

The l3tl-analysis package: analysing token lists*

The L^AT_EX3 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` $\{\langle token\ list\rangle\}$

Displays to the terminal the detailed decomposition of the $\langle token\ list\rangle$ 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` $\{\langle token\ list\rangle\}$ $\{\langle inline\ function\rangle\}$

Applies the $\langle inline\ function\rangle$ to each individual $\langle token\rangle$ in the $\langle token\ list\rangle$. The $\langle inline\ function\rangle$ receives three arguments:

- $\langle tokens\rangle$, which both o-expand and x-expand to the $\langle token\rangle$. The detailed form of $\langle token\rangle$ may change in later releases.
- $\langle catcode\rangle$, a capital hexadecimal digit which denotes the category code of the $\langle token\rangle$ (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).
- $\langle char\ code\rangle$, a decimal representation of the character code of the token, -1 if it is a control sequence (with $\langle catcode\rangle$ 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 `_t1_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-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any $\langle token \rangle$ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find $\langle tokens \rangle$ which both `o`-expand and `x`-expand to the given $\langle token \rangle$. 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

$\langle tokens \rangle \backslash s_t1 \langle catcode \rangle \langle char\ code \rangle \backslash s_t1$

The $\langle tokens \rangle$ `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-convert`). The $\langle catcode \rangle$ is given as a single hexadecimal digit, 0 for control sequences. The $\langle char\ code \rangle$ is given as a decimal number, `-1` for control sequences.

Using delimited arguments lets us build the $\langle tokens \rangle$ progressively when doing an encoding conversion in `l3str-convert`. On the other hand, the delimiter `\s_t1` may not appear unbraced in $\langle tokens \rangle$. 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 $\langle token \rangle$ to a balanced set of $\langle tokens \rangle$ which both `o`-expands and `x`-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s_t1 0 -1 \s_t1`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s_t1 1 \langle char\ code \rangle \s_t1`.
- An end-group character `}` becomes `\if_false: { \fi: } \s_t1 2 \langle char\ code \rangle \s_t1`.
- A character with any other category code becomes `\exp_not:n { \langle character \rangle } \s_t1 \langle hex\ catcode \rangle \langle char\ code \rangle \s_t1`.

2 l3tl-analysis implementation

```

1 (*initex | package)
2 <@@=tl_analysis>
3 \ProvidesExplPackage
4   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5 \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.

```

6 \__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`.

```

7 \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`.

```

8 \cs_new_eq:NN \l__tl_analysis_token ?
9 \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.

```

10 \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.

```

11 \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.

```

12 \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.

```

13 \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

$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$

14 `\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:` 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 ‘ $\langle char \rangle$ ’.

`_tl_analysis_extract_charcode_aux:w`

15 `\cs_new_nopar:Npn _tl_analysis_extract_charcode:`

16 `{`

17 `\exp_after:wN _tl_analysis_extract_charcode_aux:w`

18 `\token_to_meaning:N \l__tl_analysis_token`

19 `}`

20 `\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` 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.

`_tl_analysis_cs_space_count:w`

`_tl_analysis_cs_space_count_end:w`

21 `\cs_new:Npn _tl_analysis_cs_space_count:NN #1 #2`

22 `{`

23 `\exp_after:wN #1`

24 `__int_value:w __int_eval:w \c_zero`

25 `\exp_after:wN _tl_analysis_cs_space_count:w`

26 `\token_to_str:N #2`

27 `\fi: _tl_analysis_cs_space_count_end:w ; ~ !`

28 `}`

29 `\cs_new:Npn _tl_analysis_cs_space_count:w #1 ~`

30 `{`

31 `\if_false: #1 #1 \fi:`

32 `+ \c_one`

33 `_tl_analysis_cs_space_count:w`

34 `}`

35 `\cs_new:Npn _tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !`

36 `{ \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

$\langle token\ 1 \rangle \backslash s_tl \langle catcode\ 1 \rangle \langle char\ code\ 1 \rangle \backslash s_tl$

$\langle token\ 2 \rangle \backslash s_tl \langle catcode\ 2 \rangle \langle char\ code\ 2 \rangle \backslash s_tl$

