The l3tl-analysis package: analysing token lists*

The LaTeX3 Project†

Released 2011/12/08

1 l3tl-analysis documentation

This module mostly provides internal functions for use in the l3regex module. However, it provides as a side-effect a user debugging function, very similar to the `\ShowTokens` macro from the ted package.

`\tl_show_analysis:N` `\tl_show_analysis:n` `\tl_show_analysis:n {⟨token list⟩}`

Displays to the terminal the detailed decomposition of the ⟨*token list*⟩ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

1.1 Internal functions

`\s__tl` The format used to store token lists internally uses the scan mark `\s__tl` as a delimiter.

`\__tl_analysis_map_inline:nn` `\__tl_analysis_map_inline:nn {⟨token list⟩} {⟨inline function⟩}`

Applies the ⟨*inline function*⟩ to each individual ⟨*token*⟩ in the ⟨*token list*⟩. The ⟨*inline function*⟩ receives three arguments:

- ⟨*tokens*⟩, which both `o`-expand and `x`-expand to the ⟨*token*⟩. The detailed form of ⟨*token*⟩ may change in later releases.
- ⟨*catcode*⟩, a capital hexadecimal digit which denotes the category code of the ⟨*token*⟩ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C:other, D:active).
- ⟨*char code*⟩, a decimal representation of the character code of the token, -1 if it is a control sequence (with ⟨*catcode*⟩ 0).

*This file describes v3039, last revised 2011/12/08.

†E-mail: latex-team@latex-project.org

For optimizations in l3regex (when matching control sequences), it may be useful to provide a `\__tl_analysis_from_str_map_inline:nn` function, perhaps named `\__str_analysis_map_inline:nn`.

1.2 Internal format

The task of the l3tl-analysis module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in l3regex where we need to support arbitrary tokens, and it is used in conversion functions in l3str, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any ⟨*token*⟩ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find ⟨*tokens*⟩ which both `o`-expand and `x`-expand to the given ⟨*token*⟩. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

⟨*tokens*⟩ `\s__tl` ⟨*catcode*⟩ ⟨*char code*⟩ `\s__tl`

The ⟨*tokens*⟩ `o`- *and* `x`-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for l3str). The ⟨*catcode*⟩ is given as a single hexadecimal digit, 0 for control sequences. The ⟨*char code*⟩ is given as a decimal number, −1 for control sequences.

Using delimited arguments lets us build the ⟨*tokens*⟩ progressively when doing an encoding conversion in l3str. On the other hand, the delimiter `\s__tl` may not appear unbraced in ⟨*tokens*⟩. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a ⟨*token*⟩ to a balanced set of ⟨*tokens*⟩ which both `o`-expands and `x`-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl` 0 −1 `\s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl` 1 ⟨*char code*⟩ `\s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl` 2 ⟨*char code*⟩ `\s__tl`.
- A character with any other category code becomes `\exp_not:n {`⟨*character*⟩`} \s__tl` ⟨*hex catcode*⟩ ⟨*char code*⟩ `\s__tl`.

2 l3tl-analysis implementation

1 ⟨*initex | package⟩

2 ⟨@@=tl_analysis⟩

```
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\RequirePackage{l3str}
```

2.1 Variables and helper functions

`\s__tl` The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\__int_value:w ‘#1 \s__tl` with `\__int_value:w ‘#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an `x`-expansion.

```
\__scan_new:N \s__tl
```

(*End definition for* `\s__tl` *This variable is documented on page* 1.)

`\l__tl_analysis_internal_tl` This token list variable is used to hand the argument of `\tl_show_analysis:n` to `\tl_show_analysis:N`.

```
\tl_new:N \l__tl_analysis_internal_tl
```

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

`\l__tl_analysis_token` `\l__tl_analysis_char_token` The tokens in the token list are probed with the TEX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`.

```
\cs_new_eq:NN \l__tl_analysis_token ?
\cs_new_eq:NN \l__tl_analysis_char_token ?
```

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

`\l__tl_analysis_normal_int` The number of normal (`N`-type argument) tokens since the last special token.

```
\int_new:N \l__tl_analysis_normal_int
```

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

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
\int_new:N \l__tl_analysis_index_int
```

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

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
\int_new:N \l__tl_analysis_nesting_int
```

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

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

```
\int_new:N \l__tl_analysis_type_int
```

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

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

⟨*tokens*⟩ `\s__tl` ⟨*catcode*⟩ ⟨*char code*⟩ `\s__tl`

```
\tl_new:N \g__tl_analysis_result_tl
```

