tokcycle Package Examples

August 25, 2021

Contents

1 Examples, examples, and more examples

Often, the best way to learn a new tool is to see examples of it being used. Here, a number of examples are gathered that span the spectrum of tokcycle usage.

1.1 Application basics

1.1.1 Using the CGMS directives

Apply different directives to Characters (under-dot), Groups (visible braces), Macros (boxed, detokenized), and Spaces (visible space).

The `\underdot` macro

```
\newcommand\underdot[1]{\ooalign{#1\cr\hfil{\raisebox{-5pt}{.}}\hfil}}
```

Apply different directives to characters, groups, macros, and spaces

```
\tokcycle{\addcytoks{\underdot{#1}}}
        {\addcytoks{\{}\processtoks{#1
         \addcytoks{\}}}
        {\addcytoks{\fbox{\detokenize{#1}}}}
        {\addcytoks{\textvisiblespace}}
{This \textit{is \textbf{a}} test.}
\the\cytoks
```

This␣ \textit ｛is␣ \textbf ｛a｝｝␣test.

1.1.2 Escaping text

Text between two successive escape characters is bypassed by tokcycle and instead echoed to the output token list (`\cytoks`). The default escape character is |. One can change the tokencycle escape character by way of the `\settcEscapechar` macro. Below, the character codes are incremented by one; however, the arguments of `\rule` are protected against that.

The unexpandable `\plusl` macro to increase the ASCII value of tokens by 1

```
\newcommand\plusl[1]{\char\numexpr`#1+1\relax}
```

Escaping text and macro parameters in the input stream

```
\tokcycle
{\addcytoks{\plusl{#1}}}
{\processtoks{#1}}
{\addcytoks{#1}}
{\addcytoks{#1}}
{This \fbox{code is a test
  |(I can also escape text)|}
  of |\rule{1em}{.5em}|
  {\bfseries mine}.}
\the\cytoks
```

Uijt ｜dpef jt b uftu (I can also escape text)｜ pg ▬ **njof**/

1.1.3 Unexpandable, unexpanded, and expanded Character directives

This section concerns the issue of whether the characters of the input stream are transformed before or after being placed in the output token list (`\cytoks`).

Transform characters (+1 ASCII) via unexpandable macro:

<table>
<tr><td>

Unexpandable Character directive

```
\tokcycle
{\addcytoks{\plusl{#1}}}
{\processtoks{#1}}
{\addcytoks{#1}}
{\addcytoks{#1}}{%
  This \textit{code \textup{is} a test} of mine.}
\the\cytoks
```

</td><td>

Uijt *dpef* jt *b uftu* pg njof/

</td></tr>
</table>

\cytoks *alt*detokenization:

```
\plusl{T}\plusl{h}\plusl{i}\plusl{s} \textit{\plusl{c}\plusl{o}\plusl{d}\plusl{e} \textup{\plusl{i}
\plusl{s}} \plusl{a} \plusl{t}\plusl{e}\plusl{s}\plusl{t}} \plusl{o}\plusl{f} \plusl{m}\plusl{i}
\plusl{n}\plusl{e}\plusl{.}
```

Note how the above detokenization shows the actual transformation instructions involving `\plusl`. This is done because `\plusl` is not expandable.

Capitalize vowels (but don't expand the character directive)

The expandable `\vowelcap` macro

```
\newcommand\vowelcap[1]{%
  \ifx a#1A\else
  \ifx e#1E\else
  \ifx i#1I\else
  \ifx o#1O\else
  \ifx u#1U\else
  #1\fi\fi\fi\fi\fi
}
```

<table>
<tr><td>

Not expanded Character directive

```
\tokcycle
{\addcytoks{\vowelcap{#1}}}
{\processtoks{#1}}
{\addcytoks{#1}}
{\addcytoks{#1}}{%
  This \textit{code \textup{is} a test} of mine.}
\the\cytoks
```

</td><td>

ThIs *cOdE* Is *A tEst* Of mInE.

</td></tr>
</table>

\cytoks *alt*detokenization:

```
\vowelcap{T}\vowelcap{h}\vowelcap{i}\vowelcap{s} \textit{\vowelcap{c}\vowelcap{o}\vowelcap{d}
\vowelcap{e} \textup{\vowelcap{i}\vowelcap{s}} \vowelcap{a} \vowelcap{t}\vowelcap{e}\vowelcap{s}
\vowelcap{t}} \vowelcap{o}\vowelcap{f} \vowelcap{m}\vowelcap{i}\vowelcap{n}\vowelcap{e}\vowelcap{.}
```

The above detokenization also shows the transformation macro, `\vowelcap`, as part of `\cytoks`. But because `\vowelcap` is expandable, we did not need to leave it at this stage, as the next example shows.

Capitalize vowels (expanding the character directive)

By using the [x] option of \addcytoks, the \vowelcap is fully expanded prior to being placed in the \cytoks token list.

Expanded Character directive

```
\tokcycle
{\addcytoks[x]{\vowelcap{#1}}}
{\processtoks{#1}}
{\addcytoks{#1}}
{\addcytoks{#1}}{%
  This \textit{code \textup{is} a test} of mine.}
\the\cytoks
```

ThIs *cOdE* Is *A tEst* Of mInE.

\cytoks *alt*detokenization:

`ThIs \textit{cOdE \textup{Is} A tEst} Of mInE.`

Here, we see no vestiges of \vowelcap in \cytoks, but only the transformed result.

1.1.4   Unexpanded vs. pre-expanded input stream

Normal token cycle (input stream not pre-expanded)

```
\tokcycle
{\addcytoks[x]{\vowelcap{#1}}}
{\processtoks{#1}}
{\addcytoks{#1}}
{\addcytoks{#1}}%
{This \fbox{code
  is a test \today} of
  {\bfseries mine}.}
\the\cytoks
```

ThIs $\boxed{\text{cOdE Is A tEst August 25, 2021}}$ Of **mInE**.

\cytoks *alt*detokenization:

`ThIs \fbox{cOdE Is A tEst \today} Of {\bfseries mInE}.`

In the above case, vowel-capping is not applied to the date, because \today remains unexpanded in the \cytoks token list.

Note that, when pre-expanding the input stream, one must \noexpand the macros that are *not* to be pre-expanded.

Pre-\expanded token cycle input stream

```
\expandedtokcyclexpress
{This \noexpand\fbox{code
  is a test \today} of
  {\noexpand\bfseries mine}.}
\the\cytoks
```

ThIs $\boxed{\text{cOdE Is A tEst AUgUst 25, 2021}}$ Of **mInE**.

\cytoks *alt*detokenization:

`ThIs \fbox{cOdE Is A tEst AUgUst 25, 2021} Of {\bfseries mInE}.`

Here, we see the date subjected to vowel-capping, because the tokcycle input was expanded prior to the token cycle. Thus \today no longer appears *explicitly* in the \cytoks token list. Its pre-expansion into character tokens is now subject to the \vowelcap.

1.2 Grouping

Differentiating explicit groups, e.g., {...}, from implicit groups, e.g. `\bgroup...\egroup`, is done automatically by tokcycle. The user has options on how tokcycle should treat these tokens. The desired options are to be set prior to the tokcycle invocation.

1.2.1 Treatment options for implicit groups

The macro `\stripimplicitgroupingcase` can take three possible integer arguments: 0 (default) to automatically place unaltered implicit group tokens in the `\cytoks` output; 1 to strip implicit group tokens from the output; or −1 to instead pass the implicit group tokens to the Character directive (as implicit tokens) for separate processing (typically, when detokenization is desired).

Using `\stripimplicitgroupingcase` to affect treatment of implicit grouping tokens

```
\resettokcycle
\Characterdirective{\addcytoks[x]{%
  \vowelcap{#1}}}
\def\z{Announcement:
  {\bfseries\bgroup\itshape
  Today \egroup it is} \today,
  a Wednesday}
\expandafter\tokencyclexpress\z
\endtokencyclexpress\medskip

\detokenize\expandafter{\the\cytoks}
\bigskip

\stripimplicitgroupingcase{1}
\expandafter\tokencyclexpress\z
\endtokencyclexpress\medskip

\detokenize\expandafter{\the\cytoks}
```

AnnOUncEmEnt: ***TOdAy* It Is** August 25, 2021, A WEdnEsdAy

AnnOUncEmEnt: {\bfseries \bgroup \itshape TOdAy \egroup It Is} \today , A WEdnEsdAy

AnnOUncEmEnt: ***TOdAy It Is*** August 25, 2021, A WEdnEsdAy

AnnOUncEmEnt: {\bfseries \itshape TOdAy It Is} \today , A WEdnEsdAy

---

1.2.2 Treatment options for explicit groups

For explicit group tokens, e.g., { }, there are only two options to be had. These are embodied in the if-condition `\ifstripgrouping` (default `\stripgroupingfalse`). Regardless of which condition is set, the tokens within the explicit group are still passed to the Group directive for processing.

