The gtl package: manipulate unbalanced lists of tokens[*]

Bruno Le Floch

2013/07/28

Contents

[*]This file has version number 0.0a, last revised 2013/07/28.

1 gtl documentation

The `expl3` programming language provides various tools to manipulate lists of tokens (package l3tl). However, those token lists must have balanced braces, or more precisely balanced begin-group and end-group characters. The gtl package manipulates instead lists of tokens which may be unbalanced, with more begin-group or more end-group characters.

1.1 Creating and initialising extended token lists

`\gtl_new:N`

`\gtl_new:N ⟨gtl var⟩`

Creates a new ⟨*gtl var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*gtl var*⟩ will initially be empty.

`\gtl_const:Nn`
`\gtl_const:Nx`

`\gtl_const:Nn ⟨gtl var⟩ {⟨token list⟩}`

Creates a new constant ⟨*gtl var*⟩ or raises an error if the name is already taken. The value of the ⟨*gtl var*⟩ will be set globally to the balanced ⟨*token list*⟩.

`\gtl_clear:N`
`\gtl_gclear:N`

`\gtl_clear:N ⟨gtl var⟩`

Empties the ⟨*gtl var*⟩, locally or globally.

`\gtl_clear_new:N`
`\gtl_gclear_new:N`

`\gtl_clear_new:N ⟨gtl var⟩`

Ensures that the ⟨*gtl var*⟩ exists globally by applying `\gtl_new:N` if necessary, then applies `\gtl_(g)clear:N` to leave the ⟨*gtl var*⟩ empty.

`\gtl_set_eq:NN`
`\gtl_gset_eq:NN`

`\gtl_set_eq:NN ⟨gtl var₁⟩ ⟨gtl var₂⟩`

Sets the content of ⟨*gtl var*$_1$⟩ equal to that of ⟨*gtl var*$_2$⟩.

`\gtl_concat:NNN`
`\gtl_gconcat:NNN`

`\gtl_concat:NNN ⟨gtl var₁⟩ ⟨gtl var₂⟩ ⟨gtl var₃⟩`

Concatenates the content of ⟨*gtl var*$_2$⟩ and ⟨*gtl var*$_3$⟩ together and saves the result in ⟨*gtl var*$_1$⟩. The ⟨*gtl var*$_2$⟩ will be placed at the left side of the new extended token list.

`\gtl_if_exist_p:N` ⋆
`\gtl_if_exist:NTF` ⋆

```
\gtl_if_exist_p:N ⟨gtl var⟩
\gtl_if_exist:NTF ⟨gtl var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests whether the ⟨*gtl var*⟩ is currently defined. This does not check that the ⟨*gtl var*⟩ really is an extended token list variable.

1.2 Adding data to token list variables

`\gtl_set:Nn`
`\gtl_set:Nx`
`\gtl_gset:Nn`
`\gtl_gset:Nx`

`\gtl_set:Nn ⟨gtl var⟩ {⟨token list⟩}`

Sets ⟨*gtl var*⟩ to contain the balanced ⟨*token list*⟩, removing any previous content from the variable.

`\gtl_put_left:Nn`
`\gtl_gput_left:Nn`

`\gtl_put_left:Nn ⟨gtl var⟩ {⟨token list⟩}`

Appends the balanced ⟨*token list*⟩ to the left side of the current content of ⟨*gtl var*⟩.

`\gtl_put_right:Nn`
`\gtl_gput_right:Nn`

`\gtl_put_right:Nn ⟨gtl var⟩ {⟨token list⟩}`

Appends the balanced ⟨*token list*⟩ to the right side of the current content of ⟨*gtl var*⟩.

1.3 Extended token list conditionals

`\gtl_if_blank_p:N` ⋆
`\gtl_if_blank:NTF` ⋆

```
\gtl_if_blank_p:N {⟨gtl var⟩}
\gtl_if_blank:NTF {⟨gtl var⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*gtl var*⟩ consists only of blank spaces. The test is `true` if ⟨*gtl var*⟩ consists of zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is `false` otherwise.

`\gtl_if_empty_p:N` ⋆
`\gtl_if_empty:NTF` ⋆

```
\gtl_if_empty_p:N ⟨gtl var⟩
\gtl_if_empty:NTF ⟨gtl var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*gtl var*⟩ is entirely empty (*i.e.* contains no tokens at all).

`\gtl_if_eq_p:NN` ⋆
`\gtl_if_eq:NNTF` ⋆

```
\gtl_if_eq_p:NN {⟨gtl var1⟩} {⟨gtl var2⟩}
\gtl_if_eq:NNTF {⟨gtl var1⟩} {⟨gtl var2⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if ⟨*gtl var*$_1$⟩ and ⟨*gtl var*$_2$⟩ have the same content. The test is `true` if the two contain the same list of tokens (identical in both character code and category code).

`\gtl_if_single_token_p:N` ⋆
`\gtl_if_single_token:NTF` ⋆

```
\gtl_if_single_token_p:N ⟨gtl var⟩
\gtl_if_single_token:NTF ⟨gtl var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the content of the ⟨*gtl var*⟩ consists of a single token. Such a token list has token count 1 according to `\gtl_count_tokens:N`.

`\gtl_if_tl_p:N` ⋆
`\gtl_if_tl:NTF` ⋆

```
\gtl_if_tl_p:N ⟨gtl var⟩
\gtl_if_tl:NTF ⟨gtl var⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*gtl var*⟩ is balanced.

1.4 The first token from an extended token list

`\gtl_head:N` ⋆

`\gtl_head:N` ⟨*gtl var*⟩

Leaves in the input stream the first token in the ⟨*gtl var*⟩. If the ⟨*gtl var*⟩ is empty, nothing is left in the input stream.

`\gtl_head_do:NN` ⋆

`\gtl_head_do:NN` ⟨*gtl var*⟩ ⟨*cs*⟩

Leaves in the input stream the ⟨*control sequence*⟩ followed by the first token in ⟨*gtl var*⟩. If the ⟨*gtl var*⟩ is empty, the ⟨*cs*⟩ is followed by `\q_no_value`.

`\gtl_get_left:NN`

`\gtl_get_left:NN` ⟨*gtl var*$_1$⟩ ⟨*gtl var*$_2$⟩

Stores the first token from ⟨*gtl var*$_1$⟩ in ⟨*gtl var*$_2$⟩ as an single-token extended token list, without removing it from ⟨*gtl var*$_1$⟩.

`\gtl_pop_left:N`
`\gtl_gpop_left:N`

`\gtl_pop_left:N` ⟨*gtl var*⟩

Remove the first token from ⟨*gtl var*$_1$⟩.

`\gtl_pop_left:NN`
`\gtl_gpop_left:NN`

`\gtl_pop_left:NN` ⟨*gtl var*$_1$⟩ ⟨*gtl var*$_2$⟩

Stores the first token from ⟨*gtl var*$_1$⟩ in ⟨*gtl var*$_2$⟩ as an single-token extended token list, and remove it from ⟨*gtl var*$_1$⟩.

`\gtl_if_head_eq_catcode_p:NN` ⋆
`\gtl_if_head_eq_catcode:NNTF` ⋆

```
\gtl_if_head_eq_catcode_p:NN {⟨gtl var⟩} ⟨test token⟩
\gtl_if_head_eq_catcode:NNTF {⟨gtl var⟩} ⟨test token⟩
  {⟨true code⟩} {⟨false code⟩}
```

Tests if the first token in ⟨*gtl var*⟩ has the same category code as the ⟨*test token*⟩. In the case where ⟨*gtl var*⟩ is empty, the test will always be `false`.

