The etl package
expandable token list operations

Jonathan P. Spratte[⌜]

2021-11-07 v0.3

Contents

[⌜]jspratte@yahoo.de

1 Documentation

The etl package provides a few *slow but expandable* alternatives to unexpandable functions found inside the l3tl module of expl3. All user functions must not contain the tokens `\s__etl_stop`[1] or `\__etl_act_result:n`[2] in any argument unless specified otherwise (there might be other forbidden tokens, all of which are internals to this package, and usually shouldn't somehow end up inside the input stream by accident).

There is another limitation of this package: There are tokens which cannot expandably be differentiated from each other, those are active characters let to the same character with a different category code, something like the following:

```
\char_set_catcode_letter:N a
\char_set_active_eq:NN a a
\char_set_catcode_active:N a
```

After this the active 'a's couldn't be told apart from non-active 'a's of category letter by the parsers in this package.[3] In general two tokens are considered equal if `\etl_token_if_eq:NNTF` yields true (see there). Another limitation is that the parser doesn't consider the character code of tokens with category 1 or 2 (group begin and group end, typically {}), instead all tokens found with these two category codes are normalised to $\{_1$ and $\}_2$ (an exception to this rule are the functions `\etl_token_replace_once:nNn` and `\etl_replace_once:nnn` in which this normalisation is only done up to the replacement, and the rest of the input is forwarded unchanged).

The core macro `\etl_act:nnnnnnn` is modelled after an internal of l3tl called `\__tl_act:NNNn` but with some more possibilities added.

[1] At any nesting level of groups

[2] On the top level (so nested usages in groups are fine)

[3] Thanks to Bruno Le Floch for pointing that out.

1.1 A general loop

\etl_act:nnnnnnn ⋆ \etl_act:nnnnnnn {⟨*normal*⟩} {⟨*space*⟩} {⟨*group*⟩} {⟨*final*⟩} {⟨*status*⟩} {⟨*output*⟩} {⟨*token list*⟩}
\etl_act:nnnnnn ⋆ \etl_act:nnnnnn {⟨*normal*⟩} {⟨*space*⟩} {⟨*group*⟩} {⟨*final*⟩} {⟨*status*⟩} {⟨*token list*⟩}
\etl_act:nnnnn ⋆ \etl_act:nnnnn {⟨*normal*⟩} {⟨*space*⟩} {⟨*group*⟩} {⟨*status*⟩} {⟨*token list*⟩}

This function will act on the ⟨*token list*⟩ (somewhat a map_tokens-function). Both ⟨*normal*⟩ and ⟨*group*⟩ should be code that expects two following arguments (the first being the ⟨*status*⟩, the second the next N-type token or the contents of the next group in the ⟨*token list*⟩), and ⟨*space*⟩ should expect only the ⟨*status*⟩ as a following argument.

You can also specify ⟨*final*⟩ code which will receive the ⟨*status*⟩ followed by the output (which you can assign with \etl_act_output:n and \etl_act_output_pre:n), and will be used inside an e-expansion context (so you'll want to protect anything you want to output from further expansion using \exp_not:n). Also you can specify some ⟨*output*⟩ which should be there from the beginning.

Variants without the argument ⟨*output*⟩ will start with an empty output, and those without ⟨*final*⟩ will just output the results at the end.

TEXhackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion. If you don't need the ⟨*final*⟩ argument processor and don't have to reorder some of the output you can also use \exp_not:n to directly output tokens where you currently are, this might be faster.

All other functions which start in \etl_act_ should only be used inside the ⟨*normal*⟩, ⟨*space*⟩, or ⟨*group*⟩ code of \etl_act:nnnnnnn.

\etl_act_output:n ⋆ \etl_act_output:n {⟨*token list*⟩}
\etl_act_output_pre:n ⋆

This will add ⟨*token list*⟩ to the output of \etl_act:nnnnnnn. The normal version will add ⟨*token list*⟩ after the current output, the pre variant will put it before the current output.

\etl_act_output_rest: ⋆ \etl_act_output_rest:
\etl_act_output_rest_pre: ⋆

After this macro was used all remaining things inside the ⟨*token list*⟩ argument of \etl_act:nnnnnnn will be added to the output. The normal version will add it to the end of the output, the pre variant will put the remainder before the current output in reversed order. Any begin group and end group tokens inside ⟨*token list*⟩ will be normalised by this. A faster alternative (that doesn't normalise) would be

```
\etl_act_apply_to_rest:n { \etl_act_break_post:n }
```

\etl_act_status:n ⋆ \etl_act_status:n {⟨*status*⟩}

This will change the current status of \etl_act:nnnnnnn to ⟨*status*⟩.

\etl_act_put_back:n ⋆ `\etl_act_put_back:n {⟨token list⟩}`

You can put ⟨*token list*⟩ back into the input stream to be reconsidered by the current `\etl_act:nnnnnnn` loop (of course this doesn't have to be literally put back, you might add completely new contents with this).

\etl_act_switch:nnn ⋆ `\etl_act_switch:nnn {⟨normal⟩} {⟨space⟩} {⟨group⟩}`
\etl_act_switch_normal:n ⋆ `\etl_act_switch_normal:n {⟨normal⟩}`
\etl_act_switch_space:n ⋆ `\etl_act_switch_space:n {⟨space⟩}`
\etl_act_switch_group:n ⋆ `\etl_act_switch_group:n {⟨group⟩}`

With these functions you can change the provided code to act on ⟨*normal*⟩ (so `N`-type) tokens, ⟨*space*⟩s, and ⟨*group*⟩s.

\etl_act_apply_to_rest:n ⋆ `\etl_act_apply_to_rest:n {⟨code⟩}`

This will fetch the rest of the input ⟨*token list*⟩ of the current `\etl_act:nnnnnnn` call and leave ⟨*code*⟩{⟨*rest*⟩} in the input stream (without brace normalisation). It will not end the current `\etl_act:nnnnnnn` loop (but since the ⟨*token list*⟩ remainder is now empty it'll end after your ⟨*code*⟩ is done, except if you put new contents in the list using `\etl_act_put_back:n`) nor affect it in any other way.

\etl_act_do_final: ⋆ `\etl_act_do_final:`

This will immediately stop the current `\etl_act:nnnnnnn` invocation and execute the provided ⟨*final*⟩ code (or if a variant without the ⟨*final*⟩ code was used, the output). The ⟨*final*⟩ code will receive the current status followed by the current output as two arguments (just like it would when the end of the ⟨*token list*⟩ was reached).

\etl_act_break: ⋆ `\etl_act_break:`
\etl_act_break_discard: ⋆

This will immediately stop the current `\etl_act:nnnnnnn` invocation and leave the current output in the input stream. The `discard` variant will gobble the current output and leave nothing in the input stream.

\etl_act_break:n ⋆ `\etl_act_break:n {⟨token list⟩}`

This will immediately stop the current `\etl_act:nnnnnnn` invocation, gobble the current output and leave ⟨*token list*⟩ in the input stream.

\etl_act_break_pre:n ⋆ `\etl_act_break_pre:n {⟨token list⟩}`
\etl_act_break_post:n ⋆

This will immediately stop the current `\etl_act:nnnnnnn` invocation and leave the current output in the input stream, and in the case of the normal variant followed by ⟨*token list*⟩, in the `pre`-case preceded by ⟨*token list*⟩.

1.1.1 Examples

To give examples how you could use `\etl_act:nnnnnnn` the following could be used to reimplement `\tl_reverse:n`. We work with a bit of currying here (this gives a small speed gain in general, though for code clarity you might decide to not curry each and every argument). Since a reversing function doesn't need to postprocess its output we can use the shorter `\etl_act:nnnnn`. The three internal functions all need to gobble the status.

```
% argument #2 is curried
\cs_new:Npn \__my_reverse_normal:nN #1 { \etl_act_output_pre:n }
\cs_new:Npn \__my_reverse_space:n #1 { \etl_act_output_pre:n { ~ } }
\cs_new:Npn \__my_reverse_group:nn #1#2 { \etl_act_output_pre:n { {#2} } }
% argument #1 is curried
\cs_new:Npn \my_reverse:n
  {
    \etl_act:nnnnn
      \__my_reverse_normal:nN
      \__my_reverse_space:n
      \__my_reverse_group:nn
      {} % empty status
  }
```

We could also create a similar function that'll reverse the input up to the first space and discard the remainder (no idea why somebody would want to do that).

```
% argument #1 is curried
\cs_new:Npn \my_reverse_discard_after_space:n
  {
    \etl_act:nnnnn
      \__my_reverse_normal:nN
      \etl_act_break:
      \__my_reverse_group:nn
      {} % empty status
  }
```

Or a function that reverses the contents of groups as well.

```
\cs_generate_variant:Nn \etl_act_output_pre:n { e }
\cs_new:Npn \__my_reverse_group_reversed:nn #1#2
  { \etl_act_output_pre:e { { \my_reverse_deep:n {#2} } } }
% argument #1 is curried
\cs_new:Npn \my_reverse_deep:n
  {
    \etl_act:nnnnn
      \__my_reverse_normal:nN
      \__my_reverse_space:n
      \__my_reverse_group_reversed:nn
      {} % empty status
  }
```

Another function could count a specific token inside a token list. Since for this neither output-reordering nor post processing is needed we shortcut by not using `\etl_output:n` but instead directly leaving `+ \c_one_int` (which also doesn't need to be protected against further expansion) in the input.

```
\cs_new:Npn \__my_count_token:NN #1#2
  { \etl_token_if_eq:NNT #1#2 { + \c_one_int } }
\cs_new:Npn \my_count_token:Nn #1#2
  {
```

```
    \int_eval:n
      {
        \c_zero_int
        \etl_act:nnnnn
          \__my_count_token:NN
          \use_none:n
          \use_none:nn
          {#1}
          {#2}
      }
  }
```

As a last example we reimplement the `\etl_token_replace_once:nNn` function. The function doesn't need to reorder any tokens, so we shortcut with `\exp_not:n` to output things in place. We put the token we want to replace in the code for `N`-type processing and the replacement in the status. When we found the token we want to replace we put our replacement there and output the rest unaltered.

```
\cs_new:Npn \my_replace_token:nNn #1#2#3
  {
    \etl_act:nnnnn
      { \__my_replace_token:NnN #2 }
      { ~ \use_none:n }
      { \__my_replace_token:nn }
      {#3}
      {#1}
  }
\cs_new:Npn \__my_replace_token:nn #1#2 { { \exp_not:n {#2} } }
\cs_new:Npn \__my_replace_token:NnN #1#2#3
  {
    \etl_token_if_eq:NNTF #1#3
      { \exp_not:n {#2} \etl_act_apply_to_rest:n { \etl_act_break:n } }
      { \exp_not:N #3 }
  }
```

I hope this gave you at least an idea on how to use the loop and that the explanations on the functions not used in these examples suffice to also give you an idea on how to use them.

