xstring

v1.4

User's manual

Christian Tellechea
unbonpetit@gmail.com

November 4th 2008

Abstract

This package which requires ε-TEX, provides macros manipulating strings contaning chars, control sequences, groups between braces and other special tokens so that the macros may be used for programming purposes. Main features are

- ▷ tests:
  - does a string contains at least n times an another?
  - does a string starts (or ends) with another? etc.
  - is a string an integer? A decimal?
  - are 2 strings equal?
- ▷ extractions of substrings:
  - what is on the left (or the right) of the n^{th} occurrence of a substring;
  - what is between the occurrences of 2 substrings;
  - substring between 2 positions, etc.
- ▷ substitution of all, or the n first occurrences of a substring for an other substring;
- ▷ calculation of numbers:
  - length of a string;
  - position of the n^{th} occurrence of a substring;
  - how many times a string contains a substring?
  - comparison of 2 strings: position of the first difference.

Other macros allow to use special characters forbiden in arguments (`#` and `%`) and manage differences between catcodes for advanced programming purposes.

Contents

This manual is a translation of the french manual. I apologize for my poor english but I did my best[1], and I hope that the following is comprehensible!

1 Presentation

1.1 Description

This extension[2] provides macros and tests operating on strings of TEX code, as other programmation languages have. They provides the usual strings operations, such as: test if a string contains another, begins or ends with another, extractions of strings, calculation of the position of a substring, of the number of occurrences, etc.

xstring reads the arguments of the macros syntax unit by syntax unit[3] : when syntax units are "simple" chars (catcode 10, 11 and 12), xstring logically read the argument char by char. xstring can also be used for programming purpose, including in arguments other tokens such as control sequences, braces and tokens with other catcodes. See chapter on reading mode and arguments expansion (page 12), the command `\verbtocs` (page 13) and the command `\scancs` (page 14).

As the arguments may contain chars, advanced users could have problems with catcodes leading to unexpected behaviours. These behaviours can be controlled: read page 13.

Certainly, other packages exist (for example `substr` and `stringstrings`), but as well as differences on features, they do not take into account occurrences so I found them too limited and difficult to use for programming.

1.2 Motivation

I decided to write this package of macros because I have never really found tools in LATEX suiting my needs for strings. So, over the last few months, I wrote a few macros that I occasionally or regularly used. Their numbers have increased and become a little too dispersed in directories in my computer, so I have grouped them together in this package.
Thus, writing a coherent set of macros forces more discipline and leads to necessary improvements, which took most of the time I spent writing this package. This package is my first one as I recently discoverd LATEX[4], so my main motivation was to make progress in programming with TEX, and to tackle its specific methods.

2 The macros

2.1 Presentation of macros

In the following chapters, all the macros will be presented this plan:

- the syntax[5] and the value of optional arguments
- a short description of the operation;
- the operation under special conditions. For each conditions considered, the operation described has priority on that (those) below;
- finally, several examples[6] are given. I tried to find them most easily comprehensible and most representative of the situations met in normal use. If a doubt is possible with spaces in the result, this one will be delimited by "|", given that an empty string is represented by "||".

2.2 The tests

2.2.1 `\IfSubStr`

`\IfSubStr`⟨`[*]`⟩`[`⟨*number*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

The value of the optional argument ⟨*number*⟩ is 1 by default.

Tests if ⟨*string*⟩ contains at least ⟨*number*⟩ times ⟨*stringA*⟩ and runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

[1] Any email to tell me errors would be appreciated!
[2] This extension does not require LATEX and can be compiled with Plain ε-TEX.
[3] In the TEX code, a syntax unit is a control sequence, a group between brace or a single char. See also page 12.
[4] In november 2007, I will be a noob for a long time...
[5] The optional star, the optional argument in last position will be explained later. See page 13 for starred macros and page 11 for the optional argument.
[6] For much more examples, see the test file.

▷ If ⟨*number*⟩ ⩽ 0, runs ⟨*false*⟩;
▷ If ⟨*string*⟩ or ⟨*stringA*⟩ is empty, runs ⟨*false*⟩.

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

2.2.2 \IfSubStrBefore

\IfSubStrBefore⟨[*]⟩[⟨*number1*⟩,⟨*number2*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}

The values of the optional arguments ⟨*number1*⟩ and ⟨*number2*⟩ are 1 by default.

In ⟨*string*⟩, tests if the ⟨*number1*⟩$^{\text{th}}$ occurrence of ⟨*stringA*⟩ is on the left of the ⟨*number2*⟩$^{\text{th}}$ occurrence of ⟨*stringB*⟩. Runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

▷ If one of the occurrences is not found, it runs ⟨*false*⟩;
▷ If one of the arguments ⟨*string*⟩, ⟨*stringA*⟩ or ⟨*stringB*⟩ is empty, runs ⟨*false*⟩;
▷ If one of the optional arguments is negative or zero, runs ⟨*false*⟩.

