The l3regex module
Regular expressions in TeX

The LaTeX Project*

Released 2024-01-04

The l3regex module provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that TeX manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text "`This cat.`", where the first occurrence of "`at`" was replaced by "`is`". A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any "word" character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_set:Nn`. For example,

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\l_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses "capture" the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

*E-mail: latex-team@latex-project.org

1 Syntax of regular expressions

1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word "Cat" capitalized in this way, but also matches the beginning of the word "Cattle": use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among "a", "b", "c"; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).
- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `\_[^\_]*\_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `\_.*?\_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+\-]?\d+` matches an explicit integer with at most one sign.
- `[\+\-\␣]*\d+\␣*` matches an explicit integer with any number of + and − signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+\-\␣]*(\d+|\d*\.\d+)\␣*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+\-\␣]*(\d+|\d*\.\d+)\␣*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)\␣*` matches an explicit dimension with any unit that TeX knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+\-\␣]*((?i)nan|inf|(\d+|\d*\.\d+)(\␣*e[\+\-\␣]*\d+)?)\␣*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+\-\␣]*(\d+|\cC.)\␣*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+\-\(]*\d+\)*([\+\-*/][\+\-\(]*\d+\)*)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

1.2 Characters in regular expressions

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `\*` matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A`–`Z`, `a`–`z`, `0`–`9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`, `\^`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into TEX under normal category codes. For instance, `\\abc\%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{`⟨*regex*⟩`}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

1.3 Characters classes

Character properties.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^^I\^^J\^^L\^^M]`.

`\v` Any vertical space character, equivalent to `[\^^J\^^K\^^L\^^M]`. Note that `\^^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9\_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

`[^...]` Negative character class. Matches any token other than the specified characters.

`x-y` Within a character class, this denotes a range (can be used with escaped characters).

`[:⟨name⟩:]` Within a character class (one more set of brackets), this denotes the POSIX character class ⟨*name*⟩, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

`[:^⟨name⟩:]` Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except `p`, as well as control sequences (see below for a description of `\c`).

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[^6-9]` are equivalent.

1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

`?` 0 or 1, greedy.

`??` 0 or 1, lazy.

`*` 0 or more, greedy.

`*?` 0 or more, lazy.

`+` 1 or more, greedy.

`+?` 1 or more, lazy.

`{`n`}` Exactly n.

`{`n`,}` n or more, greedy.

`{`n`,}?` n or more, lazy.

`{`n, m`}` At least n, no more than m, greedy.

`{`n, m`}?` At least n, no more than m, lazy.

For greedy quantifiers the regex code will first investigate matches that involve as many repetitions as possible, while for lazy quantifiers it investigates matches with as few repetitions as possible first.

Alternation and capturing groups.

`A|B|C` Either one of `A`, `B`, or `C`, investigating `A` first.

`(...)` Capturing group.

`(?:...)` Non-capturing group.

`(?|...)` Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the "best" match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnNTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

1.5 Matching exact tokens

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- `C` for control sequences;
- `B` for begin-group tokens;
- `E` for end-group tokens;

- `M` for math shift;
- `T` for alignment tab tokens;
- `P` for macro parameter tokens;
- `U` for superscript tokens (up);
- `D` for subscript tokens (down);
- `S` for spaces;
- `L` for letters;
- `O` for others; and
- `A` for active characters.

The `\c` escape sequence is used as follows.

`\c{`⟨*regex*⟩`}` A control sequence whose csname matches the ⟨*regex*⟩, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, escape character sequence such as `\x{0A}`, character class, or group, and forces this object to only match tokens with category `X` (any of `CBEMTPUDSLOA`. For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.[1]

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[LO][A-F]]` matches what TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from `A` to `F` with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO\*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{`⟨*var name*⟩`}` matches the exact contents (both character codes and category codes) of the variable `\`⟨*var name*⟩, which are obtained by applying `\exp_not:v {`⟨*var name*⟩`}` at the time the regular expression is compiled. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are supported.

The `\ur` escape sequence allows to insert the contents of a `regex` variable into a larger regular expression. For instance, `A\ur{l_tmpa_regex}D` matches the tokens `A` and

[1]This last example also captures "`abc`" as a regex group; to avoid this use a non-capturing group `\cO(?:abc)`.

D separated by something that matches the regular expression \l_tmpa_regex. This behaves as if a non-capturing group were surrounding \l_tmpa_regex, and any group contained in \l_tmpa_regex is converted to a non-capturing group. Quantifiers are supported.

For instance, if \l_tmpa_regex has value B|C, then A\ur{l_tmpa_regex}D is equivalent to A(?:B|C)D (matching ABD or ACD) and not to AB|CD (matching AB or CD). To get the latter effect, it is simplest to use TeX's expansion machinery directly: if \l_mymodule_BC_tl contains B|C then the following two lines show the same result:

```
\regex_show:n { A \u{l_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
```

1.6 Miscellaneous

Anchors and simple assertions.

\b Word boundary: either the previous token is matched by \w and the next by \W, or the opposite. For this purpose, the ends of the token list are considered as \W.

\B Not a word boundary: between two \w tokens or two \W tokens (including the boundary).

^or \A Start of the subject token list.

$, \Z or \z End of the subject token list.

\G Start of the current match. This is only different from ^ in the case of multiple matches: for instance \regex_count:nnN { \G a } { aaba } \l_tmpa_int yields 2, but replacing \G by ^ would result in \l_tmpa_int holding the value 1.

The option (?i) makes the match case insensitive (treating A–Z and a–z as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using (?-i). For instance, in (?i)(a(?-i)b|c)d, the letters a and d are affected by the i option. Characters within ranges and classes are affected individually: (?i)[\?-B] is equivalent to [\?@ABab] (and differs from the much larger class [\?-b]), and (?i)[^aeiou] matches any character which is not a vowel. The i option has no effect on \c{...}, on \u{...}, on character properties, or on character classes, for instance it has no effect at all in (?i)\u{l_foo_tl}\d\d[[:lower:]].

2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- \0 is the whole match;
- \1 is the submatch that was matched by the first (capturing) group (...); similarly for \2, ..., \9 and \g{⟨*number*⟩};
- \␣ inserts a space (spaces are ignored when not escaped);

- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\xhh`, `\x{hhh}` correspond to single characters as in regular expressions;
- `\c{`⟨*cs name*⟩`}` inserts a control sequence;
- `\c`⟨*category*⟩⟨*character*⟩ (see below);
- `\u{`⟨*tl var name*⟩`}` inserts the contents of the ⟨*tl var*⟩ (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for TeX, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(ell--el)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n-th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

By default, the category code of characters inserted by the replacement are determined by the prevailing category code regime at the time where the replacement is made, with two exceptions:

- space characters (with character code 32) inserted with `\␣` or `\x20` or `\x{20}` have category code 10 regardless of the prevailing category code regime;
- if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters "..." with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{`⟨`text`⟩`}` Produces the control sequence with csname ⟨*text*⟩. The ⟨*text*⟩ may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{`⟨*var name*⟩`}` allows to insert the contents of the variable with name ⟨*var name*⟩ directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{l_my_\0_tl} } \l_my_tl
```

results in \l_my_tl holding first,\emph{second},first,first.

Regex replacement is also a convenient way to produce token lists with arbitrary category codes. For instance

```
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

results in \l_tmpa_tl containing the percent character with category code 7 (superscript) and an active tilde character.

3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the l3regex module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

\regex_new:N

New: 2017-05-26

\regex_new:N ⟨*regex var*⟩

Creates a new ⟨*regex var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*regex var*⟩ is initially such that it never matches.

\regex_set:Nn
\regex_gset:Nn

New: 2017-05-26

\regex_set:Nn ⟨*regex var*⟩ {⟨*regex*⟩}

Stores a compiled version of the ⟨*regular expression*⟩ in the ⟨*regex var*⟩. The assignment is local for \regex_set:Nn and global for \regex_gset:Nn. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

\regex_const:Nn

New: 2017-05-26

\regex_const:Nn ⟨*regex var*⟩ {⟨*regex*⟩}

Creates a new constant ⟨*regex var*⟩ or raises an error if the name is already taken. The value of the ⟨*regex var*⟩ is set globally to the compiled version of the ⟨*regular expression*⟩.

\regex_show:N
\regex_show:n
\regex_log:N
\regex_log:n

New: 2021-04-26
Updated: 2021-04-29

\regex_show:n {⟨*regex*⟩}
\regex_log:n {⟨*regex*⟩}

Displays in the terminal or writes in the log file (respectively) how l3regex interprets the ⟨*regex*⟩. For instance, \regex_show:n {\A X|Y} shows

```
+-branch
  anchor at start (\A)
  char code 88 (X)
+-branch
  char code 89 (Y)
```

indicating that the anchor \A only applies to the first branch: the second branch is not anchored to the beginning of the match.

4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a "standard" regular expression, while the later require a compiled expression as generated by `\regex_set:Nn`.

`\regex_match:nnTF`
`\regex_match:nVTF`
`\regex_match:NnTF`
`\regex_match:NVTF`

New: 2017-05-26

`\regex_match:nnTF {⟨regex⟩} {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*regular expression*⟩ matches any part of the ⟨*token list*⟩. For instance,

```
\regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves `TRUE` then `FALSE` in the input stream.

`\regex_count:nnN`
`\regex_count:nVN`
`\regex_count:NnN`
`\regex_count:NVN`

New: 2017-05-26

`\regex_count:nnN {⟨regex⟩} {⟨token list⟩} ⟨int var⟩`

Sets ⟨*int var*⟩ within the current TEX group level equal to the number of times ⟨*regular expression*⟩ appears in ⟨*token list*⟩. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcbb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

`\regex_match_case:nn`
`\regex_match_case:nnTF`

New: 2022-01-10

```
\regex_match_case:nnTF
  {
    {⟨regex_1⟩} {⟨code case_1⟩}
    {⟨regex_2⟩} {⟨code case_2⟩}
    ...
    {⟨regex_n⟩} {⟨code case_n⟩}
  } {⟨token list⟩}
  {⟨true code⟩} {⟨false code⟩}
```

Determines which of the ⟨*regular expressions*⟩ matches at the earliest point in the ⟨*token list*⟩, and leaves the corresponding ⟨*code*$_i$⟩ followed by the ⟨*true code*⟩ in the input stream. If several ⟨*regex*⟩ match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the ⟨*regex*⟩ match, the ⟨*false code*⟩ is left in the input stream. Each ⟨*regex*⟩ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then the corresponding ⟨*code*⟩ is used and everything else is discarded, while if none of the ⟨*regex*⟩ match at a given position then the next starting position is attempted. If none of the ⟨*regex*⟩ match anywhere in the ⟨*token list*⟩ then nothing is left in the input stream. Note that this differs from nested `\regex_match:nnTF` statements since all ⟨*regex*⟩ are attempted at each position rather than attempting to match ⟨*regex*$_1$⟩ at every position before moving on to ⟨*regex*$_2$⟩.

5 Submatch extraction

`\regex_extract_once:nnN`
`\regex_extract_once:nVN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:nVNTF`
`\regex_extract_once:NnN`
`\regex_extract_once:NVN`
`\regex_extract_once:NnNTF`
`\regex_extract_once:NVNTF`

New: 2017-05-26

`\regex_extract_once:nnN {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩`
`\regex_extract_once:nnNTF {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩ {⟨true code⟩} {⟨false code⟩}`

Finds the first match of the ⟨*regular expression*⟩ in the ⟨*token list*⟩. If it exists, the match is stored as the first item of the ⟨*seq var*⟩, and further items are the contents of capturing groups, in the order of their opening parenthesis. The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is cleared. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
  { true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n-th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n-1)$ in functions such as `\regex_replace_once:nnN`.

`\regex_extract_all:nnN`
`\regex_extract_all:nVN`
`\regex_extract_all:nnNTF`
`\regex_extract_all:nVNTF`
`\regex_extract_all:NnN`
`\regex_extract_all:NVN`
`\regex_extract_all:NnNTF`
`\regex_extract_all:NVNTF`

New: 2017-05-26

`\regex_extract_all:nnN {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩`
`\regex_extract_all:nnNTF {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩ {⟨true code⟩} {⟨false code⟩}`

Finds all matches of the ⟨*regular expression*⟩ in the ⟨*token list*⟩, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is cleared. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
  { true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

\regex_split:nnN
\regex_split:nVN
\regex_split:nnNTF
\regex_split:nVNTF
\regex_split:NnN
\regex_split:NVN
\regex_split:NnNTF
\regex_split:NVNTF

New: 2017-05-26

\regex_split:nnN {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_split:nnNTF {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Splits the ⟨*token list*⟩ into a sequence of parts, delimited by matches of the ⟨*regular expression*⟩. If the ⟨*regular expression*⟩ has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to ⟨*seq var*⟩ is local. If no match is found the resulting ⟨*seq var*⟩ has the ⟨*token list*⟩ as its sole item. If the ⟨*regular expression*⟩ matches the empty token list, then the ⟨*token list*⟩ is split into single tokens. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
  { true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

6 Replacement

\regex_replace_once:nnN
\regex_replace_once:nVN
\regex_replace_once:nnNTF
\regex_replace_once:nVNTF
\regex_replace_once:NnN
\regex_replace_once:NVN
\regex_replace_once:NnNTF
\regex_replace_once:NVNTF

New: 2017-05-26

\regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_once:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Searches for the ⟨*regular expression*⟩ in the contents of the ⟨*tl var*⟩ and replaces the first match with the ⟨*replacement*⟩. In the ⟨*replacement*⟩, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* The result is assigned locally to ⟨*tl var*⟩.

\regex_replace_all:nnN
\regex_replace_all:nVN
\regex_replace_all:nnNTF
\regex_replace_all:nVNTF
\regex_replace_all:NnN
\regex_replace_all:NVN
\regex_replace_all:NnNTF
\regex_replace_all:NVNTF

New: 2017-05-26

\regex_replace_all:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_all:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Replaces all occurrences of the ⟨*regular expression*⟩ in the contents of the ⟨*tl var*⟩ by the ⟨*replacement*⟩, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to ⟨*tl var*⟩.

\regex_replace_case_once:nN
\regex_replace_case_once:nNTF

New: 2022-01-10

```
\regex_replace_case_once:nNTF
  {
    {⟨regex₁⟩} {⟨replacement₁⟩}
    {⟨regex₂⟩} {⟨replacement₂⟩}
    ...
    {⟨regexₙ⟩} {⟨replacementₙ⟩}
  } ⟨tl var⟩
  {⟨true code⟩} {⟨false code⟩}
```

Replaces the earliest match of the regular expression `(?|`⟨*regex*$_1$⟩`|`...`|`⟨*regex*$_n$⟩`)` in the ⟨*token list variable*⟩ by the ⟨*replacement*⟩ corresponding to which ⟨*regex*$_i$⟩ matched, then leaves the ⟨*true code*⟩ in the input stream. If none of the ⟨*regex*⟩ match, then the ⟨*tl var*⟩ is not modified, and the ⟨*false code*⟩ is left in the input stream. Each ⟨*regex*⟩ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then it is replaced by the corresponding ⟨*replacement*⟩ as described for `\regex_replace_once:nnN`. This is equivalent to checking with `\regex_match_case:nn` which ⟨*regex*⟩ matches, then performing the replacement with `\regex_replace_once:nnN`.

\regex_replace_case_all:nN
\regex_replace_case_all:nNTF

New: 2022-01-10

```
\regex_replace_case_all:nNTF
  {
    {⟨regex₁⟩} {⟨replacement₁⟩}
    {⟨regex₂⟩} {⟨replacement₂⟩}
    ...
    {⟨regexₙ⟩} {⟨replacementₙ⟩}
  } ⟨tl var⟩
  {⟨true code⟩} {⟨false code⟩}
```

Replaces all occurrences of all ⟨*regex*⟩ in the ⟨*token list*⟩ by the corresponding ⟨*replacement*⟩. Every match is treated independently, and matches cannot overlap. The result is assigned locally to ⟨*tl var*⟩, and the ⟨*true code*⟩ or ⟨*false code*⟩ is left in the input stream depending on whether any replacement was made or not.

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then it is replaced by the corresponding ⟨*replacement*⟩, and the search resumes at the position that follows this match (and replacement). For instance

```
\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
  {
    { [A-Za-z]+ } { ``\0'' }
    { \b } { --- }
    { . } { [\0] }
  } \l_tmpa_tl
```

results in `\l_tmpa_tl` having the contents ``` ``Hello''---[,][␣]``world''---[!] ```. Note in particular that the word-boundary assertion `\b` did not match at the start of words because the case `[A-Za-z]+` matched at these positions. To change this, one could simply swap the order of the two cases in the argument of `\regex_replace_case_all:nN`.

7 Scratch regular expressions

`\l_tmpa_regex`
`\l_tmpb_regex`
New: 2017-12-11

Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_regex`
`\g_tmpb_regex`
New: 2017-12-11

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?

  Additional error-checking to come.

- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `\__regex_item_reverse:n`.
- The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

  Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `\c{...}` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `\__regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `curr_state` and `curr_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.

- Use an array rather than \g__regex_balance_tl to build the function __regex_-replacement_balance_one_match:n.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (abcade should give two states, for abc and ade). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single __regex_action_free:n.
- Optimize the use of __regex_action_success: by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of \int_step_... functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better "show" for anchors, properties, and catcode tests.
- Does \K really need a new state for itself?
- When compiling, use a boolean in_cs and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with \cs_if_exist tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- (*..) and (?..) sequences to set some options.
- UTF-8 mode for pdfTeX.
- Newline conventions are not done. In particular, we should have an option for . not to match newlines. Also, \A should differ from ^, and \Z, \z and $ should differ.
- Unicode properties: \p{..} and \P{..}; \X which should match any "extended" Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.

The following features of PCRE or Perl may or may not be implemented.

- Callout with (?C...) or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential \regex_break: and then of playing well with \tl_map_break: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since \fontdimen are global.

- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns: TEX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to TEX's own `\^^x`.
- Comments: TEX already has its own system for comments.
- `\Q...\E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: XeTEX and LuaTEX serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

9 l3regex implementation

```
⟨*package⟩
⟨@@=regex⟩
```

9.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since TEX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.

- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one "position") at a time, exploring in parallel every possible path ("active thread") through the NFA, considering active threads in an order determined by the quantifiers' greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer ⟨*position*⟩, with `min_pos` $-1 \leq$ ⟨*position*⟩ $\leq$ `max_pos`. The lowest and highest positions `min_pos` -1 and `max_pos` correspond to imaginary begin and end markers (with non-existent category code and character code). `max_pos` is only set quite late in the processing.
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer ⟨*state*⟩ with `min_state` $\leq$ ⟨*state*⟩ $<$ `max_state`.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.
- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use l3intarray to manipulate arrays of integers. We also abuse TeX's `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks`⟨*state*⟩ holds the tests and actions to perform in the ⟨*state*⟩ of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last ⟨*step*⟩ in which each ⟨*state*⟩ was active.
- `\g__regex_thread_info_intarray` consists of blocks for each ⟨*thread*⟩ (with `min_thread` $\leq$ ⟨*thread*⟩ $<$ `max_thread`). Each block has 1+2`\l__regex_capturing_group_int` entries: the ⟨*state*⟩ in which the ⟨*thread*⟩ currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The ⟨*threads*⟩ are ordered starting from the best to the least preferred.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks`⟨*position*⟩ holds ⟨*tokens*⟩ which `o`- and `e`-expand to the ⟨*position*⟩-th token in the query.

- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

9.2 Helpers

`\__regex_int_eval:w`

Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

```
\cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D
```

(*End of definition for* `\__regex_int_eval:w`*.*)

`\__regex_standard_escapechar:`

Make the `\escapechar` into the standard backslash.

```
\cs_new_protected:Npn \__regex_standard_escapechar:
  { \int_set:Nn \tex_escapechar:D { `\\ } }
```

(*End of definition for* `\__regex_standard_escapechar:`*.*)

`\__regex_toks_use:w`

Unpack a `\toks` given its number.

```
\cs_new:Npn \__regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(*End of definition for* `\__regex_toks_use:w`*.*)

`\__regex_toks_clear:N`
`\__regex_toks_set:Nn`
`\__regex_toks_set:No`

Empty a `\toks` or set it to a value, given its number.

```
\cs_new_protected:Npn \__regex_toks_clear:N #1
  { \__regex_toks_set:Nn #1 { } }
\cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
\cs_new_protected:Npn \__regex_toks_set:No #1
  { \tex_toks:D #1 \exp_after:wN }
```

(*End of definition for* `\__regex_toks_clear:N` *and* `\__regex_toks_set:Nn`*.*)

`\__regex_toks_memcpy:NNn`

Copy `#3` `\toks` registers from `#2` onwards to `#1` onwards, like `C`'s `memcpy`.

```
\cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
  {
    \prg_replicate:nn {#3}
      {
        \tex_toks:D #1 = \tex_toks:D #2
        \int_incr:N #1
        \int_incr:N #2
      }
  }
```

(*End of definition for* `\__regex_toks_memcpy:NNn`*.*)

__regex_toks_put_left:Ne
__regex_toks_put_right:Ne
__regex_toks_put_right:Nn

During the building phase we wish to add e-expanded material to \toks, either to the left or to the right. The expansion is done "by hand" for optimization (these operations are used quite a lot). The Nn version of __regex_toks_put_right:Ne is provided because it is more efficient than e-expanding with \exp_not:n.

```
\cs_new_protected:Npn \__regex_toks_put_left:Ne #1#2
  {
    \cs_set_nopar:Npe \__regex_tmp:w { #2 }
    \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
      { \exp_after:wN \__regex_tmp:w \tex_the:D \tex_toks:D #1 }
  }
\cs_new_protected:Npn \__regex_toks_put_right:Ne #1#2
  {
    \cs_set_nopar:Npe \__regex_tmp:w {#2}
    \tex_toks:D #1 \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN #1 \__regex_tmp:w }
  }
\cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
  { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
```

(*End of definition for __regex_toks_put_left:Ne and __regex_toks_put_right:Ne.*)

__regex_curr_cs_to_str:

Expands to the string representation of the token (known to be a control sequence) at the current position \l__regex_curr_pos_int. It should only be used in e/x-expansion to avoid losing a leading space.

```
\cs_new:Npn \__regex_curr_cs_to_str:
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
    \l__regex_curr_token_tl
  }
```

(*End of definition for __regex_curr_cs_to_str:.*)

__regex_intarray_item:NnF
__regex_intarray_item_aux:nNF

Item of intarray, with a default value.

```
\cs_new:Npn \__regex_intarray_item:NnF #1#2
  { \exp_args:Nf \__regex_intarray_item_aux:nNF { \int_eval:n {#2} } #1 }
\cs_new:Npn \__regex_intarray_item_aux:nNF #1#2
  {
    \if_int_compare:w #1 > \c_zero_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \__kernel_intarray_item:Nn #2 {#1} }
  }
```

(*End of definition for __regex_intarray_item:NnF and __regex_intarray_item_aux:nNF.*)

__regex_maplike_break:

Analogous to \tl_map_break:, this correctly exits \tl_map_inline:nn and similar constructions and jumps to the matching \prg_break_point:Nn __regex_maplike_break: { }.

```
\cs_new:Npn \__regex_maplike_break:
  { \prg_map_break:Nn \__regex_maplike_break: { } }
```

(*End of definition for __regex_maplike_break:.*)

__regex_tl_odd_items:n
__regex_tl_even_items:n
__regex_tl_even_items_loop:nn

Map through a token list one pair at a time, leaving the odd-numbered or even-numbered items (the first item is numbered 1).

```
\cs_new:Npn \__regex_tl_odd_items:n #1 { \__regex_tl_even_items:n { ? #1 } }
\cs_new:Npn \__regex_tl_even_items:n #1
  {
    \__regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
    \prg_break_point:
  }
\cs_new:Npn \__regex_tl_even_items_loop:nn #1#2
  {
    \__regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
    { \exp_not:n {#2} }
    \__regex_tl_even_items_loop:nn
  }
```

(*End of definition for* `\__regex_tl_odd_items:n`, `\__regex_tl_even_items:n`, *and* `\__regex_tl_even_items_loop:nn`.)

9.2.1 Constants and variables

__regex_tmp:w

Temporary function used for various short-term purposes.

```
\cs_new:Npn \__regex_tmp:w { }
```

(*End of definition for* `\__regex_tmp:w`.)

\l__regex_internal_a_tl
\l__regex_internal_b_tl
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl

Temporary variables used for various purposes.

```
\tl_new:N   \l__regex_internal_a_tl
\tl_new:N   \l__regex_internal_b_tl
\int_new:N  \l__regex_internal_a_int
\int_new:N  \l__regex_internal_b_int
\int_new:N  \l__regex_internal_c_int
\bool_new:N \l__regex_internal_bool
\seq_new:N  \l__regex_internal_seq
\tl_new:N   \g__regex_internal_tl
```

(*End of definition for* `\l__regex_internal_a_tl` *and others*.)

\l__regex_build_tl

This temporary variable is specifically for use with the `tl_build` machinery.

```
\tl_new:N \l__regex_build_tl
```

(*End of definition for* `\l__regex_build_tl`.)

\c__regex_no_match_regex

This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```
\tl_const:Nn \c__regex_no_match_regex
  {
    \__regex_branch:n
      { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
  }
```

(*End of definition for* `\c__regex_no_match_regex`.)

\l__regex_balance_int

During this phase, \l__regex_balance_int counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
\int_new:N \l__regex_balance_int
```

(*End of definition for* \l__regex_balance_int.)

9.2.2 Testing characters

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int

```
\int_const:Nn \c__regex_ascii_min_int { 0 }
\int_const:Nn \c__regex_ascii_max_control_int { 31 }
\int_const:Nn \c__regex_ascii_max_int { 127 }
```

(*End of definition for* \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, *and* \c__regex_ascii_max_int.)

\c__regex_ascii_lower_int

```
\int_const:Nn \c__regex_ascii_lower_int { `a - `A }
```

(*End of definition for* \c__regex_ascii_lower_int.)

9.2.3 Internal auxiliaries

\q__regex_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__regex_recursion_stop
```

(*End of definition for* \q__regex_recursion_stop.)

\q__regex_nil

Internal quarks.

```
\quark_new:N \q__regex_nil
```

(*End of definition for* \q__regex_nil.)

egex_use_none_delimit_by_q_recursion_stop:w
_regex_use_i_delimit_by_q_recursion_stop:nw
__regex_use_none_delimit_by_q_nil:w

Functions to gobble up to a quark.

```
\cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
  #1 \q__regex_recursion_stop { }
\cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
  #1 #2 \q__regex_recursion_stop {#1}
\cs_new:Npn \__regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }
```

(*End of definition for* __regex_use_none_delimit_by_q_recursion_stop:w, __regex_use_i_delimit_by_q_recursion_stop:nw, *and* __regex_use_none_delimit_by_q_nil:w.)

__regex_quark_if_nil_p:n
__regex_quark_if_nil:nTF

Branching quark conditional.

```
\__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }
```

(*End of definition for* __regex_quark_if_nil:nTF.)

__regex_break_point:TF
__regex_break_true:w

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

⟨*test1*⟩ ... ⟨$test_n$⟩
__regex_break_point:TF {⟨*true code*⟩} {⟨*false code*⟩}

If any of the tests succeeds, it calls `\__regex_break_true:w`, which cleans up and leaves ⟨*true code*⟩ in the input stream. Otherwise, `\__regex_break_point:TF` leaves the ⟨*false code*⟩ in the input stream.

```
\cs_new_protected:Npn \__regex_break_true:w
  #1 \__regex_break_point:TF #2 #3 {#2}
\cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(*End of definition for* `\__regex_break_point:TF` *and* `\__regex_break_true:w`.)

`\__regex_item_reverse:n`

This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by -2, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```
\cs_new_protected:Npn \__regex_item_reverse:n #1
  {
    #1
    \__regex_break_point:TF { } \__regex_break_true:w
  }
```

(*End of definition for* `\__regex_item_reverse:n`.)

`\__regex_item_caseful_equal:n`
`\__regex_item_caseful_range:nn`

Simple comparisons triggering `\__regex_break_true:w` when true.

```
\cs_new_protected:Npn \__regex_item_caseful_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_item_caseful_equal:n` *and* `\__regex_item_caseful_range:nn`.)

`\__regex_item_caseless_equal:n`
`\__regex_item_caseless_range:nn`

For caseless matching, we perform the test both on the `curr_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```
\cs_new_protected:Npn \__regex_item_caseless_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
    \__regex_maybe_compute_ccc:
    \if_int_compare:w #1 = \l__regex_case_changed_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
```

```
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
    \__regex_maybe_compute_ccc:
    \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_item_caseless_equal:n` *and* `\__regex_item_caseless_range:nn`*.*)

`\__regex_compute_case_changed_char:`

This function is called when `\l__regex_case_changed_char_int` has not yet been computed. If the current character code is in the range $[65, 90]$ (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range $[97, 122]$, subtract 32.

```
\cs_new_protected:Npn \__regex_compute_case_changed_char:
  {
    \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
    \if_int_compare:w \l__regex_curr_char_int > `Z \exp_stop_f:
      \if_int_compare:w \l__regex_curr_char_int > `z \exp_stop_f: \else:
        \if_int_compare:w \l__regex_curr_char_int < `a \exp_stop_f: \else:
          \int_sub:Nn \l__regex_case_changed_char_int
            { \c__regex_ascii_lower_int }
        \fi:
      \fi:
    \else:
      \if_int_compare:w \l__regex_curr_char_int < `A \exp_stop_f: \else:
        \int_add:Nn \l__regex_case_changed_char_int
          { \c__regex_ascii_lower_int }
      \fi:
    \fi:
    \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
  }
\cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
```

(*End of definition for* `\__regex_compute_case_changed_char:`*.*)

`\__regex_item_equal:n`
`\__regex_item_range:nn`

Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```
\cs_new_eq:NN \__regex_item_equal:n ?
\cs_new_eq:NN \__regex_item_range:nn ?
```

(*End of definition for* `\__regex_item_equal:n` *and* `\__regex_item_range:nn`*.*)

`\__regex_item_catcode:nT`
`\__regex_item_catcode_reverse:nT`
`\__regex_item_catcode:`

The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```
\cs_new_protected:Npn \__regex_item_catcode:
  {
    "
```

```
    \if_case:w \l__regex_curr_catcode_int
         1       \or: 4       \or: 10      \or: 40
    \or: 100     \or:         \or: 1000    \or: 4000
    \or: 10000   \or:         \or: 100000  \or: 400000
    \or: 1000000 \or: 4000000 \else: 1*0
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode:nT #1
  {
    \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
  { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }
```

(*End of definition for* `\__regex_item_catcode:nT`*,* `\__regex_item_catcode_reverse:nT`*, and* `\__regex_item_catcode:`*.*)

`\__regex_item_exact:nn`
`\__regex_item_exact_cs:n`

This matches an exact ⟨*category*⟩-⟨*character code*⟩ pair, or an exact control sequence, more precisely one of several possible control sequences, separated by `\scan_stop:`.

```
\cs_new_protected:Npn \__regex_item_exact:nn #1#2
  {
    \if_int_compare:w #1 = \l__regex_curr_catcode_int
      \if_int_compare:w #2 = \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__regex_item_exact_cs:n #1
  {
    \int_compare:nNnTF \l__regex_curr_catcode_int = 0
      {
        \__kernel_tl_set:Ne \l__regex_internal_a_tl
          { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
        \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
          \l__regex_internal_a_tl
          { \__regex_break_true:w } { }
      }
      { }
  }
```

(*End of definition for* `\__regex_item_exact:nn` *and* `\__regex_item_exact_cs:n`*.*)

`\__regex_item_cs:n`

Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches.

```
\cs_new_protected:Npn \__regex_item_cs:n #1
  {
    \int_compare:nNnT \l__regex_curr_catcode_int = 0
      {
        \group_begin:
```

```
        \__regex_single_match:
        \__regex_disable_submatches:
        \__regex_build_for_cs:n {#1}
        \bool_set_eq:NN \l__regex_saved_success_bool
          \g__regex_success_bool
        \exp_args:Ne \__regex_match_cs:n { \__regex_curr_cs_to_str: }
        \if_meaning:w \c_true_bool \g__regex_success_bool
          \group_insert_after:N \__regex_break_true:w
        \fi:
        \bool_gset_eq:NN \g__regex_success_bool
          \l__regex_saved_success_bool
      \group_end:
    }
  }
```

(*End of definition for* `\__regex_item_cs:n`.)

9.2.4 Character property tests

`\__regex_prop_d:`
`\__regex_prop_h:`
`\__regex_prop_s:`
`\__regex_prop_v:`
`\__regex_prop_w:`
`\__regex_prop_N:`

Character property tests for `\d`, `\W`, *etc.* These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\d=[0-9]`, `\w=[0-9A-Z_a-z]`, `\s=[\␣\^^I\^^J\^^L\^^M]`, `\h=[\␣\^^I]`, `\v=[\^^J-\^^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```
\cs_new_protected:Npn \__regex_prop_d:
  { \__regex_item_caseful_range:nn { `0 } { `9 } }
\cs_new_protected:Npn \__regex_prop_h:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
  }
\cs_new_protected:Npn \__regex_prop_s:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
    \__regex_item_caseful_equal:n { `\^^J }
    \__regex_item_caseful_equal:n { `\^^L }
    \__regex_item_caseful_equal:n { `\^^M }
  }
\cs_new_protected:Npn \__regex_prop_v:
  { \__regex_item_caseful_range:nn { `\^^J } { `\^^M } } % lf, vtab, ff, cr
\cs_new_protected:Npn \__regex_prop_w:
  {
    \__regex_item_caseful_range:nn { `a } { `z }
    \__regex_item_caseful_range:nn { `A } { `Z }
    \__regex_item_caseful_range:nn { `0 } { `9 }
    \__regex_item_caseful_equal:n { `_ }
  }
\cs_new_protected:Npn \__regex_prop_N:
  {
    \__regex_item_reverse:n
      { \__regex_item_caseful_equal:n { `\^^J } }
  }
```

(*End of definition for* `\__regex_prop_d:` *and others.*)

`\__regex_posix_alnum:`
`\__regex_posix_alpha:`
`\__regex_posix_ascii:`
`\__regex_posix_blank:`
`\__regex_posix_cntrl:`
`\__regex_posix_digit:`
`\__regex_posix_graph:`
`\__regex_posix_lower:`
`\__regex_posix_print:`
`\__regex_posix_punct:`
`\__regex_posix_space:`
`\__regex_posix_upper:`
`\__regex_posix_word:`
`\__regex_posix_xdigit:`

POSIX properties. No surprise.

