L'extension pour TEX et LATEX

tokstools

v 0.2

17 avril 2026

Christian Tellechea
unbonpetit@netc.fr

Cette extension pour TEX ou LATEX met des outils basés sur des grammaires de type PEG (Parsing Expression Grammars) pour manipuler des tokens. Dans un ensemble arbitraire de tokens équilibrés en accolades, il est possible d'établir des règles de correspondance pour tester si une correspondance existe, de les compter, d'effectuer des remplacements, de capturer des tokens, etc.

Table des matières

1 Présentation

L'extension tokstools fonctionne avec tout moteur. Elle n'est pas limité à LaTeX seulement, comme l'est malheureusement le package pegmatch, et pour l'utiliser, il faut écrire

- `\input tokstools.tex` sous (pdf)(Xe)(lua)(op)TeX (la syntaxe `\input{tokstools.tex}` est également admise) ;
- `\usepackage{tokstools}` sous (pdf)(Xe)(lua)LaTeX.

Ce package ne requiert que le package simplekv qui sera chargé si ça n'a pas été le cas.

Avertissement : dans ce manuel, une (grande) liberté sera prise quant aux termes employés et des mots anglais seront très fréquemment utilisés, autant par commodité et habitude que par pure flemme de rédiger des tournures 100 % françaises plus lourdes ou alambiquées.

mot/expression	**en français**
token	unité syntaxique
match	correspondance
les tokens ayant matché	les tokens ayant produit une correspondance
catcode	code de catégorie (est un entier de 0 à 16)
charcode	code de caractère (est un entier)
liste csv	liste d'éléments séparés par des virgules
package	extension ou paquet

1.1 Nouveau dans la version 0.2

Macro `\pegreplace` La macro `\pegreplace` change de syntaxe tout en offrant davantage de fonctionnalités, voir page 15. Ce changement casse malheureusement la syntaxe de la version 0.1, j'en suis sincèrement désolé et présente mes excuses aux utilisateurs de tokstools pour ce désagrément.

Macro `\printtoks` La macro `\printtoks` a désormais la possibilité, selon la valeur de la nouvelle clé mode, de diriger les informations des tokens vers le fichier log, voir page 3.

Validité des directives d'assignation Désormais, aucune vérification de validité n'est effectuée pour les « directives d'assignation » passées par la clé assign et similaires. L'utilisateur doit donc faire le choix éclairé entre :

- `assign={}` qui est une directive vide pour diriger le résultat dans le flux de lecture de TeX en vue de l'afficher ;
- `assign=\def`⟨*macro*⟩ pour que le résultat soit assigné à la ⟨*macro*⟩ ;
- `assign=`⟨*registre de tokens*⟩ pour assigner le résultat à un registre de token, ce qui s'avère nécessaire lorsque y figurent des tokens de catcode 6 (#) pour désigner des arguments.

1.2 Tokens, charcode, catcode, moteurs

Le package tokstools agit sur des ensembles de tokens (ou chaines de tokens). Les tokens sont la plus petite unité du code source manipulée par TeX.

L'utilisateur doit avoir conscience que la notion de tokens dépend du type de moteur utilisé. Ce n'est *que* pour les moteurs 8 bits, tels que pdfLaTeX par exemple, que 1 token = 1 octet. À titre d'exemple, le caractère « € » est vu

- par pdfLaTeX comme 3 tokens dont les charcodes sont 226 ; 130 et 172 et les catcodes sont tous égaux à 13 ;
- par luaLaTeX comme 1 seul token de charcode 8364 et de catcode 12.

Cette différence doit être prise en compte et peut être source d'erreurs plus ou moins compréhensibles lorsqu'on manipule des caractères UTF8 sur un moteur 8 bits.

Ce manuel a été compilé avec luaLaTeX, et donc des tokens ayant des charcodes supérieurs à 255 peuvent exister.

Quelles que soient les macros mises à disposition par tokstools, elles décomposent toujours leur argument contenant les tokens, noté ⟨*tokens*⟩, en liste de tokens, chacun possédant un charcode et un catcode. Les catcodes visibles par tokstools sont listés dans le tableau ci-dessous, avec pour chacun les tokens les plus courant de charcode inférieur à 255 qui possèdent ces catcodes par défaut.

Catcode	Description	Tokens
1	début de groupe	`{`
2	fin de groupe	`}`
3	délimiteur de mode mathématique	`$`
4	tabulation d'alignement	`&`
6	caractère de paramètre	`#`
7	exposant	`^`
8	indice	`_`
10	espace	␣ et *HT*
11	lettres	`a–z` et `A–Z`
12	autres caractères	chiffres (`0–9`) signes de ponctuation (`: . ; , ? ! ' "`) signes mathématiques (`+ – * / = < >`) autres signes (`[ ] ( ) | @ ‘`)
13	caractère actif	`~`
16	séquence ou caractère de contrôle	\⟨*lettres*⟩ ou \⟨*caractère*⟩

Aucune des macros mises à disposition par `tokstools` n'est développable. Si l'on souhaite que le résultat de ces macros ou bien des captures soient contenus dans des macros purement développables, il faut utiliser des consignes d'assignation qui seront définies dans ce manuel.

1.3 Notations

Ce qui est appelé « marqueur » dans ce manuel est une séquence (ou caractère) de contrôle qui agit comme un repère afin d'être reconnue dans un argument, mais qui n'est pas modifiée par `tokstools` et dont la signification n'a aucune importance.

Dans tout le manuel :

- ⟨*tokens*⟩ désigne un ensemble de tokens[1] équilibrés en tokens de catcode 1 et 2 (accolades ouvrantes et fermantes) qui se trouve comme argument des macros ou des marqueurs en charge de traiter ces tokens ;
- ⟨*1–motif*⟩ représente un motif unique, c'est à dire constitué d'un marqueur avec son éventuel argument, précédé d'un prédicat optionnel et suivi d'une spécification optionnelle de répétition ;
- ⟨*motifs*⟩ représente un seul ⟨*1–motif*⟩ ou une combinaison de plusieurs ⟨*1–motif*⟩ obtenue avec l'opérateur de concaténation « `:` » et l'opérateur de choix ordonné « `|` » (seulement l'opérateur « `|` » pour `\toksdo` et `\tokscount`) ;
- `{`⟨*motifs*⟩`}` est un groupement qui admet un prédicat et une directive de répétition optionnels et qui se comporte comme un ⟨*1–motif*⟩.