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

2.2.3 \IfSubStrBehind

\IfSubStrBehind⟨[*]⟩[⟨*number1*⟩,⟨*number2*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}

The values of the optional arguments ⟨*number1*⟩ and ⟨*number2*⟩ are 1 by default.

In ⟨*string*⟩, tests if the ⟨*number1*⟩$^{\text{th}}$ occurrence of ⟨*stringA*⟩ is on the right of the ⟨*number2*⟩$^{\text{th}}$ occurrence of ⟨*stringB*⟩. Runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

▷ If one of the occurrences is not found, it runs ⟨*false*⟩;
▷ If one of the arguments ⟨*string*⟩, ⟨*stringA*⟩ or ⟨*stringB*⟩ is empty, runs ⟨*false*⟩;
▷ If one of the optional arguments is negative or zero, runs ⟨*false*⟩.

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

2.2.4 \IfBeginWith

\IfBeginWith⟨[*]⟩{⟨*string*⟩}{⟨*stringA*⟩}{⟨*true*⟩}{⟨*false*⟩}

Tests if ⟨*string*⟩ begins with ⟨*stringA*⟩, and runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

▷ If ⟨*string*⟩ or ⟨*stringA*⟩ is empty, runs ⟨*false*⟩.

```
\IfBeginWith{xstring}{xst}{true}{false}  true
    \IfBeginWith{LaTeX}{a}{true}{false}  false
\IfBeginWith{a bc def }{a b}{true}{false}  true
 \IfBeginWith{a bc def }{ab}{true}{false}  false
```

2.2.5 \IfEndWith

\IfEndWith⟨[*]⟩{⟨*string*⟩}{⟨*stringA*⟩}{⟨*Behind*⟩}{⟨*false*⟩}

Tests if ⟨*string*⟩ ends with ⟨*stringA*⟩, and runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

▷ If ⟨*string*⟩ or ⟨*stringA*⟩ is empty, runs ⟨*false*⟩.

```
\IfEndWith{xstring}{ring}{true}{false}  true
     \IfEndWith{LaTeX}{a}{true}{false}  false
\IfEndWith{a bc def }{ef }{true}{false}  true
 \IfEndWith{a bc def }{ef}{true}{false}  false
```

2.2.6 \IfInteger

\IfInteger⟨[*]⟩{⟨*number*⟩}{⟨*true*⟩}{⟨*false*⟩}

Tests if ⟨*number*⟩ is an integer, and runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

If test is false because unexpected characters, the control sequence \@xs@afterinteger contains the illegal part of ⟨*number*⟩.

```
  \IfInteger{13}{true}{false}  true
\IfInteger{-219}{true}{false}  true
  \IfInteger{+9}{true}{false}  true
\IfInteger{3.14}{true}{false}  false
   \IfInteger{0}{true}{false}  true
 \IfInteger{49a}{true}{false}  false
   \IfInteger{+}{true}{false}  false
   \IfInteger{-}{true}{false}  false
\IfInteger{0000}{true}{false}  true
```

2.2.7 \IfDecimal

\IfDecimal⟨[*]⟩{⟨*number*⟩}{⟨*true*⟩}{⟨*false*⟩}

Tests if ⟨*number*⟩ is a decimal, and runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

Counters \integerpart and \decimalpart contain the integer part and decimal part of ⟨*number*⟩.
If test is false because unexpected characters, the control sequence \@xs@afterdecimal contains the illegal part of ⟨*number*⟩, whereas if test is false because decimal part is empty after decimal separator, it contains "X".

▷ Decimal separator can be a dot or a comma;
▷ If what is on the right of decimal separator (if it exists) is empty, the test is false;
▷ If what is on the left of decimal separator (if it exists) is empty, the integer part is assumed to be 0;

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

2.2.8 \IfStrEq

\IfStrEq⟨[*]⟩{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}

Tests if the strings ⟨*stringA*⟩ and ⟨*stringB*⟩ are equal, i.e. if they contain successively the same characters in the same order. Runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

```
\IfStrEq{a1b2c3}{a1b2c3}{true}{false}  true
\IfStrEq{abcdef}{abcd}{true}{false}  false
\IfStrEq{abc}{abcdef}{true}{false}  false
\IfStrEq{3,14}{3,14}{true}{false}  true
\IfStrEq{12.34}{12.340}{true}{false}  false
\IfStrEq{abc}{}{true}{false}  false
\IfStrEq{}{abc}{true}{false}  false
\IfStrEq{}{}{true}{false}  true
```

2.2.9 \IfEq

\IfEq⟨[*]⟩{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}

Tests if the strings ⟨*stringA*⟩ and ⟨*stringB*⟩ are equal, *except* if both ⟨*stringA*⟩ and ⟨*stringB*⟩ contain numbers in which case the macro tests if these numbers are equal. Runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

- ▷ The definition of *number* is given with the macro `IfDecimal` (see page 4), and thus :
- ▷ "+" signs are optional;
- ▷ Decimal separator can be a dot or a comma;
- ▷ It is possible to evaluate an algebric expression with the ε-TEX primitive `\numexpr`, keeping in mind that it operate on integers only, and that results of divisions are *rounded* and not truncated!

