xstring

v1.86

User's manual

Christian Tellechea
unbonpetit@netc.fr

22 August 2023

Abstract

This package provides non expandable macros for manipulating code made of tokens. For a basic use, tokens can be alphanumeric chars, but the macros can also be useful for manipulating tokens, i.e. TEX code.

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] Any email to tell me errors would be appreciated!

1 Presentation

This extension[2] provides non expandable macros and tests operating on "strings of tokens", 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.

A "string of tokens" is a list of tokens of any nature, except that braces must be balanced and tokens catcode 6 and 14 (usualy % et #) are not allowed. Apart from this, any token is allowed (including `\par`), in any order in the list, whatever be the resulting code.

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 13), the command `\verbtocs` (page 17) and the command `\scancs` (page 18).

As the arguments may contain any token, advanced users could have problems with catcodes leading to unexpected behaviours. These behaviours can be controlled: read page 14.

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.

2 The macros

For a better understanding, let's see first the macros with the simpler arguments possible. No special catcode, no exotic token, no control sequence neither: only alphanumeric chars will be contained in the arguments.

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

- the syntax[4] 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[5] 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 "||".

Important: in the following, a ⟨*number*⟩ can be an integer written with numeric chars, a counter, or the result of an arithmetic operation made with the command `\numexpr`.

2.1 The tests

2.1.1 `\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*⟩.

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

```
true
false
true
false
true
true
false
```

[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 13.
[4] The optional star, the optional argument in last position will be explained later. See page 14 for starred macros and page 13 for the optional argument.
[5] For much more examples, see the test file.

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

```
true
false
false
true
false
true
false
false
true
```

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

```
true
false
false
true
false
false
true
false
false
```

2.1.4 \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}\par
\IfBeginWith{LaTeX}{a}{true}{false}\par
\IfBeginWith{a bc def }{a b}{true}{false}\par
\IfBeginWith{a bc def }{ab}{true}{false}
```

```
true
false
true
false
```

2.1.5 \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}\par
\IfEndWith{LaTeX}{a}{true}{false}\par
\IfEndWith{a bc def }{ef }{true}{false}\par
\IfEndWith{a bc def }{ef}{true}{false}
```

```
true
false
true
false
```

2.1.6 \IfInteger

`\IfInteger{`⟨*number*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Tests if ⟨*number*⟩ is an integer (i.e whose decimal part is empty or 0), and runs ⟨*true*⟩ if so, and ⟨*false*⟩ otherwise.

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

```
\IfInteger{13}{true}{false}\par
\IfInteger{-219}{true}{false}\par
\IfInteger{+9}{true}{false}\par
\IfInteger{3.14}{true}{false}\par
\IfInteger{8.0}{true}{false}\par
\IfInteger{0}{true}{false}\par
\IfInteger{49a}{true}{false}\par
\IfInteger{+}{true}{false}\par
\IfInteger{-}{true}{false}\par
\IfInteger{0000}{true}{false}
```

```
true
true
true
false
true
true
false
false
false
true
```

2.1.7 \IfDecimal

`\IfDecimal{`⟨*number*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

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

Macros `\integerpart` and `\decimalpart` contain the integer part and decimal part of ⟨*number*⟩.

If test is false because unexpected characters, the control sequence `\afterdecimal` contain the illegal part of ⟨*number*⟩, whereas if test is false because decimal part is empty after decimal separator, it contain "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}\par
\IfDecimal{3,14}{true}{false}\par
\IfDecimal{-0.5}{true}{false}\par
\IfDecimal{.7}{true}{false}\par
\IfDecimal{,9}{true}{false}\par
\IfDecimal{1..2}{true}{false}\par
\IfDecimal{+6}{true}{false}\par
\IfDecimal{-15}{true}{false}\par
\IfDecimal{1.}{true}{false}\par
\IfDecimal{2,}{true}{false}\par
\IfDecimal{.}{true}{false}\par
\IfDecimal{,}{true}{false}\par
\IfDecimal{+}{true}{false}\par
\IfDecimal{-}{true}{false}
```

```
true
true
true
true
true
false
true
true
false
false
false
false
false
false
```

2.1.8 \IfStrEq

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

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

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

```
true
false
false
true
false
false
false
true
```

2.1.9 `\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.

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

```
true
false
false
true
true
true
false
true
false
false
true
true
true
```

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

