{\MaChaine}[\MaChaine]
\commandeA{\MaChaine}
```

2 Les macros

2.1 Présentation des macros

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

3. 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

– 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 [4]. 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 « || ».

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}`	vrai
`\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*⟩.

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

```
\IfSubStrBehind{xstring}{ri}{xs}{vrai}{faux}  vrai
  \IfSubStrBehind{xstring}{s}{i}{vrai}{faux}  faux
\IfSubStrBehind{LaTeX}{TeX}{LaT}{vrai}{faux}  vrai
\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
```

2.3 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[5] 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}  |xs|
  \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
```

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

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 ;

`\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 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}`	|LaTeX|
`\StrGobbleLeft{a bc def }{4}`	| def |

2.3.7 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, 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}`	||
`\StrLeft{a bc def }{5}`	|a bc |

2.3.8 StrGobbleRight

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

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

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

2.3.9 StrRight

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

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

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

2.3.10 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*⟩ $\leqslant 0$ 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

2.3.11 StrMid

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

Dans ⟨*chaine*⟩, renvoie la sous chaîne se trouvant entre[6] 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}`	xs
`\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

6. 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
```

3 Utilisation des macros à des fins de programmation

3.1 Verbatimiser vers une séquence de contrôle

La macro `\verbtocs` permet le lire le contenu d'un argument « verb » qui peut contenir 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*⟩ dont les caractères spéciaux auront reçu un catcode de 12. ⟨*nom*⟩ contiendra donc du `texte`$_{10,11,12}$ (voir 1.3).

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 « verbatimiser » des caractères contenant « | », on peut changer à tout moment le caractère délimitant le contenu verb par la macro :

`\setverbdelim{`⟨*caractère*⟩`}`

Tout ⟨*caractère*⟩ ayant un catcode de 11 ou 12 peut être utilisé [7]. Par exemple, après `\setverbdelim{=}`, un argument textuel verb aura cette syntaxe `=`⟨*caractères*⟩`=`.

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

Exemple :

```
\verbtocs{\resultat}  |a   & b{ c% d$ e \f|
\resultat
```

a & b{ c% d$ e \f

3.2 Tokenisation d'un texte vers une séquence de contrôle

Le processus inverse de ce qui a été vu au dessus consiste à transformer un `texte`$_{10,11,12}$ en séquences de contrôle. Pour cela, on dispose de la macro :

`\tokenize{`⟨*nom*⟩`}{`⟨*séquence de contrôle*⟩`}`

⟨*séquence de contrôle*⟩ est développée le plus possible si l'on a invoqué `\fullexpandarg` (voir page 2) ; elle subit un 1-développement si l'on invoque `\normalexpandarg`. Dans les deux cas, le développement doit donner du `texte`$_{10,11,12}$. Ensuite, ce `texte`$_{10,11,12}$ est transformé en tokens puis assigné à l 'aide d'un `\def` à la séquence de contrôle ⟨*nom*⟩.

Exemple :

7. 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.

```
\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.

3.3 Développement d'une séquence de contrôle avant une conversion en verb

3.3.1 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 de contrôle*⟩`}`

Le ⟨*nombre*⟩ vaut 1 par défaut et représente la profondeur à laquelle va être développée la ⟨*séquence de contrôle*⟩ avant d'être transformée en caractères de catcode 12 (ou 10 pour l'espace) puis assignée à la séquence de contrôle ⟨*nom*⟩ qui contiendra donc du $\texttt{texte}_{10,12}$ (voir 1.3).

3.3.2 Attention aux catcodes !

Prenons un cas simple où la ⟨*séquence de contrôle*⟩ se développe en texte, comme dans l'exemple suivant :

```
\def\test{a b1 d}
\scancs{\resultat}{\test}
\resultat
```

a b1 d

Mais attention au catcodes !
Dans l'exemple précédent, `\scancs{\resultat}{\test}` n'est pas équivalent à `\edef\resultat{\test}`.