`\gtl_if_head_eq_charcode_p:NN` ⋆
`\gtl_if_head_eq_charcode:NNTF` ⋆

```
\gtl_if_head_eq_charcode_p:NN {⟨gtl var⟩} ⟨test token⟩
\gtl_if_head_eq_charcode:NNTF {⟨gtl var⟩} ⟨test token⟩
  {⟨true code⟩} {⟨false code⟩}
```

Tests if the first token in ⟨*gtl var*⟩ has the same character code as the ⟨*test token*⟩. In the case where ⟨*gtl var*⟩ is empty, the test will always be `false`.

`\gtl_if_head_eq_meaning_p:NN` ⋆
`\gtl_if_head_eq_meaning:NNTF` ⋆

```
\gtl_if_head_eq_meaning_p:NN {⟨gtl var⟩} ⟨test token⟩
\gtl_if_head_eq_meaning:NNTF {⟨gtl var⟩} ⟨test token⟩
  {⟨true code⟩} {⟨false code⟩}
```

Tests if the first token in ⟨*gtl var*⟩ has the same meaning as the ⟨*test token*⟩. In the case where ⟨*gtl var*⟩ is empty, the test will always be `false`.

```
\gtl_if_head_is_group_begin_p:N ⋆
\gtl_if_head_is_group_begin:NTF ⋆
\gtl_if_head_is_group_end_p:N   ⋆
\gtl_if_head_is_group_end:NTF   ⋆
\gtl_if_head_is_N_type_p:N      ⋆
\gtl_if_head_is_N_type:NTF      ⋆
\gtl_if_head_is_space_p:N       ⋆
\gtl_if_head_is_space:NTF       ⋆
```

```
\gtl_if_head_is_group_begin_p:N {⟨gtl var⟩}
\gtl_if_head_is_group_begin:NTF {⟨gtl var⟩}
  {⟨true code⟩} {⟨false code⟩}
```

Tests whether the first token in ⟨*gtl var*⟩ is an explicit begin-group character, an explicit end-group character, an `N`-type token, or a space. In the case where ⟨*gtl var*⟩ is empty, the test will always be `false`.

1.5 The first few tokens from an extended token list

```
\gtl_left_tl:N ⋆
```

```
\gtl_left_tl:N ⟨gtl var⟩
```

Leaves in the input stream all tokens in ⟨*gtl var*⟩ until the first extra begin-group or extra end-group character, within `\exp_not:n`. This is the longest balanced token list starting from the left of ⟨*gtl var*⟩.

```
\gtl_pop_left_tl:N
\gtl_gpop_left_tl:N
```

```
\gtl_pop_left_tl:N ⟨gtl var⟩
```

Remove from the ⟨*gtl var*⟩ all tokens before the first extra begin-group or extra end-group character. The tokens that are removed form the longest balanced token list starting from the left of ⟨*gtl var*⟩.

```
\gtl_left_item:NF ⋆
```

```
\gtl_left_item:NF ⟨gtl var⟩ {⟨false code⟩}
```

Leaves in the input stream the first ⟨*item*⟩ of the ⟨*gtl var*⟩: this is identical to `\tl_head:n` applied to the result of `\gtl_left_tl:N`. If there is no such item, the ⟨*false code*⟩ is left in the input stream.

```
\gtl_pop_left_item:NNTF
\gtl_gpop_left_item:NNTF
```

```
\gtl_pop_left_item:NNTF ⟨gtl var⟩ ⟨tl var⟩
{⟨true code⟩} {⟨false code⟩}
```

Stores the first item of ⟨*gtl var*⟩ in ⟨*tl var*⟩, locally, and removes it from ⟨*gtl var*⟩, together with any space before it. If there is no such item, the ⟨*gtl var*⟩ is not affected, and the metatl var may or may not be affected.

```
\gtl_left_text:NF ⋆
```

```
\gtl_left_text:NF ⟨gtl var⟩ {⟨false code⟩}
```

Starting from the first token in ⟨*gtl var*⟩, this function finds a pattern of the form ⟨*tokens*$_1$⟩ `{`⟨*tokens*$_2$⟩`}`, where the ⟨*tokens*$_1$⟩ contain no begin-group nor end-group characters, then leaves ⟨*tokens*$_1$⟩ `{`⟨*tokens*$_2$⟩`}` in the input stream, within `\exp_not:n`. If no such pattern exists (this happens if the result of `\gtl_left_tl:N` contains no brace group), the ⟨*false code*⟩ is run instead.

`\gtl_pop_left_text:N`
`\gtl_gpop_left_text:N`

`\gtl_pop_left_text:N` ⟨*gtl var*⟩

Starting from the first token in ⟨*gtl var*⟩, this function finds a pattern of the form ⟨*tokens*$_1$⟩ `{`⟨*tokens*$_2$⟩`}`, where the ⟨*tokens*$_1$⟩ contain no begin-group nor end-group characters, then removes ⟨*tokens*$_1$⟩ `{`⟨*tokens*$_2$⟩`}` from ⟨*gtl var*⟩. If no such pattern exists (this happens if the result of `\gtl_left_tl:N` contains no brace group), the ⟨*gtl var*⟩ is not modified instead.

1.6 Working with the contents of extended token lists

`\gtl_count_tokens:N` ⋆

`\gtl_count_tokens:N` ⟨*gtl var*⟩

Counts the number of tokens in the ⟨*gtl var*⟩ and leaves this information in the input stream.

`\gtl_extra_begin:N` ⋆
`\gtl_extra_end:N` ⋆

`\gtl_extra_begin:N` ⟨*gtl var*⟩

Counts the number of explicit extra begin-group (or end-group) characters in the ⟨*gtl var*⟩ and leaves this information in the input stream.

`\gtl_show:N`

`\gtl_show:N` ⟨*gtl var*⟩

Displays the content of the ⟨*gtl var*⟩ on the terminal.

`\gtl_to_str:N` ⋆

`\gtl_to_str:N` ⟨*gtl var*⟩

Converts the content of the ⟨*gtl var*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This string is then left in the input stream.

1.7 Constant extended token lists

`\c_empty_gtl`

Constant that is always empty.

`\c_group_begin_gtl`

An explicit begin-group character contained in an extended token list.

`\c_group_end_gtl`

An explicit end-group character contained in an extended token list.

1.8 Future perhaps

- Test if a token appears in an extended token list.
- Test if an extended token list appears in another.
- Remove an extended token list from another, once or every time it appears.

- Replace an extended token list by another in a third: once, or every time it appears.
- Case statement.
- Mapping?
- Inserting an extended token list into the input stream, with all its glorious unbalanced braces.
- Convert in various ways to a token list.
- Reverse the order of tokens.
- Extract a token given its position.
- Extract a range of tokens given their position.
- Trim spaces.
- Crazy idea below.

We could add (with lots of work) the expandable function For each triplet, this function builds the sub-token list of <tl_i> corresponding to the tokens ranging from position <start_i> to position <stop_i> of <tl_i>. The results obtained for each triplet are then concatenated. If nothing bad happens (see below), the concatenation is left in the input stream, and the <false code> is removed. Two cases can lead to running the <false code> (and dropping the first argument altogether). The first case is when the number of brace groups in `\gtl_concat:nF` is not a multiple of 3. The second case is when the concatenation gives rise to an unbalanced token list: then the result is not a valid token list. Note that each part is allowed to be unbalanced: only the full result must be balanced.