1.2 Conditionals

`\etl_token_if_eq_p:NN` ⋆ `\etl_token_if_eq_p:NN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
`\etl_token_if_eq:NNTF` ⋆ `\etl_token_if_eq:NNTF` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares ⟨*token*$_1$⟩ and ⟨*token*$_2$⟩ and yields `true` if the two are equal. Two tokens are considered equal if they have the same meaning (so if `\if_meaning:w` is true) and the same string representation (so if `\str_if_eq:nnTF` is true).

`\etl_token_if_in_p:nN` ⋆ `\etl_token_if_in_p:nN` {⟨*token list*⟩} ⟨*token*⟩
`\etl_token_if_in:nNTF` ⋆ `\etl_token_if_in:nNTF` {⟨*token list*⟩} ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Searches for ⟨*token*⟩ inside the ⟨*token list*⟩. If it is found returns true. Brace groups inside the ⟨*token list*⟩ are ignored.

\etl_token_if_in_deep_p:nN ⋆ \etl_token_if_in_deep_p:nN {⟨token list⟩} ⟨token⟩
\etl_token_if_in_deep:nNTF ⋆ \etl_token_if_in_deep:nNTF {⟨token list⟩} ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Searches for ⟨token⟩ inside the ⟨token list⟩. If it is found returns true. Brace groups inside the ⟨token list⟩ are recursively searched as well.

\etl_if_eq_p:nn ⋆ \etl_if_eq_p:nn {⟨token list$_1$⟩} {⟨token list$_2$⟩}
\etl_if_eq:nnTF ⋆ \etl_if_eq:nnTF {⟨token list$_1$⟩} {⟨token list$_2$⟩} {⟨true code⟩} {⟨false code⟩}

Compares ⟨token list$_1$⟩ and ⟨token list$_2$⟩ with each other on a token-by-token basis. Keep in mind that there are tokens which can't be told apart from each other, and that groups are normalised. If both token lists match (modulo the mentioned limitations) the ⟨true code⟩ is left in the input stream, else the ⟨false code⟩.

\etl_if_in_p:nn ⋆ \etl_if_in_p:nn {⟨token list⟩} {⟨search text⟩}
\etl_if_in:nnTF ⋆ \etl_if_in:nnTF {⟨token list⟩} {⟨search text⟩} {⟨true code⟩} {⟨false code⟩}

Searches for ⟨search text⟩ inside of ⟨token list⟩. If it is found the ⟨true code⟩ is left in the input stream, else the ⟨false code⟩. Both macro parameter tokens as well as tokens with category code 1 and 2 (normally {}) can be part of ⟨search text⟩ (unlike for the similar function \tl_if_in:nnTF). Material inside of groups in ⟨token list⟩ is ignored (except for groups contained in ⟨search text⟩). So the following would first yield true and then false:

```
\etl_if_in:nnTF { a{b{c}} } { {b{c}} } { true } { false }
\etl_if_in:nnTF { a{b{c}} } {  b{c}  } { true } { false }
```

\etl_if_in_deep_p:nn ⋆ \etl_if_in_deep_p:nn {⟨token list⟩} {⟨search text⟩}
\etl_if_in_deep:nnTF ⋆ \etl_if_in_deep:nnTF {⟨token list⟩} {⟨search text⟩} {⟨true code⟩} {⟨false code⟩}

Does the same as \etl_if_in:nnTF but also recursively searches inside of groups in ⟨token list⟩. So this would yield true in both of the cases in above example.

1.3 Modifying token lists

\etl_token_replace_once:nNn ⋆ \etl_token_replace_once:nNn {⟨token list⟩} ⟨token⟩ {⟨replacement⟩}

This function will replace the first occurrence of ⟨token⟩ inside of ⟨token list⟩ that is not hidden inside a group with ⟨replacement⟩. The ⟨token⟩ has to be a valid N-type argument.

TEXhackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

\etl_token_replace_all:nNn ⋆ `\etl_token_replace_all:nNn {⟨token list⟩} ⟨token⟩ {⟨replacement⟩}`

This function will replace each occurrence of ⟨*token*⟩ inside of ⟨*token list*⟩ that is not hidden inside a group with ⟨*replacement*⟩. The ⟨*token*⟩ has to be a valid N-type argument.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

\etl_token_replace_all_deep:nNn ⋆ `\etl_token_replace_all_deep:nNn {⟨token list⟩} ⟨token⟩ {⟨replacement⟩}`

This function will replace each occurrence of ⟨*token*⟩ inside of ⟨*token list*⟩ with ⟨*replacement*⟩. The ⟨*token*⟩ has to be a valid N-type argument.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

\etl_replace_once:nnn ⋆ `\etl_replace_once:nnn {⟨token list⟩} {⟨search text⟩} {⟨replacement⟩}`

This function will replace the first occurrence of ⟨*search text*⟩ inside of ⟨*token list*⟩ that is not hidden inside a group with ⟨*replacement*⟩.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

\etl_replace_all:nnn ⋆ `\etl_replace_all:nnn {⟨token list⟩} {⟨search text⟩} {⟨replacement⟩}`

This function will replace all occurrences of ⟨*search text*⟩ inside of ⟨*token list*⟩ that are not hidden inside a group with ⟨*replacement*⟩.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

\etl_replace_all_deep:nnn ⋆ `\etl_replace_all_deep:nnn {⟨token list⟩} {⟨search text⟩} {⟨replacement⟩}`

This function will replace all occurrences of ⟨*search text*⟩ inside of ⟨*token list*⟩ with ⟨*replacement*⟩.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

1.4 New expandable functions

Functions generated with the means in this section are roughly as fast as the l3tl variants of them (there might be performance differences; in any case they are faster than the generic functions above), but have at least one fixed argument. They don't have the drawback of not being able to tell apart an active character from a token with the same character code and different category code if the active character was let to it and they don't normalise braces to $\{_1$ and $\}_2$.

1.4.1 Conditionals

`\etl_new_if_in:Nnn` `\etl_new_if_in:Nnn` ⟨*function*⟩ {⟨*search text*⟩} {⟨*conditions*⟩}

This will define a new ⟨*function*⟩ which will act as a conditional and search for ⟨*search text*⟩ inside of an `n`-type argument completely expandable. The ⟨*conditions*⟩ should be a comma-separated list containing one or more of `p`, `T`, `F` and `TF` (just like for `\prg_new_conditional:Npnn`). The ⟨*search text*⟩ must not contain tokens with category code 1 or 2 (normally {}) and can't contain macro parameter tokens (normally #). Unlike for the conditionals in subsection 1.2, the ⟨*search text*⟩ of functions created with `\etl_new_if_in:Nnn` might contain `\s__etl_stop` tokens.

So the following would yield `true` followed by `false`:

```
\etl_new_if_in:Nnn \my_if_a_in:n { a } { TF }
\my_if_a_in:nTF { a text } { true } { false }
\my_if_a_in:nTF {   text } { true } { false }
```

1.4.2 Modifiers

`\etl_new_replace_once:Nn` `\etl_new_replace_once:Nn` ⟨*function*⟩ {⟨*search text*⟩}

This defines a new ⟨*function*⟩ that'll accept two arguments (the first being a token list, the second a replacement). The generated ⟨*function*⟩ will replace the first occurrence of ⟨*search text*⟩ inside the token list with replacement. It'll ignore things hidden inside a group in the token list. Neither the ⟨*search text*⟩ nor the token list given to the generated ⟨*function*⟩ can contain `\s__etl_stop` (this would result in undefined behaviour), the given replacement on the other hand might contain that token. Additionally ⟨*search text*⟩ can't contain tokens of category group begin or group end (usually { and }) or macro parameters (usually #).

TEXhackers note: The result of ⟨*function*⟩ is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`- or `e`-type argument expansion. The result will be returned after exactly two steps of expansion.

So the following would yield `AcDC`:

```
\etl_new_replace_once:Nn \my_replace_C_once:nn { C }
\my_replace_C_once:nn { ACDC } { c }
```

\etl_new_replace_all:Nn

\etl_new_replace_all:Nn ⟨*function*⟩ {⟨*search text*⟩}

This behaves like \etl_new_replace_once:Nn, but the ⟨*function*⟩ will replace all occurrences of ⟨*search text*⟩ instead of just the first.