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

2.2.10 \IfStrEqCase

```
\IfStrEqCase⟨[*]⟩{⟨string⟩}{%
    {⟨string1⟩}{⟨code1⟩}%
    {⟨string2⟩}{⟨code2⟩}%
    etc...
    {⟨stringN⟩}{⟨codeN⟩}}[⟨other cases code⟩]
```

Tests successively if ⟨*string*⟩ is equal to ⟨*string1*⟩, ⟨*string2*⟩, etc. Comparison is made with `\IfStrEq` (see above). If the test i is positive (the ⟨*string*⟩ matches ⟨*string i*⟩), the macro runs ⟨*code i*⟩ and ends. If all tests fail, the macro runs the optional ⟨*other cases code*⟩, if present.

```
\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}}[other]  CC
\IfStrEqCase{d}{{a}{AA}{b}{BB}{c}{CC}}[other]  other
\IfStrEqCase{+3}{{1}{one}{2}{two}{3}{three}}[other]  other
\IfStrEqCase{0.5}{{0}{zero}{.5}{half}{1}{one}}[other]  other
```

2.2.11 \IfEqCase

\IfEqCase⟨[*]⟩{⟨*string*⟩}{%
{⟨*string1*⟩}{⟨*code1*⟩}%
{⟨*string2*⟩}{⟨*code2*⟩}%
etc...
{⟨*stringN*⟩}{⟨*codeN*⟩}}[⟨*other cases code*⟩]

Tests successively if ⟨*string*⟩ is equal to ⟨*string1*⟩, ⟨*string2*⟩, etc. Comparison is made with \IEq (see above). If the test i is positive (the ⟨*string*⟩ matches ⟨*string* i⟩), the macro runs ⟨*code* i⟩ and ends. If all tests fail, the macro runs the optional ⟨*other cases code*⟩, if present.

\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}}[other]	CC
\IfEqCase{d}{{a}{AA}{b}{BB}{c}{CC}}[other]	other
\IfEqCase{+3}{{1}{one}{2}{two}{3}{three}}[other]	three
\IfEqCase{0.5}{{0}{zero}{.5}{half}{1}{one}}[other]	half

2.3 Extraction of substrings

2.3.1 \StrBefore

\StrBefore⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]

The value of the optional argument ⟨*number*⟩ is 1 by default.

In ⟨*string*⟩, returns what is leftwards the ⟨*number*⟩$^{\text{th}}$ occurrence of ⟨*stringA*⟩.

- ▷ If ⟨*string*⟩ or ⟨*stringA*⟩ is empty, an empty string is returned;
- ▷ If ⟨*number*⟩ < 1 then the macro behaves as if ⟨*number*⟩ $= 1$;
- ▷ If the occurrence is not found, an empty string is returned.

\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⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]

The value of the optional argument ⟨*number*⟩ is 1 by default.

In ⟨*string*⟩, returns what is rightwards the ⟨*number*⟩$^{\text{th}}$ occurrence of ⟨*stringA*⟩.

- ▷ If ⟨*string*⟩ or ⟨*stringA*⟩ is empty, an empty string is returned;
- ▷ If ⟨*number*⟩ < 1 then the macro behaves as if ⟨*number*⟩ $= 1$;
- ▷ If the occurrence is not found, an empty string is returned.

\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⟨[*]⟩[⟨*number1*⟩,⟨*number2*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*stringB*⟩}[⟨*name*⟩]

The values of the optional arguments ⟨*number1*⟩ and ⟨*number2*⟩ are 1 by default.

In ⟨*string*⟩, returns the substring between[7] the ⟨*number1*⟩$^{\text{th}}$ occurrence of ⟨*stringA*⟩ and ⟨*number2*⟩$^{\text{th}}$ occurrence of ⟨*stringB*⟩.

- ▷ If the occurrences are not in this order — ⟨*stringA*⟩ *followed by* ⟨*stringB*⟩ — in ⟨*string*⟩, an empty string is returned;
- ▷ If one of the 2 occurrences doesn't exist in ⟨*string*⟩, an empty string is returned;
- ▷ If one of the optional arguments ⟨*number1*⟩ ou ⟨*number2*⟩ is negative or zero, an empty string is returned.

```
\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⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*stringB*⟩}[⟨*name*⟩]

The value of the optional argument ⟨*number*⟩ is 0 by default.

In ⟨*string*⟩, substitute the ⟨*number*⟩ first occurrences of ⟨*stringA*⟩ for ⟨*stringB*⟩, except if ⟨*number*⟩ = 0 in which case *all* the occurrences are substituted.

- ▷ If ⟨*string*⟩ is empty, an empty string is returned;
- ▷ If ⟨*stringA*⟩ is empty or doesn't exist in ⟨*string*⟩, the macro is ineffective;
- ▷ If ⟨*number*⟩ is greater than the number of occurrences of ⟨*stringA*⟩, then all the occurrences are substituted;
- ▷ If ⟨*number*⟩ < 0 the macro behaves as if ⟨*number*⟩ = 0;
- ▷ If ⟨*stringB*⟩ is empty, the occurrences of ⟨*stringA*⟩, if they exist, are deleted.