$\dots \langle token\ N \rangle \backslash s_tl \langle catcode\ N \rangle \langle char\ code\ N \rangle \backslash s_tl$

Most but not all tokens can be grabbed as an undelimited (N-type) argument by T_EX. 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 T_EX 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.

```

37 \cs_new_protected:Npn \__tl_analysis:n #1
38 {
39   \group_begin:
40   \group_align_safe_begin:
41     \__tl_analysis_setup:n {#1}
42     \__tl_analysis_a:n {#1}
43     \__tl_analysis_b:n {#1}
44   \group_align_safe_end:

```

```

45     \group_end:
46   }
(End definition for \_tl_analysis:n.)

```

2.3 Setup

`_tl_analysis_setup: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.

```

47 \cs_new_protected:Npn \_tl_analysis_setup:n #1
48   {
49     \int_set_eq:NN \tex_escapechar:D \c_minus_one
50     \exp_after:wN \_tl_analysis_disable_loop:N
51       \tl_to_str:n {#1} { ~ } { ? ~ \_prg_break: }
52     \_prg_break_point:
53   }
54 \group_begin:
55   \char_set_catcode_active:N \^^@
56   \cs_new_protected:Npn \_tl_analysis_disable_loop:N #1
57     {
58       \tex_lccode:D \c_zero ' #1 ~
59       \tl_to_lowercase:n { \tex_let:D ^^@ } \tex_undefined:D
60       \_tl_analysis_disable_loop:N
61     }
62 \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_a: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.

```

63 \cs_new_protected:Npn \__tl_analysis_a:n #1
64 {
65   \int_set:Nn \tex_escapechar:D { 92 }
66   \int_zero:N \l__tl_analysis_normal_int
67   \int_zero:N \l__tl_analysis_index_int
68   \int_zero:N \l__tl_analysis_nesting_int
69   \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
70   \int_decr:N \l__tl_analysis_index_int
71 }

```

(End definition for `__tl_analysis_a:n`.)

`__tl_analysis_a_loop:w` Read one character and check its type.

```

72 \cs_new_protected_nopar:Npn \__tl_analysis_a_loop:w
73 { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }

```

(End definition for `__tl_analysis_a_loop:w`.)

`__tl_analysis_a_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).

```

74 \cs_new_protected_nopar:Npn \__tl_analysis_a_type:w
75 {
76   \l__tl_analysis_type_int =
77   \if_meaning:w \l__tl_analysis_token \c_space_token
78     \c_zero
79   \else:
80     \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
81       \c_one
82     \else:
83       \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
84         \c_minus_one
85       \else:
86         \c_two
87     \fi:
88   \fi:
89   \if_case:w \l__tl_analysis_type_int
90     \exp_after:wN \__tl_analysis_a_space:w
91   \or: \exp_after:wN \__tl_analysis_a_bgroup:w
92   \or: \exp_after:wN \__tl_analysis_a_safe:N
93   \else: \exp_after:wN \__tl_analysis_a_egroup:w
94   \fi:
95 \fi:
96 }

```

(End definition for `__tl_analysis_a_type:w`.)

`__tl_analysis_a_space: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_a_space_test:w`, after grabbing as `\l__tl_analysis_char_token` the first character of the string representation. Also, since `__tl_analysis_a_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.

```

97 \cs_new_protected_nopar:Npn \__tl_analysis_a_space:w
98 {
99   \tex_afterassignment:D \__tl_analysis_a_space_test:w
100   \exp_after:wN \cs_set_eq:NN
101   \exp_after:wN \l__tl_analysis_char_token
102   \token_to_str:N

```



```

103 }
104 \cs_new_protected_nopar:Npn \__tl_analysis_a_space_test:w
105 {
106   \if_meaning:w \l__tl_analysis_char_token \c_space_token
107     \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
108     \__tl_analysis_a_store:
109   \else:
110     \int_incr:N \l__tl_analysis_normal_int
111   \fi:
112   \__tl_analysis_a_loop:w
113 }

```

(End definition for __tl_analysis_a_space:w. This function is documented on page ??.)