```
\cs_new_protected:Npn \__regex_posix_alnum:
  { \__regex_posix_alpha: \__regex_posix_digit: }
\cs_new_protected:Npn \__regex_posix_alpha:
  { \__regex_posix_lower: \__regex_posix_upper: }
\cs_new_protected:Npn \__regex_posix_ascii:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_int
  }
\cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\cs_new_protected:Npn \__regex_posix_cntrl:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_control_int
    \__regex_item_caseful_equal:n \c__regex_ascii_max_int
  }
\cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
\cs_new_protected:Npn \__regex_posix_graph:
  { \__regex_item_caseful_range:nn { `! } { `\~ } }
\cs_new_protected:Npn \__regex_posix_lower:
  { \__regex_item_caseful_range:nn { `a } { `z } }
\cs_new_protected:Npn \__regex_posix_print:
  { \__regex_item_caseful_range:nn { `\  } { `\~ } }
\cs_new_protected:Npn \__regex_posix_punct:
  {
    \__regex_item_caseful_range:nn { `! } { `/ }
    \__regex_item_caseful_range:nn { `: } { `@ }
    \__regex_item_caseful_range:nn { `[ } { `` }
    \__regex_item_caseful_range:nn { `\{ } { `\~ }
  }
\cs_new_protected:Npn \__regex_posix_space:
  {
    \__regex_item_caseful_equal:n { `\  }
    \__regex_item_caseful_range:nn { `\^^I } { `\^^M }
  }
\cs_new_protected:Npn \__regex_posix_upper:
  { \__regex_item_caseful_range:nn { `A } { `Z } }
\cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
\cs_new_protected:Npn \__regex_posix_xdigit:
  {
    \__regex_posix_digit:
    \__regex_item_caseful_range:nn { `A } { `F }
    \__regex_item_caseful_range:nn { `a } { `f }
  }
```

(*End of definition for* `\__regex_posix_alnum:` *and others.*)

9.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were "raw" characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `\__regex_escape_use:nnnn` ⟨*inline 1*⟩ ⟨*inline 2*⟩ ⟨*inline 3*⟩ `{`⟨*token list*⟩`}` The ⟨*token list*⟩ is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function ⟨*inline 1*⟩, and escaped characters are fed to the function ⟨*inline 2*⟩ within an `e`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to ⟨*inline 3*⟩. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an `e`-expanding assignment.

`\__regex_escape_use:nnnn`

The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through `#4` once, applying `#1`, `#2`, or `#3` as relevant to each character (after de-escaping it).

```
\cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
  {
    \group_begin:
      \tl_clear:N \l__regex_internal_a_tl
      \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
      \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
      \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
      \__regex_standard_escapechar:
      \__kernel_tl_gset:Ne \g__regex_internal_tl
        { \__kernel_str_to_other_fast:n {#4} }
      \tl_put_right:Ne \l__regex_internal_a_tl
        {
          \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
          \scan_stop: \prg_break_point:
        }
      \exp_after:wN
    \group_end:
    \l__regex_internal_a_tl
  }
```

(*End of definition for* `\__regex_escape_use:nnnn`.)

`\__regex_escape_loop:N`
`\__regex_escape_\:w`

`\__regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are "escaped".

```
\cs_new:Npn \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
      { \__regex_escape_unescaped:N #1 }
    \__regex_escape_loop:N
  }
```

```
\cs_new:cpn { __regex_escape_ \c_backslash_str :w }
    \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
      { \__regex_escape_escaped:N #1 }
    \__regex_escape_loop:N
  }
```

(*End of definition for* `\__regex_escape_loop:N` *and* `\__regex_escape_\:w`.)

__regex_escape_unescaped:N
__regex_escape_escaped:N
__regex_escape_raw:N

Those functions are never called before being given a new meaning, so their definitions here don't matter.

```
\cs_new_eq:NN \__regex_escape_unescaped:N ?
\cs_new_eq:NN \__regex_escape_escaped:N   ?
\cs_new_eq:NN \__regex_escape_raw:N       ?
```

(*End of definition for* `\__regex_escape_unescaped:N`*,* `\__regex_escape_escaped:N`*, and* `\__regex_escape_raw:N`.)

__regex_escape_\scan_stop::w
__regex_escape_/\scan_stop::w
__regex_escape_/a:w
__regex_escape_/e:w
__regex_escape_/f:w
__regex_escape_/n:w
__regex_escape_/r:w
__regex_escape_/t:w
__regex_escape_␣:w

The loop is ended upon seeing the end-marker "`break`", with an error if the string ended in a backslash. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

```
\cs_new_eq:cN { __regex_escape_ \iow_char:N\\scan_stop: :w } \prg_break:
\cs_new:cpn { __regex_escape_/ \iow_char:N\\scan_stop: :w }
  {
    \msg_expandable_error:nn { regex } { trailing-backslash }
    \prg_break:
  }
\cs_new:cpn { __regex_escape_~:w } { }
\cs_new:cpe { __regex_escape_/a:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
\cs_new:cpe { __regex_escape_/t:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
\cs_new:cpe { __regex_escape_/n:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
\cs_new:cpe { __regex_escape_/f:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
\cs_new:cpe { __regex_escape_/r:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
\cs_new:cpe { __regex_escape_/e:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }
```

(*End of definition for* `\__regex_escape_\scan_stop::w` *and others.*)

__regex_escape_/x:w
__regex_escape_x_end:w
__regex_escape_x_large:n

When `\x` is encountered, `\__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `\__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `\__regex_escape_raw:N` on the corresponding character token.

```
\cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
  {
    \exp_after:wN \__regex_escape_x_end:w
    \int_value:w "0 \__regex_escape_x_test:N
  }
\cs_new:Npn \__regex_escape_x_end:w #1 ;
  {
```

```
    \int_compare:nNnTF {#1} > \c_max_char_int
      {
        \msg_expandable_error:nnff { regex } { x-overflow }
          {#1} { \int_to_Hex:n {#1} }
      }
      {
        \exp_last_unbraced:Nf \__regex_escape_raw:N
          { \char_generate:nn {#1} { 12 } }
      }
  }
```

(*End of definition for* `\__regex_escape_/x:w`*,* `\__regex_escape_x_end:w`*, and* `\__regex_escape_x_large:n`*.*)

`\__regex_escape_x_test:N`
`\__regex_escape_x_testii:N`

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `\__regex_escape_x_loop:N` or `\__regex_escape_x:N`.

```
\cs_new:Npn \__regex_escape_x_test:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_i:nnn \exp_after:wN ;
    \fi:
    \use:n
      {
        \if_charcode:w \c_space_token #1
          \exp_after:wN \__regex_escape_x_test:N
        \else:
          \exp_after:wN \__regex_escape_x_testii:N
          \exp_after:wN #1
        \fi:
      }
  }
\cs_new:Npn \__regex_escape_x_testii:N #1
  {
    \if_charcode:w \c_left_brace_str #1
      \exp_after:wN \__regex_escape_x_loop:N
    \else:
      \__regex_hexadecimal_use:NTF #1
        { \exp_after:wN \__regex_escape_x:N }
        { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
    \fi:
  }
```

(*End of definition for* `\__regex_escape_x_test:N` *and* `\__regex_escape_x_testii:N`*.*)

`\__regex_escape_x:N`

This looks for the second digit in the unbraced case.

```
\cs_new:Npn \__regex_escape_x:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_i:nnn \exp_after:wN ;
    \fi:
    \use:n
      {
```

```
        \__regex_hexadecimal_use:NTF #1
          { ; \__regex_escape_loop:N }
          { ; \__regex_escape_loop:N #1 }
      }
  }
```

(*End of definition for* `\__regex_escape_x:N`.)

`\__regex_escape_x_loop:N`
`\__regex_escape_x_loop_error:`

Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```
\cs_new:Npn \__regex_escape_x_loop:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_ii:nnn
    \fi:
    \use_ii:nn
      { ; \__regex_escape_x_loop_error:n { } {#1} }
      {
        \__regex_hexadecimal_use:NTF #1
          { \__regex_escape_x_loop:N }
          {
            \token_if_eq_charcode:NNTF \c_space_token #1
              { \__regex_escape_x_loop:N }
              {
                ;
                \exp_after:wN
                \token_if_eq_charcode:NNTF \c_right_brace_str #1
                  { \__regex_escape_loop:N }
                  { \__regex_escape_x_loop_error:n {#1} }
              }
          }
      }
  }
\cs_new:Npn \__regex_escape_x_loop_error:n #1
  {
    \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
    \__regex_escape_loop:N #1
  }
```

(*End of definition for* `\__regex_escape_x_loop:N` *and* `\__regex_escape_x_loop_error:`.)

`\__regex_hexadecimal_use:NTF`

TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```
\prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
  {
    \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \if_case:w
        \int_eval:n { \exp_after:wN ‘ \token_to_str:N #1 - ‘a }
           A
      \or: B
      \or: C
      \or: D
```

```
      \or: E
      \or: F
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\__regex_hexadecimal_use:NTF`.)

`\__regex_char_if_alphanumeric:NTF`
`\__regex_char_if_special:NTF`

These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as "raw" characters. Namely,

- alphanumerics are "raw" if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are "raw" if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always "raw".

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, "alphanumeric" means `0`–`9`, `A`–`Z`, `a`–`z`; "special" character means non-alphanumeric but printable ascii, from space (hex `20`) to `del` (hex `7E`).

```
\prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
  {
    \if_int_compare:w ‘#1 > ‘Z \exp_stop_f:
      \if_int_compare:w ‘#1 > ‘z \exp_stop_f:
        \if_int_compare:w ‘#1 < \c__regex_ascii_max_int
          \prg_return_true: \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w ‘#1 < ‘a \exp_stop_f:
          \prg_return_true: \else: \prg_return_false: \fi:
      \fi:
    \else:
      \if_int_compare:w ‘#1 > ‘9 \exp_stop_f:
        \if_int_compare:w ‘#1 < ‘A \exp_stop_f:
          \prg_return_true: \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w ‘#1 < ‘0 \exp_stop_f:
          \if_int_compare:w ‘#1 < ‘\ \exp_stop_f:
            \prg_return_false: \else: \prg_return_true: \fi:
        \else: \prg_return_false: \fi:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
  {
    \if_int_compare:w ‘#1 > ‘Z \exp_stop_f:
      \if_int_compare:w ‘#1 > ‘z \exp_stop_f:
        \prg_return_false:
```

```
      \else:
        \if_int_compare:w `#1 < `a \exp_stop_f:
          \prg_return_false: \else: \prg_return_true: \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 > `9 \exp_stop_f:
        \if_int_compare:w `#1 < `A \exp_stop_f:
          \prg_return_false: \else: \prg_return_true: \fi:
      \else:
        \if_int_compare:w `#1 < `0 \exp_stop_f:
          \prg_return_false: \else: \prg_return_true: \fi:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_char_if_alphanumeric:NTF` *and* `\__regex_char_if_special:NTF`.)

9.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a "compiled" regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `\__regex_class:NnnnN` ⟨*boolean*⟩ {⟨*tests*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩
- `\__regex_group:nnnN` {⟨*branches*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩, also `\__regex_group_no_capture:nnnN` and `\__regex_group_resetting:nnnN` with the same syntax.
- `\__regex_branch:n` {⟨*contents*⟩}
- `\__regex_command_K:`
- `\__regex_assertion:Nn` ⟨*boolean*⟩ {⟨*assertion test*⟩}, where the ⟨*assertion test*⟩ is `\__regex_b_test:` or `\__regex_Z_test:` or `\__regex_A_test:` or `\__regex_G_test:`

Tests can be the following:

- `\__regex_item_caseful_equal:n` {⟨*char code*⟩}
- `\__regex_item_caseless_equal:n` {⟨*char code*⟩}
- `\__regex_item_caseful_range:nn` {⟨*min*⟩} {⟨*max*⟩}
- `\__regex_item_caseless_range:nn` {⟨*min*⟩} {⟨*max*⟩}
- `\__regex_item_catcode:nT` {⟨*catcode bitmap*⟩} {⟨*tests*⟩}
- `\__regex_item_catcode_reverse:nT` {⟨*catcode bitmap*⟩} {⟨*tests*⟩}
- `\__regex_item_reverse:n` {⟨*tests*⟩}
- `\__regex_item_exact:nn` {⟨*catcode*⟩} {⟨*char code*⟩}
- `\__regex_item_exact_cs:n` {⟨*csnames*⟩}, more precisely given as ⟨*csname*⟩ `\scan_stop:` ⟨*csname*⟩ `\scan_stop:` ⟨*csname*⟩ and so on in a brace group.
- `\__regex_item_cs:n` {⟨*compiled regex*⟩}

9.3.1 Variables used when compiling

\l__regex_group_level_int

We make sure to open the same number of groups as we close.

```
\int_new:N \l__regex_group_level_int
```

(*End of definition for* `\l__regex_group_level_int`*.*)

\l__regex_mode_int
\c__regex_cs_in_class_mode_int
\c__regex_cs_mode_int
\c__regex_outer_mode_int
\c__regex_catcode_mode_int
\c__regex_class_mode_int
\c__regex_catcode_in_class_mode_int

While compiling, ten modes are recognized, labelled -63, -23, -6, -2, 0, 2, 3, 6, 23, 63. See section 9.3.3. We only define some of these as constants.

```
\int_new:N \l__regex_mode_int
\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\int_const:Nn \c__regex_cs_mode_int { -2 }
\int_const:Nn \c__regex_outer_mode_int { 0 }
\int_const:Nn \c__regex_catcode_mode_int { 2 }
\int_const:Nn \c__regex_class_mode_int { 3 }
\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(*End of definition for* `\l__regex_mode_int` *and others.*)

\l__regex_catcodes_int
\l__regex_default_catcodes_int
\l__regex_catcodes_bool

We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c, for all allowed catcodes c. The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
\int_new:N \l__regex_catcodes_int
\int_new:N \l__regex_default_catcodes_int
\bool_new:N \l__regex_catcodes_bool
```

(*End of definition for* `\l__regex_catcodes_int`*,* `\l__regex_default_catcodes_int`*, and* `\l__regex_catcodes_bool`*.*)

\c__regex_catcode_C_int
\c__regex_catcode_B_int
\c__regex_catcode_E_int
\c__regex_catcode_M_int
\c__regex_catcode_T_int
\c__regex_catcode_P_int
\c__regex_catcode_U_int
\c__regex_catcode_D_int
\c__regex_catcode_S_int
\c__regex_catcode_L_int
\c__regex_catcode_O_int
\c__regex_catcode_A_int
\c__regex_all_catcodes_int

Constants: 4^c for each category, and the sum of all powers of 4.

```
\int_const:Nn \c__regex_catcode_C_int { "1 }
\int_const:Nn \c__regex_catcode_B_int { "4 }
\int_const:Nn \c__regex_catcode_E_int { "10 }
\int_const:Nn \c__regex_catcode_M_int { "40 }
\int_const:Nn \c__regex_catcode_T_int { "100 }
\int_const:Nn \c__regex_catcode_P_int { "1000 }
\int_const:Nn \c__regex_catcode_U_int { "4000 }
\int_const:Nn \c__regex_catcode_D_int { "10000 }
\int_const:Nn \c__regex_catcode_S_int { "100000 }
\int_const:Nn \c__regex_catcode_L_int { "400000 }
\int_const:Nn \c__regex_catcode_O_int { "1000000 }
\int_const:Nn \c__regex_catcode_A_int { "4000000 }
\int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(*End of definition for* `\c__regex_catcode_C_int` *and others.*)

\l__regex_internal_regex

The compilation step stores its result in this variable.

```
\cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(*End of definition for* `\l__regex_internal_regex`*.*)

\l__regex_show_prefix_seq

This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
\seq_new:N \l__regex_show_prefix_seq
```

(*End of definition for* `\l__regex_show_prefix_seq`.)

\l__regex_show_lines_int

A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
\int_new:N \l__regex_show_lines_int
```

(*End of definition for* `\l__regex_show_lines_int`.)

9.3.2 Generic helpers used when compiling

__regex_two_if_eq:NNNNTF

Used to compare pairs of things like `\__regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of `\if:w` is very useful as that means we can use `\c_left_brace_str` and the like.

```
\prg_new_conditional:Npnn \__regex_two_if_eq:NNNN #1#2#3#4 { TF }
  {
    \if_meaning:w #1 #3
      \if:w #2 #4
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__regex_two_if_eq:NNNNTF`.)

__regex_get_digits:NTFw
__regex_get_digits_loop:w

If followed by some raw digits, collect them one by one in the integer variable `#1`, and take the `true` branch. Otherwise, take the `false` branch.

```
\cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
  {
    \__regex_if_raw_digit:NNTF #4 #5
      { #1 = #5 \__regex_get_digits_loop:nw {#2} }
      { #3 #4 #5 }
  }
\cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
  {
    \__regex_if_raw_digit:NNTF #2 #3
      { #3 \__regex_get_digits_loop:nw {#1} }
      { \scan_stop: #1 #2 #3 }
  }
```

(*End of definition for* `\__regex_get_digits:NTFw` *and* `\__regex_get_digits_loop:w`.)

`\__regex_if_raw_digit:NNTF` Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```
\prg_new_conditional:Npnn \__regex_if_raw_digit:NN #1#2 { TF }
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \if_int_compare:w 1 < 1 #2 \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__regex_if_raw_digit:NNTF`.)

9.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

-6 `[\c{...}]` control sequence in a class,

-2 `\c{...}` control sequence,

0 `...` outer,

2 `\c...` catcode test,

6 `[\c...]` catcode test in a class,

-63 `[\c{[...]}]` class inside mode -6,

-23 `\c{[...]}` class inside mode -2,

3 `[...]` class inside mode 0,

23 `\c[...]` class inside mode 2,

63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \to (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.

- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`\__regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```
\cs_new:Npn \__regex_if_in_class:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End of definition for* `\__regex_if_in_class:TF`.)

`\__regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
\cs_new:Npn \__regex_if_in_cs:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_ii:nn
    \else:
      \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_if_in_cs:TF`.)

`\__regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```
\cs_new:Npn \__regex_if_in_class_or_catcode:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_if_in_class_or_catcode:TF`.)

__regex_if_within_catcode:TF This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```
\cs_new:Npn \__regex_if_within_catcode:TF
  {
    \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End of definition for* `\__regex_if_within_catcode:TF`.)

__regex_chk_c_allowed:T The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```
\cs_new_protected:Npn \__regex_chk_c_allowed:T
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
      \exp_after:wN \use:n
    \else:
      \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use:n
      \else:
        \msg_error:nn { regex } { c-bad-mode }
        \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_chk_c_allowed:T`.)

__regex_mode_quit_c: This function changes the mode as it is needed just after a catcode test.

```
\cs_new_protected:Npn \__regex_mode_quit_c:
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int =
        \c__regex_catcode_in_class_mode_int
        \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_mode_quit_c:`.)

9.3.4 Framework

__regex_compile:w
__regex_compile_end: Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with `e`-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```
\cs_new_protected:Npn \__regex_compile:w
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_group_level_int
      \int_set_eq:NN \l__regex_default_catcodes_int
        \c__regex_all_catcodes_int
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \cs_set:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
      \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected:Npn \__regex_compile_end:
  {
      \__regex_if_in_class:TF
        {
          \msg_error:nn { regex } { missing-rbrack }
          \use:c { __regex_compile_]: }
          \prg_do_nothing: \prg_do_nothing:
        }
        { }
      \if_int_compare:w \l__regex_group_level_int > \c_zero_int
        \msg_error:nne { regex } { missing-rparen }
          { \int_use:N \l__regex_group_level_int }
        \prg_replicate:nn
          { \l__regex_group_level_int }
          {
              \tl_build_put_right:Nn \l__regex_build_tl
                {
                  \if_false: { \fi: }
                  \if_false: { \fi: } { 1 } { 0 } \c_true_bool
                }
              \tl_build_end:N \l__regex_build_tl
              \exp_args:NNNo
            \group_end:
            \tl_build_put_right:Nn \l__regex_build_tl
              { \l__regex_build_tl }
          }
      \fi:
      \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNNe
    \group_end:
    \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
  }
```

(*End of definition for* `\__regex_compile:w` *and* `\__regex_compile_end:`.)

`\__regex_compile:n` The compilation is done between `\__regex_compile:w` and `\__regex_compile_end:`, starting in mode 0. Then `\__regex_escape_use:nnnn` distinguishes special characters, escaped alphanumerics, and raw characters, interpreting `\a`, `\x` and other sequences. The 4 trailing `\prg_do_nothing:` are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any `\c{...}` is properly closed. No need to check that brackets are closed properly since `\__regex_compile_end:` does that. However, catch the case of a trailing `\cL` construction.

```
\cs_new_protected:Npn \__regex_compile:n #1
  {
    \__regex_compile:w
      \__regex_standard_escapechar:
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
      \__regex_escape_use:nnnn
        {
          \__regex_char_if_special:NTF ##1
            \__regex_compile_special:N \__regex_compile_raw:N ##1
        }
        {
          \__regex_char_if_alphanumeric:NTF ##1
            \__regex_compile_escaped:N \__regex_compile_raw:N ##1
        }
        { \__regex_compile_raw:N ##1 }
        { #1 }
      \prg_do_nothing: \prg_do_nothing:
      \prg_do_nothing: \prg_do_nothing:
      \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
        { \msg_error:nn { regex } { c-trailing } }
      \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
        {
          \msg_error:nn { regex } { c-missing-rbrace }
          \__regex_compile_end_cs:
          \prg_do_nothing: \prg_do_nothing:
          \prg_do_nothing: \prg_do_nothing:
        }
    \__regex_compile_end:
  }
```

(*End of definition for* `\__regex_compile:n`.)

`\__regex_compile_use:n` Use a regex, regardless of whether it is given as a string (in which case we need to compile) or as a regex variable. This is used for `\regex_match_case:nn` and related functions to allow a mixture of explicit regex and regex variables.

```
\cs_new_protected:Npn \__regex_compile_use:n #1
  {
    \tl_if_single_token:nT {#1}
      {
        \exp_after:wN \__regex_compile_use_aux:w
        \token_to_meaning:N #1 ~ \q__regex_nil
      }
    \__regex_compile:n {#1} \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
  {
    \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
      { \use_ii:nnn }
  }
```

(*End of definition for* `\__regex_compile_use:n`.)

__regex_compile_escaped:N
__regex_compile_special:N

If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```
\cs_new_protected:Npn \__regex_compile_special:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_#1: }
      { \__regex_compile_raw:N #1 }
  }
\cs_new_protected:Npn \__regex_compile_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_/#1: }
      { \__regex_compile_raw:N #1 }
  }
```

(*End of definition for* `\__regex_compile_escaped:N` *and* `\__regex_compile_special:N`.)

__regex_compile_one:n

This is used after finding one "test", such as `\d`, or a raw character. If that followed a catcode test (*e.g.*, `\cL`), then restore the mode. If we are not in a class, then the test is "standalone", and we need to add `\__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```
\cs_new_protected:Npn \__regex_compile_one:n #1
  {
    \__regex_mode_quit_c:
    \__regex_if_in_class:TF { }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
      }
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \if_int_compare:w \l__regex_catcodes_int <
          \c__regex_all_catcodes_int
          \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
            { \exp_not:N \exp_not:n {#1} }
        \else:
          \exp_not:N \exp_not:n {#1}
        \fi:
      }
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
  }
```

(*End of definition for* `\__regex_compile_one:n`.)

__regex_compile_abort_tokens:n
__regex_compile_abort_tokens:e

This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```
\cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
  {
    \use:e
      {
        \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
          \__regex_compile_raw:N
      }
```

```
  }
\cs_generate_variant:Nn \__regex_compile_abort_tokens:n { e }
```

(*End of definition for* `\__regex_compile_abort_tokens:n`.)

9.3.5 Quantifiers

`\__regex_compile_if_quantifier:TFw`

This looks ahead and checks whether there are any quantifier (special character equal to either of `?+*{`). This is useful for the `\u` and `\ur` escape sequences.

```
\cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
  {
    \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
      { \cs_if_exist:cTF { __regex_compile_quantifier_#4:w } }
      { \use_ii:nn }
    {#1} {#2} #3 #4
  }
```

(*End of definition for* `\__regex_compile_if_quantifier:TFw`.)

`\__regex_compile_quantifier:w`

This looks ahead and finds any quantifier (special character equal to either of `?+*{`).

```
\cs_new_protected:Npn \__regex_compile_quantifier:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
        \cs_if_exist_use:cF { __regex_compile_quantifier_#2:w }
          { \__regex_compile_quantifier_none: #1 #2 }
      }
      { \__regex_compile_quantifier_none: #1 #2 }
  }
```

(*End of definition for* `\__regex_compile_quantifier:w`.)

`\__regex_compile_quantifier_none:`
`\__regex_compile_quantifier_abort:eNN`

Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

```
\cs_new_protected:Npn \__regex_compile_quantifier_none:
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_abort:eNN #1#2#3
  {
    \__regex_compile_quantifier_none:
    \msg_warning:nnee { regex } { invalid-quantifier } {#1} {#3}
    \__regex_compile_abort_tokens:e {#1}
    #2 #3
  }
```

(*End of definition for* `\__regex_compile_quantifier_none:` *and* `\__regex_compile_quantifier_abort:eNN`.)

`\__regex_compile_quantifier_lazyness:nnNN`

Once the “main” quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `\__regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```
\cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ?
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
      }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
        #3 #4
      }
  }
```

(*End of definition for* `\__regex_compile_quantifier_lazyness:nnNN`.)

__regex_compile_quantifier_?:w
__regex_compile_quantifier_*:w
__regex_compile_quantifier_+:w

For each "basic" quantifier, `?`, `*`, `+`, feed the correct arguments to `\__regex_compile_quantifier_lazyness:nnNN`, -1 means that there is no upper bound on the number of repetitions.

```
\cs_new_protected:cpn { __regex_compile_quantifier_?:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_*:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_+:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(*End of definition for* `\__regex_compile_quantifier_?:w`, `\__regex_compile_quantifier_*:w`, *and* `\__regex_compile_quantifier_+:w`.)

__regex_compile_quantifier_{:w
__regex_compile_quantifier_braced_auxi:w
__regex_compile_quantifier_braced_auxii:w
__regex_compile_quantifier_braced_auxiii:w

Three possible syntaxes: `{`⟨*int*⟩`}`, `{`⟨*int*⟩`,}`, or `{`⟨*int*⟩`,`⟨*int*⟩`}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as `raw` characters, including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the number is followed by a right brace, the range is $[a, a]$. If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range $[a, \infty]$ or $[a, b]$, encoded as $\{a\}\{-1\}$ and $\{a\}\{b-a\}$.

```
\cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
  {
    \__regex_get_digits:NTFw \l__regex_internal_a_int
      { \__regex_compile_quantifier_braced_auxi:w }
      { \__regex_compile_quantifier_abort:eNN { \c_left_brace_str } }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
  {
    \str_case_e:nnF { #1 #2 }
      {
        { \__regex_compile_special:N \c_right_brace_str }
          {
            \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
              { \int_use:N \l__regex_internal_a_int } { 0 }
          }
        { \__regex_compile_special:N , }
          {
            \__regex_get_digits:NTFw \l__regex_internal_b_int
              { \__regex_compile_quantifier_braced_auxiii:w }
```

```
              { \__regex_compile_quantifier_braced_auxii:w }
          }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
      {
        \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
          { \int_use:N \l__regex_internal_a_int } { -1 }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
      {
        \if_int_compare:w \l__regex_internal_a_int >
          \l__regex_internal_b_int
          \msg_error:nnee { regex } { backwards-quantifier }
            { \int_use:N \l__regex_internal_a_int }
            { \int_use:N \l__regex_internal_b_int }
          \int_zero:N \l__regex_internal_b_int
        \else:
          \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
        \fi:
        \exp_args:Noo \__regex_compile_quantifier_lazyness:nnNN
          { \int_use:N \l__regex_internal_a_int }
          { \int_use:N \l__regex_internal_b_int }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          {
            \c_left_brace_str
            \int_use:N \l__regex_internal_a_int ,
            \int_use:N \l__regex_internal_b_int
          }
        #1 #2
      }
  }
```

(*End of definition for* `\__regex_compile_quantifier_{:w` *and others.*)

9.3.6 Raw characters

__regex_compile_raw_error:N

Within character classes, and following catcode tests, some escaped alphanumeric sequences such as \b do not have any meaning. They are replaced by a raw character, after spitting out an error.

```
\cs_new_protected:Npn \__regex_compile_raw_error:N #1
  {
    \msg_error:nne { regex } { bad-escape } {#1}
    \__regex_compile_raw:N #1
  }
```

(*End of definition for* __regex_compile_raw_error:N.)

__regex_compile_raw:N

If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character #1 matches itself.

```
\cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
  {
    \__regex_if_in_class:TF
      {
        \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N -
          { \__regex_compile_range:Nw #1 }
          {
            \__regex_compile_one:n
              { \__regex_item_equal:n { \int_value:w `#1 } }
            #2 #3
          }
      }
      {
        \__regex_compile_one:n
          { \__regex_item_equal:n { \int_value:w `#1 } }
        #2 #3
      }
  }
```

(*End of definition for* __regex_compile_raw:N.)

__regex_compile_range:Nw
__regex_if_end_range:NNTF

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```
\prg_new_protected_conditional:Npnn \__regex_if_end_range:NN #1#2 { TF }
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \prg_return_true:
    \else:
      \if_meaning:w \__regex_compile_special:N #1
        \if_charcode:w ] #2
          \prg_return_false:
        \else:
          \prg_return_true:
        \fi:
      \else:
        \prg_return_false:
      \fi:
    \fi:
```

```
  }
\cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
  {
    \__regex_if_end_range:NNTF #2 #3
      {
        \if_int_compare:w `#1 > `#3 \exp_stop_f:
          \msg_error:nnee { regex } { range-backwards } {#1} {#3}
        \else:
          \tl_build_put_right:Ne \l__regex_build_tl
            {
              \if_int_compare:w `#1 = `#3 \exp_stop_f:
                \__regex_item_equal:n
              \else:
                \__regex_item_range:nn { \int_value:w `#1 }
              \fi:
              { \int_value:w `#3 }
            }
        \fi:
      }
      {
        \msg_warning:nnee { regex } { range-missing-end }
          {#1} { \c_backslash_str #3 }
        \tl_build_put_right:Ne \l__regex_build_tl
          {
            \__regex_item_equal:n { \int_value:w `#1 \exp_stop_f: }
            \__regex_item_equal:n { \int_value:w `- \exp_stop_f: }
          }
        #2#3
      }
  }
```

(*End of definition for* `\__regex_compile_range:Nw` *and* `\__regex_if_end_range:NNTF`.)

9.3.7 Character properties

`\__regex_compile_.:`
`\__regex_prop_.:`

In a class, the dot has no special meaning. Outside, insert `\__regex_prop_.:`, which matches any character or control sequence, and refuses -2 (end-marker).

```
\cs_new_protected:cpe { __regex_compile_.: }
  {
    \exp_not:N \__regex_if_in_class:TF
      { \__regex_compile_raw:N . }
      { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
  }
\cs_new_protected:cpn { __regex_prop_.: }
  {
    \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*End of definition for* `\__regex_compile_.:` *and* `\__regex_prop_.:`.)

`\__regex_compile_/d:`
`\__regex_compile_/D:`
`\__regex_compile_/h:`
`\__regex_compile_/H:`
`\__regex_compile_/s:`
`\__regex_compile_/S:`
`\__regex_compile_/v:`
`\__regex_compile_/V:`
`\__regex_compile_/w:`
`\__regex_compile_/W:`
`\__regex_compile_/N:`

The constants `\__regex_prop_d:`, *etc.* hold a list of tests which match the corresponding character class, and jump to the `\__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpe { __regex_compile_/#1: }
      { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
    \cs_new_protected:cpe { __regex_compile_/#2: }
      {
        \__regex_compile_one:n
          { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#1: } } }
      }
  }
\__regex_tmp:w d D
\__regex_tmp:w h H
\__regex_tmp:w s S
\__regex_tmp:w v V
\__regex_tmp:w w W
\cs_new_protected:cpn { __regex_compile_/N: }
  { \__regex_compile_one:n \__regex_prop_N: }
```

(*End of definition for* `\__regex_compile_/d:` *and others.*)

9.3.8 Anchoring and simple assertions

`\__regex_compile_anchor_letter:NNN`
`\__regex_compile_/A:`
`\__regex_compile_/G:`
`\__regex_compile_/Z:`
`\__regex_compile_/z:`
`\__regex_compile_/b:`
`\__regex_compile_/B:`
`\__regex_compile_^:`
`\__regex_compile_$:`

In modes where assertions are forbidden, anchors such as `\A` produce an error (`\A` is invalid in classes); otherwise they add an `\__regex_assertion:Nn` test as appropriate (the only negative assertion is `\B`). The test functions are defined later. The implementation for `$` and `^` is only different from `\A` etc because these are valid in a class.

```
\cs_new_protected:Npn \__regex_compile_anchor_letter:NNN #1#2#3
  {
    \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw_error:N #1 }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \__regex_assertion:Nn #2 {#3} }
      }
  }
\cs_new_protected:cpn { __regex_compile_/A: }
  { \__regex_compile_anchor_letter:NNN A \c_true_bool \__regex_A_test: }
\cs_new_protected:cpn { __regex_compile_/G: }
  { \__regex_compile_anchor_letter:NNN G \c_true_bool \__regex_G_test: }
\cs_new_protected:cpn { __regex_compile_/Z: }
  { \__regex_compile_anchor_letter:NNN Z \c_true_bool \__regex_Z_test: }
\cs_new_protected:cpn { __regex_compile_/z: }
  { \__regex_compile_anchor_letter:NNN z \c_true_bool \__regex_Z_test: }
\cs_new_protected:cpn { __regex_compile_/b: }
  { \__regex_compile_anchor_letter:NNN b \c_true_bool \__regex_b_test: }
\cs_new_protected:cpn { __regex_compile_/B: }
  { \__regex_compile_anchor_letter:NNN B \c_false_bool \__regex_b_test: }
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpn { __regex_compile_#1: }
      {
        \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw:N #1 }
          {
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_assertion:Nn \c_true_bool {#2} }
```

```
          }
      }
  }
\exp_args:Ne \__regex_tmp:w { \iow_char:N \^ } { \__regex_A_test: }
\exp_args:Ne \__regex_tmp:w { \iow_char:N \$ } { \__regex_Z_test: }
```

(*End of definition for* `\__regex_compile_anchor_letter:NNN` *and others.*)

9.3.9 Character classes

`\__regex_compile_]:`

Outside a class, right brackets have no meaning. In a class, change the mode ($m \to (m-15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[...\cL[...]...]`). quantifiers.

```
\cs_new_protected:cpn { __regex_compile_]: }
  {
    \__regex_if_in_class:TF
      {
        \if_int_compare:w \l__regex_mode_int >
          \c__regex_catcode_in_class_mode_int
          \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
        \fi:
        \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
        \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
        \if_int_odd:w \l__regex_mode_int \else:
          \exp_after:wN \__regex_compile_quantifier:w
        \fi:
      }
      { \__regex_compile_raw:N ] }
  }
```

(*End of definition for* `\__regex_compile_]:`*.*)

`\__regex_compile_[:`

In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c`⟨*category*⟩, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```
\cs_new_protected:cpn { __regex_compile_[: }
  {
    \__regex_if_in_class:TF
      { \__regex_compile_class_posix_test:w }
      {
        \__regex_if_within_catcode:TF
          {
            \exp_after:wN \__regex_compile_class_catcode:w
              \int_use:N \l__regex_catcodes_int ;
          }
          { \__regex_compile_class_normal:w }
      }
  }
```

(*End of definition for* `\__regex_compile_[:`*.*)

`\__regex_compile_class_normal:w`

In the "normal" case, we insert `\__regex_class:NnnnN` ⟨*boolean*⟩ in the compiled code. The ⟨*boolean*⟩ is true for positive classes, and false for negative classes, characterized by a leading ^. The auxiliary __regex_compile_class:TFNN also checks for a leading] which has a special meaning.