```
\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⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]

The value of the optional argument ⟨*number*⟩ is 0 by default.

Delete the ⟨*number*⟩ first occurrences of ⟨*stringA*⟩ in ⟨*string*⟩, except if ⟨*number*⟩ = 0 in which case *all* the occurrences are deleted.

- ▷ If ⟨*string*⟩ is empty, an empty string is returned;
- ▷ If ⟨*stringA*⟩ is empty or doesn't exist in ⟨*string*⟩, the macro is ineffective;
- ▷ If ⟨*number*⟩ greater then the number of occurrences of ⟨*stringA*⟩, then all the occurrences are deleted;
- ▷ If ⟨*number*⟩ < 0 the macro behaves as if ⟨*number*⟩ = 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
```

[7] In a strict sense, i.e. *without* the strings ⟨*stringA*⟩ and ⟨*stringB*⟩

2.3.6 \StrSplit

\StrSplit⟨[*]⟩{⟨*string*⟩}{⟨*number*⟩}{⟨*csA*⟩}{⟨*csB*⟩}

The ⟨*string*⟩, is splitted after the character at position ⟨*number*⟩. The left part is assigned to the control sequence ⟨*csA*⟩ and the right part is assigned to ⟨*csB*⟩.
This macro returns two strings, so it does *not* display anything. Consequently, it does not provide the optional argument in last position.

- ▷ If ⟨*number*⟩ ⩽ 0, ⟨*csA*⟩ is empty and ⟨*csB*⟩ is equal to ⟨*string*⟩;
- ▷ If ⟨*number*⟩ ⩾⟨*lengthString*⟩, ⟨*csA*⟩ is equal to ⟨*string*⟩ and ⟨*csB*⟩ is empty;
- ▷ If ⟨*string*⟩ is empty, ⟨*csA*⟩ and ⟨*csB*⟩ are empty, whatever be the integer ⟨*number*⟩.

```
\StrSplit{abcdef}{4}{\aa}{\bb}results:  |\aa| and |\bb|  results: |abcd| and |ef|
\StrSplit{a b c }{2}{\aa}{\bb}results:  |\aa| and |\bb|  results: |a | and |b c |
\StrSplit{abcdef}{1}{\aa}{\bb}results:  |\aa| and |\bb|  results: |a| and |bcdef|
\StrSplit{abcdef}{5}{\aa}{\bb}results:  |\aa| and |\bb|  results: |abcde| and |f|
\StrSplit{abcdef}{9}{\aa}{\bb}results:  |\aa| and |\bb|  results: |abcdef| and ||
\StrSplit{abcdef}{-3}{\aa}{\bb}results:  |\aa| and |\bb|  results: |abcdef| and ||
```

2.3.7 \StrGobbleLeft

\StrGobbleLeft⟨[*]⟩{⟨*string*⟩}{⟨*number*⟩}[⟨*name*⟩]

In ⟨*string*⟩, delete the ⟨*number*⟩ first characters on the left.

- ▷ If ⟨*string*⟩ is empty, an empty string is returned;
- ▷ If ⟨*number*⟩ ⩽ 0, no character is deleted;
- ▷ If ⟨*number*⟩ ⩾ ⟨*lengthString*⟩, all the characters are deleted.

```
\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⟨[*]⟩{⟨*string*⟩}{⟨*number*⟩}[⟨*name*⟩]

In ⟨*string*⟩, returns the ⟨*number*⟩ first characters on the left.

- ▷ If ⟨*string*⟩ is empty, an empty string is returned;
- ▷ If ⟨*number*⟩ ⩽ 0, no character is returned;
- ▷ If ⟨*number*⟩ ⩾ ⟨*lengthString*⟩, all the characters are returned.

```
\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⟨[*]⟩{⟨*string*⟩}{⟨*number*⟩}[⟨*name*⟩]

In ⟨*string*⟩, delete the ⟨*number*⟩ last characters on the right.

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

2.3.10 \StrRight

\StrRight⟨[*]⟩{⟨*string*⟩}{⟨*number*⟩}[⟨*name*⟩]

In ⟨*string*⟩, returns the ⟨*number*⟩ last characters on the right.

```
\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⟨[*]⟩{⟨*string*⟩}{⟨*number*⟩}[⟨*name*⟩]

Returns the character at the position ⟨*number*⟩ in ⟨*string*⟩.

- ▷ If ⟨*string*⟩ is empty, no caracter is returned;
- ▷ If ⟨*number*⟩ ⩽ 0 or if ⟨*number*⟩ > ⟨*lengthString*⟩, no character is returned.

