The ted package

Manuel Pégourié-Gonnard
mpg@math.jussieu.fr

v1.01 (2007/12/12)

Contents

1 Introduction

Just like `sed` is a stream editor, `ted` is a token list editor. Actually, it is not as powerfull as `sed`, but its main feature is that it really works with tokens, not only characters. At the moment, it can do only two things with token lists: display it with full information on each token, and perform substitutions (that is, replacing every occurence of a sublist with another token list).

The `ted` package can perform substitutions inside groups, and don't forbid any token in the lists. Actually, `ted` is designed to work well even if strange tokens (that is, unusual (charcode, `\catcode`) pairs or tokens with a confusing `\meaning`) occur in the list.

2 Usage

The `ted` package provides two user macros: `\Substitute` and `\ShowTokens`. The first one is the primary goal of the package, but to be able to do the second was the more difficult and interesting part while writing the package. I made it into a user macro since I believe it can be useful for debugging things, or for learning TEX.

`\Substitute` The syntax of `\Substitute` is as follows.

`\Substitute`⟨*⟩`[`⟨*output*⟩`]{`⟨*input*⟩`}{`⟨*from*⟩`}{`⟨*to*⟩`}`

Let's begin with the basics. Without star or optional argument, the `\Substitute` macro will replace each occurence of the ⟨*from*⟩ token list with `{`⟨*to*⟩`}` in the ⟨*input*⟩, and put the result in the `\toks` register `\ted@toks`. This macro has a `@` in its name, but since I think the `\Substitute` macro will be essentially be used by class or package writers, this should be ok.

Anyway, if you don't like this name, you can specify another one as ⟨*output*⟩ using the optional argument. Your ⟨*output*⟩ should be the name of a `\toks` register. If you want the output to be put in a macro, use `\def\macro` (or `\long\def\macro` or...) as the optional argument. Anyway, ⟨*output*⟩`{`⟨*stuff*⟩`}` must be a legal syntax for an assignment: using `\macro` as optional argument will not work (and may actually result in chaos). Of course, if you want your output to be placed in a macro, it should not contain unproperly placed hash signs (that is, macro parameter tokens).

The one-starred form of `\Substitute` is meant to help you when your ⟨*input*⟩ is not an explicit token list, but the contents of either a macro or a `\toks` register, by expanding once its first mandatory argument before proceeding. It spares you the pain of using `\expandafter`s, especially in case you want to use the optional argument too. This time, things are reversed compared to the optional argument : using a macro instead of a toks register is easier. Actually, with the starred form, the first argument can be `\macro` or `\the\toksreg`, or anything whose one-time expansion is the token list you want `\Substitute` to act upon.

The two-starred form is also meant to avoid you trouble with development. It expands its three mandatory arguments once before executiing. The remark about macros and `\toks` register still holds. I hope this three cases (from zero to two stars) will suffice for most purpose. For a better handling of arguments expansion, wait for LaTeX3!

The action of `\Substitute` is pretty obvious most of the time. Maybe a particular case needs some precision: when ⟨*from*⟩ is empty, then the ⟨*to*⟩ list gets inserted between each two tokens of the `\input`, but not before the first one. For example, `\Substitute{abc}{}{1}` puts `a1b1c` in `\ted@toks`.

`\ShowTokens` The syntax of `\ShowTokens` is as follows.

`\ShowTokens`⟨`*`⟩`{`⟨*list*⟩`}`

In its simple form, `\ShowTokens` just shows the list, one token per line. For characters tokens, its prints the character, and its category code in human-friedly form (such as "blank space", "letter", etc.). For control sequences and active characters, it also prints (the beginning of) their current `\meaning` as a bonus.

`\ShowTokensLogonly` `\ShowTokensOnline` The default is to show this list both in the terminal and in the log file. If you don't want it to be printed on the terminal, just say `\ShowTokensLogonly`. If you change your mind latter, you can restore the default behaviour with `\ShowTokensOnline`.