```
\cs_new_protected:Npn \__regex_compile_class_normal:w
  {
    \__regex_compile_class:TFNN
      { \__regex_class:NnnnN \c_true_bool }
      { \__regex_class:NnnnN \c_false_bool }
  }
```

(*End of definition for __regex_compile_class_normal:w.*)

__regex_compile_class_catcode:w

This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting __regex_item_catcode:nT or the reverse variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```
\cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
    \fi:
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_compile_class:TFNN
      { \__regex_item_catcode:nT {#1} }
      { \__regex_item_catcode_reverse:nT {#1} }
  }
```

(*End of definition for __regex_compile_class_catcode:w.*)

__regex_compile_class:TFNN
__regex_compile_class:NN

If the first character is ^, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

```
\cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
  {
    \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
    \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ^
      {
        \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
        \__regex_compile_class:NN
      }
      {
        \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
        \__regex_compile_class:NN #3 #4
      }
  }
\cs_new_protected:Npn \__regex_compile_class:NN #1#2
  {
    \token_if_eq_charcode:NNTF #2 ]
      { \__regex_compile_raw:N #2 }
      { #1 #2 }
  }
```

(*End of definition for __regex_compile_class:TFNN and __regex_compile_class:NN.*)

__regex_compile_class_posix_test:w
__regex_compile_class_posix:NNNNw
__regex_compile_class_posix_loop:w
__regex_compile_class_posix_end:w

Here we check for a syntax such as `[:alpha:]`. We also detect `[=` and `[.` which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see `[:`, grab raw characters until hopefully reaching `:]`. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra `\__regex_item_reverse:n` for negative classes (we make sure to wrap its argument in braces otherwise `\regex_show:N` would not recognize the regex as valid).

```
\cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
  {
    \token_if_eq_meaning:NNT \__regex_compile_special:N #1
      {
        \str_case:nn { #2 }
          {
            : { \__regex_compile_class_posix:NNNNw }
            = {
                \msg_warning:nne { regex }
                  { posix-unsupported } { = }
              }
            . {
                \msg_warning:nne { regex }
                  { posix-unsupported } { . }
              }
          }
      }
    \__regex_compile_raw:N [ #1 #2
  }
\cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
  {
    \__regex_two_if_eq:NNNNTF #5 #6 \__regex_compile_special:N ^
      {
        \bool_set_false:N \l__regex_internal_bool
        \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w
      }
      {
        \bool_set_true:N \l__regex_internal_bool
        \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w #5 #6
      }
  }
\cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
  {
    \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
      { #2 \__regex_compile_class_posix_loop:w }
      { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
  }
\cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N :
      { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ] }
      { \use_ii:nn }
      {
        \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
          {
```

```
      \__regex_compile_one:n
       {
        \bool_if:NTF \l__regex_internal_bool \use:n \__regex_item_reverse:n
        { \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : } }
       }
     }
     {
      \msg_warning:nne { regex } { posix-unknown }
       { \l__regex_internal_a_tl }
      \__regex_compile_abort_tokens:e
       {
        [: \bool_if:NF \l__regex_internal_bool { ^ }
        \l__regex_internal_a_tl :]
       }
     }
   }
   {
    \msg_error:nnee { regex } { posix-missing-close }
     { [: \l__regex_internal_a_tl } { #2 #4 }
    \__regex_compile_abort_tokens:e { [: \l__regex_internal_a_tl }
    #1 #2 #3 #4
   }
 }
```

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

9.3.10 Groups and alternations

`\__regex_compile_group_begin:N`
`\__regex_compile_group_end:`

The contents of a regex group are turned into compiled code in `\l__regex_build_tl`, which ends up with items of the form `\__regex_branch:n` {⟨*concatenation*⟩}. This construction is done using `\tl_build_...` functions within a TeX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument `#1` is `\__regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```
\cs_new_protected:Npn \__regex_compile_group_begin:N #1
 {
  \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
  \__regex_mode_quit_c:
  \group_begin:
   \tl_build_begin:N \l__regex_build_tl
   \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
   \int_incr:N \l__regex_group_level_int
   \tl_build_put_right:Nn \l__regex_build_tl
    { \__regex_branch:n { \if_false: } \fi: }
 }
\cs_new_protected:Npn \__regex_compile_group_end:
 {
  \if_int_compare:w \l__regex_group_level_int > \c_zero_int
    \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
    \tl_build_end:N \l__regex_build_tl
```

```
        \exp_args:NNNe
      \group_end:
      \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \exp_after:wN \__regex_compile_quantifier:w
    \else:
      \msg_warning:nn { regex } { extra-rparen }
      \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
    \fi:
  }
```

(*End of definition for* `\__regex_compile_group_begin:N` *and* `\__regex_compile_group_end:`.)

`\__regex_compile_(:` In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a `?`, denoting special groups, and run the code for the corresponding special group.

```
\cs_new_protected:cpn { __regex_compile_(: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
      {
        \if_int_compare:w \l__regex_mode_int =
          \c__regex_catcode_in_class_mode_int
          \msg_error:nn { regex } { c-lparen-in-class }
          \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
        \else:
          \exp_after:wN \__regex_compile_lparen:w
        \fi:
      }
  }
\cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ?
      {
        \cs_if_exist_use:cF
          { __regex_compile_special_group_\token_to_str:N #4 :w }
          {
            \msg_warning:nne { regex } { special-group-unknown }
              { (? #4 }
            \__regex_compile_group_begin:N \__regex_group:nnnN
              \__regex_compile_raw:N ? #3 #4
          }
      }
      {
        \__regex_compile_group_begin:N \__regex_group:nnnN
          #1 #2 #3 #4
      }
  }
```

(*End of definition for* `\__regex_compile_(:`.)

`\__regex_compile_|:` In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```
\cs_new_protected:cpn { __regex_compile_|: }
  {
```

```
    \__regex_if_in_class:TF { \__regex_compile_raw:N | }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
      }
  }
```

(*End of definition for* `\__regex_compile_|:`.)

`\__regex_compile_):`

Within a class, parentheses are not special. Outside, close a group.

```
\cs_new_protected:cpn { __regex_compile_): }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
      { \__regex_compile_group_end: }
  }
```

(*End of definition for* `\__regex_compile_):`.)

`\__regex_compile_special_group_::w`
`\__regex_compile_special_group_|:w`

Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```
\cs_new_protected:cpn { __regex_compile_special_group_::w }
  { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
\cs_new_protected:cpn { __regex_compile_special_group_|:w }
  { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }
```

(*End of definition for* `\__regex_compile_special_group_::w` *and* `\__regex_compile_special_group_-|:w`.)

`\__regex_compile_special_group_i:w`
`\__regex_compile_special_group_-:w`

The match can be made case-insensitive by setting the option with `(?i)`; the original behaviour is restored by `(?-i)`. This is the only supported option.

```
\cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N )
      {
        \cs_set:Npn \__regex_item_equal:n
          { \__regex_item_caseless_equal:n }
        \cs_set:Npn \__regex_item_range:nn
          { \__regex_item_caseless_range:nn }
      }
      {
        \msg_warning:nne { regex } { unknown-option } { (?i #2 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N i
        #1 #2
      }
  }
\cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N i
      { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ) }
      { \use_ii:nn }
      {
        \cs_set:Npn \__regex_item_equal:n
          { \__regex_item_caseful_equal:n }
```

```
        \cs_set:Npn \__regex_item_range:nn
          { \__regex_item_caseful_range:nn }
      }
      {
        \msg_warning:nne { regex } { unknown-option } { (?-#2#4 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N -
        #1 #2 #3 #4
      }
  }
```

(*End of definition for* `\__regex_compile_special_group_i:w` *and* `\__regex_compile_special_group_-:w`.)

9.3.11 Catcodes and csnames

`\__regex_compile_/c:`
`\__regex_compile_c_test:NN`

The `\c` escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```
\cs_new_protected:cpn { __regex_compile_/c: }
  { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
\cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \int_set_eq:Nc \l__regex_catcodes_int
              { c__regex_catcode_#2_int }
            \l__regex_mode_int
              = \if_case:w \l__regex_mode_int
                  \c__regex_catcode_mode_int
                \else:
                  \c__regex_catcode_in_class_mode_int
                \fi:
            \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
          }
      }
      { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
          {
            \msg_error:nne { regex } { c-missing-category } {#2}
            #1 #2
          }
  }
```

(*End of definition for* `\__regex_compile_/c:` *and* `\__regex_compile_c_test:NN`.)

`\__regex_compile_c_C:NN`

If `\cC` is not followed by `.` or `(...)` then complain because that construction cannot match anything, except in cases like `\cC[\c{...}]`, where it has no effect.

```
\cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
```

```
        \token_if_eq_charcode:NNTF #2 .
          { \use_none:n }
          { \token_if_eq_charcode:NNF #2 ( } % )
      }
      { \use:n }
    { \msg_error:nnn { regex } { c-C-invalid } {#2} }
    #1 #2
  }
```

(*End of definition for* `\__regex_compile_c_C:NN`.)

`\__regex_compile_c_[:w`
`\__regex_compile_c_lbrack_loop:NN`
`\__regex_compile_c_lbrack_add:N`
`\__regex_compile_c_lbrack_end:`

When encountering `\c[`, the task is to collect uppercase letters representing character categories. First check for `^` which negates the list of category codes.

```
\cs_new_protected:cpn { __regex_compile_c_[:w } #1#2
  {
    \l__regex_mode_int
      = \if_case:w \l__regex_mode_int
          \c__regex_catcode_mode_int
        \else:
          \c__regex_catcode_in_class_mode_int
        \fi:
    \int_zero:N \l__regex_catcodes_int
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ^
      {
        \bool_set_false:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
      }
      {
        \bool_set_true:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \exp_args:Nc \__regex_compile_c_lbrack_add:N
              { c__regex_catcode_#2_int }
            \__regex_compile_c_lbrack_loop:NN
          }
      }
      {
        \token_if_eq_charcode:NNTF #2 ]
          { \__regex_compile_c_lbrack_end: }
      }
          {
            \msg_error:nne { regex } { c-missing-rbrack } {#2}
            \__regex_compile_c_lbrack_end:
            #1 #2
          }
  }
```

```
\cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
  {
    \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
    \else:
      \int_add:Nn \l__regex_catcodes_int {#1}
    \fi:
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_end:
  {
    \if_meaning:w \c_false_bool \l__regex_catcodes_bool
      \int_set:Nn \l__regex_catcodes_int
        { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
    \fi:
  }
```

(*End of definition for* `\__regex_compile_c_[:w` *and others.*)

`\__regex_compile_c_{:` The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting `\c`. Additionally, disable submatch tracking since groups don't escape the scope of `\c{...}`.

```
\cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
  {
    \__regex_compile:w
      \__regex_disable_submatches:
      \l__regex_mode_int
        = \if_case:w \l__regex_mode_int
            \c__regex_cs_mode_int
          \else:
            \c__regex_cs_in_class_mode_int
          \fi:
  }
```

(*End of definition for* `\__regex_compile_c_{:`*.*)

`\__regex_compile_{:` We forbid unescaped left braces inside a `\c{...}` escape because they otherwise lead to the confusing question of whether the first right brace in `\c{{}x}` should end `\c` or whether one should match braces.

```
\cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \msg_error:nnn { regex } { cu-lbrace } { c } }
      { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
  }
```

(*End of definition for* `\__regex_compile_{:`*.*)

`__regex_cs`
`\__regex_compile_}:`
`\__regex_compile_end_cs:`
`\__regex_compile_cs_aux:Nn`
`\__regex_compile_cs_aux:NNnnnN`

Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: `\c{[{}]}` matches the control sequences `\{` and `\}`. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use `\__regex_item_exact_cs:n` with an argument consisting of all possibilities separated by `\scan_stop:`.

```
\flag_new:n { __regex_cs }
```

```
\cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \__regex_compile_end_cs: }
      { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
  }
\cs_new_protected:Npn \__regex_compile_end_cs:
  {
    \__regex_compile_end:
    \flag_clear:n { __regex_cs }
    \__kernel_tl_set:Ne \l__regex_internal_a_tl
      {
        \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
        \q__regex_nil \q__regex_nil \q__regex_recursion_stop
      }
    \exp_args:Ne \__regex_compile_one:n
      {
        \flag_if_raised:nTF { __regex_cs }
          { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
          {
            \__regex_item_exact_cs:n
              { \tl_tail:N \l__regex_internal_a_tl }
          }
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
  {
    \cs_if_eq:NNTF #1 \__regex_branch:n
      {
        \scan_stop:
        \__regex_compile_cs_aux:NNnnnN #2
        \q__regex_nil \q__regex_nil \q__regex_nil
        \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
        \__regex_compile_cs_aux:Nn
      }
      {
        \__regex_quark_if_nil:NF #1 { \flag_ensure_raised:n { __regex_cs } }
        \__regex_use_none_delimit_by_q_recursion_stop:w
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
  {
    \bool_lazy_all:nTF
      {
        { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
        {#2}
        { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
        { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
        { \int_compare_p:nNn {#5} = { 0 } }
      }
      {
        \prg_replicate:nn {#4}
          { \char_generate:nn { \use_ii:nn #3 } {12} }
        \__regex_compile_cs_aux:NNnnnN
```

```
      }
      {
        \__regex_quark_if_nil:NF #1
          {
            \flag_ensure_raised:n { __regex_cs }
            \__regex_use_i_delimit_by_q_recursion_stop:nw
          }
        \__regex_use_none_delimit_by_q_recursion_stop:w
      }
  }
```

(*End of definition for `__regex_cs` and others.*)

9.3.12 Raw token lists with \u

`\__regex_compile_/u:`

The `\u` escape is invalid in classes and directly following a catcode test. Otherwise test for a following `r` (for `\ur`), and call an auxiliary responsible for finding the variable name.

```
\cs_new_protected:cpn { __regex_compile_/u: } #1#2
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N u #1 #2 }
      {
        \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N r
          { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
          { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }
      }
  }
```

(*End of definition for `\__regex_compile_/u:`.*)

`\__regex_compile_u_brace:NNN`

This enforces the presence of a left brace, then starts a loop to find the variable name.

```
\cs_new:Npn \__regex_compile_u_brace:NNN #1#2#3
  {
    \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N \c_left_brace_str
      {
        \tl_set:Nn \l__regex_internal_b_tl {#1}
        \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
        \__regex_compile_u_loop:NN
      }
      {
        \msg_error:nn { regex } { u-missing-lbrace }
        \token_if_eq_meaning:NNTF #1 \__regex_compile_ur_end:
          { \__regex_compile_raw:N u \__regex_compile_raw:N r }
          { \__regex_compile_raw:N u }
        #2 #3
      }
  }
```

(*End of definition for `\__regex_compile_u_brace:NNN`.*)

`\__regex_compile_u_loop:NN`

We collect the characters for the argument of `\u` within an `e`-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```
\cs_new:Npn \__regex_compile_u_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      { #2 \__regex_compile_u_loop:NN }
      {
        \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
          {
            \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
              { \if_false: { \fi: } \l__regex_internal_b_tl }
              {
                \if_charcode:w \c_left_brace_str #2
                  \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
                \else:
                  #2
                \fi:
                \__regex_compile_u_loop:NN
              }
          }
          {
            \if_false: { \fi: }
            \msg_error:nne { regex } { u-missing-rbrace } {#2}
            \l__regex_internal_b_tl
            #1 #2
          }
      }
  }
```

(*End of definition for `\__regex_compile_u_loop:NN`.*)

__regex_compile_ur_end:
__regex_compile_ur:n
__regex_compile_ur_aux:w

For the `\ur{...}` construction, once we have extracted the variable's name, we replace all groups by non-capturing groups in the compiled regex (passed as the argument of `\__regex_compile_ur:n`). If that has a single branch (namely `\tl_if_empty:oTF` is false) and there is no quantifier, then simply insert the contents of this branch (obtained by `\use_ii:nn`, which is expanded later). In all other cases, insert a non-capturing group and look for quantifiers to determine the number of repetition etc.

```
\cs_new_protected:Npn \__regex_compile_ur_end:
  {
    \group_begin:
      \cs_set:Npn \__regex_group:nnnN { \__regex_group_no_capture:nnnN }
      \cs_set:Npn \__regex_group_resetting:nnnN { \__regex_group_no_capture:nnnN }
      \exp_args:NNe
    \group_end:
    \__regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
  }
\cs_new_protected:Npn \__regex_compile_ur:n #1
  {
    \tl_if_empty:oTF { \__regex_compile_ur_aux:w #1 {} ? ? \q__regex_nil }
      { \__regex_compile_if_quantifier:TFw }
      { \use_i:nn }
          {
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
```

```
      \__regex_compile_quantifier:w
     }
     { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
 }
\cs_new:Npn \__regex_compile_ur_aux:w \__regex_branch:n #1#2#3 \q__regex_nil {#2}
```

(*End of definition for* `\__regex_compile_ur_end:` *,* `\__regex_compile_ur:n` *, and* `\__regex_compile_ur_aux:w`*.*)

`\__regex_compile_u_end:`
`\__regex_compile_u_payload:`

Once we have extracted the variable's name, we check for quantifiers, in which case we set up a non-capturing group with a single branch. Inside this branch (we omit it and the group if there is no quantifier), `\__regex_compile_u_payload:` puts the right tests corresponding to the contents of the variable, which we store in `\l__regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in this case, the variable is turned to a string), or not.

```
\cs_new_protected:Npn \__regex_compile_u_end:
 {
  \__regex_compile_if_quantifier:TFw
   {
    \tl_build_put_right:Nn \l__regex_build_tl
     {
      \__regex_group_no_capture:nnnN { \if_false: } \fi:
      \__regex_branch:n { \if_false: } \fi:
     }
    \__regex_compile_u_payload:
    \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
    \__regex_compile_quantifier:w
   }
   { \__regex_compile_u_payload: }
 }
\cs_new_protected:Npn \__regex_compile_u_payload:
 {
  \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
  \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
   \__regex_compile_u_not_cs:
  \else:
   \__regex_compile_u_in_cs:
  \fi:
 }
```

(*End of definition for* `\__regex_compile_u_end:` *and* `\__regex_compile_u_payload:`*.*)

`\__regex_compile_u_in_cs:`

When `\u` appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```
\cs_new_protected:Npn \__regex_compile_u_in_cs:
 {
  \__kernel_tl_gset:Ne \g__regex_internal_tl
   {
    \exp_args:No \__kernel_str_to_other_fast:n
     { \l__regex_internal_a_tl }
   }
  \tl_build_put_right:Ne \l__regex_build_tl
   {
```

```
      \tl_map_function:NN \g__regex_internal_tl
        \__regex_compile_u_in_cs_aux:n
    }
  }
\cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
  {
    \__regex_class:NnnnN \c_true_bool
      { \__regex_item_caseful_equal:n { \int_value:w `#1 } }
      { 1 } { 0 } \c_false_bool
  }
```

(*End of definition for* `\__regex_compile_u_in_cs:`.)

`\__regex_compile_u_not_cs:` In mode 0, the `\u` escape adds one state to the NFA for each token in `\l__regex_internal_a_tl`. If a given ⟨*token*⟩ is a control sequence, then insert a string comparison test, otherwise, `\__regex_item_exact:nn` which compares catcode and character code.

```
\cs_new_protected:Npn \__regex_compile_u_not_cs:
  {
    \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
      {
        \tl_build_put_right:Ne \l__regex_build_tl
          {
            \__regex_class:NnnnN \c_true_bool
              {
                \if_int_compare:w "##3 = \c_zero_int
                  \__regex_item_exact_cs:n
                    { \exp_after:wN \cs_to_str:N ##1 }
                \else:
                  \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
                \fi:
              }
              { 1 } { 0 } \c_false_bool
          }
      }
  }
```

(*End of definition for* `\__regex_compile_u_not_cs:`.)

9.3.13 Other

`\__regex_compile_/K:` The `\K` control sequence is currently the only "command", which performs some action, rather than matching something. It is allowed in the same contexts as `\b`. At the compilation stage, we leave it as a single control sequence, defined later.

```
\cs_new_protected:cpn { __regex_compile_/K: }
  {
    \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
      { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
      { \__regex_compile_raw_error:N K }
  }
```

(*End of definition for* `\__regex_compile_/K:`.)

9.3.14 Showing regexes

__regex_clean_bool:n
__regex_clean_int:n
__regex_clean_int_aux:N
__regex_clean_regex:n
__regex_clean_regex_loop:w
__regex_clean_branch:n
__regex_clean_branch_loop:n
__regex_clean_assertion:Nn
__regex_clean_class:NnnnN
__regex_clean_group:nnnN
__regex_clean_class:n
__regex_clean_class_loop:nnn
__regex_clean_exact_cs:n
__regex_clean_exact_cs:w

Before showing a regex we check that it is "clean" in the sense that it has the correct internal structure. We do this (in the implementation of `\regex_show:N` and `\regex_log:N`) by comparing it with a cleaned-up version of the same regex. Along the way we also need similar functions for other types: all `\__regex_clean_`⟨*type*⟩`:n` functions produce valid ⟨*type*⟩ tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.

```
\cs_new:Npn \__regex_clean_bool:n #1
  {
    \tl_if_single:nTF {#1}
      { \bool_if:NTF #1 \c_true_bool \c_false_bool }
      { \c_true_bool }
  }
\cs_new:Npn \__regex_clean_int:n #1
  {
    \tl_if_head_eq_meaning:nNTF {#1} -
      { - \exp_args:No \__regex_clean_int:n { \use_none:n #1 } }
      { \int_eval:n { 0 \str_map_function:nN {#1} \__regex_clean_int_aux:N } }
  }
\cs_new:Npn \__regex_clean_int_aux:N #1
  {
    \if_int_compare:w 1 < 1 #1 ~
      #1
    \else:
      \exp_after:wN \str_map_break:
    \fi:
  }
\cs_new:Npn \__regex_clean_regex:n #1
  {
    \__regex_clean_regex_loop:w #1
    \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
  }
\cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \__regex_branch:n { \__regex_clean_branch:n {#2} }
    \__regex_clean_regex_loop:w
  }
\cs_new:Npn \__regex_clean_branch:n #1
  {
    \__regex_clean_branch_loop:n #1
    ? ? ? ? ? ? \prg_break_point:
  }
\cs_new:Npn \__regex_clean_branch_loop:n #1
  {
    \tl_if_single:nF {#1} { \prg_break: }
    \token_case_meaning:NnF #1
      {
        \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
        \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
        \__regex_class:NnnnN { #1 \__regex_clean_class:NnnnN }
        \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
        \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
```

```
        \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
      }
      { \prg_break: }
  }
\cs_new:Npn \__regex_clean_assertion:Nn #1#2
  {
    \__regex_clean_bool:n {#1}
    \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
    \token_case_meaning:NnTF #2
      {
        \__regex_A_test: { }
        \__regex_G_test: { }
        \__regex_Z_test: { }
        \__regex_b_test: { }
      }
      { {#2} }
      { { \__regex_A_test: } \prg_break: }
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_class:NnnnN #1#2#3#4#5
  {
    \__regex_clean_bool:n {#1}
    { \__regex_clean_class:n {#2} }
    { \int_max:nn { 0 } { \__regex_clean_int:n {#3} } }
    { \int_max:nn { -1 } { \__regex_clean_int:n {#4} } }
    \__regex_clean_bool:n {#5}
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4
  {
    { \__regex_clean_regex:n {#1} }
    { \int_max:nn { 0 } { \__regex_clean_int:n {#2} } }
    { \int_max:nn { -1 } { \__regex_clean_int:n {#3} } }
    \__regex_clean_bool:n {#4}
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_class:n #1
  { \__regex_clean_class_loop:nnn #1 ????? \prg_break_point: }
```

When cleaning a class there are many cases, including a dozen or so like `\__regex_prop_-d:` or `\__regex_posix_alpha:`. To avoid listing all of them we allow any command that starts with the 13 characters `__regex_prop_` or `__regex_posix` (handily these have the same length, except for the trailing underscore).

```
\cs_new:Npn \__regex_clean_class_loop:nnn #1#2#3
  {
    \tl_if_single:nF {#1} { \prg_break: }
    \token_case_meaning:NnTF #1
      {
        \__regex_item_cs:n { #1 { \__regex_clean_regex:n {#2} } }
        \__regex_item_exact_cs:n { #1 { \__regex_clean_exact_cs:n {#2} } }
        \__regex_item_caseful_equal:n { #1 { \__regex_clean_int:n {#2} } }
        \__regex_item_caseless_equal:n { #1 { \__regex_clean_int:n {#2} } }
        \__regex_item_reverse:n { #1 { \__regex_clean_class:n {#2} } }
      }
```

```
      { \__regex_clean_class_loop:nnn {#3} }
      {
        \token_case_meaning:NnTF #1
          {
            \__regex_item_caseful_range:nn { }
            \__regex_item_caseless_range:nn { }
            \__regex_item_exact:nn { }
          }
          {
            #1 { \__regex_clean_int:n {#2} } { \__regex_clean_int:n {#3} }
            \__regex_clean_class_loop:nnn
          }
          {
            \token_case_meaning:NnTF #1
              {
                \__regex_item_catcode:nT { }
                \__regex_item_catcode_reverse:nT { }
              }
              {
                #1 { \__regex_clean_int:n {#2} } { \__regex_clean_class:n {#3} }
                \__regex_clean_class_loop:nnn
              }
              {
                \exp_args:Nf \str_case:nnTF
                  {
                    \exp_args:Nf \str_range:nnn
                      { \cs_to_str:N #1 } { 1 } { 13 }
                  }
                  {
                    { __regex_prop_ } { }
                    { __regex_posix } { }
                  }
                  {
                    #1
                    \__regex_clean_class_loop:nnn {#2} {#3}
                  }
                  { \prg_break: }
              }
          }
      }
  }
\cs_new:Npn \__regex_clean_exact_cs:n #1
  {
    \exp_last_unbraced:Nf \use_none:n
      {
        \__regex_clean_exact_cs:w #1
        \scan_stop: \q_recursion_tail \scan_stop:
        \q_recursion_stop
      }
  }
\cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
  {
    \quark_if_recursion_tail_stop:n {#1}
    \scan_stop: \tl_to_str:n {#1}
```

```
    \__regex_clean_exact_cs:w
  }
```

(*End of definition for* `\__regex_clean_bool:n` *and others.*)

`\__regex_show:N` Within a group and within `\tl_build_begin:N` … `\tl_build_end:N` we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in `\l__regex_internal_a_tl` is then meant to be shown.

```
\cs_new_protected:Npn \__regex_show:N #1
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \cs_set_protected:Npn \__regex_branch:n
        {
          \seq_pop_right:NN \l__regex_show_prefix_seq
            \l__regex_internal_a_tl
          \__regex_show_one:n { +-branch }
          \seq_put_right:No \l__regex_show_prefix_seq
            \l__regex_internal_a_tl
          \use:n
        }
      \cs_set_protected:Npn \__regex_group:nnnN
        { \__regex_show_group_aux:nnnnN { } }
      \cs_set_protected:Npn \__regex_group_no_capture:nnnN
        { \__regex_show_group_aux:nnnnN { ~(no~capture) } }
      \cs_set_protected:Npn \__regex_group_resetting:nnnN
        { \__regex_show_group_aux:nnnnN { ~(resetting) } }
      \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
      \cs_set_protected:Npn \__regex_command_K:
        { \__regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
      \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
        {
          \__regex_show_one:n
            { \bool_if:NF ##1 { negative~ } assertion:~##2 }
        }
      \cs_set:Npn \__regex_b_test: { word~boundary }
      \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\\Z) }
      \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\\A) }
      \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\\G) }
      \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
        { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
      \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
        {
          \__regex_show_one:n
            { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
        }
      \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
        { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
      \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
        {
          \__regex_show_one:n
            { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
        }
      \cs_set_protected:Npn \__regex_item_catcode:nT
```

```
        { \__regex_show_item_catcode:NnT \c_true_bool }
      \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
        { \__regex_show_item_catcode:NnT \c_false_bool }
      \cs_set_protected:Npn \__regex_item_reverse:n
        { \__regex_show_scope:nn { Reversed~match } }
      \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
        { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
      \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
      \cs_set_protected:Npn \__regex_item_cs:n
        { \__regex_show_scope:nn { control~sequence } }
      \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
      \seq_clear:N \l__regex_show_prefix_seq
      \__regex_show_push:n { ~ }
      \cs_if_exist_use:N #1
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNNo
    \group_end:
    \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
  }
```

(*End of definition for* `\__regex_show:N`.)

`\__regex_show_char:n`

Show a single character, together with its ascii representation if available. This could be extended to beyond ascii. It is not ideal for parentheses themselves.

```
\cs_new:Npn \__regex_show_char:n #1
  {
    \int_eval:n {#1}
    \int_compare:nT { 32 <= #1 <= 126 }
      { ~ ( \char_generate:nn {#1} {12} ) }
  }
```

(*End of definition for* `\__regex_show_char:n`.)

`\__regex_show_one:n`

Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```
\cs_new_protected:Npn \__regex_show_one:n #1
  {
    \int_incr:N \l__regex_show_lines_int
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \exp_not:N \iow_newline:
        \seq_map_function:NN \l__regex_show_prefix_seq \use:n
        #1
      }
  }
```

(*End of definition for* `\__regex_show_one:n`.)

`\__regex_show_push:n`
`\__regex_show_pop:`
`\__regex_show_scope:nn`

Enter and exit levels of nesting. The `scope` function prints its first argument as an "introduction", then performs its second argument in a deeper level of nesting.

```
\cs_new_protected:Npn \__regex_show_push:n #1
  { \seq_put_right:Ne \l__regex_show_prefix_seq { #1 ~ } }
\cs_new_protected:Npn \__regex_show_pop:
  { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
```

```
\cs_new_protected:Npn \__regex_show_scope:nn #1#2
  {
    \__regex_show_one:n {#1}
    \__regex_show_push:n { ~ }
    #2
    \__regex_show_pop:
  }
```

(*End of definition for* `\__regex_show_push:n` *,* `\__regex_show_pop:` *, and* `\__regex_show_scope:nn`*.*)

`\__regex_show_group_aux:nnnnN`

We display all groups in the same way, simply adding a message, `(no capture)` or `(resetting)`, to special groups. The odd `\use_ii:nn` avoids printing a spurious `+-branch` for the first branch.

```
\cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
  {
    \__regex_show_one:n { ,-group~begin #1 }
    \__regex_show_push:n { | }
    \use_ii:nn #2
    \__regex_show_pop:
    \__regex_show_one:n
      { `-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
  }
```

(*End of definition for* `\__regex_show_group_aux:nnnnN`*.*)

`\__regex_show_class:NnnnN`

I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write `Match` or `Don't match` on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```
\cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_show_lines_int
      \__regex_show_push:n {~}
      #2
    \int_compare:nTF { \l__regex_show_lines_int = 0 }
      {
        \group_end:
        \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
      }
      {
        \bool_if:nTF
          { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
          {
            \group_end:
            #2
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_msg_repeated:nnN {#3} {#4} #5 }
          }
          {
              \tl_build_end:N \l__regex_build_tl
```

```
            \exp_args:NNNo
          \group_end:
          \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
          \__regex_show_one:n
            {
              \bool_if:NTF #1 { Match } { Don't~match }
              \__regex_msg_repeated:nnN {#3} {#4} #5
            }
          \tl_build_put_right:Ne \l__regex_build_tl
            { \exp_not:o \l__regex_internal_a_tl }
        }
    }
}
```

(*End of definition for* `\__regex_show_class:NnnnN`*.*)

`\__regex_show_item_catcode:NnT` Produce a sequence of categories which the catcode bitmap `#2` contains, and show it, indenting the tests on which this catcode constraint applies.

```
\cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
  {
    \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
    \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
      { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
    \__regex_show_scope:nn
      {
        categories~
        \seq_map_function:NN \l__regex_internal_seq \use:n
        , ~
        \bool_if:NF #1 { negative~ } class
      }
  }
```

(*End of definition for* `\__regex_show_item_catcode:NnT`*.*)

`\__regex_show_item_exact_cs:n`

```
\cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
  {
    \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
    \seq_set_map_e:NNn \l__regex_internal_seq
      \l__regex_internal_seq { \iow_char:N\\##1 }
    \__regex_show_one:n
      { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
  }
```

(*End of definition for* `\__regex_show_item_exact_cs:n`*.*)

9.4 Building

9.4.1 Variables used while building

`\l__regex_min_state_int` `\l__regex_max_state_int` The last state that was allocated is `\l__regex_max_state_int` -1, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in `\c{...}` constructions.

```
\int_new:N  \l__regex_min_state_int
```

```
\int_set:Nn \l__regex_min_state_int { 1 }
\int_new:N  \l__regex_max_state_int
```

(*End of definition for* `\l__regex_min_state_int` *and* `\l__regex_max_state_int`*.*)

`\l__regex_left_state_int`
`\l__regex_right_state_int`
`\l__regex_left_state_seq`
`\l__regex_right_state_seq`

Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```
\int_new:N  \l__regex_left_state_int
\int_new:N  \l__regex_right_state_int
\seq_new:N  \l__regex_left_state_seq
\seq_new:N  \l__regex_right_state_seq
```

(*End of definition for* `\l__regex_left_state_int` *and others.*)

`\l__regex_capturing_group_int`

`\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```
\int_new:N  \l__regex_capturing_group_int
```

(*End of definition for* `\l__regex_capturing_group_int`*.*)

9.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `\__regex_action_start_wildcard:N` ⟨*boolean*⟩ inserted at the start of the regular expression, where a `true` ⟨*boolean*⟩ makes it unanchored.
- `\__regex_action_success:` marks the exit state of the NFA.
- `\__regex_action_cost:n {`⟨*shift*⟩`}` is a transition from the current ⟨*state*⟩ to ⟨*state*⟩ + ⟨*shift*⟩, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `\__regex_action_free:n {`⟨*shift*⟩`}`, and `\__regex_action_free_group:n {`⟨*shift*⟩`}` are free transitions, which immediately perform the actions for the state ⟨*state*⟩ + ⟨*shift*⟩ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `\__regex_action_submatch:nN {`⟨*group*⟩`}` ⟨*key*⟩ where the ⟨*key*⟩ is `<` or `>` for the beginning or end of group numbered ⟨*group*⟩. This causes the current position in the query to be stored as the ⟨*key*⟩ submatch boundary.
- One of these actions, within a conditional.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group` − 1, and if a group opened now it would be labelled `capturing_group`.