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

2.3.12 \StrMid

\StrMid⟨[*]⟩{⟨*string*⟩}{⟨*numberA*⟩}{⟨*numberB*⟩}[⟨*name*⟩]

In ⟨*string*⟩, returns the substring between[8] the positions ⟨*numberA*⟩ and ⟨*numberB*⟩.

- ▷ If ⟨*string*⟩ is empty, an empty string is returned;
- ▷ If ⟨*numberA*⟩ > ⟨*numberB*⟩, an empty string is returned;
- ▷ If ⟨*numberA*⟩ < 1 and ⟨*numberB*⟩ < 1 an empty string is returned;
- ▷ If ⟨*numberA*⟩ > ⟨*lengthString*⟩ et ⟨*numberB*⟩ > ⟨*lengthString*⟩, an empty string is returned;
- ▷ If ⟨*numberA*⟩ < 1, the macro behaves as if ⟨*numberA*⟩ = 1;
- ▷ If ⟨*numberB*⟩ > ⟨*lengthString*⟩, the macro behaves as if ⟨*numberB*⟩ = ⟨*lengthString*⟩.

```
 \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 Number results

2.4.1 \StrLen

\StrLen⟨[*]⟩{⟨*string*⟩}[⟨*name*⟩]

Return the length of ⟨*string*⟩.

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

2.4.2 \StrCount

\StrCount⟨[*]⟩{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]

Counts how many times ⟨*stringA*⟩ is contained in ⟨*string*⟩.

- ▷ If one at least of the arguments ⟨*string*⟩ or ⟨*stringA*⟩ is empty, the macro return 0.

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

[8] In the broad sense, i.e. that the strings characters of the "border" are returned.

2.4.3 \StrPosition

\StrPosition⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]

The value of the optional argument ⟨*number*⟩ is 1 by default.

In ⟨*string*⟩, returns the position of the ⟨*number*⟩$^{\text{th}}$ occurrence of ⟨*stringA*⟩.

▷ If ⟨*number*⟩ is greater than the number of occurrences of ⟨*stringA*⟩, then the macro returns 0;
▷ If ⟨*string*⟩ doesn't contain ⟨*stringA*⟩, then the macro returns 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⟨[*]⟩{⟨*stringA*⟩}{⟨*stringB*⟩}[⟨*name*⟩]

This macro has 2 tolerances: the "normal" tolerance, used bu default, and the "strict" tolerance.

- The normal tolerance, activated with \comparenormal.

  The macro compares characters from left to right in ⟨*stringA*⟩ and ⟨*stringB*⟩ until a difference appears or the end of the shortest string is reached. The position of the first difference is returned and if no difference is found, the macro return 0.

- The strict tolerance, activated with \comparestrict.

  The macro compares the 2 strings. If they are equal, it returns 0. If not, the position of the first difference is returned.

It is possible to save the comparison mode with \savecomparedmode, then modify this comparison mode and come back to the situation when it was saved with \restoreecomparemode.

Examples with the normal tolerance:

```
\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
```

Examples with the strict tolerance:

```
\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 Operating modes

3.1 Expansion of arguments

3.1.1 The commands \fullexpandarg, \expandarg and \noexpandarg

The command `\fullexpandarg` is called by default, so all the arguments are fully expanded (an `\edef` is used) before the the macro is called. In most of the cases, this expansion mode avoids chains of `\expandafter` and allows lighter code.
Of course, the expansion of argument can be canceled to find back the usual behaviour of TEX with the comands `\noexpandarg` or `\normalexpandarg`.

An other expansion mode can be called with `\expandarg`. In this case, the **first token** of each argument is expanded *one time* while all other tokens are left unchanged (for the expansion of all tokens one time, you should call the macro `\scancs*`, see page 14).

The commands `\fullexpandarg`, `\noexpandarg`, `\normalexpandarg` and `\expandarg` can be called at any moment in the code; they behave as "switches". They can be locally used in a group.

It is possible to save the expansion mode with `\saveexpandmode`, then modify this expansion mode and come back to the situation when it was saved with `\restoreexpandmode`.

3.1.2 Chars allowed in arguments

First of all, whatever be the current expansion mode, **tokens with catcode 6 and 14 (usually # and %) are forbidden in all the arguments**[9].

When full expansion mode is activated with `\fullexpandarg`, arguments are expanded with an `\edef` before they are read by the macro. Consequently, are allowed in arguments :

- letters (uppercase or lowercase, accented[10] or not), figures, spaces, and any other character with a catcode of 10, 11 ou 12 (punctuation signs, calculation signs, parenthesis, square bracket, etc).;
- tokens with catcode 1 to 4, usually : `{ }`[11] `$ &`
- tokens with catcode 7 and 8, usually : `^ _`
- any purely expandable control sequence[12] or tokens with catcode 13 (active chars) whose expansion is allowed chars.

On the other hand, some chars[13] like €, ¤, ¶, etc. will provoke errors.

