xstring

v1.3a

User's manual

Christian Tellechea
unbonpetit@gmail.com

September 29th 2008

Abstract

This package which requires $\varepsilon-$TEX groups together macros manipulating strings, such as:

- ▷ 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.

For programming purposes, other macros allow to use special characters (`&`, `~`, `\`, `{`, `}`, `_`, `#`, `$`, `^` and `%`) with the macros manipulating strings.

Contents

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

1 Presentation

1.1 Description

This extension[1] provides macros and tests operating on strings, 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.

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.

There are 2 forms of each command of this package : the regular one and the starred one. The difference is that the regular *do take care of catcodes* of the characters in the string, while the starred ones, *less strict*, don't. For most users, both should behave the same way (for advanced user, read more at page 3 and page 15).

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 discoverd LaTeX less than a year ago, so my main motivation was to make progress in programming with TeX, and to tackle its specific methods.

1.3 Operation

In the following, "`text`$_{10,11,12}$" means a string made of characters whose catcodes are 10, 11 or 12.

1.3.1 Expansion of arguments

All the arguments of the macros operating on strings[2] are supposed, after a number of times of expansion, to expand to `text`$_{10,11,12}$. By *default*, to avoid many `\expandafter` and to ease the use of macros, all the arguments are fully expanded before being taken into account by the macro: for this, `\fullexpandarg` is called by default.

For example, if `\macro` is a macro of this package requiring 2 arguments (text for the first and a number for the second), the following structures are equivalent:

Structure with `\fullexpandarg`

```
\def\aa{some text}
\def\nn{2}
\macro{\aa}{\nn}
```

Usual structure with LaTeX or with `\normalexpandarg`

```
\def\aa{some text}
\def\nn{2}
\expandafter\expandafter\expandafter\macro
 \expandafter\expandafter\expandafter
  {\expandafter\aa\expandafter}\expandafter{\nn}
```

The structure on the left allow to forget the order of expansion and avoid writing many `\expandafter`. On the other hand, the arguments must be purely expandable into `text`$_{10,11,12}$ containing what is expected by the macro (number or string).

However, at any time, you can find the usual order of expansion with the macro `\normalexpandarg`, and use again `\fullexpandarg` if you want a full expansion of the arguments.

[1] This extension does not require LaTeX and can be compiled with Plain ε-TeX.
[2] Excepted the 2 last arguments of the tests.

1.3.2 Textual arguments

The macros operating on strings require one or several arguments containing – or whose expansion contains – `text`$_{10,11,12}$ (see 1.3), using the usual syntax `{text`$_{10,11,12}$`}`, and for optional arguments `[text`$_{10,11,12}$`]`.

The following rules shoud be observed for the expansion of textual arguments:

- they can contain letters (uppercase or lowercase, accented[3] or not), figures, spaces, and any other character with a catcode of 10, 11 ou 12 (punctuation signs, calculation signs, parenthesis, square bracket, etc). On the other hand, the € sign is not allowed.
- spaces are taken into account as normal characters, except if several spaces follows in which case the LaTeX rule prevails and they become a single space;
- no special character is allowed, i.e. the 10 following characters are strictly forbiden: `&`, `~`, `\`, `{`, `}`, `_`, `#`, `$`, `^` and `%`.

To circumvent some of these rules and to go further in the use of the macros operating on strings, this package provides special macros that enable special characters in textual arguments. See the detailed description of this modus operandi in chapter 3, page 12.

1.3.3 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`. Consequently, some structures are not allowed and lead to errors when compiling. If, for example, `\command{argument}` is a macro of this package operating on strings and returning a string, the following structures are not allowed:

```
\edef\Result{\command{argument}}
```

or this nested structure

```
\commandA{\commandB{\commandC{argument}}}
```

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, already seen above:

```
\edef\Result{\command{arguments}}
```

is equivalent to:

```
\command{argument}[\Result]
```

And this nested one:

```
\commandA{\commandB{\commandC{arguments}}}
```

can be replaced by:

```
\commandC{arguments}[\MyString]
\commandB{\MyString}[\MyString]
\commandA{\MyString}
```

1.3.4 Catcode of arguments

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 command like `\string` which expansion is a string with 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 12 and 10, and call the non-starred macros with these modified arguments. For more information about this, read page 15.