The starred form of `\ShowTokens` works the same as for `\Substitute`: it expands its argument once before analysing and displaying it. The same remarks hold: use `\macro` or `\the\toksreg` in the argument.

I would like to conclude with the following remark: I have really tried to make sure `ted`'s macros will work fine even with the wierdest token list. In particular, you can freely use begin-group and end-group characters, hash signs, spaces, `\bgroup` and `\egroup`, `\par`, `\if`s, as well as exotic charcode-`\catcode` pairs in every argument of the macros. As far as I am aware, the only restriction is you should not use some[1] of the very private macros of `ted` (those beginning with `\ted@@`) in your token lists.

[1] Precisely, none of the control sequence in the token lists should be `\let`-equal to `\ted@@end`, `\ted@@special`, `\ted@@active` as defined below.

3 Implementation

A important problem, when trying to substitute things in token lists, is to handle begin-group and end-group tokens, since prevent us from to reading the tokens one by one, and tend to be difficult to handle individually. Two more kinds of tokens are special: the space tokens, since they[2] cannot be grabbed as the undelimited argument of a macro, and the parameter tokens (hash signs), since they cannot be part of the delimiters in the parameter text of a macro. From now on, "special tokens" thus denotes tokens with `\catcode` 1, 2, 6 or 10.

To get rid of these problems, the `\Substitute` command procedes in three steps. First, encode the input, replacing all special tokens with nice control sequences representing them, then do the actual substitution, and finally decode the output, replacing the special control sequences with the initial special tokens.

Encoding is the hard part. The idea is to try reading the tokens one by one; for this we have two means: using a macro with one undelimited argument, or something like `\let`. The former doesn't work well with `\catcode` 1, 2 or 10 tokens, and the later do not see the name of the token (its character code, or its name for a CS). So we need to use both `\futurelet`, a "grabbing" macro with argument, and `\string` in order to scan the tokens. Actually, the encoding procedes in two passes: in the first, we try and detect the special tokens, storing their character codes for later use, then do the the actual encoding in the last pass.

Decoding also processes the tokens one by one, and is simpler, since special cases are allready detected. There is, however, a trick with groups since, when we encouter a begin-group character, we have to wait for the corresponding end-group before adding the whole thing to the output. There is also a simpler version of decoding, for `\ShowTokens`, for screen/log output, with no need to use this trick, since it only outputs `\catcode`-12 charachters. Finally, the substitution part uses a macro with delimited argument, defined on the fly, using an idea of Jean-Côme Charpentier.

The code is divided as follows.

1. Encoding – (a) pre-scan (b) encode
2. Decoding
3. Substitution
4. Screen display for `\ShowTokens`
5. Definition of the user commands

`\ted@toks` `\ted@list` `\ted@code`

Before we begin, just allocate (or give a nice name to) a few registers.

```
\@ifdefinable\ted@toks{\newtoks\ted@toks}
\@ifdefinable\ted@list{\let\ted@list\toks@}
\@ifdefinable\ted@code{\let\ted@code\count@}
```

3.1 Encoding

`\ted@encloop` `\ted@encloop@`

The two passes use the same loop for reading the input allmost token by token. This loop grabs the next token through a `\futurelet`...

[2] Actually, only tokens with charcode 32 and `\catcode` 10 (i.e. 32_{10} tokens) are concerned. However, we will process all `\catcode`-10 tokens the same way.

```
\newcommand\ted@encloop{%
  \futurelet\@let@token
  \ted@encloop@}
```

. . . then looks at it with some `\ifx` and `\ifcat` (non nested, since the token could be an `\if` itself), in order to distinguish between three cases: normal token, end reached, or special token. In the later case, remember wich kind of special token it is, using a numeric code.

```
\newcommand\ted@encloop@{%
  \let\next\ted@do@normal
  \ifx\@let@token\ted@@end
    \let\next\ted@gobble@end
  \fi
  \ifcat\noexpand\@let@token##%
    \ted@code0
    \let\next\ted@do@special
  \fi
  \ifcat\noexpand\@let@token\@sptoken
    \ted@code1
    \let\next\ted@do@special
  \fi
  \ifcat\noexpand\@let@token\bgroup
    \ted@code2
    \let\next\ted@do@special
  \fi
  \ifcat\noexpand\@let@token\egroup
    \ted@code3
    \let\next\ted@do@special
  \fi
  \next}
```