```

\__tl_analysis_a_bgroup:w
\__tl_analysis_a_egroup:w
\__tl_analysis_a_group:nw
  \__tl_analysis_a_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.

```

114 \group_begin:
115   \char_set_catcode_group_begin:N \^^@
116   \char_set_catcode_group_end:N \^^E
117   \cs_new_protected_nopar:Npn \__tl_analysis_a_bgroup:w
118     { \__tl_analysis_a_group:nw { \exp_after:wN \^^@ \if_false: \^^E \fi: } }
119   \char_set_catcode_group_begin:N \^^B
120   \char_set_catcode_group_end:N \^^@
121   \cs_new_protected_nopar:Npn \__tl_analysis_a_egroup:w
122     { \__tl_analysis_a_group:nw { \if_false: \^^B \fi: \^^@ } }
123 \group_end:
124 \cs_new_protected:Npn \__tl_analysis_a_group:nw #1
125 {
126   \tex_lccode:D \c_zero = \__tl_analysis_extract_charcode: \scan_stop:
127   \tl_to_lowercase:n { \tex_toks:D \l__tl_analysis_index_int {#1} }
128   \if_int_compare:w \tex_lccode:D \c_zero = \tex_escapechar:D
129     \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
130   \fi:
131   \tex_afterassignment:D \__tl_analysis_a_group_test:w
132   \exp_after:wN \cs_set_eq:NN
133   \exp_after:wN \l__tl_analysis_char_token
134   \token_to_str:N
135 }
136 \cs_new_protected_nopar:Npn \__tl_analysis_a_group_test:w
137 {
138   \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
139     \__tl_analysis_a_store:
140   \else:
141     \int_incr:N \l__tl_analysis_normal_int
142   \fi:

```

```

143     \_tl_analysis_a_loop:w
144 }

```

(End definition for _tl_analysis_a_bgroup:w and _tl_analysis_a_egroup:w. These functions are documented on page ??.)

_tl_analysis_a_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).

```

145 \cs_new_protected_nopar:Npn \_tl_analysis_a_store:
146 {
147     \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
148     \if_int_compare:w \tex_lccode:D \c_zero = \c_thirty_two
149         \tex_multiply:D \l__tl_analysis_type_int \c_two
150     \fi:
151     \tex_skip:D \l__tl_analysis_index_int
152         = \l__tl_analysis_normal_int sp plus \l__tl_analysis_type_int sp \scan_stop:
153     \int_incr:N \l__tl_analysis_index_int
154     \int_zero:N \l__tl_analysis_normal_int
155     \if_int_compare:w \l__tl_analysis_nesting_int = \c_minus_one
156         \cs_set_eq:NN \_tl_analysis_a_loop:w \scan_stop:
157     \fi:
158 }

```

(End definition for _tl_analysis_a_store:.)

`__tl_analysis_a_safe:N` 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.

```

159 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
160 {
161   \if_charcode:w
162     \scan_stop:
163     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
164     \scan_stop:
165     \int_incr:N \l__tl_analysis_normal_int
166   \else:
167     \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1
168   \fi:
169   \__tl_analysis_a_loop:w
170 }
171 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
172 {
173   \if_int_compare:w #1 > \c_zero
174     \tex_skip:D \l__tl_analysis_index_int
175     = \__int_eval:w \l__tl_analysis_normal_int + \c_one sp \scan_stop:
176     \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
177     \l__tl_analysis_normal_int #2 \exp_stop_f:
178   \else:
179     \tex_advance:D \l__tl_analysis_normal_int #2 \exp_stop_f:
180   \fi:
181 }

```

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

```

182 \cs_new_protected:Npn \__tl_analysis_b:n #1
183 {

```

```

184 \tl_gset:Nx \g__tl_analysis_result_tl
185 {
186   \__tl_analysis_b_loop:w 0; #1
187   \__prg_break_point:
188 }
189 }
190 \cs_new:Npn \__tl_analysis_b_loop:w #1;
191 {
192   \exp_after:wN \__tl_analysis_b_normals:ww
193   \__int_value:w \tex_skip:D #1 ; #1 ;
194 }

```

(End definition for __tl_analysis_b:n. This function is documented on page ??.)