En effet, dans le 1er cas avec `\scancs{\resultat}{\test}`, `\resultat` contient du $\texttt{texte}_{10,12}$ (des caractères de catcode 12 et des espaces de catcode 10) et se développe en :

$$\texttt{a}_{12}\ \sqcup_{10}\ \texttt{b}_{12}\ \texttt{1}_{12}\ \sqcup_{10}\ \texttt{d}_{12}$$

Dans le le second cas avec `\edef\resultat{\test}`, `\resultat` contient des caractères de catcode 11 (les lettres), un caractère de catcode 12 (le chiffre 1) et des espaces de catcode 10, et se développe en :

$$\texttt{a}_{11}\ \sqcup_{10}\ \texttt{b}_{11}\ \texttt{1}_{12}\ \sqcup_{10}\ \texttt{d}_{11}$$

3.3.3 Profondeur de développement

Si cela s'avère nécessaire, on peut aussi contrôler la profondeur de développement avec l'argument optionnel. Ainsi, dans l'exemple suivant, au 1er appel de `\scancs`, la séquence de contrôle `\c` subit un 3-developpement qui donne « $\texttt{1}_{12}\ \sqcup_{10}\ \texttt{z}_{11}\ \sqcup_{10}\ \texttt{3}_{12}$ » qui est transformé en « $\texttt{1}_{12}\ \sqcup_{10}\ \texttt{z}_{12}\ \sqcup_{10}\ \texttt{3}_{12}$ ».

Par contre, si le n-développement donne une séquence de contrôle, alors cette séquence de contrôle est transformée en caractères de catcode 12. Ainsi, au 2e appel de `\scancs`, la ligne `\scancs[2]{\resultat}{\c}` procède au 2-développement de la séquence de contrôle `\c` qui donne la séquence de contrôle `\a` qui est transformée en « $\backslash_{12}\ \texttt{a}_{12}$ ».

L'exemple montre toutes les profondeurs de développement jusqu'à la profondeur 0 :

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

1 z 3
\a
\b
\c

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.

3.3.4 Développement de plusieurs séquences de contrôle

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. Voyons cela sur un exemple :

```
\def\a{LaTeX}
\def\b{est puissant}
\scancs*[1]{\resultat}{\a \b}
\resultat\par
\scancs*[2]{\resultat}{\a\space\b}
\resultat
```

LaTeXest puissant
LaTeXestpuissant

Tout d'abord, un message d'attention a été généré à la compilation. Voyons sa signification...

En premier lieu, on constate qu'il manque une espace entre les mots « LaTeX » et « est », bien qu'une espace ait été insérée entre les 2 séquences de contrôles `\a` et `\b`. Cela est dû à ce que, lors de la lecture de l'argument, TEX ignore les espaces situées après les séquences de contrôle. Et donc, `{\a \b}` est assimilé à `{\a\b}`, quelque soit le nombre d'espaces insérées entre `\a` et `\b`. Pour avoir notre espace entre « LaTeX » et « est », on aurait pu utiliser la séquence de contrôle `\space` qui se développe en une espace, et écrire comme 3e argument : `{\a\space\b}`. On aurait aussi pu insérer une espace à la fin de la définition de `\a` et écrire `\def\a{LaTeX }`.

Ensuite, il convient cependant de faire attention lorsqu'on utilise des profondeurs de développement supérieures à 1 : si une séquence de contrôle se n-développe en du $\texttt{texte}_{10,11,12}$, au développement suivant, les espaces seront avalées à la lecture. On le voit dans après la ligne `\scancs*[2]{\result}{\a\space\b}` où `\resultat` contient « LaTeXestpuissant » où tous les espaces ont été mangés lors du 2-développement !

De plus, et c'est aussi le sens du message d'avertissement généré par la version étoilée, il faut savoir que si le n-développement contient des accolades, celles-ci, tout comme les espaces, seront supprimées au développement suivant.

Enfin, lors de l'utilisation de `\scancs`, une espace est insérée après chaque séquence de contrôle. En effet, la primitive `\detokenize` de ε-TEX qui a été utilisée ici, insère une espace apès chaque séquence de contrôle. Il n'y a pas moyen de l'éviter.

