The package for TEX and LATEX

tokstools

v 0.1

2026/03/29

Christian Tellechea
`unbonpetit@netc.fr`

This package for TEX or LATEX provides tools for manipulating tokens. Within a set of tokens, it is possible to define matching rules using PEG-type grammars. It then becomes possible to test whether a match exists, to count matches, to perform replacements, to capture tokens, and more.

Contents

1 Presentation **2**

1.1 Tokens, charcode, catcode, engines . . . 2
1.2 Notations . . . 3
1.3 The macro `\printtoks` . . . 3

2 Act on each token with `\toksdo` **3**

2.1 Elementary patterns . . . 3
2.2 The pattern `\r` . . . 4
2.3 The pattern `\R` . . . 4
2.4 The pattern `\S` . . . 4
2.5 Patterns . . . 5
2.6 Using `\toksdo` . . . 5
2.7 Using `\toksdo` and `\addtok` . . . 7

3 Counting tokens with `\tokscount` **8**

4 PEG Grammars **8**

4.1 The 5 Pattern Markers . . . 8
4.2 Repetitions . . . 9
4.3 Predicates . . . 9
4.4 Pattern concatenation . . . 9
4.5 Choosing between patterns . . . 10
4.6 Pattern grouping . . . 10
4.7 Spaces . . . 10
4.8 Precedences . . . 10
4.9 Pattern names . . . 10

5 Testing for a match with `\ifpegmatch` **11**

5.1 Syntax . . . 11
5.2 Mode . . . 12
5.3 Token Output . . . 12
5.4 Captures . . . 12
5.5 Recursive Grammars . . . 13

6 Counting matches with `\pegcount` **14**

7 Replacements with `\pegreplace` **14**

8 Errors **15**

8.1 Unbalanced Braces . . . 15
8.2 Engine not suitable for encoding . . . 16
8.3 Invalid Interval . . . 16
8.4 Invalid Pattern Syntax . . . 16
8.5 Capture Errors . . . 16

9 List of macros and markers **16**

9.1 List of commands . . . 16
9.2 Liste of markers . . . 17

1 Presentation

The `tokstools` extension works with any engine. It is not limited to LaTeX[1], and to use it, you need to write

- `\input tokstools.tex` with (pdf)(Xe)(lua)(op)TeX;
- `\usepackage{tokstools}` with (pdf)(Xe)(lua)LaTeX.

This package only requires simplekv, which will be loaded if it hasn't been.

Important notice: this English manual is provided so that as many people as possible can access the necessary instructions for using the `tokstools` package. It is very important for readers of this manual to understand that it is merely a translation generated by various AI-based tools available online. The quality and accuracy of the translation are therefore not guaranteed, especially since the resulting translation has not been reviewed by a human. Consequently, **for reliable information, please refer to the French manual**.

Throughout this manual, the term "marker" refers to a macro that serves as a reference point to be recognized within an argument, but which is not modified by `tokstools`.

1.1 Tokens, charcode, catcode, engines

The `tokstools` package operates on sets of tokens (or token strings). Tokens are the smallest unit of source code manipulated by TeX.

Users should be aware that the notion of tokens depends on the type of engine used. It is only for 8 bit engines, such as pdfLaTeX for example, that 1 token = 1 byte. As an example, the character "€" is seen

- by pdfLaTeX as 3 tokens whose charcodes are 226; 130 and 172 and catcodes are all equal to 13;
- by luaLaTeX as 1 single token with charcode 8364 and catcode 12.

This difference must be taken into account, and can be the source of more or less understandable errors when manipulating UTF8 chars on an 8 bit engine.

This manual was compiled with luaLaTeX, so tokens with charcodes greater than 255 may exist.

Whatever macros are provided by `tokstools`, they always decompose their token argument, often called ⟨*tokens*⟩, into a list of tokens, each with a charcode and a catcode. The catcodes visible to `tokstools` are listed in the table below, along with the most common tokens with charcodes less than 255 that have these catcodes by default.

Catcode	Description	Tokens
1	beginning of a group	`{`
2	end of a group	`}`
3	math shift	`$`
4	alignment tab	`&`
6	parameter character	`#`
7	superscript	`^`
8	subscript	`_`
10	space	␣ and *HT*
11	letters	`a-z` and `A-Z`
12	others	digits (`0-9`) punctuation signs (`: . ; , ? ! ' "`) math signs (`+ - * / = < >`) other signs (`[ ] ( ) | @ ‘`)
13	active character	`~`
16	control sequence	`\`⟨*letters*⟩ ou `\`⟨*char*⟩

None of the macros provided by `tokstools` are expandable. If you wish the results of these macros, or even captures, to be contained in macros that are purely expandable, you must use the assignment instructions defined in this manual.

[1]As the `pegmatch` package unfortunately is

1.2 Notations

Throughout this manual:

- ⟨*tokens*⟩ denotes a set of tokens[2] balanced in tokens of catcode 1 and 2 (opening and closing braces) that serves as an argument to macros or markers in charge of processing these tokens;
- ⟨*1-pattern*⟩ represents a single pattern, i.e., consisting of a marker with its possible argument, preceded by an optional predicate and followed by an optional repetition specification;
- ⟨*patterns*⟩ represents a single ⟨*1-pattern*⟩ or a combination of several ⟨*1-pattern*⟩ obtained using the concatenation operator « `:` » and the ordered choice operator « `|` » (only the " `|` " operator for `\toksdo` and `\tokscount`);
- `{`⟨*patterns*⟩`}` is a grouping that behaves like a ⟨*1-pattern*⟩.

1.3 The macro `\printtoks`

The macro

`\printtoks[`⟨*keys*⟩`=`⟨*values*⟩`]{`⟨*tokens*⟩`}`

displays tokens with their charcode and catcode below. If a token is a control sequence or a control character, no charcode is displayed.

```
\printtoks[code=\ttfamily]{\def\foo#1#2#3{\hbox to 1cm{\hss$#1^#2_#3$\hss}}}
\medbreak
\printtoks[code=\ttfamily]{ė€eé}
```

```
|\def|\foo|# |1 |# |2 |# |3 | { |\hbox| t | o |  |1 |c | m | { |\hss|$ |# |1 |^ |# |2 |_ |# |3 |$ |\hss| } | } |
|    |    |35|49|35|50|35|51|123|     |116|111|32|49|99|109|123|    |36|35|49|94|35|50|95|35|51|36|    |125|125|
| 16 | 16 | 6|12| 6|12| 6|12| 1 | 16  | 11| 11|10|12|11| 11| 1 | 16 | 3| 6|12| 7| 6|12| 8| 6|12| 3| 16 | 2 | 2 |

| e | é | ė | €  |
|101|233|279|8364|
| 11| 11| 11| 12 |
```