TEXhackers note: The result of ⟨*function*⟩ is returned within \exp_not:n, which means that the token list does not expand further when appearing in an x- or e-type argument expansion. The result will be returned after exactly two steps of expansion.

So the following would yield AcDc:

```
\etl_new_replace_all:Nn \my_replace_C_all:nn { C }
\my_replace_C_all:nn { ACDC } { c }
```

1.5 Bugs and Feature Requests

If you find bugs or want to request features you can do so either via email (see the first page) or via Github at https://github.com/Skillmon/ltx_etl/issues.

2 Implementation

```
⟨*pkg⟩
⟨@@=etl⟩
```

Tell who we are:

```
\ProvidesExplPackage{etl}
  {2021-11-07} {0.3}
  {expandable token list manipulation}
```

Ensure dependencies are met:

```
\cs_if_exist:NF \tex_expanded:D
  {
    \msg_new:nnn { etl } { expanded-missing }
      { The~ expanded~ primitive~ is~ required. }
    \msg_fatal:nn { etl } { expanded-missing }
  }
```

2.1 Primitives

`\__etl_expanded:w` `\__etl_unexpanded:w` `\__etl_detokenize:w`

Private copies of a few primitives (evil code for expl3).

```
\cs_new_eq:NN \__etl_expanded:w \tex_expanded:D
\cs_new_eq:NN \__etl_unexpanded:w \tex_unexpanded:D
\cs_new_eq:NN \__etl_detokenize:w \tex_detokenize:D
```

(*End definition for* `\__etl_expanded:w`, `\__etl_unexpanded:w`, *and* `\__etl_detokenize:w`.)

2.2 Variables

`\s__etl_stop` `\s__etl_mark`

Scan marks.

```
\scan_new:N \s__etl_stop
\scan_new:N \s__etl_mark
```

(*End definition for* `\s__etl_stop` *and* `\s__etl_mark`.)

2.3 Small auxiliaries

`\__etl_split_first:w`

Can be used to extract the first element from a token list, it should always be used like this: `\exp_after:wN` ⟨*function*⟩ `\@@_expanded:w { \@@_split_first:w` ⟨*arg*⟩ `}`.

```
\cs_new:Npn \__etl_split_first:w #1
  {
    { \__etl_unexpanded:w {#1} }
    \if_false: { \fi: \exp_after:wN } \exp_after:wN { \if_false: } \fi:
  }
```

(*End definition for* `\__etl_split_first:w`.)

`\__etl_turn_true:w` `\__etl_fi_turn_false:w`

Fast ways to change the outcome of a test.

```
\cs_new:Npn \__etl_turn_true:w \if_false: { \if_true: }
\cs_new:Npn \__etl_fi_turn_false:w \fi: \if_true: { \fi: \if_false: }
```

(*End definition for* `\__etl_turn_true:w` *and* `\__etl_fi_turn_false:w`.)

`\__etl_rm_space:w`

Fast macro to gobble an immediately following single space.

```
\use:n { \cs_new:Npn \__etl_rm_space:w } ~ {}
```

(*End definition for* `\__etl_rm_space:w`.)

`\__etl_if_empty:nTF` `\__etl_if_empty:nT` `\__etl_if_empty:w` `\__etl_if_empty_true:w` `\__etl_if_empty_true_TF:w`

This is a fast test whether something is empty or not. The argument must not contain `\s__etl_stop` for this to work (but since that limitation is true for most if not all user facing functions of this module, this is fine to gain a bit of speed).

```
\cs_new:Npn \__etl_if_empty:nT #1
  {
    \__etl_if_empty:w
      \s__etl_stop #1 \s__etl_stop \__etl_if_empty_true:w
      \s__etl_stop \s__etl_stop \use_none:n
  }
\cs_new:Npn \__etl_if_empty:nTF #1
  {
    \__etl_if_empty:w
      \s__etl_stop #1 \s__etl_stop \__etl_if_empty_true_TF:w
      \s__etl_stop \s__etl_stop \use_ii:nn
  }
\cs_new:Npn \__etl_if_empty:w #1 \s__etl_stop \s__etl_stop {}
\cs_new:Npn \__etl_if_empty_true:w \s__etl_stop \s__etl_stop \use_none:n #1
  {#1}
\cs_new:Npn \__etl_if_empty_true_TF:w \s__etl_stop \s__etl_stop \use_ii:nn #1#2
  {#1}
```

(*End definition for* `\__etl_if_empty:nTF` *and others.*)

`\__etl_if_head_is_group:nTF` `\__etl_if_head_is_group:nT`

This test works pretty much the same way `\tl_if_head_is_group:nTF` works, but it is faster because it gets rid of the unnecessary `\if:w` and instead only works by argument gobbling. Drawback is that if you only expand the macro twice you could end up with unbalanced braces.

```
\cs_new:Npn \__etl_if_head_is_group:nTF #1
  {
    \exp_after:wN \use_none:n \exp_after:wN
      {
        \exp_after:wN { \token_to_str:N #1 ? }
        \exp_after:wN \use_iii:nnnn \token_to_str:N
      }
    \use_ii:nn
  }
\cs_new:Npn \__etl_if_head_is_group:nT #1
  {
    \exp_after:wN \use_none:n \exp_after:wN
      {
        \exp_after:wN { \token_to_str:N #1 ? }
        \exp_after:wN \use_iii:nnn \token_to_str:N
      }
    \use_none:n
  }
```

(*End definition for* `\__etl_if_head_is_group:nTF` *and* `\__etl_if_head_is_group:nT`.)

2.4 The act loop

\etl_act:nnnnnnn
\etl_act:nnnnnn
\etl_act:nnnnn
__etl_act:nnnnnnn
__etl_act_just_result:nn

The act loop is modelled after the expl3 internal __tl_act:NNNn but with a few more features making it more general. Those are (argument to \etl_act:nnnnnnn in parentheses): a status (#5), n-type mapping instead of just N-type functions, a final result processing (#4), and the possibility to preset some output (#6). The other arguments are: #7 the token list on which we should act, #1 function to use for N-type elements in that list, #2 function to use for spaces in that list, and #3 function to use on groups in that list.

Just like the __tl_act:NNNn function, this has a token which must not occur in the arguments, in this case that token is \s__etl_stop. The result is stored as an argument to the (undefined) function __etl_act_result:n.

```
\cs_new:Npn \etl_act:nnnnnnn #1#2#3#4#5#6#7
  {
    \__etl_unexpanded:w \__etl_expanded:w
      {{
        \__etl_act:w #7 {\s__etl_stop} . \s__etl_stop {#5} {#1} {#2} {#3}
        \__etl_act_result:n {#6} {#4}
      }}
  }
```

We also provide a version without the __etl_unexpanded:w around it for internal purposes, in which we'd otherwise have to remove it for correct behaviour. For that we use \use_ii_iii:nnn which will remove one set of braces and the __etl_unexpanded:w.

```
\exp_args:NNno \exp_args:Nno \use:n
  { \cs_new:Npn \__etl_act:nnnnnnn #1#2#3#4#5#6#7 }
  {
    \exp_after:wN \use_ii_iii:nnn
    \etl_act:nnnnnnn {#1} {#2} {#3} {#4} {#5} {#6} {#7}
  }
```

We also provide two reduced function variants, the first without presetting some output, the second also without the final processor.

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_act:nnnnnn #1#2#3#4#5#6 }
  { \etl_act:nnnnnnn {#1} {#2} {#3} {#4} {#5} {} {#6} }
\exp_args:Nno \use:n { \cs_new:Npn \etl_act:nnnnn #1#2#3#4#5 }
  { \etl_act:nnnnnnn {#1} {#2} {#3} \__etl_act_just_result:nn {#4} {} {#5} }
```

The final processor is provided with two n-type arguments (both in braces) the first being the status, the second the output. To just get the output we gobble the status and put __etl_unexpanded:w there to protect the final output from further expanding.

```
\cs_new:Npn \__etl_act_just_result:nn #1 { \__etl_unexpanded:w }
```

(*End definition for* \etl_act:nnnnnnn *and others. These functions are documented on page* 3.)

__etl_if_head_is_space:nTF
__etl_if_head_is_space_true:w
__etl_if_head_is_N_type:nTF
__etl_if_head_is_N_type_false:w
__etl_act:w
__etl_act_if_space:w
__etl_act_space:w

We need a few macros with spaces at weird places so define them here. Since we got the limitation of not allowing \s__etl_stop we can use that token to get some fast tests. The first tests for a space at the front, and since that one is pretty fast we can use it to build a faster alternative to check for a starting N-type as well (with the drawback that this would yield true for an empty argument, something we have to keep in mind).

```
\group_begin:
  \cs_set:Npn \__etl_tmp:n #1
    {
      \cs_new:Npn \__etl_if_head_is_space:nTF ##1
```

```
      {
        \__etl_act_if_space:w
          \s__etl_stop ##1 \s__etl_stop \__etl_if_head_is_space_true:w
          \s__etl_stop #1  \s__etl_stop \use_ii:nn
      }
    \cs_new:Npn \__etl_if_head_is_space_true:w
        \s__etl_stop #1 \s__etl_stop \use_ii:nn ##1##2
      {##1}
    \cs_new:Npn \__etl_if_head_is_N_type:nTF ##1
      {
        \__etl_act_if_space:w
          \s__etl_stop ##1 \s__etl_stop \__etl_if_head_is_N_type_false:w
          \s__etl_stop #1  \s__etl_stop
        \__etl_if_head_is_group:nT {##1} \use_iii:nnn
        \use_i:nn
      }
    \cs_new:Npn \__etl_if_head_is_N_type_false:w
        \s__etl_stop #1 \s__etl_stop
        \__etl_if_head_is_group:nT ##1 \use_iii:nnn
        \use_i:nn
        ##2##3
      {##3}
```

The act loop `\__etl_act:w` grabs the remainder of the list, delimited by `\s__etl_stop`, picks up the status (`##2`), and the user provided functions for `N`-types (`##3`), spaces (`##4`), and groups (`##5`). We need to check which type is at the head of the token list (the space test is a bit stripped down, and very fast this way).