1.4 Informations sur les tokens avec `\printtoks`

La macro

`\printtoks[`⟨*clés*⟩`=`⟨*valeurs*⟩`]{`⟨*tokens*⟩`}`

permet d'afficher les tokens avec, au-dessous de chacun, son charcode et son catcode. Si un token est une séquence de contrôle ou un caractère de contrôle, aucun charcode n'est affiché.

```
\catcode`\!=6 % "!" est similaire à "#"
\printtoks[code=\ttfamily]{\def\foo#1!2#3{\hbox to 1cm{\hss$!1^#2_!3$\hss}}}
\medbreak
\printtoks[code=\ttfamily]{é€eé}
```

\def	\foo	#	1	!	2	#	3	{	\hbox	t	o	␣	1	c	m	{	\hss	$	!	1	^	#	2	_	!	3	$	\hss	}	}
16	16	35 6	49 12	33 6	50 12	35 6	51 12	123 1	16	116 11	111 11	32 10	49 12	99 11	109 11	123 1	16	36 3	33 6	49 12	94 7	35 6	50 12	95 8	33 6	51 12	36 3	16	125 2	125 2

e	é	ė	€
101	233	279	8364
11	11	11	12

Les ⟨*clés*⟩ disponibles sont :

1. La primitive \par peut se trouver dans ⟨*tokens*⟩.

Clé	Valeur par défaut	Description
`expand arg`	`0`	nombre de développements que doit subir l'argument contenant les ⟨*tokens*⟩ avant d'être pris en compte
`code`	⟨*vide*⟩	code exécuté avant d'afficher un token
`intertoks`	`0.33em`	espace horizontale entre chaque token, insérée par la primitive `\hskip`
`printcharcode`	`true`	affiche ou pas le charcode
`printcatcode`	`true`	affiche ou pas le catcode
`hexcharcode`	`false`	affiche le charcode en base 16 si `true` et en base 10 sinon
`baselinecoeff`	`0.8`	coefficient d'espacement vertical
`vlines`	`true`	affiche ou pas les lignes verticales entre chaque token
`boxed`	`true`	met le tout dans une `\hbox` si `true`, ce qui rend le tout insécable
`mode`	`0`	si 1 ou 2, dirige les informations sur les tokens vers le fichiers log

Lorsque la clé `mode` vaut 1, les informations sur chaque token sont aussi dirigées vers le fichier log. Lorsque cette valeur est 2, c'est uniquement vers le fichier log et aucun affichage dans le pdf n'est produit.

Ainsi, le code

```
\def\foo#1{$#1$}
\catcode`\!=6 % "!" est similaire à "#"
\printtoks[mode=2]{\def\foo#1!2#3{\hbox to 1cm{\hss$!1^#2_!3$\hss}}}
```

produit les lignes suivantes dans le fichier log :

```
------ Begin token list ------
  \def catcode 16 (primitive)
  \foo catcode 16 (macro:#1->$#1$)
  # catcode 6  (macro parameter charcater)
  1 catcode 12 (character)
  ! catcode 6  (macro parameter charcater)
  2 catcode 12 (character)
  # catcode 6  (macro parameter charcater)
  3 catcode 12 (character)
  { catcode 1  (begin-group character)
  \hbox catcode 16 (primitive)
  t catcode 11 (letter)
  o catcode 11 (letter)
    catcode 10 (blank space)
  1 catcode 12 (character)
  c catcode 11 (letter)
  m catcode 11 (letter)
    { catcode 1  (begin-group character)
    \hss catcode 16 (primitive)
    $ catcode 3  (math shift character)
    ! catcode 6  (macro parameter charcater)
    1 catcode 12 (character)
    ^ catcode 7  (superscript character)
    # catcode 6  (macro parameter charcater)
    2 catcode 12 (character)
    _ catcode 8  (subscript character)
    ! catcode 6  (macro parameter charcater)
    3 catcode 12 (character)
    $ catcode 3  (math shift character)
    \hss catcode 16 (primitive)
    } catcode 2  (end-group character)
    } catcode 2  (end-group character)
  ------- End token list -------
```

2 Agir sur chaque token avec `\toksdo`

Cette macro parcourt tous les tokens un par un et indépendamment les uns des autres. L'utilisateur peut spécifier des critères à l'aide de ⟨*motifs*⟩ et pour chacun, via un ⟨*code*⟩ une action à effectuer sur le token ayant matché avec ⟨*motifs*⟩.

2.1 Motifs élémentaires

3 motifs élémentaires susceptibles de matcher avec un token sont disponibles, chacun étant accessible par le marqueur `\r`, `\R` et `\S`.

2.1.1 Le motif \r

La syntaxe de ce ⟨*1-motif*⟩ est

\r{⟨*csv d'intervalles de tokens*⟩:⟨*csv d'intervalles de catcodes*⟩}

où un intervalle est de la forme « plage s'étendant entre deux tokens » de type ⟨*token1*⟩-⟨*token2*⟩, mais peut être réduit à un unique token ⟨*token*⟩.

La liste ⟨*csv d'intervalles de catcodes*⟩ peut être remplacée par le caractère « * » qui tient lieu de joker et remplace tout intervalle. Si la ⟨*csv d'intervalles de catcode*⟩ est absente, elle est comprise comme « * ».

La virgule, le tiret et « : » ne peuvent pas être spécifiés comme token puisqu'ils font partie de la syntaxe (voir le marqueur \R pour contourner cette limitation).

À titre d'exemple :

\r{a-z:11}	correspond avec tout token de a à z ayant un catcode de 11.
\r{a-z} ou \r{a-z:*}	correspond avec tout token de a à z quel que soit son catcode.
\r{a,e,i,o,u,y}	correspond avec toute voyelle, quel que soit son catcode.
\r{a-z,A-Z,0-9:11,12}	correspond avec tout caractère alphanumérique (minuscule, majuscule ou chiffre), ayant indifféremment un catcode de 11 ou 12.

Il est important de noter qu'*aucun espace superflu* n'est toléré dans les syntaxes des ⟨*csv*⟩ :

- \r{␣a-z␣} provoque une erreur de compilation « ! Improper alphabetic constant », il faut écrire \r{a-z} ;
- \r{1,␣,2␣} est également fautif, il faut écrire \r{1,␣,2}.

2.1.2 Le motif \R

La syntaxe de ce ⟨*1-motif*⟩ est

\R{⟨*csv d'intervalles de charcodes*⟩:⟨*csv d'intervalles de catcodes*⟩}

Dans les deux listes csv, le caractère « * » tient lieu de joker et remplace tout intervalle. Les charcodes sont des nombres écrits en chiffres (base 10) ou à la façon de TEX : ‘\⟨*char*⟩. Ainsi, « ‘\: », « ‘\, » et « ‘\- » sont les entiers correspondant respectivement aux charcodes des tokens « : », « , » et « - ».

On a donc par exemple :

\R{*:10}	correspond avec tout token de catcode 10 (qui est un espace pour TEX)
\R{*:16}	correspond avec tout token de catcode 16 (qui est un caractère ou une séquence de contrôle pour TEX)
\R{106-115:11}	correspond avec tout token de « j » à « s » ayant un catcode de 11
\R{‘\a-‘\z}	est identique à \R{97-122} qui est équivalent à \r{a-z}
\R{‘\,,‘\;,‘\::12}	est identique à \R{44,59,58:12} et correspond avec , ou ; ou : de catcode 12
\R{*:*}	correspond avec n'importe quel token

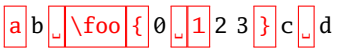
2.1.3 Le motif \S

La syntaxe de ce ⟨*1-motif*⟩ est

\S{⟨*tokens*⟩}

Un token correspond s'il figure dans ⟨*tokens*⟩ passés comme argument à \S.