Permutations of the following code are used in the subsequent examples. Group stripping, brought about by `\stripgroupingtrue`, involves removing the grouping braces from around the group. The choice of `\processtoks` vs. `\addcytoks` affects whether the tokens inside the group are recommitted to tokcycle for processing, or are merely sent to the `\cytoks` token list in their original unprocessed form.

Note that, in these examples, underdots and visible spaces will only appear on characters and spaces that have been directed to the Character and Space directives, respectively. Without `\processtoks`, that will not occur to tokens *inside* of groups.

Code permutations on group stripping and inner-group token processing

```
\stripgroupingfalse  OR  \stripgroupingtrue
\tokcycle{\addcytoks{\underdot{#1}}}
        {\processtoks{#1}}  OR  {\addcytoks{#1}}}
        {\addcytoks{#1}}
        {\addcytoks{\textvisiblespace}}
{This \fbox{is a \fbox{token}} test.}
\the\cytoks
```

\stripgroupingfalse \processtoks

This␣ is␣a␣ token ␣test.

\stripgroupingfalse \addcytoks

This␣ is a token ␣test.

\stripgroupingtrue \processtoks

This␣ i s␣a␣ t oken␣test.

\stripgroupingtrue \addcytoks

This␣ i s a token ␣test.

Note that the content of groups can be altogether eliminated if *neither* `\processtoks{#1}` nor `\addcytoks{#1}` are used in the Group directive.

1.2.3 Group nesting

The \reducecolor and \restorecolor macros

```
\newcounter{colorindex}
\newcommand\restorecolor{\setcounter{colorindex}{100}}
\newcommand\reducecolor[1]{%
  \color{red!\thecolorindex!cyan}%
  \addtocounter{colorindex}{-#1}%
  \ifnum\thecolorindex<1\relax\setcounter{colorindex}{1}\fi}
```

Group nesting is no impediment to tokcycle

```
\restorecolor
\tokcycle
  {\addcytoks{(#1)}}
  {\addcytoks{\reducecolor{11}}%
   \addcytoks{[}\processtoks{#1}%
   \addcytoks{]}}
  {\addcytoks{#1}}
  {}{%
  {1{{3{{5{{7{{9{1{}0}}8}}6}}4}}2}}}
\the\cytoks
```

[(1)[[(3)[[(5)[[(7)[[(9)[(1)[](0)]](8)]](6)]](4)]](2)]]

6

Note, above, that tokcycle does not rely on an argument-based nesting algorithm that would otherwise be limited to 9 embedded groups.

1.2.4 The use of \groupedcytoks for more sophisticated group processing

By default, tokcycle processes groups such that whatever additions to the \cytoks token list you accomplish in the Group directive will be encased in an explicit grouped scope. This is the token-cycle behavior with the default \stripgroupingfalse declaration. However, for more sophisticated tasks associated with the processing of group content, one may override the default (by way of \stripgroupingtrue) and manually dictate the groupings associated with the Group directive, with the use of \groupedcytoks{...}.

In the following example, tasks A and D are performed *outside* of the explicit grouping otherwise introduced with \groupedcytoks, whereas tasks B and C fall inside of that explicit group. The fact that, in the output, d is not italic confirms the fact that task D occurs after the conclusion of the explicit \cytoks group that encompasses the italic 'bxc'. The fact that none of a, b, c, nor d are encased in an \fbox indicates that they were not processed through the Character directive, but arose from direct additions to \cytoks, by way of their tasks.

Use of the \groupedcytoks **macro**	

``
\stripgroupingtrue
\Characterdirective{\addcytoks{\fbox{#1}}}
\Groupdirective{\taskA%
  \groupedcytoks{\taskB\processtoks{#1}\taskC}%
  \taskD}
\def\taskA{\addcytoks{\bgroup\LARGE a}}
\def\taskB{\addcytoks{\itshape b}}
\def\taskC{\addcytoks{c}}
\def\taskD{\addcytoks{d\egroup}}
 	
\tokcyclexpress{x\textbf{x}x}
\the\cytoks
``
	$\boxed{\text{x}}\mathbf{a}\textit{b}\boxed{\boldsymbol{x}}\mathbf{cd}\boxed{\text{x}}$

The final disposition of the \cytoks token list shows the effects of \stripgroupingtrue and \groupedcytoks more explicitly:

\cytoks *alt*detokenization:
`\fbox{x}\textbf\bgroup\LARGE a{\itshape b\fbox{x}c}d\egroup\fbox{x}`

Without \stripgroupingtrue, all tasks, A,B,C, and D, would have been contained within the explicit {...} group.

1.3 Direct use of **tokcycle**

The tokcycle macros and pseudo-environments (in regular or xpress form) may be invoked directly from the document, without being first encapsulated within a separate macro or environment.

1.3.1 Modifying counters as part of the Character directive

<table>
<tr><td colspan="2" style="background:#7a2214;color:white">Using a period token (.) to reset a changing color every sentence</td></tr>
<tr><td>

```
\restorecolor
\tokencycle
  {\addcytoks{\bgroup\reducecolor{3}#1\egroup}%
   \ifx.#1\addcytoks{\restorecolor}\fi}
  {\processtoks{#1}}
  {\addcytoks{#1}}
  {\addcytoks{#1}}%
This right \textit{here is a sentence in italic}.
And \textbf{here we have another sentence in bold}.

{\scshape Now in a new paragraph, the sentence
is long.} Now, it is short.
\endtokencycle
```

</td><td>

This right *here is a sentence in italic.* And **here we have another sentence in bold.** Now in a new paragraph, the sentence is long. Now, it is short.

</td></tr>
</table>

1.4 Macro encapsulation of **tokcycle**

1.4.1 Spacing out text

<table>
<tr><td style="background:#5e7d7d;color:white">The \spaceouttext macro</td></tr>
<tr><td>

```
\newcommand\spaceouttext[2]{%
  \tokcycle
    {\addcytoks{##1\nobreak\hspace{#1}}}%
    {\processtoks{##1}}
    {\addcytoks{##1}}%
    {\addcytoks{##1\hspace{#1}}}
    {#2}%
  \the\cytoks\unskip}
```

</td></tr>
</table>

<table>
<tr><td colspan="2" style="background:#7a2214;color:white">\spaceouttext demo</td></tr>
<tr><td>

```
\spaceouttext{3pt plus 3pt}{This
  \textit{text \textbf{is}
  very} spaced out}. Back
  to regular text.

\spaceouttext{1.5pt}{This
  \textit{text \textbf{is}
  somewhat} spaced out}.
  Back to regular text.
```

</td><td>

T h i s *t e x t* ***i s*** *v e r y* s p a c e d o u t. Back to regular text.
T h i s *t e x t* ***i s*** *s o m e w h a t* s p a c e d out. Back to regular text.

</td></tr>
</table>

Note that embedded macros such as `\textit` and `\textbf` do not adversely affect the capability of the `\spaceouttext` macro.

1.4.2 Alternate presentation of detokenized content

This macro attempts to give a more natural presentation of `\detokenize`'d material. It is **not** to be confused as a replacement for `\detokenize`. In certain applications, it may offer a more pleasingly formatted typesetting of detokenized material.

It is an unusual application of tokcycle in that it does not actually use the `\cytoks` token list to collect its output. This is only possible because all macros in the input stream are detokenized,

rather than executed.

The `\altdetokenize` macro

```
\newif\ifmacro
\newcommand\altdetokenize[1]{\begingroup\stripgroupingtrue\macrofalse
  \stripimplicitgroupingcase{-1}%
  \tokcycle
    {\ifmacro\def\tmp{##1}\ifcat\tmp A\else\unskip\allowbreak\fi\macrofalse\fi
     \detokenize{##1}\ifx##1\bgroup\unskip\fi\ifx##1\egroup\unskip\fi}
    {\ifmacro\unskip\macrofalse\fi\{\processtoks{##1}\ifmacro\unskip\fi\}\allowbreak}
    {\tctestifx{\\##1}{\\}{\ifmacro\unskip\allowbreak\fi
     \allowbreak\detokenize{##1}\macrotrue}}
    { \hspace{0pt plus 3em minus .3ex}}
    {#1}%
  \unskip
\endgroup}
```

`\altdetokenize` demo

```
\string\altdetokenize: \\
\texttt{\altdetokenize{a\mac a \mac2
    {\mac}\mac{a\mac\mac}\mac}}!

versus \string\detokenize: \\
   \texttt{\detokenize{a\mac a \mac2
    {\mac}\mac{a\mac\mac}\mac}}!
```

\altdetokenize:
a\mac a \mac2 {\mac}\mac{a\mac\mac}\mac!
versus \detokenize:
a\mac a \mac 2 {\mac }\mac {a\mac \mac
}\mac !