- The last allocated state is `max_state` -1, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

__regex_build:n
__regex_build_aux:Nn
__regex_build:N
__regex_build_aux:NN

The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful. A `false` boolean for argument `#1` for the auxiliaries will suppress the wildcard and make the match anchored: used for `\peek_regex:nTF` and similar.

```
\cs_new_protected:Npn \__regex_build:n
  { \__regex_build_aux:Nn \c_true_bool }
\cs_new_protected:Npn \__regex_build:N
  { \__regex_build_aux:NN \c_true_bool }
\cs_new_protected:Npn \__regex_build_aux:Nn #1#2
  {
    \__regex_compile:n {#2}
    \__regex_build_aux:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_build_aux:NN #1#2
  {
    \__regex_standard_escapechar:
    \int_zero:N \l__regex_capturing_group_int
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard:N #1 }
    \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      { \__regex_action_success: }
  }
```

(*End of definition for* `\__regex_build:n` *and others.*)

\g__regex_case_int

Case number that was successfully matched in `\regex_match_case:nn` and related functions.

```
\int_new:N \g__regex_case_int
```

(*End of definition for* `\g__regex_case_int`.)

\l__regex_case_max_group_int

The largest group number appearing in any of the ⟨*regex*⟩ in the argument of `\regex_match_case:nn` and related functions.

```
\int_new:N \l__regex_case_max_group_int
```

(*End of definition for* `\l__regex_case_max_group_int`.)

__regex_case_build:n
__regex_case_build:e
__regex_case_build_aux:Nn
__regex_case_build_loop:n

See `\__regex_build:n`, but with a loop.

```
\cs_new_protected:Npn \__regex_case_build:n #1
  {
    \__regex_case_build_aux:Nn \c_true_bool {#1}
    \int_gzero:N \g__regex_case_int
  }
\cs_generate_variant:Nn \__regex_case_build:n { e }
\cs_new_protected:Npn \__regex_case_build_aux:Nn #1#2
  {
    \__regex_standard_escapechar:
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard:N #1 }
    %
    \__regex_build_new_state:
    \__regex_toks_put_left:Ne \l__regex_left_state_int
      { \__regex_action_submatch:nN { 0 } < }
    \__regex_push_lr_states:
    \int_zero:N \l__regex_case_max_group_int
    \int_gzero:N \g__regex_case_int
    \tl_map_inline:nn {#2}
      {
        \int_gincr:N \g__regex_case_int
        \__regex_case_build_loop:n {##1}
      }
    \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
    \__regex_pop_lr_states:
  }
\cs_new_protected:Npn \__regex_case_build_loop:n #1
  {
    \int_set:Nn \l__regex_capturing_group_int { 1 }
    \__regex_compile_use:n {#1}
    \int_set:Nn \l__regex_case_max_group_int
      {
        \int_max:nn { \l__regex_case_max_group_int }
          { \l__regex_capturing_group_int }
      }
    \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
    \__regex_toks_put_left:Ne \l__regex_right_state_int
      {
        \__regex_action_submatch:nN { 0 } >
        \int_gset:Nn \g__regex_case_int
          { \int_use:N \g__regex_case_int }
        \__regex_action_success:
      }
    \__regex_toks_clear:N \l__regex_max_state_int
    \seq_push:No \l__regex_right_state_seq
      { \int_use:N \l__regex_max_state_int }
    \int_incr:N \l__regex_max_state_int
  }
```

(*End of definition for* `\__regex_case_build:n`*,* `\__regex_case_build_aux:Nn`*, and* `\__regex_case_-build_loop:n`*.*)

`\__regex_build_for_cs:n`

The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g__regex_state_active_intarray` from `\l__regex_min_state_int` to `\l__regex_max_state_` 1;

Here, in this nested call to the matching code, we need the new versions of this range to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l__regex_min_state_int` to (the old) `\l__regex_max_state_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```
\cs_new_protected:Npn \__regex_build_for_cs:n #1
  {
    \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_push_lr_states:
    #1
    \__regex_pop_lr_states:
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      {
        \if_int_compare:w -2 = \l__regex_curr_char_int
          \exp_after:wN \__regex_action_success:
        \fi:
      }
  }
```

(*End of definition for* `\__regex_build_for_cs:n`*.*)

9.4.3 Helpers for building an nfa

`\__regex_push_lr_states:`
`\__regex_pop_lr_states:`

When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from TeX's grouping.

```
\cs_new_protected:Npn \__regex_push_lr_states:
  {
    \seq_push:No \l__regex_left_state_seq
      { \int_use:N \l__regex_left_state_int }
    \seq_push:No \l__regex_right_state_seq
      { \int_use:N \l__regex_right_state_int }
  }
\cs_new_protected:Npn \__regex_pop_lr_states:
  {
    \seq_pop:NN \l__regex_left_state_seq  \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int  \l__regex_internal_a_tl
    \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*End of definition for* `\__regex_push_lr_states:` *and* `\__regex_pop_lr_states:`.)

`\__regex_build_transition_left:NNN`
`\__regex_build_transition_right:nNn`

Add a transition from `#2` to `#3` using the function `#1`. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```
\cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
  { \__regex_toks_put_left:Ne  #2 { #1 { \int_eval:n { #3 - #2 } } } }
\cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
  { \__regex_toks_put_right:Ne #2 { #1 { \int_eval:n { #3 - #2 } } } }
```

(*End of definition for* `\__regex_build_transition_left:NNN` *and* `\__regex_build_transition_right:nNn`.)

`\__regex_build_new_state:`

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously "current" state.

```
\cs_new_protected:Npn \__regex_build_new_state:
  {
    \__regex_toks_clear:N \l__regex_max_state_int
    \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
    \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
    \int_incr:N \l__regex_max_state_int
  }
```

(*End of definition for* `\__regex_build_new_state:`.)

`\__regex_build_transitions_lazyness:NNNNN`

This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by `#1`, true for lazy quantifiers, and false for greedy quantifiers.

```
\cs_new_protected:Npn \__regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \if_meaning:w \c_true_bool #1
          #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
          #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
        \else:
          #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
          #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
        \fi:
      }
  }
```

(*End of definition for* `\__regex_build_transitions_lazyness:NNNNN`.)

9.4.4 Building classes

`\__regex_class:NnnnN`
`\__regex_tests_action_cost:n`

The arguments are: ⟨*boolean*⟩ `{`⟨*tests*⟩`}` `{`⟨*min*⟩`}` `{`⟨*more*⟩`}` ⟨*lazyness*⟩. First store the tests with a trailing `\__regex_action_cost:n`, in the true branch of `\__regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer ⟨*more*⟩ is 0 for fixed repetitions, -1 for unbounded repetitions, and ⟨*max*⟩ $-$ ⟨*min*⟩ for a range of repetitions.

```
\cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
  {
    \cs_set:Npe \__regex_tests_action_cost:n ##1
      {
        \exp_not:n { \exp_not:n {#2} }
        \bool_if:NTF #1
          { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
          { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
      }
    \if_case:w - #4 \exp_stop_f:
           \__regex_class_repeat:n   {#3}
    \or:   \__regex_class_repeat:nN  {#3}      #5
    \else: \__regex_class_repeat:nnN {#3} {#4} #5
    \fi:
  }
\cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }
```

(*End of definition for* `\__regex_class:NnnnN` *and* `\__regex_tests_action_cost:n`.)

`\__regex_class_repeat:n`

This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for `#1` $= 0$ repetitions: nothing is built.

```
\cs_new_protected:Npn \__regex_class_repeat:n #1
  {
    \prg_replicate:nn {#1}
      {
        \__regex_build_new_state:
        \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
          \l__regex_left_state_int \l__regex_right_state_int
      }
  }
```

(*End of definition for* `\__regex_class_repeat:n`.)

`\__regex_class_repeat:nN`

This implements unbounded repetitions of a single class (*e.g.* the `*` and `+` quantifiers). If the minimum number `#1` of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call `\__regex_class_repeat:n` for the code to match `#1` repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean `#2`.

```
\cs_new_protected:Npn \__regex_class_repeat:nN #1#2
  {
    \if_int_compare:w #1 = \c_zero_int
      \__regex_build_transitions_lazyness:NNNNN #2
        \__regex_action_free:n       \l__regex_right_state_int
        \__regex_tests_action_cost:n \l__regex_left_state_int
    \else:
      \__regex_class_repeat:n {#1}
      \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
      \__regex_build_transitions_lazyness:NNNNN #2
        \__regex_action_free:n \l__regex_right_state_int
        \__regex_action_free:n \l__regex_internal_a_int
    \fi:
  }
```

(*End of definition for* `\__regex_class_repeat:nN`.)

`\__regex_class_repeat:nnN`

We want to build the code to match from `#1` to `#1` + `#2` repetitions. Match `#1` repetitions (can be 0). Compute the final state of the next construction as `a`. Build `#2` > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state `a`. The computation of `a` is safe because states are allocated in order, starting from `max_state`.

```
\cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
  {
    \__regex_class_repeat:n {#1}
    \int_set:Nn \l__regex_internal_a_int
      { \l__regex_max_state_int + #2 - 1 }
    \prg_replicate:nn { #2 }
      {
        \__regex_build_transitions_lazyness:NNNNN #3
          \__regex_action_free:n       \l__regex_internal_a_int
          \__regex_tests_action_cost:n \l__regex_right_state_int
      }
  }
```

(*End of definition for* `\__regex_class_repeat:nnN`.)

9.4.5 Building groups

`\__regex_group_aux:nnnnN`

Arguments: {⟨*label*⟩} {⟨*contents*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩. If ⟨*min*⟩ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents `#2` of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly `#3` times, or `#3` or more times, or between `#3` and `#3` + `#4` times, with lazyness `#5`. The ⟨*label*⟩ `#1` is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```
\cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
  {
      \if_int_compare:w #3 = \c_zero_int
        \__regex_build_new_state:
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_left_state_int \l__regex_right_state_int
      \fi:
      \__regex_build_new_state:
      \__regex_push_lr_states:
      #2
      \__regex_pop_lr_states:
      \if_case:w - #4 \exp_stop_f:
             \__regex_group_repeat:nn   {#1} {#3}
      \or:   \__regex_group_repeat:nnN  {#1} {#3}      #5
      \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
      \fi:
  }
```

(*End of definition for* `\__regex_group_aux:nnnnN`.)

__regex_group:nnnN
__regex_group_no_capture:nnnN

Hand to `\__regex_group_aux:nnnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```
\cs_new_protected:Npn \__regex_group:nnnN #1
  {
    \exp_args:No \__regex_group_aux:nnnnN
      { \int_use:N \l__regex_capturing_group_int }
      {
        \int_incr:N \l__regex_capturing_group_int
        #1
      }
  }
\cs_new_protected:Npn \__regex_group_no_capture:nnnN
  { \__regex_group_aux:nnnnN { -1 } }
```

(*End of definition for* `\__regex_group:nnnN` *and* `\__regex_group_no_capture:nnnN`.)

__regex_group_resetting:nnnN
__regex_group_resetting_loop:nnNn

Again, hand the label -1 to `\__regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `\__regex_branch:n` {⟨*branch*⟩}.

```
\cs_new_protected:Npn \__regex_group_resetting:nnnN #1
  {
    \__regex_group_aux:nnnnN { -1 }
      {
        \exp_args:Noo \__regex_group_resetting_loop:nnNn
          { \int_use:N \l__regex_capturing_group_int }
          { \int_use:N \l__regex_capturing_group_int }
          #1
          { ?? \prg_break:n } { }
        \prg_break_point:
      }
  }
\cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
  {
    \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
    \int_set:Nn \l__regex_capturing_group_int {#2}
    #3 {#4}
    \exp_args:Nf \__regex_group_resetting_loop:nnNn
      { \int_max:nn {#1} { \l__regex_capturing_group_int } }
      {#2}
  }
```

(*End of definition for* `\__regex_group_resetting:nnnN` *and* `\__regex_group_resetting_loop:nnNn`.)

__regex_branch:n

Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```
\cs_new_protected:Npn \__regex_branch:n #1
  {
    \__regex_build_new_state:
    \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
```

```
      \l__regex_left_state_int \l__regex_right_state_int
    #1
    \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*End of definition for* `\__regex_branch:n`.)

`\__regex_group_repeat:nn`

This function is called to repeat a group a fixed number of times `#2`; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label). Otherwise, the auxiliary `\__regex_group_repeat_aux:n` copies `#2` times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```
\cs_new_protected:Npn \__regex_group_repeat:nn #1#2
  {
    \if_int_compare:w #2 = \c_zero_int
      \int_set:Nn \l__regex_max_state_int
        { \l__regex_left_state_int - 1 }
      \__regex_build_new_state:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \__regex_build_new_state:
    \fi:
  }
```

(*End of definition for* `\__regex_group_repeat:nn`.)

`\__regex_group_submatches:nNN`

This inserts in states `#2` and `#3` the code for tracking submatches of the group `#1`, unless inhibited by a label of -1.

```
\cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
  {
    \if_int_compare:w #1 > - \c_one_int
      \__regex_toks_put_left:Ne #2 { \__regex_action_submatch:nN {#1} < }
      \__regex_toks_put_left:Ne #3 { \__regex_action_submatch:nN {#1} > }
    \fi:
  }
```

(*End of definition for* `\__regex_group_submatches:nNN`.)

`\__regex_group_repeat_aux:n`

Here we repeat `\toks` ranging from `left_state` to `max_state`, `#1` >0 times. First add a transition so that the copies "chain" properly. Compute the shift `c` between the original copy and the last copy we want. Shift the `right_state` and `max_state` to their final values. We then want to perform `c` copy operations. At the end, `b` is equal to the `max_state`, and `a` points to the left of the last copy of the group.

```
\cs_new_protected:Npn \__regex_group_repeat_aux:n #1
  {
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_max_state_int
    \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
    \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
```

```
    \if_int_compare:w \int_eval:n {#1} > \c_one_int
      \int_set:Nn \l__regex_internal_c_int
        {
          ( #1 - 1 )
          * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
        }
      \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
      \int_add:Nn \l__regex_max_state_int   { \l__regex_internal_c_int }
      \__regex_toks_memcpy:NNn
        \l__regex_internal_b_int
        \l__regex_internal_a_int
        \l__regex_internal_c_int
    \fi:
  }
```

(*End of definition for* `\__regex_group_repeat_aux:n`.)

`\__regex_group_repeat:nnN`

This function is called to repeat a group at least n times; the case $n = 0$ is very different from $n > 0$. Assume first that $n = 0$. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the "true" left state `a` (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `\__regex_group_repeat_aux:n`.

```
\cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
  {
    \if_int_compare:w #2 = \c_zero_int
      \__regex_group_submatches:nNN {#1}
        \l__regex_left_state_int \l__regex_right_state_int
      \int_set:Nn \l__regex_internal_a_int
        { \l__regex_left_state_int - 1 }
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_internal_a_int
      \__regex_build_new_state:
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_left:NNN \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \else:
        \__regex_build_transition_right:nNn \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \fi:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
```

```
      \else:
        \__regex_build_transition_left:NNN \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \fi:
      \__regex_build_new_state:
    \fi:
  }
```

(*End of definition for* `\__regex_group_repeat:nnN`*.*)

`\__regex_group_repeat:nnnN`

We wish to repeat the group between #2 and #2 + #3 times, with a lazyness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it "by hand" earlier.

```
\cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
  {
    \__regex_group_submatches:nNN {#1}
      \l__regex_left_state_int \l__regex_right_state_int
    \__regex_group_repeat_aux:n { #2 + #3 }
    \if_meaning:w \c_true_bool #4
      \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
      \prg_replicate:nn { #3 }
        {
          \int_sub:Nn \l__regex_left_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_left:NNN \__regex_action_free:n
            \l__regex_left_state_int \l__regex_max_state_int
        }
    \else:
      \prg_replicate:nn { #3 - 1 }
        {
          \int_sub:Nn \l__regex_right_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_right:nNn \__regex_action_free:n
            \l__regex_right_state_int \l__regex_max_state_int
        }
      \if_int_compare:w #2 = \c_zero_int
        \int_set:Nn \l__regex_right_state_int
          { \l__regex_left_state_int - 1 }
      \else:
        \int_sub:Nn \l__regex_right_state_int
          { \l__regex_internal_b_int - \l__regex_internal_a_int }
      \fi:
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_max_state_int
    \fi:
```

```
    \__regex_build_new_state:
  }
```

(*End of definition for* `\__regex_group_repeat:nnnN`.)

9.4.6 Others

`\__regex_assertion:Nn`
`\__regex_b_test:`
`\__regex_A_test:`
`\__regex_G_test:`
`\__regex_Z_test:`

Usage: `\__regex_assertion:Nn` ⟨*boolean*⟩ `{`⟨*test*⟩`}`, where the ⟨*test*⟩ is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The `\__regex_b_test:` test is used by the `\b` and `\B` escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose.

```
\cs_new_protected:Npn \__regex_assertion:Nn #1#2
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \exp_not:n {#2}
        \__regex_break_point:TF
          \bool_if:NF #1 { { } }
          {
            \__regex_action_free:n
              {
                \int_eval:n
                  { \l__regex_right_state_int - \l__regex_left_state_int }
              }
          }
          \bool_if:NT #1 { { } }
      }
  }
\cs_new_protected:Npn \__regex_b_test:
  {
    \group_begin:
      \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
      \__regex_prop_w:
      \__regex_break_point:TF
        { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
        { \group_end: \__regex_prop_w: }
  }
\cs_new_protected:Npn \__regex_Z_test:
  {
    \if_int_compare:w -2 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_A_test:
  {
    \if_int_compare:w -2 = \l__regex_last_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_G_test:
  {
    \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
```

```
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*End of definition for* `\__regex_assertion:Nn` *and others.*)

`\__regex_command_K:` Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```
\cs_new_protected:Npn \__regex_command_K:
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \__regex_action_submatch:nN { 0 } <
        \bool_set_true:N \l__regex_fresh_thread_bool
        \__regex_action_free:n
          {
            \int_eval:n
              { \l__regex_right_state_int - \l__regex_left_state_int }
          }
        \bool_set_false:N \l__regex_fresh_thread_bool
      }
  }
```

(*End of definition for* `\__regex_command_K:`.)

9.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains "free" transitions to other states, and transitions which "consume" the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of "active states", stored in `\g__regex_thread_info_intarray` (together with submatch information): this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA "collide" in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn't it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing "normal" free transitions `\__regex_action_free:n` from transitions `\__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a "completed" thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

9.5.1 Variables used when matching

`\l__regex_min_pos_int`
`\l__regex_max_pos_int`
`\l__regex_curr_pos_int`
`\l__regex_start_pos_int`
`\l__regex_success_pos_int`

The tokens in the query are indexed from `min_pos` for the first to `max_pos`-1 for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We match without backtracking, keeping all threads in lockstep at the `curr_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```
\int_new:N \l__regex_min_pos_int
\int_new:N \l__regex_max_pos_int
\int_new:N \l__regex_curr_pos_int
\int_new:N \l__regex_start_pos_int
\int_new:N \l__regex_success_pos_int
```

(*End of definition for* `\l__regex_min_pos_int` *and others.*)

`\l__regex_curr_char_int`
`\l__regex_curr_catcode_int`
`\l__regex_curr_token_tl`
`\l__regex_last_char_int`
`\l__regex_last_char_success_int`
`\l__regex_case_changed_char_int`

The character and category codes of the token at the current position and a token list expanding to that token; the character code of the token at the previous position; the character code of the token just before a successful match; and the character code of the result of changing the case of the current token (`A-Z`↔`a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `curr_char` variable is also used in various other phases to hold a character code.

```
\int_new:N \l__regex_curr_char_int
\int_new:N \l__regex_curr_catcode_int
\tl_new:N \l__regex_curr_token_tl
\int_new:N \l__regex_last_char_int
\int_new:N \l__regex_last_char_success_int
\int_new:N \l__regex_case_changed_char_int
```

(*End of definition for* `\l__regex_curr_char_int` *and others.*)

`\l__regex_curr_state_int`

For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
\int_new:N \l__regex_curr_state_int
```

(*End of definition for* `\l__regex_curr_state_int`*.*)

`\l__regex_curr_submatches_tl`
`\l__regex_success_submatches_tl`

The submatches for the thread which is currently active are stored in the `curr_submatches` list, which is almost a comma list, but ends with a comma. This list is stored by `\__regex_store_state:n` into an intarray variable, to be retrieved when matching at the next position. When a thread succeeds, this list is copied to `\l__regex_success_submatches_tl`: only the last successful thread remains there.

```
\tl_new:N \l__regex_curr_submatches_tl
\tl_new:N \l__regex_success_submatches_tl
```

(*End of definition for* `\l__regex_curr_submatches_tl` *and* `\l__regex_success_submatches_tl`*.*)

`\l__regex_step_int`

This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each ⟨*state*⟩ in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks`⟨*state*⟩, but not finished yet. However, once we finish, we store `step` + 1 in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase). The `step` is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
\int_new:N \l__regex_step_int
```

(*End of definition for* `\l__regex_step_int`.)

`\l__regex_min_thread_int`
`\l__regex_max_thread_int`

All the currently active threads are kept in order of precedence in `\g__regex_thread_info_intarray` together with the corresponding submatch information. Data in this intarray is organized as blocks from `min_thread` (included) to `max_thread` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_thread` is reset to `min_thread`, effectively clearing the array.

```
\int_new:N \l__regex_min_thread_int
\int_new:N \l__regex_max_thread_int
```

(*End of definition for* `\l__regex_min_thread_int` *and* `\l__regex_max_thread_int`.)

`\g__regex_state_active_intarray`
`\g__regex_thread_info_intarray`

`\g__regex_state_active_intarray` stores the last ⟨*step*⟩ in which each ⟨*state*⟩ was active. `\g__regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
\intarray_new:Nn \g__regex_state_active_intarray { 65536 }
\intarray_new:Nn \g__regex_thread_info_intarray { 65536 }
```

(*End of definition for* `\g__regex_state_active_intarray` *and* `\g__regex_thread_info_intarray`.)

`\l__regex_matched_analysis_tl`
`\l__regex_curr_analysis_tl`

The list `\l__regex_curr_analysis_tl` consists of a brace group containing three brace groups corresponding to the current token, with the same syntax as `\tl_analysis_map_inline:nn`. The list `\l__regex_matched_analysis_tl` (constructed under the `tl_build` machinery) has one item for each token that has already been treated so far in a given match attempt: each item consists of three brace groups with the same syntax as `\tl_analysis_map_inline:nn`.

```
\tl_new:N \l__regex_matched_analysis_tl
\tl_new:N \l__regex_curr_analysis_tl
```

(*End of definition for* `\l__regex_matched_analysis_tl` *and* `\l__regex_curr_analysis_tl`.)

`\l__regex_every_match_tl`

Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `\__regex_single_match:` and `\__regex_multi_match:n`.

```
\tl_new:N \l__regex_every_match_tl
```

(*End of definition for* `\l__regex_every_match_tl`.)

\l__regex_fresh_thread_bool
\l__regex_empty_success_bool
__regex_if_two_empty_matches:F

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `\__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
\bool_new:N \l__regex_fresh_thread_bool
\bool_new:N \l__regex_empty_success_bool
\cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(*End of definition for* `\l__regex_fresh_thread_bool`*,* `\l__regex_empty_success_bool`*, and* `\__regex_if_two_empty_matches:F`*.*)

\g__regex_success_bool
\l__regex_saved_success_bool
\l__regex_match_success_bool

The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
\bool_new:N \g__regex_success_bool
\bool_new:N \l__regex_saved_success_bool
\bool_new:N \l__regex_match_success_bool
```

(*End of definition for* `\g__regex_success_bool`*,* `\l__regex_saved_success_bool`*, and* `\l__regex_match_success_bool`*.*)

9.5.2 Matching: framework

__regex_match:n
__regex_match_cs:n
__regex_match_init:

Initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```
\cs_new_protected:Npn \__regex_match:n #1
  {
    \__regex_match_init:
    \__regex_match_once_init:
    \tl_analysis_map_inline:nn {#1}
      { \__regex_match_one_token:nnN {##1} {##2} ##3 }
    \__regex_match_one_token:nnN { } { -2 } F
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_cs:n #1
  {
    \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
    \__regex_match_init:
    \__regex_match_once_init:
    \str_map_inline:nn {#1}
```

```
      {
        \tl_if_blank:nTF {##1}
          { \__regex_match_one_token:nnN {##1} {`##1} A }
          { \__regex_match_one_token:nnN {##1} {`##1} C }
      }
    \__regex_match_one_token:nnN { } { -2 } F
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_init:
  {
    \bool_gset_false:N \g__regex_success_bool
    \int_step_inline:nnn
      \l__regex_min_state_int { \l__regex_max_state_int - 1 }
      {
        \__kernel_intarray_gset:Nnn
          \g__regex_state_active_intarray {##1} { 1 }
      }
    \int_zero:N \l__regex_step_int
    \int_set:Nn \l__regex_min_pos_int { 2 }
    \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
    \int_set:Nn \l__regex_last_char_success_int { -2 }
    \tl_build_begin:N \l__regex_matched_analysis_tl
    \tl_clear:N \l__regex_curr_analysis_tl
    \int_set:Nn \l__regex_min_submatch_int { 1 }
    \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
    \bool_set_false:N \l__regex_empty_success_bool
  }
```

(*End of definition for* `\__regex_match:n`, `\__regex_match_cs:n`, *and* `\__regex_match_init:`.)

`\__regex_match_once_init:` This function resets various variables used when finding one match. It is called before the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start because `\__regex_match_one_token:nnN` increments `\l__regex_curr_pos_int` and saves `\l__regex_curr_char_int` as the `last_char` so that word boundaries can be correctly identified.

```
\cs_new_protected:Npn \__regex_match_once_init:
  {
    \if_meaning:w \c_true_bool \l__regex_empty_success_bool
      \cs_set:Npn \__regex_if_two_empty_matches:F
        {
          \int_compare:nNnF
            \l__regex_start_pos_int = \l__regex_curr_pos_int
        }
    \else:
      \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
    \fi:
    \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
    \bool_set_false:N \l__regex_match_success_bool
    \tl_set:Ne \l__regex_curr_submatches_tl
```

```
      { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
    \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
    \__regex_store_state:n { \l__regex_min_state_int }
    \int_set:Nn \l__regex_curr_pos_int
      { \l__regex_start_pos_int - 1 }
    \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
    \tl_build_get_intermediate:NN \l__regex_matched_analysis_tl \l__regex_internal_a_tl
    \exp_args:NNf \__regex_match_once_init_aux:
    \tl_map_inline:nn
      { \exp_after:wN \l__regex_internal_a_tl \l__regex_curr_analysis_tl }
      { \__regex_match_one_token:nnN ##1 }
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_once_init_aux:
  {
    \tl_build_begin:N \l__regex_matched_analysis_tl
    \tl_clear:N \l__regex_curr_analysis_tl
  }
```

(*End of definition for* `\__regex_match_once_init:`.)

`\__regex_single_match:`
`\__regex_multi_match:n`

For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action `#1`, then find the next match.

```
\cs_new_protected:Npn \__regex_single_match:
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \bool_gset_eq:NN
          \g__regex_success_bool
          \l__regex_match_success_bool
        \__regex_maplike_break:
      }
  }
\cs_new_protected:Npn \__regex_multi_match:n #1
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \if_meaning:w \c_false_bool \l__regex_match_success_bool
          \exp_after:wN \__regex_maplike_break:
        \fi:
        \bool_gset_true:N \g__regex_success_bool
        #1
        \__regex_match_once_init:
      }
  }
```

(*End of definition for* `\__regex_single_match:` *and* `\__regex_multi_match:n`.)

`\__regex_match_one_token:nnN`
`\__regex_match_one_active:n`

At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_thread`). This results in a sequence of `\__regex_use_state_and_submatches:w` ⟨*state*⟩`,`⟨*submatch-clist*⟩`;` and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is the match. We explain the `fresh_thread` business when describing `\__regex_action_wildcard:`.

```
\cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
  {
    \int_add:Nn \l__regex_step_int { 2 }
    \int_incr:N \l__regex_curr_pos_int
    \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
    \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
    \tl_set:Nn \l__regex_curr_token_tl {#1}
    \int_set:Nn \l__regex_curr_char_int {#2}
    \int_set:Nn \l__regex_curr_catcode_int { "#3 }
    \tl_build_put_right:Ne \l__regex_matched_analysis_tl
      { \exp_not:o \l__regex_curr_analysis_tl }
    \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
    \use:e
      {
        \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
        \int_step_function:nnN
          { \l__regex_min_thread_int }
          { \l__regex_max_thread_int - 1 }
          \__regex_match_one_active:n
      }
    \prg_break_point:
    \bool_set_false:N \l__regex_fresh_thread_bool
    \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
      \if_int_compare:w -2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
      \fi:
    \fi:
    \l__regex_every_match_tl
  }
\cs_new:Npn \__regex_match_one_active:n #1
  {
    \__regex_use_state_and_submatches:w
    \__kernel_intarray_range_to_clist:Nnn
      \g__regex_thread_info_intarray
      { 1 + #1 * (\l__regex_capturing_group_int * 2 + 1) }
      { (1 + #1) * (\l__regex_capturing_group_int * 2 + 1) }
    ;
  }
```

(*End of definition for* `\__regex_match_one_token:nnN` *and* `\__regex_match_one_active:n`.)

9.5.3 Using states of the nfa

`\__regex_use_state:` Use the current NFA instruction. The state is initially marked as belonging to the current `step`: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as `step` + 1: any thread hitting it at that point will be terminated.

```
\cs_new_protected:Npn \__regex_use_state:
  {
    \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
```

```
      { \l__regex_curr_state_int } { \l__regex_step_int }
    \__regex_toks_use:w \l__regex_curr_state_int
    \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
      { \l__regex_curr_state_int }
      { \int_eval:n { \l__regex_step_int + 1 } }
  }
```

(*End of definition for* `\__regex_use_state:`.)

`\__regex_use_state_and_submatches:w`

This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `curr_state` and `curr_submatches` and use the state if it has not yet been encountered at this step.

```
\cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
  {
    \int_set:Nn \l__regex_curr_state_int {#1}
    \if_int_compare:w
        \__kernel_intarray_item:Nn \g__regex_state_active_intarray
          { \l__regex_curr_state_int }
                      < \l__regex_step_int
      \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
      \exp_after:wN \__regex_use_state:
    \fi:
    \scan_stop:
  }
```

(*End of definition for* `\__regex_use_state_and_submatches:w`.)

9.5.4 Actions when matching

`\__regex_action_start_wildcard:N`

For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `\__regex_match_one_token:nnN` too.

```
\cs_new_protected:Npn \__regex_action_start_wildcard:N #1
  {
    \bool_set_true:N \l__regex_fresh_thread_bool
    \__regex_action_free:n {1}
    \bool_set_false:N \l__regex_fresh_thread_bool
    \bool_if:NT #1 { \__regex_action_cost:n {0} }
  }
```

(*End of definition for* `\__regex_action_start_wildcard:N`.)

`\__regex_action_free:n`
`\__regex_action_free_group:n`
`\__regex_action_free_aux:nn`

These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l__regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the "normal" version will revisit a state even within the thread itself.

```
\cs_new_protected:Npn \__regex_action_free:n
  { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
```

```
\cs_new_protected:Npn \__regex_action_free_group:n
  { \__regex_action_free_aux:nn { < \l__regex_step_int } }
\cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
  {
    \use:e
      {
        \int_add:Nn \l__regex_curr_state_int {#2}
        \exp_not:n
          {
            \if_int_compare:w
                \__kernel_intarray_item:Nn \g__regex_state_active_intarray
                  { \l__regex_curr_state_int }
                #1
              \exp_after:wN \__regex_use_state:
            \fi:
          }
        \int_set:Nn \l__regex_curr_state_int
          { \int_use:N \l__regex_curr_state_int }
        \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
          { \exp_not:o \l__regex_curr_submatches_tl }
      }
  }
```

(*End of definition for `\__regex_action_free:n`, `\__regex_action_free_group:n`, and `\__regex_action_free_aux:nn`.*)

`\__regex_action_cost:n`

A transition which consumes the current character and shifts the state by `#1`. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```
\cs_new_protected:Npn \__regex_action_cost:n #1
  {
    \exp_args:Ne \__regex_store_state:n
      { \int_eval:n { \l__regex_curr_state_int + #1 } }
  }
```

(*End of definition for `\__regex_action_cost:n`.*)

`\__regex_store_state:n`
`\__regex_store_submatches:`

Put the given state and current submatch information in `\g__regex_thread_info_intarray`, and increment the length of the array.

```
\cs_new_protected:Npn \__regex_store_state:n #1
  {
    \exp_args:No \__regex_store_submatches:nn
      \l__regex_curr_submatches_tl {#1}
    \int_incr:N \l__regex_max_thread_int
  }
\cs_new_protected:Npn \__regex_store_submatches:nn #1#2
  {
    \__kernel_intarray_gset_range_from_clist:Nnn
      \g__regex_thread_info_intarray
      {
        \__regex_int_eval:w
        1 + \l__regex_max_thread_int *
        (\l__regex_capturing_group_int * 2 + 1)
      }
```

```
      { #2 , #1 }
  }
```

(*End of definition for* `\__regex_store_state:n` *and* `\__regex_store_submatches:`.)

`\__regex_disable_submatches:`

Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```
\cs_new_protected:Npn \__regex_disable_submatches:
  {
    \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
    \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
  }
```

(*End of definition for* `\__regex_disable_submatches:`.)

`\__regex_action_submatch:nN`
`\__regex_action_submatch_aux:w`
`\__regex_action_submatch_auxii:w`
`\__regex_action_submatch_auxiii:w`
`\__regex_action_submatch_auxiv:w`

Update the current submatches with the information from the current position. Maybe a bottleneck.

```
\cs_new_protected:Npn \__regex_action_submatch:nN #1#2
  {
    \exp_after:wN \__regex_action_submatch_aux:w
    \l__regex_curr_submatches_tl ; {#1} #2
  }
\cs_new_protected:Npn \__regex_action_submatch_aux:w #1 ; #2#3
  {
    \tl_set:Ne \l__regex_curr_submatches_tl
      {
        \prg_replicate:nn
          { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }
          { \__regex_action_submatch_auxii:w }
        \__regex_action_submatch_auxiii:w
        #1
      }
  }
\cs_new:Npn \__regex_action_submatch_auxii:w
    #1 \__regex_action_submatch_auxiii:w #2 ,
  { #2 , #1 \__regex_action_submatch_auxiii:w }
\cs_new:Npn \__regex_action_submatch_auxiii:w #1 ,
  { \int_use:N \l__regex_curr_pos_int , }
```

(*End of definition for* `\__regex_action_submatch:nN` *and others.*)

`\__regex_action_success:`

There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is "fresh"; and we store the current position and submatches. The current step is then interrupted with `\prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```
\cs_new_protected:Npn \__regex_action_success:
  {
    \__regex_if_two_empty_matches:F
      {
        \bool_set_true:N \l__regex_match_success_bool
        \bool_set_eq:NN \l__regex_empty_success_bool
```

```
        \l__regex_fresh_thread_bool
      \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
      \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
      \tl_build_begin:N \l__regex_matched_analysis_tl
      \tl_set_eq:NN \l__regex_success_submatches_tl
        \l__regex_curr_submatches_tl
      \prg_break:
    }
  }
```

(*End of definition for* `\__regex_action_success:`.)

9.6 Replacement

9.6.1 Variables and helpers used in replacement

`\l__regex_replacement_csnames_int`

The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{` has been encountered. The number of “open” such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `}`.