2 gtl implementation

Some support packages are loaded first, then we declare the package's name, date, version, and purpose.

```
⟨*package⟩
\RequirePackage{expl3}[2013/07/01]
\ProvidesExplPackage
  {gtl} {2013/07/28} {0.0a} {Manipulate unbalanced lists of tokens}
⟨@@=gtl⟩
```

2.1 Helpers

```
\cs_generate_variant:Nn \use:nn { no }
```

`\__gtl_exp_not_n:N` Used in one case where we need to prevent expansion of a token within an x-expanding definition. Using `\exp_not:N` there would fail when the argument is a macro parameter character.

```
\cs_new:Npn \__gtl_exp_not_n:N #1 { \exp_not:n {#1} }
```

(*End definition for* `\__gtl_exp_not_n:N`.)

`\__gtl_brace:nn`
`\__gtl_brace_swap:nn` Those functions are used to add some tokens, #1, to an item #2 in an extended token list: `\__gtl_brace:nn` adds tokens on the left, while `\__gtl_brace_swap:nn` adds them on the right.

```
\cs_new:Npn \__gtl_brace:nn #1#2 { { #1 #2 } }
\cs_new:Npn \__gtl_brace_swap:nn #1#2 { { #2 #1 } }
```

(*End definition for* `\__gtl_brace:nn` *and* `\__gtl_brace_swap:nn`.)

`\__gtl_strip_nil_mark:w`
`\__gtl_strip_nil_mark_aux:w` Removes the following `\q_nil \q_mark` without losing any braces, and places the result into `\exp_not:n`.

```
\cs_new_nopar:Npn \__gtl_strip_nil_mark:w
  { \__gtl_strip_nil_mark_aux:w \prg_do_nothing: }
\cs_new:Npn \__gtl_strip_nil_mark_aux:w #1 \q_nil \q_mark
  { \exp_not:o {#1} }
```

(*End definition for* `\__gtl_strip_nil_mark:w`. *This function is documented on page* **??**.)

2.2 Structure of extended token lists

Token lists must have balanced braces (or rather, begin-group and end-group characters). Extended token lists lift this requirement, and can represent arbitrary lists of tokens. A list of tokens can fail to be balanced in two ways: one may encounter too many end-group characters near the beginning of the list, or too many begin-group characters near the end of the list. In fact, a list of tokens always has the form

$$\langle b_1\rangle\ \texttt{\}}\ \ldots\ \langle b_n\rangle\ \texttt{\}}\ \langle c\rangle\ \texttt{\{}\ \langle e_1\rangle\ \ldots\ \texttt{\{}\ \langle e_p\rangle$$

where the $\langle b_i\rangle$, $\langle c\rangle$, and $\langle e_i\rangle$ are all balanced token lists. This can be seen by listing the tokens, and keeping track of a counter, which starts at 0, and is incremented at each begin-group character, and decremented at each end-group character: then the $\langle b_i\rangle$ are delimited by positions where the counter reaches a new minimum, whereas the $\langle e_i\rangle$ are delimited by positions where the counter last takes a given negative value. Such a token list is stored as

$$\texttt{\textbackslash s\_\_gtl \{ \{}\langle b_1\rangle\texttt{\}}\ \ldots\ \texttt{\{}\langle b_n\rangle\texttt{\} \} \{}\langle c\rangle\texttt{\} \{ \{}\langle e_p\rangle\texttt{\}}\ \ldots\ \texttt{\{}\langle e_1\rangle\texttt{\} \} \textbackslash s\_\_stop}$$

Note that the $\langle e_i\rangle$ are in a reversed order, as this makes the ends of extended token lists more accessible. Balanced token lists have $n = p = 0$: the first and third parts are empty, while the second contains the tokens.

\s__gtl This marker appears at the start of extended token lists.

```
\__scan_new:N \s__gtl
```

(*End definition for* \s__gtl. *This variable is documented on page* ??.)

\gtl_set:Nn
\gtl_gset:Nn
\gtl_const:Nn
\gtl_set:Nx
\gtl_gset:Nx
\gtl_const:Nx

Storing a balanced token list into an extended token list variable simply means adding \s__gtl, \s__stop, and two empty brace groups.

```
\cs_new_protected_nopar:Npn \gtl_set:Nn   { \__gtl_set:NNn \tl_set:Nn   }
\cs_new_protected_nopar:Npn \gtl_gset:Nn  { \__gtl_set:NNn \tl_gset:Nn  }
\cs_new_protected_nopar:Npn \gtl_const:Nn { \__gtl_set:NNn \tl_const:Nn }
\cs_new_protected_nopar:Npn \gtl_set:Nx   { \__gtl_set:NNn \tl_set:Nx   }
\cs_new_protected_nopar:Npn \gtl_gset:Nx  { \__gtl_set:NNn \tl_gset:Nx  }
\cs_new_protected_nopar:Npn \gtl_const:Nx { \__gtl_set:NNn \tl_const:Nx }
\cs_new_protected:Npn \__gtl_set:NNn   #1#2#3
  { #1 #2 { \s__gtl { } {#3} { } \s__stop } }
```

(*End definition for* \gtl_set:Nn *and others. These functions are documented on page* ??.)

\c_empty_gtl An empty extended token list, obtained thanks to the \gtl_const:Nn function just defined.

```
\gtl_const:Nn \c_empty_gtl { }
```

(*End definition for* \c_empty_gtl. *This variable is documented on page* 6.)

\c_group_begin_gtl
\c_group_end_gtl

An extended token list with exactly one begin-group/end-group character.

```
\tl_const:Nn \c_group_end_gtl   { \s__gtl { { } } { } {     } \s__stop }
\tl_const:Nn \c_group_begin_gtl { \s__gtl {     } { } { { } } \s__stop }
```

(*End definition for* \c_group_begin_gtl *and* \c_group_end_gtl. *These variables are documented on page* 6.)

2.3 Creating extended token list variables

\gtl_new:N A new extended token list is created empty.

```
\cs_new_protected:Npn \gtl_new:N #1
  { \cs_new_eq:NN #1 \c_empty_gtl }
```

(*End definition for* \gtl_new:N. *This function is documented on page* 2.)

\gtl_set_eq:NN
\gtl_gset_eq:NN

All the data about an extended token list is stored as a single token list, so copying is easy.

```
\cs_new_eq:NN \gtl_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \gtl_gset_eq:NN \tl_gset_eq:NN
```

(*End definition for* \gtl_set_eq:NN *and* \gtl_gset_eq:NN. *These functions are documented on page* 2.)

\gtl_clear:N
\gtl_gclear:N

Clearing an extended token list by setting it to the empty one.

```
\cs_new_protected:Npn \gtl_clear:N  #1
  { \gtl_set_eq:NN  #1 \c_empty_gtl }
\cs_new_protected:Npn \gtl_gclear:N #1
  { \gtl_gset_eq:NN #1 \c_empty_gtl }
```

(*End definition for* \gtl_clear:N *and* \gtl_gclear:N. *These functions are documented on page* 2.)