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

`\__tl_analysis_extract_charcode:`
`\__tl_analysis_extract_charcode_aux:w`

Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘⟨*char*⟩.

```
\cs_new_nopar:Npn \__tl_analysis_extract_charcode:
  {
    \exp_after:wN \__tl_analysis_extract_charcode_aux:w
      \token_to_meaning:N \l__tl_analysis_token
  }
\cs_new:Npn \__tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ‘ }
```

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

`\__tl_analysis_cs_space_count:NN`
`\__tl_analysis_cs_space_count:w`
`\__tl_analysis_cs_space_count_end:w`

Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

```
\cs_new:Npn \__tl_analysis_cs_space_count:NN #1 #2
  {
    \exp_after:wN #1
    \__int_value:w \__int_eval:w \c_zero
      \exp_after:wN \__tl_analysis_cs_space_count:w
        \token_to_str:N #2
        \fi: \__tl_analysis_cs_space_count_end:w ; ~ !
  }
\cs_new:Npn \__tl_analysis_cs_space_count:w #1 ~
  {
    \if_false: #1 #1 \fi:
    + \c_one
    \__tl_analysis_cs_space_count:w
  }
\cs_new:Npn \__tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
  { \exp_after:wN ; \__int_value:w \str_count_ignore_spaces:n {#1} ; }
```

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

2.2 Plan of attack

Our goal is to produce a token list of the form roughly

⟨*token 1*⟩ `\s__tl` ⟨*catcode 1*⟩ ⟨*char code 1*⟩ `\s__tl`
⟨*token 2*⟩ `\s__tl` ⟨*catcode 2*⟩ ⟨*char code 2*⟩ `\s__tl`
. . . ⟨*token N*⟩ `\s__tl` ⟨*catcode N*⟩ ⟨*char code N*⟩ `\s__tl`

Most but not all tokens can be grabbed as an undelimited (`N`-type) argument by TEX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an `x`-expanding assignment, normal tokens are taken in as `N`-type arguments, and special tokens are

retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

To ease the difficult first pass, we first do some setup with `\__tl_analysis_setup:n`. Active characters set equal to non-active characters cause trouble, so we disable all active characters by setting them equal to `undefined` locally. We also set there the escape character to be printable (backslash, but this later oscillates between slash and backslash): this makes it possible to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for TEX when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an `N`-type argument).

The token itself can "look like" one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it "true" character;
- an active character (we eliminate those in the setup step);
- a control sequence.

The only tokens which are not valid `N`-type arguments are true begin-group characters, true end-group characters, and true spaces. We will detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`\__tl_analysis:n` Everything is done within a group, and all definitions will be local. We use `\group_align_safe_begin/end:` to avoid problems in case `\__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```
\cs_new_protected:Npn \__tl_analysis:n #1
  {
    \group_begin:
      \group_align_safe_begin:
        \__tl_analysis_setup:n {#1}
        \__tl_analysis_i:n {#1}
        \__tl_analysis_ii:n {#1}
      \group_align_safe_end:
    \group_end:
  }
```

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

2.3 Setup

__tl_analysis_setup:n
__tl_analysis_disable_loop:N

Active characters can cause problems later on in the processing, so the first step is to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead loop over the input token list as a string: any active character in the token list must appear in its string representation. The string is shortened a little by making the escape character unprintable. The active space must be disabled separately (the loop skips over it otherwise), and we end the loop by feeding an odd non-N-type argument to the looping macro.

```
\cs_new_protected:Npn \__tl_analysis_setup:n #1
  {
    \int_set_eq:NN \tex_escapechar:D \c_minus_one
    \exp_after:wN \__tl_analysis_disable_loop:N
      \tl_to_str:n {#1} { ~ } { ? ~ \__prg_break: }
    \__prg_break_point:
  }
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_new_protected:Npn \__tl_analysis_disable_loop:N #1
    {
      \tex_lccode:D \c_zero `#1 ~
      \tl_to_lowercase:n { \tex_let:D ^^@ } \c_undefined:D
      \__tl_analysis_disable_loop:N
    }
\group_end:
```

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

2.4 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

After the setup step, we have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an undefined active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);

10. a control sequence equal to a space token (character code 32, category code 10);

11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases will be distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence's string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid `N`-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {`⟨*token*⟩`}` is non-empty, because the escape character is printable.

`\__tl_analysis_i:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches −1 when we read the closing brace.