Dans les tokens « ab \foo{0 123}c d », ceux qui matchent avec le motif « \S{1{a }\foo} » sont encadrés en rouge :



Les ⟨*tokens*⟩ peuvent contenir un ou plusieurs occurrences *non imbriquées* du marqueur \c. La syntaxe de ce marqueur est

`\c{`⟨*catcode*⟩`}{`⟨*tokens*⟩`}`

Dans l'argument de `\S`, l'action de `\c` est de donner aux ⟨*tokens*⟩ le ⟨*catcode*⟩ passé dansle permier argument. Si le ⟨*catcode*⟩ est réduit à 1 chiffre, les accolades sont facultatives. Il en est de même pour le 2e argument s'il contient un seul token.

Si un token est une ⟨*macro*⟩, le seul catcode qu'elle peut recevoir est 12 auquel cas, les tokens de catcode 12 provenant de `\string`⟨*macro*⟩ seront créés.

Les catcodes acceptés dans le 1er argument de`\c` sont : 1, 2, 3, 4, 6, 7, 8, 10, 11, 12 et 13. L'utilisateur devra être *extrêmement attentif* aux conséquences d'un changement de catcode vers les catégories sensibles portant les numéros 1, 2 et 6.

Si l'on écrit

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

alors les ⟨*tokens*⟩ contenus dans l'argument de `\S` sont :

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

2.2 Motifs

Ce que l'on note ⟨*motifs*⟩ pour `\toksdo` est constitué d'un ⟨*1–motif*⟩ élémentaire ou de plusieurs séparés par « | » qui signifie « ou » :

`\r{a-z:11} | \R{*:12}`	correspond avec tout token étant une lettre de a à z de catcode 11 ou tout token de catcode 12
`\R{*:16} | S{01223456789.}`	correspond avec tout token étant une macro, un chiffre ou un point
`\r{0-9:12} | \r{a-z,A-Z:11} | \R{*:10}`	correspond avec tout token ayant le catcode par défaut étant un chiffre, une lettre, ou un espace

2.3 Utiliser `\toksdo`

Cette macro s'utilise de la façon suivante :

```
\toksdo[⟨clés⟩=⟨valeurs⟩]
      {
      ⟨motifs1⟩ -> ⟨code1⟩,
      ⟨motifs2⟩ -> ⟨code2⟩,
      etc.
      ⟨motifsn⟩ -> ⟨coden⟩
      }{⟨tokens⟩}
```

On peut spécifier autant de ⟨*motifs*⟩ et de ⟨*codes*⟩ associés que l'on souhaite.

Chaque ⟨*code*⟩ est un code arbitraire [2] susceptible de modifier le token qui a matché avec ⟨*motifs*⟩. Dans ce ⟨*code*⟩, on peut utiliser les macros suivantes :

- `\tokslen` qui est le nombre total de tokens ;
- `\selfindex`, `\selfcharcode` et `\selfcatcode` qui, pour le token ayant matché, représentent son index (qui commence à 1 et finit à `\tokslen`), son charcode et son catcode. Si le token est une macro, `\selfcharcode` n'est *pas* un nombre et vaut la macro elle-même ;
- `\addtok{`⟨*code*⟩`}` qui indique de quelle façon ajouter le token ayant matché au collecteur interne qui le recueille tous en vue des les afficher ou de les assigner en fin de processus ;
- `\deltok` supprime le token ayant matché ;
- `\setcharcode{`⟨*expression arithmétique*⟩`}` et `\setcatcode{`⟨*expression arithmétique*⟩`}` changent le charcode et le catcode du token ayant matché. Une ⟨*macro*⟩ ne peut recevoir que le catcode 12 et dans ce cas, les tokens provenant de `\string`⟨*macro*⟩ sont créés. Pour tout autre token, les catcodes acceptés par `\setcatcode` sont 1, 2, 3, 4, 6, 7, 8, 10, 11, 12 et 13. Une attention particulière doit être portée à ce que l'on fait si on change un catcode pour 1, 2 ou 6.

Les ⟨*clés*⟩ acceptées par la macro `\toksdo` sont :

2. Si ce ⟨*code*⟩ contient une virgule qui n'est pas entre accolades, il faut utiliser la syntaxe ⟨*motifs*⟩`->{`⟨*code*⟩`}`.

Clé	Valeur par défaut	Description
`expand arg`	`0`	nombre de développements que doit subir l'argument contenant les ⟨*tokens*⟩ avant d'être pris en compte
`collect`	`true`	collecte tous les tokens retenus dans le but de les afficher ou les stocker. Aucune collecte n'est faite si ce booléen est `false`
`assign`	⟨*vide*⟩	consigne d'assignation : une consigne d'assignation est tout code susceptible de se trouver devant le résultat mis entre accolades et le code qui est exécuté est donc ⟨*consigne*⟩{⟨*résultat*⟩}. Si ⟨*vide*⟩, affiche le résultat. Si la valeur est une ⟨*macro*⟩, cette ⟨*macro*⟩ doit être un registre de tokens qui recevra le résultat. La valeur peut également être de la forme `\def`⟨*macro*⟩ où la ⟨*macro*⟩ recevra le résultat

Voici comment programmer le ROT13 :

```
\toksdo{
        \r{a-m,A-M} -> \setcharcode{\selfcharcode + 13},
        \r{n-z,N-Z} -> \setcharcode{\selfcharcode - 13}
        }{Deux plus deux font quatre}
```

Qrhk cyhf qrhk sbag dhnger

Dans cet exemple, on remplace tout chiffre ou tiret par « x » et tout espace par « _ » de catcode 12 :

```
\toksdo{
        \r{0-9} | \S{-} -> \setcharcode{`x},
        \R{*:10}        -> \setcharcode{`\_}\setcatcode{12}
        }{Votre mot de passe est : \textbf{758-457-384}, ne n'oubliez pas !}
```

Votre_mot_de_passe_est_:_**xxxxxxxxxxx**,_ne_n'oubliez_pas_!

Ne garde que la première moitié des tokens :

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

123ab

Il est parfois nécessaire d'assigner le résultat à un registre de token au cas où les tokens obtenus contiennent le token de catcode 6 (# par défaut) :

```
\newtoks\myresult
\toksdo[assign=\myresult]
        {\S{2} -> \deltok }% supprime les "2"
        {\def\foo#1{02#120}}
\the\myresult% exécute : \def\foo#1{0#10}
\foo{A}, \foo{xYz}
```

0A0, 0xYz0

Si l'on avant écrit « `assign = \def\mymacro` », le code qui aurait été exécuté en coulisses est

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

Ce code est illégal en TEX (erreur du type « `Illegal parameter number in definition of \mymacro` »).