\gtl_clear_new:N
\gtl_gclear_new:N

If the variable exists, clear it. Otherwise declare it.

```
\cs_new_protected:Npn \gtl_clear_new:N #1
  { \gtl_if_exist:NTF #1 { \gtl_clear:N #1 } { \gtl_new:N #1 } }
\cs_new_protected:Npn \gtl_gclear_new:N #1
  { \gtl_if_exist:NTF #1 { \gtl_gclear:N #1 } { \gtl_new:N #1 } }
```

(*End definition for* `\gtl_clear_new:N` *and* `\gtl_gclear_new:N`*. These functions are documented on page 2.*)

\gtl_if_exist_p:N
\gtl_if_exist:N*TF*

Again a copy of token list functions.

```
\prg_new_eq_conditional:NNn \gtl_if_exist:N \tl_if_exist:N
  { p , T , F , TF }
```

(*End definition for* `\gtl_if_exist:N`*. These functions are documented on page 2.*)

2.4 Adding data to extended token list variables

\gtl_put_left:Nn
\gtl_gput_left:Nn
__gtl_put_left:wn

```
\cs_new_protected:Npn \gtl_put_left:Nn #1#2
  { \tl_set:Nx #1 { \exp_after:wN \__gtl_put_left:wn #1 {#2} } }
\cs_new_protected:Npn \gtl_gput_left:Nn #1#2
  { \tl_gset:Nx #1 { \exp_after:wN \__gtl_put_left:wn #1 {#2} } }
\cs_new:Npn \__gtl_put_left:wn \s__gtl #1#2#3 \s__stop #4
  {
    \tl_if_empty:nTF {#1}
      { \exp_not:n { \s__gtl { } { #4 #2 } {#3} \s__stop } }
      {
        \s__gtl
        { \exp_not:o { \__gtl_brace:nn {#4} #1 } }
        { \exp_not:n {#2} }
        { \exp_not:n {#3} }
        \s__stop
      }
  }
```

(*End definition for* `\gtl_put_left:Nn` *and* `\gtl_gput_left:Nn`*. These functions are documented on page 3.*)

\gtl_put_right:Nn
\gtl_gput_right:Nn
__gtl_put_right:wn

```
\cs_new_protected:Npn \gtl_put_right:Nn #1#2
  { \tl_set:Nx #1 { \exp_after:wN \__gtl_put_right:wn #1 {#2} } }
\cs_new_protected:Npn \gtl_gput_right:Nn #1#2
  { \tl_gset:Nx #1 { \exp_after:wN \__gtl_put_right:wn #1 {#2} } }
\cs_new:Npn \__gtl_put_right:wn \s__gtl #1#2#3 \s__stop #4
  {
    \tl_if_empty:nTF {#3}
      { \exp_not:n { \s__gtl {#1} { #2 #4 } { } \s__stop } }
      {
        \s__gtl
        { \exp_not:n {#1} }
        { \exp_not:n {#2} }
        { \exp_not:o { \__gtl_brace_swap:nn {#4} #3 } }
        \s__stop
      }
  }
```

(*End definition for* `\gtl_put_right:Nn` *and* `\gtl_gput_right:Nn`*. These functions are documented on page 3.*)

`\gtl_concat:NNN`
`\gtl_gconcat:NNN`
`\__gtl_concat:ww`
`\__gtl_concat_aux:nnnnnn`
`\__gtl_concat_auxi:nnnnnn`
`\__gtl_concat_auxii:nnnnnn`
`\__gtl_concat_auxiii:w`
`\__gtl_concat_auxiv:nnnn`
`\__gtl_concat_auxv:wnwnn`
`\__gtl_concat_auxvi:nnwnwnn`

Concatenating two lists of tokens of the form

$$\texttt{\textbackslash s\_\_gtl \{ \{}\langle b_1\rangle\texttt{\} ... \{}\langle b_n\rangle\texttt{\} \} \{}\langle c\rangle\texttt{\} \{ \{}\langle e_p\rangle\texttt{\} ... \{}\langle e_1\rangle\texttt{\} \} \textbackslash s\_\_stop}$$

is not an easy task. The $\langle e\rangle$ parts of the first join with the $\langle b\rangle$ parts of the second to make balanced pairs, and the follow-up depends on whether there were more $\langle e\rangle$ parts or more $\langle b\rangle$ parts.

```
\cs_new_protected:Npn \gtl_concat:NNN #1#2#3
  { \tl_set:Nx  #1 { \exp_last_two_unbraced:Noo \__gtl_concat:ww #2 #3 } }
\cs_new_protected:Npn \gtl_gconcat:NNN #1#2#3
  { \tl_gset:Nx #1 { \exp_last_two_unbraced:Noo \__gtl_concat:ww #2 #3 } }
\cs_new:Npn \__gtl_concat:ww \s__gtl #1#2#3 \s__stop \s__gtl #4#5#6 \s__stop
  {
    \tl_if_blank:nTF {#3}
      {
        \tl_if_blank:nTF {#4}
          { \__gtl_concat_aux:nnnnnn }
          { \__gtl_concat_auxi:nnnnnn }
      }
      {
        \tl_if_blank:nTF {#4}
          { \__gtl_concat_auxii:nnnnnn }
          { \__gtl_concat_auxiv:nnnn }
      }
      {#1} {#2} {#3} {#4} {#5} {#6}
    \s__stop
  }
\cs_new:Npn \__gtl_concat_aux:nnnnnn #1#2#3#4#5#6
  { \exp_not:n { \s__gtl {#1} { #2 #5 } {#6} } }
\cs_new:Npn \__gtl_concat_auxi:nnnnnn #1#2#3#4#5#6
  {
    \s__gtl
    {
      \exp_not:n {#1}
      \exp_not:f
        { \__gtl_concat_auxiii:w \__gtl_brace:nn {#2} #4 ~ \q_stop }
    }
    { \exp_not:n {#5} }
    { \exp_not:n {#6} }
  }
\cs_new:Npn \__gtl_concat_auxii:nnnnnn #1#2#3#4#5#6
  {
    \s__gtl
    { \exp_not:n {#1} }
    { \exp_not:n {#2} }
    {
      \exp_not:n {#6}
      \exp_not:f
        { \__gtl_concat_auxiii:w \__gtl_brace_swap:nn {#5} #3 ~ \q_stop }
    }
  }
\cs_new:Npn \__gtl_concat_auxiii:w #1 ~ #2 \q_stop {#1}
\cs_new:Npn \__gtl_concat_auxiv:nnnn #1#2#3#4
  {
    \tl_if_single:nTF {#3}
      { \__gtl_concat_auxv:wnwnn }
      { \__gtl_concat_auxvi:nnwnwnn }
    #3 ~ \q_mark #4 ~ \q_mark {#1} {#2}
  }
\cs_new:Npn \__gtl_concat_auxv:wnwnn
    #1#2 \q_mark #3#4 \q_mark #5#6
  {
    \__gtl_concat:ww
      \s__gtl {#5} { #6 { #1 #3 } } { } \s__stop
      \s__gtl {#4}
  }
\cs_new:Npn \__gtl_concat_auxvi:nnwnwnn
    #1#2#3 \q_mark #4#5 \q_mark #6#7
  {
    \__gtl_concat:ww
      \s__gtl {#6} {#7} { { #2 { #1 #4 } } #3 } \s__stop
      \s__gtl {#5}
  }
```