The ⟨*keys*⟩ available are :

Key	Default value	Description
`expand arg`	`0`	number of expansions for the argument containing the ⟨*tokens*⟩ before being taken into account
`code`	⟨*empty*⟩	code executed before displaying a token
`intertoks`	`0.33em`	horizontal space between each token, inserted by the primitive `\hskip`
`printcharcode`	`true`	display or not the charcode
`printcatcode`	`true`	display or not the catcode
`hexcharcode`	`false`	display the charcode in base 16 if `true` and in base 10 otherwise
`baselinecoeff`	`0.8`	vertical spacing coefficient
`vlines`	`true`	display or not the vertical lines between each token
`boxed`	`true`	put everything in a `\hbox` if `true`, which makes everything unbreakable

2 Act on each token with `\toksdo`

This macro scans all tokens one by one, independently of each other. The user can specify criteria using ⟨*patterns*⟩ and for each, an ⟨*action*⟩ to be performed if the token matches ⟨*patterns*⟩.

2.1 Elementary patterns

3 elementary patterns likely to match a token are available, each accessible via the marker `\r`, `\R` and `\S`.

[2]The primitive `\par` may be found in ⟨*tokens*⟩.

2.2 The pattern `\r`

The syntax of this ⟨*1-pattern*⟩ is

`\r{`⟨*csv of token intervals*⟩`:` ⟨*csv of catcode intervals*⟩`}`

where an interval is of the form "range extending between two tokens" of type ⟨*token1*⟩-⟨*token2*⟩, but can be reduced to a single token ⟨*token*⟩.

In the ⟨*csv list of catcode intervals*⟩ , the "*" character acts as a wildcard and replaces any interval. If the ⟨*csv of catcode intervals*⟩ is absent, it is understood as "*".

Commas, hyphens and "`:`" cannot be specified as tokens, as they are part of the syntax (see the R| pattern to get around this limitation).

For example:

`\r{a-z:11}`	matches with a token from a to z with an 11 catcode.
`\r{a-z}` ou `\r{a-z:*}`	matches with a token from a to z with any catcode.
`\r{a,e,i,o,u,y}`	matches with any voyel with any catcode.
`\r{a-z,A-Z,0-9:11,12}`	matches any alphanumeric character (lowercase, uppercase or numeric), with either an 11 or 12 catcode.

2.3 The pattern `\R`

The syntax of this ⟨*1-pattern*⟩ is

`\r{`⟨*csv of charcode intervals*⟩`:`⟨*csv of catcode intervals*⟩`}`

In both csv lists, the "*" character is a wildcard and replaces any interval. The charcodes are numbers written in digits (base 10) or in the manner of TEX, where "`` `\: ``", "`` `\, ``" and "`` `\- ``" are the integers corresponding respectively to the charcodes of the "`:`", "`,`" and "`-`" tokens.

For example:

`\R{*:10}`	matches any catcode token 10 (which is a space for TEX)
`\R{*:16}`	matches any catcode token 16 (which is a character or control sequence for TEX)
`\R{106-115: 11}`	matches with any token from "j" to "s" having a catcode of 11
`` \R{`\a-`\z} ``	is identical to `\R{97-122}` which is equivalent to `\r{a-z}`
`` \R{`\,,`\;,`\: :12} ``	is identical to `\R{44,59,58:12}` and matches `,` or `;` or `:` with catcode 12
`\R{*:*}`	matches any token

2.4 The pattern `\S`

The syntax of this ⟨*1-pattern*⟩ is

`\S{`⟨*tokens*⟩`}`

A token matches if it appears in ⟨*tokens*⟩ passed as an argument to `\S`.

In the tokens "`ab \foo{0 123}c d`", those that match the pattern "`\S{1{a }\foo}`" are boxed in red:

$\boxed{\texttt{a}}\ \texttt{b}\ \boxed{␣}\ \boxed{\texttt{\textbackslash foo}}\ \boxed{\texttt{\{}}\ \texttt{0}\ \boxed{␣}\ \boxed{\texttt{1}}\ \texttt{2}\ \texttt{3}\ \boxed{\texttt{\}}}\ \texttt{c}\ \boxed{␣}\ \texttt{d}$

The ⟨*tokens*⟩ may contain one or more nonnested occurrences of the marker `\c`. The syntax of this marker is

`\c{`⟨*catcode*⟩`}{`⟨*tokens*⟩`}`

In the argument of `\S`, the action of `\c` is to assign the ⟨*catcode*⟩ passed in the first argument to the ⟨*tokens*⟩. If the ⟨*catcode*⟩ is reduced to a single digit, the curly braces are optional. The same applies to the second argument if it contains a single token.

If a token is a ⟨*macro*⟩, the only catcode it can receive is 12, in which case catcode 12 tokens from `\string`⟨*macro*⟩ will be created.

The catcodes accepted in the first argument of `\c` are: 1, 2, 3, 4, 6, 7, 8, 10, 11, 12 and 13. The user should be extremely careful about the consequences of changing catcodes to the sensitive categories numbered 1, 2 and 6.

If we write

```
\S{12\c{12}\bar3abc\c{11}{4 5.6}7\c78 9}
```

then the ⟨*set of tokens*⟩ contained in the \S argument is:

1	2	\	b	a	r	3	a	b	c	4		5	.	6	7	8		9
49	50	92	98	97	114	51	97	98	99	52	32	53	46	54	55	56	32	57
12	12	12	12	12	12	12	11	11	11	11	11	11	11	11	12	7	10	12

2.5 Patterns

The ⟨*patterns*⟩ listed for \toksdo consist of a single basic ⟨*1-pattern*⟩ or multiple patterns separated by « | », which means «or»:

`\r{a-z:11} | \R{*:12}`	matches any token that is a letter from a to z with catcode 11 or any token with catcode 12
`\R{*:16} | S{01223456789.}`	matches any token that is a macro, a digit, or a period
`\r{0-9:12} | \r{a-z,A-Z:11} | \R{*:10}`	matches any token with the default catcode being a digit, a letter, or a space

2.6 Using \toksdo

This macro is used as follows:

```
\toksdo[⟨keys⟩=⟨values⟩]
   {
    ⟨patterns1⟩ -> ⟨code1⟩,
    ⟨patterns2⟩ -> ⟨code2⟩,
    etc.
   }{⟨tokens⟩}
```

You can specify as many ⟨*patterns*⟩ and associated ⟨*codes*⟩ as you wish.

