The l3regex package: regular expressions in TEX*

The LATEX3 Project†

Released 2013/12/14

1 l3regex documentation

The l3regex package 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 add a comma at the end of each word:

```
\regex_replace_all:nnN { \w+ } { \0 , } \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).

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

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\c_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.

*This file describes v4623, last revised 2013/12/14.

†E-mail: latex-team@latex-project.org

1.1 Syntax of 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 will 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).

Character types.

`.` 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.*, alpha-numerics and underscore, equivalent to `[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` will 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`).

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.

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.

Alternation and capturing groups.

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

`(...)` Capturing group.

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

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

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, character property, 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.

`\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{`⟨*tl var name*⟩`}` matches the exact contents of the token list ⟨*tl var*⟩. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

The option `(?i)` makes the match case insensitive (identifying `A`–`Z` with `a`–`z`; no Unicode support yet). 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)[Y-\\]` is equivalent to `[YZ\[\\yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

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.

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.

1.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Escaped characters are supported as inside regular expressions. The whole match is accessed as `\0`, and the first 9 submatches are accessed as `\1`, ..., `\9`. Submatches with numbers higher than 9 are accessed as `\g{`⟨*number*⟩`}` instead.

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 characters inserted by the replacement have category code 12 (other) by default. The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cXY` Produces the character `Y` (which can be given as an escape sequence such as `\t` for tab) with category code `X`, which must be one of `CBEMTPUDSLOA`.

`\c{`⟨*text*⟩`}` Produces the control sequence with csname ⟨*text*⟩. The ⟨*text*⟩ may contain references to the submatches `\0`, `\1` *etc.*

1.3 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` `\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*⟩ will initially be such that it never matches.

`\regex_set:Nn` `\regex_gset:Nn` `\regex_const:Nn` `\regex_set:Nn` ⟨*regex var*⟩ {⟨*regex*⟩}

Stores a compiled version of the ⟨*regular expression*⟩ in the ⟨*regex var*⟩. 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) }
```

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which will never change.

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

`\regex_show:N` Shows how l3regex interprets the ⟨*regex*⟩. For instance, `\regex_show:n {\A X|Y}` shows

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

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

1.4 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_(g)set:Nn`.

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

`\regex_match:NnTF` 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:NnN

\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 ungreedy 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.

1.5 Submatch extraction

\regex_extract_once:nnNTF
\regex_extract_once:NnNTF

\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 zeroeth 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`) will match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` will contain the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream.

\regex_extract_all:nnNTF
\regex_extract_all:NnNTF

\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 will match twice, and the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

\regex_split:nnNTF
\regex_split:NnNTF

\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.

1.6 Replacement

\regex_replace_once:nnNTF
\regex_replace_once:NnNTF

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

\regex_replace_all:nnNTF
\regex_replace_all:NnNTF

\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 ⟨*token list*⟩ 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*⟩.

1.7 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Change user function names!
- Clean up the use of messages.
- Rewrite the documentation in a more ordered way, perhaps add a BNF?

Additional error-checking to come.

- Detect that a trailing \c⟨*category*⟩ is an invalid regex.
- Currently, a{\x34} is recognized as a{4}.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references.
- 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.
- Enforce that \cC can only be followed by a match-all dot.

Code improvements to come.

- Change \skip to \dimen for the array of active threads, and shift the array of submatch informations so that it starts at \skip0.
- Optimize \c{abc} for matching a specific control sequence.
- Only build .$_$. once.
- Use \skip 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.

- Improve digit grabbing for the `\g` escape in replacement. Allow arbitrary integer expressions for all those numbers?
- When matching, keep track of an explicit stack of `current_state` and `current_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.
- Use `\dimen` registers rather than `\l__regex_balance_tl` to build `\__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.
- Decide and document what `\c{\c{...}}` should do in the replacement text, similar questions for `\u`.
- 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.

- Allow `\cL(abc)` in replacement text.
- 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 will probably not be implemented.

- `\ddd`, matching the character with octal code `ddd`;
- Callout with `(?C...)`, we cannot run arbitrary user code during the matching, because the regex code uses registers in an unsafe way;
- 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.

- `\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.
- 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. Also, we cannot afford to run user code within the regular expression matching, because of our "misuse" of registers.
- Recursion: this is a non-regular feature.
- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\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.

2 l3regex implementation

```
⟨*initex | package⟩
⟨@@=regex⟩
⟨*package⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\RequirePackage{l3tl-build, l3tl-analysis, l3flag, l3str, l3str-convert}
⟨/package⟩
```

2.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (backreferences 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 roughly 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 $\langle position \rangle$, with $\texttt{min\_pos} - 1 \leq \langle position \rangle \leq \texttt{max\_pos}$. The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer $\langle state \rangle$ with $\texttt{min\_state} \leq \langle state \rangle < \texttt{max\_state}$.
- *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.

To achieve a good performance, we abuse TEX's registers in two ways. We access registers directly by number rather than tying them to control sequence using `\int_new:N` and other allocation functions. And we store integers in `\dimen` registers in scaled points (`sp`), using TEX's implicit conversion from dimensions to integers in some contexts. Specifically, the registers are used as follows. When compiling, `\toks` registers are used under the hood by functions from the l3tl-build module. When building,

- `\toks`⟨*state*⟩ holds the tests and actions to perform in the ⟨*state*⟩ of the NFA.
- (Not implemented yet.) `\skip`*i* has the form ⟨*group id*⟩ `plus` ⟨*left state*⟩ `minus` ⟨*right state*⟩.

When matching,

- `\dimen`⟨*state*⟩ is equal to the last ⟨*step*⟩ in which the ⟨*state*⟩ was active.
- (Currently, we use `\skip` instead of `\dimen`.) `\dimen`⟨*thread*⟩, with `min_active` ≤ ⟨*thread*⟩ < `max_active`, is equal to the ⟨*state*⟩ in which the ⟨*thread*⟩ currently is. The ⟨*threads*⟩ or ordered starting from the best to the least preferred.
- `\toks`⟨*thread*⟩ holds the submatch information for the ⟨*thread*⟩, as the contents of a property list.
- `\muskip`⟨*position*⟩ holds as its main and stretch components the character and category code of the token at this ⟨*position*⟩ in the query.
- `\toks`⟨*position*⟩ holds ⟨*tokens*⟩ which `o`- and `x`-expand to the ⟨*position*⟩-th token in the query.
- `\skip` registers hold the value of end-points of all submatches as would be extracted by the `\regex_extract` functions. Since smaller `\skip` registers are used, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `capturing_group`, each block corresponding to one match with all its submatches stored in consecutive `\skip`s.

`\count` registers are not abused, which means that we can safely use named integers in this module. Note that `\box` registers are not abused either; maybe we could leverage those for some purpose.

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.

2.2 Helpers

`\tl_to_str:V` A variant we need for the `\u` escape in the replacement text.

```
\cs_generate_variant:Nn \tl_to_str:n { V }
```

(*End definition for* `\tl_to_str:V`.)

2.2.1 Constants and variables

__regex_tmp:w Temporary function used for various short-term purposes.

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

(*End 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 definition for* `\l__regex_internal_a_tl` *and others. These variables are documented on page* **??**.)

\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 definition for* `\c__regex_no_match_regex`. *This variable is documented on page* **??**.)

\l__regex_balance_int The first thing we do when matching is to go once through the query token list and store the information for each token as `\muskip` and `\toks` registers. 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, and we store it as the shrink component of each `\muskip` register. This variable is also used to keep track of the balance in the replacement text.

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

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

2.2.2 Testing characters

__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 definition for* `\__regex_break_point:TF`.)

`\__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 will thus match `\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 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_current_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_current_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_current_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End 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 `current_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_current_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
    \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
      \__regex_compute_case_changed_char:
    \fi:
    \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_current_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_current_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
    \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
      \__regex_compute_case_changed_char:
    \fi:
    \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 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 (or rather, when it is set to the marker value `\c_max_int`). 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_nopar:Npn \__regex_compute_case_changed_char:
  {
    \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_current_char_int
    \if_int_compare:w \l__regex_current_char_int < \c_ninety_one
      \if_int_compare:w \l__regex_current_char_int < \c_sixty_five
      \else:
        \int_add:Nn \l__regex_case_changed_char_int { \c_thirty_two }
      \fi:
    \else:
      \if_int_compare:w \l__regex_current_char_int < \c_one_hundred_twenty_three
        \if_int_compare:w \l__regex_current_char_int < \c_ninety_seven
        \else:
          \int_sub:Nn \l__regex_case_changed_char_int { \c_thirty_two }
        \fi:
      \fi:
    \fi:
  }
```

(*End 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 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_current_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*\c_zero
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode:nT #1
  {
    \if_int_odd:w \__int_eval:w #1 / \__regex_item_catcode: \__int_eval_end:
      \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 definition for* `\__regex_item_catcode:nT` *and* `\__regex_item_catcode_reverse:nT`.)

__regex_item_exact:nn
__regex_item_exact_cs:c

This matches an exact ⟨*category*⟩-⟨*character code*⟩ pair, or an exact control sequence.