`\ted@@end` `\ted@gobble@end`

Here we used the following to detect the end, then gobble it when reached.

```
\newcommand\ted@@end{\ted@@end@}
\@ifdefinable\ted@gobble@end{%
  \def\ted@gobble@end\ted@@end{}}
```

`\ted@sanitize` `\ted@@active`

Now, this detection method, with `\futurelet` and `\ifcat`, is unable to distinguish the following three cases for potential special tokens: (i) a "true" (explicit) special token, (ii) a CS `\let`-equal to a special token, (iii) an active character `\let`-equal to a special token. While this is pre-scanning's job to detect the (ii) case, the (iii) can be easily got rid of by redefining locally all active characters.

```
\count@\catcode\z@ \catcode\z@\active
\newcommand\ted@sanitize{%
  \count@\z@ \@whilenum\count@<\@cclvi \do{%
    \uccode\z@\count@
    \uppercase{\let^^00\ted@@active}%
    \advance\count@\@ne}}
\catcode\z@\count@
\newcommand\ted@@active{\ted@@active@}
```

This sanitizing macro also mark active characters by `\let`-ing them equal to `\ted@@active` in ordrer to detect them easily later, for exemple while displaying on-screen token analysis. All operations (scanning, replacing, display and decoding) are going to happen inside a group where `\ted@sanitize` has been executed, so that active characters are no longer an issue.

\ted@encode
\ted@do@normal
\ted@do@special

The \ted@encode macro is the master macro for encoding. It only initialise a few things and launches the two loops. We select one of the tree stpes by \let-ing \ted@do@normal and \ted@do@special to the appropriate action.

```
\newcommand\ted@encode[1]{%
  \ted@list{}%
  \let\ted@do@normal\ted@gobble@encloop
  \let\ted@do@special\ted@scan@special
  \ted@encloop#1\ted@@end
  \ted@toks{}%
  \let\ted@do@normal\ted@addtoks@encloop
  \let\ted@do@special\ted@special@out
  \ted@encloop#1\ted@@end
  \ted@assert@listempty}
```

\ted@assert@listempty

After the last loop, \ted@list should be empty. If it's not, it means something very weird happened during the encoding procedure. I hope the code below will never be executed :)

```
\newcommand\ted@assert@listempty{%
  \edef\next{\the\ted@list}%
  \ifx\next\@empty \else
    \PackageError{ted}{%
      Assertion `\string\ted@list\space is empty' failed}{%
      This should not happen. Please report this bug to the author.
      \MessageBreak By the way, you're in trouble there... I'm sorry.}%
  \fi}
```

a Pre-scanning

\ted@gobble@encloop

For normal tokens, things are pretty easy: just gobble them!

```
\newcommand\ted@gobble@encloop{%
  \afterassignment\ted@encloop
  \let\@let@token= }
```

\ted@scan@special

For special tokens, it's harder. We must distinguish explicit character tokens from control sequences \let-equal to special tokens. For this, we use \string, then grab the next character to see wether its code is \escapechar or not. Actually, things are not this easy, for two reasons. First, we have to make sure the next character's code is not allready \escapechar before the \string, by accident. For this purpose, we set \escapechar to 0 except if next character's code is also 0, in which case we prefer 1.

```
\count@\catcode\z@ \catcode\z@ 12
\newcommand\ted@scan@special{%
  \begingroup
  \escapechar\if\@let@token^^00 \@ne \else \z@ \fi
  \expandafter\ted@check@space\string}
\catcode\z@\count@
```