1.4.3   Capitalize all words, including compound and parenthetical words

---

The `\Titlecase` and `\nextcap` macros

```
\newcommand\TitleCase[1]{%
  \def\capnext{T}% <- INITIAL ENTRY
  \tokcycle
    {\nextcap{##1}}
    {\def\capnext{T}\processtoks{##1}}% <- GROUP ENTRY
    {\addcytoks{##1}}
    {\addcytoks{##1}\def\capnext{T}}% <-CAT-10 SPACE
    {#1}%
  \the\cytoks
}
\newcommand\nextcap[1]{%
      \edef\tmp{#1}%
      \tctestifx{-#1}{\def\capnext{T}}{}% <- TEST FOR HYPHEN
      \tctestifcon{\if T\capnext}%
        {\tctestifcon{\ifcat\tmp A}% <- TEST FOR NEXT CAT-11
          {\addcytoks{\uppercase{#1}}\def\capnext{F}}%
          {\addcytoks{#1}}}%
        {\addcytoks{#1}}%
}
```

> **A demo of `\Titlecase` showing raw (escaped) input and processed output**
>
> ```
> \TitleCase{%
> |here, {\bfseries\today{}, is [my]}
>   really-big-test
>   (\textit{capitalizing} words).|
>
>  here, {\bfseries\today{}, is [my]}
>   really-big-test
>   (\textit{capitalizing} words).}
> ```
>
> here, **August 25, 2021, is [my]** really-big-test (*capitalizing* words).
> Here, **August 25, 2021, Is [My]** Really-Big-Test (*Capitalizing* Words).

As it is currently set up, a future-capitalization of the next catcode-11 token will be triggered when any of the following is encountered in the input stream:

- The initial entry into the routine

- a catcode-10 space token

- a hyphen `-`

- upon entry into a group `{...}`

Obviously, additional tests could be added.

1.4.4   Scaling rule dimensions

This example only applies if one can guarantee that the input stream will contain only text and rules...

> **The `\growdim` macro**
>
> ```
> \newcommand\growdim[2]{%
> \tokcycle{\addcytoks{##1}}
>          {\addcytoks{#1\dimexpr##1}}
>          {\addcytoks{##1}}
>          {\addcytoks{##1}}{%
>     #2}%
> \the\cytoks}
> ```

> **Using tokcycle to change `\rule` dimensions**
>
> ```
> \growdim{2}{This rule is exactly 4pt:
>   \rule|{4pt}{4pt}| , whereas this
>   rule is 2x bigger than 4pt:
>   \rule{4pt}{4pt} .}\par
> \growdim{4}{This rule is exactly 5pt:
>   \rule|{5pt}{5pt}| , whereas this
>   rule is 4x bigger than 5pt:
>   \rule{5pt}{5pt} .}
> ```
>
> This rule is exactly 4pt: ▪ , whereas this rule is 2x bigger than 4pt: ■ .
> This rule is exactly 5pt: ▪ , whereas this rule is 4x bigger than 5pt: ■ .

1.4.5 String search, including non-regex material

The `\findinstring` macro for string searches

```
\newcommand\findinstring[2]{\begingroup%
  \stripgroupingtrue
  \setcounter{runcount}{0}%
  \tokcycle
    {\nextctltok{##1}}
    {\nextctltok{\opengroup}\processtoks{##1}\nextctltok{\closegroup}}
    {\nextctltok{##1}}
    {\nextctltok{\tcspace}}
    {#1}%
  \edef\numlet{\theruncount}%
  \expandafter\def\expandafter\searchword\expandafter{\the\cytoks}%
%
  \aftertokcycle{\matchfound}%
  \setcounter{runcount}{0}%
  \def\matchfound{F}%
  \tokcycle
    {\nextcmptok{##1}}
    {\nextcmptok{\opengroup}\processtoks{##1}\nextcmptok{\closegroup}}
    {\nextcmptok{##1}}
    {\nextcmptok{\tcspace}}
    {#2}%
\endgroup}
\newcounter{runcount}
\makeatletter
\newcommand\rotcytoks[1]{\cytoks\expandafter\expandafter\expandafter{%
  \expandafter\tc@gobble\the\cytoks#1}}
\makeatother
\newcommand\testmatch[1]{\ifx#1\searchword\gdef\matchfound{T}\fi}%
\newcommand\rotoradd[2]{\stepcounter{runcount}%
  \ifnum\theruncount>\numlet\relax#1\else#2\fi
  \expandafter\def\expandafter\tmp\expandafter{\the\cytoks}}
\newcommand\nextcmptok[1]{\rotoradd{\rotcytoks{#1}}{\addcytoks{#1}}\testmatch{\tmp}}
\newcommand\nextctltok[1]{\stepcounter{runcount}\addcytoks{#1}}
```

Demo of the `\findinstring` macro

```
1. \findinstring{this}{A test of the times}
   \findinstring{the} {A test of the times}\par
2. \findinstring{This is}{Here, This is a test}
   \findinstring{Thisis} {Here, This is a test}\par
3. \findinstring{the}        {This is the\bfseries{} test}
   \findinstring{he\bfseries}{This is the\bfseries{} test}\par
4. \findinstring{a{bc}} {gf{vf{a{b c}g}gh}hn}
   \findinstring{a{b c}}{gf{vf{a{b c}g}gh}hn}\par
5. \findinstring{a\notmymac{b c}}{gf{vf{a\mymac{b c}g}gh}hn}
   \findinstring{a\mymac{b c}}   {gf{vf{a\mymac{b c}g}gh}hn}\par
6. \findinstring{\textit{Italic}}{this is an \textit{italic} test}
   \findinstring{\textit{italic}}{this is an \textit{italic} test}
```

1. F T
2. T F
3. T T
4. F T
5. F T
6. F T

1.4.6 Counting the significant digits in a number

The problem of counting the significant digits in a number is an interesting one, in that leading zeros are not significant, trailing zeros are sometimes significant, but zeros in the middle of the number are significant. In a problem like this, the \cytoks token list is not used to carry a transformation of the input tokens, but only to carry a record of errant tokens from the input (as seen from the helper macro \throwerr). If \the\cytoks is anything other than empty when the macro completes, the macro will set the significant-digit counter sigd to a value of -1 to signify an error condition.

The macro expects its input to comprise digits in the 0–9 range, an [optional] single decimal point, and ignorable spaces. Any other token will throw an error.

The main work of the \sigdigits macro is to use the counters sigd to count significant digits and trailingz to tabulate the number of trailing-zeros with which to decrement sigd.

```
The \sigdigits macro for counting significant digits
\newcounter{sigd}
\newcounter{trailingz}
\newif\iffounddot
\newif\ifzerosig
\newcommand\sigdigits[1]{%
  \setcounter{sigd}{0}% SIGNIFICANT DIGIT COUNT
  \setcounter{trailingz}{0}% TRAILING ZERO COUNT
  \founddotfalse% MADE T WHEN DECIMAL HAS BEEN 1ST LOCATED
  \zerosigfalse% MADE T WHEN ZERO BECOMES SIGNIFICANT
  \tokcycle% CYCLE THROOUGH EACH TOKEN
  {\tctestifx{.##1}%
   {\iffounddot\throwerr\fi\founddottrue}% IF .
   {\tctestifnum{`##1>`/}%
    {\tctestifnum{`##1<`:}%
     {\tctestifnum{##1>0}%
       {\stepcounter{sigd}\zerosigtrue\setcounter{trailingz}{0}}% IF 1-9
       {\ifzerosig\stepcounter{sigd}\fi\stepcounter{trailingz}}% IF 0
     }%
     {\throwerr}% IF ASCII > `9
    }%
    {\throwerr}% IF ASCII < `0
   }%
  }% APPLY ABOVE LOGIC FOR CHAR TOKENS
  {\throwerr}% IF BRACES
  {\throwerr}% IF CONTROL SEQUENCE
  {}% IGNORE SPACES
  {#1}% THE ARGUMENT
  \iffounddot\else\addtocounter{sigd}{-\thetrailingz}\fi% TRAILING ZEROS
  \expandafter\ifx\expandafter\relax\detokenize\expandafter{\the\cytoks}%
  \relax\else\setcounter{sigd}{-1}\fi% CHECK FOR INVALID INPUT
  \thesigd
}
\newcommand\throwerr{\addcytoks{##1}}% ADD INVALID TOKS TO \cytoks
```

Example use of the `\sigdigits` macro

1) `\sigdigits{12.3}\\`	1) 3
2) `\sigdigits{0.123}\\`	2) 3
3) `\sigdigits{1.2.3}\\`	3) -1
4) `\sigdigits{1.3\today}\\`	4) -1
5) `\sigdigits{321.345 678}\\`	5) 9
6) `\sigdigits{000321.305}\\`	6) 6
7) `\sigdigits{.000 300 345}\\`	7) 6
8) `\sigdigits{0003x1.345}\\`	8) -1
9) `\sigdigits{123000}\\`	9) 3
A) `\sigdigits{123.000}\\`	A) 6
B) `\sigdigits{0003;345}`	B) -1

1.5 Creating your own **tokcycle**-based environments

The `\tokcycleenvironment` macro allows users to define their own tokcycle environments. The more recently introduced `\xtokcycleenvironment` macro extends that capability to allow initialization and postprocessing when defining a named tokcycle environment. Here are some examples of both types.