```
\cs_new_protected:Npn \__tl_analysis_i:n #1
  {
    \int_set:Nn \tex_escapechar:D { 92 }
    \int_zero:N \l__tl_analysis_normal_int
    \int_zero:N \l__tl_analysis_index_int
    \int_zero:N \l__tl_analysis_nesting_int
    \if_false: { \fi: \__tl_analysis_i_loop:w #1 }
    \int_decr:N \l__tl_analysis_index_int
  }
```

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

`\__tl_analysis_i_loop:w` Read one character and check its type.

```
\cs_new_protected_nopar:Npn \__tl_analysis_i_loop:w
  { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_i_type:w }
```

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

`\__tl_analysis_i_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, −1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```
\cs_new_protected_nopar:Npn \__tl_analysis_i_type:w
```

```
{
  \l__tl_analysis_type_int =
    \if_meaning:w \l__tl_analysis_token \c_space_token
      \c_zero
    \else:
      \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
        \c_one
      \else:
        \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
          \c_minus_one
        \else:
          \c_two
        \fi:
      \fi:
    \fi:
  \if_case:w \l__tl_analysis_type_int
       \exp_after:wN \__tl_analysis_i_space:w
  \or: \exp_after:wN \__tl_analysis_i_bgroup:w
  \or: \exp_after:wN \__tl_analysis_i_safe:N
  \else: \exp_after:wN \__tl_analysis_i_egroup:w
  \fi:
}
```

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

`\__tl_analysis_i_space:w`
`\__tl_analysis_i_space_test:w`

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: if it is a control sequence the result starts with the escape character; otherwise it is a true blank space, whose string representation is also a blank space. We test for that in `\__tl_analysis_i_space_test:w`, after grabbing as `\l__tl_analysis_char_token` the first character of the string representation. Also, since `\__tl_analysis_i_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```
\cs_new_protected_nopar:Npn \__tl_analysis_i_space:w
  {
    \tex_afterassignment:D \__tl_analysis_i_space_test:w
    \exp_after:wN \cs_set_eq:NN
    \exp_after:wN \l__tl_analysis_char_token
    \token_to_str:N
  }
\cs_new_protected_nopar:Npn \__tl_analysis_i_space_test:w
  {
    \if_meaning:w \l__tl_analysis_char_token \c_space_token
      \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
```

```
      \__tl_analysis_i_store:
    \else:
      \int_incr:N \l__tl_analysis_normal_int
    \fi:
    \__tl_analysis_i_loop:w
  }
```

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

`\__tl_analysis_i_bgroup:w`
`\__tl_analysis_i_egroup:w`
`\__tl_analysis_i_group:nw`
`\__tl_analysis_i_group_test:w`

The token might be either a true character token with catcode 1 or 2, or it could be a control sequence. The only tricky case is if the character code happens to be equal to the escape character: then we change the escape character from backslash to solidus or back, so that the string representation of the true character and of a control sequence set equal to it start differently. Then probe what the first character of that string representation is: this is the place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```
\group_begin:
  \char_set_catcode_group_begin:N \^^@
  \char_set_catcode_group_end:N \^^E
  \cs_new_protected_nopar:Npn \__tl_analysis_i_bgroup:w
    { \__tl_analysis_i_group:nw { \exp_after:wN ^^@ \if_false: ^^E \fi: } }
  \char_set_catcode_group_begin:N \^^B
  \char_set_catcode_group_end:N \^^@
  \cs_new_protected_nopar:Npn \__tl_analysis_i_egroup:w
    { \__tl_analysis_i_group:nw { \if_false: ^^B \fi: ^^@ } }
\group_end:
\cs_new_protected:Npn \__tl_analysis_i_group:nw #1
  {
    \tex_lccode:D \c_zero = \__tl_analysis_extract_charcode: \scan_stop:
    \tl_to_lowercase:n { \tex_toks:D \l__tl_analysis_index_int {#1} }
    \if_int_compare:w \tex_lccode:D \c_zero = \tex_escapechar:D
      \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
    \fi:
    \tex_afterassignment:D \__tl_analysis_i_group_test:w
    \exp_after:wN \cs_set_eq:NN
    \exp_after:wN \l__tl_analysis_char_token
    \token_to_str:N
  }
\cs_new_protected_nopar:Npn \__tl_analysis_i_group_test:w
  {
    \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
      \__tl_analysis_i_store:
    \else:
      \int_incr:N \l__tl_analysis_normal_int
    \fi:
    \__tl_analysis_i_loop:w
  }
```

(*End definition for* `\__tl_analysis_i_bgroup:w` *and* `\__tl_analysis_i_egroup:w` *These functions are documented on page* **??**.)