[3] For a reliable operation with accented letters, the `\fontenc` package with option `[T1]` and `\inputenc` with appropriated option must be loaded

2 The macros

2.1 Presentation of macros

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

- the syntax 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 are given. I tried to find them most easily comprehensible and most representative of the situations met in normal use[4]. 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.

▷ If ⟨*number*⟩ $\leqslant 0$, 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
```

[4]For more examples, see the test file.

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 6), 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[5] 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 1 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

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

`\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 1 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

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: || and |abcdef|

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}`	|LaTeX|
`\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}   ||
\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}   ||
  \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}   |xstring|
  \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[6] 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*⟩.

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

`\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 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

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}`	1
`\StrPosition{abracadabra}{z}`	1
`\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.

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 Using the macros for programming purposes

3.1 Verbatimize to a control sequence

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*⟩. Consequently, ⟨*name*⟩ contains $\texttt{text}_{10,11,12}$ (see 1.3).

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 characters 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[7]. 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;
- 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

3.2 Tokenization of a text to a control sequence

The reverse process of what has been seen above is to transform a $\texttt{text}_{10,11,12}$ into control sequences. This is done by the macro:

`\tokenize{`⟨*name*⟩`}{`⟨*control sequence*⟩`}`

⟨*control sequence*⟩ is fully expanded if `\fullexpandarg` has been called (see page 2), and is not expanded if `\normalexpandarg` has benn called. In both cases, the expansion must be $\texttt{text}_{10,11,12}$. Then, this $\texttt{text}_{10,11,12}$ is converted into tokens and assigned with a `\def` to the control sequence ⟨*name*⟩.

[7] 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.

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.

3.3 Expansion of a control sequence before verbatimize

3.3.1 The scancs macro

It is possible to expand n times 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*⟩.

3.3.2 Mind the catcodes !

Let's take a simple example where ⟨*control sequence*⟩ expands to text:

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

a b1 d

But mind the catcodes !
In this example, `\scancs{\result}{\test}` is not equivalent to `\edef\result{\test}`.

Indeed, with `\scancs{\resultat}{\test}`, `\result` contains $\texttt{text}_{10,12}$ and expands to:

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

With `\edef\resultat{\test}`, `\resultat` contains $\texttt{text}_{10,11,12}$, i.e. characters whose catcodes are 11 (the letters), 12 (the figure 1) and 10 (the spaces). It expands to:

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

3.3.3 Depth of expansion

If necessary, the number of expansions can be controled with the optional argument. In the following example, when `\scancs` is called the first time, `\c` is expanded 3 times and gives "$\texttt{1}_{12}\ \sqcup_{10}\ \texttt{z}_{11}\ \sqcup_{10}\ \texttt{3}_{12}$" which is converted into "$\texttt{1}_{12}\ \sqcup_{10}\ \texttt{z}_{12}\ \sqcup_{10}\ \texttt{3}_{12}$".

On the other hand, if after n expansions, the result is a control sequence, this control sequence is transformed into characters with catcodes 12. In the example above, when `\scancs` is called the second time, `\scancs[2]{\resultat}{\c}` expands `\c` 2 times: this gives the control sequence `\a` which is converted into "$\backslash_{12}\ \texttt{a}_{12}$".

This example show all the "depths" of expansion, from 3 to 0:

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

1 z 3
\a
\b
\c

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

3.3.4 Expansion of several control sequences

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. Let's see this with this example:

```
\def\a{LaTeX}
\def\b{is powerful}
\scancs*[1]{\result}{\a \b}
\result\par
\scancs*[2]{\result}{\a\space\b}
\result
```

LaTeXis powerful
LaTeXispowerful

First of all, a warning message has been sent to log: "if third argument or its expansion have braces or spaces, they will be removed when scanned! Use starred `\scancs*` macro with care". Let's see what it means...

In the first result, a space is missing between the words "LaTeX" and "is", though a space was present in the code between the 2 control sequences `\a` and `\b`. Indeed, TEX ignores spaces that follow control sequences. Consequently, `{\a \b}` is read as `{\a\b}`, whatever be the number of spaces in the code between `\a` and `\b`. To obtain a space between "LaTeX" and "is", we could have used the control sequence `\space` whose expansion is a space, and write for the third argument: `{\a\space\b}`. We could also have modified the defintion of `\a` with a space after the word "LaTeX" like this: `\def\a{LaTeX }`.

However, it is necessary to be carfull when expanding control sequences more than one time: if a control sequence is expanded n times and gives $\texttt{text}_{10,11,12}$, the next expansion gobbles spaces. The second result shows that the second expansion gobbled all the spaces and consequently, `\result` contains "LaTeXispowerful"!

Moreover, it's also the meaning of the warning message, if the n^{th} expansion of a control sequence contains braces, they will be gobbled, like spaces.

Finaly, when using `\scancs` a space is inserted after each control sequence. Indeed, `\detokenize` (an ε-TEX command) called by `\scancs` inserts a space after each control sequence. There is no way to avoid this.

3.3.5 Examples

In the following example, control sequences are expanded 2 times: `\d` gives `\b`, and `\b` gives `\textbf{a}\textit{b}`. Notice that a space is inserted after each control sequence.

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

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

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

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

\a A
1{2}A
12A

Finaly, here is an example where we take advantage of the space inserted after each sequence control to find the n^{th} control sequence in the expansion of a control sequence.

In the example above, we find the fourth control sequence in `\myCS` whose expansion is:

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

Obviously, we expect: `\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}}
\verbtocs{\myCS}|\a xy{3 2}\b7\c123 {m}\d{8}\e|
% here, \myCS contains text
\findcs{4}{\myCS}
\theCS\par
\def\myCS{\a xy{3 2}\b7\c123 {m}\d{8}\e}
% here, \myCS contains control sequences
\findcs{4}{\myCS}
\theCS
```

```
\d
\d
```

3.4 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[8], 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 }
```

3.5 Starred macros

Commands manipulating strings take catcodes of characters into account. To prevent any differences between catcode and then avoid unexpected results, all these macros have starred version. Starred macros convert their textual arguments into arguments with catcode 12 and 10 and call the non-starred macros with these modified arguments.

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

Here is an example:

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

false
true

In this example, the first argument expands into `{ab}` where, because of `\string`, both characters have a catcode 12. The second argument is `{ab}` where characters have their natural catcode 11. The strings are *not equal* because of unmatching catcodes. Therefore, the test is negative. It is positive only with the starred macro.

Here is an other ewample :

```
\scancs[0]\mytext{abc}
\StrPosition{\mytext}{b}\par
\StrPosition*{\mytext}{b}\par
```

1
2

The control sequence `\mytext` contains `{abc}` with catcodes 12 : indeed, `\scancs` returns strings with catcodes 12 and 10 for space. Logically, the non-starred macro return 0 which means that it considers that the character "$\texttt{b}_{11}$" is not contained in the string "$\texttt{a}_{12}\texttt{b}_{12}\texttt{c}_{12}$". The starred macro behaves as expected and returns the correct position.

For the macros returning a string, if the starred version is used, the result will be a string with characters' catcode 12 and 10 fot space. For example, after "`\StrBefore*{a b c d}{c}[\mytext]`", the expansion of the control sequence `\mytext` will be "$\texttt{a}_{12\sqcup 10}\texttt{b}_{12\sqcup 10}$".

3.6 Examples

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

3.6.1 Example 1

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

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

We expect: **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 Example 2

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

```
\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 Example 3

In a control sequence `\text`, let's try to write in bold the first word that follows the word "new". In this example, `\text` contains:

```
Try the new package xstring !
```

```
\def\text{Try the new package xstring !}
\def\word{new}
\StrBehind[1]{\text}{\word}[\name]
\IfBeginWith{\name}{ }%
  {\StrGobbleLeft{\name}{1}[\name]}%
  {}%
\StrBefore{\name}{ }[\name]
\verbtocs{\before}|\textbf{|
\verbtocs{\after}|}|
\StrSubstitute[1]%
  {\text}{\name}{\before\name\after}[\text]
\tokenize{\text}{\text}
\text
```

Try the new **package** xstring !

3.6.4 Example 4

A control sequence `\myCS` défined with an `\edef` contains control sequences with their possible arguments. How to reverse the order of the 2 first control sequences? In this example, `\myCS` contains:

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

We expect a final result containing `\textit{B}\textbf{A}\texttt{C}` and displaying $B\mathbf{A}\texttt{C}$

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

result: $B\mathbf{A}\texttt{C}$

3.6.5 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