1.5.1 "Removing" spaces, but still breakable/hyphenatable

The `\spaceBgone` environment

```
\tokcycleenvironment\spaceBgone
    {\addcytoks{##1}}
    {\processtoks{##1}}
    {\addcytoks{##1}}
    {\addcytoks{\hspace{.2pt plus .2pt minus .8pt}}}%
```

Example of using the `\spaceBgone` environment

```
\spaceBgone
  Here we have a \textit{test} of
  whether the spaces are removed.
  We are choosing to use the
  tokencycle environment.

  We are also testing the use of
  paragraph breaks in the
  environment.
\endspaceBgone
```

Herewehavea*test*ofwhetherthespacesareremoved.We arechoosingtousethetokencycleenvironment. Wearealsotestingtheuseofparagraphbreaksintheen-vironment.

1.5.2 Remapping text

The `\remaptext` environment with supporting macros

```
\tokcycleenvironment\remaptext
    {\addcytoks[x]{\tcremap{##1}}}
    {\processtoks{##1}}
    {\addcytoks{##1}}
    {\addcytoks{##1}}
\newcommand*\tcmapto[2]{\expandafter\def\csname tcmapto#1\endcsname{#2}}
\newcommand*\tcremap[1]{\ifcsname tcmapto#1\endcsname
  \csname tcmapto#1\expandafter\endcsname\else\expandafter#1\fi}
\tcmapto am    \tcmapto bf    \tcmapto cz    \tcmapto de    \tcmapto ey
\tcmapto fl    \tcmapto gx    \tcmapto hb    \tcmapto ic    \tcmapto jn
\tcmapto ki    \tcmapto lr    \tcmapto mh    \tcmapto nt    \tcmapto ok
\tcmapto ps    \tcmapto qa    \tcmapto ro    \tcmapto sq    \tcmapto tw
\tcmapto uj    \tcmapto vp    \tcmapto wd    \tcmapto xg    \tcmapto yu
\tcmapto zv
```

Demo of `\remaptext`

```
\remaptext
What can't we \textit{accomplish} if we try?


Let us be of good spirit and put our minds to it!
\endremaptext
```

Wbmw zmt'w dy *mzzkhsrcqb* cl dy wou?

Lyw jq fy kl xkke qscocw mte sjw kjo hcteq wk cw!

Because `\tcremap` is expandable, the original text is totally absent from the processed output:

`\cytoks` *alt*detokenization:

```
Wbmw zmt'w dy \textit{mzzkhsrcqb} cl dy wou?  \par Lyw jq fy kl xkke qscocw mte sjw kjo hcteq wk cw!
```

This approach can be used, for example, to temporarily remap Latin into Greek in math mode. In the example given below, only several of the Latin letters are remapped for demonstration purposes, but others can be added. Also, remember that one can use the `|...|` delimiters to escape back to Latin within the environment.

The `\greek` environment with supporting macros

```
\tokcycleenvironment\greek%        NEW TOKCYCLE ENVIRONMENT:
{\addcytoks[4]{\tcremap{##1}}}%  HOW TO TREAT CHARACTERS
{\processtoks{##1}}%             HOW TO TREAT GROUP CONTENTS
{\addcytoks{##1}}%               HOW TO TREAT MACROS
{\addcytoks{##1}}%               HOW TO TREAT SPACES
\newcommand*\tcmapto[2]{\expandafter\def\csname tcmapto#1\endcsname{#2}}
\newcommand*\tcremap[1]{\ifcsname tcmapto#1\endcsname
  \csname tcmapto#1\expandafter\endcsname\else\expandafter#1\fi}
\tcmapto a\alpha \tcmapto b\beta \tcmapto e\epsilon
\tcmapto g\gamma \tcmapto p\pi
```

<div style="border: box">

Demo of `\greek`

```
\[
  y =
  \greek a^2b\frac{2e}{2 + p^|p| + g}\endgreek
  + x
\]
```

$$y = \alpha^2\beta\frac{2\epsilon}{2 + \pi^p + \gamma} + x$$

</div>

1.5.3   Creating an "extended" **tokcycle** environment

The `\xtokcycleenvironment` macro extends the capability of `\tokcycleenvironment` by adding two arguments that correspond, respectively, to initialization code and "close-out" code that is run after the token cycle is complete (but before it is typeset). Let's say I wanted an environment that capitalizes instances of the letter 't', but I want it to tell me how many letter `t`'s are affected before typesetting the result. Furthermore, I want it always to set the result in red and add two exclamation points to the end of the environment output.

<div style="border: box">

The `\capTive` extended-environment definition

```
\newcounter{Tcount}
\xtokcycleenvironment\capTive% ENVIRONMENT NAME
{\tctestifx{t##1}{\stepcounter{Tcount}\addcytoks{T}}{\addcytoks{##1}}}% CHAR. DIRECTIVE
{\processtoks{##1}}% GROUP DIRECTIVE
{\addcytoks{##1}}% MACRO DIRECTIVE
{\addcytoks{##1}}% SPACE DIRECTIVE
{\setcounter{Tcount}{0}\color{red}\tcafterenv{\ Here}}% INITIALIZATION CODE
{\theTcount{} t's capitalized: \addcytoks{!!}}% CLOSE-OUT CODE
```

</div>

<div style="border: box">

Demo of `\capTive` environment

```
Before \capTive the cytoks \footnotesize token
    list\endcapTive After
```

Before 4 t's capitalized: The cyToks Token lisT!! HereAfter

</div>

In the initialization code, the `Tcount` counter is reset and the color of what follows is made red. It also makes use of the specialized `\tcafterenv` macro that can **only be used in `\xtokcycle-environment` definitions**. This macro indicates code to execute *after* the `capTive` environment scope is exited. This invocation typesets the word 'Here'. We see that in the output, it occurs at normal size and in black, because it is no longer subject to the color and fontsize that was left inside the environment. The `\tcafterenv` macro is unique in that it is an appending, rather than a defining, macro. Therefore, multiple invocations of it will append multiple code segments, each of which will be executed upon environment exit. It may be invoked in any of the directives, as well as the initialization and close-out code segments.

For the close-out code, because it is executed after the token cycle is complete, but before it is typeset, the updated value of `Tcount` is available. Anything typeset directly in the close-out code will occur before the `\cytoks` token-list output is typeset. Thus, the "...t's capitalized." phrase is typeset before `\cytoks` itself, even as `\theTcount` is accurate. The `\addcytoks` occurring in the close-out code is placed at the end of `\cytoks` and thus takes on the `\footnotesize` declaration that is still in force.

Next, we consider a **tokcycle** extended environment that counts words and letters. We try to build some features into the code so that, for example, spaces and punctuation do not count as letters. Further, we try to build the logic so that the following cases would count as two words, not one: "here...now", "Case(number)".

We provide two modes of output: one that provides an ending summary, and the other that keeps a running count. As with other tokcycle environments and macros, portions can be exempted from the count (for example macro arguments) by using the escape delimiters provided. An even more elaborate version of this environment can be found at https://tex.stackexchange.com/questions/44618/dynamically-count-and-return-number-of-words-in-a-section/587567#587567, including a version that runs in Plain TeX.

The `\countem` extended-environment definition and supporting macros

```
\newcounter{wordcount}
\newcounter{lettercount}
\newif\ifinword
\newif\ifrunningcount
\newcommand\dumpword{%
  \addcytoks[1]{\accumword}%
  \tctestifcon\ifinword{\stepcounter{wordcount}%
    \ifrunningcount\addcytoks[x]{$^{\thewordcount,\thelettercount}$}\fi}{}%
  \inwordfalse\def\accumword{}}
\makeatletter
\newcommand\addletter[1]{%
  \tctestifcatnx A#1{\stepcounter{lettercount}\inwordtrue}{\dumpword}%
  \tc@defx\accumword{\accumword#1}}
\makeatother

\xtokcycleenvironment\countem
  {\addletter{##1}}
  {\processtoks{##1}\dumpword}
  {\dumpword\addcytoks{##1}}
  {\dumpword\addcytoks{##1}}
  {\def\accumword{}\setcounter{wordcount}{0}\setcounter{lettercount}{0}}
  {\dumpword\ifrunningcount\else\tcafterenv{\par(Wordcount=\thewordcount,
    Lettercount=\thelettercount)}\fi}
```

Demo of the two modes of `\countem`

```
\countem
This is a test.

But...punctuation does not count.
\endcountem

\bigskip\runningcounttrue
\countem
This is a test.

But...punctuation does not count.
\endcountem
```

This is a test.
But...punctuation does not count.
(Wordcount=9, Lettercount=37)

This[1,4] is[2,6] a[3,7] test[4,11].
But[5,14]...punctuation[6,25] does[7,29] not[8,32] count[9,37].