```
\int_new:N \l__regex_replacement_csnames_int
```

(*End of definition for* `\l__regex_replacement_csnames_int`.)

`\l__regex_replacement_category_tl`
`\l__regex_replacement_category_seq`

This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_)d)`.

```
\tl_new:N \l__regex_replacement_category_tl
\seq_new:N \l__regex_replacement_category_seq
```

(*End of definition for* `\l__regex_replacement_category_tl` *and* `\l__regex_replacement_category_seq`.)

`\g__regex_balance_tl`

This token list holds the replacement text for `\__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
\tl_new:N \g__regex_balance_tl
```

(*End of definition for* `\g__regex_balance_tl`.)

`\__regex_replacement_balance_one_match:n`

This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading `+` in the actual definition).

```
\cs_new:Npn \__regex_replacement_balance_one_match:n #1
  { - \__regex_submatch_balance:n {#1} }
```

(*End of definition for* `\__regex_replacement_balance_one_match:n`.)

__regex_replacement_do_one_match:n

The input is the same as __regex_replacement_balance_one_match:n. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
\cs_new:Npn \__regex_replacement_do_one_match:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
  }
```

(*End of definition for* __regex_replacement_do_one_match:n.)

__regex_replacement_exp_not:N

This function lets us navigate around the fact that the primitive \exp_not:n requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as \c_parameter_token. Indeed, within an e/x-expanding assignment, \exp_not:N # behaves as a single #, whereas \exp_not:n {#} behaves as a doubled ##.

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

(*End of definition for* __regex_replacement_exp_not:N.)

__regex_replacement_exp_not:V

This is used for the implementation of \u, and it gets redefined for \peek_regex_replace_once:nnTF.

```
\cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(*End of definition for* __regex_replacement_exp_not:V.)

9.6.2 Query and brace balance

__regex_query_range:nn
__regex_query_range_loop:ww

When it is time to extract submatches from the token list, the various tokens are stored in \toks registers numbered from \l__regex_min_pos_int inclusive to \l__regex_max_pos_int exclusive. The function __regex_query_range:nn {⟨*min*⟩} {⟨*max*⟩} unpacks registers from the position ⟨*min*⟩ to the position ⟨*max*⟩ $-$ 1 included. Once this is expanded, a second e-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```
\cs_new:Npn \__regex_query_range:nn #1#2
  {
    \exp_after:wN \__regex_query_range_loop:ww
    \int_value:w \__regex_int_eval:w #1 \exp_after:wN ;
    \int_value:w \__regex_int_eval:w #2 ;
    \prg_break_point:
  }
\cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
  {
    \if_int_compare:w #1 < #2 \exp_stop_f:
    \else:
      \exp_after:wN \prg_break:
    \fi:
```

```
    \__regex_toks_use:w #1 \exp_stop_f:
    \exp_after:wN \__regex_query_range_loop:ww
      \int_value:w \__regex_int_eval:w #1 + 1 ; #2 ;
  }
```

(*End of definition for* `\__regex_query_range:nn` *and* `\__regex_query_range_loop:ww`.)

`\__regex_query_submatch:n`

Find the start and end positions for a given submatch (of a given match).

```
\cs_new:Npn \__regex_query_submatch:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
  }
```

(*End of definition for* `\__regex_query_submatch:n`.)

`\__regex_submatch_balance:n`

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the ⟨*max pos*⟩ and ⟨*min pos*⟩. These two positions are found in the corresponding "submatch" arrays.

```
\cs_new_protected:Npn \__regex_submatch_balance:n #1
  {
    \int_eval:n
      {
        \__regex_intarray_item:NnF \g__regex_balance_intarray
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_end_intarray {#1}
          }
          { 0 }
        -
        \__regex_intarray_item:NnF \g__regex_balance_intarray
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_begin_intarray {#1}
          }
          { 0 }
      }
  }
```

(*End of definition for* `\__regex_submatch_balance:n`.)

9.6.3 Framework

`\__regex_replacement:n`
`\__regex_replacement:e`
`\__regex_replacement_apply:Nn`
`\__regex_replacement_set:n`

The replacement text is built incrementally. We keep track in `\l__regex_balance_int` of the balance of explicit begin- and end-group tokens and we store in `\g__regex_balance_tl` some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing `\prg_do_nothing:` because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the `balance_one_match` and `do_one_match` functions.

```
\cs_new_protected:Npn \__regex_replacement:n
```

```
  { \__regex_replacement_apply:Nn \__regex_replacement_set:n }
\cs_new_protected:Npn \__regex_replacement_apply:Nn #1#2
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_balance_int
      \tl_gclear:N \g__regex_balance_tl
      \__regex_escape_use:nnnn
        {
          \if_charcode:w \c_right_brace_str ##1
            \__regex_replacement_rbrace:N
          \else:
            \if_charcode:w \c_left_brace_str ##1
              \__regex_replacement_lbrace:N
            \else:
              \__regex_replacement_normal:n
            \fi:
          \fi:
          ##1
        }
        { \__regex_replacement_escaped:N ##1 }
        { \__regex_replacement_normal:n ##1 }
        {#2}
      \prg_do_nothing: \prg_do_nothing:
      \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
        \msg_error:nne { regex } { replacement-missing-rbrace }
          { \int_use:N \l__regex_replacement_csnames_int }
        \tl_build_put_right:Ne \l__regex_build_tl
          { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
      \fi:
      \seq_if_empty:NF \l__regex_replacement_category_seq
        {
          \msg_error:nne { regex } { replacement-missing-rparen }
            { \seq_count:N \l__regex_replacement_category_seq }
          \seq_clear:N \l__regex_replacement_category_seq
        }
      \tl_gput_right:Ne \g__regex_balance_tl
        { + \int_use:N \l__regex_balance_int }
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNo
    \group_end:
    #1 \l__regex_build_tl
  }
\cs_generate_variant:Nn \__regex_replacement:n { e }
\cs_new_protected:Npn \__regex_replacement_set:n #1
  {
    \cs_set:Npn \__regex_replacement_do_one_match:n ##1
      {
        \__regex_query_range:nn
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_prev_intarray {##1}
          }
          {
```

```
            \__kernel_intarray_item:Nn
              \g__regex_submatch_begin_intarray {##1}
          }
        #1
      }
    \exp_args:Nno \use:n
      { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
      {
        \g__regex_balance_tl
        - \__regex_submatch_balance:n {##1}
      }
  }
```

(*End of definition for* `\__regex_replacement:n`*,* `\__regex_replacement_apply:Nn`*, and* `\__regex_replacement_set:n`*.*)

`\__regex_case_replacement:n`
`\__regex_case_replacement:e`

```
\tl_new:N \g__regex_case_replacement_tl
\tl_new:N \g__regex_case_balance_tl
\cs_new_protected:Npn \__regex_case_replacement:n #1
  {
    \tl_gset:Nn \g__regex_case_balance_tl
      {
        \if_case:w
          \__kernel_intarray_item:Nn
            \g__regex_submatch_case_intarray {##1}
      }
    \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
    \tl_map_tokens:nn {#1}
      { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
    \tl_gset:No \g__regex_balance_tl
      { \g__regex_case_balance_tl \fi: }
    \exp_args:No \__regex_replacement_set:n
      { \g__regex_case_replacement_tl \fi: }
  }
\cs_generate_variant:Nn \__regex_case_replacement:n { e }
\cs_new_protected:Npn \__regex_case_replacement_aux:n #1
  {
    \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
    \tl_gput_right:No \g__regex_case_balance_tl
      { \exp_after:wN \or: \g__regex_balance_tl }
  }
```

(*End of definition for* `\__regex_case_replacement:n`*.*)

`\__regex_replacement_put:n` This gets redefined for `\peek_regex_replace_once:nnTF`.

```
\cs_new_protected:Npn \__regex_replacement_put:n
  { \tl_build_put_right:Nn \l__regex_build_tl }
```

(*End of definition for* `\__regex_replacement_put:n`*.*)

`\__regex_replacement_normal:n`
`\__regex_replacement_normal_aux:N`

Most characters are simply sent to the output by `\tl_build_put_right:Nn`, unless a particular category code has been requested: then `\__regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_-category_tl`. The argument `#1` is a single character (including the case of a catcode-other space). In case no specific catcode is requested, we taked into account the current catcode regime (at the time the replacement is performed) as much as reasonable, with all impossible catcodes (escape, newline, etc.) being mapped to "other".

```
\cs_new_protected:Npn \__regex_replacement_normal:n #1
  {
    \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
      { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
      {
        \tl_if_empty:NTF \l__regex_replacement_category_tl
          { \__regex_replacement_normal_aux:N #1 }
          { % (
            \token_if_eq_charcode:NNTF #1 )
              {
                \seq_pop:NN \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
              }
              {
                \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
                ? #1
              }
          }
      }
  }
\cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
  {
    \token_if_eq_charcode:NNTF #1 \c_space_token
      { \__regex_replacement_c_S:w }
      {
        \exp_after:wN \exp_after:wN
        \if_case:w \tex_catcode:D `#1 \exp_stop_f:
             \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_B:w
        \or: \__regex_replacement_c_E:w
        \or: \__regex_replacement_c_M:w
        \or: \__regex_replacement_c_T:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_P:w
        \or: \__regex_replacement_c_U:w
        \or: \__regex_replacement_c_D:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_S:w
        \or: \__regex_replacement_c_L:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_A:w
        \else: \__regex_replacement_c_O:w
        \fi:
      }
    ? #1
  }
```

(*End of definition for* `\__regex_replacement_normal:n` *and* `\__regex_replacement_normal_aux:N`.)

__regex_replacement_escaped:N

As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character.

```
\cs_new_protected:Npn \__regex_replacement_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_replacement_#1:w }
      {
        \if_int_compare:w 1 < 1#1 \exp_stop_f:
          \__regex_replacement_put_submatch:n {#1}
        \else:
          \__regex_replacement_normal:n {#1}
        \fi:
      }
  }
```

(*End of definition for* `\__regex_replacement_escaped:N`.)

9.6.4 Submatches

__regex_replacement_put_submatch:n
__regex_replacement_put_submatch_aux:n

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

```
\cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
  {
    \if_int_compare:w #1 < \l__regex_capturing_group_int
      \__regex_replacement_put_submatch_aux:n {#1}
    \else:
      \msg_expandable_error:nnff { regex } { submatch-too-big }
        {#1} { \int_eval:n { \l__regex_capturing_group_int - 1 } }
    \fi:
  }
\cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \tl_gput_right:Nn \g__regex_balance_tl
        { + \__regex_submatch_balance:n { \int_eval:n { #1 + ##1 } } }
    \fi:
  }
```

(*End of definition for* `\__regex_replacement_put_submatch:n` *and* `\__regex_replacement_put_submatch_aux:n`.)

`\__regex_replacement_g:w`
__regex_replacement_g_digits:NN

Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```
\cs_new_protected:Npn \__regex_replacement_g:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
      { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
      { \__regex_replacement_error:NNN g #1 #2 }
  }
```

```
\cs_new:Npn \__regex_replacement_g_digits:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \if_int_compare:w 1 < 1#2 \exp_stop_f:
          #2
          \exp_after:wN \use_i:nnn
          \exp_after:wN \__regex_replacement_g_digits:NN
        \else:
          \exp_stop_f:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
      {
        \exp_stop_f:
        \if_meaning:w \__regex_replacement_rbrace:N #1
          \exp_args:No \__regex_replacement_put_submatch:n
            { \int_use:N \l__regex_internal_a_int }
          \exp_after:wN \use_none:nn
        \else:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
    #1 #2
  }
```

(*End of definition for* `\__regex_replacement_g:w` *and* `\__regex_replacement_g_digits:NN`.)

9.6.5 Csnames in replacement

`\__regex_replacement_c:w`

`\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```
\cs_new_protected:Npn \__regex_replacement_c:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \cs_if_exist:cTF { __regex_replacement_c_#2:w }
          { \__regex_replacement_cat:NNN #2 }
          { \__regex_replacement_error:NNN c #1#2 }
      }
      {
        \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
          { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
          { \__regex_replacement_error:NNN c #1#2 }
      }
  }
```

(*End of definition for* `\__regex_replacement_c:w`.)

`\__regex_replacement_cu_aux:Nw`

Start a control sequence with `\cs:w`, protected from expansion by `#1` (either `\__regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction \c or \u. We use \tl_to_str:V rather than \tl_to_str:N to deal with integers and other registers.

```
\cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:n { \exp_after:wN #1 \cs:w } }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
    \fi:
    \int_incr:N \l__regex_replacement_csnames_int
  }
```

(*End of definition for* __regex_replacement_cu_aux:Nw.)

__regex_replacement_u:w

Check that \u is followed by a left brace. If so, start a control sequence with \cs:w, which is then unpacked either with \exp_not:V or \tl_to_str:V depending on the current context.

```
\cs_new_protected:Npn \__regex_replacement_u:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
      { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:V }
      { \__regex_replacement_error:NNN u #1#2 }
  }
```

(*End of definition for* __regex_replacement_u:w.)

__regex_replacement_rbrace:N

Within a \c{...} or \u{...} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```
\cs_new_protected:Npn \__regex_replacement_rbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
      \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
      \int_decr:N \l__regex_replacement_csnames_int
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*End of definition for* __regex_replacement_rbrace:N.)

__regex_replacement_lbrace:N

Within a \c{...} or \u{...} construction, this is forbidden. Otherwise, this is a raw left brace.

```
\cs_new_protected:Npn \__regex_replacement_lbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
      \msg_error:nnn { regex } { cu-lbrace } { u }
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*End of definition for* __regex_replacement_lbrace:N.)

9.6.6 Characters in replacement

__regex_replacement_cat:NNN Here, #1 is a letter among BEMTPUDSLOA and #2#3 denote the next character. Complain if we reach the end of the replacement or if the construction appears inside \c{...} or \u{...}, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```
\cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
  {
    \token_if_eq_meaning:NNTF \prg_do_nothing: #3
      { \msg_error:nn { regex } { replacement-catcode-end } }
      {
        \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
          {
            \msg_error:nnnn
              { regex } { replacement-catcode-in-cs } {#1} {#3}
            #2 #3
          }
          {
            \__regex_two_if_eq:NNNNTF #2 #3 \__regex_replacement_normal:n (
              {
                \seq_push:NV \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
                \tl_set:Nn \l__regex_replacement_category_tl {#1}
              }
              {
                \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
                  {
                    \__regex_char_if_alphanumeric:NTF #3
                      {
                        \msg_error:nnnn
                          { regex } { replacement-catcode-escaped }
                          {#1} {#3}
                      }
                      { }
                  }
                \use:c { __regex_replacement_c_#1:w } #2 #3
              }
          }
      }
  }
```

(*End of definition for* __regex_replacement_cat:NNN*.*)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
\group_begin:
```

__regex_replacement_char:nNN The only way to produce an arbitrary character–catcode pair is to use the \lowercase or \uppercase primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: #3 is the character whose character code to reproduce. We could use

\char_generate:nn but only for some catcodes (active characters and spaces are not supported).

```
\cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
  {
    \tex_lccode:D 0 = `#3 \scan_stop:
    \tex_lowercase:D { \__regex_replacement_put:n {#1} }
  }
```

(*End of definition for* __regex_replacement_char:nNN.)

__regex_replacement_c_A:w

For an active character, expansion must be avoided, twice because we later do two e-expansions, to unpack \toks for the query, and to expand their contents to tokens of the query.

```
\char_set_catcode_active:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_A:w
  { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N ^^@ } } }
```

(*End of definition for* __regex_replacement_c_A:w.)

__regex_replacement_c_B:w

An explicit begin-group token increases the balance, unless within a \c{...} or \u{...} construction. Add the desired begin-group character, using the standard \if_false: trick. We eventually e-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The \exp_after:wN is not strictly needed, but is more consistent with l3tl-analysis.

```
\char_set_catcode_group_begin:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_B:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \int_incr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
  }
```

(*End of definition for* __regex_replacement_c_B:w.)

__regex_replacement_c_C:w

This is not quite catcode-related: when the user requests a character with category "control sequence", the one-character control symbol is returned. As for the active character, we prepare for two e-expansions.

```
\cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
  }
```

(*End of definition for* __regex_replacement_c_C:w.)

__regex_replacement_c_D:w

Subscripts fit the mould: \lowercase the null byte with the correct category.

```
\char_set_catcode_math_subscript:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_D:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* __regex_replacement_c_D:w.)

__regex_replacement_c_E:w

Similar to the begin-group case, the second e-expansion produces the bare end-group token.

```
\char_set_catcode_group_end:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_E:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \int_decr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \if_false: { \fi:  ^^@ } }
  }
```

(*End of definition for* __regex_replacement_c_E:w.)

__regex_replacement_c_L:w

Simply \lowercase a letter null byte to produce an arbitrary letter.

```
\char_set_catcode_letter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_L:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* __regex_replacement_c_L:w.)

__regex_replacement_c_M:w

No surprise here, we lowercase the null math toggle.

```
\char_set_catcode_math_toggle:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_M:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* __regex_replacement_c_M:w.)

__regex_replacement_c_O:w

Lowercase an other null byte.

```
\char_set_catcode_other:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_O:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* __regex_replacement_c_O:w.)

__regex_replacement_c_P:w

For macro parameters, expansion is a tricky issue. We need to prepare for two e-expansions and passing through various macro definitions. Note that we cannot replace one \exp_not:n by doubling the macro parameter characters because this would misbehave if a mischievous user asks for \c{\cP\#}, since that macro parameter character would be doubled.

```
\char_set_catcode_parameter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_P:w
  {
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
  }
```

(*End of definition for* __regex_replacement_c_P:w.)

__regex_replacement_c_S:w

Spaces are normalized on input by TeX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```
\cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
  {
```

```
      \if_int_compare:w `#2 = \c_zero_int
        \msg_error:nn { regex } { replacement-null-space }
      \fi:
      \tex_lccode:D `\ = `#2 \scan_stop:
      \tex_lowercase:D { \__regex_replacement_put:n {~} }
    }
```

(*End of definition for* `\__regex_replacement_c_S:w`*.*)

`\__regex_replacement_c_T:w`

No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```
  \char_set_catcode_alignment:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_T:w
    { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_T:w`*.*)

`\__regex_replacement_c_U:w`

Simple call to `\__regex_replacement_char:nNN` which lowercases the math superscript `^^@`.

```
  \char_set_catcode_math_superscript:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_U:w
    { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_U:w`*.*)

Restore the catcode of the null byte.

```
\group_end:
```

9.6.7 An error

`\__regex_replacement_error:NNN`

Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
\cs_new_protected:Npn \__regex_replacement_error:NNN #1#2#3
  {
    \msg_error:nne { regex } { replacement-#1 } {#3}
    #2 #3
  }
```

(*End of definition for* `\__regex_replacement_error:NNN`*.*)

9.7 User functions

`\regex_new:N`

Before being assigned a sensible value, a regex variable matches nothing.

```
\cs_new_protected:Npn \regex_new:N #1
  { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(*End of definition for* `\regex_new:N`*. This function is documented on page 9.*)

`\l_tmpa_regex`
`\l_tmpb_regex`
`\g_tmpa_regex`
`\g_tmpb_regex`

The usual scratch space.

```
\regex_new:N \l_tmpa_regex
\regex_new:N \l_tmpb_regex
\regex_new:N \g_tmpa_regex
\regex_new:N \g_tmpb_regex
```

(*End of definition for* `\l_tmpa_regex` *and others. These variables are documented on page 14.*)

\regex_set:Nn
\regex_gset:Nn
\regex_const:Nn

Compile, then store the result in the user variable with the appropriate assignment function.

```
\cs_new_protected:Npn \regex_set:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_set_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_gset:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_gset_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_const:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_const:Ne #1 { \exp_not:o \l__regex_internal_regex }
  }
```

(*End of definition for* `\regex_set:Nn`*,* `\regex_gset:Nn`*, and* `\regex_const:Nn`*. These functions are documented on page 9.*)

\regex_show:n
\regex_log:n
__regex_show:Nn
\regex_show:N
\regex_log:N
__regex_show:NN

User functions: the n variant requires compilation first. Then show the variable with some appropriate text. The auxiliary __regex_show:N is defined in a different section.

```
\cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nneeee }
\cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nneeee }
\cs_new_protected:Npn \__regex_show:Nn #1#2
  {
    \__regex_compile:n {#2}
    \__regex_show:N \l__regex_internal_regex
    #1 { regex } { show }
      { \tl_to_str:n {#2} } { }
      { \l__regex_internal_a_tl } { }
  }
\cs_new_protected:Npn \regex_show:N { \__regex_show:NN \msg_show:nneeee }
\cs_new_protected:Npn \regex_log:N { \__regex_show:NN \msg_log:nneeee }
\cs_new_protected:Npn \__regex_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { regex }
      { \exp_args:No \__regex_clean_regex:n {#2} }
      {
        \__regex_show:N #2
        #1 { regex } { show }
          { } { \token_to_str:N #2 }
          { \l__regex_internal_a_tl } { }
      }
  }
```

(*End of definition for* `\regex_show:n` *and others. These functions are documented on page 9.*)

\regex_match:nnTF
\regex_match:nVTF
\regex_match:NnTF
\regex_match:NVTF

Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to \prg_return_-true: or false.

```
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
```

```
  {
    \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
    \__regex_return:
  }
\prg_generate_conditional_variant:Nnn \regex_match:nn { nV } { T , F , TF }
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:N #1 } {#2}
    \__regex_return:
  }
\prg_generate_conditional_variant:Nnn \regex_match:Nn { NV } { T , F , TF }
```

(*End of definition for* `\regex_match:nnTF` *and* `\regex_match:NnTF`*. These functions are documented on page 10.*)

`\regex_count:nnN`
`\regex_count:nVN`
`\regex_count:NnN`
`\regex_count:NVN`

Again, use an auxiliary whose first argument builds the NFA.

```
\cs_new_protected:Npn \regex_count:nnN #1
  { \__regex_count:nnN { \__regex_build:n {#1} } }
\cs_new_protected:Npn \regex_count:NnN #1
  { \__regex_count:nnN { \__regex_build:N #1 } }
\cs_generate_variant:Nn \regex_count:nnN { nV }
\cs_generate_variant:Nn \regex_count:NnN { NV }
```

(*End of definition for* `\regex_count:nnN` *and* `\regex_count:NnN`*. These functions are documented on page 10.*)

`\regex_match_case:nn`
`\regex_match_case:nnTF`

The auxiliary errors if `#1` has an odd number of items, and otherwise it sets `\g__regex_case_int` according to which case was found (zero if not found). The `true` branch leaves the corresponding code in the input stream.

```
\cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
  {
    \__regex_match_case:nnTF {#1} {#2}
      {
        \tl_item:nn {#1} { 2 * \g__regex_case_int }
        #3
      }
  }
\cs_new_protected:Npn \regex_match_case:nn #1#2
  { \regex_match_case:nnTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_match_case:nnT #1#2#3
  { \regex_match_case:nnTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_match_case:nnF #1#2
  { \regex_match_case:nnTF {#1} {#2} { } }
```

(*End of definition for* `\regex_match_case:nnTF`*. This function is documented on page 10.*)

`\regex_extract_once:nnN`
`\regex_extract_once:nVN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:nVNTF`
`\regex_extract_once:NnN`
`\regex_extract_once:NVN`
`\regex_extract_once:NnNTF`
`\regex_extract_once:NVNTF`
`\regex_extract_all:nnN`
`\regex_extract_all:nVN`
`\regex_extract_all:nnNTF`
`\regex_extract_all:nVNTF`
`\regex_extract_all:NnN`
`\regex_extract_all:NVN`
`\regex_extract_all:NnNTF`
`\regex_extract_all:NVNTF`
`\regex_replace_once:nnN`
`\regex_replace_once:nVN`

We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `\__regex_build:n` or `\__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.* The conditionals call `\__regex_return:` to return either `true` or `false` once matching has been performed.

```
\cs_set_protected:Npn \__regex_tmp:w #1#2#3
  {
    \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
```

```
    \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N  ##1  } }
    \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
    \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:N  ##1  } {##2} ##3 \__regex_return: }
    \cs_generate_variant:Nn #2 { nV }
    \prg_generate_conditional_variant:Nnn #2 { nV } { T , F , TF }
    \cs_generate_variant:Nn #3 { NV }
    \prg_generate_conditional_variant:Nnn #3 { NV } { T , F , TF }

  }
\__regex_tmp:w \__regex_extract_once:nnN
  \regex_extract_once:nnN \regex_extract_once:NnN
\__regex_tmp:w \__regex_extract_all:nnN
  \regex_extract_all:nnN \regex_extract_all:NnN
\__regex_tmp:w \__regex_replace_once:nnN
  \regex_replace_once:nnN \regex_replace_once:NnN
\__regex_tmp:w \__regex_replace_all:nnN
  \regex_replace_all:nnN \regex_replace_all:NnN
\__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN
```

(*End of definition for* `\regex_extract_once:nnNTF` *and others. These functions are documented on page* *11*.)

\regex_replace_case_once:nN
\regex_replace_case_once:nN*TF*

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_once:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use. The `\tl_item:nn` is only expanded once we know the value of `\g__regex_case_int`, namely which case matched.

```
\cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_replace_once_aux:nnN
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          { \__regex_replacement:e { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
          #2
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new_protected:Npn \regex_replace_case_once:nN #1#2
  { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
  { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
  { \regex_replace_case_once:nNTF {#1} {#2} { } }
```

(*End of definition for* `\regex_replace_case_once:nNTF`. *This function is documented on page* *13*.)

\regex_replace_case_all:nN
\regex_replace_case_all:nNTF

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as \regex_replace_all:nnN, but with more complicated code to build the automaton, and to find what replacement text to use.

```
\cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_replace_all_aux:nnN
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          { \__regex_case_replacement:e { \__regex_tl_even_items:n {#1} } }
          #2
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new_protected:Npn \regex_replace_case_all:nN #1#2
  { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
  { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
  { \regex_replace_case_all:nNTF {#1} {#2} { } }
```

(*End of definition for \regex_replace_case_all:nNTF. This function is documented on page 13.*)

9.7.1 Variables and helpers for user functions

\l__regex_match_count_int

The number of matches found so far is stored in \l__regex_match_count_int. This is only used in the \regex_count:nnN functions.

```
\int_new:N \l__regex_match_count_int
```

(*End of definition for \l__regex_match_count_int.*)

__regex_begin
__regex_end

Those flags are raised to indicate begin-group or end-group tokens that had to be added when extracting submatches.

```
\flag_new:n { __regex_begin }
\flag_new:n { __regex_end }
```

(*End of definition for __regex_begin and __regex_end.*)

\l__regex_min_submatch_int
\l__regex_submatch_int
\l__regex_zeroth_submatch_int

The end-points of each submatch are stored in two arrays whose index ⟨*submatch*⟩ ranges from \l__regex_min_submatch_int (inclusive) to \l__regex_submatch_int (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at zeroth_submatch. The entry \l__regex_zeroth_submatch_int in \g__regex_submatch_prev_intarray holds the position at which that match attempt started: this is used for splitting and replacements.

```
\int_new:N \l__regex_min_submatch_int
\int_new:N \l__regex_submatch_int
\int_new:N \l__regex_zeroth_submatch_int
```

(*End of definition for* `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, *and* `\l__regex_zeroth_submatch_int`.)

`\g__regex_submatch_prev_intarray` `\g__regex_submatch_begin_intarray` `\g__regex_submatch_end_intarray` `\g__regex_submatch_case_intarray`

Hold the place where the match attempt begun, the end-points of each submatch, and which regex case the match corresponds to, respectively.

```
\intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }
```

(*End of definition for* `\g__regex_submatch_prev_intarray` *and others.*)

`\g__regex_balance_intarray`

The first thing we do when matching is to store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
\intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(*End of definition for* `\g__regex_balance_intarray`.)

`\l__regex_added_begin_int` `\l__regex_added_end_int`

Keep track of the number of left/right braces to add when performing a regex operation such as a replacement.

```
\int_new:N \l__regex_added_begin_int
\int_new:N \l__regex_added_end_int
```

(*End of definition for* `\l__regex_added_begin_int` *and* `\l__regex_added_end_int`.)

`\__regex_return:`

This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```
\cs_new_protected:Npn \__regex_return:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__regex_return:`.)

`\__regex_query_set:n` `\__regex_query_set_aux:nN`

To easily extract subsets of the input once we found the positions at which to cut, store the input tokens one by one into successive `\toks` registers. Also store the brace balance (used to check for overall brace balance) in an array.

```
\cs_new_protected:Npn \__regex_query_set:n #1
  {
    \int_zero:N \l__regex_balance_int
    \int_zero:N \l__regex_curr_pos_int
    \__regex_query_set_aux:nN { } F
    \tl_analysis_map_inline:nn {#1}
      { \__regex_query_set_aux:nN {##1} ##3 }
    \__regex_query_set_aux:nN { } F
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
  }
\cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
  {
    \int_incr:N \l__regex_curr_pos_int
```

```
    \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
    \__kernel_intarray_gset:Nnn \g__regex_balance_intarray
      { \l__regex_curr_pos_int } { \l__regex_balance_int }
    \if_case:w "#2 \exp_stop_f:
    \or: \int_incr:N \l__regex_balance_int
    \or: \int_decr:N \l__regex_balance_int
    \fi:
  }
```

(*End of definition for* `\__regex_query_set:n` *and* `\__regex_query_set_aux:nN`.)

9.7.2 Matching

`\__regex_if_match:nn`

We don't track submatches, and stop after a single match. Build the NFA with `#1`, and perform the match on the query `#2`.

```
\cs_new_protected:Npn \__regex_if_match:nn #1#2
  {
    \group_begin:
      \__regex_disable_submatches:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
    \group_end:
  }
```

(*End of definition for* `\__regex_if_match:nn`.)

`\__regex_match_case:nnTF`
`\__regex_match_case_aux:nn`

The code would get badly messed up if the number of items in `#1` were not even, so we catch this case, then follow the same code as `\regex_match:nnTF` but using `\__regex_case_build:n` and without returning a result.

```
\cs_new_protected:Npn \__regex_match_case:nnTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_match_case:nn(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_if_match:nn
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          {#2}
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }
```

(*End of definition for* `\__regex_match_case:nnTF` *and* `\__regex_match_case_aux:nn`.)

`\__regex_count:nnN`

Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```
\cs_new_protected:Npn \__regex_count:nnN #1#2#3
  {
    \group_begin:
      \__regex_disable_submatches:
      \int_zero:N \l__regex_match_count_int
      \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
      #1
      \__regex_match:n {#2}
      \exp_args:NNNo
    \group_end:
    \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
  }
```

(*End of definition for* `\__regex_count:nnN`.)

9.7.3 Extracting submatches

`\__regex_extract_once:nnN`
`\__regex_extract_all:nnN`

Match once or multiple times. After each match (or after the only match), extract the submatches using `\__regex_extract:`. At the end, store the sequence containing all the submatches into the user variable `#3` after closing the group.

```
\cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
      \__regex_extract:
      \__regex_query_set:n {#2}
    \__regex_group_end_extract_seq:N #3
  }
\cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \__regex_match:n {#2}
      \__regex_query_set:n {#2}
    \__regex_group_end_extract_seq:N #3
  }
```

(*End of definition for* `\__regex_extract_once:nnN` *and* `\__regex_extract_all:nnN`.)

`\__regex_split:nnN`

Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```
\cs_new_protected:Npn \__regex_split:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n
        {
```

```
          \if_int_compare:w
            \l__regex_start_pos_int < \l__regex_success_pos_int
            \__regex_extract:
            \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
              { \l__regex_zeroth_submatch_int } { 0 }
            \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
              { \l__regex_zeroth_submatch_int }
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
                  { \l__regex_zeroth_submatch_int }
              }
            \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
              { \l__regex_zeroth_submatch_int }
              { \l__regex_start_pos_int }
          \fi:
        }
      #1
      \__regex_match:n {#2}
      \__regex_query_set:n {#2}
      \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
        { \l__regex_submatch_int } { 0 }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
        { \l__regex_submatch_int }
        { \l__regex_max_pos_int }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
        { \l__regex_submatch_int }
        { \l__regex_start_pos_int }
      \int_incr:N \l__regex_submatch_int
      \if_meaning:w \c_true_bool \l__regex_empty_success_bool
        \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
          \int_decr:N \l__regex_submatch_int
        \fi:
      \fi:
    \__regex_group_end_extract_seq:N #3
  }
```

(*End of definition for* `\__regex_split:nnN`*.*)

`\__regex_group_end_extract_seq:N`
`\__regex_extract_seq:N`
`\__regex_extract_seq:NNn`
`\__regex_extract_seq_loop:Nw`

The end-points of submatches are stored as entries of two arrays from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` (exclusive). Extract the relevant ranges into `\g__regex_internal_tl`, separated by `\__regex_tmp:w {}`. We keep track in the two flags `__regex_begin` and `__regex_end` of the number of begin-group or end-group tokens added to make each of these items overall balanced. At this step, `}{` is counted as being balanced (same number of begin-group and end-group tokens). This problem is caught by `\__regex_extract_check:w`, explained later. After complaining about any begin-group or end-group tokens we had to add, we are ready to construct the user's sequence outside the group.

```
\cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
  {
      \flag_clear:n { __regex_begin }
      \flag_clear:n { __regex_end }
      \cs_set_eq:NN \__regex_tmp:w \scan_stop:
      \__kernel_tl_gset:Ne \g__regex_internal_tl
        {
```

```
          \int_step_function:nnN { \l__regex_min_submatch_int }
            { \l__regex_submatch_int - 1 } \__regex_extract_seq_aux:n
          \__regex_tmp:w
        }
      \int_set:Nn \l__regex_added_begin_int
        { \flag_height:n { __regex_begin } }
      \int_set:Nn \l__regex_added_end_int
        { \flag_height:n { __regex_end } }
      \tex_afterassignment:D \__regex_extract_check:w
      \__kernel_tl_gset:Ne \g__regex_internal_tl
        { \g__regex_internal_tl \if_false: { \fi: } }
      \int_compare:nNnT
        { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
        {
          \msg_error:nneee { regex } { result-unbalanced }
            { splitting~or~extracting~submatches }
            { \int_use:N \l__regex_added_begin_int }
            { \int_use:N \l__regex_added_end_int }
        }
    \group_end:
    \__regex_extract_seq:N #1
  }
\cs_gset_protected:Npn \__regex_extract_seq:N #1
  {
    \seq_clear:N #1
    \cs_set_eq:NN \__regex_tmp:w  \__regex_extract_seq_loop:Nw
    \exp_after:wN \__regex_extract_seq:NNn
    \exp_after:wN #1
    \g__regex_internal_tl \use_none:nnn
  }
\cs_new_protected:Npn \__regex_extract_seq:NNn #1#2#3
  { #3 #2 #1 \prg_do_nothing: }
\cs_new_protected:Npn \__regex_extract_seq_loop:Nw #1#2 \__regex_tmp:w #3
  {
    \seq_put_right:No #1 {#2}
    #3 \__regex_extract_seq_loop:Nw #1 \prg_do_nothing:
  }
```

(*End of definition for* `\__regex_group_end_extract_seq:N` *and others.*)

`\__regex_extract_seq_aux:n`
`\__regex_extract_seq_aux:ww`

The `:n` auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```
\cs_new:Npn \__regex_extract_seq_aux:n #1
  {
    \__regex_tmp:w { }
    \exp_after:wN \__regex_extract_seq_aux:ww
    \int_value:w \__regex_submatch_balance:n {#1} ; #1;
  }
\cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
  {
    \if_int_compare:w #1 < \c_zero_int
      \prg_replicate:nn {-#1}
        {
```

```
          \flag_raise:n { __regex_begin }
          \exp_not:n { { \if_false: } \fi: }
        }
    \fi:
    \__regex_query_submatch:n {#2}
    \if_int_compare:w #1 > \c_zero_int
      \prg_replicate:nn {#1}
        {
          \flag_raise:n { __regex_end }
          \exp_not:n { \if_false: { \fi: } }
        }
    \fi:
  }
```

(*End of definition for* `\__regex_extract_seq_aux:n` *and* `\__regex_extract_seq_aux:ww`.)

`\__regex_extract_check:w`
`\__regex_extract_check:n`
`\__regex_extract_check_loop:w`
`\__regex_extract_check_end:w`

In `\__regex_group_end_extract_seq:N` we had to expand `\g__regex_internal_tl` to turn `\if_false:` constructions into actual begin-group and end-group tokens. This is done with a `\__kernel_tl_gset:Ne` assignment, and `\__regex_extract_check:w` is run immediately after this assignment ends, thanks to the `\afterassignment` primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so `}{` is not) then `\__regex_extract_check:w` is called just before the closing brace of the `\__kernel_tl_gset:Ne` (thanks to our sneaky `\if_false: { \fi: }` construction), and finds that there is nothing left to expand. If any of the items is unbalanced, the assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `\__kernel_tl_gset:Ne` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```
\cs_new_protected:Npn \__regex_extract_check:w
  {
    \exp_after:wN \__regex_extract_check:n
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new_protected:Npn \__regex_extract_check:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \int_incr:N \l__regex_added_begin_int
        \int_incr:N \l__regex_added_end_int
        \tex_afterassignment:D \__regex_extract_check:w
        \__kernel_tl_gset:Ne \g__regex_internal_tl
          {
            \exp_after:wN \__regex_extract_check_loop:w
            \g__regex_internal_tl
            \__regex_tmp:w \__regex_extract_check_end:w
            #1
          }
      }
  }
\cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
  {
    #2
    \exp_not:o {#1}
```

```
    \__regex_tmp:w { }
    \__regex_extract_check_loop:w \prg_do_nothing:
  }
```

Arguments of `\__regex_extract_check_end:w` are: `#1` is the part of the item before the extra end-group token; `#2` is junk; `#3` is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like `#3`), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```
\cs_new:Npn \__regex_extract_check_end:w
    \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
  {
    { \exp_not:o {#1} }
    #3
    \if_false: { \fi: }
    \__regex_tmp:w
  }
```

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

`\__regex_extract:`
`\__regex_extract_aux:w`

Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. First, we store in `\g__regex_submatch_prev_intarray` the position at which the match attempt started. We extract the rest from the comma list `\l__regex_success_submatches_tl`, which starts with entries to be stored in `\g__regex_submatch_begin_intarray` and continues with entries for `\g__regex_submatch_end_intarray`.