When expansion is not full (`\expandarg` or `\noexpandarg` are active), allowed char in arguments are:

- those cited above;
- any control sequence or token catocode 13, even undefined;
- the special chars (€, ¤, ¶, etc.).

3.2 Expansion of macros, optional argument

The macros of this package are not purely expandable, i.e. they cannot be put in the argument of an `\edef`. Nestling macros is not possible neither, even with `\expandafter`.

For this reason, all the macros returning a result (i.e. all excepted the tests and `\StrSplit`) have an optional argument in last position. The syntax is `[`⟨*name*⟩`]`, where ⟨*name*⟩ is the name of the control sequence that will receive the result of the macro: the assignment is made with an `\edef` which make the result of the macro ⟨*name*⟩ purely expandable. Of course, if an optional argument is present, the macro does not display anything.

Thus, this structure not allowed, supposed to assign to `\Result` the 4 chars on the left of `xstring`:

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

is equivalent to :

[9] Maybe, the token # will be allowed in a further version.
[10] For a reliable operation with accented letters, the `\fontenc` package with option `[T1]` and `\inputenc` with appropriated option must be loaded
[11] Warning : braces **must** be balanced in arguments !
[12] i.e. this control sequence can be `\edef`ed.
[13] These chars are obtained with the keys AltGr + letter under GNU/Linux distributions.

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

And this not allowed nested structure, supposed to remove the first and last char of `xstring`:

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

can be written like this:

```
\StrGobbleRight{xstring}{1}[\mystring]
\StrGobbleleft{\mystring}{1}
```

3.3 How xstring reads the arguments?

3.3.1 Syntax unit by syntax unit

The macros of xstring read their arguments syntax unit par syntax unit. In the TEX code, a syntax unit is either:

- a control sequence;
- a group, i.e. what is between 2 balanced braces;
- a char.

Let's see what is a syntax unit with an example. Let's take this argument : `"ab\textbf{xyz}cd"`
It has 6 syntax units: `"a"`, `"b"`, `"\textbf"`, `"{xyz}"`, `"c"` and `"d"`.

What will happen while `\noexpandarg` is active, we ask xstring to find the length of this argument and find its 4^{th} "char'?

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

6
macro:->{xyz}

It is necessary to use `\meaning` to see the real expansion of `\mychar`, and not simply call `\mychar` which make loose informations (braces here). We do not obtain a "char" but a syntax unit, as expected.

3.3.2 Exploration of groups

By default, the command `\noexploregroups` is called, so in the main argument, xstring does not look into groups, and simply consider them as syntax unit.
For specific uses, it can be necessary to look into groups: `\exploregroups` changes the exploration mode an makes the macros look inside groups.

What does this exploration mode in the preceding example? xstring does not count the group as a single syntax unit but look inside it and counts the syntax unit found inside (`x`, `y` and `z`), and so on if there were several nested groups:

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

8
macro:->x

Exploring the groups can be usefull for counting a substring in a string (`\StrCount`), for the position of a substring in a string (`\StrPosition`) or for tests, but has a severe limitation with macros returning a string: when an argument is cut inside a group, **the result does not take into account what is outside this group**. This exploration mode must be used knowingly this limitation when calling macros returning a string.

Let's see what this means with an example. We want to know what is on the left of the second appearance of `\a` in the argument `\a1{\b1\a2}\a3`. As groups are explored, this appearance is inside this group : `{\b1\a2}`. The result will be `\b1`. Let's check:

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

macro:->\b 1

Exploring the groups[14] can change the behaviour of most of the macros of xstring, excepted these macros untouched by the exploration mode; their behaviour is the same in any case: `\IfInteger`, `\IfDecimal`, `\IfStrEq`, `\StrEq` et `\StrCompare`.

[14]The file test of xstring has many examples underlining differences between exploration modes.

Moreover, 2 macros run in `\noexploregroups` mode, whatever be the current mode: `\StrBetween` et `\StrMid`.

It is possible to save the "explore mode" with `\saveexploremode`, then modify this "explore mode" and come back to the situation when it was saved with `\restoreexploremode`.

3.4 Catcode of arguments, starred macros

Macros of this package take the catcodes of characters into account. To avoid unexpected behaviour (particulary with tests), you should keep in mind that characters *and their catcodes* are examined.

For instance, these two arguments:

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

do *not* expand into equal strings for xstring! Because of the command `\string`, the first expands into "`ab`" with catcodes 12 while the second have characters with their natural catcodes 11. Catcodes do not match! It is necessary to be aware of this, particulary with TEX commands like `\string` whose expansions are a strings with char catcodes 12 and 10 : `\detokenize`, `\meaning`, `\jobname`, `\fontname`, `\romannumeral`, etc.

Starred macros do not take catcodes into account. They simply convert their textual arguments into arguments with catcodes 10, 11 and 12, and call the non-starred macros with these modified arguments.

Warning: the use of a strarred macro has consequences! The arguments are "detokenized", thus, there is no more control sequences, groups, neither any special char: everything is converted into chars with "harmless" catcodes.