1.6 Truncation of input

tokcycle has several facilities for truncating input, either to the end of the current group or to the end of the input stream. The basic commands are `\truncategroup` and `\truncatecycle`, which can be part of the Character, Macro, or Space directives (see the package documentation for truncation from the Group directive).

These commands may be issued directly or they may be placed as one fork of a conditional test. If the conditional test is the occurrence of a particular token in the input stream, special conditional shortcuts have been formulated: `\truncategroupiftokis` and `\truncatecycleiftokis`. Both of these commands take two arguments: the first is the comparison token against which to compare the tokens of the input stream, and the second is the directive to follow for tokens that fail the comparison test. If the input stream token matches the comparitor, the relevant truncation command is executed.

The description below shows the truncations to a given input text, resulting from several different forms of truncating directives. In the first case, the token cycle is to be truncated if a space is found in the input stream. Comparing to the input text, we see that the space occurs after the word "until", and that is indeed where the tokcycle output ceases.

In the second case, the truncation is limited to the current group, rather than the remainder of the input stream. Again, the space following "until" is found, and the words "certain token" are excised from the input stream until the end-of-group is located. The cycle continues successfully with "-is" until the next space is found before "located". Since this space is found in the top grouping level of the input, the remainder of the input stream is thereafter truncated.

In the third case, the cycle is to be truncated in the event the letter 'a' is found in the input stream. Examining the input text, the first 'a' is in the word "certain" and so the input is truncated after the string "cert" is digested.

The final case, calls for group truncation if an 'a' is found in the input stream. As before, the first 'a' is found after "cert". However, in the group case, only the remaining group content, consisting of "ain token", is discarded. At group end, the cycle successfully resumes until the 'a' in "located" is reached. Thus, "-is loc" is also added to the output stream.

Example: Effects of various truncation directives

`Input text`: Continue-\textit{until certain token}-is located.

`\Spacedirective{\truncatecycle}`: Continue-*until*

`\Spacedirective{\truncategroup}`: Continue-*until*-is

`\Characterdirective{\truncatecycleiftokis{a}{\addcytoks{#1}}}`: Continue-*until cert*

`\Characterdirective{\truncategroupiftokis{a}{\addcytoks{#1}}}`: Continue-*until cert*-is loc

Note how the macro function, in this case `\textit`, is not disrupted by the truncation. Even as tokens are truncated from the input stream, group closures are retained so that the final output is group-balanced. For example, a detokenization of the final case reveals this to be the case:

`\cytoks` *alt*detokenization:
`Continue-\textit{until cert}-is loc`

1.7   Look-ahead features

The whole concept of the tokcycle package is to examine the input stream one token at a time and use the directives to process the tokens. Occasionally, however, there are tasks that involve several adjacent tokens. In the example of section 1.4.3, for instance, in order to capitalize words, each space token triggers an action on the character that follows. To accomplish this without any look-ahead features, the Space directive sets a flag, `\def\capnext{T}`, in order to communicate with a the Character directive about a subsequent token. That is a perfectly valid approach to communicate conditional action.

However, tokcycle also provides look-ahead features, to avoid the necessity of communicating with future tokens in the cycle by way of flag settings. These tools allow a directive to directly "peek"

ahead at the input stream (without disturbing it) and, if desired, push and pop tokens to and from the input stream. The package manual should be consulted for a full list of macros provided to facilitate look-ahead capability. But here are some examples to look at.

1.7.1 Count in advance that which follows

Specify the point values for questions and let tokcycle perform the cumulative tallies in advance, using a single-pass compilation.

The `pointtracker` environment to count point values in advance

```
\newcounter{pointcount}
\newcommand\addtopoints[1]{(\textit{#1 point}%
  \ifnum#1=1\relax\else\textit{s}\fi)}
\NewEnviron{pointtracker}[2][points]{%
  \par\bigskip\resettokcycle
  \setcounter{pointcount}{0}%
  \Macrodirective{\addcytoks{##1}\tctestifx{\addtopoints##1}{\sumpoints}{}}%
  \def\tmp{\tokencyclexpress{\large\bfseries#2: \thepointcount\ #1}}%
  \expandafter\tmp\BODY\endtokencyclexpress
}
\def\sumpoints{\tcpop\Q\addtocounter{pointcount}{\Q}\tcpushgroup{\Q}}
```

Demo of the pointtracker environment

```
\begin{pointtracker}[total
    points]{\Large Science Test}
\begin{pointtracker}{Advanced
    Geology}
  \begin{itemize}
    \item Count the fingers on your
    left hand. Are there more than 9?
        \addtopoints{1}
    \item Draw a rectangle with five
    edges. \addtopoints{3}
  \end{itemize}
\end{pointtracker}
\begin{pointtracker}{Meteorology}
  \begin{itemize}
    \item How many borgs does it
    take to brew a Danish beer?
        \addtopoints{2}
    \item What is the meaning of
    life? \addtopoints{4}
  \end{itemize}
\end{pointtracker}
\end{pointtracker}
```

Science Test: 10 total points

Advanced Geology: 4 points

- Count the fingers on your left hand. Are there more than 9? (*1 point*)
- Draw a rectangle with five edges. (*3 points*)

Meteorology: 6 points

- How many borgs does it take to brew a Danish beer? (*2 points*)
- What is the meaning of life? (*4 points*)

Note that this example has several neat features. The points for Geology, Meteorology, and the Test total are not specified, but calculated. They are calculated based on content that appears later in the environment. Traditionally, multiple compilations would be required with external data written to the aux file, but that does not occur here—there is only a single compilation. Finally, becaue we are nesting the tokcycle part of `pointtracker` inside of a `\NewEnviron`, we see that the `pointtracker` environment can be nested, without adverse effect.

1.7.2 Modify the arguments of every instance of a particular macro

This example, through the use of the `\scalerule` macro, shows how the `\rule` arguments, including the optional argument, can be popped off the input stream, adjusted, and placed back in the stream.

The `\scalerule` macro to scale all `\rule` arguments

```
\newcommand\scalerule[1]{%
  \tcpeek\zA
  \ifx[\zA\tcpopuntil]\opt\fi% IF PRESENT, POP OPTIONAL ARG
  \tcpop\ArgA% POP ARG1
  \tcpop\ArgB% POP ARG2
  \tcpushgroup[x]{#1\dimexpr\ArgB}% PUSH BACK IN REVERSE (LIFO) FASHION
  \tcpushgroup[x]{#1\dimexpr\ArgA}
  \ifx[\zA\tcpush[x]{\expandafter\scaleopt\opt{#1}}\fi
}
\def\scaleopt[#1]#2[{[#2\dimexpr#1]}
```

Scale every `\rule` argument

```
\resettokcycle
\settcEscapechar{!}% SO AS NOT TO INTERFERE WITH '|'
    TOKS IN tabular
\Macrodirective{\addcytoks{#1}\ifx\rule#1\scalerule{2}\fi}
\tokencyclexpress
The rules appearing in the following input stream are
    literally
!\rule[-2pt]{1.5ex}{1.5ex}! and !\rule{2ex}{3pt}!:

My environment can include all kinds of rules
  \rule[-2pt]{1.5ex}{1.5ex} and other macros, such as
    \today{} and
\begin{tabular}{c|c}
23 & 454\\\hline
x & y
\end{tabular}
but each rule \rule{2ex}{3pt} is grown by a factor of
    two.
\endtokencyclexpress
```

The rules appearing in the following input stream are literally ■ and ▬:
My environment can include all kinds of rules ■ and other macros, such as August 25, 2021 and $\frac{23 \mid 454}{\text{x} \mid \text{y}}$ but each rule ■ is grown by a factor of two.

1.7.3 Protect arguments of specified macro from Character directive modification

Tokcycle provides the escape character | to delimit tokens to be protected from processing by tokcycle directives. But that requires the input stream to be adjusted to include these delimiters. Here's an alternative, not requiring input stream adjustment. The Macro directive protects all `\rule` arguments against contamination by the Character directive.

When a `\rule` is encountered, `\protectruleargs` is called to pop the rule arguments from the input stream and place them directly in the `\cytoks` token list, out of reach from the Character directive.

```
\newcommand\protectruleargs[1]{\ifx\rule#1%
  \tcpeek\zA
  \ifx[\zA\tcpopuntil]\opt
  \addcytoks[1]{\opt}\fi% IF OPTIONAL ARG PRESENT, POP AND SAVE TO \cytoks
  \tcpopliteral\ArgA% MUST POP ARG 1 WITH BRACES
  \addcytoks[1]{\ArgA}%
  \tcpopliteral\ArgB%  MUST POP ARG 2 WITH BRACES
  \addcytoks[1]{\ArgB}%
  \fi
}
```

Here, every character is typeset with a slash suffix. If this action were to be imposed to `\rule` argument characters, it would destroy their integrity. Fortunately, when the Macro directive detects a `\rule`, it can take evasive action.