(*End definition for* `\gtl_concat:NNN` *and* `\gtl_gconcat:NNN`*. These functions are documented on page 2.*)

2.5 Showing extended token lists

`\gtl_to_str:N`
`\gtl_to_str:n`
`\__gtl_to_str:w`
`\__gtl_to_str_loopi:nnw`
`\__gtl_to_str_testi:nnw`
`\__gtl_to_str_endi:nnn`
`\__gtl_to_str_loopii:nnw`
`\__gtl_to_str_endii:nnw`

```
\cs_new:Npn \gtl_to_str:N #1 { \exp_after:wN \__gtl_to_str:w #1 }
\cs_new:Npn \gtl_to_str:n #1 { \__gtl_to_str:w #1 }
\cs_new:Npn \__gtl_to_str:w \s__gtl #1#2#3 \s__stop
  { \__gtl_to_str_loopi:nnw { } #1 \q_nil \q_mark {#2} {#3} }
\cs_new:Npx \__gtl_to_str_loopi:nnw #1#2
  {
    \exp_not:N \quark_if_nil:nTF {#2}
      { \exp_not:N \__gtl_to_str_testi:nnw {#1} {#2} }
      { \exp_not:N \__gtl_to_str_loopi:nnw { #1 #2 \iow_char:N \} } }
  }
\cs_new:Npx \__gtl_to_str_testi:nnw #1#2#3 \q_mark
  {
    \exp_not:N \tl_if_empty:nTF {#3}
      { \exp_not:N \__gtl_to_str_endi:nnn {#1} }
      {
        \exp_not:N \__gtl_to_str_loopi:nnw
          { #1 #2 \iow_char:N \} } #3 \exp_not:N \q_mark
      }
  }
\cs_new:Npn \__gtl_to_str_endi:nnn #1#2#3
  { \__gtl_to_str_loopii:nnw #3 { #1 #2 } \q_nil \q_stop }
\cs_new:Npx \__gtl_to_str_loopii:nnw #1#2
  {
    \exp_not:N \quark_if_nil:nTF {#2}
      { \exp_not:N \__gtl_to_str_testii:nnw {#1} {#2} }
      { \exp_not:N \__gtl_to_str_loopii:nnw { #2 \iow_char:N \{ #1 } }
  }
\cs_new:Npx \__gtl_to_str_testii:nnw #1#2#3 \q_stop
  {
    \exp_not:N \tl_if_empty:nTF {#3}
      { \exp_not:N \tl_to_str:n {#1} }
      {
        \exp_not:N \__gtl_to_str_loopii:nnw
          { #2 \iow_char:N \{ #1 } #3 \exp_not:N \q_stop
      }
  }
```

(*End definition for* `\gtl_to_str:N` *and* `\gtl_to_str:n`*. These functions are documented on page* **??**.)

`\gtl_show:N` Display the variable name, then its string representation.

```
\cs_new_protected:Npn \gtl_show:N #1
  { \exp_args:Nx \tl_show:n { \token_to_str:N #1 = \gtl_to_str:N #1 } }
```

(*End definition for* `\gtl_show:N`*. This function is documented on page 6.*)

2.6 Extended token list conditionals

`\gtl_if_eq_p:NN`
`\gtl_if_eq:NNTF`

Two extended token lists are equal if their contents agree.

```
\prg_new_conditional:Npnn \gtl_if_eq:NN #1#2 { p , T , F , TF }
  { \tl_if_eq:NNTF #1 #2 { \prg_return_true: } { \prg_return_false: } }
```

(*End definition for* `\gtl_if_eq:NN`*. These functions are documented on page 3.*)

`\gtl_if_empty_p:N`
`\gtl_if_empty:NTF`

An extended token list is empty if it is equal to the empty one.

```
\prg_new_conditional:Npnn \gtl_if_empty:N #1 { p , T , F , TF }
  {
    \tl_if_eq:NNTF #1 \c_empty_gtl
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End definition for* `\gtl_if_empty:N`*. These functions are documented on page 3.*)

`\gtl_if_tl_p:N`
`\gtl_if_tl:NTF`
`\__gtl_if_tl_return:w`

```
\prg_new_conditional:Npnn \gtl_if_tl:N #1 { p , T , F , TF }
  { \exp_after:wN \__gtl_if_tl_return:w #1 }
\cs_new:Npn \__gtl_if_tl_return:w \s__gtl #1#2#3 \s__stop
```

```
  {
    \tl_if_empty:nTF { #1 #3 }
      { \prg_return_true: } { \prg_return_false: }
  }
```
(*End definition for* `\gtl_if_tl:N`*. These functions are documented on page 3.*)

\gtl_if_single_token_p:N
\gtl_if_single_token:NTF
__gtl_if_single_token_return:w

```
\prg_new_conditional:Npnn \gtl_if_single_token:N #1 { p , T , F , TF }
  { \exp_after:wN \__gtl_if_single_token_return:w #1 }
\cs_new:Npn \__gtl_if_single_token_return:w \s__gtl #1#2#3 \s__stop
  {
    \bool_if:nTF
      {
        \tl_if_empty_p:n {#2}
        && \tl_if_single_p:n { #1 #3 }
        && \tl_if_empty_p:o { \use:n #1 #3 }
        ||
        \tl_if_single_token_p:n {#2}
        && \tl_if_empty_p:n { #1 #3 }
      }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```
(*End definition for* `\gtl_if_single_token:N`*. These functions are documented on page 3.*)

\gtl_if_blank_p:N
\gtl_if_blank:NTF
__gtl_if_blank_return:w

```
\prg_new_conditional:Npnn \gtl_if_blank:N #1 { p , T , F , TF }
  { \exp_after:wN \__gtl_if_blank_return:w #1 }
\cs_new:Npn \__gtl_if_blank_return:w \s__gtl #1#2#3 \s__stop
  {
    \tl_if_blank:nTF { #1 #2 #3 }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```
(*End definition for* `\gtl_if_blank:N`*. These functions are documented on page 3.*)

\gtl_if_head_is_group_begin_p:N
\gtl_if_head_is_group_end_p:N
\gtl_if_head_is_space_p:N
\gtl_if_head_is_N_type_p:N
\gtl_if_head_is_group_begin:NTF
\gtl_if_head_is_group_end:NTF
\gtl_if_head_is_space:NTF
\gtl_if_head_is_N_type:NTF

```
\prg_new_conditional:Npnn \gtl_if_head_is_group_begin:N #1
  { p , T , F , TF }
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { \prg_return_false: }
      { \prg_return_true: }
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_false: \use_none:n }
  }
\prg_new_conditional:Npnn \gtl_if_head_is_group_end:N #1
  { p , T , F , TF }
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_true: }
      { \prg_return_false: }
      { \prg_return_false: \use_none:n }
  }
\prg_new_conditional:Npnn \gtl_if_head_is_space:N #1
  { p , T , F , TF }
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_true: }
      { \prg_return_false: \use_none:n }
  }
\prg_new_conditional:Npnn \gtl_if_head_is_N_type:N #1
  { p , T , F , TF }
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_false: }
      { \prg_return_true: \use_none:n }
  }
```

(*End definition for* `\gtl_if_head_is_group_begin:N` *and others. These functions are documented on page 5.*)

`\gtl_if_head_eq_catcode_p:NN`
`\gtl_if_head_eq_catcode:NNTF`
`\gtl_if_head_eq_charcode_p:NN`
`\gtl_if_head_eq_charcode:NNTF`
`\__gtl_if_head_eq_code_return:NNN`

In the empty case, `?` can match with `#2`, but then `\use_none:nn` gets rid of `\prg_-return_true:` and `\else:`, to correctly leave `\prg_return_false:`. We could not simplify this by placing the `\exp_not:N #2` after the construction involving `#1`, because `#2` must be taken into the TeX primitive test, in case `#2` itself is a primitive TeX conditional, which would mess up conditional nesting.