```
    \cs_new:Npn \__etl_act:w ##1 \s__etl_stop ##2##3##4##5
      {
        \__etl_act_if_space:w
          \s__etl_stop ##1 \s__etl_stop \__etl_act_space:w {##4}
          \s__etl_stop #1  \s__etl_stop
        \__etl_if_head_is_group:nT {##1} \__etl_act_group:w
        \__etl_act_normal:w {##3} {##5}
        {##2} ##1 \s__etl_stop {##2} {##3} {##4} {##5}
      }
```

The check for spaces just gobbles everything up to the first `\s__etl_stop␣`. If we found a space at the head we remove that space and leave in the input the space function, the status, and `\__etl_act:w` for the next iteration.

```
    \cs_new:Npn \__etl_act_if_space:w ##1 \s__etl_stop #1 ##2 \s__etl_stop {}
    \cs_new:Npn \__etl_act_space:w
        ##1 \s__etl_stop #1 \s__etl_stop
        \__etl_if_head_is_group:nT ##2 \__etl_act_group:w \__etl_act_normal:w ##3 ##4
        ##5 #1
      { ##1 {##5} \__etl_act:w }
  }
 \__etl_tmp:n { ~ }
\group_end:
```

(*End definition for* `\__etl_if_head_is_space:nTF` *and others.*)

`\__etl_act_normal:w` For a normal token we can act quite easy, just pick up that token and leave the next iteration in the input stream (`#2` is the group code, which is gobbled).

```
\cs_new:Npn \__etl_act_normal:w #1#2#3#4 { #1 {#3} #4 \__etl_act:w }
```

(*End definition for* `\__etl_act_normal:w`.)

`\__etl_act_group:w`
`\__etl_act_if_end:w`

Since the end marker is a single `\s__etl_stop` in a group, we have to test whether that end marker is found. The test here leads to undefined behaviour if the user supplied token list contains such a marker at an any point. If the end marker is found we call the final handler (for which we have to remove the `\s__etl_stop` to correctly grab its arguments), else we provide the user supplied function the next group and input the next iteration. `#1` is the normal function, which is gobbled.

```
\cs_new:Npn \__etl_act_group:w \__etl_act_normal:w #1#2#3#4
  {
    \__etl_act_if_end:w #4 \use_i:nn \etl_act_do_final: \s__etl_stop
    #2 {#3} {#4} \__etl_act:w
  }
\cs_new:Npn \__etl_act_if_end:w #1 \s__etl_stop {}
```

(*End definition for* `\__etl_act_group:w` *and* `\__etl_act_if_end:w`.)

`\etl_act_output:n`
`\etl_act_output_pre:n`
`\etl_act_output_rest:`
`\etl_act_output_rest_pre:`
`\__etl_act_output_normal:nN`
`\__etl_act_output_space:n`
`\__etl_act_output_group:nn`
`\__etl_act_output_normal_pre:nN`
`\__etl_act_output_space_pre:n`
`\__etl_act_output_group_pre:nn`
`\__etl_act_unexpanded_normal:nN`
`\__etl_act_unexpanded_space:n`
`\__etl_act_unexpanded_group:nn`

To allow reordering the output we unfortunately can't just use `\__etl_unexpanded:w` and be done, so we have to shift a few tokens around instead. All the output macros work by the same idea, except for `\etl_act_output_rest:` and `\etl_act_output_rest_pre:`, since it's a non-trivial task to get the remainder of the argument. Instead these two swap out the user provided functions for some that only pass through the input as output, for which we need six internal output macros.

In those cases in which we don't need reordering, we can internally shortcut using `\__etl_unexpanded:w`.

```
\cs_new:Npn \etl_act_output:n #1 #2 \__etl_act_result:n #3
  { #2 \__etl_act_result:n { #3 #1 } }
\cs_new:Npn \etl_act_output_pre:n #1 #2 \__etl_act_result:n #3
  { #2 \__etl_act_result:n { #1 #3 } }
\cs_new:Npn \etl_act_output_rest: #1 \s__etl_stop #2#3#4#5
  {
    #1 \s__etl_stop {#2}
    \__etl_act_output_normal:nN \__etl_act_output_space:n \__etl_act_output_group:nn
  }
\cs_new:Npn \etl_act_output_rest_pre: #1 \s__etl_stop #2#3#4#5
  {
    #1 \s__etl_stop {#2}
    \__etl_act_output_normal_pre:nN
    \__etl_act_output_space_pre:n
    \__etl_act_output_group_pre:nn
  }
\cs_new:Npn \__etl_act_output_normal:nN #1#2 #3 \__etl_act_result:n #4
  { #3 \__etl_act_result:n { #4 #2 } }
\cs_new:Npn \__etl_act_output_space:n #1 #2 \__etl_act_result:n #3
  { #2 \__etl_act_result:n { #3 ~ } }
\cs_new:Npn \__etl_act_output_group:nn #1#2 #3 \__etl_act_result:n #4
  { #3 \__etl_act_result:n { #4 {#2} } }
\cs_new:Npn \__etl_act_output_normal_pre:nN #1#2 #3 \__etl_act_result:n #4
  { #3 \__etl_act_result:n { #2 #4 } }
\cs_new:Npn \__etl_act_output_space_pre:n #1 #2 \__etl_act_result:n #3
  { #2 \__etl_act_result:n { ~ #3 } }
\cs_new:Npn \__etl_act_output_group_pre:nn #1#2 #3 \__etl_act_result:n #4
  { #3 \__etl_act_result:n { {#2} #4 } }
```

```
\cs_new:Npn \__etl_act_unexpanded_normal:nN #1 { \exp_not:N }
\cs_new:Npn \__etl_act_unexpanded_space:n #1 { ~ }
\cs_new:Npn \__etl_act_unexpanded_group:nn #1#2 { { \__etl_unexpanded:w {#2} } }
```

(*End definition for* `\etl_act_output:n` *and others. These functions are documented on page* 3.)

`\__etl_act_unexpanded_rest:w`
`\__etl_act_unexpanded_rest_aux:w`
`\__etl_act_unexpanded_rest_aux:n`
`\__etl_act_unexpanded_rest_done:w`

This function tries to shortcut as much as possible to output the remainder of the token list in an unchanged way. This can be done by grabbing everything up to the next opening brace (since the stop marker is contained in a group), outputting that, and checking whether we're done. If not output the next group and loop. The first step removes the act code.

Drawback of this is that spaces and braces are not normalised for the remainder of that call (which is what we had documented).

```
\cs_new:Npn \__etl_act_unexpanded_rest:w #1 \__etl_act:w #2#
  {
    \__etl_unexpanded:w {#2}
    \__etl_act_unexpanded_rest_aux:n
  }
\cs_new:Npn \__etl_act_unexpanded_rest_aux:w #1#
  {
    \__etl_unexpanded:w {#1}
    \__etl_act_unexpanded_rest_aux:n
  }
\cs_new:Npn \__etl_act_unexpanded_rest_aux:n #1
  {
    \__etl_act_if_end:w #1 \__etl_act_unexpanded_rest_done:w \s__etl_stop
    { \__etl_unexpanded:w {#1} }
    \__etl_act_unexpanded_rest_aux:w
  }
\cs_new:Npn \__etl_act_unexpanded_rest_done:w
    \s__etl_stop #1 \__etl_act_unexpanded_rest_aux:w
    . \s__etl_stop #2#3#4#5
    \__etl_act_result:n #6#7
  {}
```

(*End definition for* `\__etl_act_unexpanded_rest:w` *and others.*)

`\etl_act_status:n`

Just switch out the status which is stored immediately after `\s__etl_stop`.

```
\cs_new:Npn \etl_act_status:n #1 #2 \s__etl_stop #3
  { #2 \s__etl_stop {#1} }
```

(*End definition for* `\etl_act_status:n`. *This function is documented on page* 3.)

`\etl_act_put_back:n`

Place the first argument after the next iteration of the loop. This macro might strip a set of braces around #2, because it could happen that the user provided code only leaves one group between this functions argument and `\__etl_act:w`, but that would arguably be wrong input anyway, an easy fix would be to use

```
\cs_new:Npn \etl_act_put_back:n #1
  { \@@_act_put_back:nw {#1} \prg_do_nothing: }
\cs_new:Npn \@@_act_put_back:nw #1 #2 \@@_act:w { #2 \@@_act:w #1 }
```

instead of:

```
\cs_new:Npn \etl_act_put_back:n #1 #2 \__etl_act:w { #2 \__etl_act:w #1 }
```

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

`\etl_act_switch:nnn`
`\etl_act_switch_normal:n`
`\etl_act_switch_space:n`
`\etl_act_switch_group:n`

Pretty straight forward, just switch out the user provided functions for the new argument.