\ted@check@space
\ted@check@space@

Second, we have to handle carefully the case of the next token being the 32_{10} token, since we cannot grab this one with a macro. We are in this case if and only if the token we just \stringed was a character token with code 32, and it is enough to check if next token's \catcode is 10 in order to detect it, since it will be 12 otherwise. In order to check this, we use \futurelet again for pre-scanning.

```
\newcommand\ted@check@space{%
```

```
  \futurelet\@let@token
  \ted@check@space@}
\newcommand\ted@check@space@{%
  \ifcat\@let@token\@sptoken
    \endgroup
    \ted@addlist{32}%
    \expandafter\ted@gobble@encloop
  \else
    \expandafter\ted@list@special
  \fi}
```

`\ted@list@special` Now that we got rid of this nasty space problem, we know for sure that the next token has `\catcode` 12, so we can easily grab it as an argument, find its charcode, and decide wether the original token was a control sequence or not. Note the `\expandafter` over `\endgroup` trick, since we need to add the charcode to the list outside the group (opened for the modified `\escapechar`) though it was set inside.

```
\newcommand*\ted@list@special[1]{%
  \ted@code`#1\relax
  \expandafter\expandafter\expandafter
  \endgroup
  \ifnum\ted@code=\escapechar
    \ted@addlist{\m@ne}%
  \else
    \expandafter\ted@addlist\expandafter{\the\ted@code}%
  \fi
  \ted@encloop}
```

`\ted@addlist` Here we used the following macro to add an element to the list, which is space-separated.

```
\newcommand*\ted@addlist[1]{%
  \ted@list\expandafter{\the\ted@list#1 }}
```

b Actually encoding

Remember that, before this last encoding pass, `\ted@encode` did the following:

```
\let\ted@do@normal\ted@addtoks@encloop
\let\ted@do@special\ted@special@out
```

`\ted@addtoks@encloop` The first one is very easy : normal tokens are just grabbed as arguments and appended to the output, then the loop continues.

```
\newcommand\ted@addtoks@encloop[1]{%
  \ted@toks\expandafter{\the\ted@toks#1}%
  \ted@encloop}
```

`\ted@special@out` Special tokens need to be encoded, but before, just check if they are really special: they aren't if the corresponding code is -1.

```
\newcommand\ted@special@out{%
  \ifnum\ted@list@read=\m@ne
    \ted@list@advance
    \expandafter\ted@cs@clean
  \else
    \expandafter\ted@special@encode
  \fi}
```

\ted@cs@clean Even if the potentially special token was not a real one, we have work to do. Indeed, in the first pass we did break it using a \string, and thus we introduced some foreign tokens in the stream. Most of them are not important since they have \catcode 12. Anyway, some of them may be space tokens : in this case we have extra 32's in our list. So, we need to check this before going any further.

```
\newcommand\ted@cs@clean[1]{%
  \expandafter\ted@add@toks{#1}%
  \expandafter\ted@cscl@loop\string#1 \@nil}
```

\ted@cscl@loop We first add the CS to the output, then break it with a \string in order to look at its name with the folowing loop. It first grabs everything to the first space...

```
\@ifdefinable\ted@cscl@loop{%
  \def\ted@cscl@loop#1 {%
    \futurelet\@let@token
    \ted@cscl@loop@}}
```

\ted@cscl@loop@ ...and carefully look at the next token in order to know if we are finished or not.

```
\newcommand\ted@cscl@loop@{%
  \ifx\@let@token\@nil
    \expandafter\ted@gobble@encloop
  \else
    \ted@list@advance
    \expandafter\ted@cscl@loop
  \fi}
```

\ted@special@encode Now, let's come back to the special tokens. As we don't need the token to encode it (we allready know its \catcode from \ted@code, and its charcode is stored in the list), we first gobble it in order to prepare for next iteration.

```
\newcommand\ted@special@encode{%
    \afterassignment\ted@special@encode@
  \let\@let@token= }
```