Each ⟨*code*⟩ is an arbitrary code[3] that can modify the ⟨*token*⟩ that matched ⟨*patterns*⟩. Within this ⟨*code*⟩, the following macros can be used:

- \tokslen, which is the total number of tokens;
- \selfindex, \selfcharcode, and \selfcatcode, which, for the matched token, represent its index (starting at 1 and ending at \tokslen), its charcode, and its catcode. If the token is a macro, \selfcharcode is *not* a number and is equal to the macro itself;
- \addtok{⟨*code*⟩} which specifies how to add the matched token to the internal collector that gathers them all for display or assignment at the end of the process;
- \deltok deletes the matched token;
- \setcharcode{⟨*arithmetic expression*⟩} and \setcatcode{⟨*arithmetic expression*⟩} change the charcode and catcode of the matched token. It is impossible to change the catcode of a macro, and for all other tokens, the catcodes accepted by \setcatcode are 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, and 13. Special care must be taken regarding what happens if a catcode is changed to 1, 2, or 6.

The ⟨*keys*⟩ accepted by the \toksdo macro are:

Key	Default value	Description
`expand arg`	`0`	number of expansions for the argument containing the ⟨*tokens*⟩ before being processed
`collect`	`true`	collects all selected tokens for the purpose of displaying or storing them. No collection is performed if this boolean is `false`
`assign`	⟨*empty*⟩	assignment instruction[4] of the result. If ⟨*empty*⟩, displays the result. If the value is a ⟨*macro*⟩, this ⟨*macro*⟩ must be a token register that will receive the result. The value can also be of the form \def⟨*macro*⟩, where the ⟨*macro*⟩ will receive the result

[3]If this ⟨*code*⟩ contains a comma that is not enclosed in curly braces, you must use the syntax ⟨*patterns*⟩->{⟨*code*⟩}.

[4]An assignment instruction is any code that may appear before the result enclosed in curly braces. The code that is executed is therefore ⟨*assignment*⟩{⟨*result*⟩}.

Here's how to program Rot13:

```
\toksdo{
        \r{a-m,A-M} -> \setcharcode{\selfcharcode + 13},
        \r{n-z,N-Z} -> \setcharcode{\selfcharcode - 13}
        }{Two plus two equals four}
```

Gjb cyhf gjb rdhnyf sbhe

In this example, any digit or hyphen is replaced with "x", and any space is replaced with "_" (catcode 12):

```
\toksdo{
        \r{0-9} | \S{-} -> \setcharcode{`x},
        \R{*:10}        -> \setcharcode{`\_}\setcatcode{12}
        }{Your password is: \textbf{758-457-384}, don't forget it!}
```

Your_password_is:_**xxxxxxxxxxx**,_don't_forget_it!

Keep only the first half of the tokens:

```
\toksdo{\R{*:*} -> \ifnum\selfindex>\numexpr\tokslen/2\relax \deltok \fi}{123abcd689}
```

123ab

It is sometimes necessary to assign the result to a token register in case the tokens obtained contain the catcode 6 token (# by default):

```
\newtoks\myresult
\toksdo[assign=\myresult]
        {\S{2} -> \deltok }% remove all "2"
        {\def\foo#1{02#120}}
\the\myresult% execution of \def\foo#1{0#10}
\foo{A}, \foo{xYz}
```

0A0, 0xYz0

If we had written "`assign = \def\mymacro`", the code that would have been executed behind the scenes is

```
\def\mymacro{\def\foo#1{0#10}}
```

This code is illegal in TeX (error of the type "`Illegal parameter number in definition of \mymacro`").

The `collect` keyword, when set to `false`, indicates that tokens should not be collected. This makes sense if the code processing the matched tokens does not modify them. Here's how to count vowels using a counter:

```
\newcount\voyel \voyel=0
\toksdo[collect=false]{ \S{aeiouy} -> \advance\voyel 1 }{happy texing}<\the\voyel>
```

<4>

The argument of `\setcharcode` and `\setcatcode` is evaluated using the `\numexpr` primitive; therefore, any arithmetic expression accepted by this primitive is valid. Here, all vowels are capitalized:

```
\toksdo{ \S{aeiouy} -> \setcharcode{\selfcharcode + `A - `a} }{Two plus two equals four}
```

TwO plUs twO EqUAls fOUr

Here's how to extract the tokens located in the highest level of nested braces in two steps:

```
\newcount\nestcnt \nestcnt=0 % count level of nesting
\newcount\maxnestcnt \maxnestcnt=0 % is the highest level of nesting
\def\mycode{12{34{5{67}8}}9{{ab}c}{{d{e}f}}g}
\toksdo[expand arg=1,collect=false]{
        \R{*:1} -> \advance\nestcnt1 \ifnum\nestcnt>\maxnestcnt \maxnestcnt=\nestcnt \fi,
        \R{*:2} -> \advance\nestcnt-1
        }{\mycode}Max nesting level = \the\maxnestcnt\par
Most nested tokens:
\toksdo[expand arg=1]{
        \R{*:1} -> \advance\nestcnt1  \deltok,% delete "{"
        \R{*:2} -> \advance\nestcnt-1 \deltok,% delete "}"
        \R{*:*} -> \ifnum\nestcnt<\maxnestcnt \deltok \fi% delete everything except at the highest nesting level
        }{\mycode}
```

Max nesting level = 3
Most nested tokens: 67e

2.7 Using \toksdo and \addtok

When the `collect` key is set to `true`, tokens are collected. Once they have been processed and optionally modified by `\setcharcode` or `\setcatcode`, each is added as-is to the internal collector, which is called upon at the end of the process to display the tokens or assign them if the `assign` key specifies this.

It is possible to modify how tokens are added to the internal collector via the `\addtok`⟨*code*⟩`}` macro, where ⟨*code*⟩ is an arbitrary code in which `\self` represents the token itself.

By default, and at the beginning of each code block following "⟨*patterns*⟩ `->` ", the `\addtok` macro is initialized to its default value:

```
\addtok{\self}
```

which means that each token must be added as-is.

If we write `\addtok{\self\self}`, the matched tokens are duplicated. With `\addtok{\fbox{\self}}`, they are framed using the `\fbox` macro from LaTeX.

The `\deltok` macro is equivalent to `\addtok{}`.

In most cases, `\self` represents a single token: the one currently being processed. It is *only* when a macro is detokenized (assigning 12 to its catcode) that `\self` represents multiple tokens.