```
\cs_new_protected:Npn \__regex_item_exact:nn #1#2
  {
    \if_int_compare:w #1 = \l__regex_current_catcode_int
      \if_int_compare:w #2 = \l__regex_current_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:c #1
  {
    \int_compare:nNnTF \l__regex_current_catcode_int = \c_zero
      {
        \str_if_eq_x:nnTF
          {
            \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
            \tex_the:D \tex_toks:D \l__regex_current_pos_int
          }
          { #1 }
          { \__regex_break_true:w } { }
      }
      { }
  }
```

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

`\__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. The three `\exp_after:wN` expand the contents of the `\toks`⟨*current position*⟩ (of the form `\exp_not:n {`⟨*control sequence*⟩`}`) to ⟨*control sequence*⟩.

```
\cs_new_protected:Npn \__regex_item_cs:n #1
  {
    \int_compare:nNnT \l__regex_current_catcode_int = \c_zero
      {
        \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:Nx \__regex_match:n
            {
              \exp_after:wN \exp_after:wN
              \exp_after:wN \cs_to_str:N
              \tex_the:D \tex_toks:D \l__regex_current_pos_int
            }
          \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 definition for* `\__regex_item_cs:n`.)

2.2.3 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_nopar:Npn \__regex_prop_d:
  { \__regex_item_caseful_range:nn \c_forty_eight { 57 } } % 0--9
\cs_new_protected_nopar:Npn \__regex_prop_h:
  {
    \__regex_item_caseful_equal:n \c_thirty_two % space
    \__regex_item_caseful_equal:n \c_nine       % tab
  }
\cs_new_protected_nopar:Npn \__regex_prop_s:
  {
    \__regex_item_caseful_equal:n \c_thirty_two % space
    \__regex_item_caseful_equal:n \c_nine       % tab
    \__regex_item_caseful_equal:n \c_ten        % lf
```

```
    \__regex_item_caseful_equal:n \c_twelve     % ff
    \__regex_item_caseful_equal:n \c_thirteen   % cr
  }
\cs_new_protected_nopar:Npn \__regex_prop_v:
  { \__regex_item_caseful_range:nn \c_ten \c_thirteen } % lf, vtab, ff, cr
\cs_new_protected_nopar:Npn \__regex_prop_w:
  {
    \__regex_item_caseful_range:nn \c_ninety_seven { 122 } % a--z
    \__regex_item_caseful_range:nn \c_sixty_five   {  90 } % A--Z
    \__regex_item_caseful_range:nn \c_forty_eight  {  57 } % 0--9
    \__regex_item_caseful_equal:n  { 95 } % _
  }
\cs_new_protected_nopar:Npn \__regex_prop_N:
  { \__regex_item_reverse:n { \__regex_item_caseful_equal:n \c_ten } }
```

(*End 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_nopar:Npn \__regex_posix_alnum:
  { \__regex_posix_alpha: \__regex_posix_digit: }
\cs_new_protected_nopar:Npn \__regex_posix_alpha:
  { \__regex_posix_lower: \__regex_posix_upper: }
\cs_new_protected_nopar:Npn \__regex_posix_ascii:
  { \__regex_item_caseful_range:nn \c_zero \c_one_hundred_twenty_seven }
\cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\cs_new_protected_nopar:Npn \__regex_posix_cntrl:
  {
    \__regex_item_caseful_range:nn \c_zero { 31 }
    \__regex_item_caseful_equal:n \c_one_hundred_twenty_seven
  }
\cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
\cs_new_protected_nopar:Npn \__regex_posix_graph:
  { \__regex_item_caseful_range:nn { 33 } { 126 } }
\cs_new_protected_nopar:Npn \__regex_posix_lower:
  { \__regex_item_caseful_range:nn \c_ninety_seven { 122 } }
\cs_new_protected_nopar:Npn \__regex_posix_print:
  { \__regex_item_caseful_range:nn \c_thirty_two { 126 } }
\cs_new_protected_nopar:Npn \__regex_posix_punct:
  {
    \__regex_item_caseful_range:nn { 33 } { 47 }
    \__regex_item_caseful_range:nn { 58 } { 64 }
    \__regex_item_caseful_range:nn { 91 } { 96 }
    \__regex_item_caseful_range:nn { 123 } { 126 }
  }
\cs_new_protected_nopar:Npn \__regex_posix_space:
  {
    \__regex_item_caseful_equal:n \c_thirty_two
    \__regex_item_caseful_range:nn \c_nine \c_thirteen
  }
\cs_new_protected_nopar:Npn \__regex_posix_upper:
  { \__regex_item_caseful_range:nn \c_sixty_five { 90 } }
```

```
\cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
\cs_new_protected_nopar:Npn \__regex_posix_xdigit:
  {
    \__regex_posix_digit:
    \__regex_item_caseful_range:nn \c_sixty_five { 70 }
    \__regex_item_caseful_range:nn \c_ninety_seven { 102 }
  }
```

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

2.2.4 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 `x`-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 mostly done within an `x`-expanding assignment, except for the `\x` escape sequence, which is not amenable to that in general. For this, we use the general framework of `\__tl_build:Nw`.

`\__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Go through `#4` once, applying `#1`, `#2`, or `#3` as relevant to each character (after de-escaping it). Note that we cannot replace `\tl_set:Nx` and `\__tl_build_one:o` by a single call to `\__tl_build_one:x`, because the `x`-expanding assignment may be interrupted by `\x`.

```
\cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
  {
⟨trace⟩    \trace_push:nnn { regex } { 1 } { __regex_escape_use:nnnn }
    \__tl_build:Nw \l__regex_internal_a_tl
      \cs_set_nopar:Npn \__regex_escape_unescaped:N ##1 { #1 }
      \cs_set_nopar:Npn \__regex_escape_escaped:N ##1 { #2 }
      \cs_set_nopar:Npn \__regex_escape_raw:N ##1 { #3 }
      \int_set:Nn \tex_escapechar:D { 92 }
      \__str_gset_other:Nn \g__regex_internal_tl { #4 }
      \tl_set:Nx \l__regex_internal_b_tl
        {
          \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
          { break } \__prg_break_point:
        }
```

```
      \__tl_build_one:o \l__regex_internal_b_tl
    \__tl_build_end:
⟨trace⟩    \trace_pop:nnn { regex } { 1 } { __regex_escape_use:nnnn }
    \l__regex_internal_a_tl
  }
```

(*End 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_nopar: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 definition for* `\__regex_escape_loop:N`.)

`\__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 definition for* `\__regex_escape_unescaped:N`, `\__regex_escape_escaped:N`, *and* `\__regex_escape_raw:N`.)

`\__regex_escape_break:w` `\__regex_escape_/break: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:NN \__regex_escape_break:w \__prg_break:
\cs_new_nopar:cpn { __regex_escape_/break:w }
  {
    \if_false: { \fi: }
    \__msg_kernel_error:nn { regex } { trailing-backslash }
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
\cs_new_nopar:cpn { __regex_escape_~:w } { }
\cs_new_nopar:cpx { __regex_escape_/a:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
\cs_new_nopar:cpx { __regex_escape_/t:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
\cs_new_nopar:cpx { __regex_escape_/n:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
\cs_new_nopar:cpx { __regex_escape_/f:w }
```

```
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
\cs_new_nopar:cpx { __regex_escape_/r:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
\cs_new_nopar:cpx { __regex_escape_/e:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }
```

(*End definition for* `\__regex_escape_break: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 < 256, then it is turned into a byte and fed to `\__regex_escape_raw:N`. Otherwise, interrupt the assignment, and either produce an error, or use a standard `\lowercase` trick depending on the precise value.