La clé `collect`, lorsque `false` indique de ne pas collecter les tokens. Cela peut faire sens si le code sur les tokens ayant matché ne les modifie pas. Voici comment compter les voyelles à l'aide d'un compteur :

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

<4>

L'argument de `\setcharcode` et de `\setcatcode` est évalué avec la primitive `\numexpr` et donc, toute expression arithmétique acceptée par cette primitive est correcte. Ici, toutes les voyelles sont mises en majuscule :

```
\toksdo{ \S{aeiouy} -> \setcharcode{\selfcharcode + `A - `a} }{Deux plus deux font quatre}
```

DEUx plUs dEUx fOnt qUAtrE

Voici comment, en 2 passes, extraire les tokens se trouvant dans le plus haut niveau d'imbrication d'accolades :

```
\newcount\nestcnt \nestcnt=0 % compte le niveau d'imbrication
\newcount\maxnestcnt \maxnestcnt=0 % est le plus haut niveau d'imbrication
\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}Imbrication max = \the\maxnestcnt\par
Tokens les plus imbriqués :
\toksdo[expand arg=1]{
        \R{*:1} -> \advance\nestcnt1  \deltok,% supprimer accolades ouvrantes
        \R{*:2} -> \advance\nestcnt-1 \deltok,% supprimer accolades fermantes
        \R{*:*} -> \ifnum\nestcnt<\maxnestcnt \deltok \fi% supprimer tout sauf au plus haut niveau d'imbrication
        }{\mycode}
```

Imbrication max = 3
Tokens les plus imbriqués : 67e

2.4 Utiliser \toksdo et \addtok

Lorsque la clé `collect` est `true`, les tokens sont collectés. Une fois qu'ils ont été traités et éventuellement modifié par `\setcharcode` ou `\setcatcode`, chacun est ajouté tel quel au collecteur interne qui est sollicité en fin de processus pour afficher les tokens ou les assigner si la clé `assign` le spécifie.

Il est possible de modifier la façon dont sont ajoutés les tokens au collecteur interne via la macro `\addtok{`⟨*code*⟩`}`, où ⟨*code*⟩ est un code arbitraire dans lequel `\self` représente le token lui-même.

Par défaut et au début de chaque code qui suit « ⟨*motifs*⟩ `->` », la macro `\addtok` est initialisée à sa valeur par défaut :

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

ce qui signifie que chaque token doit être ajouté tel quel.

Si l'on écrit `\addtok{\self\self}`, les tokens ayant matché sont doublés. Avec `\addtok{\fbox{\self}}`, ils sont encadrés à l'aide de la macro `\fbox` de LaTeX.

La macro `\deltok` est équivalente à `\addtok{}`.

Dans la majorité des cas, `\self` représente un seul token : celui qui est en cours de traitement. Ce n'est *que* lorsqu'une macro est détokénisée via `\setcatcode{12}` que `\self` représente plusieurs tokens.

Dans cet exemple, chaque macro est détokénisée et encadrée, et tous les espaces sont remplacés par « ␣␣ » :

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

a _ b \foo 12. _ 3 \baz - _ 9

Chaque chiffre 1 est doublé et chaque 0 est remplacé par un espace :

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

11 1111 11 11

Les macros `\selfindex`, `\selfcharcode` et `\selfcatcode` ne doivent *pas* être utilisées dans l'argument de `\addtok`, car elles seront ajoutées telles quelles dans le collecteur interne. Ce n'est qu'au moment de l'exécution et après avoir collecté le dernier token qu'elles seront développées, et contiendront à ce moment les données du dernier token. Si l'on souhaite effectuer des tests sur l'index, le charcode ou le catcode du token à ajouter, il faut le faire à l'extérieur de l'argument de `\addtok`. Voici le code source dont le résultat est visible en page 5 :

```
Dans les tokens «\verb|ab \foo{0 123}c d|», ceux qui matchent avec
le motif «\verb|\S{1{a }\foo}|» sont encadrés en rouge :\par
\hfill
\fboxsep=1.5pt
\toksdo{ \S{1{a }\foo} -> \ifnum\selfcatcode=10 % si espace
                             \addtok{{\color{red}\fbox{\strut\textvisiblespace}}}%
                         \else% si pas espace
                             \setcatcode{12}% détokéniser
                             \addtok{{\ttfamily\color{red}\fbox{\strut\self}}}% encadrer en rouge
                         \fi,
        \R{*:*}      -> \addtok{\,{\ttfamily\self}\,}% pour tout autre token, espacer un petit peu
      }{ab \foo{0 123}c d}%
\hfill\null
```

Dans les tokens « ab \foo{0 123}c d », ceux qui matchent avec le motif « \S{1{a }\foo} » sont encadrés en rouge :
a b ␣ \foo { 0 ␣ 1 2 3 } c ␣ d

3 Compter les tokens avec \tokscount

La macro

\tokscount[⟨*clés*⟩=⟨*valeurs*⟩]{⟨*motifs*⟩}{⟨*tokens*⟩}

compte combien de tokens matchent avec ⟨*motifs*⟩ dans les ⟨*tokens*⟩. Si aucun motif n'est spécifié (argument vide), la macro compte tous les tokens.

Les ⟨*clés*⟩ disponibles sont les suivantes :

Clé	Valeur par défaut	Description
`expand arg`	`0`	nombre de développements que doit subir l'argument contenant les ⟨*tokens*⟩ avant d'être pris en compte
`assign`	⟨*vide*⟩	consigne d 'assignation. Si ⟨*vide*⟩, affiche le nombre de tokens ayant matché. Sinon, doit contenir une consigne d'assignation du type \def⟨*macro*⟩ pour assigner à la ⟨*macro*⟩ le résultat
`assign match`	⟨*vide*⟩	consigne d'assignation. Si ⟨*vide*⟩ les tokens ayant matché ne sont pas collectés. Sinon, doit contenir une consigne d'assignation \def⟨*macro*⟩ pour les assigner à une macro ou ⟨*macro*⟩ pour les assigner à un registre de tokens

Les ⟨*motifs*⟩ acceptés par \tokscount sont les mêmes que pour \toksdo.

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

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

4 Grammaires PEG

Dans un ensemble de tokens, certains peuvent matcher avec des motifs que l'on peut organiser en grammaire de type PEG (Parsing Expresion Grammar). À chaque fois qu'une correspondance a lieu avec un motif, les tokens ayant matché sont consommés et ne sont plus disponibles pour les motifs suivants (sauf pour les prédicats, voir plus bas). S'il n'y a pas de correspondance, aucun token n'est consommé.

4.1 Les 5 marqueurs de motifs

Le package tokstools met à disposition 5 motifs élémentaires définis par 5 marqueurs pour construire des motifs plus complexes :

— le motif \r{⟨*csv intervalles de caracatères*⟩:⟨*csv intervalles de catcode*⟩} décrit à la page 5 ;
— le motif \R{⟨*csv intervalles de charcodes*⟩:⟨*csv intervalles de catcode*⟩} décrit à la page 5 ;
— le motif \S{⟨*tokens*⟩} décrit à la page 5 ;
— le motif \s{⟨*tokens*⟩} : tout ensemble de tokens exactement constitué des ⟨*tokens*⟩ passé en argument de \s matche avec ce motif : il s'agit donc d'une correspondance exacte entre chaines de tokens.