In this example, each macro is de-tokenized and boxed, and all spaces are replaced by "␣_␣":

```
\fboxsep=2pt
\toksdo{ \R{*:16} -> \setcatcode{12}\addtok{\fbox{\self}},% detokenize and \fbox
         \R{*:10} -> \addtok{ \string_ }
       }{a b\foo12. 3\baz- 9}
```

a _ b$\boxed{\texttt{\\foo}}$12. _ 3$\boxed{\texttt{\\baz}}$- _ 9

Each 1 is doubled, and each 0 is replaced with a space:

```
\toksdo{ \S{1} -> \addtok{\self\self},
         \S{0} -> \setcatcode{10}% also possible: \addtok{ }
       }{10110101}
```

11 1111 11 11

The macros `\selfindex`, `\selfcharcode`, and `\selfcatcode` must *not* be used within the argument of `\addtok`, as they will be added as-is to the internal collector. They will only be expanded at runtime, at which point they will contain the data from the last token. If you wish to perform tests on the index, charcode, or catcode of the token to be added, you must do so outside the argument of `\addtok`. Here is the source code, the result of which can be seen on page 4:

```
In the tokens “\verb|ab \foo{0 123}c d”|, those that match the pattern “\verb|\S{1{a }\foo”}| are boxed in red:
\hfill
\fboxsep=1.5pt
\toksdo{ \S{1{a }\foo} -> \ifnum\selfcatcode=10 % if space
                              \addtok{{\color{red}\fbox{\strut\textvisiblespace}}}%
                          \else% if not a space
                              \setcatcode{12}% detokenize
                              \addtok{{\ttfamily\color{red}\fbox{\strut\self}}}% frame red
                          \fi,
          \R{*:*}      -> \addtok{\,{\ttfamily\self}\,}% for other token add thin space
       }{ab \foo{0 123}c d}
\hfill\null
```

In the tokens “ab \foo{0 123}c d”, those that match the pattern “\S{1{a }\foo}” are boxed in red:
a b ␣ \foo { 0 ␣ 1 2 3 } c ␣ d

3 Counting tokens with \tokscount

The macro

\tokscount[⟨*keys*⟩=⟨*values*⟩]{⟨*patterns*⟩}{⟨*tokens*⟩}

counts how many tokens match the ⟨*patterns*⟩ in the ⟨*tokens*⟩. If no patterns are specified (empty argument), the macro counts all tokens.

The available ⟨*keys*⟩ are as follows:

Key	Default value	Description
expand arg	0	number of expansions for the argument containing the ⟨*tokens*⟩ before being taken into account
assign	⟨*empty*⟩	assignment directive. If ⟨*empty*⟩, displays the number of tokens that matched. Otherwise, must contain an assignment statement of the type \def⟨*macro*⟩ to assign the result to the ⟨*macro*⟩
assign match	⟨*empty*⟩	assignment directive. If ⟨*empty*⟩, the matched tokens are not collected. Otherwise, must contain an assignment statement \def⟨*macro*⟩ to assign them to a macro or ⟨*macro*⟩ to assign them to a token register

The ⟨*patterns*⟩ accepted by \tokscount are the same as those for \toksdo.

```
1) \tokscount{ \r{a-z} | \S{10} }{ab1023truc098}\quad
2) \tokscount[assign=\def\cc]{ \R{*:16} }{ab\x123truc\zzz0\yy98}<\cc>\quad% counts macros
3) \tokscount[assign=\def\cc,assign match=\def\xx]{ \S{01} }{1{a01}1{0{b100}}1}<\cc><\xx>\quad% counts 0 and 1
4) \tokscount{ \R{*:12} | \S{\foo\bar\zid}  }{ab\foo c12d*-ef}\quad% counts tokens "letters" or macros \foo\bar\zid
5) \tokscount{}{12{34}5}% counts all tokens
```

1) 9   2) <3>   3) <9><101101001>   4) 5   5) 7

4 PEG Grammars

In a set of tokens, some may match patterns that can be organized into a PEG (Parsing Expression Grammar). Whenever a match occurs with a pattern, the tokens that matched are consumed and are no longer available for subsequent patterns (except for predicates; see below). If there is no match, no tokens are consumed.

4.1 The 5 Pattern Markers

The tokstools package provides 5 basic patterns defined by 5 markers for constructing more complex patterns:

- the pattern \r{⟨*csv character ranges*⟩:⟨*csv catcode ranges*⟩} described on page 4;

- the pattern `\R{`⟨*csv charcode ranges*⟩`:`⟨*csv catcode ranges*⟩`}` described on page 4;
- the pattern `\S{`⟨*tokens*⟩`}` described on page 4;
- the pattern `\s{`⟨*tokens*⟩`}`: any set of tokens consisting exactly of the ⟨*tokens*⟩ passed as an argument to `\s` matches this pattern: this is therefore an exact match between strings of tokens.

  Just as with the `\S` marker, it is possible to change the catcodes of certain tokens using the `\c` marker according to the syntax `\c{`⟨*catcode*⟩`}{`⟨*tokens*⟩`}`;
- the pattern `\.` which matches any token and is equivalent to `\R{*:*}`.

4.2 Repetitions

Each ⟨*1-pattern*⟩ can be followed by a repetition marker if you wish to specify the number of times the match occurs. The repetition markers are:

- `^`⟨*digit*⟩ or `^{`⟨*number*⟩`}`: requires that the match be repeated exactly the number of times specified as an argument to `^`;
- `^{`⟨*min*⟩`-`⟨*max*⟩`}`: specifies that the match must occur between ⟨*min*⟩ and ⟨*max*⟩. If ⟨*min*⟩ is omitted, it is taken as 0. If ⟨*max*⟩ is omitted, there is no upper limit on the number of repetitions. Both numbers cannot be omitted at the same time.
- "+": 1 or more repetitions (equivalent to `^{1-}`);
- "*": 0 or more repetitions (equivalent to `^{0-}`);
- "?": 0 or 1 repetition (equivalent to `^{0-1}`);

If no repetition marker is present after a ⟨*1-pattern*⟩, the directive `^1` is implicitly applied.

It is important to note that *in all cases*, the maximum possible number of repetitions is consumed; this behavior is always "greedy".

4.3 Predicates

Each ⟨*1-pattern*⟩ can be preceded by a predicate, which is either "`!`" or "`&`":

- The predicate `!`⟨*1-pattern*⟩ matches if there is no match with the ⟨*1-pattern*⟩
- The predicate `&`⟨*1-pattern*⟩ matches if there is a match with the ⟨*1-pattern*⟩.

In both cases, no tokens are consumed if there is a match.

4.4 Pattern concatenation

The character "`:`" between two ⟨*1-pattern*⟩ indicates that the first *and* the second should match. Of course, we are not limited to two patterns; there can be as many as desired.