```
BB
||
CC
other
other
other
```

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

```
BB
||
CC
other
trois
demi
```

2.2 Extraction of substrings

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

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

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

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

2.2.3 `\StrCut`

Here is the syntax of this macro:

`\StrCut`⟨`[*]`⟩`[`⟨*nunber*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*macroA*⟩`}{`⟨*macroB*⟩`}`

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

The ⟨*string*⟩ is cut in two parts at the occurrence `[`⟨*number*⟩`]` of `{`⟨*stringA*⟩`}`. The left part is stored in the control sequence ⟨*macroA*⟩ and the right part in ⟨*macroB*⟩.
Since this macro returns *two strings*, it does *not* display anything. Consequently, it does not provide the optional argument in last position.

- ▷ If ⟨*string*⟩ or ⟨*stringA*⟩ is empty, ⟨*macroA*⟩ and ⟨*macroB*⟩ are empty;
- ▷ If ⟨*number*⟩ < 1, the macro behaves as if ⟨*number*⟩ $= 1$;
- ▷ If the occurrence is not found, ⟨*macroA*⟩ receives the whole ⟨*string*⟩ while ⟨*macroB*⟩ is empty.

```
\def\seprouge{{\color{red}|}}
\StrCut{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut[2]{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut[3]{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut[4]{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut[5]{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut[6]{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut[-4]{abracadabra}{a}\csA\csB |\csA\seprouge\csB|\par
\StrCut{abracadabra}{brac}\csA\csB |\csA\seprouge\csB|\par
\StrCut{abracadabra}{foo}\csA\csB |\csA\seprouge\csB|\par
\StrCut{abracadabra}{}\csA\csB |\csA\seprouge\csB|
```

	bracadabra
abr	cadabra
abrac	dabra
abracad	bra
abracadabr	
abracadabra	
	bracadabra
a	adabra
abracadabra	

2.2.4 \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[6] 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}\par
|\StrBetween{xstring}{i}{n}|\par
|\StrBetween{xstring}{a}{tring}|\par
|\StrBetween{a bc def }{a}{d}|\par
|\StrBetween{a bc def }{a }{f}|\par
\StrBetween{a1b1a2b2a3b3}{a}{b}\par
\StrBetween[2,3]{a1b1a2b2a3b3}{a}{b}\par
\StrBetween[1,3]{a1b1a2b2a3b3}{a}{b}\par
|\StrBetween[3,1]{a1b1a2b2a3b3}{a}{b}|\par
\StrBetween[3,2]{abracadabra}{a}{bra}
```

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

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

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

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

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

brcdbr
bracadabra
brcdbra
brcdbr
abcdef
|a bc |

2.2.7 \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*⟩ $\leqslant 0$, no character is deleted;
- ▷ If ⟨*number*⟩ $\geqslant$ ⟨*lengthString*⟩, all the characters are deleted.

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

tring
||
X
LaTeX
| def |

2.2.8 \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*⟩ $\leqslant 0$, no character is returned;
- ▷ If ⟨*number*⟩ $\geqslant$ ⟨*lengthString*⟩, all the characters are returned.

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

xs
xstring
LaTe
||
|a bc |

2.2.9 \StrGobbleRight

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

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

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

xstri
||
L
|LaTeX|
|a bc |

2.2.10 `\StrRight`

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

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

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

ng
xstring
aTeX
||
def

2.2.11 `\StrChar`

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

Returns the syntax unit at the position ⟨*number*⟩ in ⟨*string*⟩.

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

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

r
||
||
d

2.2.12 `\StrMid`

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

In ⟨*string*⟩, returns the substring between[7] 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}\par
\StrMid{xstring}{-4}{2}\par
|\StrMid{xstring}{5}{1}|\par
\StrMid{xstring}{6}{15}\par
\StrMid{xstring}{3}{3}\par
|\StrMid{a bc def }{2}{7}|
```

stri
xs
||
ng
t
| bc de|

2.3 Macros returning a number

2.3.1 `\StrLen`

`\StrLen{`⟨*string*⟩`}[`⟨*name*⟩`]`

Return the length of ⟨*string*⟩.

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

7
1
9

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

2.3.2 \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}\par
\StrCount{abracadabra}{bra}\par
\StrCount{abracadabra}{tic}\par
\StrCount{aaaaaa}{aa}
```

```
5
2
0
3
```

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

```
4
8
9
0
0
6
5
```

2.3.4 \StrCompare

\StrCompare⟨[*]⟩{⟨*stringA*⟩}{⟨*stringB*⟩}[⟨*name*⟩]

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

- The normal tolerance is 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 is 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 \savecomparemode, then modify this comparison mode and come back to the situation when it was saved with \restorecomparemode.