  Tout comme avec le marqueur \S, il est possible de changer les catcodes de certains tokens à l'aide du marqueur \c selon la syntaxe \c{⟨*catcode*⟩}{⟨*tokens*⟩} ;
— le motif \. qui matche avec tout token et qui est équivalent à \R{*:*}.

4.2 Répétitions

Chaque ⟨*1-motif*⟩ peut être suivi d'un indicateur de répétition si l'on souhaite spécifier le nombre de fois que la correspondance se produit. Les marqueurs de répétition sont :
— ^⟨*chiffre*⟩ ou ^{⟨*nombre*⟩} : demande à ce que la correspondance se répète exactement le nombre de fois passé en argument de ^ ;
— ^{⟨*min*⟩-⟨*max*⟩} : demande à ce que la correspondance se répète entre ⟨*min*⟩ et ⟨*max*⟩. Si ⟨*min*⟩ est absent, il est compris comme 0. Si ⟨*max*⟩ est absent, il n'y a aucune limite sur nombre de répétition maximal. Les deux nombres ne peuvent pas être absents en même temps.
— « + » : 1 ou plusieurs répétitions (est équivalent à ^{1-}) ;
— « * » : 0 ou plusieurs répétitions (est équivalent à ^{0-}) ;
— « ? » : 0 ou 1 répétition (est équivalent à ^{0-1}) ;

Si aucun marqueur de répétition n'est présent après un ⟨*1-motif*⟩, la consigne ^1 est implicitement passée.

Il est important de noter que *dans tous les cas*, le plus grand nombre possible de répétitions est consommé ; c'est un comportement qui est toujours « gourmand ».

4.3 Prédicats

Chaque ⟨*1-motif*⟩ peut être précédé du caractère « ! » ou « & » qui en fait un prédicat :
— le prédicat !⟨*1-motif*⟩ matche s'il n'y a pas correspondance avec le ⟨*1-motif*⟩
— le prédicat &⟨*1-motif*⟩ matche s'il y a correspondance avec le ⟨*1-motif*⟩.

La règle dans les grammaires PEG est qu'*un prédicat ne consomme jamais de tokens* : on peut donc comprendre un prédicat comme un test sur ce qui va suivre sans bouger de position.

4.4 Concaténation de motifs

Le caractère « : » entre 2 ⟨*1-motif*⟩ indique de matcher avec le premier *et* le second. On n'est évidemment pas limité à 2 ⟨*1-motif*⟩, il peut y en avoir autant que l'on veut.

`&\r{0-9}^{3-} : \r{0-9}^2`	s'il y a au moins 3 chiffres, matche avec les 2 premiers
`\r{a-z}+ : \R{*:10}? : \S{01}^2`	matche avec une ou plusieurs lettres minuscules suivies d'un espace optionnel suivi de 2 chiffres binaires
`\r{1-9}+ : \s{00} : !\.`	matche avec un nombre constitué d'au moins 1 chiffre non nul suivi de "00" qui doivent être les *derniers tokens*, car le prédicat « !\. » signifie « ne doit pas être suivi d'un token »

4.5 Choix entre motifs

Le caractère « | » entre 2 ⟨*1-motif*⟩ indique de matcher avec le premier *ou* avec le second. On n'est évidemment pas limité à 2 motifs, il peut y en avoir autant que l'on veut. Si une correspondance a lieu pour un ⟨*1-motif*⟩, elle est retenue et aucun autre ⟨*1-motif*⟩ qui suit ne sera testé : il s'agit donc d'un choix ordonné. Si l'on écrit

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

le 2e test ne sera jamais effectué puisqu'il est inclus dans le premier. Il est donc important de commencer par les tests les plus spécifiques et de terminer par les plus généraux si les portées des motifs se chevauchent ou pire sont imbriquées.

L'opérateur de concaténation « : » est prioritaire sur celui du choix « | » et donc

⟨*a*⟩ : ⟨*b*⟩ | ⟨*c*⟩ | ⟨*d*⟩ : ⟨*e*⟩

se comprend comme ⟨*a*⟩:⟨*b*⟩ ou ⟨*c*⟩ ou ⟨*d*⟩:⟨*e*⟩.

4.6 Groupement de motifs

Tout ⟨*1-motif*⟩ ou combinaison de ⟨*1-motif*⟩ notée ⟨*motifs*⟩, peut être mis entre accolades pour devenir un nouveau ⟨*1-motif*⟩ qui peut, à son tour, être précédé d'un prédicat et suivi d'une spécification de répétitions selon la syntaxe déjà décrite :

⟨*prédicat*⟩{⟨*motifs*⟩}⟨*répétitions*⟩

4.7 Espaces

Dans la syntaxe des motifs, les espaces sont ignorés.

Ainsi

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

est équivalent à

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

4.8 Priorités

Les priorités entre opérateurs sur les ⟨*1-motif*⟩ sont les suivantes (plus le numéro est élevé, plus la priorité l'est) :

Opérateur	Priorité	Description
{⟨*motifs*⟩}	5	groupement de motifs qui devient un ⟨*1-motif*⟩
⟨*1-motif*⟩?	4	matche 0 ou 1 fois
⟨*1-motif*⟩*	4	matche 0 fois ou plus
⟨*1-motif*⟩+	4	matche 1 fois ou plus
⟨*1-motif*⟩^{n}	4	matche n fois
⟨*1-motif*⟩^{a-b}	4	matche entre a et b fois
!⟨*1-motif*⟩	3	matche si ⟨*1-motif*⟩ ne matche pas, sans consommer de token
&⟨*1-motif*⟩	3	matche si ⟨*1-motif*⟩ matche, sans consommer de token
⟨*1-motif*$_1$⟩:⟨*1-motif*$_2$⟩	2	matche avec ⟨*1-motif*$_1$⟩ puis avec ⟨*1-motif*$_2$⟩
⟨*1-motif*$_1$⟩|⟨*1-motif*$_2$⟩	1	matche avec ⟨*1-motif*$_1$⟩ ou ⟨*1-motif*$_2$⟩ (choix ordonné)

4.9 Nom des motifs

Tout motif peut être défini et nommé pour réutiliser ce motif ultérieurement avec une syntaxe plus légère et facile à retenir :

\defpattern⟨*nom de motif*⟩{⟨*motifs*⟩}

Le ⟨*nom de motif*⟩ *doit* être une séquence de contrôle. Cette séquence de contrôle n'est pas définie ou redéfinie par \defpattern, c'est également un marqueur dont la signification n'a pas d'importance.

5 Tester une correspondance avec \ifpegmatch

5.1 Syntaxe

La macro \ifpegmatch a la syntaxe suivante

\ifpegmatch[⟨*clés*⟩=⟨*valeurs*⟩]{⟨*motifs*⟩}{⟨*tokens*⟩}{⟨*code si match*⟩}{⟨*code si non match*⟩}

Lorsque toutes les valeurs sont par défaut, cette macro teste si les ⟨*motifs*⟩ font matcher des tokens se trouvant en début des ⟨*tokens*⟩.