`&\r{0-9}^{3-} : \r{0-9}^2`	if there are at least 3 digits, matches the first 2
`\r{a-z}+ : \R{*:10} : \S{01}^2`	matches one or more letters followed by a space followed by 2 binary digits
`\r{1-9}+ : \s{00} : !\.`	matches a number consisting of at least 1 non-zero digit followed by "`00`", which must be the *last tokens*, because the predicate "`!\.`" means "must not be followed by a token"

4.5 Choosing between patterns

The character "`|`" between two ⟨*1-patterns*⟩ indicates that a match should be made with either the first *or* the second. Obviously, we are not limited to two patterns; there can be as many as desired. If a match occurs for one of the patterns, it is selected, and none of the subsequent patterns are tested: this is therefore an ordered choice. If we write

```
\r{0-9}^5 | \S{01}^2
```

the second test will never be performed since it is included in the first. It is therefore important to start with the most specific tests and end with the most general ones if the ranges of the patterns overlap or, worse, are nested.

The concatenation operator "`:`" takes precedence over the selection operator "`|`", so

⟨*a*⟩ `:` ⟨*b*⟩ `|` ⟨*c*⟩ `|` ⟨*d*⟩ `:` ⟨*e*⟩

is interpreted as ⟨*a*⟩ `:` ⟨*b*⟩ or ⟨*c*⟩ or ⟨*d*⟩`:`⟨*e*⟩.

4.6 Pattern grouping

Any ⟨*1-pattern*⟩ or combination of <1-pattern>s denoted by ⟨*patterns*⟩ can be enclosed in curly braces to form a new ⟨*1-pattern*⟩, which can, in turn, be preceded by a predicate and followed by a repetition specification according to the syntax described above:

⟨*predicate*⟩`{`⟨*pattern combination*⟩`}`⟨*repetitions*⟩

4.7 Spaces

In pattern syntax, spaces are ignored.

Thus

```
\r{0-9} ^ 2 :  \r{a-z}^3 : \r{a-z}^ 2
```

is equivalent to

```
\r{0-9}^2:\r{a-z}^3:\r{a-z}^2
```

4.8 Precedences

The precedences of operators on patterns are (the higher the number, the higher the precedence):

`Operator`	Precedence	Description
`{`⟨*patterns*⟩`}`	5	pattern grouping
⟨*1-pattern*⟩`?`	4	matches 0 or 1 time
⟨*1-pattern*⟩`*`	4	matches 0 or more time
⟨*1-pattern*⟩`+`	4	matches 1 or more times
⟨*1-pattern*⟩`^{n}`	4	matches `n` times
⟨*1-pattern*⟩`^{a-b}`	4	matches between `a` and `b` times
`!`⟨*1-pattern*⟩	3	matches if ⟨*1-pattern*⟩ does not match, without consuming tokens
`&`⟨*1-pattern*⟩	3	matches if ⟨*1-pattern*⟩ matches, without consuming tokens
⟨*1-pattern1*⟩`:`⟨*1-pattern2*⟩	2	matches ⟨*1-pattern1*⟩ then ⟨*1-pattern2*⟩
⟨*1-pattern1*⟩`|`⟨*1-pattern2*⟩	1	matches ⟨*1-pattern1*⟩ or ⟨*1-pattern2*⟩ (ordered choice)

4.9 Pattern names

Any pattern can be defined and named so that it can be reused later using a simpler, easier-to-remember syntax:

`\defpattern⟨pattern name⟩{⟨patterns⟩}`

The ⟨`pattern name`⟩ *must* be a control sequence. This control sequence is not defined or redefined by `\defpattern`; it is also a marker whose meaning is irrelevant.

If ⟨`patterns`⟩ contains a ⟨`pattern name`⟩, that name must have been defined previously, otherwise a compilation error will occur.

5 Testing for a match with `\ifpegmatch`

5.1 Syntax

The `\ifpegmatch` macro has the following syntax

`\ifpegmatch[⟨keys⟩=⟨values⟩]{⟨patterns⟩}{⟨tokens⟩}{⟨code if match⟩}{⟨code if no match⟩}`

When all values are set to their defaults, this macro tests whether the ⟨`patterns`⟩ match tokens found at the beginning of the ⟨`tokens`⟩.

The available ⟨`keys`⟩ are:

Key	Default value	Description
`expand arg`	`0`	number of expansions the argument containing the ⟨`tokens`⟩ must undergo before being considered
`mode`	`1`	match search mode
`capture name`	⟨`empty`⟩	capture names
`assign prematch`	`\def\prematchtoks`	assignment rule for tokens preceding those that matched
`assign match`	`\def\matchtoks`	assignment rule for tokens that have matched
`assign postmatch`	`\def\remaintoks`	assignment rule for tokens preceding those that have matched

In this example, we check whether the ⟨`tokens`⟩ begin with two digits, an optional space, and at least one lowercase letter

```
\defpattern\okmatch{ \r{0-9:12}^2 : \R{*:10}? : \r{a-z:11}+ }
1) \ifpegmatch{\okmatch}{73 ab:*ij}{T}{F}\quad
2) \ifpegmatch{\okmatch}{45foobar2000}{T}{F}\quad
3) \ifpegmatch{\okmatch}{854tex 8}{T}{F}\quad
4) \ifpegmatch{\okmatch}{1 2 b3c}{T}{F}\quad
```

1) T   2) T   3) F   4) F

Let's build a simple grammar to check whether an argument begin with an arithmetic operation of the form

⟨`relative integer`⟩⟨`operation`⟩⟨`positive integer`⟩

```
\defpattern\sp{ \R{*:10} } % space
\defpattern\digit{ \r{0-9} }% digit
\defpattern\posint{ \digit+ }% positive integer
\defpattern\int{ \S{+-}? : \posint }% +- integer
\defpattern\op{ \S{+-*/} }% math operation
\defpattern\okmatch{ \sp* : \int : \sp* : \op : \sp* : \posint : \sp* }
1) \ifpegmatch\okmatch{2*3}{T}{F}\quad
2) \ifpegmatch\okmatch{-7 + 1 }{T}{F}\quad
3) \ifpegmatch\okmatch{ a + 9 }{T}{F}\quad
4) \ifpegmatch\okmatch{ -2 / 6 + 3 }{T}{F}\quad
5) \ifpegmatch\okmatch{ +2026- 4068}{T}{F}\quad
6) \ifpegmatch\okmatch{ -3 }{T}{F}\quad
7) \ifpegmatch\okmatch{2a-3b}{T}{F}\par
The primitive \verb|\int| is not redefined: $\int x^2dx$
```

1) T   2) T   3) F   4) T   5) T   6) F   7) F
The primitive `\int` is not redefined: $\int x^2dx$

With the predicate "`: !\.`" added to the end of `\okmatch`, line 4 would have displayed "F" because "+␣3␣" remains after the tokens that matched.