```
\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_two_hundred_fifty_six
      {
        \exp_last_unbraced:Nf \__regex_escape_raw:N
          { \__str_output_byte:n {#1} }
      }
      { \__regex_escape_x_large:n {#1} }
  }
\group_begin:
  \char_set_catcode_other:n { 0 }
  \cs_new:Npn \__regex_escape_x_large:n #1
    {
      \if_false: { \fi: }
      \__tl_build_one:o \l__regex_internal_b_tl
      \int_compare:nNnTF {#1} > \c_max_char_int
        {
          \__msg_kernel_error:nnx { regex } { x-overflow } {#1}
          \tl_set:Nx \l__regex_internal_b_tl
            { \if_false: } \fi:
        }
        {
          \char_set_lccode:nn { \c_zero } {#1}
          \tl_to_lowercase:n
            {
              \tl_set:Nx \l__regex_internal_b_tl
                { \if_false: } \fi:
                \__regex_escape_raw:N ^^@
            }
        }
    }
\group_end:
```

(*End definition for* `\__regex_escape_/x:w`.)

`\__regex_escape_x_test: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
  {
    \str_if_eq_x:nnTF {#1} { break } { ; }
      {
        \if_charcode:w \c_space_token #1
          \exp_after:wN \__regex_escape_x_test:N
        \else:
          \exp_after:wN \__regex_escape_x_test_two:N
          \exp_after:wN #1
        \fi:
      }
  }
\cs_new:Npn \__regex_escape_x_test_two:N #1
  {
    \if_charcode:w \c_left_brace_str #1
      \exp_after:wN \__regex_escape_x_loop:N
    \else:
      \__str_hexadecimal_use:NTF #1
        { \exp_after:wN \__regex_escape_x:N }
        { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
    \fi:
  }
```

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

`\__regex_escape_x:N` This looks for the second digit in the unbraced case.

```
\cs_new:Npn \__regex_escape_x:N #1
  {
    \str_if_eq_x:nnTF {#1} { break } { ; }
      {
        \__str_hexadecimal_use:NTF #1
          { ; \__regex_escape_loop:N }
          { ; \__regex_escape_loop:N #1 }
      }
  }
```

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

`\__regex_escape_x_loop:N` 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
  {
    \__str_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 }
              {
                \if_false: { \fi: }
                \__tl_build_one:o \l__regex_internal_b_tl
                \__msg_kernel_error:nn { regex } { x-missing-rbrace } {#1}
                \tl_set:Nx \l__regex_internal_b_tl
                  { \if_false: } \fi: \__regex_escape_loop:N #1
              }
          }
      }
  }
```

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

`\__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 < \c_ninety_one
      \if_int_compare:w `#1 < \c_fifty_eight
        \if_int_compare:w `#1 < \c_forty_eight
          \if_int_compare:w `#1 < \c_thirty_two
            \prg_return_false: \else: \prg_return_true: \fi:
        \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w `#1 < \c_sixty_five
          \prg_return_true: \else: \prg_return_false: \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 < \c_one_hundred_twenty_three
        \if_int_compare:w `#1 < \c_ninety_seven
          \prg_return_true: \else: \prg_return_false: \fi:
```

```
      \else:
        \if_int_compare:w `#1 < \c_one_hundred_twenty_seven
          \prg_return_true: \else: \prg_return_false: \fi:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
  {
    \if_int_compare:w `#1 < \c_ninety_one
      \if_int_compare:w `#1 < \c_fifty_eight
        \if_int_compare:w `#1 < \c_forty_eight
          \prg_return_false: \else: \prg_return_true: \fi:
      \else:
        \if_int_compare:w `#1 < \c_sixty_five
          \prg_return_false: \else: \prg_return_true: \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 < \c_one_hundred_twenty_three
        \if_int_compare:w `#1 < \c_ninety_seven
          \prg_return_false: \else: \prg_return_true: \fi:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

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

2.3 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_anchor:N` ⟨*integer*⟩`}`

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:c {`⟨*csname*⟩`}`
- `\__regex_item_cs:n {`⟨*compiled regex*⟩`}`

2.3.1 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 definition for* `\l__regex_group_level_int`*. This variable is documented on page* ??.)

`\l__regex_mode_int` While compiling, ten modes are recognized, labelled −63, −23, −6, −2, 0, 2, 3, 6, 23, 63. See section 2.3.3.

```
\int_new:N \l__regex_mode_int
```

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

`\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 definition for* `\l__regex_catcodes_int` *and* `\l__regex_default_catcodes_int`*. These variables are documented on page* ??.)

`\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 definition for* `\c__regex_catcode_C_int` *and others. These variables are documented on page* **??**.)

`\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 definition for* `\l__regex_internal_regex`. *This variable is documented on page* **??**.)

`\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 definition for* `\l__regex_show_prefix_seq`. *This variable is documented on page* **??**.)

`\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 definition for* `\l__regex_show_lines_int`. *This variable is documented on page* **??**.)

2.3.2 Generic helpers used when compiling

`\__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 definition for* `\__regex_get_digits:NTFw`.)

`\__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 \c_one < 1 #2 \exp_stop_f:
```

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

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

2.3.3 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 −3,

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_nopar: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 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_nopar: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_zero
        \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 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_nopar: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_zero
        \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 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_nopar:Npn \__regex_if_within_catcode:TF
  {
    \if_int_compare:w \l__regex_mode_int > \c_zero
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End 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_zero
      \exp_after:wN \use:n
    \else:
      \if_int_compare:w \l__regex_mode_int = \c_three
        \exp_after:wN \exp_after:wN \exp_after:wN \use:n
      \else:
        \__msg_kernel_error:nn { regex } { c-bad-mode }
        \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
      \fi:
    \fi:
  }
```

(*End 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_two
      \l__regex_mode_int = \c_zero
    \else:
      \if_int_compare:w \l__regex_mode_int = \c_six
        \l__regex_mode_int = \c_three
      \fi:
    \fi:
  }
```

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

2.3.4 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 `x`-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_nopar:Npn \__regex_compile:w
  {
    \__tl_build_x:Nw \l__regex_internal_regex
      \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_nopar:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
      \cs_set_nopar:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      \__tl_build_one:n { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected_nopar:Npn \__regex_compile_end:
  {
      \__regex_if_in_class:TF
        {
          \__msg_kernel_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
        \__msg_kernel_error:nnx { regex } { missing-rparen }
          { \int_use:N \l__regex_group_level_int }
        \prg_replicate:nn
          { \l__regex_group_level_int }
          {
              \__tl_build_one:n
                {
                  \if_false: { \fi: }
                  \if_false: { \fi: } { 1 } { 0 } \c_true_bool
                }
            \__tl_build_end:
            \__tl_build_one:o \l__regex_internal_regex
          }
      \fi:
      \__tl_build_one:n { \if_false: { \fi: } }
    \__tl_build_end:
  }
```

(*End 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.

```
\cs_new_protected:Npn \__regex_compile:n #1
  {
    \__regex_compile:w
```

```
      \int_set:Nn \tex_escapechar:D { 92 }
      \int_set_eq:NN \l__regex_mode_int \c_zero
      \__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_zero
        {
          \__msg_kernel_error:nn { regex } { c-missing-rbrace }
          \__regex_compile_end:
          \__regex_compile_one:x
            { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
          \prg_do_nothing: \prg_do_nothing:
          \prg_do_nothing: \prg_do_nothing:
        }
    \__regex_compile_end:
  }
```

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

`\__regex_compile_one:x`

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:x #1
```

```
  {
    \__regex_mode_quit_c:
    \__regex_if_in_class:TF { }
      {
        \__tl_build_one:n
          { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
      }
    \__tl_build_one:x
      {
        \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 definition for* `\__regex_compile_one:x`.)

`\__regex_compile_abort_tokens:n` `\__regex_compile_abort_tokens:x` 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:x
      {
        \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 { x }
```

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

2.3.5 Quantifiers

`\__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 definition for* `\__regex_compile_quantifier:w`.)

__regex_compile_quantifier_none:
__regex_compile_quantifier_abort:xNN

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_one:n { \if_false: { \fi: } { 1 } { 0 } \c_false_bool } }
\cs_new_protected:Npn \__regex_compile_quantifier_abort:xNN #1#2#3
  {
    \__regex_compile_quantifier_none:
    \__msg_kernel_warning:nnxx { regex } { invalid-quantifier } {#1} {#3}
    \__regex_compile_abort_tokens:x {#1}
    #2 #3
  }
```

(*End definition for* `\__regex_compile_quantifier_none:`.)

__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
  {
    \str_if_eq:nnTF { #3 #4 } { \__regex_compile_special:N ? }
      { \__tl_build_one:n { \if_false: { \fi: } { #1 } { #2 } \c_true_bool } }
      {
        \__tl_build_one:n { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
        #3 #4
      }
  }
```

(*End 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_nopar:cpn { __regex_compile_quantifier_?:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
\cs_new_protected_nopar:cpn { __regex_compile_quantifier_*:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
\cs_new_protected_nopar:cpn { __regex_compile_quantifier_+:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(*End definition for* `\__regex_compile_quantifier_?:w`, `\__regex_compile_quantifier_*:w`, *and* `\__regex_compile_quantif`

__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:xNN { \c_left_brace_str } }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
  {
    \str_case_x:nnn { #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:xNN
          { \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
  {
    \str_if_eq_x:nnTF
      { #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:xNN
          { \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
  {
    \str_if_eq_x:nnTF
      { #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_kernel_error:nnxx { 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:xNN
          {
            \c_left_brace_str
            \int_use:N \l__regex_internal_a_int ,
            \int_use:N \l__regex_internal_b_int
          }
        #1 #2
      }
  }
```

(*End definition for* `\__regex_compile_quantifier_{:w.}`)

2.3.6 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_kernel_error:nnx { regex } { bad-escape } {#1}
    \__regex_compile_raw:N #1
  }
```

(*End 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
      {
        \str_if_eq:nnTF {#2#3} { \__regex_compile_special:N - }
          { \__regex_compile_range:Nw #1 }
          {
            \__regex_compile_one:x
              { \__regex_item_equal:n { \__int_value:w `#1 ~ } }
            #2 #3
          }
      }
      {
        \__regex_compile_one:x
          { \__regex_item_equal:n { \__int_value:w `#1 ~ } }
        #2 #3
      }
```