Les ⟨*clés*⟩ disponibles sont :

Clé	Valeur par défaut	Description
`expand arg`	`0`	nombre de développements que doit subir l'argument contenant les ⟨*tokens*⟩ avant d'être pris en compte
`mode`	`1`	mode de recherche de correspondances
`capture name`	⟨*vide*⟩	nom des captures
`assign prematch`	`\def\prematchtoks`	consigne d'assignation pour les tokens précédant ceux qui ont matché
`assign match`	`\def\matchtoks`	consigne d'assignation pour les tokens ayant matché
`assign postmatch`	`\def\remaintoks`	consigne d'assignation pour les tokens précédant ceux qui ont matché

Ici, on teste si les ⟨*tokens*⟩ commencent par 2 chiffres, 1 espace optionnel et au moins 1 lettre minuscule

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

1) V 2) V 3) F 4) F

Construisons une grammaire simple pour tester si un argument commence une opération arithmétique du type

⟨*entier relatif*⟩⟨*opération*⟩⟨*entier positif*⟩

```
\defpattern\sp{ \R{*:10} } % espace
\defpattern\digit{ \r{0-9} }% chiffre
\defpattern\posint{ \digit+ }% entier positif
\defpattern\int{ \S{+-}? : \posint }% entier relatif
\defpattern\op{ \S{+-*/} }% opération math
\defpattern\okmatch{ \sp* : \int : \sp* : \op : \sp* : \posint : \sp* }
1) \ifpegmatch\okmatch{2*3}{V}{F}\quad
2) \ifpegmatch\okmatch{-7 + 1 }{V}{F}\quad
3) \ifpegmatch\okmatch{ a + 9 }{V}{F}\quad
4) \ifpegmatch\okmatch{ -2 / 6 + 3 }{V}{F}\quad
5) \ifpegmatch\okmatch{ +2026- 4068}{V}{F}\quad
6) \ifpegmatch\okmatch{ -3 }{V}{F}\quad
7) \ifpegmatch\okmatch{2a-3b}{V}{F}\par
La primitive \verb|\int| n'est pas redéfinie : $\int x^2dx$
```

1) V 2) V 3) F 4) V 5) V 6) F 7) F
La primitive \int n'est pas redéfinie : $\int x^2 dx$

Avec le prédicat « `: !\.` » rajouté à la fin de \okmatch, la ligne 4 aurait affiché « F » car il reste « +␣3␣ » après les tokens ayant matché.

5.2 Mode

La macro \ifpegmatch peut chercher des correspondances selon plusieurs modes, spécifiés via la clé « mode » :

- par défaut, `mode=1` indique que le match doit se produire au début des ⟨*tokens*⟩ ;
- `mode=0` indique que tous les ⟨*tokens*⟩ doivent matcher ; c'est le mode le plus strict ;
- `mode=2` indique que le match peut se produire n'importe où dans les ⟨*tokens*⟩ ; c'est le mode le moins contraignant.

5.3 Délivrance de tokens

La macro \ifpegmatch renvoie également quelques informations :

- par défaut, la macro \matchtoks contient les tokens qui ont matché et elle est vide si aucune correspondance n'a été trouvée ;
- par défaut, la macro \remaintoks contient les tokens qui restent après ceux qui ont matché ;
- par défaut, la macro \prematchtoks contient les tokens qui précèdent ceux qui ont matché (ne peut contenir des tokens uniquement lorsque `mode=2`) ;
- la macro \matchposition contient la position du premier token ayant matché et 0 si aucun match ne s'est produit.

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

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

5.4 Captures

Le marqueur \c, lorsqu'il est placé devant un ⟨*1-motif*⟩, indique que les tokens qui matchent avec ce ⟨*1-motif*⟩ doivent être capturés ainsi que leur position. Si un ⟨*1-motif*⟩ est un prédicat, aucune capture n'est faite. Les captures sont rangée dans l'ordre chronologique dans lequel elles ont été faites et sont délivrées, en 2 développements, par \tokscapture{⟨*index*⟩} ou par \tokscapture{⟨*nom*⟩:⟨*index*⟩} . Le ⟨*nom*⟩ est optionnel se règle avec la clé name et l'⟨*index*⟩ est le numéro d'ordre de la capture.

Si l'⟨*index*⟩ vaut 0, toutes les captures sont mises entre accolades et listées dans une csv de la forme :

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

De la même façon, la macro \poscapture{⟨*index*⟩} ou \poscapture{⟨*nom*⟩:⟨*index*⟩} rend les positions de captures à la différence que si ⟨*index*⟩ vaut 0, les positions sont mises dans une csv *sans être mises entre accolades* :

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

Si un ⟨*index*⟩ dépasse l'index maximal, un message d'erreur est émis.

Si \c se trouve après un ⟨*1-motif*⟩, seule la position est capturée.

Dans cet exemple, 2 captures complètes (tokens+position) et une capture de position sont faites :

```
\ifpegmatch[mode=2]{ \c\r{a-z}+ : \c\r{0-9}^2\c }{12abc666def}{V}{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}>
```

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

Dans cet exemple, on définit une grammaire qui fait matcher une écriture scientifique de la forme $a \times 10^b$ et qui capture les deux nombres a et 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 Grammaires récursives

Le package `tokstools` admet des grammaires récursives, mais aucune optimisation n'est faite et la façon de les traiter reste naïve. Par conséquent, certaines grammaires récursives, en particulier celles conduisant à des « récursivités à gauche » vont mener à des boucles infinies.

Cela étant, il est possible d'utiliser des grammaires récursives, mais y mettre des captures tend à être un peu aléatoire car l'ordre de ces captures n'est pas évident et dépend de la définition même de cette grammaire et donc du parcours de l'arbre qui en découle.

Voici une grammaire récursive capturant une expression arithmétique comportant les 4 opérations avec parenthèses :

```
\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}{V}{F}\quad
2) \ifpegmatch[mode=0]\expr{1-2*3}{V}{F}\quad
3) \ifpegmatch[mode=0]\expr{3*4+6}{V}{F}\quad
4) \ifpegmatch[mode=0]\expr{3-(1-2*3)}{V}{F}\quad
5) \ifpegmatch[mode=0]\expr{3*4*(1-3*2-(1-3/7)*3)/(1/7+2*3)*3-5}{V}{F}\quad
6) \ifpegmatch[mode=0]\expr{6-9*(2-3)+4/5}{V}{F}
```

1) V 2) V 3) V 4) V 5) V 6) V

Cette grammaire s'assure que l'expression est commence par une parenthèse, contient une expression équilibrée en parenthèses :

```
\defpattern\nobrtext{ { !\S{()} : \. }+ }
\defpattern\inparen{ \nobrtext : \inparen* | \s{(} : \inparen* : \s{)} }
\defpattern\expr{ &\s{(} : \inparen : !\. }% prédicats -> doit commencer par '(' et finir par ')'
1) \ifpegmatch\expr{a(b)c}{V}{F}\quad
2) \ifpegmatch\expr{(a(abc)()d)}{V}{F}\quad
3) \ifpegmatch\expr{(a(bc))df)}{V}{F}\quad
4) \ifpegmatch\expr{((abc)d((e)f)g)}{V}{F}\quad
5) \ifpegmatch\expr{((foo)b(((b)a)r)}{V}{F}
```

1) F 2) V 3) F 4) V 5) F

6 Compter les correspondances avec \pegcount

La macro

\pegcount[⟨*clés*⟩=⟨*valeurs*⟩]{⟨*motifs*⟩}{⟨*tokens*⟩}

compte combien de fois les ⟨*motifs*⟩ matchent dans les ⟨*tokens*⟩. Chaque position est sauvegardée et chaque correspondance est capturée pour être facilement restituée.