__tl_analysis_b_normals:ww
 __tl_analysis_b_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.

```

195 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
196 {
197   \if_int_compare:w #1 = \c_zero
198     \__tl_analysis_b_special:w
199   \fi:
200   \__tl_analysis_b_normal:wwN #1;
201 }
202 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
203 {
204   \exp_not:n { \exp_not:n { #3 } } \s__tl
205   \if_charcode:w
206     \scan_stop:
207     \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
208     \scan_stop:
209     \exp_after:wN \__tl_analysis_b_char:Nww
210   \else:
211     \exp_after:wN \__tl_analysis_b_cs:Nww
212   \fi:
213   #3 #1; #2;
214 }

```

(End definition for __tl_analysis_b_normals:ww. This function is documented on page ??.)

__tl_analysis_b_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_b_normals:ww` with its first argument decremented.

```

215 \group_begin:
216   \char_set_catcode_other:N A
217   \char_set_catcode_other:N B

```

```

218 \char_set_catcode_other:N C
219 \char_set_uccode:nn { ' ? } { ' D }
220 \tl_to_uppercase:n
221 {
222   \cs_new:Npn \__tl_analysis_b_char:Nww #1
223   {
224     \if_meaning:w #1 \tex_undefined:D ? \else:
225     \if_catcode:w #1 \c_catcode_other_token C \else:
226     \if_catcode:w #1 \c_catcode_letter_token B \else:
227     \if_catcode:w #1 \c_math_toggle_token 3 \else:
228     \if_catcode:w #1 \c_alignment_token 4 \else:
229     \if_catcode:w #1 \c_math_superscript_token 7 \else:
230     \if_catcode:w #1 \c_math_subscript_token 8 \else:
231     \if_catcode:w #1 \c_space_token A \else:
232     6
233     \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
234     \__int_value:w '#1 \s__tl
235     \exp_after:wN \__tl_analysis_b_normals:ww
236     \int_use:N \__int_eval:w \c_minus_one +
237   }
238 }
239 \group_end:

```

(End definition for __tl_analysis_b_char:Nww.)

__tl_analysis_b_cs:Nww 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_b_normals:ww with updated arguments.

```

240 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
241 {
242   0 -1 \s__tl
243   \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
244 }
245 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
246 {
247   \exp_after:wN \__tl_analysis_b_normals:ww
248   \int_use:N \__int_eval:w
249   \if_int_compare:w #1 = \c_zero
250   #3
251   \else:
252   \tex_skip:D \__int_eval:w #4 + #1 \__int_eval_end:
253   \fi:
254   - #2
255   \exp_after:wN ;
256   \int_use:N \__int_eval:w #4 + #1 ;
257 }

```

(End definition for __tl_analysis_b_cs:Nww. This function is documented on page ??.)

__tl_analysis_b_special: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

```

\__tl_analysis_b_special_char:wN
\__tl_analysis_b_special_space:w

```

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_b_loop:w` with the next index.

```

258 \group_begin:
259   \char_set_catcode_other:N A
260   \cs_new:Npn \__tl_analysis_b_special:w
261     \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
262   {
263     \fi:
264     \if_int_compare:w #1 = \l__tl_analysis_index_int
265       \exp_after:wN \__prg_break:
266     \fi:
267     \tex_the:D \tex_toks:D #1 \s__tl
268     \if_case:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
269       A
270     \or: 1
271     \or: 1
272     \else: 2
273     \fi:
274     \if_int_odd:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
275       \exp_after:wN \__tl_analysis_b_special_char:wN \int_use:N
276     \else:
277       \exp_after:wN \__tl_analysis_b_special_space:w \int_use:N
278     \fi:
279     \__int_eval:w \c_one + #1 \exp_after:wN ;
280     \token_to_str:N
281   }
282 \group_end:
283 \cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
284   {
285     \__int_value:w '#2 \s__tl
286     \__tl_analysis_b_loop:w #1 ;
287   }
288 \cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
289   {
290     32 \s__tl
291     \__tl_analysis_b_loop:w #1 ;
292   }