Protect every `\rule` argument from Character directive contamination

```
\resettokcycle
\Characterdirective{\addcytoks{#1\textup{/}\allowbreak}}
\Macrodirective{\addcytoks{#1}\protectruleargs{#1}}
\tokencyclexpress
My \textit{environment} modifies each character
    token, but rule
  \rule[-4pt]{1.5ex}{1.5ex} arguments \rule{2ex}{3pt}
    are protected.
\endtokencyclexpress
```

M/y/ e/n/v/i/r/o/n/m/e/
n/t/ m/o/d/i/f/i/e/s/ e/a/
c/h/ c/h/a/r/a/c/t/e/r/ t/o/
k/e/n/,/ b/u/t/ r/u/l/e/ ▪
a/r/g/u/m/e/n/t/s/ ▬ a/r/
e/ p/r/o/t/e/c/t/e/d/./

1.7.4   Find a string of character tokens

Earlier in this manuscript, in section 1.4.5, a token cycle was developed to search a token stream for a string match. Here, we do something similar, except we use the new "look ahead" features of tokcycle to achieve it. In this case, however, the search string cannot cross space or group boundaries. The `\matchnextchar` macro demonstrates a use of `\tcpopappto`, which is one of the newly introduced tokcycle commands; instead of popping an argument from the input stream *into* a macro, it pops it and *appends* it to a macro.

The `\findstring` macro to perform a look-ahead string search

```
\newcommand\findstring[4]{% {CURRENT TOK}{MATCH STRING}{SUCCESS OP}{FAIL CMDS}
  \foundstringfalse% DEFAULT CONDITION
  \def\poptoks{}% INITIALIZATION. IF MATCH FAILS, MUST TO RESTORE POPPED TOKENS
  \expandafter\matchnextchar\ifx#1#2\relax\relax\fi\empty\relax
  \iffoundstring\addcytoks{#3{#2}}\else\tcpush{\poptoks}#4\fi
}
\def\matchnextchar#1#2\relax{\tctestifx{#1\empty}{}{% NO MATCH FROM 1ST TOKEN
  \tctestifx{\relax#1}{%
    \foundstringtrue% NOTHING LEFT TO COMPARE; MATCH FOUND
  }{%
    \tcpeek\Q% WITHOUT POPPING...
    \tctestifx{\Q\bgroup}{}{% PEEK FOR GROUPS OR...
      \tctestifx{\Q\tcsptoken}{}{% SPACES THAT INDICATE MATCH FAILURE
        \tcpopappto\poptoks% POP TOKEN AND APPEND TO \poptoks
        \expandafter\tctestifx\expandafter{\Q#1}{%
          \matchnextchar#2\relax% STILL MATCHING; KEEP GOING
        }{}%
      }%
    }%
  }%
}}
```

Search for the string "red"

```
\resettokcycle
\Characterdirective{\findstring{#1}{red}%
  {\textcolor{purple}}{\addcytoks{#1}}}
\tokencyclexpress
Can I \textit{find red in the text}, even in the
  MIDredDLE of words and groups?
\endtokencyclexpress
```

Can I *find red in the text*, even in the MIDredDLE of words and groups?

1.7.5   Eliminate *prior* spaces and pars before a particular macro

The goal here is to eliminate spaces and paragraphs that occur *prior* to the occurrence of the macro `\foo`, which sets a `\bullet`. Looking backwards is not part of the TeX approach. Nonetheless, the problem can, with a token cycle, be turned on its head, such that all occurrences of spaces and `\par` can be made to look for a future instance of `\foo`.

Here, if a space or `\par` is detected, `\foocheck` is called. It peeks at the token that follows the space or `\par`. If the peeked argument is a `\par` or `\foo` itself, it discards the current space or `\par`.

The `\dofoo` environment to remove prior spaces and paragraphs before `\foo`

```
\newcommand\foo{$\bullet$}
\newcommand\foocheck[1]{\tcpeek\z\ifx\foo\z\else
  \ifx\par\z\else\addcytoks{#1}\fi\fi}
\tokcycleenvironment\dofoo
  {\addcytoks{##1}}
  {\processtoks{##1}}
  {\ifx\par##1\foocheck{##1}\else\addcytoks{##1}\fi}
  {\foocheck{##1}}
```

<table>
<tr><td colspan="2">Remove spaces and paragraphs before \foo</td></tr>
<tr>
<td>

```
\dofoo
Lorem ipsum dolor sit amet. (No foo)


Lorem ipsum dolor sit amet. \foo


Lorem ipsum dolor sit amet.\foo


Lorem ipsum dolor sit amet.
\foo


Lorem ipsum dolor sit amet.

\foo
\enddofoo
```

</td>
<td>

Lorem ipsum dolor sit amet. (No foo)
Lorem ipsum dolor sit amet.●
Lorem ipsum dolor sit amet.●
Lorem ipsum dolor sit amet.●
Lorem ipsum dolor sit amet.●

</td>
</tr>
</table>

1.7.6 Replace := with \coloneqq (mathtools package required)

In most cases, creating a macro to signify a composite symbol makes the most sense. However, sometimes you don't realize that until you've typed in a lot of code that fails to use such a macro. In such a situation, tokcycle can help.

In the `mathtools` package, there is a symbol, `\coloneqq`, a colon followed by an equal sign, in which the colon has been vertically adjusted to be symmetric with respect to the = sign. The adjustment is slight, but noticeable. A token cycle can be set up to look for instances of the : character, peek ahead to see if it is followed by an = character and, if so, replace the pair with `\coloneqq`. However, the process is further complicated, if one is a user of `\usepackage[french]{babel}`, where the : character is is made an active character.

Therefore, in this case, not only does : need to be located by the Character directive (in the normal case), but it also need be searched for by the Macro directive, in the event that `\usepackage[french]{babel}` is being employed. In the example below, there are two such colons that satisfy the criterion for replacement.

<table>
<tr><td>The \coleqchek macro and the \subcoloneqq environment</td></tr>
<tr><td>

```
\newcommand\coleqchek[1]{\tcpeek\z
  \ifx=\z\tcpop\z\addcytoks{\coloneqq}\else\addcytoks{#1}\fi}
\tokcycleenvironment\subcoloneqq
{\tctestifx{:##1}{\coleqchek{##1}}{\addcytoks{##1}}}% <- IF : A CHARACTER
{\processtoks{##1}}
{\tctestifcon{\if\detokenize{:}\detokenize{##1}}% <- IF : AN ACTIVE MACRO
  {\coleqchek{##1}}{\addcytoks{##1}}}
{\addcytoks{##1}}
```

</td></tr>
</table>

<table>
<tr><td colspan="2">replace := with \coloneqq</td></tr>
<tr>
<td>

```
\subcoloneqq
\LARGE Compare: $::=$

$a := b$ vs
  $a \ordinarycolon= b$ vs
  $a:b$.
\endsubcoloneqq
```

</td>
<td>

Compare: ::=
$a := b$ vs $a := b$ vs $a : b.$

</td>
</tr>
</table>

2 Advanced topics: implicit tokens and catcode changes

2.1 Trap active characters (catcode 13)

Active characters in the tokcycle input stream are processed in their original form. Their active substitutions arising from `\def`s only occur *afterwards*, when the tokcycle output is typeset. They may be identified with the `\ifactivetok` test. If `\let` to a character, they may be identified in the Character directive; If `\let` to a control sequence or defined via `\def`, they may be identified in the Macro directive. For information on how to process active spaces, please refer to section 2.5.

Processing active characters

```
\resettokcycle
\tokencyclexpress
1. This is a test!!\endtokencyclexpress

2. \catcode'!=\active
\def !{?}%
\tokencyclexpress
This is a test!!\endtokencyclexpress

3.
\Characterdirective{\tctestifcon\ifactivetok
 {\addcytoks{\fbox{#1-chr}}}{\addcytoks{#1}}}%
\Macrodirective{\tctestifcon\ifactivetok
 {\addcytoks{\fbox{#1-mac}}}{\addcytoks{#1}}}%
\tokencyclexpress
This is a test!!\endtokencyclexpress

4. \catcode'T=\active
\let T+%
\tokencyclexpress
This is a test!!\endtokencyclexpress

5. \detokenize\expandafter{\the\cytoks}
```

1. This is a test!!
2. This is a test??
3. This is a test ?-mac ?-mac
4. +-chr his is a test ?-mac ?-mac
5. \fbox {T-chr}his is a test\fbox {!-mac}\fbox {!-mac}

If the input stream is pre-*expanded*, any active substitutions that are expandable (i.e., those involving `\def` as well as those `\let` to something expandable) are made before reaching tokcycle processing. They are, thus, no longer detected as active, unless `\noexpand` is applied before the pre-expansion. In this example, the '!' that is *not* subject to a `\noexpand` is converted to a '?' prior to reaching tokcycle processing (and thus, not detected as `\active`). However, the 'T' is not pre-converted to a '+', because it is implicit, and not a definition:

Expanded input stream acts upon active \defed characters unless \noexpand is applied

```
\expandedtokcyclexpress{This is a
    test!\noexpand!}
\the\cytoks\par
\detokenize\expandafter{\the\cytoks}
```

+-chr his is a test? ?-mac
\fbox {T-chr}his is a test?\fbox {!-mac}

However, pre-tokenization does not suffer this behavior. The first '!' is tokenized as active during its absorption into `\tmp`, as shown in the next example:

Pre-tokenized input stream does not affect active characters	

`\def\tmp{This is a test!!}`
`\expandafter\tokcyclexpress\expandafter{\tmp}`
`\the\cytoks\par`
`\detokenize\expandafter{\the\cytoks}`
	+-chr his is a test ?-mac ?-mac
	\fbox {T-chr}his is a test\fbox {!-mac}\fbox {!-mac}

One aspect of TeX to remember is that catcodes are assigned at tokenization; however, for active characters, the substitution assignment is evaluated only upon execution. So, if a cat-13 token is placed into a `\def`, it will remain active even if the catcode of that character code is later changed. But if the cat-13 active definition is changed prior to the execution of the `\def`'ed token, the revised token assignment will apply.