Aucune capture explicitement demandée par \c n'est permise et est ignorée.

Les ⟨*clés*⟩ disponibles sont :

Clé	Valeur par défaut	Description
expand arg	0	nombre de développements que doit subir l'argument contenant les ⟨*tokens*⟩ avant d'être pris en compte.
assign	⟨*vide*⟩	si vide, affiche le nombre de correspondances. Sinon, doit contenir une consigne d'assignation du type \def⟨*macro*⟩ pour assigne à la ⟨*macro*⟩ le résultat.
assign positions	\def\matchposlist	consigne d'assignation de la liste des positions. Si ⟨*vide*⟩ aucune capture n'est effectuée. Sinon, doit contenir une consigne d'assignation \def⟨*macro*⟩ pour assigner la liste des positions à une ⟨*macro*⟩
name	⟨*vide*⟩	est le nom des captures qui sont restituées par \tokscapture

Les captures sont délivrées, en 2 développements, par \tokscapture{[⟨*nom*⟩:]⟨*index*⟩}. Le ⟨*nom*⟩ est optionnel se règle avec la clé name et l'⟨*index*⟩ est le numéro d'ordre de la capture. Si l'⟨*index*⟩ vaut 0, toutes les captures sont mises entre accolades et listées dans une csv de la forme :

{⟨$capture_1$⟩},{⟨$capture_2$⟩},...,{⟨$capture_n$⟩}

```
1) \pegcount{ \r{0-9}+ : \r{a-z}+ }{foo25bar}, <\matchposlist>\par% chiffres suivis de lettres
2) \pegcount{ \r{0-9}+ : \r{a-z}+ }{a12bcd,4b,z875bar}, <\matchposlist>,
"\tokscapture{0}", 1="\tokscapture{1}", 2="\tokscapture{2}", 3="\tokscapture{3}"\par% chiffres suivis de lettres
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, <1,6,17>, 1="+ab3+", 2="+bb6ab8ca7+", 3="+aa1bb2+"

7 Remplacements avec \pegreplace

La macro

\pegreplace[⟨*clés*⟩=⟨*valeurs*⟩]
{
⟨$motifs_1$⟩ -> ⟨$remplacement_1$⟩,
⟨$motifs_2$⟩ -> ⟨$remplacement_2$⟩,
etc.
⟨$motifs_n$⟩ -> ⟨$remplacement_n$⟩
}{⟨*tokens*⟩}

cherche tous les ⟨*motifs*⟩ dans les ⟨*tokens*⟩ et pour chaque correspondance, remplace les tokens ayant matché par le code défini dans ⟨*remplacement*⟩.

Les ⟨*clés*⟩ disponibles sont :

Clé	Valeur par défaut	Description
`expand arg`	`0`	nombre de développements que doit subir l'argument contenant les ⟨*tokens*⟩ avant d'être pris en compte.
`mode`	`2`	définit selon quel mode `\pegreplace` doit chercher les correspondances
`assign`	⟨*vide*⟩	si vide, affiche le ⟨*tokens*⟩ obtenus après les avoir fait les remplacements. Sinon, doit contenir une consigne d'assignation pour assigner le résultat à une macro avec `\def`⟨*macro*⟩ ou à un registre de tokens.

Il faut comprendre toutes les positions dans ⟨*tokens*⟩ sont envisagées, de la position 1 à la position L, si L est le nombre de tokens. Pour chaque position, tous les ⟨*motifs*⟩ sont successivement testés pour savoir si une correspondance se produit à cette position. Si un ⟨$motif_i$⟩ matche à la position k, ⟨$remplacement_i$⟩ correspondant est effectué et `\pegreplace` est prêt à passer à la position $k+1$. La clé `mode` doit être un entier compris entre 0 et 2 qui spécifie comment doit se comporter la macro `\pegreplace` après que la première correspondance ait été rencontrée et le premier remplacement fait :

- `mode=0` indique qu'après la première correspondance à la position k, aucune autre position ne sera examinée, plus aucune action n'est effectuée et la macro `\pegreplace` a terminé son travail ;
- `mode=1` indique que toutes les positions suivantes seront examinées, mais si un ⟨$motifs_k$⟩ matche, il est ensuite neutralisé et ne pourra plus matcher par la suite, autrement dit, chaque ⟨*motifs*⟩ ne peut matcher qu'une fois au maximum ;
- `mode=2`, qui est la valeur par défaut, indique que toutes les positions suivantes seront examinées et que chaque ⟨*motifs*⟩ peut matcher autant de fois que nécessaire.

Le ⟨*remplacement*⟩ est un code arbitraire ne contenant pas de virgule où peut figurer « `\0` » qui signifie « tokens qui ont matché », « `\1` » est la première capture faite par `\c`, « `\2` » est la deuxième et ainsi de suite jusqu'à « `\9` ». Il faut noter que les espaces avant et après ⟨*remplacement*⟩ sont supprimés. Si un ⟨*remplacement*⟩ est susceptible de contenir une virgule ou si l'on souhaite préserver les espaces avant ou après lui, il faut mettre le tout entre accolades.

Par exemple, pour transformer chaque mot en list csv et augmenter les espaces, il faudrait écrire

```
\pegreplace{
        \r{a-z,A-Z} : &\. : !\R{*:10} -> {\0, } ,% accolades nécessaires ici
        \R{*:10}                       -> \quad
        }{Happy TeXing}
```

H, a, p, p, y T, e, X, i, n, g

Pour illustrer le comportement de `\pegreplace` selon la valeur de la clé `mode`, dans cet exemple, on encadre toute suite de 2 lettres minuscules et on met entre chevrons tout chiffre de 1 à 5 :

```
\fboxsep=1pt
\pegreplace[mode=0]{
        \r{a-z}^2 -> \fbox{\strut\0},
        \S{12345} -> $\langle\0\rangle$
        }{6foob1baz327z}

\pegreplace[mode=1]{
        \r{a-z}^2 -> \fbox{\strut\0},
        \S{12345} -> $\langle\0\rangle$
        }{6foob1baz327z}

\pegreplace[mode=2]{
        \r{a-z}^2 -> \fbox{\strut\0},
        \S{12345} -> $\langle\0\rangle$
        }{6foob1baz327z}
```

6 fo ob1baz327z
6 fo ob⟨1⟩baz327z
6 fo ob ⟨1⟩ ba z⟨3⟩⟨2⟩7z

Encadrement de `\alpha` et `\beta` avec leur coefficient en mode mathématique :

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

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

On créé une grammaire qui matche avec le code postal et le nom d'une ville (composé de mots et de tirets).