```
\prg_new_conditional:Npnn \gtl_if_head_eq_catcode:NN #1#2
  { p , T , F , TF }
  { \__gtl_if_head_eq_code_return:NNN \if_catcode:w #1#2 }
\prg_new_conditional:Npnn \gtl_if_head_eq_charcode:NN #1#2
  { p , T , F , TF }
  { \__gtl_if_head_eq_code_return:NNN \if_charcode:w #1#2 }
\cs_new:Npn \__gtl_if_head_eq_code_return:NNN #1#2#3
  {
    #1
        \exp_not:N #3
        \exp_after:wN \__gtl_head:wnnnnn #2
          { ? \use_none:nn }
          { \c_group_begin_token }
          { \c_group_end_token }
          { \c_space_token }
          { \exp_not:N }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\gtl_if_head_eq_catcode:NN`. *These functions are documented on page 4.*)

`\gtl_if_head_eq_meaning_p:NN`
`\gtl_if_head_eq_meaning:NN`*TF*
`\__gtl_if_head_eq_meaning_return:NN`

```
\prg_new_conditional:Npnn \gtl_if_head_eq_meaning:NN #1#2
  { p , T , F , TF }
  { \__gtl_if_head_eq_meaning_return:NN #1#2 }
\cs_new:Npn \__gtl_if_head_eq_meaning_return:NN #1#2
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { \if_false: }
      { \if_meaning:w #2 \c_group_begin_token }
      { \if_meaning:w #2 \c_group_end_token }
      { \if_meaning:w #2 \c_space_token }
      { \if_meaning:w #2 }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\gtl_if_head_eq_meaning:NN`. *These functions are documented on page 4.*)

2.7 First token of an extended token list

`\__gtl_head:wnnnnn`
`\__gtl_head_aux:nwnnnn`
`\__gtl_head_auxii:N`
`\__gtl_head_auxiii:Nnn`

This function performs `#4` if the `gtl` is empty, `#5` if it starts with a begin-group character, `#6` if it starts with an end-group character, `#7` if it starts with a space, and in other cases (when the first token is `N`-type), it performs `#8` followed by the first token.

```
\cs_new:Npn \__gtl_head:wnnnnn \s__gtl #1#2#3 \s__stop #4#5#6#7#8
  {
    \tl_if_empty:nTF {#1}
      {
        \tl_if_empty:nTF {#2}
          { \tl_if_empty:nTF {#3} {#4} {#5} }
          { \__gtl_head_aux:nwnnnn {#2} \q_stop {#5} {#6} {#7} {#8} }
      }
      { \__gtl_head_aux:nwnnnn #1 \q_stop {#5} {#6} {#7} {#8} }
  }
\cs_new:Npn \__gtl_head_aux:nwnnnn #1#2 \q_stop #3#4#5#6
  {
    \tl_if_head_is_group:nTF {#1} {#3}
      {
        \tl_if_empty:nTF {#1} {#4}
```

```
          {
            \tl_if_head_is_space:nTF {#1} {#5}
              { \if_false: { \fi: \__gtl_head_auxii:N #1 } {#6} }
          }
      }
  }
\cs_new:Npn \__gtl_head_auxii:N #1
  {
    \exp_after:wN \__gtl_head_auxiii:Nnn
    \exp_after:wN #1
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new:Npn \__gtl_head_auxiii:Nnn #1#2#3 { #3 #1 }
```

(*End definition for* `\__gtl_head:wnnnnn`. *This function is documented on page* ??.)

`\gtl_head:N` If `#1` is empty, do nothing. If it starts with a begin-group character or an end-group character leave the appropriate brace (thanks to `\if_false:` tricks). If it starts with a space, leave that, and finally if it starts with a normal token, leave it, within `\exp_not:n`.

```
\cs_new:Npn \gtl_head:N #1
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { }
      { \exp_after:wN { \if_false: } \fi: }
      { \if_false: { \fi: } }
      { ~ }
      { \__gtl_exp_not_n:N }
  }
```

(*End definition for* `\gtl_head:N`. *This function is documented on page 4.*)

`\gtl_head_do:NN` Similar to `\gtl_head:N`, but inserting `#2` before the resulting token.

```
\cs_new:Npn \gtl_head_do:NN #1#2
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { #2 \q_no_value }
      { \exp_after:wN #2 \exp_after:wN { \if_false: } \fi: }
      { \if_false: { \fi: #2 } }
      { #2 ~ }
      { #2 }
  }
```

(*End definition for* `\gtl_head_do:NN`. *This function is documented on page 4.*)

`\gtl_get_left:NN`

```
\cs_new_protected:Npn \gtl_get_left:NN #1#2
  {
    \exp_after:wN \__gtl_head:wnnnnn #1
      { \gtl_set:Nn #2 { \q_no_value } }
      { \gtl_set_eq:NN #2 \c_group_begin_gtl }
      { \gtl_set_eq:NN #2 \c_group_end_gtl }
      { \gtl_set:Nn #2 { ~ } }
      { \gtl_set:Nn #2 }
  }
```

(*End definition for* `\gtl_get_left:NN`*. This function is documented on page 4.*)

`\gtl_pop_left:N`
`\gtl_gpop_left:N`
`\__gtl_pop_left:w`
`\__gtl_pop_left_auxi:n`
`\__gtl_pop_left_auxii:nnnw`
`\__gtl_pop_left_auxiii:nnnw`
`\__gtl_pop_left_auxiv:nn`
`\__gtl_pop_left_auxv:nnn`
`\__gtl_pop_left_auxvi:n`