```
}
```

(*End 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_kernel_error:nnxx { regex } { range-backwards } {#1} {#3}
        \else:
          \__tl_build_one:x
            {
              \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_kernel_warning:nnxx { regex } { range-missing-end }
          {#1} { \c_backslash_str #3 }
        \__tl_build_one:x
          {
            \__regex_item_equal:n { \__int_value:w `#1 ~ }
            \__regex_item_equal:n { \__int_value:w `- ~ }
          }
        #2#3
```

```
      }
  }
```

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

2.3.7 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_nopar:cpx { __regex_compile_.: }
  {
    \exp_not:N \__regex_if_in_class:TF
      { \__regex_compile_raw:N . }
      { \__regex_compile_one:x \exp_not:c { __regex_prop_.: } }
  }
\cs_new_protected_nopar:cpn { __regex_prop_.: }
  {
    \if_int_compare:w \l__regex_current_char_int > - \c_two
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*End 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_nopar:cpx { __regex_compile_/#1: }
      { \__regex_compile_one:x \exp_not:c { __regex_prop_#1: } }
    \cs_new_protected_nopar:cpx { __regex_compile_/#2: }
      {
        \__regex_compile_one:x
          { \__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_nopar:cpn { __regex_compile_/N: }
  { \__regex_compile_one:x \__regex_prop_N: }
```

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

2.3.8 Anchoring and simple assertions

`\__regex_compile_anchor:NF` `\__regex_compile_^:` `\__regex_compile_/A:` `\__regex_compile_/G:` `\__regex_compile_$:` `\__regex_compile_/Z:` `\__regex_compile_/z:`

In modes where assertions are allowed, anchor to the start of the query, the start of the match, or the end of the query, depending on the integer `#1`. In other modes, `#2` treats

the character as raw, with an error for escaped letters (`$` is valid in a class, but `\A` is definitely a mistake on the user's part).

```
\cs_new_protected:Npn \__regex_compile_anchor:NF #1#2
  {
    \__regex_if_in_class_or_catcode:TF {#2}
      {
        \__tl_build_one:n
          { \__regex_assertion:Nn \c_true_bool { \__regex_anchor:N #1 } }
      }
  }
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected_nopar:cpn { __regex_compile_/#1: }
      { \__regex_compile_anchor:NF #2 { \__regex_compile_raw_error:N #1 } }
  }
\__regex_tmp:w A \l__regex_min_pos_int
\__regex_tmp:w G \l__regex_start_pos_int
\__regex_tmp:w Z \l__regex_max_pos_int
\__regex_tmp:w z \l__regex_max_pos_int
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected_nopar:cpn { __regex_compile_#1: }
      { \__regex_compile_anchor:NF #2 { \__regex_compile_raw:N #1 } }
  }
\exp_args:Nx \__regex_tmp:w { \iow_char:N \^ } \l__regex_min_pos_int
\exp_args:Nx \__regex_tmp:w { \iow_char:N \$ } \l__regex_max_pos_int
```

(*End definition for* `\__regex_compile_anchor:NF`.)

`\__regex_compile_/b:`
`\__regex_compile_/B:`

Contrarily to `^` and `$`, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

```
\cs_new_protected_nopar:cpn { __regex_compile_/b: }
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N b }
      {
        \__tl_build_one:n
          { \__regex_assertion:Nn \c_true_bool { \__regex_b_test: } }
      }
  }
\cs_new_protected_nopar:cpn { __regex_compile_/B: }
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N B }
      {
        \__tl_build_one:n
          { \__regex_assertion:Nn \c_false_bool { \__regex_b_test: } }
      }
  }
```

(*End definition for* `\__regex_compile_/b:` *and* `\__regex_compile_/B:`.)

2.3.9 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_sixteen
          \__tl_build_one:n { \if_false: { \fi: } }
        \fi:
        \tex_advance:D \l__regex_mode_int - \c_fifteen
        \tex_divide:D \l__regex_mode_int \c_thirteen
        \if_int_odd:w \l__regex_mode_int \else:
          \exp_after:wN \__regex_compile_quantifier:w
        \fi:
      }
      { \__regex_compile_raw:N ] }
  }
```

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

`\__regex_compile_class_normal:w` In the "normal" case, we will 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_nopar:Npn \__regex_compile_class_normal:w
```

```
  {
    \__regex_compile_class:TFNN
      { \__regex_class:NnnnN \c_true_bool }
      { \__regex_class:NnnnN \c_false_bool }
  }
```

(*End 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_two
      \__tl_build_one:n
        { \__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 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:
    \str_if_eq:nnTF { #3 #4 } { \__regex_compile_special:N ^ }
      {
        \__tl_build_one:n { #2 { \if_false: } \fi: }
        \__regex_compile_class:NN
      }
      {
        \__tl_build_one:n { #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 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.

```
\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_kernel_warning:nnx { regex } { posix-unsupported } { = } }
            . { \__msg_kernel_warning:nnx { regex } { posix-unsupported } { . } }
          }
      }
    \__regex_compile_raw:N [ #1 #2
  }
\cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
  {
    \str_if_eq:nnTF { #5 #6 } { \__regex_compile_special:N ^ }
      {
        \bool_set_false:N \l__regex_internal_bool
        \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w
      }
      {
        \bool_set_true:N \l__regex_internal_bool
        \tl_set:Nx \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
  {
    \str_if_eq:nnTF { #1 #2 #3 #4 }
      { \__regex_compile_special:N : \__regex_compile_special:N ] }
      {
        \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
          {
            \__regex_compile_one:x
              {
                \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
                \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
```

```
              }
          }
          {
            \__msg_kernel_warning:nnx { regex } { posix-unknown }
              { \l__regex_internal_a_tl }
            \__regex_compile_abort_tokens:x
              {
                [: \bool_if:NF \l__regex_internal_bool { ^ }
                \l__regex_internal_a_tl :]
              }
          }
      }
      {
        \__msg_kernel_error:nnxx { regex } { posix-missing-close }
          { [: \l__regex_internal_a_tl } { #2 #4 }
        \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl }
        #1 #2 #3 #4
      }
  }
```

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

2.3.10 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_internal_regex`, which ends up with items of the form `\__regex_branch:n {`⟨*concatenation*⟩`}`. This construction is done using l3tl-build 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_one:n { #1 { \if_false: } \fi: }
    \__regex_mode_quit_c:
    \__tl_build:Nw \l__regex_internal_regex
      \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
      \int_incr:N \l__regex_group_level_int
      \__tl_build_one:n { \__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
        \__tl_build_one:n { \if_false: { \fi: } }
      \__tl_build_end:
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \__tl_build_one:o \l__regex_internal_regex
      \exp_after:wN \__regex_compile_quantifier:w
    \else:
```

```
      \__msg_kernel_warning:nn { regex } { extra-rparen }
      \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
    \fi:
  }
```

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

`\__regex_compile_(:` In a class, parentheses are not special. Outside, check for a `?`, denoting special groups, and run the code for the corresponding special group.

```
\cs_new_protected_nopar:cpn { __regex_compile_(: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
      { \__regex_compile_lparen:w }
  }
\cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
  {
    \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ? }
      {
        \cs_if_exist_use:cF
          { __regex_compile_special_group_\token_to_str:N #4 :w }
          {
            \__msg_kernel_warning:nnx { regex } { special-group-unknown }
              { (? \token_to_str:N #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 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_nopar:cpn { __regex_compile_|: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N | }
      {
        \__tl_build_one:n
          { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
      }
  }
```

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

`\__regex_compile_):` Within a class, parentheses are not special. Outside, close a group.

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

```
      { \__regex_compile_group_end: }
  }
```
(*End 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 will come when building.

```
\cs_new_protected_nopar:cpn { __regex_compile_special_group_::w }
  { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
\cs_new_protected_nopar:cpn { __regex_compile_special_group_|:w }
  { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }
```
(*End definition for* `\__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
  {
    \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ) }
      {
        \cs_set_nopar:Npn \__regex_item_equal:n  { \__regex_item_caseless_equal:n }
        \cs_set_nopar:Npn \__regex_item_range:nn { \__regex_item_caseless_range:nn }
      }
      {
        \__msg_kernel_warning:nnx { regex } { unknown-option } { (?i #2 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N i
        #1 #2
      }
  }
\cs_new_protected_nopar:cpn { __regex_compile_special_group_-:w } #1#2#3#4
  {
    \str_if_eq:nnTF { #1 #2 #3 #4 }
      { \__regex_compile_raw:N i \__regex_compile_special:N ) }
      {
        \cs_set_nopar:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
        \cs_set_nopar:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      }
      {
        \__msg_kernel_warning:nnx { regex } { unknown-option } { (?-#2#4 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N -
        #1 #2 #3 #4
      }
  }
```
(*End definition for* `\__regex_compile_special_group_i:w` *and* `\__regex_compile_special_group_-:w`.)

2.3.11 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_two \else: \c_six \fi:
          }
      }
      { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
          {
            \__msg_kernel_error:nnx { regex } { c-missing-category } {#2}
            #1 #2
          }
  }
```

(*End definition for* __regex_compile_/c: *and* __regex_compile_c_test: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_two \else: \c_six \fi:
    \int_zero:N \l__regex_catcodes_int
    \str_if_eq:nnTF { #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_kernel_error:nnx { 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:w \l__regex_catcodes_int / #1 \__int_eval_end:
    \else:
      \tex_advance:D \l__regex_catcodes_int #1
    \fi:
  }
\cs_new_protected_nopar: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 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_nopar: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_two \else: \c_six \fi:
  }
```

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

`\__regex_compile_}:` 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 `\{`... Admittedly, that would be better done as `\c{[{}]}`. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex.