5.2 Mode

The `\ifpegmatch` macro can search for matches in several modes, specified via the "`mode`" key:

- By default, `mode=1` indicates that the match must occur at the beginning of the ⟨*tokens*⟩;
- `mode=0` indicates that all ⟨*tokens*⟩ must match; this is the strictest mode;
- `mode=2` indicates that the match can occur anywhere within the ⟨*tokens*⟩; this is the least restrictive mode.

5.3 Token Output

The `\ifpegmatch` macro also returns some information:

- by default, the `\matchtoks` macro contains the tokens that matched and is empty if no match was found;
- by default, the `\remaintoks` macro contains the tokens remaining after those that matched;
- by default, the `\prematchtoks` macro contains the tokens preceding those that matched (can only contain tokens when `mode=2`);
- the `\matchposition` macro contains the position of the first token that matched and 0 if no match occurred.

```
1) \ifpegmatch[mode=0]{ \r{A-Z}^3 }{1ABC6}{T}{F}, match=<\matchtoks>, remain=<\remaintoks>, pos=\matchposition\par
2) \ifpegmatch[mode=1]{ \r{A-Z}^3 }{1ABC6}{T}{F}, match=<\matchtoks>, remain=<\remaintoks>, pos=\matchposition\par
3) \ifpegmatch[mode=2]{ \r{A-Z}^3 }{1ABC6}{T}{F}, match=<\matchtoks>, remain=<\remaintoks>, pos=\matchposition\par
4) \ifpegmatch[mode=0]{ \r{A-Z}^3 }{ZZZ12}{T}{F}, match=<\matchtoks>, remain=<\remaintoks>, pos=\matchposition\par
5) \ifpegmatch[mode=1]{ \r{A-Z}^3 }{ZZZ12}{T}{F}, match=<\matchtoks>, remain=<\remaintoks>, pos=\matchposition\par
6) \ifpegmatch[mode=2]{ \r{A-Z}^3 }{ZZZ12}{T}{F}, match=<\matchtoks>, remain=<\remaintoks>, pos=\matchposition
```

```
1) F, match=<>, remain=<1ABC6>, pos=0
2) F, match=<>, remain=<1ABC6>, pos=0
3) T, match=<ABC>, remain=<6>, pos=2
4) F, match=<>, remain=<ZZZ12>, pos=0
5) T, match=<ZZZ>, remain=<12>, pos=1
6) T, match=<ZZZ>, remain=<12>, pos=1
```

5.4 Captures

The `\c` token, when placed before a ⟨*1-pattern*⟩, indicates that tokens matching this ⟨*1-pattern*⟩ must be captured along with their positions. If a ⟨*1-pattern*⟩ is preceded by a predicate, no capture is performed. Captures are sorted in the chronological order in which they were made and are returned, in two forms, by `\tokscapture{`⟨*index*⟩`}` or by `\tokscapture{`⟨*name*⟩`:`⟨*index*⟩`}`. The ⟨*name*⟩ is optional and is set with the name key, and the ⟨*index*⟩ is the capture's sequence number.

If ⟨*index*⟩ is 0, all captures are enclosed in curly braces and listed in a CSV file in the following format:

`{`⟨*capture*$_1$⟩`},{`⟨*capture*$_2$⟩`},...,{`⟨*capture*$_n$⟩`}`

Similarly, the macro `\poscapture{`⟨*index*⟩`}` or `\poscapture{[`⟨*name*⟩`:]`⟨*index*⟩`}` returns the capture positions, with the difference that if ⟨*index*⟩ is 0, the positions are placed in a CSV file *without being enclosed in curly braces*:

⟨*position*$_1$⟩`,`⟨*position*$_2$⟩`,...,`⟨*position*$_n$⟩

If a ⟨*index*⟩ exceeds the maximum index, an error message is issued.

If `\c` appears after a ⟨*1-pattern*⟩, only the position is captured.

In this example, two complete captures (tokens+position) and one position capture are performed:

```
\ifpegmatch[mode=2]{ \c\r{a-z}+ : \c\r{0-9}^2\c }{12abc666def}{T}{F}\par
toks : <\detokenize\expandafter\expandafter\expandafter{\tokscapture{0}}>\qquad
1=<\tokscapture{1}>, 2=<\tokscapture{2}>\par

pos : <\poscapture{0}>\qquad
1=<\poscapture{1}>, 2=<\poscapture{2}>, 3=<\poscapture{3}>
```

```
T
toks : <{abc},{66}>        1=<abc>, 2=<66>
pos : <3,6,8>        1=<3>, 2=<6>, 3=<8>
```

In this example, we define a grammar that matches a scientific notation of the form $a \times 10^b$ and captures both numbers a and b:

```
\defpattern\sp{ \R{*:10} }
\defpattern\sign{ \S{+-} }
\defpattern\digit{ \r{0-9} }
\defpattern\integer{ \digit+ }
\defpattern\decsep{ \S{.,} }
\defpattern\scidec{ \sign? : \r{1-9} : {\decsep : \digit+}?  }
\defpattern\opbr{ \R{*:1} }
\defpattern\clbr{ \R{*:2} }
\defpattern\^{ \R{*:7} }
\defpattern\exponent{ \opbr : \sp? : \c{\sign? : \sp? : \integer} : \sp? : \clbr | \c\digit }
\defpattern\sci{\c\scidec : \sp? : \s{\times10} : \sp? : \^ : \sp? :\exponent}
1) \ifpegmatch\sci{3\times10^5}            {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
2) \ifpegmatch\sci{-2.25\times10^{-3}}      {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
3) \ifpegmatch\sci{-0.75\times10^7 }        {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
4) \ifpegmatch\sci{15\times10^0}            {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
5) \ifpegmatch\sci{1.5\times10^ 1 }         {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
6) \ifpegmatch\sci{-2.75 \times 10 ^ { 11 }}{<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
7) \ifpegmatch\sci{-9.96\times10^{ -2 }}    {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
8) \ifpegmatch\sci{-0\times10 ^0}           {<\tokscapture{1}> <\tokscapture{2}>}{Faux}\par
9) \ifpegmatch\sci{-1\times10^ {- 7 }}      {<\tokscapture{1}> <\tokscapture{2}>}{Faux}
```

```
1) <3> <5>
2) <-2.25> <-3>
3) Faux
4) Faux
5) <1.5> <1>
6) <-2.75> <11>
7) <-9.96> <-2>
8) Faux
9) <-1> <- 7>
```

5.5 Recursive Grammars

The `tokstools` package supports recursive grammars, but no optimization is performed, and the way they are handled remains naive. Consequently, certain recursive grammars—particularly those involving “left recursion”—will result in infinite loops.