```
\cs_new_protected:Npn \gtl_pop_left:N #1
  { \tl_set:Nx #1 { \exp_after:wN \__gtl_pop_left:w #1 } }
\cs_new_protected:Npn \gtl_gpop_left:N #1
  { \tl_gset:Nx #1 { \exp_after:wN \__gtl_pop_left:w #1 } }
\cs_new:Npn \__gtl_pop_left:w \s__gtl #1#2#3 \s__stop
  {
    \tl_if_empty:nTF {#1}
      {
        \tl_if_empty:nTF {#2}
          { \__gtl_pop_left_auxi:n {#3} }
          { \__gtl_pop_left_auxiv:nn {#2} {#3} }
      }
      { \__gtl_pop_left_auxv:nnn {#1} {#2} {#3} }
  }
\cs_new:Npn \__gtl_pop_left_auxi:n #1
  {
    \s__gtl
    { }
    \__gtl_pop_left_auxii:nnnw { } { } #1 \q_nil \q_stop
    \s__stop
  }
\cs_new:Npn \__gtl_pop_left_auxii:nnnw #1#2#3
  {
    \quark_if_nil:nTF {#3}
      { \__gtl_pop_left_auxiii:nnnw {#1} {#2} {#3} }
      { \__gtl_pop_left_auxii:nnnw { #1 #2 } { {#3} } }
  }
\cs_new:Npn \__gtl_pop_left_auxiii:nnnw #1#2#3#4 \q_stop
  {
    \tl_if_empty:nTF {#4}
      { \exp_not:n { #2 {#1} } }
      { \__gtl_pop_left_auxii:nnnw { #1 #2 } { {#3} } }
  }
\cs_new:Npn \__gtl_pop_left_auxiv:nn #1#2
  {
    \s__gtl
    { \tl_if_head_is_group:nT {#1} { { \tl_head:n {#1} } } }
    { \tl_if_head_is_space:nTF {#1} { \exp_not:f } { \tl_tail:n } {#1} }
    { \exp_not:n {#2} }
    \s__stop
  }
\cs_new:Npn \__gtl_pop_left_auxv:nnn #1#2#3
  {
    \s__gtl
    { \if_false: { \fi: \__gtl_pop_left_auxvi:n #1 } }
    { \exp_not:n {#2} }
    { \exp_not:n {#3} }
    \s__stop
  }
\cs_new:Npn \__gtl_pop_left_auxvi:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \tl_if_head_is_group:nT {#1} { { \tl_head:n {#1} } }
        {
          \tl_if_head_is_space:nTF {#1}
            { \exp_not:f } { \tl_tail:n } {#1}
        }
      }
    \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi:
  }
```

(*End definition for* `\gtl_pop_left:N` *and* `\gtl_gpop_left:N`*. These functions are documented on page 4.*)

`\gtl_pop_left:NN`
`\gtl_gpop_left:NN`

Getting the first token and removing it from the extended token list is done in two steps.

```
\cs_new_protected:Npn \gtl_pop_left:NN #1#2
  {
    \gtl_get_left:NN #1 #2
    \gtl_pop_left:N #1
  }
\cs_new_protected:Npn \gtl_gpop_left:NN #1#2
  {
    \gtl_get_left:NN #1 #2
    \gtl_gpop_left:N #1
  }
```

(*End definition for* `\gtl_pop_left:NN` *and* `\gtl_gpop_left:NN`*. These functions are documented on page 4.*)

2.8 Longest token list starting an extended token list

`\gtl_left_tl:N`
`\__gtl_left_tl:w`

```
\cs_new:Npn \gtl_left_tl:N #1
  { \exp_after:wN \__gtl_left_tl:w #1 }
\cs_new:Npn \__gtl_left_tl:w \s__gtl #1#2#3 \s__stop
  { \tl_if_empty:nTF {#1} { \exp_not:n {#2} } { \tl_head:n {#1} } }
```

(*End definition for* `\gtl_left_tl:N`*. This function is documented on page 5.*)

`\gtl_pop_left_tl:N`
`\gtl_gpop_left_tl:N`

```
\cs_new_protected:Npn \gtl_pop_left_tl:N #1
  { \tl_set:Nx #1 { \exp_after:wN \__gtl_pop_left_tl:w #1 } }
\cs_new_protected:Npn \gtl_gpop_left_tl:N #1
  { \tl_gset:Nx #1 { \exp_after:wN \__gtl_pop_left_tl:w #1 } }
\cs_new:Npn \__gtl_pop_left_tl:w \s__gtl #1#2#3 \s__stop
  {
    \s__gtl
    \tl_if_empty:nTF {#1}
      { { } { } }
      {
        { { } \tl_tail:n {#1} }
        { \exp_not:n {#2} }
      }
    { \exp_not:n {#3} }
    \s__stop
  }
```

(*End definition for* `\gtl_pop_left_tl:N` *and* `\gtl_gpop_left_tl:N`*. These functions are documented on page 5.*)

2.9 First item of an extended token list

`\gtl_left_item:NF`
`\__gtl_left_item:wF`
`\__gtl_left_item_auxi:nwF`

The left-most item of an extended token list is the head of its left token list. The code thus starts like `\gtl_left_tl:N`. It ends with a check to test if we should use the head, or issue the false code.

```
\cs_new:Npn \gtl_left_item:NF #1
  { \exp_after:wN \__gtl_left_item:wF #1 }
\cs_new:Npn \__gtl_left_item:wF \s__gtl #1#2#3 \s__stop
  { \__gtl_left_item_auxi:nwF #1 {#2} \q_stop }
\cs_new:Npn \__gtl_left_item_auxi:nwF #1#2 \q_stop #3
  { \tl_if_blank:nTF {#1} {#3} { \tl_head:n {#1} } }
```

(*End definition for* `\gtl_left_item:NF`*. This function is documented on page 5.*)

`\gtl_pop_left_item:NNTF`
`\gtl_gpop_left_item:NNTF`
`\__gtl_pop_left_item:wNNN`
`\__gtl_pop_left_item_aux:nwnnNNN`

If there is no extra end-group characters, and if the balanced part is blank, we cannot extract an item: return `false`. If the balanced part is not blank, store its first item into `#4`, and store the altered generalized token list into `#6`, locally or globally. Otherwise, pick out the part before the first extra end-group character as `#1` of the second auxiliary, and do essentially the same: if it is blank, there is no item, and if it is not blank, pop its first item.

```
\prg_new_protected_conditional:Npnn \gtl_pop_left_item:NN #1#2 { TF , T , F }
  { \exp_after:wN \__gtl_pop_left_item:wNNN #1#2 \tl_set:Nx #1 }
\prg_new_protected_conditional:Npnn \gtl_gpop_left_item:NN #1#2 { TF , T , F }
  { \exp_after:wN \__gtl_pop_left_item:wNNN #1#2 \tl_gset:Nx #1 }
\cs_new_protected:Npn \__gtl_pop_left_item:wNNN
    \s__gtl #1#2#3 \s__stop #4#5#6
  {
    \tl_if_empty:nTF {#1}
      {
        \tl_if_blank:nTF {#2} { \prg_return_false: }
          {
            \tl_set:Nx #4 { \tl_head:n {#2} }
            #5 #6
              {
                \s__gtl { } { \tl_tail:n {#2} }
                { \exp_not:n {#3} } \s__stop
              }
            \prg_return_true:
          }
      }
      {
        \__gtl_pop_left_item_aux:nwnnNNN #1 \q_nil \q_stop
          {#2} {#3} #4 #5 #6
      }
  }
\cs_new_protected:Npn \__gtl_pop_left_item_aux:nwnnNNN
    #1#2 \q_stop #3#4#5#6#7
  {
    \tl_if_blank:nTF {#1} { \prg_return_false: }
      {
        \tl_set:Nx #5 { \tl_head:n {#1} }
        #6 #7
          {
            \s__gtl
            { { \tl_tail:n {#1} } \__gtl_strip_nil_mark:w #2 \q_mark }
            { \exp_not:n {#3} }
            { \exp_not:n {#4} }
            \s__stop
          }
        \prg_return_true:
      }
  }
```

(*End definition for* `\gtl_pop_left_item:NN` *and* `\gtl_gpop_left_item:NN`*. These functions are documented on page 5.*)

2.10 First group in an extended token list

The functions of this section extract from an extended token list the tokens that would be absorbed after `\def\foo`, namely tokens with no begin-group nor end-group characters, followed by one group. Those tokens are either left in the input stream or stored in a token list variable, and the `pop` functions also remove those tokens from the extended token list variable.

\gtl_left_text:NF
__gtl_left_text:wF
__gtl_left_text_auxi:nwF
__gtl_left_text_auxii:wnwF
__gtl_left_text_auxiii:nnwF