Examples with the normal tolerance:

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

```
0
0
0
2
1
1
6
0
```

Examples with the strict tolerance:

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

```
0
4
4
2
1
1
6
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 works on them. 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 commands \noexpandarg or \normalexpandarg.

Another 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 (if you want the expansion of all tokens one time, you should call the macro \StrExpand, see page 18).

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

In the following list, for every macro of the previous chapter, the arguments colored in purple will possibly be expanded, according to the expansion mode:

- \IfSubStr⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfSubStrBefore⟨[*]⟩[⟨*number1*⟩,⟨*number2*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfSubStrBehind⟨[*]⟩[⟨*number1*⟩,⟨*number2*⟩]{⟨*string*⟩}{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfBeginWith⟨[*]⟩{⟨*string*⟩}{⟨*stringA*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfEndWith⟨[*]⟩{⟨*string*⟩}{⟨*stringA*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfInteger{⟨*number*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfDecimal{⟨*number*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfStrEq⟨[*]⟩{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfEq⟨[*]⟩{⟨*stringA*⟩}{⟨*stringB*⟩}{⟨*true*⟩}{⟨*false*⟩}
- \IfStrEqCase⟨[*]⟩{⟨*string*⟩}{{⟨*string1*⟩}{⟨*code1*⟩}
  {⟨*string2*⟩}{⟨*code2*⟩}
  ...
  {⟨*string n*⟩}{⟨*code n*⟩}}[⟨*other cases code*⟩]
- \IfEqCase⟨[*]⟩{⟨*string*⟩}{{⟨*string1*⟩}{⟨*code1*⟩}
  {⟨*string2*⟩}{⟨*code2*⟩}
  ...
  {⟨*string n*⟩}{⟨*code n*⟩}}[⟨*other cases code*⟩]
- \StrBefore⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]
- \StrBehind⟨[*]⟩[⟨*number*⟩]{⟨*string*⟩}{⟨*stringA*⟩}[⟨*name*⟩]

- `\StrBetween`⟨`[*]`⟩`[`⟨*number1*⟩`,`⟨*number2*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}[`⟨*name*⟩`]`
- `\StrSubstitute[`⟨*number*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}[`⟨*name*⟩`]`
- `\StrDel[`⟨*number*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}[`⟨*name*⟩`]`
- `\StrSplit{`⟨*string*⟩`}{`⟨*number*⟩`}{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}` (see macro `StrSplit` page 16)
- `\StrGobbleLeft{`⟨*string*⟩`}{`⟨*number*⟩`}[`⟨*name*⟩`]`
- `\StrLeft{`⟨*string*⟩`}{`⟨*number*⟩`}[`⟨*name*⟩`]`
- `\StrGobbleRight{`⟨*string*⟩`}{`⟨*number*⟩`}[`⟨*name*⟩`]`
- `\StrRight{`⟨*string*⟩`}{`⟨*number*⟩`}[`⟨*name*⟩`]`
- `\StrChar{`⟨*string*⟩`}{`⟨*number*⟩`}[`⟨*name*⟩`]`
- `\StrMid{`⟨*string*⟩`}{`⟨*number1*⟩`}{`⟨*number2*⟩`}[`⟨*name*⟩`]`
- `\StrLen{`⟨*string*⟩`}[`⟨*name*⟩`]`
- `\StrCount{`⟨*string*⟩`}{`⟨*stringA*⟩`}[`⟨*name*⟩`]`
- `\StrPosition[`⟨*number*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}[`⟨*name*⟩`]`
- `\StrCompare{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}[`⟨*name*⟩`]`

3.1.2 Chars and tokens 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**[8].

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[9] 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 : `{ } $ &`[10]
- tokens with catcode 7 and 8, usually : `^ _`
- any purely expandable control sequence[11] or tokens with catcode 13 (active chars) whose expansion is allowed chars.

When arguments are not expanded with the use of `\noexpandarg`, other tokens can be put in a string whatever be the code they make: any control sequence, even undefined, any token catcode 13. Moreover, test tokens are allowed like `\if` or `\ifx`, even without their `\fi`. On the same way, a `\csname` without its `\endcsname` is allowed.
In this example, the argument contains a `\ifx` *without* the `\fi`: the `\StrBetween` command extracts and dislpays what is between `\ifx` and `\else`:

```
1 \noexpandarg
2 \StrBetween{\ifx ab false \else true}{\ifx}{\else}
```

ab false

When `\expandarg` is used, the first token needs precaution since it is expanded one time: it must be defined. The other tokens are left unchanged like with `\noexpandarg`.

[8] Maybe, the token # will be allowed in a future version.
[9] For a reliable operation with accented letters, the `\fontenc` package with option `[T1]` and `\inputenc` with appropriated option must be loaded
[10] Warning : braces **must** be balanced in arguments !
[11] i.e. this control sequence can be `\edefed`.

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.

For this reason, all the macros returning a result (i.e. all excepted the tests) 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 :

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

And this nested structure, not allowed neither, supposed to remove the first and last char of `xstring`:

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

should be written like this:

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

3.3 How does xstring read 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[12] is either:

- a control sequence;
- a group, i.e. what is between 2 balanced braces (usually tokens catcode 1 and 2);
- 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 if, while `\noexpandarg` is active, we ask xstring to find the length of this argument and find its 4th "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` to display it, 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 argument containing the string of tokens, xstring does not look into groups, and simply consider them as a syntax unit.
For specific uses, it can be necessary to look into groups: `\exploregroups` changes the exploration mode and forces the macros to look inside groups.

What does this exploration mode in the previous example? xstring does not count the group as a single syntax unit but looks 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
```

[12] For advanced users used to LATEX programming, a syntax unit is what is gobbled by the macro `\@gobble` whose code is: `\def\@gobble#1{}`

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 a string 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[13] 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`.
Moreover, 2 macros run in `\noexploregroups` mode, whatever be the current mode: `\StrBetween` et `\StrMid`.

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

3.4 Catcode and starred macros

Macros of this package take the catcodes of tokens into account. To avoid unexpected behaviour (particulary with tests), you should keep in mind that tokens *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 chars catcodes 12 and 10 : `\detokenize`, `\meaning`, `\jobname`, `\fontname`, `\romannumeral`, etc.

Starred macros do not take catcodes into account. They simply convert some arguments into arguments with catcodes 10, 11 and 12, and call the non-starred macros with these modified arguments. The optional arguments are not modified and the catcodes are left unchanged.

Here is an example:

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

false
true

The strings do not match because of catcode differences: the test is negative in the non-starred macro.

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.

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 "$\texttt{a}_{12\sqcup 10}\texttt{\textbackslash}_{12}\texttt{b}_{12\sqcup 10}$".

The macro with a starred version are listed below. For these macros, if starred version is used, the purple arguments will be detokenized:

- `\IfSubStr`⟨`[*]`⟩`[`⟨*number*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`
- `\IfSubStrBefore`⟨`[*]`⟩`[`⟨*number1*⟩,⟨*number2*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`
- `\IfSubStrBehind`⟨`[*]`⟩`[`⟨*number1*⟩,⟨*number2*⟩`]{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`
- `\IfBeginWith`⟨`[*]`⟩`{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`
- `\IfEndWith`⟨`[*]`⟩`{`⟨*string*⟩`}{`⟨*stringA*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

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

- `\IfStrEq⟨[*]⟩{⟨stringA⟩}{⟨stringB⟩}{⟨true⟩}{⟨false⟩}`
- `\IfEq⟨[*]⟩{⟨stringA⟩}{⟨stringB⟩}{⟨true⟩}{⟨false⟩}`
- `\IfStrEqCase⟨[*]⟩{⟨string⟩}{{⟨string1⟩}{⟨code1⟩}`
  `{⟨string2⟩}{⟨code2⟩}`
  `...`
  `{⟨string n⟩}{⟨code n⟩}}[⟨other cases code⟩]`
- `\IfEqCase⟨[*]⟩{⟨string⟩}{{⟨string1⟩}{⟨code1⟩}`
  `{⟨string2⟩}{⟨code2⟩}`
  `...`
  `{⟨string n⟩}{⟨code n⟩}}[⟨other cases code⟩]`
- `\StrBefore⟨[*]⟩[⟨number⟩]{⟨string⟩}{⟨stringA⟩}[⟨name⟩]`
- `\StrBehind⟨[*]⟩[⟨number⟩]{⟨string⟩}{⟨stringA⟩}[⟨name⟩]`
- `\StrBetween⟨[*]⟩[⟨number1⟩,⟨number2⟩]{⟨string⟩}{⟨stringA⟩}{⟨stringB⟩}[⟨name⟩]`
- `\StrCompare⟨[*]⟩{⟨stringA⟩}{⟨stringB⟩}[⟨name⟩]`