```

(End definition for `__tl_analysis_b_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.

```

293 \cs_new_protected:Npn \__tl_analysis_map_inline:nn #1
294 {
295   \__tl_analysis:n {#1}
296   \int_gincr:N \g__prg_map_int
297   \exp_args:Nc \__tl_analysis_map_inline_aux:Nn
298     { \__tl_analysis_map_inline_ \int_use:N \g__prg_map_int :wNw }
299 }
300 \cs_new_protected:Npn \__tl_analysis_map_inline_aux:Nn #1#2
301 {
302   \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
303     {
304       \use_none:n ##2
305       #2
306       #1
307     }
308   \exp_after:wN #1
309     \g__tl_analysis_result_tl
310     \s__tl { ? \tl_map_break: } \s__tl
311   \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
312 }

```

(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
__tl_analysis_show:N

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

```

313 \cs_new_protected:Npn \tl_show_analysis:N #1
314 {
315   \exp_args:No \__tl_analysis:n {#1}
316   \__tl_analysis_show:N #1
317 }
318 \cs_new_protected:Npn \tl_show_analysis:n #1
319 {
320   \__tl_analysis:n {#1}
321   \tl_set:Nn \l__tl_analysis_internal_tl {#1}
322   \__tl_analysis_show:N \l__tl_analysis_internal_tl
323 }
324 \cs_new_protected:Npn \__tl_analysis_show:N #1
325 {
326   \group_begin:
327   \use:x
328     {
329       \group_end:
330       \exp_not:n { \_msg_show_variable:Nnn #1 }
331       { tl-analysis }
332       {
333         \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
334         \s__tl { ? \__prg_break: } \s__tl

```

```

335         \prg_break_point:
336     }
337 }
338 }

```

(End definition for \tl_show_analysis:N and \tl_show_analysis:n. These functions are 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.

```

339 \cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
340 {
341     \use_none:n #2
342     \exp_not:n { \> \ }
343     \if_int_compare:w "#2 = \c_zero
344         \exp_after:wN \__tl_analysis_show_cs:n
345     \else:
346         \if_int_compare:w "#2 = \c_thirteen
347             \exp_after:wN \exp_after:wN
348             \exp_after:wN \__tl_analysis_show_active:n
349         \else:
350             \exp_after:wN \exp_after:wN
351             \exp_after:wN \__tl_analysis_show_normal:n
352         \fi:
353     \fi:
354     {#1}
355     \__tl_analysis_show_loop:wNw
356 }

```

(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.

```

357 \cs_new:Npn \__tl_analysis_show_normal:n #1
358 {
359     \exp_after:wN \token_to_str:N #1 ~
360     ( \exp_after:wN \token_to_meaning:N #1 )
361 }

```

(End definition for __tl_analysis_show_normal:n.)

__tl_analysis_show_value:N

This expands to the value of #1 if it has any.

```

362 \cs_new:Npn \__tl_analysis_show_value:N #1
363 {
364     \token_if_expandable:NF #1
365     {
366         \token_if_chardef:NTF #1 \prg_break: { }
367         \token_if_mathchardef:NTF #1 \prg_break: { }
368         \token_if_dim_register:NTF #1 \prg_break: { }

```



```

369 \token_if_int_register:NTF #1 \prg_break: { }
370 \token_if_skip_register:NTF #1 \prg_break: { }
371 \token_if_toks_register:NTF #1 \prg_break: { }
372 \use_none:nnn
373 \prg_break_point:
374 \use:n { = \tex_the:D #1 }
375 }
376 }

```

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

`\tl_analysis_show_cs:n` 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`.

```

\__tl_analysis_show_active:n
\tl_analysis_show_long:nnn
\tl_analysis_show_long_aux:nnnn
377 \cs_new:Npn \tl_analysis_show_cs:n #1
378 { \exp_args:No \tl_analysis_show_long:nn {#1} { control~sequence= } }
379 \cs_new:Npn \tl_analysis_show_active:n #1
380 { \exp_args:No \tl_analysis_show_long:nn {#1} { active~character= } }
381 \cs_new:Npn \tl_analysis_show_long:nn #1
382 {
383   \tl_analysis_show_long_aux:oofn
384   { \token_to_str:N #1 }
385   { \token_to_meaning:N #1 }
386   { \tl_analysis_show_value:N #1 }
387 }
388 \cs_new:Npn \tl_analysis_show_long_aux:nnnn #1#2#3#4
389 {
390   \int_compare:nNnTF
391   { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
392   > { \l_iow_line_count_int - \c_three }
393   {
394     \str_range:nnn { #1 ~ ( #4 #2 #3 ) } \c_one
395     {
396       \l_iow_line_count_int - \c_three
397       - \str_count:N \c_tl_analysis_show_etc_str
398     }
399     \c_tl_analysis_show_etc_str
400   }
401   { #1 ~ ( #4 #2 #3 ) }
402 }
403 \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.