Here is an example:

```
\IfStrEq{\string a\string b}{ab}{true}{false}\par
\IfStrEq*{\string a\string b}{ab}{true}{false}
```

false
true

In the example above, after expansion (assumed that `\fullexpandarg` is active), the first argument contains `{ab}` where, because of `\string`, both chars have catcode 12. The second argument is `{ab}` where both chars have their natural catcode 11. Strings are not equal because of the catcode and the test is negative, if the unstarred version is used.

For the macros returning a string, if the starred version is used, the result will be a string in which chars have catcodes 12 and 10 for space. For example, after a "`\StrBefore*{a \b c d}{c}[\mytext]`", the control sequence `\mytext` expands to "$\texttt{a}_{12\sqcup 10}\texttt{\textbackslash}_{12}\texttt{b}_{12\sqcup 10}$".

4 Other macros for assistance in programming

Though xstring is able to read arguments containing TEX or LATEX code, for some advanced programming needs, it can be insufficient. This chapter presents other macros able to get round some limitations.

4.1 Assign a verb content, the macro `\verbtocs`

The macro `\verbtocs` allow to read the content of a "verb" argument containing special characters: `&`, `~`, `\`, `{`, `}`, `_`, `#`, `$`, `^` et `%`. The catcodes of "normal" characters are left unchanged while special characters take a catcode 12. Then, these characters are assigned to a control sequence. The syntax is:

`\verbtocs{`⟨*name*⟩`}|`⟨*characters*⟩`|`

⟨*name*⟩ is the name of the control sequence receiving, with an `\edef`, the ⟨*characters*⟩. ⟨*name*⟩ thus contains char with catcodes 12 or 10 for space.

By default, the character delimiting the verb content is "|". Obviously, this character cannot be both delimiting and being contained into what it delimits. If you need to verbatimize strings containing "|", you can change at any time the character delimiting the verb content with the macro:

`\setverbdelim{`⟨*character*⟩`}`

Any ⟨*character*⟩ with a catcode 11 or 12 can be used[15]. For example, after `\setverbdelim{=}`, a verb argument look like this: `=`⟨*characters*⟩`=`.

About verb arguments, keep in mind that:

- all the characters before `|`⟨*characters*⟩`|` are ignored;

[15] Several characters can be used, but the syntax of `\verbtocs` becomes less readable ! For this reason, a warning occurs when the argument of `\setverbdelim` contains more than a single character.

- inside the verb argument, all the spaces are taken into account, even if they are consecutive.

Example:

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

a & b{ c% d$ e \f

4.2 Tokenization of a text to a control sequence, the macro \tokenize

The reverse process of what has been seen above is to transform chars into tokens. This is done by the macro:

\tokenize{⟨*name*⟩}{⟨*control sequences*⟩}

⟨*control sequences*⟩ is fully expanded if \fullexpandarg has been called, and is not expanded if \noexpandarg or \expandarg are active. After expansion, the chars are tokenized to tokens and assigned to ⟨*name*⟩ with a \def.

Example:

```
\verbtocs{\text}|\textbf{a} $\frac{1}{2}$|
text: \text
\tokenize{\result}{\text}
\par
result: \result
```

text: \textbf{a} $\frac{1}{2}$
result: **a** $\frac{1}{2}$

Obviously, the control sequence \result can be called at the last line since the control sequences it contains are defined.

4.3 Expansion of a control sequence before verbatimize, the macro \scancs

It is possible to expand a control sequence before converting this expansion into text. This is done by the macro:

\scancs[⟨*number*⟩]{⟨*name*⟩}{⟨*control sequence*⟩}

⟨*number*⟩ = 1 by default and represents the number of times ⟨*control sequence*⟩ will be expanded before being converted in characters with catcodes 12 (or 10 for spaces). These characters are then assigned to ⟨*name*⟩.
If necessary, the depth of expansion can be controlled with the optional argument. If the n^{th} expansion is a control sequence, the control sequence is verbatimized into chars catcodes 12. The following shows all the "depths" of expansion, from 0 to 3:

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

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

Obviously, it is necessary to ensure that the expansion to the desired depth is possible.

In normal use, the third argument ⟨*control sequence*⟩ (or one of its expansions) must contain a single control sequence that will be expanded. If this third argument or one of its expansion contains several control sequences, compilation stops with an error message asking you to use the starred version. This starred version, more difficult to use allows to expand ⟨*number*⟩ times *all* the control sequences contained in the third argument. It is necessary to keep in mind that if the $n-1^{\text{th}}$ expansion contains a group between braces, this group will be expanded at the n^{th} expansion and will loose its braces! It is the same for spaces[16].

This is an example that shows the deletion of braces during the next expansion:

[16] Any call to \scancs* provoque a warning message warning against this behaviour.

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

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

4.4 Inside the definition of a macro

Some difficulties arise inside the definition of a macro, i.e. between braces following a `\def\macro` or a `\newcommand\macro`.