```
\cs_new:Npn \etl_act_switch:nnn #1#2#3 #4 \s__etl_stop #5#6#7#8
  { #4 \s__etl_stop {#5} {#1} {#2} {#3} }
\cs_new:Npn \etl_act_switch_normal:n #1 #2 \s__etl_stop #3#4
  { #2 \s__etl_stop {#3} {#1} }
\cs_new:Npn \etl_act_switch_space:n #1 #2 \s__etl_stop #3#4#5
  { #2 \s__etl_stop {#3} {#4} {#1} }
\cs_new:Npn \etl_act_switch_group:n #1 #2 \s__etl_stop #3#4#5#6
  { #2 \s__etl_stop {#3} {#4} {#5} {#1} }
```

(*End definition for* `\etl_act_switch:nnn` *and others. These functions are documented on page* 4.)

`\etl_act_do_final:`
`\etl_act_break:`
`\etl_act_break_discard:`
`\etl_act_break:n`
`\etl_act_break_pre:n`
`\etl_act_break_post:n`

These are different forms to end the loop. The first will gobble the remainder and apply the final action on the token list currently stored for output.

The `break` variants will gobble the final action and output what's currently there (except for the `discard` variant).

```
\cs_new:Npn \etl_act_do_final: #1 \s__etl_stop #2#3 \__etl_act_result:n #4#5
  { #5 {#2} {#4} }
\cs_new:Npn \etl_act_break: #1 \__etl_act_result:n #2#3 { \__etl_unexpanded:w {#2} }
\cs_new:Npn \etl_act_break_discard: #1 \__etl_act_result:n #2#3 {}
\cs_new:Npn \etl_act_break:n #1 #2 \__etl_act_result:n #3#4
  { \__etl_unexpanded:w {#1} }
\cs_new:Npn \etl_act_break_pre:n #1 #2 \__etl_act_result:n #3#4
  { \__etl_unexpanded:w { #1 #3 } }
\cs_new:Npn \etl_act_break_post:n #1 #2 \__etl_act_result:n #3#4
  { \__etl_unexpanded:w { #3 #1 } }
```

(*End definition for* `\etl_act_do_final:` *and others. These functions are documented on page* 4.)

`\etl_act_apply_to_rest:n`

This function can be used to get the remainder of the input ⟨*token list*⟩ and apply some user code on it. To get past the user code we use `\__etl_expanded:w` with a brace trick. The heavy lifting is done by `\__etl_act_get_rest:w`. The `\prg_do_nothing:` prevents accidental brace stripping and will be removed by `\__etl_act_get_rest:w`.

```
\cs_new:Npn \etl_act_apply_to_rest:n #1
  {
    \__etl_expanded:w { \__etl_unexpanded:w {#1} { \if_false: }} \fi:
    \__etl_act_get_rest:w \prg_do_nothing:
  }
```

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

`\__etl_act_get_rest:w`

This macro should be used with two unbalanced opening braces before it and inside an `\__etl_expanded:w` context with a `\prg_do_nothing:` following it to prevent accidental brace stripping. It'll expand to the remainder of the input ⟨*token list*⟩ followed by a closing brace and leaves everything else untouched.

`\__etl_act_get_rest_aux:nw`
`\__etl_act_get_rest_aux:nn`
`\__etl_act_get_rest_done:w`

```
\cs_new:Npn \__etl_act_get_rest:w #1 \__etl_act:w #2#
  {
    \__etl_unexpanded:w {#2}
    \__etl_act_get_rest_aux:nn {#1}
```

```
  }
\cs_new:Npn \__etl_act_get_rest_aux:nw #1 #2#
  {
    \__etl_unexpanded:w {#2}
    \__etl_act_get_rest_aux:nn {#1}
  }
```

Each group might be the end marker, so we check that, else leave the group there and grab until the next group starts.

```
\cs_new:Npn \__etl_act_get_rest_aux:nn #1#2
  {
    \__etl_act_if_end:w #2 \__etl_act_get_rest_done:w \s__etl_stop
    { \__etl_unexpanded:w {#2} }
    \__etl_act_get_rest_aux:nw {#1}
  }
```

This closes the two unbalanced opening braces (ending the `\__etl_expanded:w` context) and puts the remaining code back in the input stream (as well as the now empty next `\__etl_act:w` step). The `\exp_after:wN` removes the leading `\prg_do_nothing:`.

```
\cs_new:Npn \__etl_act_get_rest_done:w \s__etl_stop #1 \__etl_act_get_rest_aux:nw #2
  {
    \if_false: {{ \fi: } \exp_after:wN }
    #2
    \__etl_act:w { \s__etl_stop }
  }
```

(*End definition for* `\__etl_act_get_rest:w` *and others.*)

2.5 Expandable tests

`\etl_token_if_eq_p:NN`
`\etl_token_if_eq:NNTF`

We consider two tokens equal when they have the same meaning and the same string representation. This isn't always correct. If an active character is let to the same character with a different category code those two tokens aren't distinguishable by expansion, *afaik*. To get the optimisation of `\prg_new_conditional:Npnn` we use `\if_false:` and turn it true if both tests are true (this is easier than coding all four variants by hand, even though that could give slightly better performance). The exception being the `TF` variant, since that is used in the inner loop of many functions. The braces around the arguments of `\token_if_eq_meaning:NNT` are necessary because of the first step of expansion applied to that function.

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_token_if_eq:NNTF #1#2 }
  {
    \token_if_eq_meaning:NNT {#1} {#2} { \str_if_eq:nnT #1#2 \use_ii:nnn }
    \use_ii:nn
  }
\exp_args:Nno
\use:n { \prg_new_conditional:Npnn \etl_token_if_eq:NN #1#2 { T , F , p } }
  {
    \token_if_eq_meaning:NNT {#1} {#2} { \str_if_eq:nnT #1#2 \__etl_turn_true:w }
    \if_false:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

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

`\etl_token_if_in_p:nN`
`\etl_token_if_in:nNTF`
`\__etl_token_if_in:NnN`

Searching for just a single token is rather easy, we just loop over the list and compare the N-type tokens to the one token provided. If we find a match we break and return `true`, else we'll return `false` eventually.

```
\exp_args:Nno
\use:n { \prg_new_conditional:Npnn \etl_token_if_in:nN #1#2 { TF , T , F , p } }
  {
    \__etl_act:nnnnnnn
        \__etl_token_if_in:NN \use_none:n \use_none:nn
        \__etl_act_just_result:nn
        {#2}
        \if_false:
        {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_token_if_in:NN #1#2 }
  {
    \etl_token_if_eq:NNTF {#1} {#2} { \etl_act_break:n \if_true: } {}
  }
```

(*End definition for* `\etl_token_if_in:nNTF` *and* `\__etl_token_if_in:NnN`. *This function is documented on page 6.*)

`\etl_token_if_in_deep_p:nN`
`\etl_token_if_in_deep:nNTF`
`\__etl_token_if_in_deep:Nn`

The deep variant just has to recursively call itself on groups to also search those.

```
\exp_args:Nno \use:n
  { \prg_new_conditional:Npnn \etl_token_if_in_deep:nN #1#2 { TF , T , F , p } }
  {
    \__etl_act:nnnnnnn
        \__etl_token_if_in:NN \use_none:n \__etl_token_if_in_deep:Nn
        \__etl_act_just_result:nn
        {#2}
        \if_false:
        {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_token_if_in_deep:Nn #1#2 }
  { \etl_token_if_in_deep:nNT {#2} {#1} { \etl_act_break:n \if_true: } }
```

(*End definition for* `\etl_token_if_in_deep:nNTF` *and* `\__etl_token_if_in_deep:Nn`. *This function is documented on page 7.*)

`\etl_if_eq_p:nn`
`\etl_if_eq:nnTF`
`\__etl_if_eq_normal:nN`
`\__etl_if_eq_normal:NnN`
`\__etl_if_eq_space:n`
`\__etl_if_eq_group:nn`
`\__etl_if_eq_group:nnn`
`\__etl_if_eq_final:nn`

The test needs to compare the full lists on a token-by-token basis. One of the two lists is stored inside the status the other is processed. The `act` code will then leave either `\if_false:` or `\if_true:` in the input stream.

```
\exp_args:Nno
\use:n { \prg_new_conditional:Npnn \etl_if_eq:nn #1#2 { TF , T , F , p } }
  {
```

```
    \__etl_act:nnnnnnn
        \__etl_if_eq_normal:nN
        \__etl_if_eq_space:n
        \__etl_if_eq_group:nn
        \__etl_if_eq_final:nn
        {#2}
        {}
        {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

To compare the next token we need to check whether the status is already empty (which would mean that token list is longer, hence not equal), if it's not empty and the head is `N`-type we compare these two (the test here for `N`-type fails for empty arguments, hence we have to test this separately). If they are equal we store the rest of the second token list in the status and go on with the loop, else we break out and return false.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_eq_normal:nN #1#2 }
  {
    \__etl_if_empty:nT {#1} { \etl_act_break:n \if_false: }
    \__etl_if_head_is_N_type:nTF {#1}
      {
        \exp_after:wN \__etl_if_eq_normal:NnN
          \__etl_expanded:w { \__etl_split_first:w #1 }
          #2
      }
      { \etl_act_break:n \if_false: }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_eq_normal:NnN #1#2#3 }
  {
    \etl_token_if_eq:NNTF {#1} {#3}
      { \etl_act_status:n {#2} }
      { \etl_act_break:n \if_false: }
  }
```

Spaces are pretty similar, but easier, we don't need to split of the first token in a complicated manner, instead we just gobble a leading space.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_eq_space:n #1 }
  {
    \__etl_if_head_is_space:nTF {#1}
      { \exp_after:wN \etl_act_status:n \exp_after:wN { \__etl_rm_space:w #1 } }
      { \etl_act_break:n \if_false: }
  }
```

Groups are similarly handled to normal arguments, but instead of comparing only two tokens we have to compare by recursion.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_eq_group:nn #1 }
  {
    \__etl_if_head_is_group:nTF {#1}
      {
        \exp_after:wN \__etl_if_eq_group:nnn
          \__etl_expanded:w { \__etl_split_first:w #1 }
      }
```

```
      { \etl_act_break:n \if_false: }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_eq_group:nnn #1#2#3 }
  {
    \etl_if_eq:nnTF {#1} {#3}
      { \etl_act_status:n {#2} }
      { \etl_act_break:n \if_false: }
  }
```

Finally, if the loop didn't break until the first token list is empty we just have to make sure that the second list is also empty by now. If that's the case the two are equal, else not. We need to leave either true or false (protected against the `\__etl_expanded:w` expansion) in the input.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_eq_final:nn #1#2 }
  {
    \exp_after:wN
    \__etl_unexpanded:w
    \__etl_if_empty:nT {#1} { { \if_true: } \use_none:n } { \if_false: }
  }
```

(*End definition for* `\etl_if_eq:nnTF` *and others. This function is documented on page* 7.)