`\__tl_analysis_i_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those will behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens, and the type of special token, are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```
\cs_new_protected_nopar:Npn \__tl_analysis_i_store:
  {
    \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
    \if_int_compare:w \tex_lccode:D \c_zero = \c_thirty_two
      \tex_multiply:D \l__tl_analysis_type_int \c_two
    \fi:
    \tex_skip:D \l__tl_analysis_index_int
      = \l__tl_analysis_normal_int sp plus \l__tl_analysis_type_int sp \scan_stop:
    \int_incr:N \l__tl_analysis_index_int
    \int_zero:N \l__tl_analysis_normal_int
    \if_int_compare:w \l__tl_analysis_nesting_int = \c_minus_one
      \cs_set_eq:NN \__tl_analysis_i_loop:w \scan_stop:
    \fi:
  }
```

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

`\__tl_analysis_i_safe:N` `\__tl_analysis_i_cs:ww` This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. However, other branches of the code must pass their tokens through `\string`, hence we do it here as well, with some optimizations. If the token is a single character (including space), the `\if_charcode:w` test yields true, and we simply count one "normal" token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead

of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of "normal" tokens.

```
\cs_new_protected:Npn \__tl_analysis_i_safe:N #1
  {
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
        \scan_stop:
      \int_incr:N \l__tl_analysis_normal_int
    \else:
      \__tl_analysis_cs_space_count:NN \__tl_analysis_i_cs:ww #1
    \fi:
    \__tl_analysis_i_loop:w
  }
\cs_new_protected:Npn \__tl_analysis_i_cs:ww #1; #2;
  {
    \if_int_compare:w #1 > \c_zero
      \tex_skip:D \l__tl_analysis_index_int
        = \__int_eval:w \l__tl_analysis_normal_int + \c_one sp \scan_stop:
      \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
      \l__tl_analysis_normal_int #2 \exp_stop_f:
    \else:
      \tex_advance:D \l__tl_analysis_normal_int #2 \exp_stop_f:
    \fi:
  }
```

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

2.5 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`\__tl_analysis_ii:n`
`\__tl_analysis_ii_loop:w`

Start the loop with the index 0. No need for an end-marker: the loop will stop by itself when the last index is read. We will repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```
\cs_new_protected:Npn \__tl_analysis_ii:n #1
  {
    \tl_gset:Nx \g__tl_analysis_result_tl
      {
        \__tl_analysis_ii_loop:w 0; #1
        \__prg_break_point:
      }
  }
```

```
\cs_new:Npn \__tl_analysis_ii_loop:w #1;
  {
    \exp_after:wN \__tl_analysis_ii_normals:ww
      \__int_value:w \tex_skip:D #1 ; #1 ;
  }
```

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

`\__tl_analysis_ii_normals:ww`
`\__tl_analysis_ii_normal:wwN`

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n {`⟨*token*⟩`}` `\s__tl` in the input stream (after x-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because `#3` could be `\s__tl`, hence must be hidden behind braces in the result.

```
\cs_new:Npn \__tl_analysis_ii_normals:ww #1;
  {
    \if_int_compare:w #1 = \c_zero
      \__tl_analysis_ii_special:w
    \fi:
    \__tl_analysis_ii_normal:wwN #1;
  }
\cs_new:Npn \__tl_analysis_ii_normal:wwN #1; #2; #3
  {
    \exp_not:n { \exp_not:n { #3 } } \s__tl
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
        \scan_stop:
      \exp_after:wN \__tl_analysis_ii_char:Nww
    \else:
      \exp_after:wN \__tl_analysis_ii_cs:Nww
    \fi:
    #3 #1; #2;
  }
```

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

`\__tl_analysis_ii_char:Nww`

If the normal token we grab is a character, leave ⟨*catcode*⟩ ⟨*charcode*⟩ followed by `\s__tl` in the input stream, and call `\__tl_analysis_ii_normals:ww` with its first argument decremented.

```
\group_begin:
  \char_set_catcode_other:N A
  \char_set_catcode_other:N B
  \char_set_catcode_other:N C
  \char_set_uccode:nn { `? } { `D }
  \tl_to_uppercase:n
    {
      \cs_new:Npn \__tl_analysis_ii_char:Nww #1
        {
```

```
          \if_meaning:w #1 \c_undefined:D            ? \else:
          \if_catcode:w #1 \c_catcode_other_token    C \else:
          \if_catcode:w #1 \c_catcode_letter_token   B \else:
          \if_catcode:w #1 \c_math_toggle_token      3 \else:
          \if_catcode:w #1 \c_alignment_token        4 \else:
          \if_catcode:w #1 \c_math_superscript_token 7 \else:
          \if_catcode:w #1 \c_math_subscript_token   8 \else:
          \if_catcode:w #1 \c_space_token            A \else:
            6
          \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
          \__int_value:w `#1 \s__tl
          \exp_after:wN \__tl_analysis_ii_normals:ww
            \int_use:N \__int_eval:w \c_minus_one +
        }
    }
\group_end:
```