It is forbidden to use the command `\verb` inside the definition of a macro. For the same reasons:

Do not use `\verbtocs` inside the definition of a macro.

But then, how to manipulate special characters and "verbatimize" inside the définition of macros ?

The `\detokenize` primitive of ε-TEXcan be used but it has limitations:

- braces must be balanced;
- consecutive spaces make a single space;
- the `%` sign is not allowed;
- a space is inserted after each control sequence;
- `#` signs become `##`.

It is better to use `\scancs` and define *outside the definition of the macros* control sequences containing special characters with `\verbtocs`. It is also possible to use `\tokenize` to transform the final result (which is generaly $\texttt{text}_{10,11,12}$) into control sequences. See example using these macros at the end of this manual, page 16.

In the following teaching example[17], the macro `\bracearg` adds braces to its argument. To make this possible, 2 control sequences `\Ob` and `\Cb` containing `"{"` and `"}"` are defined outside the definition of `\bracearg`, and expanded inside it:

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

{xstring}
{\a }

4.5 The macro `\StrRemoveBraces`

Advanced users may need to remove the braces of an argument. The macro `\StrRemoveBraces` does this. Its syntax is:

`\StrRemoveBraces{`⟨*stringe*⟩`}[`⟨*name*⟩`]`

This macro is sensitive to exploration mode and will remove *all* the braces with `\exploregroups` while it will remove braces of higher level with `\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
```

[17] It is possible to make much more simple using `\detokenize`. The macro becomes:
`\newcommand\bracearg[1]{\detokenize{{#1}}}`

4.6 Examples

Here are some very simple examples involving the macros of this package in programming purposes.

4.6.1 Example 1

We want to substitute the 2 first `\textit` by `\textbf` in the control sequence `\myCS` which contains

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

We expect: **AB***C*

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

AB*C*

4.6.2 Example 2

Let's try to write a macro `\tofrac` that transforms an argument of this type "a/b" into "$\frac{a}{b}$":

First of all, let's cancel the expansion of arguments with `\noexpandarg`, we do not need expansion here. Then, it's easy to cut what is before and behind the first occurrence of "/" (assumed there is a single occurrence) and assign it to `\num` and `\den` and simply call the 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 Example 3

Let's try to write a macro `\boldafter` which writes in bold the first word that follows the word contained in the expansion of `\word`.

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

\boldafter{The xstring package is new}{xstring}

\boldafter{The xstring package is new}{ring}

\boldafter{The xstring package is new}{is}
```

The xstring **package** is new
The xstring package is new
The xstring package is **new**

4.6.4 Example 4

A control sequence `\myCS` défined with an `\def` contains control sequences with their possible arguments. How to reverse the order of the 2 first control sequences? For this, a macro `\swaptwofirst` does the job and displays the result. But this time, it is not possible to seek the token `\` (catcode 0) with the macros of xstring. This is why the use of `\scancs` is necessary: after the detokenization of the argument, it becomes possible to search the char `\` (catcode 12). After 4 lines, the process made by the macros of xstring (`\StrBefore` and `\StrBehind`) is finish, a retokenization is done by `\tokenize` and `\before` and `\after` are swapped at this moment.

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

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

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

BA*C*

A**B**123

4.6.5 Example 5

A control sequence `\myCS` defined with an `\edef` contains control sequences and "groups" between braces. Let's try to find the n^{th} group, i.e. what is between the n^{th} pair of balanced braces. In this example, `\myCS` contains:

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

```
\newcount\occurr
\newcount\nbgroup
\newcommand\findgroup[2]{%
  \scancs[1]{\text}{#2}%
  \occurr=0
  \nbgroup=0
  \def\findthegroup{%
    \StrBehind{\text}{\Obr}[\remain]%
    \advance\occurr by 1% next "{"
    \StrBefore[\the\occurr]{\remain}{\Cbr}[\group]%
    \StrCount{\group}{\Obr}[\nbA]%
    \StrCount{\group}{\Cbr}[\nbB]%
    \ifnum\nbA=\nbB% balanced braces ?
      \advance\nbgroup by 1
      \ifnum\nbgroup<#1% not the good group ?
        \StrBehind{\text}{\group}[\text]%
        \occurr=0% initialise \text & \occur
        \findthegroup% do it again
      \fi
    \else% unbalanced braces ?
      % look for next "}"
      \findthegroup
    \fi}
  \findthegroup
  \group}

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

group 1: \findgroup{1}{\myCS}\par
group 2: \findgroup{2}{\myCS}\par
group 3: \findgroup{3}{\myCS}
```

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

Notice that 2 counters, 2 tests and a double recursion are necessary to find the group: one of each to find which "`}`" delimits the end of the current group, and the others to calculate the number of the group being read.

⋆
⋆ ⋆

That's all, I hope you will find this package useful !
Please, send me an email if you find a bug or if you have any idea of improvement...

Christian Tellechea