`\etl_if_in_p:nn`
`\etl_if_in:nnTF`
`\__etl_if_in_normal:nN`
`\__etl_if_in_normal:nnnN`
`\__etl_if_in_normal:NnnnN`
`\__etl_if_in_space:n`
`\__etl_if_in_space:nnn`
`\__etl_if_in_group:nn`
`\__etl_if_in_group:nnnn`
`\__etl_if_in_group:nnnnn`
`\__etl_if_in_put_back:n`

`\etl_if_in:nn` has to reevaluate every token but the very first in order to compare them, else something like `aab` wouldn't contain `ab` according to the test, because the second `a` would've been gobbled. For this we need `\__etl_if_in_put_back:n` which will remove the first token (we need to watch out for spaces) and puts the rest back using `\etl_act_put_back:n`.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_put_back:n #1 }
  {
    \__etl_if_head_is_space:nTF {#1}
      { \exp_after:wN \etl_act_put_back:n \exp_after:wN { \__etl_rm_space:w #1 } }
      { \exp_after:wN \etl_act_put_back:n \exp_after:wN { \use_none:n #1 } }
  }
```

As already said, we'll need to reinsert some tokens, and we'll might have to revert what was already matched, so inside of the status we store the remainder of the pattern which needs to be matched, followed by the entire pattern, followed by the tokens which were already matched (and might need to be put back). As soon as the pattern is matched the remainder will be empty and we'll leave `\if_true:` in the input, at the end of the entire list we'll leave `\if_false:`, which we store in the prefilled output. The emptiness of the pattern will be checked before the next token is evaluated, so the trailing space after `#1` does no harm but allows the token list to end in the pattern.

All of the macros used as arguments to `\__etl_act:nnnnnnn` will need to unbrace the status which will then lead to three arguments. Else this is pretty much the same idea as `\etl_if_eq:nnTF`.

```
\exp_args:Nno
\use:n { \prg_new_conditional:Npnn \etl_if_in:nn #1#2 { TF , T , F , p } }
  {
    \__etl_act:nnnnnnn
        \__etl_if_in_normal:nN \__etl_if_in_space:n \__etl_if_in_group:nn
        \__etl_act_just_result:nn
        { { #2 } { #2 } {} }
        \if_false:
        { #1 ~ }
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

Just like `\__etl_if_in_group:nn`, `\__etl_if_in_normal:nN` needs to split off the first token of the pattern, for which `\__etl_split_first:w` is used, and `\__etl_if_in_space:n` needs to trim off a leading space.

```
\cs_new:Npn \__etl_if_in_normal:nN #1 { \__etl_if_in_normal:nnnN #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_normal:nnnN #1#2#3#4 }
  {
    \__etl_if_empty:nT {#1} { \etl_act_break:n \if_true: }
    \__etl_if_head_is_N_type:nTF {#1}
      {
        \exp_after:wN \__etl_if_in_normal:NnnnN
          \__etl_expanded:w { \__etl_split_first:w #1 } {#2} {#3} #4
      }
      {
        \etl_act_status:n { {#2} {#2} {} }
        \__etl_if_in_put_back:n { #3 #4 }
      }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_normal:NnnnN #1#2#3#4#5 }
  {
    \etl_token_if_eq:NNTF {#1} {#5}
      { \etl_act_status:n { {#2} {#3} {#4#5} } }
      {
        \__etl_if_in_put_back:n { #4 #5 }
        \etl_act_status:n { {#3} {#3} {} }
      }
  }
\cs_new:Npn \__etl_if_in_space:n #1 { \__etl_if_in_space:nnn #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_space:nnn #1#2#3 }
  {
    \__etl_if_empty:nT {#1} { \etl_act_break:n \if_true: }
    \__etl_if_head_is_space:nTF {#1}
      {
        \exp_after:wN \etl_act_status:n \exp_after:wN
          { \exp_after:wN { \__etl_rm_space:w #1 } {#2} { #3 ~ } }
      }
      {
        \__etl_if_in_put_back:n { #3 ~ }
        \etl_act_status:n { {#2} {#2} {} }
      }
  }
\cs_new:Npn \__etl_if_in_group:nn #1 { \__etl_if_in_group:nnnn #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_group:nnnn #1 }
  {
    \__etl_if_empty:nT {#1} { \etl_act_break:n \if_true: }
    \__etl_if_head_is_group:nTF {#1}
      {
        \exp_after:wN \__etl_if_in_group:nnnnn
          \__etl_expanded:w { \__etl_split_first:w #1 }
```

```
      }
      { \__etl_if_in_group_false:nnn }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_group:nnnnn #1#2#3#4#5 }
  {
    \etl_if_eq:nnTF {#1} {#5}
      { \etl_act_status:n { {#2} {#3} { #4 {#5} } } }
      {
        \__etl_if_in_put_back:n { #4 {#5} }
        \etl_act_status:n { {#3} {#3} {} }
      }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_group_false:nnn #1#2#3 }
  {
    \__etl_if_in_put_back:n { #2 {#3} }
    \etl_act_status:n { {#1} {#1} {} }
  }
```

(*End definition for* `\etl_if_in:nnTF` *and others. This function is documented on page* 7.)

`\etl_if_in_deep_p:nn`
`\etl_if_in_deep:nnTF`
`\__etl_if_in_group_deep:nn`
`\__etl_if_in_group_deep:nnnn`
`\__etl_if_in_group_deep:nnnnn`
`\__etl_if_in_group_deep_false:nnn`

In essence this is the same as `\etl_if_in:nnTF`, but additionally every time a group is encountered we need to search that group by recursion as well after directly comparing it to the pattern.

```
\exp_args:Nno
\use:n { \prg_new_conditional:Npnn \etl_if_in_deep:nn #1#2 { TF , T , F , p } }
  {
    \__etl_act:nnnnnnn
        \__etl_if_in_normal:nN \__etl_if_in_space:n \__etl_if_in_group_deep:nn
        \__etl_act_just_result:nn
        { { #2 } { #2 } {} }
        \if_false:
        { #1 ~ }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__etl_if_in_group_deep:nn #1 { \__etl_if_in_group_deep:nnnn #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_group_deep:nnnn #1 }
  {
    \__etl_if_empty:nT {#1} { \etl_act_break:n \if_true: }
    \__etl_if_head_is_group:nTF {#1}
      {
        \exp_after:wN \__etl_if_in_group_deep:nnnnn
          \__etl_expanded:w { \__etl_split_first:w #1 }
      }
      { \__etl_if_in_group_deep_false:nnn }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_group_deep:nnnnn #1#2#3#4#5 }
  {
    \etl_if_eq:nnTF {#1} {#5}
      { \etl_act_status:n { {#2} {#3} { #4 {#5} } } }
      {
        \etl_if_in_deep:nnT {#5} {#3} { \etl_act_break:n \if_true: }
```

```
        \__etl_if_in_put_back:n { #4 {#5} }
        \etl_act_status:n { {#3} {#3} {} }
      }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_if_in_group_deep_false:nnn #1#2#3 }
  {
    \etl_if_in_deep:nnT {#3} {#1} { \etl_act_break:n \if_true: }
    \__etl_if_in_put_back:n { #2 {#3} }
    \etl_act_status:n { {#1} {#1} {} }
  }
```

(*End definition for* `\etl_if_in_deep:nnTF` *and others. This function is documented on page* 7.)

2.6 Expandably modify token lists

`\etl_token_replace_all:nNn`
`\__etl_token_replace:NnnN`

Replacing a single token (and in fact the same is true for all the replacement actions in this package) doesn't need reordering and no post processing, so we can use in place output using `\__etl_unexpanded:w`. We store the token we want to replace inside the act function, as well as an additional argument which will be executed once a replacement was done (this is used for the `\etl_token_replace_once:nNn` function).