```
\cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
  {
    \__regex_if_in_cs:TF
      {
        \__regex_compile_end:
        \__regex_compile_one:x
          { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
      }
      { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
  }
```

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

2.3.12 Raw token lists with \u

`\__regex_compile_/u:`
`\__regex_compile_u_loop:NN`

The `\u` escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then collect the characters for the argument of `\u` within an `x`-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace is missing, then we will 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_protected:cpn { __regex_compile_/u: } #1#2
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N u #1 #2 }
      {
        \str_if_eq_x:nnTF {#1#2} { \__regex_compile_special:N \c_left_brace_str }
          {
            \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
            \__regex_compile_u_loop:NN
          }
          {
            \__msg_kernel_error:nn { regex } { u-missing-lbrace }
            \__regex_compile_raw:N u #1 #2
          }
      }
  }
\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: } \__regex_compile_u_end: }
              { #2 \__regex_compile_u_loop:NN }
          }
          {
```

```
            \if_false: { \fi: }
            \__msg_kernel_error:nnx { regex } { u-missing-rbrace } {#2}
            \__regex_compile_u_end:
            #1 #2
          }
      }
  }
```

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

`\__regex_compile_u_end:` Once we have extracted the variable's name, we store the contents of that variable 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:
  {
    \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
    \if_int_compare:w \l__regex_mode_int = \c_zero
      \__regex_compile_u_not_cs:
    \else:
      \__regex_compile_u_in_cs:
    \fi:
  }
```

(*End definition for* `\__regex_compile_u_end:.`)

`\__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:
  {
    \exp_args:NNo \__str_gset_other:Nn \g__regex_internal_tl
      { \l__regex_internal_a_tl }
    \__tl_build_one:x
      {
        \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 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:
```

```
  {
    \exp_args:No \__tl_analysis_map_inline:nn { \l__regex_internal_a_tl }
      {
        \__tl_build_one:n
          {
            \__regex_class:NnnnN \c_true_bool
              {
                \if_int_compare:w "##2 = \c_zero
                  \__regex_item_exact_cs:c { \exp_after:wN \cs_to_str:N ##1 }
                \else:
                  \__regex_item_exact:nn { \__int_value:w "##2 } { ##3 }
                \fi:
              }
              { 1 } { 0 } \c_false_bool
          }
      }
  }
```

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

2.3.13 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_nopar:cpn { __regex_compile_/K: }
  {
    \int_compare:nNnTF \l__regex_mode_int = \c_zero
      { \__tl_build_one:n { \__regex_command_K: } }
      { \__regex_compile_raw_error:N K }
  }
```

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

2.3.14 Showing regexes

`\__regex_show:Nx` Within a `\__tl_build:Nw` ... `\__tl_build_end:` group, we redefine all the function that can appear in a compiled regex, then run the regex. The result is then shown.

```
\cs_new_protected:Npn \__regex_show:Nx #1#2
  {
    \__tl_build:Nw \l__regex_internal_a_tl
      \cs_set_protected_nopar: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_nopar:Npn \__regex_group:nnnN
        { \__regex_show_group_aux:nnnnN { } }
      \cs_set_protected_nopar:Npn \__regex_group_no_capture:nnnN
```

```
      { \__regex_show_group_aux:nnnnN { ~(no~capture) } }
    \cs_set_protected_nopar: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_nopar: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_nopar:Npn \__regex_b_test: { word~boundary }
    \cs_set_eq:NN \__regex_anchor:N \__regex_show_anchor_to_str:N
    \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
      { \__regex_show_one:n { char~code~\int_eval:n{##1} } }
    \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
      { \__regex_show_one:n { range~[\int_eval:n{##1}, \int_eval:n{##2}] } }
    \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
      { \__regex_show_one:n { char~code~\int_eval:n{##1}~(caseless) } }
    \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
      {
        \__regex_show_one:n
          { Range~[\int_eval:n{##1}, \int_eval: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~##2,~catcode~##1 } }
    \cs_set_protected:Npn \__regex_item_exact_cs:c ##1
      { \__regex_show_one:n { control~sequence~\iow_char:N\\##1 } }
    \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 { ~ }
    #1
  \__tl_build_end:
  \__msg_show_variable:n { >~Compiled~regex~#2: \l__regex_internal_a_tl }
}
```

(*End definition for* `\__regex_show:Nx`.)

`\__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_one:x
      {
```

```
        \exp_not:N \\
        \seq_map_function:NN \l__regex_show_prefix_seq \use:n
        #1
      }
  }
```

(*End 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:Nx \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 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 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
  {
    \__tl_build:Nw \l__regex_internal_a_tl
      \int_zero:N \l__regex_show_lines_int
      \__regex_show_push:n {~}
      #2
```

```
      \exp_last_unbraced:Nf
    \int_case:nnF { \l__regex_show_lines_int }
      {
        {0}
          {
            \__tl_build_end:
            \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
          }
        {1}
          {
            \__tl_build_end:
            \bool_if:NTF #1
              {
                #2
                \__tl_build_one:n { \__regex_msg_repeated:nnN {#3} {#4} #5 }
              }
              {
                \__regex_show_one:n
                  { Don't~match~\__regex_msg_repeated:nnN {#3} {#4} #5 }
                \__tl_build_one:o \l__regex_internal_a_tl
              }
          }
      }
      {
        \__tl_build_end:
        \__regex_show_one:n
          {
            \bool_if:NTF #1 { M } { Don't~m } atch
            \__regex_msg_repeated:nnN {#3} {#4} #5
          }
        \__tl_build_one:o \l__regex_internal_a_tl
      }
  }
```

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

`\__regex_show_anchor_to_str:N` The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```
\cs_new:Npn \__regex_show_anchor_to_str:N #1
  {
    anchor~at~
    \str_case:nnF { #1 }
      {
        { \l__regex_min_pos_int   } { start~(\iow_char:N\\A) }
        { \l__regex_start_pos_int } { start~of~match~(\iow_char:N\\G) }
        { \l__regex_max_pos_int   } { end~(\iow_char:N\\Z) }
      }
      { <error:~'#1'~not~recognized> }
  }
```

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

__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 definition for* `\__regex_show_item_catcode:NnT`.)

2.4 Building

2.4.1 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 always 0, but is included to avoid hard-coding this value.

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

(*End definition for* `\l__regex_min_state_int` *and* `\l__regex_max_state_int`. *These variables are documented on page* **??**.)

`\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 definition for* `\l__regex_left_state_int` *and* `\l__regex_right_state_int`. *These variables are documented on page* **??**.)

\l__regex_capturing_group_int `\l__regex_capturing_group_int` is the ID number that will be assigned to a capturing group if one was opened now. 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 definition for* `\l__regex_capturing_group_int`. *This variable is documented on page* **??**.)

2.4.2 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:` inserted at the start of the regular expression to make 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:n {`⟨*key*⟩`}` where the ⟨*key*⟩ is a group number followed by `<` or `>` for the beginning or end of group. This causes the current position in the query to be stored as the ⟨*key*⟩ submatch boundary.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group` − 1, and if a group is opened now, it will 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:N`

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, which will be numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful.

```
\cs_new_protected:Npn \__regex_build:n #1
  {
    \__regex_compile:n {#1}
    \__regex_build:N \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_build:N #1
  {
```

```
⟨trace⟩    \trace_push:nnn { regex } { 1 } { __regex_build }
    \int_set:Nn \tex_escapechar:D { 92 }
    \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: }
    \__regex_group:nnnN {#1} { 1 } { 0 } \c_false_bool
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      { \__regex_action_success: }
⟨trace⟩    \__regex_trace_states:n { 2 }
⟨trace⟩    \trace_pop:nnn { regex } { 1 } { __regex_build }
  }
```

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

`\__regex_build_for_cs:n` 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
  {
⟨trace⟩    \trace_push:nnn { regex } { 1 } { __regex_build_for_cs }
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_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 \l__regex_current_pos_int = \l__regex_max_pos_int
          \exp_after:wN \__regex_action_success:
        \fi:
      }
⟨trace⟩    \__regex_trace_states:n { 2 }
⟨trace⟩    \trace_pop:nnn { regex } { 1 } { __regex_build_for_cs }
  }
```

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

2.4.3 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_nopar: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_nopar: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 definition for* `\__regex_push_lr_states:` *and* `\__regex_pop_lr_states:`.)

`\__regex_toks_put_left:Nx`
`\__regex_toks_put_right:Nx`
`\__regex_toks_put_right:Nn`

During the building phase we wish to add `x`-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:Nx` is provided because it is more efficient than `x`-expanding with `\exp_not:n`.