The following example demonstrates this concept, while showing, without changing the input in any way, that tokcycle can properly digest active and implicit grouping (cat-1,2) characters:

Active and implicit grouping tokens digestible by tokcycle	

`\catcode‘Y=13`
`\catcode‘Z=13`
`\let Y{`
`\let Z}`
`\let\Y{`
`\let\Z}`
`\def\tmp{\textit YabcZ de\Y\itshape f\Zg}%`
 	
`\def Y{\bgroup[NEW]}% APPLIES AT EXECUTION`
`\catcode‘Y=11% DOES NOT AFFECT Y IN \tmp`
 	
`\expandafter\tokcyclexpress\expandafter{\tmp}`
`\the\cytoks`
 	
`\detokenize\expandafter{\the\cytoks}`
	[NEW]abc de*fg*
	\textit YabcZ de\Y \itshape f\Zg

2.2 Trap catcode 6 (explicit & implicit) tokens

Typically, cat-6 tokens (like #) are used to designate the following digit (1-9) as a parameter. Since they are unlikely to be used in that capacity inside a tokcycle input stream, the package behavior is to convert them into something cat-12 and set the if-condition `\catSIXtrue`. In this manner, `\ifcatSIX` can be used inside the Character directive to convert cat-6 tokens into something of the user's choosing.

As to this cat-12 conversion, explicit cat-6 characters are converted into the same character with cat-12. On the other hand, implicit cat-6 control sequences (e.g., `\let\myhash#`) are converted into a fixed-name macro, `\implicitsixtok`, whose cat-12 substitution text is a `\string` of the original implicit-macro name.

<table>
<tr><td colspan="2">Treatment of cat-6 tokens</td></tr>
</table>

```
\resettokcycle
\Characterdirective{\ifcatSIX
  \addcytoks{\fbox{#1}}
  \else\addcytoks{#1}\fi}
\let\myhash#
\tokcyclexpress{This# isQ
  \textit{a Q# test\myhash}!}
\the\cytoks\bigskip\par
\detokenize\expandafter{\the\cytoks}
```

This ⬚#⬚ isQ *a Q* ⬚#⬚ *test* ⬚\myhash⬚!

This\fbox {#} isQ \textit {a Q\fbox {#} test\fbox {\implicitsixtok }}!

<table>
<tr><td colspan="2">Multiple explicit cat-6 tokens are not a problem</td></tr>
</table>

```
\catcode`Q=6
\tokcyclexpress{This# isQ
  \textit{a Q# test\myhash}!}
\the\cytoks
```

This ⬚#⬚ is ⬚Q⬚ *a* ⬚Q⬚⬚#⬚ *test* ⬚\myhash⬚!

⚠⚠ For what is, perhaps, a rare situation, one can even process input streams that contain cat-6 macro parameters. A package macro, `\whennotprocessingparameter#1{<`*directive when not a parameter*`>}`, can be used inside of the Character directive to intercept parameters. In this example, a macro is defined and then executed, subject to token replacements brought about by the expandable Character directive.

<table>
<tr><td colspan="2">Preserving parameters (e.g. #1, #2) in the tokcycle input stream</td></tr>
</table>

```
\Characterdirective{%
  \whennotprocessingparameter#1{%
    \addcytoks[x]{\vowelcap{#1}}}}
\tokcyclexpress{%
  \def\zQ#1#2{[one:#1](two:#2)}
  This is a \zQ big test.

  \renewcommand\zQ[2]{\ifx t#1[#1]\fi(#2)}
  This is a \zQ test.}
\the\cytoks
```

ThIs Is A [OnE:b](twO:I)g tEst.
ThIs Is A [t](E)st.

`\cytoks` *alt*detokenization:

```
\def\zQ#1#2{[OnE:#1](twO:#2)} ThIs Is A \zQ bIg tEst.  \par\renewcommand\zQ[2]{\ifx t#1[#1]\fi(#2)}
ThIs Is A \zQ tEst.
```

2.3   Trap implicit tokens in general

Implicit control sequences (assigned via `\let` to characters) were already mentioned in the context of cat-6. However, implicit control sequences can be of any valid catcode (except for cat-0, which we instead call macros or primitives). The condition `\ifimplicittok` is used to flag such tokens for special processing, as well as active tokens that are `\let` to anything unexpandable.

In the next example, implicit, cat-6 and implicit-cat-6 tokens may all be differentiated, shown here with a multiplicity of `\fbox`es.

Implicit = single box, cat-6 = double box, implicit-cat-6 = triple box

```
\catcode`Q=\active \let QN
\let\littlet=t
\let\littlel=l
\let\svhash#
\Characterdirective{\ifimplicittok
  \ifcatSIX\addcytoks{\fbox{\fbox{\fbox{#1}}}}%
  \else\addcytoks{\fbox{#1}}\fi\else\ifcatSIX
  \addcytoks{\fbox{\fbox{#1}}}\else
  \addcytoks{#1}\fi\fi}

\tokencyclexpress We wi\littlel\littlel#
  \textit{ make a \littlet est #} \littlet

This \textit{is a \textbf{big}} \littlet est.

Qext pa#agraph ending with implicit cat six
  \svhash.\endtokencyclexpress
```

We wi⸦l⸧⸨#⸩ *make a* ⸦t⸧*est* ⸨#⸩ ⸦t⸧
This *is a* **big** ⸦t⸧est.
⸦N⸧ext pa⸨#⸩agraph ending with implicit cat six ⸨⸨\svhash⸩⸩.

In the following example, we use both control sequences and active characters in `\def` and `\let` capacities, to demonstrate how tokcycle digests things. Implicit tokens (tokens `\let` to characters) are shown in a box, with both the token name and the implicit value (note that tokens `\let` to macros and primitives are not considered implicit). Active tokens processed through the character directive are followed with a †, whereas those processed through the macro directive are followed with a ‡.

Non-active vs. active `\def` & `\let`

```
\Characterdirective{\ifimplicittok
  \addcytoks{\fbox{\detokenize{#1}:#1}}%
  \else\addcytoks{#1}\fi\ifactivetok
  \addcytoks{\rlap{\dag}}\fi\addcytoks{~\,}}
\Macrodirective{\ifimplicittok
  \addcytoks{\fbox{\detokenize{#1}}}%
  \else\addcytoks{#1}\fi\ifactivetok
  \addcytoks{\rlap{\ddag}}\fi
  \ifx\par#1\else\addcytoks{~\,}\fi}

\def\A{a}
\let\B i
\let\C\today
\let\D\relax
\def\E{\relax}
\catcode`V=13 \def V{a}
\catcode`W=13 \let Ww
\catcode`X=13 \let X\today
\catcode`Y=13 \let Y\relax
\catcode`Z=13 \def Z{\relax}
\tokcyclexpress{\A\B\C\D\E ab\par VWXYZab}
\the\cytoks
```

a ⸦\B :i⸧ August 25, 2021   a b
a‡⸦W:w⸧†August 25, 2021‡‡a b

If the input stream is subject to pre-expansion, one will require `\noexpand` for macros where no pre-expansion is desired.

If the input stream is provided pre-tokenized via `\def`, TEX convention requires cat-6 tokens to appear in the input stream as duplicate, e.g. `##`.

2.4 Changing grouping tokens (catcodes 1,2)

Changing grouping tokens (catcodes 1,2) may require something more, if the output stream is to be detokenized. In the following examples, pay attention to the detokenized grouping around the argument to `\fbox`.