\ted@special@encode@ Then we encode it in two steps : first, create a control sequence with name \ted@@⟨*code*⟩⟨*charcode*⟩, where code is a digit denoting[3] the \catcode of the special token, ...

```
\newcommand\ted@special@encode@{%
  \expandafter\ted@special@encode@@\expandafter{%
    \csname ted@@\the\ted@code\ted@list@read\endcsname}}
```

\ted@special@encode@@ \ted@@special ...then, mark this CS as a special token encoding, in order to make it easier to detect later, add it to the output and loop again.

```
\newcommand*\ted@special@encode@@[1]{%
  \ted@list@advance
  \let#1\ted@@special
  \ted@addtoks@encloop{#1}}

\newcommand\ted@@special{\ted@@special@}
```

\ted@list@read \ted@list@read@ Here we used the folowing macros in order to manage our charcode list. The reading one is fully expandable.

```
\newcommand\ted@list@read{%
  \expandafter\ted@list@read@\the\ted@list\@nil}
```

[3] I don't store the \catcode for two reasons : first, having a single digit is easier; second, having the true catcode would be useless (though it could maybe make the code more readable).

```
\@ifdefinable\ted@list@read@{%
  \def\ted@list@read@#1 #2\@nil{%
    #1}}
```

\ted@list@advance
\ted@list@advance@

Since it's expandable, it cannot change the list, so we need a separate macro to remove the first element from the list, once read.

```
\newcommand\ted@list@advance{%
  \expandafter\ted@list@advance@\the\ted@list\@nil}
\@ifdefinable\ted@list@advance@{
  \def\ted@list@advance@#1 #2\@nil{%
    \ted@list{#2}}}
```

3.2 Decoding

\ted@add@toks

Main decoding macro is `\ted@decode`. It is again a loop, processing the token list one by one. For normal tokens, things are easy as allways: just add them to the output, via

```
\newcommand\ted@add@toks[1]{%
  \ted@toks\expandafter{\the\ted@toks#1}}
```

\ted@decode

Encoded special tokens are easily recognized, since they were `\let` equal to `\ted@@special`. In order to decode it, we use the name of the CS. The following macro uses LaTeX-style `\if` in order to avoid potential nesting problems when `\ifs` are present in the token list being processed.

```
\newcommand\ted@decode[1]{%
  \ifx#1\ted@@end \expandafter\@gobble\else\expandafter\@firstofone\fi{%
    \ifx#1\ted@@special
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi{%
      \begingroup \escapechar\m@ne \expandafter\endgroup
      \expandafter\ted@decode@special\string#1\@nil
      }{%
      \ted@add@toks{#1}}%
    \ted@decode}}
```

\ted@decode@special

The next macro should then gobble the `ted@@` part of the CS name, and use the last part as two numeric codes (here we use the fact that the first one is only a digit).

```
\@ifdefinable\ted@decode@special{%
  \begingroup\escapechar\m@ne \expandafter\endgroup\expandafter
  \def\expandafter\ted@decode@special\string\ted@@#1#2\@nil{%
```

It then prodeces according to the first code, building back the original token and adding it to the output. The first two kinds of tokens (macro parameter characters and blank spaces) are easily dealt with.

```
    \ifcase#1
      \begingroup \uccode`##=#2 \uppercase{\endgroup
        \ted@add@toks{##}}%
    \or
      \begingroup \uccode32=#2 \uppercase{\endgroup
        \ted@add@toks{ }}%
    \or
```

For begin-group and end-group characters, we have a problem, since they are impossible to handle individually: we can only add a ⟨*balanced text*⟩ to the output. So, when we find a begin-group character, we just open a group (a real one), and start decoding again inside the group, until we find the correponding end-group character. Then, we enclose the local decoded list of tokens into the correct begin-group/end-group pair, and then add it to the output one group level below, using the `\expandafter`-over-`\endgroup` trick (essential here).

```
      \begingroup \ted@toks{}%
      \uccode`{=#2
    \or
      \uccode`}=#2
      \uppercase{\ted@toks\expandafter{\expandafter{\the\ted@toks}}}
      \expandafter\endgroup
      \expandafter\ted@add@toks\expandafter{\the\ted@toks}%
    \fi}}
```

3.3 Substitution

For this part, the idea[4] is to use a macro whose first argument is delimited with the ⟨*from*⟩ string, wich outputs the first argument followed by the ⟨*to*⟩ string, and loops. Obviously this macro has to be defined on the fly. All tokens lists need to be encoded first, and the output decoded at end. Since all this needs to happens inside a group (for `\ted@sanitize` and the marking up of special-charaters control sequences), remember to "export" `\ted@toks` when done.