```
\cs_new_protected:Npn \__regex_toks_put_left:Nx #1#2
  {
    \cs_set_nopar:Npx \__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:Nx #1#2
  {
    \cs_set_nopar:Npx \__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 definition for* `\__regex_toks_put_left:Nx`.)

`\__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:Nx  #2 { #1 { \int_eval:n { #3 - #2 } } } }
\cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
  { \__regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
```

(*End 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_nopar:Npn \__regex_build_new_state:
  {
⟨*trace⟩
    \trace:nnx { regex } { 2 }
```

```
      {
        regex~new~state~
        L=\int_use:N \l__regex_left_state_int ~ -> ~
        R=\int_use:N \l__regex_right_state_int ~ -> ~
        M=\int_use:N \l__regex_max_state_int ~ -> ~
        \int_eval:n { \l__regex_max_state_int + \c_one }
      }
⟨/trace⟩
    \tex_toks:D \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 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:Nx \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 definition for* `\__regex_build_transitions_lazyness:NNNNN`.)

2.4.4 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_nopar:Npx \__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 definition for* `\__regex_class:NnnnN`.)

`\__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 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
      \__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 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 - \c_one }
    \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 definition for* `\__regex_class_repeat:nnN`.)

2.4.5 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 will 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
  {
⟨trace⟩        \trace_push:nnn { regex } { 1 } { __regex_group }
      \if_int_compare:w #3 = \c_zero
        \__regex_build_new_state:
⟨assert⟩\assert_int:n { \l__regex_max_state_int = \l__regex_right_state_int + 1 }
        \__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:
⟨trace⟩       \trace_pop:nnn { regex } { 1 } { __regex_group }
  }
```

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

`\__regex_group:nnnN` Hand to `\__regex_group_aux:nnnnnN` the label of that group (expanded), and the group
`\__regex_group_no_capture:nnnN` 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_nopar:Npn \__regex_group_no_capture:nnnN
  { \__regex_group_aux:nnnnN { -1 } }
```

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

`\__regex_group_resetting:nnnN` Again, hand the label -1 to `\__regex_group_aux:nnnnN`, but this time we work a little
`\__regex_group_resetting_loop:nnNn` 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 definition for* `\__regex_group_resetting: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
  {
⟨trace⟩    \trace_push:nnn { regex } { 1 } { __regex_branch }
    \__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
⟨trace⟩    \trace_pop:nnn { regex } { 1 } { __regex_branch }
  }
```

(*End 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_set:Nn \l__regex_max_state_int
        { \l__regex_left_state_int - \c_one }
      \__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 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_minus_one
      \__regex_toks_put_left:Nx #2 { \__regex_action_submatch:n { #1 < } }
      \__regex_toks_put_left:Nx #3 { \__regex_action_submatch:n { #1 > } }
    \fi:
  }
```

(*End 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 will "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:w #1 > \c_one
      \int_set:Nn \l__regex_internal_c_int
        {
          ( #1 - \c_one )
          * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
        }
      \tex_advance:D \l__regex_right_state_int \l__regex_internal_c_int
      \tex_advance:D \l__regex_max_state_int   \l__regex_internal_c_int
      \prg_replicate:nn \l__regex_internal_c_int
        {
          \tex_toks:D \l__regex_internal_b_int
            = \tex_toks:D \l__regex_internal_a_int
          \tex_advance:D \l__regex_internal_a_int \c_one
          \tex_advance:D \l__regex_internal_b_int \c_one
        }
    \fi:
  }
```

(*End 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
      \__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 - \c_one }
    \__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 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 - \c_one }
        {
          \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_set:Nn \l__regex_right_state_int
          { \l__regex_left_state_int - \c_one }
      \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 definition for* `\__regex_group_repeat:nnnN`.)

2.4.6 Others

`\__regex_assertion:Nn` `\__regex_b_test:` `\__regex_anchor:N`

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. Anchors at the start or end of match use `\__regex_anchor:N`, with a position controlled by the integer `#1`.

```
\cs_new_protected:Npn \__regex_assertion:Nn #1#2
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Nx \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_anchor:N #1
  {
    \if_int_compare:w #1 = \l__regex_current_pos_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected_nopar:Npn \__regex_b_test:
  {
    \group_begin:
      \int_set_eq:NN \l__regex_current_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: }
  }
```

(*End definition for* `\__regex_assertion:Nn`, `\__regex_b_test:`, *and* `\__regex_anchor:N`.)

`\__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_nopar:Npn \__regex_command_K:
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Nx \l__regex_left_state_int
      {
        \__regex_action_submatch:n { 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 definition for* `\__regex_command_K:`.)

2.5 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 `\skip` registers: this thread will be 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 the future execution will 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 will keep 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.

2.5.1 Variables used when matching

`\l__regex_min_pos_int`
`\l__regex_max_pos_int`
`\l__regex_current_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 `\muskip` and `\toks` registers with those numbers. We don't start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the `current_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_current_pos_int
\int_new:N \l__regex_start_pos_int
\int_new:N \l__regex_success_pos_int
```

(*End definition for* `\l__regex_min_pos_int` *and others. These variables are documented on page* ??.)

`\l__regex_current_char_int`
`\l__regex_current_catcode_int`
`\l__regex_last_char_int`
`\l__regex_case_changed_char_int`

The character and category codes of the token at the current position; the character code of the token at the previous position; 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 `current_char` variable is also used in various other phases to hold a character code.

```
\int_new:N \l__regex_current_char_int
\int_new:N \l__regex_current_catcode_int
\int_new:N \l__regex_last_char_int
\int_new:N \l__regex_case_changed_char_int
```

(*End definition for* `\l__regex_current_char_int` *and others. These variables are documented on page* ??.)

\l__regex_current_state_int

For every character in the token list, each of the active states is considered in turn. The variable \l__regex_current_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_current_state_int
```

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

\l__regex_current_submatches_prop
\l__regex_success_submatches_prop

The submatches for the thread which is currently active are stored in the current_submatches property list variable. This property list is stored by __regex_action_cost:n into the \toks register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to \l__regex_success_submatches_prop: only the last successful thread will remain there.

```
\prop_new:N \l__regex_current_submatches_prop
\prop_new:N \l__regex_success_submatches_prop
```

(*End definition for* \l__regex_current_submatches_prop *and* \l__regex_success_submatches_prop*. These variables are documented on page* ??.)

\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. For each ⟨*state*⟩ in the NFA we store in \dimen⟨*state*⟩ the last step in which this state was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, \dimen⟨*state*⟩ is equal step when we have started performing the operations of \toks⟨*state*⟩, but not finished yet. However, once we finish, we set \dimen⟨*state*⟩ to step + 1. 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 definition for* \l__regex_step_int*. This variable is documented on page* ??.)

\l__regex_min_active_int
\l__regex_max_active_int

All the currently active states are kept in order of precedence in the \skip registers, and the corresponding submatches in the \toks. For our purposes, those serve as an array, indexed from min_active (inclusive) to max_active (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and max_active is reset to min_active, effectively clearing the array.

```
\int_new:N \l__regex_min_active_int
\int_new:N \l__regex_max_active_int
```

(*End definition for* \l__regex_min_active_int *and* \l__regex_max_active_int*. These variables are documented on page* ??.)

\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 definition for* \l__regex_every_match_tl*. This variable is documented on page* ??.)

\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 definition for* `\l__regex_fresh_thread_bool` *and* `\l__regex_empty_success_bool`. *These variables are documented on page* ??.)

\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 definition for* `\g__regex_success_bool`, `\l__regex_saved_success_bool`, *and* `\l__regex_match_success_bool`. *These variables are documented on page* ??.)