That said, it is possible to use recursive grammars, but adding captures tends to be somewhat unpredictable because the order of these captures is not obvious and depends on the very definition of the grammar and thus on the resulting tree traversal.

Here is a recursive grammar capturing an arithmetic expression involving the four operations with parentheses:

```
\defpattern\num{ \r{0-9}+ }
\defpattern\term{ \num | \s{(} : \expr : \s{)} }
\defpattern\factor{ \term : {\S{*/} : \term }* }
\defpattern\expr{ \factor : {\S{+-} : \factor }* }
1) \ifpegmatch[mode=0]\expr{1+3}{T}{F}\quad
2) \ifpegmatch[mode=0]\expr{1-2*3}{T}{F}\quad
3) \ifpegmatch[mode=0]\expr{3*4+6}{T}{F}\quad
4) \ifpegmatch[mode=0]\expr{3-(1-2*3)}{T}{F}\quad
5) \ifpegmatch[mode=0]\expr{3*4*(1-3*2-(1-3/7)*3)/(1/7+2*3)*3-5}{T}{F}\quad
6) \ifpegmatch[mode=0]\expr{6-9*(2-3)+4/5}{T}{F}
```

```
1) T   2) T   3) T   4) T   5) T   6) T
```

This grammar ensures that the expression begins with a parenthesis and contains balanced parentheses:

```
\defpattern\nobrtext{ { !\S{()} : \. }+ }
\defpattern\inparen{ \nobrtext : \inparen* | \s{(} : \inparen* : \s{)} }
\defpattern\expr{ &\s{(} : \inparen : !\. }% predicates -> must start with '(' and end with ')'
1) \ifpegmatch\expr{a(b)c}{T}{F}\quad
2) \ifpegmatch\expr{(a(abc)()d)}{T}{F}\quad
3) \ifpegmatch\expr{(a(bc))df)}{T}{F}\quad
4) \ifpegmatch\expr{((abc)d((e)f)g)}{T}{F}\quad
5) \ifpegmatch\expr{((foo)b(((b)a)r)}{T}{F}
```

1) F   2) T   3) F   4) T   5) F

6 Counting matches with \pegcount

The macro

`\pegcount[⟨keys⟩=⟨values⟩]{⟨patterns⟩}{⟨tokens⟩}`

counts how many times ⟨*patterns*⟩ match in the ⟨*tokens*⟩. Each position is saved, and each match is captured so it can be easily retrieved.

No capture explicitly requested by \c is allowed and is ignored.

The available ⟨*keys*⟩ are:

Key	Default value	Description
`expand arg`	`0`	number of expansions the argument containing the ⟨*tokens*⟩ must undergo before being taken into account.
`assign`	⟨*empty*⟩	if empty, displays the number of matches. Otherwise, must contain an assignment statement of the type `\def`⟨*macro*⟩ to assign the result to the ⟨*macro*⟩
`assign positions`	`\def\matchposlist`	assignment statement for the list of positions. If ⟨*empty*⟩, no capture is performed. Otherwise, must contain an assignment statement `\def`⟨*macro*⟩ to assign the list of positions to a ⟨*macro*⟩
`name`	⟨*empty*⟩	is the name of the captures returned by `\tokscapture`

Captures are returned, in two expansions, by `\tokscapture{[`⟨*name*⟩`:]`⟨*index*⟩`}`. The ⟨*name*⟩ is optional and is set using the ⟨*name*⟩ key, and the ⟨*index*⟩ is the capture index. If ⟨*index*⟩ is 0, all captures are enclosed in curly braces and listed in a CSV file in the following format:

`{⟨capture₁⟩},{⟨capture₂⟩},...,{⟨captureₙ⟩}`

```
1) \pegcount{ \r{0-9}+ : \r{a-z}+ }{foo25bar}, <\matchposlist>\par% numbers followed by letters
2) \pegcount{ \r{0-9}+ : \r{a-z}+ }{a12bcd,4b,z875bar}, <\matchposlist>,
"\tokscapture{0}", 1="\tokscapture{1}", 2="\tokscapture{2}", 3="\tokscapture{3}"\par% numbers followed by letters
3) \pegcount{ \s{+} : {\r{a-c}^2 : \r{0-9}}+ : \s{+} }{+ab3+..+bb6ab8ca7+..+aa1bb2+..},
      <\matchposlist>,
      1="\tokscapture{1}", 2="\tokscapture{2}", 3="\tokscapture{3}"
```

1) 1, <4>
2) 3, <2,8,12>, "12bcd,4b,875bar", 1="12bcd", 2="4b", 3="875bar"
3) 3, <1,8,21>, 1="+ab3+", 2="+bb6ab8ca7+", 3="+aa1bb2+"

7 Replacements with \pegreplace

The macro

`\pegreplace[⟨keys⟩=⟨values⟩]{⟨patterns⟩}{⟨tokens⟩}{⟨replacement⟩}`

searches for all occurrences of ⟨*patterns*⟩ within the ⟨*tokens*⟩ and, for each match, replaces the tokens with the code defined in ⟨*replacement*⟩.

The available ⟨*keys*⟩ are:

Key	Default value	Description
`expand arg`	`0`	number of expansions the argument containing the ⟨*tokens*⟩ must undergo before being taken into account.
`all`	`true`	searches for all matches if `true` and only the first if `false`
`assign`	⟨*empty*⟩	if empty, displays the ⟨*tokens*⟩ obtained after performing the replacements. Otherwise, must contain an assignment statement to assign the result to a macro using `\def`⟨*macro*⟩ or to a token register.

The ⟨*replacement*⟩ is an arbitrary code that may contain "`\0`", which means "tokens that matched", "`\1`" is the first capture made by `\c`, "`\2`" is the second, and so on up to "`\9`".