```

404 \tl_const:Nx \c_tl_analysis_show_etc_str % (
405 { \token_to_str:N \ETC.) }

```

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

```

406 \__msg_kernel_new:nnn { kernel } { show-tl-analysis }
407 {
408   The~token~list~
409   \str_if_eq:nnF {#1} { \l__tl_analysis_internal_tl }
410   { \token_to_str:N #1 ~ }
411   \tl_if_empty:NTF #1
412   { is~empty }
413   { contains~the~tokens: }
414 }
415 </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	
<code>\</code>	342
<code>\^</code>	55, 115, 116, 119, 120
<code>__int_eval:w</code> 24, 175, 236, 248, 252, 256, 279	
<code>__int_eval_end:</code>	252
<code>__int_value:w</code>	24, 36, 193, 234, 285
<code>__msg_kernel_new:nnn</code>	406
<code>__msg_show_variable:Nnn</code>	330
<code>__prg_break:</code>	51, 265, 334, 366, 367, 368, 369, 370, 371
<code>__prg_break_point:</code>	52, 187, 335, 373
<code>__prg_break_point:Nn</code>	311
<code>__scan_new:N</code>	6
<code>__tl_analysis:n</code>	37, 37, 295, 315, 320
<code>__tl_analysis_a:n</code>	42, 63, 63
<code>__tl_analysis_a_bgroup:w</code>	92, 114, 117
<code>__tl_analysis_a_cs:ww</code>	159, 167, 171
<code>__tl_analysis_a_egroup:w</code>	94, 114, 121
<code>__tl_analysis_a_group:nw</code>	114, 118, 122, 124
<code>__tl_analysis_a_group_test:w</code>	114, 131, 136
<code>__tl_analysis_a_loop:w</code>	69, 72, 72, 112, 143, 156, 169
<code>__tl_analysis_a_safe:N</code>	93, 159, 159
<code>__tl_analysis_a_space:w</code>	91, 97, 97
<code>__tl_analysis_a_space_test:w</code> 97, 99, 104	
<code>__tl_analysis_a_store:</code> 108, 139, 145, 145	
<code>__tl_analysis_a_type:w</code>	73, 74, 74
<code>__tl_analysis_b:n</code>	43, 182, 182
<code>__tl_analysis_b_char:Nww</code>	209, 215, 222
<code>__tl_analysis_b_cs:Nww</code>	211, 240, 240
<code>__tl_analysis_b_cs_test:ww</code> 240, 243, 245	
<code>__tl_analysis_b_loop:w</code>	182, 186, 190, 286, 291
<code>__tl_analysis_b_normal:wwN</code>	195, 200, 202, 261
<code>__tl_analysis_b_normals:ww</code>	192, 195, 195, 235, 247
<code>__tl_analysis_b_special:w</code> 198, 258, 260	
<code>__tl_analysis_b_special_char:wN</code>	258, 275, 283
<code>__tl_analysis_b_special_space:w</code>	258, 277, 288
<code>__tl_analysis_cs_space_count:NN</code>	21, 21, 167, 243
<code>__tl_analysis_cs_space_count:w</code>	21, 25, 29, 33
<code>__tl_analysis_cs_space_count_end:w</code>	21, 27, 35
<code>__tl_analysis_disable_loop:N</code>	47, 50, 56, 60
<code>__tl_analysis_extract_charcode:</code>	15, 15, 126
<code>__tl_analysis_extract_charcode_aux:w</code>	15, 17, 20
<code>__tl_analysis_map_inline:nn</code> 1, 293, 293	