3.3.5 Exemples

Dans l'exemple suivant, les séquences de contrôle subissent un 2-développement. `\d` donne donc `\b`, et `\b` donne `\textbf{a}\textit{b}`, et on constate bien qu'une espace est insérée après chaque séquence de contrôle.

```
\def\a{\textbf{a}\textit{b}}
\def\b{\a}
\def\c{\b}
\def\d{\c}
\scancs*[2]{\resultat}{\d\b}
\resultat
```

\b \textbf {a}\textit {b}

Voici un exemple illustrant la perte des accolades au développement suivant :

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

\a A
1{2}A
12A

Enfin, voici un dernier exemple où l'on met à profit l'espace inséré après chaque séquence de contrôle par `\scancs` pour trouver la n^{e} séquence de contrôle dans le développement d'une séquence de contrôle.

Dans l'exemple ci-dessous, on cherchera la 4e séquence de contrôle dans `\myCS` qui contient :

```
\a xy{3 2}\b7\c123 {m}\d{8}\e
```

On cherche bien évidemment à obtenir `\d` !

```
\verbtocs{\antislash}|\|
\newcommand\findcs[2]{%
 \scancs[1]{\theCS}{#2}%
 \tokenize{\theCS}{\theCS}%
 \scancs[1]{\theCS}{\theCS}%
 \StrBehind[#1]{\theCS}{\antislash}[\theCS]%
 \StrBefore{\theCS}{ }[\theCS]%
 \edef\theCS{\antislash\theCS}}                    \d
\verbtocs{\myCS}|\a xy{3 2}\b7\c123 {m}\d{8}\e|    \d
% ici, \myCS contient du texte
\findcs{4}{\myCS}
\theCS\par
\def\myCS{\a xy{3 2}\b7\c123 {m}\d{8}\e}
% ici, \myCS contient des séquences de contrôle
\findcs{4}{\myCS}
\theCS
```

Au début de la macro `\findcs`, les appels successifs à `\tokenize` puis une deuxième fois à `\scancs` sont nécessaires pour que la macro fonctionne dans le cas où l'argument est une séquence de contrôle se développant en du `texte`$_{10,11,12}$: de cette façon en « tokenisant » puis « detokenisant », des espaces seront bien insérés après chaque séquence de contrôle.

3.4 À l'interieur 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 $|\langle caractères\rangle|$ sont également interdits. Il faut donc observer la règle suivante :

Ne pas utiliser d'argument textuel verb « $|\langle texte\rangle|$ » à l'intérieur de la définition d'une macro.

Par conséquent, la macro `\verbtocs` est interdite 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. 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 13, à la fin de ce manuel.

Dans l'exemple artificiel[8] 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. Ces séquences de contrôle sont ensuite développées et utilisées à l'intérieur de la macro pour obtenir le résultat voulu :

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

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

{xstring}
{\a }

3.5 Macros étoilées

Comme `\scancs` donne du $\texttt{texte}_{10,12}$ (voir 1.3), cela pose des problèmes pour les macros listées au chapitre 2, car elles tiennent compte du catcode des caractères de leurs arguments textuels.

Voici un exemple :

```
\verbtocs{\mytext}|a b c|
\IfSubStr{\mytext}{b}{vrai}{faux}
\par
\edef\onecs{x y z}
\scancs[1]\mycs\onecs
\IfSubStr{\mycs}{y}{vrai}{faux}
```

vrai
faux

Le premier test renvoie vrai puisque la macro `\verbtocs` ne modifie pas les catcodes des caractères non spéciaux : ainsi, `\mytext` se développe en « $\texttt{a}_{11}\ \sqcup_{10}\ \texttt{b}_{11}\ \sqcup_{10}\ \texttt{c}_{11}$ » qui contient bien le deuxième argument qui est « $\texttt{b}_{11}$ ».