```
\cs_new_protected:Npn \__regex_extract:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
      \prg_replicate:nn \l__regex_capturing_group_int
        {
          \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
            { \l__regex_submatch_int } { 0 }
          \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
            { \l__regex_submatch_int } { 0 }
          \int_incr:N \l__regex_submatch_int
        }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
        { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
        { \l__regex_zeroth_submatch_int } { \g__regex_case_int }
      \int_zero:N \l__regex_internal_a_int
      \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
        \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
        \q__regex_recursion_stop
    \fi:
  }
\cs_new_protected:Npn \__regex_extract_aux:w #1 ,
  {
    \prg_break: #1 \prg_break_point:
    \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
```

```
      \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
        { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#1
    \else:
      \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
        { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int - \l_
    \fi:
    \int_incr:N \l__regex_internal_a_int
    \__regex_extract_aux:w
  }
```

(*End of definition for* `\__regex_extract:` *and* `\__regex_extract_aux:w`.)

9.7.4 Replacement

`\__regex_replace_once:nnN`
`\__regex_replace_once_aux:nnN`

Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional e-expansion, and checks that braces are balanced in the final result.

```
\cs_new_protected:Npn \__regex_replace_once:nnN #1#2
  { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
\cs_new_protected:Npn \__regex_replace_once_aux:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \exp_args:No \__regex_match:n {#3}
    \bool_if:NTF \g__regex_success_bool
      {
        \__regex_extract:
        \exp_args:No \__regex_query_set:n {#3}
        #2
        \int_set:Nn \l__regex_balance_int
          {
            \__regex_replacement_balance_one_match:n
              { \l__regex_zeroth_submatch_int }
          }
        \__kernel_tl_set:Ne \l__regex_internal_a_tl
          {
            \__regex_replacement_do_one_match:n
              { \l__regex_zeroth_submatch_int }
            \__regex_query_range:nn
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
                  { \l__regex_zeroth_submatch_int }
              }
              { \l__regex_max_pos_int }
          }
        \__regex_group_end_replace:N #3
      }
      { \group_end: }
```

```
  }
```

(*End of definition for* `\__regex_replace_once:nnN` *and* `\__regex_replace_once_aux:nnN`.)

`\__regex_replace_all:nnN`

Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```
\cs_new_protected:Npn \__regex_replace_all:nnN #1#2
  { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
\cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \exp_args:No \__regex_match:n {#3}
      \exp_args:No \__regex_query_set:n {#3}
      #2
      \int_set:Nn \l__regex_balance_int
        {
          0
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - 1 }
            \__regex_replacement_balance_one_match:n
        }
      \__kernel_tl_set:Ne \l__regex_internal_a_tl
        {
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - 1 }
            \__regex_replacement_do_one_match:n
          \__regex_query_range:nn
            \l__regex_start_pos_int \l__regex_max_pos_int
        }
    \__regex_group_end_replace:N #3
  }
```

(*End of definition for* `\__regex_replace_all:nnN`.)

`\__regex_group_end_replace:N`
`\__regex_group_end_replace_try:`
`\__regex_group_end_replace_check:w`
`\__regex_group_end_replace_check:n`

At this stage `\l__regex_internal_a_tl` (e-expands to the desired result). Guess from `\l__regex_balance_int` the number of braces to add before or after the result then try expanding. The simplest case is when `\l__regex_internal_a_tl` together with the braces we insert via `\prg_replicate:nn` give a balanced result, and the assignment ends at the `\if_false: { \fi: }` construction: then `\__regex_group_end_replace_check:w` sees that there is no material left and we successfully found the result. The harder case is that expanding `\l__regex_internal_a_tl` may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has not yet been expanded so that `\__regex_group_end_replace_check:n` grabs everything until the last brace in `\__regex_group_end_replace_try:`, letting us try again with an extra surrounding pair of braces.

```
\cs_new_protected:Npn \__regex_group_end_replace:N #1
  {
    \int_set:Nn \l__regex_added_begin_int
      { \int_max:nn { - \l__regex_balance_int } { 0 } }
    \int_set:Nn \l__regex_added_end_int
      { \int_max:nn { \l__regex_balance_int } { 0 } }
    \__regex_group_end_replace_try:
    \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
      {
        \msg_error:nneee { regex } { result-unbalanced }
          { replacing } { \int_use:N \l__regex_added_begin_int }
          { \int_use:N \l__regex_added_end_int }
      }
    \group_end:
    \tl_set_eq:NN #1 \g__regex_internal_tl
  }
\cs_new_protected:Npn \__regex_group_end_replace_try:
  {
    \tex_afterassignment:D \__regex_group_end_replace_check:w
    \__kernel_tl_gset:Ne \g__regex_internal_tl
      {
        \prg_replicate:nn { \l__regex_added_begin_int } { { \if_false: } \fi: }
        \l__regex_internal_a_tl
        \prg_replicate:nn { \l__regex_added_end_int } { \if_false: { \fi: } }
        \if_false: { \fi: }
      }
  }
\cs_new_protected:Npn \__regex_group_end_replace_check:w
  {
    \exp_after:wN \__regex_group_end_replace_check:n
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new_protected:Npn \__regex_group_end_replace_check:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \int_incr:N \l__regex_added_begin_int
        \int_incr:N \l__regex_added_end_int
        \__regex_group_end_replace_try:
      }
  }
```

(*End of definition for* `\__regex_group_end_replace:N` *and others.*)

9.7.5 Peeking ahead

`\l__regex_peek_true_tl`
`\l__regex_peek_false_tl`

True/false code arguments of `\peek_regex:nTF` or similar.

```
\tl_new:N \l__regex_peek_true_tl
\tl_new:N \l__regex_peek_false_tl
```

(*End of definition for* `\l__regex_peek_true_tl` *and* `\l__regex_peek_false_tl`.)

\l__regex_replacement_tl

When peeking in `\peek_regex_replace_once:nnTF` we need to store the replacement text.

```
\tl_new:N \l__regex_replacement_tl
```

(*End of definition for* `\l__regex_replacement_tl`*.*)

\l__regex_input_tl
__regex_input_item:n

Stores each token found as `\__regex_input_item:n {`⟨*tokens*⟩`}`, where the ⟨*tokens*⟩ `o`-expand to the token found, as for `\tl_analysis_map_inline:nn`.

```
\tl_new:N \l__regex_input_tl
\cs_new_eq:NN \__regex_input_item:n ?
```

(*End of definition for* `\l__regex_input_tl` *and* `\__regex_input_item:n`*.*)

\peek_regex:n*TF*
\peek_regex:N*TF*
\peek_regex_remove_once:n*TF*
\peek_regex_remove_once:N*TF*

The `T` and `F` functions just call the corresponding `TF` function. The four `TF` functions differ along two axes: whether to remove the token or not, distinguished by using `\__regex_peek_end:` or `\__regex_peek_remove_end:n` (the latter case needs an argument, as we will see), and whether the regex has to be compiled or is already in an `N`-type variable, distinguished by calling `\__regex_build_aux:Nn` or `\__regex_build_aux:NN`. The first argument of these functions is `\c_false_bool` to indicate that there should be no implicit insertion of a wildcard at the start of the pattern: otherwise the code would keep looking further into the input stream until matching the regex.

```
\cs_new_protected:Npn \peek_regex:nTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:Nn \c_false_bool {#1} }
      { \__regex_peek_end: }
  }
\cs_new_protected:Npn \peek_regex:nT #1#2
  { \peek_regex:nTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
\cs_new_protected:Npn \peek_regex:NTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:NN \c_false_bool #1 }
      { \__regex_peek_end: }
  }
\cs_new_protected:Npn \peek_regex:NT #1#2
  { \peek_regex:NTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
\cs_new_protected:Npn \peek_regex_remove_once:nTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:Nn \c_false_bool {#1} }
      { \__regex_peek_remove_end:n {##1} }
  }
\cs_new_protected:Npn \peek_regex_remove_once:nT #1#2
  { \peek_regex_remove_once:nTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex_remove_once:nF #1
  { \peek_regex_remove_once:nTF {#1} { } }
\cs_new_protected:Npn \peek_regex_remove_once:NTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:NN \c_false_bool #1 }
```

```
      { \__regex_peek_remove_end:n {##1} }
  }
\cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
  { \peek_regex_remove_once:NTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex_remove_once:NF #1
  { \peek_regex_remove_once:NTF #1 { } }
```

(*End of definition for* `\peek_regex:nTF` *and others. These functions are documented on page* **??***.*)

`\__regex_peek:nnTF`
`\__regex_peek_aux:nnTF`

Store the user's true/false codes (plus `\group_end:`) into two token lists. Then build the automaton with `#1`, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like `\regex_-match:nnTF`, with the addition of `\l__regex_input_tl` that keeps track of the tokens seen, to reinsert them at the end. Instead of `\tl_analysis_map_inline:nn` on the input, we call `\peek_analysis_map_inline:n` to go through tokens in the input stream. Since `\__regex_match_one_token:nnN` calls `\__regex_maplike_break:` we need to catch that and break the `\peek_analysis_map_inline:n` loop instead.

```
\cs_new_protected:Npn \__regex_peek:nnTF #1
  {
    \__regex_peek_aux:nnTF
      {
        \__regex_disable_submatches:
        #1
      }
  }
\cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
  {
    \group_begin:
      \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
      \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
      \__regex_single_match:
      #1
      \__regex_match_init:
      \tl_build_begin:N \l__regex_input_tl
      \__regex_match_once_init:
      \peek_analysis_map_inline:n
        {
          \tl_build_put_right:Nn \l__regex_input_tl
            { \__regex_input_item:n {##1} }
          \__regex_match_one_token:nnN {##1} {##2} ##3
          \use_none:nnn
          \prg_break_point:Nn \__regex_maplike_break:
            { \peek_analysis_map_break:n {#2} }
        }
  }
```

(*End of definition for* `\__regex_peek:nnTF` *and* `\__regex_peek_aux:nnTF`*.*)

`\__regex_peek_end:`
`\__regex_peek_remove_end:n`

Once the regex matches (or permanently fails to match) we call `\__regex_peek_end:`, or `\__regex_peek_remove_end:n` with argument the last token seen. For `\peek_regex:nTF` we reinsert tokens seen by calling `\__regex_peek_reinsert:N` regardless of the result of the match. For `\peek_regex_remove_once:nTF` we reinsert the tokens seen only if the match failed; otherwise we just reinsert the tokens `#1`, with one expansion. To be more precise, `#1` consists of tokens that `o`-expand and `e`-expand to the last token seen, for example it is `\exp_not:N` ⟨*cs*⟩ for a control sequence. This means that just doing `\exp_after:wN \l__regex_peek_true_tl #1` would be unsafe because the expansion of ⟨*cs*⟩ would be suppressed.

```
\cs_new_protected:Npn \__regex_peek_end:
  {
    \bool_if:NTF \g__regex_success_bool
      { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
\cs_new_protected:Npn \__regex_peek_remove_end:n #1
  {
    \bool_if:NTF \g__regex_success_bool
      { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
```

(*End of definition for `\__regex_peek_end:` and `\__regex_peek_remove_end:n`.*)

`\__regex_peek_reinsert:N`
`\__regex_reinsert_item:n`

Insert the true/false code `#1`, followed by the tokens found, which were stored in `\l__regex_input_tl`. For this, loop through that token list using `\__regex_reinsert_item:n`, which expands `#1` once to get a single token, and jumps over it to expand what follows, with suitable `\exp:w` and `\exp_end:`. We cannot just use `\use:e` on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```
\cs_new_protected:Npn \__regex_peek_reinsert:N #1
  {
    \tl_build_end:N \l__regex_input_tl
    \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
    \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
  }
\cs_new_protected:Npn \__regex_reinsert_item:n #1
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \exp_end:
    \exp_after:wN \exp_after:wN
    #1
    \exp:w
  }
```

(*End of definition for `\__regex_peek_reinsert:N` and `\__regex_reinsert_item:n`.*)

`\peek_regex_replace_once:nn`
`\peek_regex_replace_once:nnTF`
`\peek_regex_replace_once:Nn`
`\peek_regex_replace_once:NnTF`

Similar to `\peek_regex:nTF` above.

```
\cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
  { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
\cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
  { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2
  { \peek_regex_replace_once:nnTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
  { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
\cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
  { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
\cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
  { \peek_regex_replace_once:NnTF #1 {#2} {#3} { } }
```

```
\cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
  { \peek_regex_replace_once:NnTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
  { \peek_regex_replace_once:NnTF #1 {#2} { } { } }
```

(*End of definition for* `\peek_regex_replace_once:nnTF` *and* `\peek_regex_replace_once:NnTF`. *These functions are documented on page* **??**.)

`\__regex_peek_replace:nnTF`

Same as `\__regex_peek:nnTF` (used for `\peek_regex:nTF` above), but without disabling submatches, and with a different end. The replacement text `#2` is stored, to be analyzed later.

```
\cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
  {
    \tl_set:Nn \l__regex_replacement_tl {#2}
    \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }
  }
```

(*End of definition for* `\__regex_peek_replace:nnTF`.)

`\__regex_peek_replace_end:`

If the match failed `\__regex_peek_reinsert:N` reinserts the tokens found. Otherwise, finish storing the submatch information using `\__regex_extract:`, and store the input into `\toks`. Redefine a few auxiliaries to change slightly their expansion behaviour as explained below. Analyse the replacement text with `\__regex_replacement:n`, which as usual defines `\__regex_replacement_do_one_match:n` to insert the tokens from the start of the match attempt to the beginning of the match, followed by the replacement text. The `\use:e` expands for instance the trailing `\__regex_query_range:nn` down to a sequence of `\__regex_reinsert_item:n` {⟨*tokens*⟩} where ⟨*tokens*⟩ `o`-expand to a single token that we want to insert. After `e`-expansion, `\use:e` does `\use:n`, so we have `\exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:`. This is set up such as to obtain `\l__regex_peek_true_tl` followed by the replaced tokens (possibly unbalanced) in the input stream.

```
\cs_new_protected:Npn \__regex_peek_replace_end:
  {
    \bool_if:NTF \g__regex_success_bool
      {
        \__regex_extract:
        \__regex_query_set_from_input_tl:
        \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
        \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
          \__regex_peek_replacement_put_submatch_aux:n
        \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
        \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
        \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
        \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
        \use:e
          {
            \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
            \__regex_replacement_do_one_match:n
              { \l__regex_zeroth_submatch_int }
            \__regex_query_range:nn
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
                  { \l__regex_zeroth_submatch_int }
```

```
            }
            { \l__regex_max_pos_int }
          \exp_end:
        }
    }
    { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
```

(*End of definition for* `\__regex_peek_replace_end:`.)

`\__regex_query_set_from_input_tl:`
`\__regex_query_set_item:n`

The input was stored into `\l__regex_input_tl` as successive items `\__regex_input_item:n` {⟨*tokens*⟩}. Store that in successive `\toks`. It's not clear whether the empty entries before and after are both useful.

```
\cs_new_protected:Npn \__regex_query_set_from_input_tl:
  {
    \tl_build_end:N \l__regex_input_tl
    \int_zero:N \l__regex_curr_pos_int
    \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
    \__regex_query_set_item:n { }
    \l__regex_input_tl
    \__regex_query_set_item:n { }
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
  }
\cs_new_protected:Npn \__regex_query_set_item:n #1
  {
    \int_incr:N \l__regex_curr_pos_int
    \__regex_toks_set:Nn \l__regex_curr_pos_int { \__regex_input_item:n {#1} }
  }
```

(*End of definition for* `\__regex_query_set_from_input_tl:` *and* `\__regex_query_set_item:n`.)

`\__regex_peek_replacement_put:n`

While building the replacement function `\__regex_replacement_do_one_match:n`, we often want to put simple material, given as `#1`, whose `e`-expansion `o`-expands to a single token. Normally we can just add the token to `\l__regex_build_tl`, but for `\peek_regex_replace_once:nnTF` we eventually want to do some strange expansion that is basically using `\exp_after:wN` to jump through numerous tokens (we cannot use `e`-expansion like for `\regex_replace_once:nnNTF` because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because `\cs:w` ... `\cs_end:` does all the expansion we need.

```
\cs_new_protected:Npn \__regex_peek_replacement_put:n #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:N \__regex_reinsert_item:n {#1} }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl {#1}
    \fi:
  }
```

(*End of definition for* `\__regex_peek_replacement_put:n`.)

__regex_peek_replacement_token:n

When hit with `\exp:w`, `\__regex_peek_replacement_token:n` {⟨*token*⟩} stops `\exp_-end:` and does `\exp_after:wN` ⟨*token*⟩ `\exp:w` to continue expansion after it.

```
\cs_new_protected:Npn \__regex_peek_replacement_token:n #1
  { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }
```

(*End of definition for* `\__regex_peek_replacement_token:n`*.*)

__regex_peek_replacement_put_submatch_aux:n

While analyzing the replacement we also have to insert submatches found in the query. Since query items `\__regex_input_item:n` {⟨*tokens*⟩} expand correctly only when surrounded by `\exp:w` ... `\exp_end:`, and since these expansion controls are not there within csnames (because `\cs:w` ... `\cs_end:` make them unnecessary in most cases), we have to put `\exp:w` and `\exp_end:` by hand here.

```
\cs_new_protected:Npn \__regex_peek_replacement_put_submatch_aux:n #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp:w \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } \exp_end: }
    \fi:
  }
```

(*End of definition for* `\__regex_peek_replacement_put_submatch_aux:n`*.*)

__regex_peek_replacement_var:N

This is used for `\u` outside csnames. It makes sure to continue expansion with `\exp:w` before expanding the variable `#1` and stopping the `\exp:w` that precedes.

```
\cs_new_protected:Npn \__regex_peek_replacement_var:N #1
  {
    \exp_after:wN \exp_last_unbraced:NV
    \exp_after:wN \exp_end:
    \exp_after:wN #1
    \exp:w
  }
```

(*End of definition for* `\__regex_peek_replacement_var:N`*.*)

9.8 Messages

Messages for the preparsing phase.

```
\use:e
  {
    \msg_new:nnn { regex } { trailing-backslash }
      { Trailing~'\iow_char:N\\'~in~regex~or~replacement. }
    \msg_new:nnn { regex } { x-missing-rbrace }
      {
        Missing~brace~'\iow_char:N\}'~in~regex~
        '...\iow_char:N\\x\iow_char:N\{...##1'.
      }
    \msg_new:nnn { regex } { x-overflow }
      {
        Character~code~##1~too~large~in~
        \iow_char:N\\x\iow_char:N\{##2\iow_char:N\}~regex.
      }
  }
```

Invalid quantifier.

```
\msg_new:nnnn { regex } { invalid-quantifier }
  { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
  {
    The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
    The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
    '{<min>,}'~and~'{<min>,<max>}',~optionally~followed~by~'?'.
  }
```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```
\msg_new:nnnn { regex } { missing-rbrack }
  { Missing~right~bracket~inserted~in~regular~expression. }
  {
    LaTeX~was~given~a~regular~expression~where~a~character~class~
    was~started~with~'[',~but~the~matching~']'~is~missing.
  }
\msg_new:nnnn { regex } { missing-rparen }
  {
    Missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
    inserted~in~regular~expression.
  }
  {
    LaTeX~was~given~a~regular~expression~with~\int_eval:n {#1} ~
    more~left~parentheses~than~right~parentheses.
  }
\msg_new:nnnn { regex } { extra-rparen }
  { Extra~right~parenthesis~ignored~in~regular~expression. }
  {
    LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
    was~open.~The~parenthesis~will~be~ignored.
  }
```

Some escaped alphanumerics are not allowed everywhere.

```
\msg_new:nnnn { regex } { bad-escape }
  {
    Invalid~escape~'\iow_char:N\\#1'~
    \__regex_if_in_cs:TF { within~a~control~sequence. }
      {
        \__regex_if_in_class:TF
          { in~a~character~class. }
          { following~a~category~test. }
      }
  }
  {
    The~escape~sequence~'\iow_char:N\\#1'~may~not~appear~
    \__regex_if_in_cs:TF
      {
        within~a~control~sequence~test~introduced~by~
        '\iow_char:N\\c\iow_char:N\{'.
      }
      {
        \__regex_if_in_class:TF
          { within~a~character~class~ }
```

```
        { following~a~category~test~such~as~'\iow_char:N\\cL'~ }
      because~it~does~not~match~exactly~one~character.
    }
  }
```

Range errors.

```
\msg_new:nnnn { regex } { range-missing-end }
  { Invalid~end-point~for~range~'#1-#2'~in~character~class. }
  {
    The~end-point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
    end-point~for~a~range:~alphanumeric~characters~should~not~be~
    escaped,~and~non-alphanumeric~characters~should~be~escaped.
  }
\msg_new:nnnn { regex } { range-backwards }
  { Range~'[#1-#2]'~out~of~order~in~character~class. }
  {
    In~ranges~of~characters~'[x-y]'~appearing~in~character~classes,~
    the~first~character~code~must~not~be~larger~than~the~second.~
    Here,~'#1'~has~character~code~\int_eval:n {`#1},~while~
    '#2'~has~character~code~\int_eval:n {`#2}.
  }
```

Errors related to \c and \u.

```
\msg_new:nnnn { regex } { c-bad-mode }
  { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
  {
    The~'\iow_char:N\\c'~escape~cannot~be~used~within~
    a~control~sequence~test~'\iow_char:N\\c{...}'~
    nor~another~category~test.~
    To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
  }
\msg_new:nnnn { regex } { c-C-invalid }
  { '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'(',~not~'#1'. }
  {
    The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
    control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
    It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
  }
\msg_new:nnnn { regex } { cu-lbrace }
  { Left~braces~must~be~escaped~in~'\iow_char:N\\#1{...}'. }
  {
    Constructions~such~as~'\iow_char:N\\#1{...\iow_char:N\{...}'~are~
    not~allowed~and~should~be~replaced~by~
    '\iow_char:N\\#1{...\token_to_str:N\{...}'.
  }
\msg_new:nnnn { regex } { c-lparen-in-class }
  { Catcode~test~cannot~apply~to~group~in~character~class }
  {
    Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
    class~'[...]'~because~classes~do~not~match~multiple~characters~at~once.
  }
\msg_new:nnnn { regex } { c-missing-rbrace }
  { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
  {
    LaTeX~was~given~a~regular~expression~where~a~
```

```
    '\iow_char:N\\c\iow_char:N\{...'~construction~was~not~ended~
    with~a~closing~brace~'\iow_char:N\}'.
  }
\msg_new:nnnn { regex } { c-missing-rbrack }
  { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
  {
    A~construction~'\iow_char:N\\c[...'~appears~in~a~
    regular~expression,~but~the~closing~']'~is~not~present.
  }
\msg_new:nnnn { regex } { c-missing-category }
  { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
  {
    In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
    may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
    capital~letter~representing~a~character~category,~namely~
    one~of~'ABCDELMOPSTU'.
  }
\msg_new:nnnn { regex } { c-trailing }
  { Trailing~category~code~escape~'\iow_char:N\\c'... }
  {
    A~regular~expression~ends~with~'\iow_char:N\\c'~followed~
    by~a~letter.~It~will~be~ignored.
  }
\msg_new:nnnn { regex } { u-missing-lbrace }
  { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
  {
    The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
    a~brace~group~with~the~name~of~the~variable~to~use.
  }
\msg_new:nnnn { regex } { u-missing-rbrace }
  { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
  {
    LaTeX~
    \str_if_eq:eeTF { } {#2}
      { reached~the~end~of~the~string~ }
      { encountered~an~escaped~alphanumeric~character '\iow_char:N\\#2'~ }
    when~parsing~the~argument~of~an~
    '\iow_char:N\\u\iow_char:N\{...\}'~escape.
  }
```

Errors when encountering the POSIX syntax `[:...:]`.

```
\msg_new:nnnn { regex } { posix-unsupported }
  { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
  {
    The~'[.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
    in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
    Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
  }
\msg_new:nnnn { regex } { posix-unknown }
  { POSIX~class~'[:#1:]'~unknown. }
  {
    '[:#1:]'~is~not~among~the~known~POSIX~classes~
    '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
    '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
    '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
```

```
    '[:word:]',~and~'[:xdigit:]'.
  }
\msg_new:nnnn { regex } { posix-missing-close }
  { Missing~closing~':]'~for~POSIX~class. }
  { The~POSIX~syntax~'#1'~must~be~followed~by~':]',~not~'#2'. }
```

In various cases, the result of a l3regex operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```
\msg_new:nnnn { regex } { result-unbalanced }
  { Missing~brace~inserted~when~#1. }
  {
    LaTeX~was~asked~to~do~some~regular~expression~operation,~
    and~the~resulting~token~list~would~not~have~the~same~number~
    of~begin-group~and~end-group~tokens.~Braces~were~inserted:~
    #2~left,~#3~right.
  }
```

Error message for unknown options.

```
\msg_new:nnnn { regex } { unknown-option }
  { Unknown~option~'#1'~for~regular~expressions. }
  {
    The~only~available~option~is~'case-insensitive',~toggled~by~
    '(?i)'~and~'(?-i)'.
  }
\msg_new:nnnn { regex } { special-group-unknown }
  { Unknown~special~group~'#1~...'~in~a~regular~expression. }
  {
    The~only~valid~constructions~starting~with~'(?'~are~
    '(?:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
  }
```

Errors in the replacement text.

```
\msg_new:nnnn { regex } { replacement-c }
  { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
    or~a~brace~group,~not~by~'#1'.
  }
\msg_new:nnnn { regex } { replacement-u }
  { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
    must~be~~followed~by~a~brace~group~holding~the~name~of~the~
    variable~to~use.
  }
\msg_new:nnnn { regex } { replacement-g }
  {
    Missing~brace~for~the~'\iow_char:N\\g'~construction~
    in~a~replacement~text.
  }
  {
    In~the~replacement~text~for~a~regular~expression~search,~
    submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
```

```
    or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
  }
\msg_new:nnnn { regex } { replacement-catcode-end }
  {
    Missing~character~for~the~'\iow_char:N\\c<category><character>'~
    construction~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow.~LaTeX~
    reached~the~end~of~the~replacement~when~looking~for~that.
  }
\msg_new:nnnn { regex } { replacement-catcode-escaped }
  {
    Escaped~letter~or~digit~after~category~code~in~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow,~not~
    '\iow_char:N\\#2'.
  }
\msg_new:nnnn { regex } { replacement-catcode-in-cs }
  {
    Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
    '\iow_char:N\\c\{...\}'~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~category~codes~of~the~argument~of~
    '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control~
    sequence~name.
  }
\msg_new:nnnn { regex } { replacement-null-space }
  { TeX~cannot~build~a~space~token~with~character~code~0. }
  {
    You~asked~for~a~character~token~with~category~space,~
    and~character~code~0,~for~instance~through~
    '\iow_char:N\\cS\iow_char:N\\x00'.~
    This~specific~case~is~impossible~and~will~be~replaced~
    by~a~normal~space.
  }
\msg_new:nnnn { regex } { replacement-missing-rbrace }
  { Missing~right~brace~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
  }
\msg_new:nnnn { regex } { replacement-missing-rparen }
  { Missing~right~parenthesis~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
```

```
  }
\msg_new:nnn { regex } { submatch-too-big }
  { Submatch~#1~used~but~regex~only~has~#2~group(s) }
```

Some escaped alphanumerics are not allowed everywhere.

```
\msg_new:nnnn { regex } { backwards-quantifier }
  { Quantifer~"{#1,#2}"~is~backwards. }
  { The~values~given~in~a~quantifier~must~be~in~order. }
```

Used in user commands, and when showing a regex.

```
\msg_new:nnnn { regex } { case-odd }
  { #1~with~odd~number~of~items }
  {
    There~must~be~a~#2~part~for~each~regex:~
    found~odd~number~of~items~(#3)~in\\
    \iow_indent:n {#4}
  }
\msg_new:nnn { regex } { show }
  {
    >~Compiled~regex~
    \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
    #3
  }
\prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { regex } { }
```

__regex_msg_repeated:nnN

This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (-1 for infinite number), and #3 tells us about lazyness.

```
\cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
  {
    \str_if_eq:eeF { #1 #2 } { 1 0 }
      {
        , ~ repeated ~
        \int_case:nnF {#2}
          {
            { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
            {  0 } { #1~times }
          }
          {
            between~#1~and~\int_eval:n {#1+#2}~times,~
            \bool_if:NTF #3 { lazy } { greedy }
          }
      }
  }
```

(*End of definition for* __regex_msg_repeated:nnN.)

9.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

__regex_trace_push:nnN
__regex_trace_pop:nnN
__regex_trace:nne

Here #1 is the module name (regex) and #2 is typically 1. If the module's current tracing level is less than #2 show nothing, otherwise write #3 to the terminal.

```
\cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
  { \__regex_trace:nne {#1} {#2} { entering~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
  { \__regex_trace:nne {#1} {#2} { leaving~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace:nne #1#2#3
  {
    \int_compare:nNnF
      { \int_use:c { g__regex_trace_#1_int } } < {#2}
      { \iow_term:e { Trace:~#3 } }
  }
```

(*End of definition for* __regex_trace_push:nnN, __regex_trace_pop:nnN, *and* __regex_trace:nne.)

\g__regex_trace_regex_int

No tracing when that is zero.

```
\int_new:N \g__regex_trace_regex_int
```

(*End of definition for* \g__regex_trace_regex_int.)

__regex_trace_states:n

This function lists the contents of all states of the NFA, stored in \toks from 0 to \l__regex_max_state_int (excluded).

```
\cs_new_protected:Npn \__regex_trace_states:n #1
  {
    \int_step_inline:nnn
      \l__regex_min_state_int
      { \l__regex_max_state_int - 1 }
      {
        \__regex_trace:nne { regex } {#1}
          { \iow_char:N \\toks ##1 = { \__regex_toks_use:w ##1 } }
      }
  }
```

(*End of definition for* __regex_trace_states:n.)