4 Advanced macros for 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.

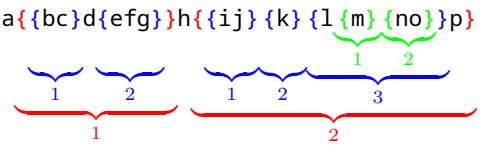
4.1 Finding a group, macros `\StrFindGroup` and `\groupID`

When `\exploregroups` mode is active, the macro `\StrFindGroup` finds a group between braces with its identifier:
`\StrFindGroup{⟨argument⟩}{⟨identifier⟩}[⟨name⟩]`

When the group matching the identifier does not exist, an empty string is assigned to ⟨*name*⟩. If the group is found, this group *with its braces* is assigned to ⟨*name*⟩.

This identifier characterizes the nestling position of the group. It is a list of one or several integers separated with commas. n_1, the first integer is the number of the group (not nestled in another) in which the sought group is. Inside this group, the second integer n_2 is the number of the group (not nestled in another) in which the sought group is...and so on until the necessary nestling depth is reached to obtain the sougth after group.

Let's take an example with 3 levels of nestled groups. In this example, braces delimiting groups are colored in red for nestling level 1, in blue for level 2 and in green for level 3. The groups are numbered with the rule seen above:



In this example:

- the group `{{bc}d{efg}}` has the identifier: `1`
- the group `{ij}` has the identifier: `2,1`
- the group `{no}` has the identifier: `2,3,2`
- the whole argument `a{{bc}d{efg}}h{{ij}{k}{l{m}{no}}p}` has the identifier `0`, only case where the integer 0 is appears in the identifier of a group.

Here is the full example:

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

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

The reverse process exists, and several macros of xstring provide the identifier of the group in which they made a cut or they found a substring. These macros are: `\IfSubStr`, `\StrBefore`, `\StrBehind`, `\StrSplit`, `\StrLeft`, `\StrGobbleLeft`, `\StrRight`, `\StrGobbleRight`, `\StrChar`, `\StrPosition`.

After these macros, the control sequence `\groupID` expands to the identifier of the group where the cut has been done or the search has succeeded. When not cut can be done or the search fails, `\groupID` is empty. Obvioulsly, the use of `\groupID` has sense only when `\exploregroups` mode is acive and when non starred macros are used.

Here are some examples with the macro `\StrChar`:

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

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

4.2 Splitting a string, the macro `\StrSplit`

Here is the syntax:

`\StrSplit{`⟨*string*⟩`}{`⟨*number*⟩`}{`⟨*stringA*⟩`}{`⟨*stringB*⟩`}`

The ⟨*string*⟩, is splitted after the syntax unit at position ⟨*number*⟩. The left part is assigned to the control sequence ⟨*stringA*⟩ and the right part is assigned to ⟨*stringB*⟩.
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, ⟨*stringA*⟩ is empty and ⟨*stringB*⟩ is equal to ⟨*string*⟩;
- ▷ If ⟨*number*⟩ ⩾⟨*lengthString*⟩, ⟨*stringA*⟩ is equal to ⟨*string*⟩ and ⟨*stringB*⟩ is empty;
- ▷ If ⟨*string*⟩ is empty, ⟨*stringA*⟩ and ⟨*stringB*⟩ are empty, whatever be the integer ⟨*number*⟩.