`\ted@Substitude` The main substitution macro is as follows. Arguments are ⟨*input*⟩, ⟨*from*⟩, ⟨*to*⟩. `\ted@output` will be discussed later.

```
\newcommand\ted@Substitute[3]{%
  \begingroup \ted@sanitize
  \ted@encode{#3}%
  \expandafter\ted@def@subsmac\expandafter{\the\ted@toks}{#2}%
  \ted@encode{#1}%
  \ted@subsmac
  \ted@toks\expandafter{\expandafter}%
  \expandafter\ted@decode\the\ted@toks\ted@@end
  \expandafter\endgroup
  \expandafter\ted@output\expandafter{\the\ted@toks}}
```

`\ted@def@subsmac` The actual iterative substitution macro is defined by the folowing macro, whose arguments are the ⟨*to*⟩ string, encoded, and the plain ⟨*from*⟩ string.

```
\newcommand\ted@def@subsmac[2]{%
  \ted@encode{#2}%
  \long\expandafter\def\expandafter\ted@subsmac@loop
  \expandafter##\expandafter1\the\ted@toks##2{%
    \ted@add@toks{##1}%
    \ifx##2\ted@@end
      \expandafter\@gobble
    \else
      \expandafter\@firstofone
    \fi{%
      \ted@add@toks{#1}\ted@subsmac@loop##2}}%
```

[4] for which I am grateful to Jean-Côme Charpentier, who first taught me the clever use delimited arguments (and lots of other wonderful things) in `fr.comp.text.tex`

```
188   \expandafter\ted@def@subsmac@\expandafter{\the\ted@toks}}
```

`\ted@def@subsmac@` While we have the encoded ⟨*from*⟩ string at hand, define the start-loop macro.

```
189 \newcommand\ted@def@subsmac@[1]{%
190   \def\ted@subsmac{%
191     \ted@toks\expandafter{\expandafter}%
192     \expandafter\ted@subsmac@loop\the\ted@toks#1\ted@@end}}
```

3.4 Display

`\ted@ShowTokens` In order to display the tokens one by one, we first encode the string

```
193 \newcommand\ted@ShowTokens[1]{%
194   \begingroup \ted@sanitize
195   \ted@toks{#1}%
196   \ted@typeout{--- Begin token decompositon of:}%
197   \ted@typeout{\@spaces \the\ted@toks}%
198   \ted@encode{#1}%
199   \expandafter\ted@show@toks\the\ted@toks\ted@@end
200   \endgroup
201   \ted@typeout{--- End token decomposition.}}
```

`\ted@show@toks` Then we proceed, allmost like decoding, iteratively, processing the encoded tokens one by one. We detect control sequences the same way as in pre-scanning.

```
202 \count@\catcode\z@ \catcode\z@ 12
203 \newcommand\ted@show@toks[1]{%
204   \ifx#1\ted@@end \expandafter\@gobble\else\expandafter\@firstofone\fi{%
205     \ted@toks{#1}%
206     \begingroup
207     \escapechar\if\noexpand#1^^00 \@ne \else \z@ \fi
208     \expandafter\ted@show@toks@\string#1\@nil
209     \ted@show@toks}}
210 \catcode\z@\count@
```