As we will see, the issues raised here only affect the situation when detokenization of the output stream is required.

tokcycle defaults grouping tokens to braces:	

`\tokencycle`
` {\addcytoks{(#1)}}`
` {\processtoks{#1}}`
` {\addcytoks{#1}}`
` {\addcytoks{ }}`
`This \fbox{is a} test.`
`\endtokencycle\medskip`
 	
`\detokenize\expandafter{\the\cytoks}`
	(T)(h)(i)(s) $\boxed{(i)(s)\ (a)}$ (t)(e)(s)(t)(.)
	
	(T)(h)(i)(s) \fbox {(i)(s) (a)} (t)(e)(s)(t)(.)

One can make brackets cat-1,2, redefining bgroup/egroup to [ ]. However, while one can now use brackets in input stream, braces will still appear in the detokenized tokcycle output stream:

tokcycle will not automatically change its grouping tokens	

`\catcode'\[=1`
`\catcode'\]=2`
`\let\bgroup[`
`\let\egroup]`
`\tokencycle`
` {\addcytoks{(#1)}}`
` {\processtoks{#1}}`
` {\addcytoks{#1}}`
` {\addcytoks{ }}`
`This \fbox[is a] test.`
`\endtokencycle\medskip`
 	
`\detokenize\expandafter{\the\cytoks}`
	(T)(h)(i)(s) $\boxed{(i)(s)\ (a)}$ (t)(e)(s)(t)(.)
	
	(T)(h)(i)(s) \fbox {(i)(s) (a)} (t)(e)(s)(t)(.)

If it is necessary to reflect revised grouping tokens in the output stream, the `\settcgrouping` macro is to be used.

<div style="border:1px solid; padding:0.5em;">

Redefine tokcycle grouping tokens as angle brackets using `\settcGrouping`

```
\catcode'\<=1
\catcode'\>=2
\catcode'\{=12
\catcode'\}=12
\let\bgroup<
\let\egroup>
\settcGrouping<<#1>>
\tokencycle
 <\addcytoks<(#1)>>
 <\processtoks<#1>>
 <\addcytoks<#1>>
 <\addcytoks< >>
This \fbox<is a> test.
\endtokencycle\medskip

\detokenize\expandafter<\the\cytoks>
```

$(T)(h)(i)(s)$ $\boxed{(i)(s)\ (a)}$ $(t)(e)(s)(t)(.)$

$(T)(h)(i)(s)$ \fbox $<(i)(s)\ (a)>$ $(t)(e)(s)(t)(.)$

</div>

Angle brackets are now seen in the above detokenization. Until subsequently changed, cat-1,2 angle brackets now appear in detokenized **tokcycle** groups, even if other cat-1,2 tokens were used in the input stream. Bottom line:

- adding, deleting, or changing catcode 1,2 explicit grouping tokens, e.g., {}, (in conjunction with their associated implicit `\bgroup\egroup`) tokens will not affect **tokcycle**'s ability to digest proper grouping of the input stream, regardless of which tokens are catcode 1,2 at the moment.

- The grouping tokens used in **tokcycle**'s output default to {} braces (with cat-1,2), but can be changed deliberately using `\settcGrouping`.

- The package, currently, has no way to reproduce in the output stream the actual grouping tokens that occur in the input stream, but one should ask, for the particular application, if it really matters, as long as the the proper catcodes-1,2 are preserved?

2.5 Catcode 10 space tokens

Here we demonstrate that **tokcycle** can handle arbitrary redesignation of tokens to cat-10, as well as implicit space tokens (both implicit macro spaces and active-implicit character spaces).

While it should seem natural, we note that implicit space tokens are directed to the Space directive rather than the Character directive. However, `\ifimplicittok` may still be used to differentiate an explicit space from an implicit one.
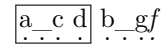
If the implicit space is *also* an active character, `\ifactivetok` is also set, for the user's benefit. Likewise, `\ifactivechar` is also checked, to see if the charcode of the active space is still, indeed, active. While `#1` may be used to represent the space in the space directive, a special technique may be required to recover the detokenized name of the active space character. In particular, if the active token is an implicit space, but the charcode of that token is no longer active, `#1` will, for this case, contain a generic implicit space token, `\tcsptoken`. However, for all active space tokens, the name of the associated active character in the input stream will be defined as a cat-12 token in `\theactivespace`.

Note in the following examples that cat-10 tokens do *not* get under-dots. The next three examples all use the same input, but with different catcode settings for the space and the underscore.

space cat-10, underscore cat-12

```
\catcode'\_=12 %
\catcode'\ =10 %

\tokencycle{\addcytoks{\underdot{#1}}}%
{\processtoks{#1}}%
{\addcytoks{#1}}%
{\addcytoks{#1}}%
\fbox{a_c d} b_g\itshape f\upshape\endtokencycle
```
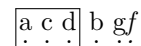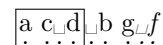
a_c d b_gf

space cat-10, underscore cat-10

```
\catcode'\_=10 %
\catcode'\ =10 %

\tokencycle{\addcytoks{\underdot{#1}}}%
{\processtoks{#1}}%
{\addcytoks{#1}}%
{\addcytoks{#1}}%
\fbox{a_c d} b_g\itshape f\upshape\endtokencycle
```
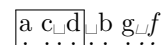
a c d b gf

space cat-12, underscore cat-10

```
\catcode'\_=10 %
\catcode'\ =12 %

\tokencycle{\addcytoks{\underdot{#1}}}%
{\processtoks{#1}}%
{\addcytoks{#1}}%
{\addcytoks{#1}}%
\fbox{a_c d} b_g\itshape f\upshape\endtokencycle
```

a c␣d␣b g␣f

The next two examples introduce implicit and active-implicit spaces. In the first example, functionally identical to the prior example, we see that tokcycle can digest and direct such tokens to the Space directive, here in the form of \z.

Implicit spaces also work

```
\resettokcycle
\Characterdirective{\addcytoks{\underdot{#1}}}
\def\:{\let\z= } \: %
\catcode'\_=10 %
\catcode'\ =12 %

\tokencyclexpress
\fbox{a\z{}c d} b_g\itshape f\upshape
\endtokencyclexpress
```

a c␣d␣b g␣f

In the second example, an active implicit space token is created, Q, and stored in a definition, \tmp. A \string is applied by the token cycle at one point to the space tokens and we see that the proper implicit tokens are decoded, whether they are active or macro.

The one case deserving special note is the final case of the example, in which the catcode of Q, the active character, is restored to a value of 12, before the previously tokenized, active-implicit Q is passed to the token cycle. In such a case, where future incantations of Q are no longer active, then

the active-implicit space token that gets passed to the Space directive (by way of `\tmp`) is taken as a generic active-implicit `\tcsptoken`, instead of the original active-implicit token (`Q`).

Nonetheless, the original string of the active-implicit token (that is, a cat-12 `Q`) is preserved in `\theactivespace`. The reason for this behavior is that the absorption of spaces in tokcycle is a destructive process, in which the spaces are absorbed as strings, so that a proper interpretation of their implicitness can be ascertained. With the character no longer active, it would be more difficult to recreate an active-implicit version of *that* token to pass to the Space directive. Thus, a generic active-implicit space token is, instead, employed.

Active Implicit spaces work, too

```
\resettokcycle
\def\:{\let\z= } \: %
\catcode`Q=\active %
\def\:{\let Q= } \: %
\catcode`\_=10 %

\def\tmp{xQx x_x\z{}x}
\expandafter\tokencyclexpress\tmp\endtokencyclexpress

\Spacedirective{\addcytoks{\textcolor{cyan}\bgroup}%
  \addcytoks[x]{(\string#1\ifactivetok
    \ifimplicittok\ifactivechar\else\noexpand
    \textcolor{red}{:\theactivespace}\fi\fi\fi)}%
  \addcytoks{\egroup}}

\medskip
\expandafter\tokencyclexpress\tmp\endtokencyclexpress

\medskip
\catcode`Q=12
\expandafter\tokencyclexpress\tmp\endtokencyclexpress
```

x x x x x

x(Q)x( )x( )x(\z)x

x(\tcsptoken:Q)x( )x( )x(\z)x

2.6   Changes to catcode 0

Cat-0 changes are not a hindrance to tokcycle

```
\let\littlet=t
\catcode`\! 0 !catcode`!\ 12
!Characterdirective{!ifimplicittok
  !addcytoks{!fbox{#1}}!else!ifcatSIX
  !addcytoks{!fbox{!fbox{#1}}}
  !else!addcytoks{#1}!fi!fi}
!tokencyclexpress Here, {!scshape!bgroup
  on !today!itshape{} we are !egroup
  !littlet es!littlet ing} cat-0
  changes{!bgroup}!egroup
!endtokencyclexpress!medskip

!detokenize!expandafter{!the!cytoks}
```

Here, on August 25, 2021 *we are* T̄ES̄T̄ING cat-0 changes

Here, {\scshape \bgroup on \today \itshape {} we are \egroup \fbox {\littlet }es\fbox {\littlet }ing} cat-0 changes{\bgroup }\egroup