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_token_replace_all:nNn #1#2#3 }
  {
    \etl_act:nnnnnn
      { \__etl_token_replace:NnnN #2 {} }
      \__etl_act_unexpanded_space:n
      \__etl_act_unexpanded_group:nn
      \use_none:nn
      {#3}
      {#1}
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_token_replace:NnnN #1#2#3#4 }
  {
    \etl_token_if_eq:NNTF {#1} {#4}
      { \__etl_unexpanded:w {#3} #2 }
      { \__etl_unexpanded:w {#4} }
  }
```

(*End definition for* `\etl_token_replace_all:nNn` *and* `\__etl_token_replace:NnnN`. *This function is documented on page* 8.)

`\etl_token_replace_all_deep:nNn`
`\__etl_token_replace_deep:Nnn`

Deep replacement is done by recursion. Since the deep variant will not execute any additional code we omit such an additional argument for it.

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_token_replace_all_deep:nNn #1#2#3 }
  {
    \etl_act:nnnnnn
      { \__etl_token_replace:NnnN #2 {} }
      \__etl_act_unexpanded_space:n
      { \__etl_token_replace_deep:Nnn #2 }
      \use_none:nn
      {#3}
      {#1}
  }
```

Here {#1} is used to get correct results from the first step of expansion done directly.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_token_replace_deep:Nnn #1#2#3 }
  { \exp_after:wN { \etl_token_replace_all_deep:nNn {#3} {#1} {#2} } }
```

(*End definition for* `\etl_token_replace_all_deep:nNn` *and* `\__etl_token_replace_deep:Nnn`*. This function is documented on page 8.*)

`\etl_token_replace_once:nNn`

To only handle the first matching token we just let the replacement internal exchange the function to directly output any token.

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_token_replace_once:nNn #1#2#3 }
  {
    \etl_act:nnnnnn
      { \__etl_token_replace:NnnN #2 \__etl_act_unexpanded_rest:w }
      \__etl_act_unexpanded_space:n
      \__etl_act_unexpanded_group:nn
      \use_none:nn
      {#3}
      {#1}
  }
```

(*End definition for* `\etl_token_replace_once:nNn`*. This function is documented on page 7.*)

`\etl_replace_all:nnn`
`\__etl_replace_put_back:nnnN`
`\__etl_replace_put_back_normal:Nn`
`\__etl_replace_put_back_group:nn`
`\__etl_replace_normal:nN`
`\__etl_replace_normal:nnnnNN`
`\__etl_replace_normal:NnnnnNN`
`\__etl_replace_normal_false:nnnNN`
`\__etl_replace_space:n`
`\__etl_replace_space:nnnnN`
`\__etl_replace_space_aux:nnnnN`
`\__etl_replace_space_false:nnnN`
`\__etl_replace_group:nn`
`\__etl_replace_group:nnnnNn`
`\__etl_replace_group:nnnnnNn`
`\__etl_replace_group_false:nnnNn`
`\__etl_replace_final:nn`
`\__etl_replace_final:nnnnNn`

Replacing an arbitrary number of tokens (which might include braces and spaces) is quite a bit harder than a single N-type. We place in the status the remainder of the pattern, the full pattern, delayed tokens (those which matched the pattern), the replacement, and a marker which should tell us whether we want to only replace the first match (if so use `\__etl_act_unexpanded_rest:w`, else `\prg_do_nothing:`).

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_replace_all:nnn #1#2#3 }
  {
    \etl_act:nnnnnn
      \__etl_replace_normal:nN
      \__etl_replace_space:n
      \__etl_replace_group:nn
      \__etl_replace_final:nn
      { {#2} {#2} {} {#3} \prg_do_nothing: }
      {#1}
  }
```

We again need to be able to put back a few tokens, but this time we also need to know whether the first token is an N-type or group, because we can't just gobble the first element but need to output it unchanged.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_put_back:nnnN #1#2#3#4 }
  {
    \__etl_if_head_is_space:nTF {#1}
      {
        \exp_after:wN \etl_act_put_back:n \exp_after:wN { \__etl_rm_space:w #1 } ~
      }
      {
        \__etl_if_head_is_group:nTF {#1}
          { \exp_after:wN \__etl_replace_put_back_group:nn }
          { \exp_after:wN \__etl_replace_put_back_normal:Nn }
          \__etl_expanded:w { \__etl_split_first:w #1 }
      }
    \etl_act_status:n { {#2} {#2} {} {#3} #4 }
```

```
  }
\cs_new:Npn \__etl_replace_put_back_group:nn #1
  {
    \__etl_unexpanded:w { {#1} }
    \etl_act_put_back:n
  }
\cs_new:Npn \__etl_replace_put_back_normal:Nn #1
  {
    \__etl_unexpanded:w {#1}
    \etl_act_put_back:n
  }
\cs_new:Npn \__etl_replace_normal:nN #1 { \__etl_replace_normal:nnnnNN #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_normal:nnnnNN #1 }
  {
    \__etl_if_head_is_N_type:nTF {#1}
      {
        \exp_after:wN \__etl_replace_normal:NnnnnNN
          \__etl_expanded:w { \__etl_split_first:w #1 }
      }
      { \__etl_replace_normal_false:nnnNN }
  }
```

Just to keep track of the different arguments here: #1 is the next token in the pattern, #2 is the remainder of the pattern, #3 is the full pattern stored for reuse, #4 are the delayed tokens, which might need to be put back, #5 is the replacement text, #6 is the marker which might indicate the once function, and #7 is the next token of the input.

```
\exp_args:Nno
\use:n { \cs_new:Npn \__etl_replace_normal:NnnnnNN #1#2#3#4#5#6#7 }
  {
    \etl_token_if_eq:NNTF {#1} {#7}
      {
        \__etl_if_empty:nTF {#2}
          {
            \__etl_unexpanded:w {#5}
            #6
            \etl_act_status:n { {#3} {#3} {} {#5} #6 }
          }
          { \etl_act_status:n { {#2} {#3} { #4 #7 } {#5} #6 } }
      }
      { \__etl_replace_put_back:nnnN { #4 #7 } {#3} {#5} #6 }
  }
\exp_args:Nno
\use:n { \cs_new:Npn \__etl_replace_normal_false:nnnNN #1#2#3#4#5 }
  { \__etl_replace_put_back:nnnN { #2 #5 } {#1} {#3} {#4} }
\cs_new:Npn \__etl_replace_space:n #1 { \__etl_replace_space:nnnnN #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_space:nnnnN #1 }
  {
    \__etl_if_head_is_space:nTF {#1}
      {
        \exp_after:wN \__etl_replace_space_aux:nnnnN \exp_after:wN
          { \__etl_rm_space:w #1 }
      }
      { \__etl_replace_space_false:nnnN }
  }
```

Again, to keep track, #1 is the remainder of the pattern, #2 is the full pattern, #3 the delayed tokens, #4 the replacement text, #5 the marker for the once function.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_space_aux:nnnnN #1#2#3#4#5 }
  {
    \__etl_if_empty:nTF {#1}
      {
        \__etl_unexpanded:w {#4}
        #5
        \etl_act_status:n { {#2} {#2} {} {#4} #5 }
      }
      { \etl_act_status:n { {#1} {#2} { #3 ~ } {#4} #5 } }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_space_false:nnnN #1#2#3#4 }
  { \__etl_replace_put_back:nnnN { #2 ~ } {#1} {#3} {#4} }
\cs_new:Npn \__etl_replace_group:nn #1 { \__etl_replace_group:nnnnNn #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_group:nnnnNn #1 }
  {
    \__etl_if_head_is_group:nTF {#1}
      {
        \exp_after:wN \__etl_replace_group:nnnnnNn
          \__etl_expanded:w { \__etl_split_first:w #1 }
      }
      { \__etl_replace_group_false:nnnNn }
  }
```

And again, #1 the next group of the pattern, #2 the remainder of the pattern, #3 the full pattern, #4 the delayed stuff, #5 the replacement text, #6 the marker for the once function, #7 the next group in the input.

```
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_group:nnnnnNn #1#2#3#4#5#6#7 }
  {
    \etl_if_eq:nnTF {#1} {#7}
      {
        \__etl_if_empty:nTF {#2}
          {
            \__etl_unexpanded:w {#5}
            #6
            \etl_act_status:n { {#3} {#3} {} {#5} #6 }
          }
          { \etl_act_status:n { {#2} {#3} { #4 {#7} } {#5} #6 } }
      }
      { \__etl_replace_put_back:nnnN { #4 {#7} } {#3} {#5} #6 }
  }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_group_false:nnnNn #1#2#3#4#5 }
  { \__etl_replace_put_back:nnnN { #2 {#5} } {#1} {#3} {#4} }
\cs_new:Npn \__etl_replace_final:nn #1 { \__etl_replace_final:nnnnNn #1 }
\cs_new:Npn \__etl_replace_final:nnnnNn #1#2#3#4#5#6 { \__etl_unexpanded:w { #6#3 } }
```

(*End definition for* `\etl_replace_all:nnn` *and others. This function is documented on page* 8.)

`\etl_replace_all_deep:nnn`
`\__etl_replace_group_deep:nn`
`\__etl_replace_group_deep:nnnnNn`
`\__etl_replace_group_deep:nnnnnNn`
`\__etl_replace_group_deep_false:nnnNn`

The deep variant works again pretty much the same as the `all` variant, except that it searches groups recursively.

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_replace_all_deep:nnn #1#2#3 }
  {
    \etl_act:nnnnnn
```

```
      \__etl_replace_normal:nN
      \__etl_replace_space:n
      \__etl_replace_group_deep:nn
      \__etl_replace_final:nn
      { {#2} {#2} {} {#3} \prg_do_nothing: }
      {#1}
  }
\cs_new:Npn \__etl_replace_group_deep:nn #1
  { \__etl_replace_group_deep:nnnnNn #1 }
\exp_args:Nno \use:n { \cs_new:Npn \__etl_replace_group_deep:nnnnNn #1 }
  {
    \__etl_if_head_is_group:nTF {#1}
      {
        \exp_after:wN \__etl_replace_group_deep:nnnnnNn
          \__etl_expanded:w { \__etl_split_first:w #1 }
      }
      { \__etl_replace_group_deep_false:nnnNn }
  }
\exp_args:Nno
\use:n { \cs_new:Npn \__etl_replace_group_deep:nnnnnNn #1#2#3#4#5#6#7 }
  {
    \etl_if_eq:nnTF {#1} {#7}
      {
        \__etl_if_empty:nTF {#2}
          {
            \__etl_unexpanded:w {#5}
            \etl_act_status:n { {#3} {#3} {} {#5} #6 }
          }
          { \etl_act_status:n { {#2} {#3} { #4 {#7} } {#5} #6 } }
      }
      {
        \__etl_if_empty:nTF {#4}
          {
            { \etl_replace_all_deep:nnn {#7} {#3} {#5} }
            \etl_act_status:n { {#3} {#3} {} {#5} #6 }
          }
          { \__etl_replace_put_back:nnnN { #4 {#7} } {#3} {#5} #6 }
      }
  }
\exp_args:Nno
\use:n { \cs_new:Npn \__etl_replace_group_deep_false:nnnNn #1#2#3#4#5 }
  {
    \__etl_if_empty:nTF {#2}
      {
        { \etl_replace_all_deep:nnn {#5} {#1} {#3} }
        \etl_act_status:n { {#1} {#1} {} {#3} #4 }
      }
      { \__etl_replace_put_back:nnnN { #2 {#5} } {#1} {#3} #4 }
  }
```

(*End definition for* `\etl_replace_all_deep:nnn` *and others. This function is documented on page* 8.)