```
\defpattern\sp{ \R{*:10} }
\defpattern\CP{\c\r{0-9}^2 : \sp? : \c\r{0-9}^3 }% \1=2 premiers chiffres  \2=3 derniers
\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 }% capturer CP et capturer Ville
1) \pegreplace{\CPville -> CodePostal=\textbf{\1\2} est \fbox{\3} }{Destination 75000 Paris}\par
2) \pegreplace{\CPville -> CodePostal=\textbf{\1\2} est \fbox{\3} }{Destination 64 500 Saint-Jean-de-Luz suite}\par
3) \pegreplace{\CPville -> CodePostal=\textbf{\1\2} est \fbox{\3} }{Destination 38120 Saint-Égrève}
```

1) Destination CodePostal=**75000** est Paris
2) Destination CodePostal=**64500** est Saint-Jean-de-Luz suite
3) Destination CodePostal=**38120** est Saint-Égrève

Crée une petite devinette concernant des nombres ayant de 2 chiffres non nuls :

```
\defpattern\num{ \c\r{1-9} : \c\r{1-9} : !\r{0-9} }
Si \pegreplace{\num -> \1\2 donne \the\numexpr\1*\2+\2\relax }{27, 34 et 43}, que donne 57 ? (Répondez dans 7,5~millions
    d'années)
```

Si 27 donne 21, 34 donne 16 et 43 donne 15, que donne 57 ? (Répondez dans 7,5 millions d'années)

8 Erreurs

Parmi les nombreuses erreurs qui peuvent se produire, en voici quelques unes…

8.1 Accolades non équilibrées

Si un résultat ou une capture est constitué de tokens dont les accolades ne sont pas équilibrées, une erreur de compilation se produira.

Par exemple, si à l'aide de `\toksdo`, on supprime les accolades fermantes :

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

l'erreur émise par `tokstools` est « `! Unbalanced open-group token, 1 close-group token added` ».

Pour équilibrer les accolades, une accolade fermante est donc rajoutée à la fin des tokens passés en argument et les tokens « `foo{\bfseries123bar}` » sont dirigés vers l'affichage.

De la même façon, si on supprime les accolades ouvrantes :

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

l'erreur émise est « `! Unbalanced close-group token ignored` ».

L'accolade fermante en sur-nombre est donc ignorée et les tokens « `foo\bfseries123bar` » sont dirigés vers l'affichage

8.2 Moteur non adapté

L'utilisation d'un moteur 8 bits *impose* de n'utiliser que des tokens, notamment pour le marqueur `\r`.

Compiler ce code avec un moteur 8 bits

```
\ifpegmatch{ \r{a,e,i,o,u,y,é,à,ê,ù} }{Un été à l'océan}{V}{F}
```

provoque une erreur de compilation car « é » est constitué de 2 tokens (charcodes 195 et 169, catcodes égaux à 13) alors que la syntaxe de `\r` impose qu'il n'y en ait qu'un. L'erreur de compilation émaise par `tokstools` est « `! Multiple token 'é', '0' inserted` ».

La syntaxe correcte serait

```
\ifpegmatch{ \r{a,e,i,o,u,y} | \S{éàêù} }{Un été à l'océan}{V}{F}
```

8.3 Intervalle incorrect

Les intervalles de la forme ⟨*a*⟩–⟨*b*⟩, que a et b soit des tokens ou des nombres *doivent* être mis dans l'odre.

Comme le token « ń » vient *après* « ó » en UTF8, le code

```
\ifpegmatch{ \r{ń–ó} }{Un texte quelconque}{V}{F}
```

provoque l'erreur de compilation « `! Unsorted interval 'ń–ó', 'ó–ń' inserted` ». L'intervalle est corrigé par `tokstools`.

8.4 Syntaxe de motif non respectée

Espaces mis à part car ils sont ignorés, toute syntaxe de motif non conforme à ce qui a été décrit à partir de la page 9 provoque une erreur de compilation.

Par exemple

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

provoque l'erreur « `! Found "S" when expecting \r, \R, \s, \S or \.` ».

8.5 Erreurs de capture

Si un ⟨*nom*⟩ n'a pas été défini ou si un ⟨*index*⟩ est demandé hors de ceux assignés lors des captures, une erreur de compilation est renvoyée.

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

crée 3 captures et donc demander

```
\tokscapture{4}
```

envoie l'erreur de compilation « `! Undefined token capture at index "4"` ».

9 Listes des macros et marqueurs

9.1 Liste des commandes

Voici les macros mises à disposition de l'utilisateur :

- `\printtoks`, voir page 3 ;
- `\toksdo`, `\setcharcode`, `\setcatcode`, `\deltok`, `\addtok`, `\selfcharcode`, `\selfcatcode` et `\selfindex`, voir pages 6 et suivantes ;
- `\tokscount`, voir page 9 ;
- `\defpattern`, voir page 11 ;
- `\ifpegmatch`, voir page 11 ;
- `\pegcount`, voir page 15 ;
- `\pegreplace`, voir page 15.

Les macros suivantes sont modifiées par la macro `\toksdo`, mais sont cependant restaurées à leur état antérieur après la fin de l'exécution de `\toksdo` : `\setcharcode`, `\setcatcode`, `\deltok`, `\addtok`, `\selfcharcode`, `\selfcatcode` et `\selfindex`.

9.2 Liste des marqueurs

- `\r{`⟨*csv car*⟩`:`⟨*csv catcode*⟩`}`, voir page 5 ;
- `\R{`⟨*csv charcodes*⟩`:`⟨*csv catcode*⟩`}`, voir page 5 ;
- `\s{`⟨*tokens*⟩`}`, voir page 10 ;
- `\S{`⟨*tokens*⟩`}`, voir page 5 ;
- `\.`, voir page 10 ;

— `\c` peut avoir la syntaxe

  `\c{`⟨*catcode*⟩`}{`⟨*tokens*⟩`}` lorsque utilisé dans l'argument de `\s` ou `\S`, voir page 5 ;

  `\c` lorsque placé avant ou après un ⟨*1-motif*⟩, voir page 13 ;

— `\self`, voir page 8 ;

— `\0`, `\1` jusqu'à `\9`, voir page 16 ;

— toute ⟨*macro*⟩, existante ou non, passée en premier argument de `\defpattern`.

★
★ ★

Ce package en version 0.2 se trouve encore à un stade expérimental et il se peut que, malgré les tests qui ont été faits, il contienne de nombreux bugs.

Par ailleurs, certaines fonctionnalités ou syntaxes sont encore susceptible d'évoluer à la marge.

Quoiqu'il en soit, j'espère que `tokstools` vous est utile. N'hésitez pas à me contacter par **`email`** pour me faire part de bugs, dysfonctionnements ou suggestions de fonctionnalités *réalistes* et surtout, ne perdez pas votre temps à le faire sur `https://tex.stackexchange.com` ou tout autre site, vu qu'il est très probable que je n'en prenne pas connaissance.