2.5.2 Matching: framework

__regex_match:n

First store the query into `\toks` and `\muskip` registers (see `\__regex_query_set:nnn`). Then 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 (`\dimen` registers), 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
  {
⟨trace⟩    \trace_push:nnx { regex } { 1 } { __regex_match }
⟨trace⟩    \trace:nnx { regex } { 1 } { analyzing~query~token~list }
    \int_zero:N \l__regex_balance_int
    \int_set:Nn \l__regex_current_pos_int { \c_two * \l__regex_max_state_int }
    \__regex_query_set:nnn { } { -1 } { -2 }
    \int_set_eq:NN \l__regex_min_pos_int \l__regex_current_pos_int
    \__tl_analysis_map_inline:nn {#1}
      { \__regex_query_set:nnn {##1} {"##2} {##3} }
```

```
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_current_pos_int
    \__regex_query_set:nnn { } { -1 } { -2 }
⟨trace⟩    \trace:nnx { regex } { 1 } { initializing }
    \bool_gset_false:N \g__regex_success_bool
    \int_step_inline:nnnn
      \l__regex_min_state_int \c_one { \l__regex_max_state_int - \c_one }
      { \tex_dimen:D ##1 \c_one sp \scan_stop: }
    \int_set_eq:NN \l__regex_min_active_int \l__regex_max_state_int
    \int_set_eq:NN \l__regex_step_int \c_zero
    \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
    \int_set:Nn \l__regex_submatch_int
      { \c_two * \l__regex_max_state_int }
    \bool_set_false:N \l__regex_empty_success_bool
    \__regex_match_once:
⟨trace⟩    \trace_pop:nnx { regex } { 1 } { __regex_match }
  }
```

(*End definition for* `\__regex_match:n`.)

`\__regex_match_once:` This function finds one match, then does some action defined by the `every_match` token list, which may recursively call `\__regex_match_once:`. 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, and `get` that token, so that the `last_char` will be set properly for word boundaries. Then call `\__regex_match_loop:`, which runs through the query until the end or until a successful match breaks early.

```
\cs_new_protected_nopar:Npn \__regex_match_once:
  {
    \if_meaning:w \c_true_bool \l__regex_empty_success_bool
      \cs_set_nopar:Npn \__regex_if_two_empty_matches:F
        { \int_compare:nNnF \l__regex_start_pos_int = \l__regex_current_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
    \prop_clear:N \l__regex_current_submatches_prop
    \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
    \__regex_store_state:n { \l__regex_min_state_int }
    \int_set:Nn \l__regex_current_pos_int
      { \l__regex_start_pos_int - \c_one }
    \__regex_query_get:
    \__regex_match_loop:
    \l__regex_every_match_tl
  }
```

(*End definition for* `\__regex_match_once:`.)

`\__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_nopar: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 }
  }
\cs_new_protected:Npn \__regex_multi_match:n #1
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \if_meaning:w \c_true_bool \l__regex_match_success_bool
          \bool_gset_true:N \g__regex_success_bool
          #1
          \exp_after:wN \__regex_match_once:
        \fi:
      }
  }
```

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

`\__regex_match_loop:`
`\__regex_match_one_active:w`

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_active`). This results in a sequence of `\__regex_use_state_and_submatches:nn {`⟨*state*⟩`} {`⟨*prop*⟩`}`, 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 what `\__regex_match_once:` matches. We explain the `fresh_thread` business when describing `\__regex_action_wildcard:`.

```
\cs_new_protected_nopar:Npn \__regex_match_loop:
  {
    \tex_advance:D \l__regex_step_int \c_two
    \int_incr:N \l__regex_current_pos_int
    \int_set_eq:NN \l__regex_last_char_int \l__regex_current_char_int
    \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
    \__regex_query_get:
    \use:x
      {
        \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
        \exp_after:wN \__regex_match_one_active:w
          \int_use:N \l__regex_min_active_int ;
      }
    \__prg_break_point:
    \bool_set_false:N \l__regex_fresh_thread_bool %^^A was arg of break_point:n
    \if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
      \if_int_compare:w \l__regex_current_pos_int < \l__regex_max_pos_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_match_loop:
      \fi:
```

```
    \fi:
  }
\cs_new:Npn \__regex_match_one_active:w #1;
  {
    \if_int_compare:w #1 < \l__regex_max_active_int
      \__regex_use_state_and_submatches:nn
        { \__int_value:w \tex_skip:D #1 }
        { \tex_the:D \tex_toks:D #1 }
      \exp_after:wN \__regex_match_one_active:w
        \int_use:N \__int_eval:w #1 + \c_one \exp_after:wN ;
    \fi:
  }
```

(*End definition for* `\__regex_match_loop:`.)

`\__regex_query_set:nnn` The arguments are: tokens that `o` and `x` expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a `\muskip` register and a `\toks`, then update the `balance`.

```
\cs_new_protected:Npn \__regex_query_set:nnn #1#2#3
  {
    \tex_muskip:D \l__regex_current_pos_int
      = \etex_gluetomu:D
        #3 sp
        plus #2 sp
        minus \l__regex_balance_int sp
      \scan_stop:
    \tex_toks:D \l__regex_current_pos_int {#1}
    \int_incr:N \l__regex_current_pos_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 definition for* `\__regex_query_set:nnn`.)

`\__regex_query_get:` Extract the current character and category codes from the `\muskip` register of the current position: those are the main and the stretch components, and we need a conversion to avoid TEX's "incompatible glue units" error.

```
\cs_new_protected_nopar:Npn \__regex_query_get:
  {
    \l__regex_current_char_int
      = \etex_mutoglue:D \tex_muskip:D \l__regex_current_pos_int
    \l__regex_current_catcode_int = \etex_gluestretch:D
      \etex_mutoglue:D \tex_muskip:D \l__regex_current_pos_int
  }
```

(*End definition for* `\__regex_query_get:`.)

2.5.3 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_nopar:Npn \__regex_use_state:
  {
⟨*trace⟩
    \trace:nnx { regex } { 2 } { state~\int_use:N \l__regex_current_state_int }
⟨/trace⟩
    \tex_dimen:D \l__regex_current_state_int
      = \l__regex_step_int sp \scan_stop:
    \tex_the:D \tex_toks:D \l__regex_current_state_int
    \tex_dimen:D \l__regex_current_state_int
      = \__int_eval:w \l__regex_step_int + \c_one \__int_eval_end: sp \scan_stop:
  }
```

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

__regex_use_state_and_submatches:nn 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 `current_state` and `current_submatches` and use the state if it has not yet been encountered at this step.

```
\cs_new_protected:Npn \__regex_use_state_and_submatches:nn #1 #2
  {
    \int_set:Nn \l__regex_current_state_int {#1}
    \if_int_compare:w \tex_dimen:D \l__regex_current_state_int
                      < \l__regex_step_int
      \tl_set:Nn \l__regex_current_submatches_prop {#2}
      \exp_after:wN \__regex_use_state:
    \fi:
    \scan_stop:
  }
```

(*End definition for* `\__regex_use_state_and_submatches:nn`.)

2.5.4 Actions when matching

__regex_action_start_wildcard: 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_loop:` too.

```
\cs_new_protected_nopar:Npn \__regex_action_start_wildcard:
  {
    \bool_set_true:N \l__regex_fresh_thread_bool
    \__regex_action_free:n {1}
    \bool_set_false:N \l__regex_fresh_thread_bool
    \__regex_action_cost:n {0}
  }
```

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

`\__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_current_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 when within the thread itself.

```
\cs_new_protected_nopar:Npn \__regex_action_free:n
  { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
\cs_new_protected_nopar: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:x
      {
        \int_add:Nn \l__regex_current_state_int {#2}
        \exp_not:n
          {
            \if_int_compare:w \tex_dimen:D \l__regex_current_state_int #1
              \exp_after:wN \__regex_use_state:
            \fi:
          }
        \int_set:Nn \l__regex_current_state_int
          { \int_use:N \l__regex_current_state_int }
        \tl_set:Nn \exp_not:N \l__regex_current_submatches_prop
          { \exp_not:o \l__regex_current_submatches_prop }
      }
  }
```

(*End definition for* `\__regex_action_free:n` *and* `\__regex_action_free_group:n`.)

`\__regex_action_cost:n`

A transition which consumes the current character and shifts the state by `#1`. The resulting state is stored in the `\skip` 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:No \__regex_store_state:n
      { \int_use:N \__int_eval:w \l__regex_current_state_int + #1 }
  }
```

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

`\__regex_store_state:n`
`\__regex_store_submatches:`

Put the given state in the array of `\skip` registers (converted to a dimension in scaled points), and increment the length of the array. Then store the current submatch in the This is done by increasing the pointer `\l__regex_max_active_int`, and converting the integer to a dimension (suitable for a `\skip` assignment) in scaled points.

```
\cs_new_protected:Npn \__regex_store_state:n #1
  {
```

```
    \__regex_store_submatches:
    \tex_skip:D \l__regex_max_active_int = #1 sp \scan_stop:
    \int_incr:N \l__regex_max_active_int
  }
\cs_new_protected_nopar:Npn \__regex_store_submatches:
  {
    \tex_toks:D \l__regex_max_active_int \exp_after:wN
      { \l__regex_current_submatches_prop }
  }
```

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

`\__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_nopar:Npn \__regex_disable_submatches:
  {
    \cs_set_protected_nopar:Npn \__regex_store_submatches: { }
    \cs_set_protected:Npn \__regex_action_submatch:n ##1 { }
  }
```

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

`\__regex_action_submatch:n` Update the current submatches with the information from the current position. Maybe a bottleneck.

```
\cs_new_protected:Npn \__regex_action_submatch:n #1
  {
    \prop_put:Nno \l__regex_current_submatches_prop {#1}
      { \int_use:N \l__regex_current_pos_int }
  }
```

(*End definition for* `\__regex_action_submatch:n`.)