`\ted@show@toks@` We stored the current token so that it can be used in the next macro, though previously broken by the `\string`, and moreover we can nest the `\if`s freely since it is hidden in a register (in case it would be a `\if` itself). The four cases are : CS representing a special token, normal CS, active character (since we cannot show its category with `\meaning`), and finally normal character token.

```
211 \@ifdefinable\ted@show@toks@{%
212   \long\def\ted@show@toks@#1#2\@nil{%
213     \expandafter\endgroup
214     \ifnum`#1=\escapechar
215       \expandafter\ifx\the\ted@toks\ted@@special
216         \ted@show@special#2\@nil
217       \else
```

It's time to think about the following: we are inside a group where all active characters were redefined, but we nonetheless want to display their meaning. In order to do this, the display need to actually happen after the current group is finished. For this we use `\aftergroup` (with specialized macro for displaying each kind of token).

```
218         \aftergroup\ted@type@cs
219         \expandafter\aftergroup\the\ted@toks
220       \fi
```

```
    \else \expandafter
      \ifx\the\ted@toks\ted@@active
        \aftergroup\ted@type@active
        \expandafter\aftergroup\the\ted@toks
      \else
        \aftergroup\ted@type@normal
        \expandafter\aftergroup\the\ted@toks
      \fi
    \fi}}
```

`\ted@show@special` Let's begin our tour of specialized display macro with the most important one: `\ted@show@special`. Displaying the special token goes mostly the same way as decoding them, but is far easier, since we don't need to care about groups: display is done with `\catcode` 12 characters.

```
\@ifdefinable\ted@show@special{%
  \begingroup\escapechar\m@ne \expandafter\endgroup
  \expandafter\def\expandafter\ted@show@special\string\ted@@#1#2\@nil{%
    \ifcase#1
      \aftergroup\ted@type@hash
    \or
      \aftergroup\ted@type@blank
    \or
      \aftergroup\ted@type@bgroup
    \or
      \aftergroup\ted@type@egroup
    \fi
    \begingroup \uccode`1#2
    \uppercase{\endgroup\aftergroup1}}}
```

`\ted@type@hash` `\ted@type@blank` `\ted@type@bgroup` `\ted@type@egroup` `\ted@type@normal` The four macros for special tokens are obvious. So is the macro for normal tokens. By the way, `\ted@typeout` will be discussed in the next section.

```
\newcommand\ted@type@hash[1]{%
  \ted@typeout{#1 (macro paramater character #1)}}
\newcommand\ted@type@blank[1]{%
  \ted@typeout{#1 (blank space #1)}}
\newcommand\ted@type@bgroup[1]{%
  \ted@typeout{#1 (begin-group character #1)}}
\newcommand\ted@type@egroup[1]{%
  \ted@typeout{#1 (end-group character #1)}}
\newcommand\ted@type@normal[1]{%
  \ted@typeout{#1 (\meaning#1)}}
```

`\ted@type@cs` `\ted@type@active` For control sequences and active characters, we use more sophisticated macros. Indeed, their `\meaning` can be quite long, and since it is not so important (ted's work is lexical analysis, displaying the `\meaning` is just an add-on), we cut it so that lines are shorter than 80 colons, in order to save our one-token-a-line presentation.

```
\newcommand\ted@type@cs[1]{%
  \ted@type@long{\string#1 (control sequence=\meaning#1}}%
\newcommand\ted@type@active[1]{%
  \ted@type@long{\string#1 (active character=\meaning#1}}%
```

\ted@type@long Lines are cut and displayed by \ted@type@long. This macro uses a loop, conting down how many columns remain on the current line. The input need to be fully expanded first, and the output is stored in \ted@toks.

```
\newcommand\ted@type@long[1]{%
  \ted@toks{}%
  \ted@code72
  \edef\next{#1}%
  \expandafter\ted@tl@loop\next\@nil}
```

\ted@tl@loop The only difficult thing in this loop is to take care of space tokens. For this we use again our \futurelet trick:

```
\newcommand\ted@tl@loop{%
  \futurelet\@let@token
  \ted@tl@loop@}
```