`\etl_replace_once:nnn` And this is the same as the `all` variant except that we now use the `\__etl_act_unexpanded_rest:w` marker inside the status.

```
\exp_args:Nno \use:n { \cs_new:Npn \etl_replace_once:nnn #1#2#3 }
  {
    \etl_act:nnnnnn
      \__etl_replace_normal:nN
      \__etl_replace_space:n
      \__etl_replace_group:nn
      \__etl_replace_final:nn
      { {#2} {#2} {} {#3} \__etl_act_unexpanded_rest:w }
      {#1}
  }
```

(*End definition for* `\etl_replace_once:nnn`*. This function is documented on page 8.*)

2.7 Defining new tests

`\etl_new_if_in:Nnn`
`\__etl_new_if_in:NnNnn`
`\__etl_new_if_in:NNnn`

These tests work essentially in the same way as `\tl_if_in:nnTF`, but instead they use a predefined internal macro so that no definition at use time is necessary. We use a small loop to get a unique auxiliary macro name for the search text.

```
\exp_args:Nno \use:n { \cs_new_protected:Npn \etl_new_if_in:Nnn #1#2#3 }
  {
    \scan_stop:
    \if_false: { \fi:
    \exp_args:Nc \__etl_new_if_in:NnNnn
      { __etl_user_function ~ if_in ~ \tl_to_str:n {#2} :w }
      ?
      #1 {#2}
      {#3}
    \if_false: } \fi:
  }
\cs_new_protected:Npn \__etl_new_if_in:NnNnn #1#2#3#4
  {
    \cs_if_exist:NTF #1
      {
        \cs_set:Npn \__etl_tmp:w ##1 #4 {}
        \cs_if_eq:NNTF #1 \__etl_tmp:w
          { \__etl_new_if_in:NNnn #1 #3 {#4} }
          {
            \exp_args:Nc \__etl_new_if_in:NnNnn
              { __etl_user_function ~ if_in #2 ~ \tl_to_str:n {#4} :w }
              { #2? }
              #3 {#4}
          }
      }
      { \__etl_new_if_in:NNnn #1 #3 {#4} }
  }
\cs_new_protected:Npn \__etl_new_if_in:NNnn #1#2#3#4
  {
    \cs_gset:Npn #1 ##1 #3 {}
    \prg_new_conditional:Npnn #2 ##1 {#4}
      {
        \if:w
            \scan_stop:
            \__etl_detokenize:w \exp_after:wN { #1 ##1 {}{} #3 }
            \scan_stop:
```

```
          \__etl_fi_turn_false:w
        \fi:
        \if_true:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
  }
```

(*End definition for* `\etl_new_if_in:Nnn`, `\__etl_new_if_in:NnNnn`, *and* `\__etl_new_if_in:NNnn`. *This function is documented on page* 9.)

2.8 Defining new modifiers

The implementation of `replace_once` and `replace_all` is modelled closely on the implementation used in l3tl. The difference is that we use a hard coded delimiter (`\s__etl_stop`) instead of searching for one that is always legal (we can't do redefinitions, so can't change the delimiter later based on the token list input).

`\__etl_new_replace_def:NNn`
`\__etl_new_replace_def_aux:NnNnN`
`\__etl_new_replace_def_aux:Nn`

We need another loop to guarantee unique names, if everything's alright we go on and define the user function using `#1` of `\__etl_new_replace_def:NNn`. An empty search pattern is forbidden and should throw an error.

```
\msg_new:nnn { etl } { empty-search-text }
  { The~ search~ text~ of~ #1 must~ not~ be~ empty. }
\cs_new_protected:Npn \__etl_new_replace_def:NNn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      { \msg_error:nnn { etl } { empty-search-text } { #2 } }
      {
        \scan_stop:
        \if_false: { \fi:
        \exp_args:Nc \__etl_new_replace_def_aux:NnNnN
          { __etl_user_function ~ replace ~ \tl_to_str:n {#3} ~ :Nnw }
          ?
          #2 {#3}
          #1
        \if_false: } \fi:
      }
  }
\cs_new_protected:Npn \__etl_new_replace_def_aux:NnNnN #1#2#3#4#5
  {
    \cs_if_exist:NTF #1
      {
        \__etl_new_replace_def_aux:Nn \__etl_tmp:w {#4}
        \cs_if_eq:NNTF #1 \__etl_tmp:w
          { #5 #1#3 {#4} }
          {
            \exp_args:Nc \__etl_new_replace_def_aux:NnNnN
              { __etl_user_function ~ replace #2 ~ \tl_to_str:n {#4} ~ :Nnw }
              { #2? } #3 {#4} #5
          }
      }
      {
```

```
        \__etl_new_replace_def_aux:Nn #1 {#4}
        #5 #1#3 {#4}
      }
  }
```

The auxiliary macro uses a loop for the replacement. This is also used for the once variant. This saves internal functions if both an `all` and a once function are generated for the same search text (though the once variant could be coded easier and faster otherwise, but the performance hit should be small).

```
\cs_new_protected:Npn \__etl_new_replace_def_aux:Nn #1#2
  {
    \cs_gset:Npn #1 ##1##2 ##3#2
      {
        \__etl_new_replace_wrap:w ##3 \s__etl_stop \__etl_unexpanded:w {##2}
        ##1 #1 {##2} {}{}
      }
  }
```

(*End definition for* `\__etl_new_replace_def:NNn`, `\__etl_new_replace_def_aux:NnNnN`, *and* `\__etl_new_replace_def_aux:Nn`.)

`\__etl_new_replace_wrap:w`
`\__etl_new_replace_once:w`
`\__etl_new_replace_done:w`

We need a few auxiliaries for the two replacement variants here. The first just grabs the already processed part of the token list and protects it from further expanding. The second breaks the loop for the once variant by protecting the remainder of the token list from further expanding. The last just gobbles the remainder of the loop by using an unbalanced brace trick.

```
\cs_new:Npn \__etl_new_replace_wrap:w #1\s__etl_stop
  { \__etl_unexpanded:w \exp_after:wN { \use_none:nn #1 } }
\cs_new:Npn \__etl_new_replace_once:w #1#2 #3\s__etl_stop
  { \__etl_unexpanded:w \exp_after:wN { \use_none:nn #3 } }
\cs_new:Npn \__etl_new_replace_done:w
  { \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi: }
```

(*End definition for* `\__etl_new_replace_wrap:w`, `\__etl_new_replace_once:w`, *and* `\__etl_new_replace_done:w`.)

`\etl_new_replace_once:Nn`
`\__etl_new_replace_once:NNn`

The once variant will use `\__etl_new_replace_done_once:w` if the replacement is successful (that will remove the remainder of the loop, and protect both the replacement and the rest of the token list on which we work from further expanding).

```
\cs_new_protected:Npn \etl_new_replace_once:Nn
  { \__etl_new_replace_def:NNn \__etl_new_replace_once:NNn }
\cs_new_protected:Npn \__etl_new_replace_once:NNn #1#2#3
  {
    \cs_new:Npn #2 ##1##2
      {
        \__etl_unexpanded:w \__etl_expanded:w
          {{
            \if_false: { \fi:
              #1 \__etl_new_replace_once:w {##2} {}{} ##1 \s__etl_stop
              \__etl_new_replace_done:w #3
            }
          }}
      }
  }
```

(*End definition for* `\etl_new_replace_once:Nn` *and* `\__etl_new_replace_once:NNn`. *This function is documented on page 9.*)

`\etl_new_replace_all:Nn`
`\__etl_new_replace_all:NNn`

The `all` variant will directly protect the replacement from further expanding and reiterate (due to the way the auxiliary is defined) until the replacement isn't found anymore.

```
\cs_new_protected:Npn \etl_new_replace_all:Nn
  { \__etl_new_replace_def:NNn \__etl_new_replace_all:NNn }
\cs_new_protected:Npn \__etl_new_replace_all:NNn #1#2#3
  {
    \cs_new:Npn #2 ##1##2
      {
        \__etl_unexpanded:w \__etl_expanded:w
          {{
            \if_false: { \fi:
              #1 #1 {##2} {}{} ##1 \s__etl_stop
              \__etl_new_replace_done:w #3
            }
          }}
      }
  }
```

(*End definition for* `\etl_new_replace_all:Nn` *and* `\__etl_new_replace_all:NNn`. *This function is documented on page 10.*)

2.9 Undefine now unnecessary functions

```
\cs_undefine:N \__etl_act:nnnnnnn
```

⟨/pkg⟩

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.