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

`\__tl_analysis_ii_cs:Nww`
`\__tl_analysis_ii_cs_test:ww`

If the token we grab is a control sequence, leave `0 -1` (as category code and character code) in the input stream, followed by `\s__tl`, and call `\__tl_analysis_ii_normals:ww` with updated arguments.

```
\cs_new:Npn \__tl_analysis_ii_cs:Nww #1
  {
    0 -1 \s__tl
    \__tl_analysis_cs_space_count:NN \__tl_analysis_ii_cs_test:ww #1
  }
\cs_new:Npn \__tl_analysis_ii_cs_test:ww #1 ; #2 ; #3 ; #4 ;
  {
    \exp_after:wN \__tl_analysis_ii_normals:ww
    \int_use:N \__int_eval:w
    \if_int_compare:w #1 = \c_zero
      #3
    \else:
      \tex_skip:D \__int_eval:w #4 + #1 \__int_eval_end:
    \fi:
    - #2
    \exp_after:wN ;
    \int_use:N \__int_eval:w #4 + #1 ;
  }
```

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

`\__tl_analysis_ii_special:w`
`\__tl_analysis_ii_special_char:wN`
`\__tl_analysis_ii_special_space:w`

Here, `#1` is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the `\toks` register: when `x`-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `\__tl_analysis_ii_loop:w` with the next index.

```
\group_begin:
  \char_set_catcode_other:N A
  \cs_new:Npn \__tl_analysis_ii_special:w
```

```
      \fi: \__tl_analysis_ii_normal:wwN 0 ; #1 ;
    {
      \fi:
      \if_int_compare:w #1 = \l__tl_analysis_index_int
        \exp_after:wN \__prg_break:
      \fi:
      \tex_the:D \tex_toks:D #1 \s__tl
      \if_case:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
            A
      \or:   1
      \or:   1
      \else: 2
      \fi:
      \if_int_odd:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
        \exp_after:wN \__tl_analysis_ii_special_char:wN \int_use:N
      \else:
        \exp_after:wN \__tl_analysis_ii_special_space:w \int_use:N
      \fi:
      \__int_eval:w \c_one + #1 \exp_after:wN ;
      \token_to_str:N
    }
\group_end:
\cs_new:Npn \__tl_analysis_ii_special_char:wN #1 ; #2
  {
    \__int_value:w `#2 \s__tl
    \__tl_analysis_ii_loop:w #1 ;
  }
\cs_new:Npn \__tl_analysis_ii_special_space:w #1 ; ~
  {
    32 \s__tl
    \__tl_analysis_ii_loop:w #1 ;
  }
```

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

2.6 Mapping through the analysis

`\__tl_analysis_map_inline:nn`
`\__tl_analysis_map_inline_aux:Nn`

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__prg_map_int` (shared between all modules), then define the looping macro, which has a name specific to that nesting depth. That looping grabs the ⟨*tokens*⟩, ⟨*catcode*⟩ and ⟨*char code*⟩; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then performs the user's code `#2`, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```
\cs_new_protected:Npn \__tl_analysis_map_inline:nn #1
  {
    \__tl_analysis:n {#1}
    \int_gincr:N \g__prg_map_int
    \exp_args:Nc \__tl_analysis_map_inline_aux:Nn
      { __tl_analysis_map_inline_ \int_use:N \g__prg_map_int :wNw }
```

```
  }
\cs_new_protected:Npn \__tl_analysis_map_inline_aux:Nn #1#2
  {
    \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
      {
        \use_none:n ##2
        #2
        #1
      }
    \exp_after:wN #1
      \g__tl_analysis_result_tl
      \s__tl { ? \tl_map_break: } \s__tl
    \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
  }
```

(*End definition for* `\__tl_analysis_map_inline:nn` *This function is documented on page 1.*)

2.7 Showing the results

`\tl_show_analysis:N`
`\tl_show_analysis:n`

Add to `\__tl_analysis:n` a third pass to display tokens to the terminal.