\ted@tl@loop@ ... then check what to do.

```
\newcommand\ted@tl@loop@{%
  \ifx\@let@token\@nil
    \let\next\ted@tl@finish
  \else
    \advance\ted@code\m@ne
    \ifnum\ted@code<\z@
      \let\next\ted@tl@finish
    \else
      \ifx\@let@token\@sptoken
        \let\next\ted@tl@space
      \else
        \let\next\ted@tl@add
      \fi
    \fi
  \fi
  \next}
```

\ted@tl@add \ted@tl@space Normal characters are just grabbed and added without care, and spaces are gobbled with a special macro which also add a space to the output.

```
\newcommand*\ted@tl@add[1]{%
  \ted@toks\expandafter{\the\ted@toks #1}%
  \ted@tl@loop}
```

```
\@ifdefinable\ted@tl@space{%
  \expandafter\def\expandafter\ted@tl@space\space{%
    \ted@tl@add{ }}}
```

\ted@tl@finish When the end has been reached (either because a \@nil was encountered or because the line is allmost full), it's time to actually display the result. We add \ETC. at the end when the full \meaning isn't displayed.

```
\@ifdefinable\ted@tl@finish{%
  \def\ted@tl@finish#1\@nil{%
    \ifnum\ted@code<\z@
      \ted@typeout{\the\ted@toks\string\ETC.)}
    \else
      \ted@typeout{\the\ted@toks)}
    \fi}}
```

3.5 User macros

\ted@typeout

Since we just discussed display, let's see the related user commands. Output is done with

```
\newcommand\ted@typeout{%
  \immediate\write\ted@outfile}
```

\ShowTokensOnline
\ShowTokensLogonly

allowing the user to choose between online display, or log output. Default is online.

```
\newcommand\ShowTokensOnline{%
  \let\ted@outfile\@unused}
\newcommand\ShowTokensLogonly{%
  \let\ted@outfile\m@ne}
\ShowTokensOnline
```

\ShowTokens
\ted@ShowTokens@exp

The user macro for showing tokens is a simple call to the internal macro, just expanding its argument once in its stared form.

```
\newcommand\ShowTokens{%
  \@ifstar{\ted@ShowTokens@exp}{\ted@ShowTokens}}
\newcommand\ted@ShowTokens@exp[1]{%
  \expandafter\ted@ShowTokens\expandafter{#1}}
```

\Substitute
\ted@Subs@star

Now, the user macro for substitution. First, check how many stars there are, if any, and set `\ted@subs@cmd` accordingly.

```
\newcommand\Substitute{%
  \@ifstar
  {\ted@Subs@star}
  {\let\ted@Subs@cmd\ted@Substitute \ted@Subs}}
\newcommand\ted@Subs@star{%
  \@ifstar
  {\let\ted@Subs@cmd\ted@Subs@exp@iii \ted@Subs}
  {\let\ted@Subs@cmd\ted@Subs@exp@i \ted@Subs}}
```

\ted@Subs@exp@i
\ted@Subs@exp@iii

Here are the intermediate macros that expand either the first or all three arguments before calling `\ted@Substitute`.

```
\newcommand\ted@Subs@exp@i{%
  \expandafter\ted@Substitute\expandafter}
\newcommand\ted@Subs@exp@iii[3]{%
  \begingroup
  \toks0{\ted@Substitute}%
  \toks2\expandafter{#1}%
  \toks4\expandafter{#2}%
  \toks6\expandafter{#3}%
  \xdef\ted@Subs@cmd{\the\toks0{\the\toks2}{\the\toks4}{\the\toks6}}%
  \endgroup
  \ted@Subs@cmd}
```

\ted@Subs

Now, the last macro checks and process the optional argument. Here we set `\ted@output`, which will be used at the end of `\ted@Substitute`.

```
\newcommand\ted@Subs[1][\ted@toks]{%
  \def\ted@output{#1}%
  \ted@Subs@cmd}
```

`\ted@output` Finally set a default `\ted@output` for advanced users who may want to use `\ted@Substitute` directly.

```
328 \let\ted@output\ted@toks
```

That's all folks!
Happy TEXing!