Frame `x` and `y` in a mathematical expression, then `\alpha` and `\beta` along with their coefficients, but in math mode:

```
\fboxsep=1pt
\pegreplace{ \S{xy} }{2x-3y=-4-x+4y}{\fbox{\strut\0}}\par
\pegreplace{ \r{1-9}* : \S{\alpha\beta} }{$2\alpha-3\beta=-4-\alpha+4\beta$}{\fbox{\strut$\0$}}
```

2$\boxed{\text{x}}$-3$\boxed{\text{y}}$=-4-$\boxed{\text{x}}$+4$\boxed{\text{y}}$

$\boxed{2\alpha} - \boxed{3\beta} = -4 - \boxed{\alpha} + \boxed{4\beta}$

In this example, we create a grammar that matches the French standards for postal codes and city names (consisting of words and hyphens).

```
\defpattern\sp{ \R{*:10} }
\defpattern\CP{\c\r{0-9}^2 : \sp? : \c\r{0-9}^3 }% \1=2 first digits  \2=3 last digits
\defpattern\upcase{ \r{A-Z,À,É} }
\defpattern\lowcase{ \r{a-z,é,è,à,ê,ô,ç} }
\defpattern\ville{ \upcase : \lowcase+ : { \S{-} : {\upcase | \lowcase} : \lowcase+ }* }
\defpattern\CPville{ \CP : \sp : \c\ville }% capture CP and capture name
1) \pegreplace\CPville{The town 75000 Paris}{CodePostal=\textbf{\1\2} est \fbox{\3}}\par
2) \pegreplace\CPville{Destination 64 500 Saint-Jean-de-Luz suite}{CodePostal=\textbf{\1\2} est \fbox{\3}}
```

1) The town CodePostal=**75000** est $\boxed{\text{Paris}}$

2) Destination CodePostal=**64500** est $\boxed{\text{Saint-Jean-de-Luz}}$ suite

Let's come up with a little riddle involving two-digit numbers that are not zero:

```
\defpattern\num{ \c\r{1-9} : \c\r{1-9} : !\r{0-9} }
If \pegreplace\num{27, 34 and 43}{\1\2 gives \the\numexpr\1*\2+\2\relax}, what gives 57 ? (Answer in 7.5 million years)
```

If 27 gives 21, 34 gives 16 and 43 gives 15, what gives 57 ? (Answer in 7.5 million years)

8 Errors

Among the many errors that can occur, here are a few...

8.1 Unbalanced Braces

If a result or capture consists of tokens with unbalanced braces, a compilation error will occur.

For example, if you use `\toksdo` to remove the closing braces:

```
\toksdo{ \R{*:2} -> \deltok }{foo{\bfseries123}bar}
```

the error reported by `tokstools` is "`! Unbalanced open-group token, 1 close-group token added`".

To balance the braces, a closing brace is therefore added to the end of the tokens passed as arguments, and the tokens "`foo{\bfseries123bar}`" are sent to the display.

Similarly, if we remove the opening braces:

```
\toksdo{ \R{*:1} -> \deltok}{foo{\bfseries123}bar}
```

the error message is "`! Unbalanced close-group token ignored`".

The extra closing brace is therefore ignored, and the tokens "`foo\bfseries123bar`" are sent to the display

8.2 Engine not suitable for encoding

Using an 8-bit engine *requires* the use of tokens only, particularly for the `\r` token.

Compiling this code with an 8-bit engine

```
    \ifpegmatch{ \r{a,e,i,o,u,y,é,à,ê,ù} }{Un été à l'océan}{T}{F}
```

causes a compilation error because "é" consists of 2 tokens (charcodes 195 and 169, catcodes equal to 13), whereas the syntax of `\r` requires that there be only one. The compilation error reported by `tokstools` is "`! Multiple token 'é', '0' inserted`".

The correct syntax would be

```
    \ifpegmatch{ \r{a,e,i,o,u,y} | \S{éàêù} }{Un été à l'océan}{T}{F}
```

8.3 Invalid Interval

Intervals of the form ⟨*a*⟩-⟨*b*⟩, where `a` and `b` are either tokens or numbers, *must* be in the correct order.

Since the token "ń" comes *after* "ó" in UTF-8, the code

```
    \ifpegmatch{ \r{ń-ó} }{Some text}{T}{F}
```

causes the compilation error "`! Unsorted interval in 'ń-ó', 'ó-ń' inserted`". The range is corrected by `tokstools`.

8.4 Invalid Pattern Syntax

Spaces are ignored, but any pattern syntax that does not conform to what is described starting on page **??** will cause a compilation error.

For example

```
    \pegcount{ \r{a-z} | S{10} }{ab1023truc098}
```

causes the error "`! Found "S" when expecting \r, \R, \s, \S or \.`".

8.5 Capture Errors

If a ⟨*name*⟩ has not been defined or if a ⟨*index*⟩ is requested outside of those assigned during captures, a compilation error is returned.

```
    \pegcount{ \r{a-z}^2 }{ab1023truc098}
```

creates 3 captures, so requesting

```
    \tokscapture{4}
```

results in the compilation error "`! Undefined token capture at index "4"`".

9 List of macros and markers

9.1 List of commands

Here are the macros available to the user:

- `\printtoks`, see page 3;
- `\toksdo`, `\setcharcode`, `\setcatcode`, `\deltok`, `\addtok`, `\selfcharcode`, `\selfcatcode` and `\selfindex`, see pages 5 and following;
- `\tokscount`, see page 8;
- `\defpattern`, see page 10;
- `\ifpegmatch`, see page 11;
- `\pegcount`, see page 14;
- `\pegreplace`, see page 14.

The following macros are modified by the `\toksdo` macro, but are restored to their previous state once `\toksdo` has finished executing: `\setcharcode`, `\setcatcode`, `\deltok`, `\addtok`, `\selfcharcode`, `\selfcatcode` and `\selfindex`.

9.2 Liste of markers

- `\r{`⟨*csv car*⟩`:`⟨*csv catcode*⟩`}`, see page 4;
- `\R{`⟨*csv charcodes*⟩`:`⟨*csv catcode*⟩`}`, see page 4;
- `\s{`⟨*tokens*⟩`}`, see page 9;
- `\S{`⟨*tokens*⟩`}`, see page 4;
- `\.`, see page 9;
- `\c` has two syntaxes

  `\c{`⟨*catcode*⟩`}{`⟨*tokens*⟩`}` when used in the argument of `\s` or `\S`, see page 4;

  `\c` when placed before or after a ⟨*1-pattern*⟩, see page 12;
- `\self`, see page 7;
- `\0`, `\1` up to `\9`, see page 15.

⋆

⋆ ⋆

This package, version 0.1, is still in the experimental stage, and it is very likely that, despite the extensive testing that has been done, it contains many bugs.

Furthermore, some features or syntax may still change.

In any case, I hope you find `tokstools` useful. Feel free to contact me via **`email`** to report bugs, glitches, or suggestions for *realistic* features and above all, don't waste your time doing so on `https://tex.stackexchange.com` or any other site.