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

abcd	ef
a	b c
a	bcdef
abcde	f
abcdef	
	abcdef

When the exploration of groups is active and the cut is made at the end of a group, the content of the left string will tbe the entire group while the right string will be empty. The example shows this:

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

macro:->ef
macro:->

This macro provides a star version: in this case, the cut is made just before the syntax unit which follows the syntax unit at position ⟨*number*⟩. Both version give same results, except when the cut is made at the end of a group; in that case, `\StrSplit` closes as many group as necessary until it finds the next syntax unit: the cut is made just before this syntax unit.

```
\exploregroups
Use without star :\par
\StrSplit{ab{cd{ef}gh}ij}{6}\strA\strB
\meaning\strA\par
\meaning\strB\par
\string\groupID\ = \groupID\par\medskip
Use with star :\par
\StrSplit*{ab{cd{ef}gh}ij}{6}\strA\strB
\meaning\strA\par
\meaning\strB\par
\string\groupID\ = \groupID
```

Use without star :
macro:->ef
macro:->
\groupID = 1,1

Use with star :
macro:->cd{ef}
macro:->gh
\groupID = 1,1

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

The macro `\verbtocs` allow to read the content of a "verb" argument containing special characters: &, ~, \, {, }, _, #, $, ^ and %. 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 tokens with catcodes 12 (or 10 for space).

By default, the token delimiting the verb content is "|". Obviously, this token 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 token delimiting the verb content with the macro:

`\setverbdelim{`⟨*character*⟩`}`

Any ⟨*token*⟩ can be used[14]. 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{\resultat}  |a   & b{ c% d$ e \f|
here is the result:\par\resultat
```

here is the result:
a & b{ c% d$ e \f

4.4 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{\resultat}{\text}\par
result : \resultat
```

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.

[14] Several tokens 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 token.

4.5 Expansion of a control sequence before verbatimize, the macros `\StrExpand` and `\scancs`

The macro `\StrExpand` expands the tokens of a string:

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

By default, ⟨*number*⟩ is 1 and represnts the number of times each token of ⟨*string*⟩ has to be expanded. The ⟨*name*⟩ is a control sequence receiving the result of the expansions.

Macro works sequentially and by pass: in a pass, each token from left to right is replaced by its 1-expansion. After this expansion, if the ⟨*number*⟩ of expansions is not reached, an other pass starts, and so on.

Here is an example:

```
\def\csA{1 2}
\def\csB{a \csA}
\def\csC{\csB\space}
\def\csD{x{\csA y}\csB{\csC z}}
Expansion of \string\csD\ au\par
\StrExpand[0]{\csD}{\csE} level 0 :
\detokenize\expandafter{\csE}\par
\StrExpand[1]{\csD}{\csE} level 1 :
\detokenize\expandafter{\csE}\par
\StrExpand[2]{\csD}{\csE} level 2 :
\detokenize\expandafter{\csE}\par
\StrExpand[3]{\csD}{\csE} level 3 :
\detokenize\expandafter{\csE}\par
\StrExpand[4]{\csD}{\csE} level 4 :
\detokenize\expandafter{\csE}
```

Expansion of \csD au
level 0 : \csD
level 1 : x{\csA y}\csB {\csC z}
level 2 : x{1 2y}a \csA {\csB \space z}
level 3 : x{1 2y}a 1 2{a \csA z}
level 4 : x{1 2y}a 1 2{a 1 2 z}

The macro expands each token consecutively, and does not see what follows: tokens whose expansion depends on other tokens cannot be expanded with this macro. For instance, though "`\iftrue A\else B\fi`" has a natural expansion ("A"), it cannot be put in the argument of `\StrExpand` and:

```
\StrExpand{\iftrue A\else B\fi}\result
```

provokes an error because the first token "`\iftrue`" is expanded *alone* without seeing its `\fi` which makes TEX angry. Expansion inside groups is *independant* of the exploration mode: this macro has its own command to expand or not what is inside the groups. By default, tokens inside groups are expanded, but this can be changed with the macro `\noexpandingroups`. The default behaviour can be recoverd with `\expandingroups`.

The `\scancs` macro returns the detokenized result:

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

The ⟨*number*⟩ is 1 by default and represents the number of expansions.

`\scancs` has been kept for compatibility with older versions of xstring. This macro now unnecessry, simply takes the result of `\StrExpand` and `\detokenize` it.

4.6 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 `text`$_{10,11,12}$) into control sequences.

In the following teaching example[15], 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 }
```

4.7 The macro `\StrRemoveBraces`

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

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

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

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

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

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

⋆
⋆ ⋆

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

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