```
⟨/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

\$ 991
\\ . 5, 329, 330, 1725, 1732, 1733, 1734, 1858, 3685, 3689, 3694, 3728, 3737, 3741, 3746, 3766, 3768, 3769, 3771, 3774, 3776, 3781, 3783, 3785, 3790, 3794, 3797, 3801, 3803, 3807, 3809, 3815, 3817, 3821, 3823, 3827, 3832, 3834, 3876, 3878, 3883, 3885, 3891, 3896, 3897, 3901, 3905, 3915, 3918, 3922, 3923, 3927, 3935, 3961, 4006
\{ 278, 3689, 3694, 3741, 3783, 3785, 3797, 3834, 3923, 3927
\} 3688, 3694, 3798, 3834, 3923, 3927
\␣ 223, 228, 272, 282, 466, 2949
\^ 224, 229, 230, 231, 232, 235, 246, 283, 337, 339, 341, 343, 345, 347, 990, 2900, 2903, 2917, 2920, 2929, 2932, 2935, 2938, 2952, 2955
\~ 268, 272, 278

B

bool commands:
\bool_gset_eq:NN 214, 2383
\bool_gset_false:N 2330
\bool_gset_true:N 2396
\bool_if:NTF 1106, 1115, 1556, 1729, 1815, 1833, 1851, 2005, 2224, 2232, 2465, 3077, 3100, 3173, 3395, 3565, 3571, 3612, 3979, 3984
\bool_if:nTF 1818
\bool_lazy_all:nTF 1389
\bool_new:N 71, 501, 2299, 2300, 2302, 2303, 2304
\bool_set_eq:NN 208, 2544
\bool_set_false:N 1080, 1285, 2274, 2345, 2359, 2422, 2464
\bool_set_true:N 1085, 1289, 2268, 2462, 2543
\c_false_bool *117*, 766, 784, 978, 1025, 1324, 1526, 1543, 1556, 1752, 1888, 2393, 3500, 3509, 3518, 3528, 3590, 3598
\c_true_bool . . . 78, 211, 649, 723, 780, 968, 970, 972, 974, 976, 986, 1024, 1031, 1524, 1534, 1556, 1557, 1750, 1871, 1873, 1896, 1991, 2162, 2173, 2188, 2349, 3124, 3241, 3354

C

char commands:
\char_generate:nn *100*, 362, 1399, 1773
\char_set_catcode_active:N . . . 2900
\char_set_catcode_alignment:N . 2952
\char_set_catcode_group_begin:N 2903
\char_set_catcode_group_end:N . 2920
\char_set_catcode_letter:N . . . 2929
\char_set_catcode_math_subscript:N 2917
\char_set_catcode_math_superscript:N 2955
\char_set_catcode_math_toggle:N 2932
\char_set_catcode_other:N 2935
\char_set_catcode_parameter:N . 2938

cs commands:
\cs:w *97*, *98*, *121*, *122*, 2830, 2833
\cs_end: *121*, *122*, 2639, 2846
\cs_generate_variant:Nn 746, 1899, 2654, 2695, 3024, 3025, 3048, 3050
\cs_gset:Npn 2671
\cs_gset_protected:Npn 3277
\cs_if_eq:NNTF 1374
\cs_if_eq_p:NN 1391
\cs_if_exist:NTF 750, 1102, 2816
\cs_if_exist_use:N 1763
\cs_if_exist_use:NTF 315, 322, 709, 714, 758, 1170, 1256, 2752
\cs_new:Npe 336, 338, 340, 342, 344, 346
\cs_new:Npn 6, 35, 40, 42, 51, 53, 54, 59, 65, 87, 89, 91, 313, 319, 330, 335, 348, 353, 365, 380, 390, 402, 425, 536, 554, 562, 574, 586, 1090, 1372, 1387, 1421, 1437, 1484, 1522, 1553, 1559, 1565, 1573, 1578, 1584, 1589, 1603, 1618, 1627, 1635, 1637, 1689, 1698, 1769, 2015, 2430, 2534, 2537, 2558, 2560, 2566, 2568, 2575, 2585, 2785, 3176, 3292, 3298, 3337, 3344, 3972
\cs_new_eq:NN 3, 9, 157, 158, 159, 258, 266, 287, 326, 327, 328, 329, 515, 2301, 2567, 2965, 3496
\cs_new_protected:Npe . 930, 944, 946
\cs_new_protected:Npn 4, 7, 10, 12, 21, 27, 33, 93, 95, 96, 101, 107, 115, 125, 139, 160, 170, 178, 180, 188, 200, 219, 221, 226, 234, 236, 243, 248, 250, 252, 259, 267, 269, 271, 273, 280, 285, 288, 294, 530, 594, 607, 618, 631, 664, 693, 702, 707, 712, 717, 738, 747, 754, 763, 768, 775, 788, 790, 792, 794, 800, 822, 835, 862, 867, 901, 936, 957, 959, 967, 969, 971, 973, 975, 977, 981, 992, 1008, 1021, 1027, 1038, 1051, 1057, 1076, 1096, 1127, 1138, 1153, 1166, 1184, 1192, 1197, 1199, 1201, 1218, 1237, 1239, 1262, 1274, 1294, 1315, 1322, 1329, 1340, 1347, 1353, 1411, 1463, 1472, 1485, 1500, 1509, 1528, 1547, 1704, 1775, 1785, 1787, 1789, 1796, 1841, 1854, 1870, 1872, 1874, 1879, 1894, 1900, 1923, 1946, 1961, 1968, 1975, 1977, 1979, 1986, 2000, 2016, 2025, 2039, 2051, 2068, 2077, 2079, 2091, 2100, 2112, 2125, 2132, 2152, 2183, 2217, 2235, 2244, 2250, 2256, 2262, 2305, 2314, 2328, 2347, 2374, 2379, 2389, 2401, 2439, 2448, 2460, 2467, 2469, 2471, 2491, 2496, 2502, 2513, 2518, 2523, 2539, 2591, 2610, 2612, 2655, 2679, 2696, 2702, 2704, 2724, 2750, 2761, 2770, 2779, 2812, 2826, 2837, 2843, 2852, 2860, 2895, 2901, 2904, 2912, 2918, 2921, 2930, 2933, 2936, 2939, 2944, 2953, 2956, 2959, 2964, 2970, 2975, 2980, 2985, 2986, 2987, 2995, 2996, 2997,

3020, 3022, 3026, 3034, 3036, 3038, 3042, 3043, 3063, 3080, 3082, 3084, 3086, 3103, 3105, 3107, 3122, 3130, 3140, 3151, 3160, 3177, 3189, 3199, 3208, 3248, 3285, 3287, 3316, 3321, 3352, 3374, 3387, 3389, 3420, 3422, 3451, 3467, 3478, 3483, 3497, 3503, 3505, 3506, 3512, 3514, 3515, 3521, 3523, 3525, 3531, 3533, 3535, 3543, 3563, 3569, 3575, 3581, 3589, 3591, 3593, 3595, 3597, 3599, 3601, 3603, 3605, 3610, 3639, 3649, 3654, 3663, 3665, 3675, 3988, 3990, 3992, 3999
\cs_set:Npe 2002
\cs_set:Npn 298, 299, 300, 626, 627, 1205, 1207, 1224, 1226, 1466, 1467, 1731, 1732, 1733, 1734, 1760, 1805, 2350, 2657
\cs_set_eq:NN 155, 1723, 1757, 2356, 2406, 3252, 3280, 3578, 3616, 3617, 3619, 3620, 3621, 3643
\cs_set_nopar:Npe 23, 29
\cs_set_protected:Npn . . 942, 979, 1708, 1717, 1719, 1721, 1724, 1726, 1735, 1737, 1742, 1744, 1749, 1751, 1753, 1755, 1758, 2515, 2516, 3040
\cs_to_str:N 37, 1538, 1674

E

else commands:
\else: 46, 143, 144, 149, 150, 167, 174, 374, 384, 434, 443, 455, 456, 458, 460, 463, 464, 467, 468, 477, 479, 481, 484, 485, 487, 523, 526, 547, 550, 558, 566, 569, 578, 581, 590, 598, 601, 611, 731, 845, 889, 893, 896, 907, 912, 1002, 1148, 1161, 1250, 1279, 1318, 1336, 1449, 1505, 1539, 1569, 1994, 2012, 2031, 2065, 2118, 2165, 2169, 2176, 2197, 2208, 2355, 2468, 2578, 2622, 2625, 2745, 2756, 2765, 2793, 2805, 2831, 2848, 2856, 3126, 3380, 3659, 3670
exp commands:
\exp:w *119, 120*, *122*, 3579, 3587, 3625, 3664, 3672, 3680
\exp_after:wN *100*, *119–122*, 11, 24, 25, 30, 31, 34, 37, 45, 47, 104, 111, 118, 122, 129, 135, 173, 175, 184, 306, 309, 350, 368, 373, 375, 376, 383, 386, 387, 393, 405, 417, 436, 445, 557, 559, 565, 568, 570, 577, 580, 582, 589, 591, 597, 600, 603, 697, 939, 1003, 1015, 1147, 1150, 1160, 1162, 1344, 1351, 1359, 1444, 1538, 1570, 1957, 2247, 2253, 2259, 2370, 2394, 2425, 2456, 2482, 2520, 2570, 2571, 2579, 2582, 2700, 2729, 2791, 2792, 2795, 2796, 2804, 2806, 2807, 2830, 2833, 2910, 3281, 3282, 3295, 3318, 3319, 3330, 3369, 3480, 3481, 3579, 3583, 3584, 3585, 3625, 3664, 3677, 3678, 3679
\exp_args:Nc 1300
\exp_args:Ne 210, 990, 991, 1362, 2493
\exp_args:Nf 41, 1671, 1673, 2096
\exp_args:NNe 1468
\exp_args:NNf 2368
\exp_args:NNNe 660, 1143
\exp_args:NNNo . 652, 1765, 1828, 3185
\exp_args:NNo 2650, 3572
\exp_args:Nno 2670
\exp_args:No . . 742, 806, 826, 1513, 1562, 2070, 2498, 2692, 2707, 2802, 3000, 3394, 3398, 3427, 3428, 3622
\exp_args:Noo 848, 2083
\exp_end: *119, 120*, *122*, 3579, 3584, 3634, 3664, 3672, 3678
\exp_last_unbraced:Nf 361, 1691
\exp_last_unbraced:NV 3677
\exp_not:N *91*, *119*, 337, 339, 341, 343, 345, 347, 730, 732, 932, 934, 945, 949, 1107, 1780, 2487, 2902, 2915, 3658
\exp_not:n *6*, *19*, *91*, *97*, *98*, *101*, *113*, 62, 730, 732, 1365, 1837, 2004, 2222, 2411, 2476, 2488, 2566, 2567, 2830, 2833, 2902, 2910, 2927, 2942, 2983, 3176, 3304, 3312, 3340, 3345, 3347, 3625
\exp_stop_f: 142, 143, 144, 150, 172, 432, 452, 453, 457, 461, 462, 465, 466, 474, 475, 478, 482, 483, 486, 545, 905, 910, 924, 925, 938, 1000, 1001, 1040, 1317, 2009, 2062, 2577, 2581, 2730, 2754, 2789, 2794, 2800, 3146

F

fi commands:
\fi: *112, 113, 115*, 48, 105, 112, 113, 119, 123, 130, 131, 136, 137, 147, 148, 153, 154, 168, 176, 185, 186, 213, 369, 377, 388, 394, 406, 446, 448, 455, 458, 459, 463, 467, 468, 469, 470, 479, 480, 484, 487, 488, 489, 525, 528, 549, 552, 560, 571, 572, 583, 584, 592, 604, 605, 615, 616, 629, 648, 649,

657, 658, 723, 733, 766, 780, 784, 847, 895, 898, 899, 914, 917, 940, 998, 999, 1004, 1031, 1032, 1043, 1047, 1081, 1086, 1094, 1129, 1136, 1141, 1151, 1163, 1189, 1252, 1281, 1320, 1327, 1338, 1426, 1445, 1451, 1456, 1479, 1491, 1492, 1495, 1507, 1541, 1571, 1958, 1997, 2013, 2037, 2057, 2066, 2123, 2130, 2150, 2168, 2179, 2181, 2211, 2214, 2248, 2254, 2260, 2357, 2395, 2426, 2427, 2457, 2483, 2528, 2580, 2627, 2628, 2640, 2691, 2693, 2746, 2758, 2768, 2777, 2797, 2808, 2834, 2850, 2858, 2908, 2910, 2925, 2927, 2948, 3128, 3149, 3227, 3244, 3245, 3265, 3304, 3306, 3312, 3314, 3319, 3349, 3372, 3383, 3472, 3474, 3475, 3481, 3661, 3673

flag commands:
\flag_clear:n 1356, 3250, 3251
\flag_ensure_raised:n . . . 1383, 1405
\flag_height:n 3260, 3262
\flag_if_raised:nTF 1364
\flag_new:n 1346, 3110, 3111
\flag_raise:n 3303, 3311

G

group commands:
\group_begin: 204, 296, 620, 1131, 1465, 1706, 1807, 2237, 2614, 2894, 3153, 3179, 3191, 3201, 3210, 3391, 3424, 3545
\group_end: . . . *118*, 216, 310, 653, 661, 1144, 1469, 1766, 1814, 1821, 1829, 2241, 2242, 2651, 2958, 3158, 3186, 3274, 3418, 3464, 3546, 3547
\group_insert_after:N 212

I

if commands:
\if:w *34*, 521
\if_case:w 163, 435, 1248, 1277, 1334, 2009, 2062, 2683, 2730, 2828, 3146, 3656, 3667
\if_charcode:w 372, 382, 891, 1447, 2620, 2623
\if_false: *100*, *112*, *113*, *115*, 629, 648, 649, 658, 723, 766, 780, 784, 998, 1031, 1043, 1047, 1081, 1086, 1094, 1129, 1136, 1141, 1189, 1426, 1445, 1456, 1479, 1491, 1492, 1495, 2910, 2927, 3265, 3304, 3312, 3319, 3349, 3472, 3474, 3475, 3481
\if_int_compare:w 44, 103, 109, 110, 117, 121, 127, 128, 133, 134, 142, 143, 144, 150, 182, 183, 432, 452, 453, 454, 457, 461, 462, 465, 466, 474, 475, 478, 482, 483, 486, 545, 567, 579, 588, 596, 599, 609, 612, 640, 727, 839, 905, 910, 938, 996, 1029, 1140, 1157, 1503, 1536, 1567, 1956, 2027, 2053, 2114, 2127, 2138, 2154, 2205, 2246, 2252, 2258, 2423, 2424, 2451, 2478, 2577, 2635, 2754, 2763, 2774, 2789, 2845, 2854, 2906, 2923, 2946, 3213, 3242, 3300, 3308, 3377
\if_int_odd:w 172, 556, 564, 576, 1002, 1317
\if_meaning:w 211, 367, 392, 404, 520, 544, 887, 890, 1324, 1991, 2162, 2173, 2188, 2349, 2393, 2528, 2801, 3124, 3241, 3354

int commands:
\int_add:Nn 151, 1319, 2144, 2145, 2403, 2475
\int_case:nnTF 3977
\int_compare:nNnTF 190, 202, 355, 682, 684, 1549, 2352, 2706, 2865, 3266, 3458, 3994
\int_compare:nTF 1772, 1812, 3713, 3942, 3943, 3948, 3950
\int_compare_p:n 1819
\int_compare_p:nNn 1394, 1395
\int_const:Nn 81, 82, 83, 84, 493, 494, 495, 496, 497, 498, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514
\int_decr:N . . . 2847, 2924, 3148, 3243
\int_eval:n *18*, 41, 172, 436, 1317, 1563, 1771, 1976, 1978, 1992, 1993, 1995, 1996, 2138, 2228, 2271, 2446, 2494, 2593, 2767, 2773, 2776, 3669, 3672, 3717, 3762, 3763, 3983
\int_gincr:N 1917
\int_gset:Nn 1937
\int_gzero:N 1897, 1914
\int_if_exist:NTF 1243, 1298
\int_if_odd:nTF 3065, 3088, 3162
\int_if_odd_p:n 1845
\int_incr:N 17, 18, 1134, 1777, 1944, 1984, 2073, 2404, 2500, 2835, 2907, 3142, 3147, 3182, 3240, 3325, 3326, 3362, 3384, 3487, 3488, 3651
\int_max:nn 1622, 1623, 1630, 1631, 1929, 2097, 3454, 3456
\int_new:N 68, 69, 70, 80, 491, 492, 499, 500, 517, 1862, 1864, 1865, 1866, 1869,

1892, 1893, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2285, 2286, 2287, 2288, 2291, 2292, 2293, 2554, 3109, 3112, 3113, 3114, 3120, 3121, 3998
\int_set:Nn 5, 1325, 1863, 1925, 1927, 1933, 1971, 1973, 2042, 2093, 2094, 2104, 2115, 2139, 2157, 2206, 2338, 2340, 2343, 2364, 2408, 2409, 2450, 2485, 3187, 3259, 3261, 3400, 3430, 3453, 3455
\int_set_eq:NN 141, 610, 614, 623, 625, 668, 735, 1033, 1133, 1146, 1245, 1883, 1903, 1920, 1948, 1982, 1983, 2033, 2136, 2137, 2189, 2238, 2316, 2339, 2344, 2358, 2362, 2366, 2405, 2415, 2546, 2547, 3138, 3355, 3647
\int_step_function:nnN . . . 2416, 3255
\int_step_function:nnnN . . 3433, 3441
\int_step_inline:nnn 2331, 4001
\int_sub:Nn . 145, 846, 2192, 2200, 2209
\int_to_Hex:n 358
\int_use:N 642, 729, 807, 818, 827, 831, 842, 843, 849, 850, 856, 857, 1016, 1845, 1938, 1943, 1964, 1966, 2071, 2084, 2085, 2486, 2538, 2637, 2648, 2803, 3187, 3271, 3272, 3461, 3462, 3995
\int_value:w 351, 875, 881, 913, 915, 924, 925, 1040, 1525, 1540, 2571, 2572, 2583, 3296
\int_zero:N 622, 844, 1282, 1809, 1882, 1913, 2337, 2616, 3132, 3133, 3181, 3368, 3642
\c_max_char_int 355
\c_max_int *81*
\c_one_int 2127, 2138
\l_tmpa_int *7*
\c_zero_int 44, 640, 1140, 1536, 2027, 2053, 2114, 2154, 2205, 2635, 2774, 2845, 2854, 2906, 2923, 2946, 3300, 3308
intarray commands:
\intarray_new:Nn 2294, 2295, 3115, 3116, 3117, 3118, 3119
iow commands:
\iow_char:N 329, 330, 337, 339, 341, 343, 345, 347, 990, 991, 1725, 1732, 1733, 1734, 1858, 3685, 3688, 3689, 3694, 3728, 3737, 3741, 3746, 3766, 3768, 3769, 3771, 3774, 3776, 3781, 3783, 3785, 3790, 3794, 3797, 3798, 3801, 3803, 3807, 3809, 3815, 3817, 3821, 3823, 3827, 3832, 3834, 3876, 3878, 3883, 3885, 3891, 3896, 3901, 3905, 3915, 3918, 3922, 3923, 3927, 3935, 4006
\iow_indent:n 3962
\iow_newline: 1780
\iow_term:n 3996

K

kernel internal commands:
__kernel_chk_tl_type:NnnTF . . 2999
__kernel_intarray_gset:Nnn 2334, 2441, 2444, 3144, 3216, 3218, 3224, 3232, 3234, 3237, 3358, 3360, 3364, 3366, 3378, 3381
__kernel_intarray_gset_range_-from_clist:Nnn 2504
__kernel_intarray_item:Nn 49, 2452, 2479, 2563, 2564, 2588, 2589, 2597, 2604, 2661, 2665, 2684, 3221, 3411, 3630
__kernel_intarray_range_to_-clist:Nnn 2433
__kernel_quark_new_conditional:Nn 92
__kernel_str_to_other_fast:n 303, 1513
__kernel_tl_gset:Nn *112*, 302, 1511, 3253, 3264, 3328, 3470
__kernel_tl_set:Nn 192, 1081, 1086, 1357, 1426, 3405, 3439

M

msg commands:
\msg_error:nn 602, 635, 683, 686, 1159, 1430, 2863, 2947
\msg_error:nnn 641, 864, 1258, 1271, 1310, 1343, 1457, 2636, 2643, 2855, 2961
\msg_error:nnnn 841, 906, 1121, 2867, 2883
\msg_error:nnnnn 3269, 3460
\msg_error:nnnnnn . . . 3067, 3090, 3164
\msg_expandable_error:nn 332
\msg_expandable_error:nnn . 427, 1448
\msg_expandable_error:nnnn 357, 2766
\msg_log:nnnnnn 2986, 2996
\g_msg_module_name_prop 3970
\g_msg_module_type_prop 3971
\msg_new:nnn 3684, 3686, 3691, 3952, 3964
\msg_new:nnnn 3697, 3704, 3710, 3720, 3726, 3750, 3757, 3765, 3773, 3780, 3787, 3793, 3800, 3806, 3814, 3820, 3826, 3836, 3843, 3852, 3855, 3863,

3869, 3875, 3882, 3889, 3899, 3910, 3920, 3930, 3939, 3945, 3954, 3957
\msg_show:nnnnnn 2985, 2995
\msg_warning:nn 1149
\msg_warning:nnn 1065, 1069, 1111, 1173, 1211, 1230
\msg_warning:nnnn 771, 920

O

or commands:
\or: 164, 165, 166, 167, 438, 439, 440, 441, 442, 2011, 2064, 2698, 2700, 2732, 2733, 2734, 2735, 2736, 2737, 2738, 2739, 2740, 2741, 2742, 2743, 2744, 3147, 3148

P

peek commands:
\peek_analysis_map_break:n . . . 3560
\peek_analysis_map_inline:n *118*, 3553
\peek_regex:NTF 3497, 3506, 3512, 3513, 3514
\peek_regex:nTF *69*, *116*, *118–120*, 3497, 3497, 3503, 3504, 3505
\peek_regex_remove_once:NTF 3497, 3525, 3531, 3532, 3533, 3534
\peek_regex_remove_once:nTF *118*, 3497, 3515, 3521, 3522, 3523, 3524
\peek_regex_replace_once:Nn 3589, 3603
\peek_regex_replace_once:nn 3589, 3595
\peek_regex_replace_once:NnTF 3589, 3597, 3599, 3600, 3601, 3602, 3604
\peek_regex_replace_once:nnTF *91*, *94*, *117*, *121*, 3589, 3589, 3591, 3592, 3593, 3594, 3596

prg commands:
\prg_break: *89*, 61, 329, 333, 1591, 1601, 1606, 1615, 1639, 1684, 2551, 2579, 3376
\prg_break:n 2087
\prg_break_point: . . 57, 307, 1587, 1636, 2088, 2421, 2573, 3370, 3376
\prg_break_point:Nn *19*, 2312, 2326, 2372, 3559
\prg_do_nothing: *38*, *92*, *113*, 155, 637, 680, 681, 688, 689, 2634, 2862, 3286, 3290, 3342
\prg_generate_conditional_-variant:Nnn . 3013, 3019, 3049, 3051
\prg_map_break:Nn 52
\prg_new_conditional:Npnn 430, 450, 472, 518, 542
\prg_new_protected_conditional:Npnn 885, 3008, 3014, 3044, 3046
\prg_replicate:nn *115*, 14, 643, 1398, 2018, 2044, 2190, 2198, 2361, 2527, 2639, 3301, 3309, 3356, 3472, 3474
\prg_return_false: . *107*, 444, 455, 458, 463, 467, 468, 476, 479, 484, 487, 524, 527, 548, 551, 892, 897, 3127
\prg_return_true: *103*, *107*, 433, 447, 455, 458, 463, 467, 479, 484, 487, 522, 546, 888, 894, 3125

prop commands:
\prop_gput:Nnn 3970, 3971

Q

quark commands:
\quark_if_recursion_tail_stop:n 1580, 1700
\quark_new:N 85, 86
\q_recursion_stop 1576, 1695
\q_recursion_tail 1576, 1694

quark internal commands:
\q__regex_nil 56, 61, 86, 91, 698, 702, 1360, 1378, 1379, 1474, 1484
\q__regex_recursion_stop 85, 88, 90, 1360, 1379, 3371

R

regex commands:
\regex_const:Nn *9*, 2970, 2980
\regex_count:NnN . *10*, 3020, 3022, 3025
\regex_count:nnN *10*, *106*, 3020, 3020, 3024
\regex_extract_all:NnN *11*, 3040, 3057
\regex_extract_all:nnN *2*, *11*, *17*, 3040, 3057
\regex_extract_all:NnNTF . . . *11*, 3040
\regex_extract_all:nnNTF . . . *11*, 3040
\regex_extract_once:NnN *11*, 3040, 3055
\regex_extract_once:nnN *11*, 3040, 3055
\regex_extract_once:NnNTF . . *11*, 3040
\regex_extract_once:nnNTF *5*, *11*, 3040
\regex_gset:Nn *9*, 2970, 2975
\regex_log:N *9*, *61*, 2985, 2996
\regex_log:n *9*, 2985, 2986
\regex_match:Nn 3014, 3019
\regex_match:nn 3008, 3013
\regex_match:NnTF *10*, 3008
\regex_match:nnTF . . *10*, *108*, *118*, 3008
\regex_match_case:nn *10*, *13*, *39*, *69*, 3026, 3034, 3165

\regex_match_case:nnTF . . *10*, 3026, 3026, 3035, 3036, 3037, 3038, 3039
\regex_new:N *9*, *20*, 2964, 2964, 2966, 2967, 2968, 2969
\regex_replace_all:NnN *12*, 3040, 3061
\regex_replace_all:nnN *2*, *12*, *106*, 3040, 3061
\regex_replace_all:NnNTF . . . *12*, 3040
\regex_replace_all:nnNTF . . . *12*, 3040
\regex_replace_case_all:nN *13*, 3086, 3091, 3103
\regex_replace_case_all:nNTF *13*, 3086, 3086, 3104, 3105, 3106, 3107, 3108
\regex_replace_case_once:nN *13*, 3063, 3068, 3080
\regex_replace_case_once:nNTF *13*, 3063, 3063, 3081, 3082, 3083, 3084, 3085
\regex_replace_once:NnN *12*, 3040, 3059
\regex_replace_once:nnN *11–13*, *105*, 3040, 3059
\regex_replace_once:NnNTF . . *12*, 3040
\regex_replace_once:nnNTF *12*, *121*, 3040
\regex_set:Nn . . . *1*, *9*, *10*, 2970, 2970
\regex_show:N . . . *9*, *49*, *61*, 2985, 2995
\regex_show:n *2*, *7*, *9*, 2985, 2985
\regex_split:NnN *12*, 3040, 3062
\regex_split:nnN *12*, 3040, 3062
\regex_split:NnNTF *12*, 3040
\regex_split:nnNTF *12*, 3040
\g_tmpa_regex *14*, 2966
\l_tmpa_regex *14*, 2966
\g_tmpb_regex *14*, 2966
\l_tmpb_regex *14*, 2966
regex internal commands:
__regex_A_test: *32*, 968, 990, 1606, 1609, 1615, 1733, 2217, 2250
__regex_action_cost:n *68*, *72*, 2006, 2007, 2015, 2465, 2491, 2491
__regex_action_free:n . . . *68*, *81*, 2029, 2035, 2036, 2047, 2105, 2109, 2134, 2159, 2163, 2166, 2194, 2202, 2212, 2226, 2269, 2463, 2467, 2467
__regex_action_free_aux:nn 2467, 2468, 2470, 2471
__regex_action_free_group:n *68*, *81*, 2055, 2174, 2177, 2467, 2469
__regex_action_start_wildcard:N *68*, 1887, 1907, 2460, 2460
__regex_action_submatch:nN *68*, 1911, 1936, 2128, 2129, 2267, 2516, 2518, 2518
__regex_action_submatch_aux:w 2518, 2520, 2523
__regex_action_submatch_auxii:w 2518, 2529, 2534
__regex_action_submatch_-auxiii:w 2518, 2530, 2535, 2536, 2537
__regex_action_submatch_auxiv:w 2518
__regex_action_success: *68*, 1890, 1939, 1957, 2539, 2539
__regex_action_wildcard: *86*
\l__regex_added_begin_int 3120, 3259, 3267, 3271, 3325, 3453, 3458, 3461, 3472, 3487
\l__regex_added_end_int 3120, 3261, 3267, 3272, 3326, 3455, 3458, 3462, 3474, 3488
\c__regex_all_catcodes_int 502, 624, 728, 1326
\c__regex_ascii_lower_int 84, 146, 152
\c__regex_ascii_max_control_int 81, 263
\c__regex_ascii_max_int 81, 256, 264, 454
\c__regex_ascii_min_int . 81, 255, 262
__regex_assertion:Nn *32*, *46*, *79*, 964, 986, 1595, 1726, 2217, 2217
__regex_b_test: *32*, *79*, 976, 978, 1612, 1731, 2217, 2235
\l__regex_balance_int *21*, *92*, *115*, 80, 2616, 2648, 2907, 2924, 3132, 3145, 3147, 3148, 3400, 3430, 3454, 3456
\g__regex_balance_intarray *18*, *107*, 2595, 2602, 3119, 3144
\g__regex_balance_tl . . . *92*, 2557, 2617, 2647, 2673, 2690, 2700, 2775
__regex_begin 3110
__regex_branch:n *32*, *50*, *75*, 77, 629, 704, 1136, 1189, 1374, 1484, 1492, 1576, 1578, 1581, 1708, 2100, 2100
__regex_break_point:TF *21*, *22*, *45*, *72*, 93, 94, 95, 99, 2006, 2007, 2223, 2240
__regex_break_true:w . . . *22*, 93, 93, 99, 104, 111, 118, 122, 129, 135, 184, 196, 212, 939, 2247, 2253, 2259
__regex_build:N *104*, 1870, 1872, 3016, 3023, 3043, 3047
__regex_build:n *70*, *104*, 1870, 1870, 3010, 3021, 3042, 3045
__regex_build_aux:NN . *117*, 1870, 1873, 1877, 1879, 3509, 3528, 3598

__regex_build_aux:Nn *117*, 1870, 1871, 1874, 3500, 3518, 3590
__regex_build_for_cs:n 207, 1946, 1946
__regex_build_new_state: 1884, 1885, 1904, 1905, 1909, 1949, 1950, 1979, 1979, 1988, 2020, 2054, 2058, 2102, 2117, 2122, 2161, 2180, 2215, 2219, 2264
\l__regex_build_tl *50*, *121*, 74, 621, 628, 646, 651, 654, 655, 658, 659, 662, 722, 725, 765, 779, 783, 908, 922, 963, 985, 998, 1030, 1043, 1047, 1129, 1132, 1135, 1141, 1142, 1145, 1188, 1478, 1482, 1489, 1495, 1516, 1532, 1550, 1707, 1764, 1767, 1778, 1808, 1823, 1827, 1830, 1836, 2615, 2638, 2649, 2652, 2703, 2772, 2829, 2832, 2846, 2914, 3657, 3660, 3668, 3671
__regex_build_transition_-left:NNN 1975, 1975, 2163, 2177, 2194
__regex_build_transition_-right:nNn 1975, 1977, 2021, 2055, 2105, 2109, 2134, 2159, 2166, 2174, 2202, 2212
__regex_build_transitions_-lazyness:NNNNN 1986, 1986, 2028, 2034, 2046
\l__regex_capturing_group_int *17*, *68*, *115*, 1869, 1882, 1920, 1925, 1930, 2071, 2073, 2084, 2085, 2093, 2094, 2097, 2361, 2435, 2436, 2509, 2528, 2763, 2767, 3356, 3377, 3382, 3435, 3443
\g__regex_case_balance_tl 2678, 2681, 2687, 2691, 2699
__regex_case_build:n *108*, 1894, 1894, 1899, 3074, 3097, 3171
__regex_case_build_aux:Nn 1894, 1896, 1900
__regex_case_build_loop:n 1894, 1918, 1923
\l__regex_case_changed_char_int . *23*, 121, 133, 134, 141, 145, 151, 2282
\g__regex_case_int *104*, *105*, 1892, 1897, 1914, 1917, 1937, 1938, 3030, 3075, 3367
\l__regex_case_max_group_int 1893, 1913, 1920, 1927, 1929
__regex_case_replacement:n 2677, 2679, 2695, 3098
__regex_case_replacement_aux:n 2689, 2696
\g__regex_case_replacement_tl 2677, 2687, 2693, 2698
\c__regex_catcode_A_int 502
\c__regex_catcode_B_int 502
\c__regex_catcode_C_int 502
\c__regex_catcode_D_int 502
\c__regex_catcode_E_int 502
\c__regex_catcode_in_class_mode_-int . 492, 613, 997, 1158, 1251, 1280
\c__regex_catcode_L_int 502
\c__regex_catcode_M_int 502
\c__regex_catcode_mode_int 492, 609, 682, 1029, 1249, 1278
\c__regex_catcode_O_int 502
\c__regex_catcode_P_int 502
\c__regex_catcode_S_int 502
\c__regex_catcode_T_int 502
\c__regex_catcode_U_int 502
\l__regex_catcodes_bool 499, 1285, 1289, 1324
\l__regex_catcodes_int *33*, 499, 625, 727, 729, 735, 1016, 1033, 1133, 1146, 1245, 1282, 1317, 1319, 1325, 1326
__regex_char_if_alphanumeric:N 472
__regex_char_if_alphanumeric:NTF 450, 675, 2881
__regex_char_if_special:N 450
__regex_char_if_special:NTF 450, 671
__regex_chk_c_allowed:TF 594, 594, 1238
__regex_class:NnnnN . *32*, *40*, *41*, *47*, 78, 723, 1024, 1025, 1031, 1391, 1524, 1534, 1596, 1723, 2000, 2000
\c__regex_class_mode_int 492, 599, 614
__regex_class_repeat:n *73*, 2010, 2016, 2016, 2032, 2041
__regex_class_repeat:nN 2011, 2025, 2025
__regex_class_repeat:nnN 2012, 2039, 2039
__regex_clean_assertion:Nn 1553, 1595, 1603
__regex_clean_bool:n 1553, 1553, 1605, 1620, 1624, 1632
__regex_clean_branch:n 1553, 1581, 1584
__regex_clean_branch_loop:n 1553, 1586, 1589, 1594, 1616, 1625, 1633
__regex_clean_class:n 1553, 1621, 1635, 1646, 1667
__regex_clean_class:NnnnN 1553, 1596, 1618

__regex_clean_class_loop:nnn 1553, 1636, 1637, 1648, 1658, 1668, 1682
__regex_clean_exact_cs:n 1553, 1643, 1689
__regex_clean_exact_cs:w 1553, 1693, 1698, 1702
__regex_clean_group:nnnN 1553, 1597, 1598, 1599, 1627
__regex_clean_int:n 1553, 1559, 1562, 1622, 1623, 1630, 1631, 1644, 1645, 1657, 1667
__regex_clean_int_aux:N 1553, 1563, 1565
__regex_clean_regex:n 1553, 1573, 1629, 1642, 3000
__regex_clean_regex_loop:w 1553, 1575, 1578, 1582
__regex_command_K: 32, 1550, 1594, 1724, 2262, 2262
__regex_compile:n 664, 664, 700, 1876, 2972, 2977, 2982, 2989
__regex_compile:w 38, 618, 618, 666, 1331
__regex_compile_$: 959
__regex_compile_(: 1153
__regex_compile_): 1192
__regex_compile_.: 930
__regex_compile_/A: 959
__regex_compile_/B: 959
__regex_compile_/b: 959
__regex_compile_/c: 1237
__regex_compile_/D: 942
__regex_compile_/d: 942
__regex_compile_/G: 959
__regex_compile_/H: 942
__regex_compile_/h: 942
__regex_compile_/K: 1547
__regex_compile_/N: 942
__regex_compile_/S: 942
__regex_compile_/s: 942
__regex_compile_/u: 1411
__regex_compile_/V: 942
__regex_compile_/v: 942
__regex_compile_/W: 942
__regex_compile_/w: 942
__regex_compile_/Z: 959
__regex_compile_/z: 959
__regex_compile_[: 1008
__regex_compile_]: 992
__regex_compile_^: 959
__regex_compile_abort_tokens:n 738, 738, 746, 772, 1113, 1123
__regex_compile_anchor_letter:NNN 959, 959, 968, 970, 972, 974, 976, 978
__regex_compile_c_[:w 1274
__regex_compile_c_C:NN 1253, 1262, 1262
__regex_compile_c_lbrack_add:N 1274, 1300, 1315
__regex_compile_c_lbrack_end: 1274, 1307, 1311, 1322
__regex_compile_c_lbrack_-loop:NN 1274, 1286, 1290, 1294, 1302
__regex_compile_c_test:NN 1237, 1238, 1239
__regex_compile_class:NN 1038, 1044, 1048, 1051
__regex_compile_class:TFNN 48, 1023, 1034, 1038, 1038
__regex_compile_class_catcode:w 1015, 1027, 1027
__regex_compile_class_normal:w 1018, 1021, 1021
__regex_compile_class_posix:NNNNw 1057, 1063, 1076
__regex_compile_class_posix_-end:w 1057, 1094, 1096
__regex_compile_class_posix_-loop:w . 1057, 1082, 1087, 1090, 1093
__regex_compile_class_posix_-test:w 1011, 1057, 1057
__regex_compile_cs_aux:Nn 1346, 1359, 1372, 1380
__regex_compile_cs_aux:NNnnnN 1346, 1377, 1387, 1400
__regex_compile_end: 38, 39, 618, 631, 691, 1355
__regex_compile_end_cs: 687, 1346, 1350, 1353
__regex_compile_escaped:N 676, 707, 712
__regex_compile_group_begin:N 1127, 1127, 1175, 1180, 1198, 1200
__regex_compile_group_end: 1127, 1138, 1195
__regex_compile_if_quantifier:TFw 747, 747, 1475, 1487
__regex_compile_lparen:w 1162, 1166
__regex_compile_one:n . . 717, 717, 874, 880, 934, 945, 948, 958, 1104, 1362
__regex_compile_quantifier:w . . . 736, 754, 754, 1003, 1147, 1480, 1496
__regex_compile_quantifier_*:w 788
__regex_compile_quantifier_+:w 788
__regex_compile_quantifier_?:w 788