```
\cs_new_protected:Npn \tl_show_analysis:N #1
  {
    \exp_args:No \__tl_analysis:n {#1}
    \__msg_show_variable:Nnn #1
      { tl-analysis }
      {
        \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
          \s__tl { ? \__prg_break: } \s__tl
        \__prg_break_point:
      }
  }
\cs_new_protected:Npn \tl_show_analysis:n #1
  {
    \tl_set:Nn \l__tl_analysis_internal_tl {#1}
    \tl_show_analysis:N \l__tl_analysis_internal_tl
  }
```

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

`\__tl_analysis_show_loop:wNw`

Here, `#1` o- and x-expands to the token; `#2` is the category code (one uppercase hexadecimal digit), 0 for control sequences; `#3` is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```
\cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
  {
    \use_none:n #2
    \iow_newline: > \c_space_tl \c_space_tl
    \if_int_compare:w "#2 = \c_zero
      \exp_after:wN \__tl_analysis_show_cs:n
    \else:
      \if_int_compare:w "#2 = \c_thirteen
```

```
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__tl_analysis_show_active:n
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__tl_analysis_show_normal:n
      \fi:
    \fi:
    {#1}
    \__tl_analysis_show_loop:wNw
  }
```

(*End definition for* `\__tl_analysis_show_loop:wNw`)

`\__tl_analysis_show_normal:n` Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up TEX's alignment status.

```
\cs_new:Npn \__tl_analysis_show_normal:n #1
  {
    \exp_after:wN \token_to_str:N #1 ~
    ( \exp_after:wN \token_to_meaning:N #1 )
  }
```

(*End definition for* `\__tl_analysis_show_normal:n`)

`\__tl_analysis_show_value:N` This expands to the value of `#1` if it has any.

```
\cs_new:Npn \__tl_analysis_show_value:N #1
  {
    \token_if_expandable:NF #1
      {
        \token_if_chardef:NTF       #1 \__prg_break: { }
        \token_if_mathchardef:NTF   #1 \__prg_break: { }
        \token_if_dim_register:NTF  #1 \__prg_break: { }
        \token_if_int_register:NTF  #1 \__prg_break: { }
        \token_if_skip_register:NTF #1 \__prg_break: { }
        \token_if_toks_register:NTF #1 \__prg_break: { }
        \use_none:nnn
        \__prg_break_point:
        \use:n { = \tex_the:D #1 }
      }
  }
```

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

`\__tl_analysis_show_cs:n`
`\__tl_analysis_show_active:n`
`\__tl_analysis_show_long:nn`
`\__tl_analysis_show_long_aux:nnnn`

Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c__tl_analysis_show_etc_str`.

```
\cs_new:Npn \__tl_analysis_show_cs:n #1
  { \exp_args:No \__tl_analysis_show_long:nn {#1} { control~sequence= } }
\cs_new:Npn \__tl_analysis_show_active:n #1
  { \exp_args:No \__tl_analysis_show_long:nn {#1} { active~character= } }
\cs_new:Npn \__tl_analysis_show_long:nn #1
  {
```

```
    \__tl_analysis_show_long_aux:oofn
      { \token_to_str:N #1 }
      { \token_to_meaning:N #1 }
      { \__tl_analysis_show_value:N #1 }
  }
\cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
  {
    \int_compare:nNnTF
      { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
      > { \l_iow_line_count_int - \c_three }
      {
        \str_substr:nnn { #1 ~ ( #4 #2 #3 ) } \c_one
          {
            \l_iow_line_count_int - \c_three
            - \str_count:N \c__tl_analysis_show_etc_str
          }
        \c__tl_analysis_show_etc_str
      }
      { #1 ~ ( #4 #2 #3 ) }
  }
\cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }
```

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

2.8 Messages

`\c__tl_analysis_show_etc_str`

When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```
\tl_const:Nx \c__tl_analysis_show_etc_str % (
  { \token_to_str:N \ETC.) }
```

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

```
\__msg_kernel_new:nnn { kernel } { show-tl-analysis }
  {
    The~token~list~
    \str_if_eq:nnF {#1} { \l__tl_analysis_internal_tl }
      { \token_to_str:N #1 ~ }
    \tl_if_empty:NTF #1
      { is~empty }
      { contains~the~tokens: }
  }
```

⟨/initex | package⟩

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols

C

E

F

G

I

L

O

P

R

S

T

U