Pour le second test, puisque `\scancs` donne du $\texttt{texte}_{10,12}$, il renvoit faux puisque `\mycs` se développe en « $\texttt{x}_{12}\ \sqcup_{10}\ \texttt{y}_{12}\ \sqcup_{10}\ \texttt{z}_{12}$ » qui ne contient pas le deuxième argument qui est « $\texttt{y}_{11}$ ».

Pour éviter ce désagrément dû à cette différence de catcodes, et les rendre « compatibles » avec `\scancs`, les macros présentées au chapitre 2 ont toutes une version étoilée qui converti le contenu des arguments textuels en caractères dont les catcodes sont 12 uniquement, ou 10 pour les espaces :

```
\edef\onecs{x y z}
\scancs[1]\mycs\onecs
\IfSubStr*{\mycs}{y}{vrai}{faux}
```

vrai

3.6 Exemples d'utilisation en programmation

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

3.6.1 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 : **AB***C*

```
\def\myCS{\textit{A}\textit{B}\textit{C}}
\scancs[1]{\text}{\myCS}
\StrSubstitute*[2]{\text}{textit}{textbf}[\text]
\tokenize{\myCS}{\text}
\myCS
```

AB*C*

3.6.2 Exemple 2

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

```
\verbtocs{\csfrac}|\frac|%
\verbtocs{\Ob}|{|%
\verbtocs{\Cb}|}|%
\newcommand\tofrac[1]{%
  \scancs[0]{\myfrac}{#1}%
  \StrBefore{\myfrac}{/}[\num]%
  \StrBehind{\myfrac}{/}[\den]%
  \tokenize\myfrac{\csfrac\Ob\num\Cb\Ob\den\Cb}%
  $\myfrac$}
\tofrac{15/9}
\tofrac{u_{n+1}/u_n}
\tofrac{a^m/a^n}
\tofrac{x+\sqrt{x}/\sqrt{x^2+x+1}}
```

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

3.6.3 Exemple 3

Soit une phrase composée de texte. Dans une séquence de contrôle `\phrase`, essayons de mettre en gras le 1er mot qui suit le mot « package ». Nous prendrons par exemple comme phrase :

`Le package xstring est nouveau.`

```
\def\phrase{Le package xstring est nouveau.}
\def\mot{package}
\StrBehind[1]{\phrase}{\mot}[\nom]
\IfBeginWith{\nom}{ }%
  {\StrGobbleLeft{\nom}{1}[\nom]}%
  {}%
\StrBefore{\nom}{ }[\nom]
\verbtocs{\avant}|\textbf{|
\verbtocs{\apres}|}|
\StrSubstitute[1]%
  {\phrase}{\nom}{\avant\nom\apres}[\phrase]
\tokenize{\phrase}{\phrase}
\phrase
```

Le package **xstring** est nouveau.

3.6.4 Exemple 4

Soit une séquence de contrôle `\myCS` définie par un `\def` contenant des séquences de contrôles avec leurs éventuels arguments. Comment intervertir les 2 premières séquences de contrôle ? On prendra comme exemple une séquence de contrôle contenant :

`\textbf{A}\textit{B}\texttt{C}`

On cherche à obtenir une séquence de contrôle finale contenant `\textit{B}\textbf{A}\texttt{C}` et donnant : $B\mathbf{A}\mathtt{C}$

```
\def\myCS{\textbf{A}\textit{B}\texttt{C}}
\scancs[1]{\chaine}{\myCS}
\verbtocs{\antislash}|\|
\StrBefore[3]{\chaine}{\antislash}[\firsttwo]
\StrBehind{\chaine}{\firsttwo}[\others]
\StrBefore[2]{\firsttwo}{\antislash}[\avant]
\StrBehind{\firsttwo}{\avant}[\apres]%
\tokenize{\myCS}{\apres\avant\others}%
résultat : \myCS
```

résultat : $B\mathbf{A}\mathtt{C}$

3.6.5 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
        \findthegroup% et on recommence
      \fi
    \else% accolades non équilibrées ?
      % cherche accolade fermante suivante
      \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