```
\cs_new:Npn \gtl_left_text:NF #1
  { \exp_after:wN \__gtl_left_text:wF #1 }
\cs_new:Npn \__gtl_left_text:wF \s__gtl #1#2#3 \s__stop
  {
    \tl_if_empty:nTF {#1}
      { \__gtl_left_text_auxi:nwF {#2} \q_stop }
      { \__gtl_left_text_auxi:nwF #1 \q_stop }
  }
\cs_new:Npn \__gtl_left_text_auxi:nwF #1#2 \q_stop
  { \__gtl_left_text_auxii:wnwF #1 \q_mark { } \q_mark \q_stop }
\cs_new:Npn \__gtl_left_text_auxii:wnwF #1 #
```

```
  { \__gtl_left_text_auxiii:nnwF {#1} }
\cs_new:Npn \__gtl_left_text_auxiii:nnwF #1#2 #3 \q_mark #4 \q_stop #5
  { \tl_if_empty:nTF {#4} {#5} { \exp_not:n { #1 {#2} } } }
```

(*End definition for* `\gtl_left_text:NF`. *This function is documented on page 5.*)

`\gtl_pop_left_text:N`
`\gtl_gpop_left_text:N`
`\__gtl_pop_left_text:w`
`\__gtl_pop_left_text_auxi:n`
`\__gtl_pop_left_text_auxii:wnw`
`\__gtl_pop_left_text_auxiii:nnw`
`\__gtl_pop_left_text_auxiv:nw`

```
\cs_new_protected:Npn \gtl_pop_left_text:N #1
  { \tl_set:Nx #1 { \exp_after:wN \__gtl_pop_left_text:w #1 } }
\cs_new_protected:Npn \gtl_gpop_left_text:N #1
  { \tl_gset:Nx #1 { \exp_after:wN \__gtl_pop_left_text:w #1 } }
\cs_new:Npn \__gtl_pop_left_text:w \s__gtl #1#2#3 \s__stop
  {
    \s__gtl
    \tl_if_empty:nTF {#1}
      {
        { }
        { \__gtl_pop_left_text_auxi:n {#2} }
      }
      {
        { \__gtl_pop_left_text_auxiv:nw #1 \q_nil \q_mark }
        { \exp_not:n {#2} }
      }
    { \exp_not:n {#3} }
    \s__stop
  }
\cs_new:Npn \__gtl_pop_left_text_auxi:n #1
  {
    \__gtl_pop_left_text_auxii:wnw #1
      \q_nil \q_mark { } \q_mark \q_stop
  }
\cs_new:Npn \__gtl_pop_left_text_auxii:wnw #1 #
  { \__gtl_pop_left_text_auxiii:nnw {#1} }
\cs_new:Npn \__gtl_pop_left_text_auxiii:nnw #1#2#3 \q_mark #4 \q_stop
  {
    \tl_if_empty:nTF {#4}
      { \__gtl_strip_nil_mark:w #1 }
      { \__gtl_strip_nil_mark:w #3 \q_mark }
  }
\cs_new:Npn \__gtl_pop_left_text_auxiv:nw #1
  {
    { \__gtl_pop_left_text_auxi:n {#1} }
    \__gtl_strip_nil_mark:w
  }
```

(*End definition for* `\gtl_pop_left_text:N` *and* `\gtl_gpop_left_text:N`. *These functions are documented on page 6.*)

2.11 Counting tokens

`\__gtl_tl_count:n`
`\__gtl_tl_count_loop:n`
`\__gtl_tl_count_test:w`

A more robust version of `\tl_count:n`, which will however break if the token list contains `\q_stop` at the outer brace level. This cannot happen when `\__gtl_tl_count:n` is called with lists of braced items. The technique is to loop, and when seeing `\q_mark`, make sure that this is really the end of the list.

```
\cs_new:Npn \__gtl_tl_count:n #1
  { \int_eval:n { \c_zero \__gtl_tl_count_loop:n #1 \q_nil \q_stop } }
\cs_new:Npn \__gtl_tl_count_loop:n #1
  {
    \quark_if_nil:nTF {#1}
      { \__gtl_tl_count_test:w }
      { + \c_one \__gtl_tl_count_loop:n }
  }
\cs_new:Npn \__gtl_tl_count_test:w #1 \q_stop
  { \tl_if_empty:nF {#1} { + \c_one \__gtl_tl_count_loop:n #1 \q_stop } }
```

(*End definition for* `\__gtl_tl_count:n`*. This function is documented on page* **??**.)

`\gtl_extra_begin:N`
`\gtl_extra_end:N`
`\__gtl_extra_begin:w`
`\__gtl_extra_end:w`

Count the number of extra end-group or of extra begin-group characters in an extended token list. This is the number of items in the first or third brace groups. We cannot use `\tl_count:n`, as gtl is meant to be robust against inclusion of quarks.

```
\cs_new:Npn \gtl_extra_end:N #1
  { \exp_after:wN \__gtl_extra_end:w #1 }
\cs_new:Npn \__gtl_extra_end:w \s__gtl #1#2#3 \s__stop
  { \__gtl_tl_count:n {#1} }
\cs_new:Npn \gtl_extra_begin:N #1
  { \exp_after:wN \__gtl_extra_begin:w #1 }
\cs_new:Npn \__gtl_extra_begin:w \s__gtl #1#2#3 \s__stop
  { \__gtl_tl_count:n {#3} }
```

(*End definition for* `\gtl_extra_begin:N` *and* `\gtl_extra_end:N`*. These functions are documented on page 6.*)

`\gtl_count_tokens:N`
`\__gtl_count_tokens:w`
`\__gtl_count_auxi:nw`
`\__gtl_count_auxii:w`
`\__gtl_count_auxiii:n`

```
\cs_new:Npn \gtl_count_tokens:N #1
  { \exp_after:wN \__gtl_count_tokens:w #1 }
\cs_new:Npn \__gtl_count_tokens:w \s__gtl #1#2#3 \s__stop
  {
    \int_eval:n
      { \c_minus_one \__gtl_count_auxi:nw #1 {#2} #3 \q_nil \q_stop }
  }
\cs_new:Npn \__gtl_count_auxi:nw #1
  {
    \quark_if_nil:nTF {#1}
      { \__gtl_count_auxii:w }
      {
        + \c_one
        \__gtl_count_auxiii:n {#1}
        \__gtl_count_auxi:nw
      }
  }
\cs_new:Npn \__gtl_count_auxii:w #1 \q_stop
  {
    \tl_if_empty:nF {#1}
```

```
      {
        + \c_two
        \__gtl_count_auxi:nw #1 \q_stop
      }
  }
\cs_new:Npn \__gtl_count_auxiii:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \tl_if_head_is_group:nTF {#1}
          {
            + \c_two
            \exp_args:No \__gtl_count_auxiii:n { \use:n #1 }
          }
          {
            + \c_one
            \tl_if_head_is_N_type:nTF {#1}
              { \exp_args:No \__gtl_count_auxiii:n { \use_none:n #1 } }
              { \exp_args:Nf \__gtl_count_auxiii:n {#1} }
          }
      }
  }
```

(*End definition for `\gtl_count_tokens:N`. This function is documented on page 6.*)

2.12 Messages

```
⟨/package⟩
```