__regex_compile_quantifier_-abort:nNN 763, 768, 798, 817, 830, 853
__regex_compile_quantifier_-braced_auxi:w 794, 797, 800
__regex_compile_quantifier_-braced_auxii:w 794, 813, 822
__regex_compile_quantifier_-braced_auxiii:w 794, 812, 835
__regex_compile_quantifier_-lazyness:nnNN *42*, 775, 775, 789, 791, 793, 806, 826, 848
__regex_compile_quantifier_-none: 759, 761, 763, 763, 770
__regex_compile_range:Nw 872, 885, 901
__regex_compile_raw:N 544, 672, 676, 678, 710, 715, 743, 865, 867, 867, 887, 933, 983, 1006, 1054, 1074, 1092, 1150, 1155, 1160, 1176, 1186, 1194, 1212, 1213, 1214, 1220, 1231, 1232, 1233, 1241, 1296, 1344, 1351, 1416, 1432, 1433, 1439
__regex_compile_raw_error:N 862, 862, 961, 1414, 1551
__regex_compile_special:N *34*, 672, 707, 707, 749, 756, 777, 804, 809, 824, 837, 871, 890, 1041, 1059, 1078, 1098, 1099, 1168, 1203, 1221, 1264, 1283, 1423, 1442
__regex_compile_special_group_--:w 1201
__regex_compile_special_group_-::w 1197
__regex_compile_special_group_-i:w 1201, 1201
__regex_compile_special_group_-|:w 1197
__regex_compile_u_brace:NNN 1417, 1418, 1421, 1421
__regex_compile_u_end: 1418, 1485, 1485
__regex_compile_u_in_cs: 1506, 1509, 1509
__regex_compile_u_in_cs_aux:n 1519, 1522
__regex_compile_u_loop:NN 1427, 1437, 1437, 1440, 1452
__regex_compile_u_not_cs: 1504, 1528, 1528
__regex_compile_u_payload: *59*, 1485, 1494, 1498, 1500
__regex_compile_ur:n *58*, 1463, 1470, 1472
__regex_compile_ur_aux:w 1463, 1474, 1484
__regex_compile_ur_end: 1417, 1431, 1463, 1463
__regex_compile_use:n 693, 693, 1926
__regex_compile_use_aux:w . 697, 702
__regex_compile_|: 1184
__regex_compute_case_changed_-char: 139, 139, 157, 2406
__regex_count:nnN 3021, 3023, 3177, 3177
__regex_cs 1346
\c__regex_cs_in_class_mode_int 492, 1337
\c__regex_cs_mode_int 492, 1335
\l__regex_curr_analysis_tl *82*, 2296, 2342, 2370, 2377, 2411, 2412
\l__regex_curr_catcode_int 163, 182, 190, 202, 2282, 2409
\l__regex_curr_char_int . *84*, 103, 109, 110, 117, 127, 128, 141, 142, 143, 144, 150, 183, 938, 1956, 2238, 2246, 2282, 2366, 2405, 2408, 2424
__regex_curr_cs_to_str: 35, 35, 193, 210
\l__regex_curr_pos_int *19*, *84*, 2258, 2277, 2353, 2364, 2404, 2538, 2546, 3133, 3138, 3142, 3143, 3145, 3642, 3647, 3651, 3652
\l__regex_curr_state_int *81*, *87*, 2288, 2442, 2443, 2445, 2450, 2453, 2475, 2480, 2485, 2486, 2494
\l__regex_curr_submatches_tl 2289, 2360, 2455, 2487, 2488, 2499, 2521, 2525, 2550
\l__regex_curr_token_tl 38, 2282, 2407
\l__regex_default_catcodes_int . . *33*, 499, 623, 625, 735, 1033, 1133, 1146
__regex_disable_submatches: . . . 206, 1332, 2513, 2513, 3154, 3180, 3539
\l__regex_empty_success_bool 2299, 2345, 2349, 2544, 3241
__regex_end 3110
__regex_escape_␣:w 329
__regex_escape_/\scan_stop::w . 329
__regex_escape_/a:w 329
__regex_escape_/e:w 329
__regex_escape_/f:w 329
__regex_escape_/n:w 329
__regex_escape_/r:w 329
__regex_escape_/t:w 329
__regex_escape_/x:w 348
__regex_escape_\:w 313
__regex_escape_\scan_stop::w . . 329

__regex_escape_escaped:N 299, 323, 326, 327
__regex_escape_loop:N 27, 306, 313, 313, 317, 320, 324, 348, 387, 398, 399, 419, 428
__regex_escape_raw:N 28, 300, 326, 328, 337, 339, 341, 343, 345, 347, 361
__regex_escape_unescaped:N 298, 316, 326, 326
__regex_escape_use:nnnn 27, 38, 294, 294, 669, 2618
__regex_escape_x:N 29, 386, 390, 390
__regex_escape_x_end:w 28, 348, 350, 353
__regex_escape_x_large:n 348
__regex_escape_x_loop:N 29, 383, 402, 402, 411, 414
__regex_escape_x_loop_error: . . 402
__regex_escape_x_loop_error:n 408, 420, 425
__regex_escape_x_test:N 28, 351, 365, 365, 373
__regex_escape_x_testii:N 365, 375, 380
\l__regex_every_match_tl 2298, 2381, 2391, 2428
__regex_extract: 109, 120, 3195, 3202, 3215, 3352, 3352, 3397, 3425, 3614
__regex_extract_all:nnN 3056, 3189, 3199
__regex_extract_aux:w 3352, 3369, 3374, 3385
__regex_extract_check:n 3316, 3318, 3321
__regex_extract_check:w 110, 112, 3263, 3316, 3316, 3327
__regex_extract_check_end:w 113, 3316, 3332, 3344
__regex_extract_check_loop:w 3316, 3330, 3337, 3342, 3345
__regex_extract_once:nnN 3054, 3189, 3189
__regex_extract_seq:N 3248, 3275, 3277
__regex_extract_seq:NNn 3248, 3281, 3285
__regex_extract_seq_aux:n 3256, 3292, 3292
__regex_extract_seq_aux:ww 3292, 3295, 3298
__regex_extract_seq_loop:Nw 3248, 3280, 3287, 3290
\l__regex_fresh_thread_bool 83, 87, 2268, 2274, 2299, 2422, 2462, 2464, 2545
__regex_G_test: 32, 970, 1610, 1734, 2217, 2256
__regex_get_digits:NTFw 530, 530, 796, 811
__regex_get_digits_loop:nw 533, 536, 539
__regex_get_digits_loop:w 530
__regex_group:nnnN 32, 50, 1175, 1180, 1466, 1597, 1717, 1888, 2068, 2068
__regex_group_aux:nnnnN 75, 2051, 2051, 2070, 2078, 2081
__regex_group_aux:nnnnnN 75
__regex_group_end_extract_seq:N . . . 112, 3197, 3206, 3246, 3248, 3248
__regex_group_end_replace:N 3416, 3449, 3451, 3451
__regex_group_end_replace_check:n 116, 3451, 3480, 3483
__regex_group_end_replace_check:w 115, 3451, 3469, 3478
__regex_group_end_replace_try: 116, 3451, 3457, 3467, 3489
\l__regex_group_level_int 491, 622, 640, 642, 644, 1134, 1140
__regex_group_no_capture:nnnN 32, 1198, 1466, 1467, 1479, 1491, 1598, 1719, 2068, 2077
__regex_group_repeat:nn 2063, 2112, 2112
__regex_group_repeat:nnN 2064, 2152, 2152
__regex_group_repeat:nnnN 2065, 2183, 2183
__regex_group_repeat_aux:n 76, 77, 2119, 2132, 2132, 2170, 2187
__regex_group_resetting:nnnN . . . 32, 1200, 1467, 1599, 1721, 2079, 2079
__regex_group_resetting_loop:nnNn . . 2079, 2083, 2091, 2096
__regex_group_submatches:nNN 2120, 2125, 2125, 2155, 2171, 2185
__regex_hexadecimal_use:N 430
__regex_hexadecimal_use:NTF 385, 397, 410, 430
__regex_if_end_range:NN 885
__regex_if_end_range:NNTF . 885, 903
__regex_if_in_class:TF 554, 554, 633, 720, 736, 869, 932, 994, 1010, 1155, 1186, 1194, 3731, 3744

__regex_if_in_class_or_catcode:TF 574, 574, 961, 983, 1413
__regex_if_in_cs:TF 562, 562, 1342, 1349, 3729, 3738
__regex_if_match:nn 3010, 3016, 3151, 3151, 3170
__regex_if_raw_digit:NN 542
__regex_if_raw_digit:NNTF 532, 538, 542
__regex_if_two_empty_matches:TF 83, 2299, 2301, 2350, 2356, 2541
__regex_if_within_catcode:TF 586, 586, 1013
__regex_input_item:n 117, 121, 122, 3495, 3496, 3556, 3578, 3619, 3643, 3652
\l__regex_input_tl 118, 119, 121, 3495, 3551, 3555, 3577, 3579, 3641, 3645
__regex_int_eval:w 3, 3, 2507, 2571, 2572, 2583, 3379, 3382
__regex_intarray_item:NnTF 40, 40, 2595, 2602
__regex_intarray_item_aux:nNTF 40, 41, 42
\l__regex_internal_a_int 42, 96, 66, 796, 807, 818, 827, 831, 839, 842, 846, 849, 856, 2033, 2036, 2042, 2047, 2121, 2136, 2142, 2148, 2157, 2160, 2164, 2167, 2172, 2175, 2178, 2193, 2201, 2210, 2782, 2803, 3368, 3377, 3379, 3382, 3384
\l__regex_internal_a_tl 27, 59, 60, 64, 115, 66, 192, 195, 297, 304, 311, 1081, 1086, 1102, 1107, 1112, 1116, 1122, 1123, 1357, 1368, 1426, 1470, 1502, 1514, 1530, 1711, 1714, 1767, 1788, 1830, 1837, 1932, 1933, 1970, 1971, 1972, 1973, 2103, 2104, 2108, 2110, 2367, 2370, 2993, 3005, 3405, 3439, 3473
\l__regex_internal_b_int 66, 811, 840, 843, 844, 846, 850, 857, 2137, 2142, 2147, 2193, 2201, 2210
\l__regex_internal_b_tl 66, 1425, 1445, 1458
\l__regex_internal_bool 66, 1080, 1085, 1106, 1115
\l__regex_internal_c_int 66, 2139, 2144, 2145, 2149
\l__regex_internal_regex 37, 515, 662, 700, 1359, 1365, 1877, 2973, 2978, 2983, 2990
\l__regex_internal_seq . . 66, 1843, 1844, 1849, 1856, 1857, 1858, 1860
\g__regex_internal_tl 110, 112, 66, 302, 306, 1511, 1518, 3253, 3264, 3265, 3283, 3328, 3331, 3465, 3470
__regex_item_caseful_equal:n 32, 101, 101, 223, 224, 228, 229, 230, 231, 232, 241, 246, 264, 282, 626, 1225, 1393, 1525, 1644, 1735
__regex_item_caseful_range:nn 32, 101, 107, 220, 235, 238, 239, 240, 254, 261, 268, 270, 272, 275, 276, 277, 278, 283, 286, 291, 292, 627, 1227, 1652, 1737
__regex_item_caseless_equal:n 32, 115, 115, 1206, 1645, 1742
__regex_item_caseless_range:nn 32, 115, 125, 1208, 1653, 1744
__regex_item_catcode: . 160, 160, 172
__regex_item_catcode:nTF . . . 32, 48, 160, 170, 179, 729, 1035, 1663, 1749
__regex_item_catcode_reverse:nTF 32, 160, 178, 1036, 1664, 1751
__regex_item_cs:n 32, 200, 200, 1365, 1642, 1758
__regex_item_equal:n . . 158, 158, 626, 875, 881, 911, 924, 925, 1205, 1224
__regex_item_exact:nn 32, 60, 180, 180, 1540, 1654, 1755
__regex_item_exact_cs:n 32, 55, 180, 188, 1367, 1537, 1643, 1757
__regex_item_range:nn 158, 159, 627, 913, 1207, 1226
__regex_item_reverse:n 32, 49, 96, 96, 179, 245, 949, 1106, 1646, 1753, 2241
\l__regex_last_char_int 2238, 2252, 2282, 2405, 2547
\l__regex_last_char_success_int 2282, 2340, 2366, 2547
\l__regex_left_state_int . . 1865, 1886, 1906, 1910, 1964, 1971, 1982, 1989, 1992, 1993, 1995, 1996, 2022, 2030, 2033, 2056, 2104, 2106, 2116, 2136, 2156, 2158, 2186, 2189, 2192, 2195, 2207, 2220, 2229, 2265, 2272
\l__regex_left_state_seq 1865, 1963, 1970, 2103
__regex_maplike_break: 19, 118, 51, 51, 52, 2312, 2326, 2372, 2386, 2394, 3559
__regex_match:n . 2305, 2305, 3157, 3184, 3194, 3204, 3230, 3394, 3427

__regex_match_case:nnTF 3028, 3160, 3160
__regex_match_case_aux:nn 3160, 3176
\l__regex_match_count_int 106, 108, 3109, 3181, 3182, 3187
__regex_match_cs:n . . 210, 2305, 2314
__regex_match_init: 2305, 2307, 2317, 2328, 3550
__regex_match_once_init: 2308, 2318, 2347, 2347, 2398, 3552
__regex_match_once_init_aux: 2368, 2374
__regex_match_one_active:n 2401, 2419, 2430
__regex_match_one_token:nnN 84, 87, 118, 2310, 2311, 2322, 2323, 2325, 2371, 2401, 2401, 3557
\l__regex_match_success_bool 83, 2302, 2359, 2385, 2393, 2543
\l__regex_matched_analysis_tl . . . 82, 2296, 2341, 2367, 2376, 2410, 2548
\l__regex_max_pos_int 91, 2277, 3138, 3236, 3242, 3414, 3447, 3633, 3647
\l__regex_max_state_int 67, 71, 129, 1862, 1883, 1903, 1941, 1943, 1944, 1948, 1981, 1983, 1984, 2043, 2115, 2135, 2137, 2145, 2189, 2195, 2203, 2213, 2332, 4003
\l__regex_max_thread_int 2292, 2316, 2362, 2415, 2418, 2423, 2500, 2508
__regex_maybe_compute_ccc: 120, 132, 155, 157, 2406
\l__regex_min_pos_int 91, 2277, 2338, 2339
\l__regex_min_state_int . 71, 1862, 1883, 1903, 1948, 2332, 2363, 4002
\l__regex_min_submatch_int 106, 110, 115, 2343, 2344, 3112, 3255, 3434, 3442
\l__regex_min_thread_int 2292, 2316, 2362, 2415, 2417, 2423
\l__regex_mode_int 492, 556, 564, 567, 576, 579, 588, 596, 599, 609, 610, 612, 614, 668, 682, 684, 996, 1000, 1001, 1002, 1029, 1040, 1157, 1247, 1248, 1276, 1277, 1333, 1334, 1503, 1549
__regex_mode_quit_c: 607, 607, 719, 1130
__regex_msg_repeated:nnN 1803, 1824, 1834, 3972, 3972
__regex_multi_match:n 82, 2379, 2389, 3182, 3202, 3211, 3425
\c__regex_no_match_regex 75, 515, 2965
\c__regex_outer_mode_int 492, 567, 579, 588, 596, 610, 668, 684, 1503, 1549
__regex_peek:nnTF 120, 3499, 3508, 3517, 3527, 3535, 3535
__regex_peek_aux:nnTF 3535, 3537, 3543, 3608
__regex_peek_end: 117, 118, 3501, 3510, 3563, 3563
\l__regex_peek_false_tl 3492, 3547, 3567, 3573, 3637
__regex_peek_reinsert:N . . . 118, 120, 3566, 3567, 3573, 3575, 3575, 3637
__regex_peek_remove_end:n 117, 118, 3519, 3529, 3563, 3569
__regex_peek_replace:nnTF 3590, 3598, 3605, 3605
__regex_peek_replace_end: 3608, 3610, 3610
__regex_peek_replacement_put:n 3616, 3654, 3654
__regex_peek_replacement_put_-submatch_aux:n . . 3618, 3665, 3665
__regex_peek_replacement_-token:n 122, 3620, 3663, 3663
__regex_peek_replacement_var:N 3621, 3675, 3675
\l__regex_peek_true_tl 119, 120, 3492, 3546, 3566, 3572, 3625
__regex_pop_lr_states: 1921, 1953, 1961, 1968, 2061
__regex_posix_alnum: 248, 248
__regex_posix_alpha: 62, 248, 249, 250
__regex_posix_ascii: 248, 252
__regex_posix_blank: 248, 258
__regex_posix_cntrl: 248, 259
__regex_posix_digit: 248, 249, 266, 290
__regex_posix_graph: 248, 267
__regex_posix_lower: . 248, 251, 269
__regex_posix_print: 248, 271
__regex_posix_punct: 248, 273
__regex_posix_space: 248, 280
__regex_posix_upper: . 248, 251, 285
__regex_posix_word: 248, 287
__regex_posix_xdigit: 248, 288
__regex_prop_.: 45, 930
__regex_prop_d: . 45, 62, 219, 219, 266
__regex_prop_h: 219, 221, 258
__regex_prop_N: 219, 243, 958
__regex_prop_s: 219, 226
__regex_prop_v: 219, 234

__regex_prop_w: 219, 236, 287, 2239, 2241, 2242
__regex_push_lr_states: 1912, 1951, 1961, 1961, 2059
__regex_quark_if_nil:N 92
__regex_quark_if_nil:NTF 1383, 1403
__regex_quark_if_nil:nTF 92
__regex_quark_if_nil_p:n 92
__regex_query_range:nn 91, 120, 2562, 2568, 2568, 2587, 2659, 3409, 3446, 3628
__regex_query_range_loop:ww 2568, 2570, 2575, 2582
__regex_query_set:n 3130, 3130, 3196, 3205, 3231, 3398, 3428
__regex_query_set_aux:nN 3130, 3134, 3136, 3137, 3140
__regex_query_set_from_input_-tl: 3615, 3639, 3639
__regex_query_set_item:n 3639, 3643, 3644, 3646, 3649
__regex_query_submatch:n 2585, 2585, 2773, 3307, 3669, 3672
__regex_reinsert_item:n 119, 120, 3575, 3578, 3581, 3619, 3658
__regex_replace_all:nnN 3060, 3420, 3420
__regex_replace_all_aux:nnN 3096, 3421, 3422
__regex_replace_once:nnN 3058, 3387, 3387
__regex_replace_once_aux:nnN 3073, 3387, 3388, 3389
__regex_replacement:n . 120, 2610, 2610, 2654, 3075, 3388, 3421, 3622
__regex_replacement_apply:Nn 2610, 2611, 2612, 2689
__regex_replacement_balance_-one_match:n 90, 91, 2558, 2558, 2671, 3402, 3437
__regex_replacement_c:w . 2812, 2812
__regex_replacement_c_A:w 94, 2744, 2900, 2901
__regex_replacement_c_B:w 2732, 2903, 2904
__regex_replacement_c_C:w 2912, 2912
__regex_replacement_c_D:w 2739, 2917, 2918
__regex_replacement_c_E:w 2733, 2920, 2921
__regex_replacement_c_L:w 2742, 2929, 2930
__regex_replacement_c_M:w 2734, 2932, 2933
__regex_replacement_c_O:w 2731, 2736, 2740, 2743, 2745, 2935, 2936
__regex_replacement_c_P:w 2737, 2938, 2939
__regex_replacement_c_S:w 2727, 2741, 2944, 2944
__regex_replacement_c_T:w 2735, 2952, 2953
__regex_replacement_c_U:w 2738, 2955, 2956
__regex_replacement_cat:NNN 2817, 2860, 2860
\l__regex_replacement_category_-seq 2555, 2641, 2644, 2645, 2714, 2874
\l__regex_replacement_category_-tl 95, 2555, 2709, 2715, 2718, 2875, 2876
__regex_replacement_char:nNN 102, 2895, 2895, 2902, 2909, 2919, 2926, 2931, 2934, 2937, 2941, 2954, 2957
\l__regex_replacement_csnames_-int . 90, 2554, 2635, 2637, 2639, 2706, 2774, 2828, 2835, 2845, 2847, 2854, 2865, 2906, 2923, 3656, 3667
__regex_replacement_cu_aux:Nw 2822, 2826, 2826, 2840
__regex_replacement_do_one_-match:n 120, 121, 2560, 2560, 2657, 3407, 3445, 3626
__regex_replacement_error:NNN 2783, 2795, 2806, 2818, 2823, 2841, 2959, 2959
__regex_replacement_escaped:N 2631, 2750, 2750, 2879
__regex_replacement_exp_not:N 97, 2566, 2566, 2822, 2915, 3620
__regex_replacement_exp_not:n 2567, 2567, 2840, 3621
__regex_replacement_g:w . 2779, 2779
__regex_replacement_g_digits:NN 2779, 2782, 2785, 2792
__regex_replacement_lbrace:N 2624, 2781, 2821, 2839, 2852, 2852
__regex_replacement_normal:n 2626, 2632, 2704, 2704, 2757, 2787, 2814, 2849, 2857, 2872
__regex_replacement_normal_-aux:N 2704, 2710, 2724
__regex_replacement_put:n 2702, 2702, 2707, 2898, 2950, 3616
__regex_replacement_put_-submatch:n . 2755, 2761, 2761, 2802

__regex_replacement_put_-submatch_aux:n 2761, 2764, 2770, 3617
__regex_replacement_rbrace:N 2621, 2801, 2843, 2843
__regex_replacement_set:n 2610, 2611, 2655, 2692
\l__regex_replacement_tl 3494, 3607, 3622
__regex_replacement_u:w . 2837, 2837
__regex_return: 104, 3011, 3017, 3045, 3047, 3122, 3122
\l__regex_right_state_int 1865, 1889, 1933, 1934, 1954, 1966, 1973, 1982, 1983, 2022, 2029, 2035, 2048, 2056, 2106, 2110, 2121, 2135, 2144, 2156, 2160, 2164, 2167, 2172, 2175, 2178, 2186, 2200, 2203, 2206, 2209, 2213, 2229, 2272
\l__regex_right_state_seq 1865, 1932, 1942, 1965, 1972, 2108
\l__regex_saved_success_bool 83, 208, 215, 2302
__regex_show:N 103, 1704, 1704, 2990, 3002
__regex_show:NN 2985, 2995, 2996, 2997
__regex_show:Nn 2985, 2985, 2986, 2987
__regex_show_char:n 1736, 1740, 1743, 1747, 1756, 1769, 1769
__regex_show_class:NnnnN 1723, 1805, 1805
__regex_show_group_aux:nnnnN 1718, 1720, 1722, 1796, 1796
__regex_show_item_catcode:NnTF 1750, 1752, 1841, 1841
__regex_show_item_exact_cs:n 1757, 1854, 1854
\l__regex_show_lines_int 517, 1777, 1809, 1812, 1819
__regex_show_one:n 1712, 1725, 1728, 1736, 1739, 1743, 1746, 1756, 1760, 1775, 1775, 1791, 1798, 1802, 1815, 1831, 1859
__regex_show_pop: 1785, 1787, 1794, 1801
\l__regex_show_prefix_seq 516, 1710, 1713, 1761, 1781, 1786, 1788
__regex_show_push:n 1762, 1785, 1785, 1792, 1799, 1810
__regex_show_scope:nn 1754, 1759, 1785, 1789, 1846
__regex_single_match: 82, 205, 2379, 2379, 3155, 3192, 3392, 3548
__regex_split:nnN . . 3062, 3208, 3208
__regex_standard_escapechar: 4, 4, 301, 667, 1881, 1902
\l__regex_start_pos_int 2258, 2277, 2353, 2358, 2365, 3214, 3226, 3239, 3242, 3365, 3447
\g__regex_state_active_intarray 17, 71, 82, 83, 2294, 2335, 2441, 2444, 2452, 2479
\l__regex_step_int 17, 2291, 2337, 2403, 2442, 2446, 2454, 2468, 2470
__regex_store_state:n 81, 2363, 2493, 2496, 2496
__regex_store_submatches: . . . 2496
__regex_store_submatches:n . . 2515
__regex_store_submatches:nn 2498, 2502
__regex_submatch_balance:n 2559, 2591, 2591, 2674, 2776, 3296
\g__regex_submatch_begin_-intarray 17, 90, 113, 2564, 2588, 2605, 2666, 3115, 3221, 3224, 3237, 3378
\g__regex_submatch_case_intarray 2685, 3115, 3360, 3366
\g__regex_submatch_end_intarray 17, 113, 2589, 2598, 3115, 3218, 3234, 3381, 3411, 3630
\l__regex_submatch_int 17, 106, 109, 110, 115, 2344, 3112, 3233, 3235, 3238, 3240, 3243, 3256, 3355, 3359, 3361, 3362, 3436, 3444
\g__regex_submatch_prev_intarray 17, 106, 113, 2563, 2662, 3115, 3216, 3232, 3358, 3364
\g__regex_success_bool 83, 209, 211, 214, 2302, 2330, 2384, 2396, 3077, 3100, 3124, 3173, 3354, 3395, 3565, 3571, 3612
\l__regex_success_pos_int 2277, 2339, 2358, 2546, 3214
\l__regex_success_submatches_tl 81, 113, 2289, 2549, 3369
__regex_tests_action_cost:n 2000, 2002, 2015, 2021, 2030, 2048
\g__regex_thread_info_intarray 17, 80, 82, 88, 2294, 2434, 2505
__regex_tl_even_items:n 53, 53, 54, 3098
__regex_tl_even_items_loop:nn 53, 56, 59, 63
__regex_tl_odd_items:n 53, 53, 3074, 3097, 3171
__regex_tmp:w 110, 23, 25, 29, 31, 65, 65,

942, 952, 953, 954, 955, 956, 979, 990, 991, 3040, 3054, 3056, 3058, 3060, 3062, 3252, 3257, 3280, 3287, 3294, 3332, 3337, 3341, 3345, 3350
__regex_toks_clear:N . . . 7, 7, 1941, 1981
__regex_toks_memcpy:NNn . . . 12, 12, 2146
__regex_toks_put_left:Nn 21, 21, 1910, 1934, 1976, 2128, 2129
__regex_toks_put_right:Nn 19, 21, 27, 33, 1886, 1889, 1906, 1954, 1978, 1989, 2220, 2265
__regex_toks_set:Nn 7, 8, 9, 10, 3143, 3652
__regex_toks_use:w 6, 6, 2443, 2581, 4006
__regex_trace:nnn 3988, 3989, 3991, 3992, 4005
__regex_trace_pop:nnN . . . 3988, 3990
__regex_trace_push:nnN . . 3988, 3988
\g__regex_trace_regex_int 3998
__regex_trace_states:n . . 3999, 3999
__regex_two_if_eq:NNNN 518
__regex_two_if_eq:NNNNTF 518, 777, 824, 837, 871, 1041, 1078, 1098, 1099, 1168, 1203, 1220, 1221, 1283, 1416, 1423, 2872
__regex_use_i_delimit_by_q_recursion_stop:nw . . . 87, 89, 1406
__regex_use_none_delimit_by_q_nil:w 61, 87, 91
__regex_use_none_delimit_by_q_recursion_stop:w 87, 87, 1384, 1408, 3370
__regex_use_state: 2439, 2439, 2456, 2482
__regex_use_state_and_submatches:w 85, 2432, 2448, 2448
__regex_Z_test: 32, 972, 974, 991, 1611, 1732, 2217, 2244
\l__regex_zeroth_submatch_int 106, 113, 3112, 3217, 3219, 3222, 3225, 3355, 3365, 3367, 3379, 3382, 3403, 3408, 3412, 3627, 3631
reverse commands:
\reverse_if:N 109, 110, 127, 128, 133, 134

S

scan commands:
\scan_stop: 24, 32, 55, 193, 194, 307, 367, 392, 404, 540, 1376, 1694, 1698, 1701, 1856, 2458, 2897, 2949, 3252

seq commands:
\seq_clear:N 1761, 2645, 3279
\seq_count:N 2644
\seq_get:NN 2103, 2108
\seq_if_empty:NTF 2641
\seq_item:Nn 11
\seq_map_function:NN 1781, 1849
\seq_new:N . . 72, 516, 1867, 1868, 2556
\seq_pop:NN . . . 1932, 1970, 1972, 2714
\seq_pop_right:NN 1710, 1788
\seq_push:Nn . . 1942, 1963, 1965, 2874
\seq_put_right:Nn . . . 1713, 1786, 3289
\seq_set_filter:NNn 1844
\seq_set_map_e:NNn 1857
\seq_set_split:Nnn 1843, 1856
\seq_use:Nn 1860
str commands:
\c_backslash_str 319, 921
\c_left_brace_str 34, 382, 794, 798, 818, 831, 855, 1329, 1340, 1344, 1423, 1447, 2623
\c_right_brace_str 418, 804, 824, 837, 1347, 1351, 1444, 2620
\str_case:nn 1061
\str_case:nnTF 1671
\str_case_e:nnTF 802
\str_if_eq:nnTF 704, 3830, 3974
\str_map_break: 1570
\str_map_function:nN 1563
\str_map_inline:nn 2319
\str_range:nnn 1673

T

TeX and LaTeX 2_ε commands:
\afterassignment 112
\escapechar 18
\fontdimen 15
\lowercase 99–101
\newtoks 17
\toks 17–19, 68, 76, 81, 82, 91, 100, 107, 120, 121, 129
\uppercase 99
tex commands:
\tex_advance:D 1000
\tex_afterassignment:D 3263, 3327, 3469
\tex_catcode:D 2730
\tex_divide:D 1001
\tex_escapechar:D 5
\tex_lccode:D 2897, 2949
\tex_lowercase:D 2898, 2950
\tex_numexpr:D 3
\tex_the:D 6, 25, 31, 34
\tex_toks:D 6, 9, 11, 16, 24, 25, 30, 31, 34

tl commands:
\tl_analysis_map_inline:Nn . . . 1530
\tl_analysis_map_inline:nn *82*, *117*, *118*, 2309, 3135
\tl_build_begin:N *64*, 621, 1132, 1707, 1808, 2341, 2376, 2548, 2615, 3551
\tl_build_end:N *64*, 651, 659, 1142, 1764, 1827, 2649, 3577, 3641
\tl_build_get_intermediate:NN . 2367
\tl_build_put_right:Nn . . *94*, 628, 646, 654, 658, 722, 725, 765, 779, 783, 908, 922, 963, 985, 998, 1030, 1043, 1047, 1129, 1135, 1141, 1145, 1188, 1478, 1482, 1489, 1495, 1516, 1532, 1550, 1778, 1823, 1836, 2410, 2638, 2703, 2772, 2829, 2832, 2846, 2914, 3555, 3657, 3660, 3668, 3671
\tl_clear:N 297, 2342, 2377
\tl_const:Nn 75, 2983
\tl_count:n 1394, 3065, 3069, 3088, 3092, 3162, 3166
\tl_gclear:N 2617
\tl_gput_right:Nn 2647, 2698, 2699, 2775
\tl_gset:Nn 2681, 2690
\tl_gset_eq:NN 2687, 2978
\tl_if_blank:nTF 2321
\tl_if_empty:NTF 2709
\tl_if_empty:nTF *58*, 1474, 3323, 3485, 3967
\tl_if_head_eq_meaning:nNTF . . 1561
\tl_if_head_eq_meaning_p:nN . . 1393
\tl_if_in:nnTF 194
\tl_if_single:nTF 1555, 1591, 1606, 1639
\tl_if_single_token:nTF 695
\tl_item:nn *105*, 3030, 3075
\tl_map_break: *15*, *19*
\tl_map_function:NN 1518
\tl_map_function:nN 742
\tl_map_inline:nn *19*, 1915, 2369
\tl_map_tokens:nn 2688
\tl_new:N 66, 67, 73, 74, 2284, 2289, 2290, 2296, 2297, 2298, 2555, 2557, 2677, 2678, 3492, 3493, 3494, 3495
\tl_put_right:Nn 304
\tl_set:Nn 662, 1425, 1502, 1767, 1830, 2360, 2381, 2391, 2407, 2412, 2455, 2487, 2525, 2876, 3546, 3547, 3607
\tl_set_eq:NN 2549, 2973, 3465
\tl_tail:N 1368
\tl_to_str:N *98*
\tl_to_str:n *6*, *8*, *97*, *98*, 742, 1701, 2833, 2992, 3069, 3092, 3166
\l_tmpa_tl *13*
token commands:
\c_parameter_token *91*
\c_space_token 372, 413, 2726
\token_case_meaning:NnTF 1592, 1607, 1640, 1650, 1661
\token_if_eq_charcode:NNTF 413, 418, 1053, 1253, 1266, 1268, 1306, 1444, 2712, 2726
\token_if_eq_meaning:NNTF 749, 756, 1059, 1092, 1241, 1264, 1296, 1431, 1439, 1442, 2781, 2787, 2814, 2821, 2839, 2862, 2879
\token_to_meaning:N 698
\token_to_str:N 315, 322, 432, 436, 1171, 2707, 3004, 3068, 3091, 3165, 3785, 3989, 3991

U

use commands:
\use:N 636, 1470, 2718, 2889
\use:n *83*, *119*, *120*, 173, 370, 395, 597, 600, 740, 1106, 1270, 1715, 1781, 1849, 2301, 2356, 2413, 2473, 2670, 3623, 3682
\use:nn 3572
\use_i:nn 45, 557, 568, 577, 580, 589, 1476
\use_i:nnn 368, 393, 2791
\use_ii:nn . . . *58*, *66*, 47, 407, 559, 565, 570, 582, 591, 751, 1100, 1222, 1399, 1482, 1800, 3070, 3093, 3167
\use_ii:nnn 405, 705
\use_none:n 175, 445, 603, 1267, 1562, 1691, 2425
\use_none:nn 2093, 2804
\use_none:nnn 3283, 3558