`\__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_nopar: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_current_pos_int
        \prop_set_eq:NN \l__regex_success_submatches_prop
          \l__regex_current_submatches_prop
        \__prg_break:
      }
  }
```

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

2.6 Replacement

2.6.1 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 definition for* `\l__regex_replacement_csnames_int`. *This variable is documented on page* ??.)

`\l__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 \l__regex_balance_tl
```

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

`\__regex_replacement_balance_one_match:n`

This expects as an argument the first index of a range of `\skip` registers 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 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, will produce the fully replaced token list. The initialization does not matter, but we set it as for an empty replacement.

```
\cs_new:Npn \__regex_replacement_do_one_match:n #1
  {
    \__regex_query_range:nn
      { \etex_glueshrink:D \tex_skip:D #1 }
      { \tex_skip:D #1 }
  }
```

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

2.6.2 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 x-expansion will result 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_use:N \__int_eval:w #1 \exp_after:wN ;
    \int_use:N \__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:
    \tex_the:D \tex_toks:D #1 \exp_stop_f:
    \exp_after:wN \__regex_query_range_loop:ww
      \int_use:N \__int_eval:w #1 + \c_one ; #2 ;
  }
```

(*End definition for* `\__regex_query_range:nn`.)

__regex_query_submatch:n When this function is called, `\skip`*i* holds the start and end positions for the *i*-th overall submatch as its main and stretch components. In the case of repeated matches, submatches from all the matches are put one after the other in blocks of `\l__regex_capturing_group_int \skip` registers.

```
\cs_new:Npn \__regex_query_submatch:n #1
  {
    \__regex_query_range:nn
      { \tex_skip:D \__int_eval:w #1 }
      { \etex_gluestretch:D \tex_skip:D \__int_eval:w #1 }
  }
```

(*End 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 as the shrink component of \muskip registers, hence the contribution from a given range is the difference between the shrink components of \muskip⟨*max pos*⟩ and \muskip⟨*min pos*⟩. For the i-th submatch, the end-points of the range are the main and stretch components of \skip$i$. The trailing \scan_stop: is gobbled by \etex_muexpr:D, and the whole expression can be cast safely to an integer (no trailing expansion).

```
\cs_new_protected:Npn \__regex_submatch_balance:n #1
  {
    \etex_glueshrink:D \etex_mutoglue:D \etex_muexpr:D
      \tex_muskip:D \etex_gluestretch:D \tex_skip:D #1
      - \tex_muskip:D \tex_skip:D #1
    \scan_stop:
  }
```

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

2.6.3 Framework

__regex_replacement:n
__regex_replacement_aux:n

The replacement text is built incrementally by abusing \toks within a group (see l3tl-build). We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and \l__regex_balance_tl will consist of 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 #1
  {
⟨trace⟩    \trace_push:nnn { regex } { 1 } { __regex_replacement:n }
    \__tl_build:Nw \l__regex_internal_a_tl
      \int_zero:N \l__regex_balance_int
      \tl_clear:N \l__regex_balance_tl
      \__regex_escape_use:nnnn
        {
          \if_charcode:w \c_right_brace_str ##1
            \__regex_replacement_rbrace:N \else: \__tl_build_one:n \fi: ##1
        }
        { \__regex_replacement_escaped:N ##1 }
        { \__tl_build_one:n ##1 }
        {#1}
      \prg_do_nothing: \prg_do_nothing:
      \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero
        \__msg_kernel_error:nnx { regex } { replacement-missing-rbrace }
          { \int_use:N \l__regex_replacement_csnames_int }
        \__tl_build_one:x
          { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
      \fi:
      \cs_gset:Npx \__regex_replacement_balance_one_match:n ##1
        {
```

```
          + \int_use:N \l__regex_balance_int
          \l__regex_balance_tl
          - \__regex_submatch_balance:n {##1}
        }
    \__tl_build_end:
    \exp_args:No \__regex_replacement_aux:n \l__regex_internal_a_tl
⟨trace⟩    \trace_pop:nnn { regex } { 1 } { __regex_replacement:n }
  }
\cs_new_protected:Npn \__regex_replacement_aux:n #1
  {
    \cs_set:Npn \__regex_replacement_do_one_match:n ##1
      {
        \__regex_query_range:nn
          { \etex_glueshrink:D \tex_skip:D ##1 }
          { \tex_skip:D ##1 }
        #1
      }
  }
```

(*End definition for* `\__regex_replacement: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 \c_one < 1#1 \exp_stop_f:
          \__regex_replacement_put_submatch:n {#1}
        \else:
          \__tl_build_one:n #1
        \fi:
      }
  }
```

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

2.6.4 Submatches

`\__regex_replacement_put_submatch: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. Here, `##1` will receive a pointer to the 0-th submatch for a given match. We cannot use `\int_eval:n` because it is expandable, and would be expanded too early (short of adding `\exp_not:N`, making the code messy again).

```
\cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
  {
    \if_int_compare:w #1 < \l__regex_capturing_group_int
      \__tl_build_one:n { \__regex_query_submatch:n { #1 + ##1 } }
```

```
      \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero
        \tl_put_right:Nn \l__regex_balance_tl
          { + \__regex_submatch_balance:n { \__int_eval:w #1+##1 \__int_eval_end: } }
      \fi:
    \fi:
  }
```

(*End definition for* `\__regex_replacement_put_submatch:n`.)

`\__regex_replacement_g:w` `\__regex_replacement_g_digits:NN`

An ugly method to grab digits for the \g escape sequence. 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
  {
    \str_if_eq_x:nnTF { #1#2 } { \__tl_build_one:n \c_left_brace_str }
      {
        \int_zero:N \l__regex_internal_a_int
        \__regex_replacement_g_digits:NN
      }
      { \__regex_replacement_error:NNN g #1 #2 }
  }
\cs_new_protected:Npn \__regex_replacement_g_digits:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__tl_build_one:n
      {
        \if_int_compare:w \c_one < 1#2 \exp_stop_f:
          \int_set:Nn \l__regex_internal_a_int
            { \c_ten * \l__regex_internal_a_int + #2 }
          \exp_after:wN \use_i:nnn
          \exp_after:wN \__regex_replacement_g_digits:NN
        \else:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
      {
        \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 definition for* `\__regex_replacement_g:w` *and* `\__regex_replacement_g_digits:NN`.)

2.6.5 Csnames in replacement

__regex_replacement_c:w
__regex_replacement_c_{:w

\c can be followed by a left brace, or by a letter for which we have defined a way to produce that category of characters. The appropriate definitions for catcodes are introduced later. For control sequences, if we are within a control sequence, convert the token list to a string, otherwise simply prevent expansion, with a weird cross-over between \exp_not:n and \exp_not:N (see this helper's description for an explanation).

```
\cs_new_protected:Npn \__regex_replacement_c:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__tl_build_one:n
      {
        \cs_if_exist_use:cF { __regex_replacement_c_#2:w }
          { \__regex_replacement_error:NNN c #1#2 }
      }
      { \__regex_replacement_error:NNN c #1#2 }
  }
\cs_new_protected_nopar:cpn { __regex_replacement_c_ \c_left_brace_str :w }
  {
    \if_case:w \l__regex_replacement_csnames_int
      \__tl_build_one:n
        { \exp_not:n { \exp_after:wN \__regex_replacement_exp_not:N \cs:w } }
    \else:
      \__tl_build_one:n { \exp_not:n { \exp_after:wN \tl_to_str:N \cs:w } }
    \fi:
    \int_incr:N \l__regex_replacement_csnames_int
  }
```

(*End definition for* __regex_replacement_c:w.)

__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
  {
    \str_if_eq_x:nnTF { #1#2 } { \__tl_build_one:n \c_left_brace_str }
      {
        \if_case:w \l__regex_replacement_csnames_int
          \__tl_build_one:n { \exp_not:n { \exp_after:wN \exp_not:V \cs:w } }
        \else:
          \__tl_build_one:n { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
        \fi:
        \int_incr:N \l__regex_replacement_csnames_int
      }
      { \__regex_replacement_error:NNN u #1#2 }
  }
```

(*End 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
      \__tl_build_one:n \cs_end:
      \int_decr:N \l__regex_replacement_csnames_int
    \else:
      \__tl_build_one:n #1
    \fi:
  }
```

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

2.6.6 Characters in replacement

We will 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.

```
\cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
  {
    \if_meaning:w \prg_do_nothing: #3
      \__msg_kernel_error:nn { regex } { replacement-catcode-end }
    \else:
      \tex_lccode:D \c_zero = `#3 \scan_stop:
      \tl_to_lowercase:n { \__tl_build_one:n {#1} }
    \fi:
  }
```

(*End 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 `x`-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_nopar:Npn \__regex_replacement_c_A:w
  { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N ^^@ } } }
```

(*End 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 `x`-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_nopar:Npn \__regex_replacement_c_B:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero
      \int_incr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
  }
```

(*End 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 `x`-expansions.

```
\cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
  { \__tl_build_one:n { \exp_not:N \exp_not:N \exp_not:c {#2} } }
```

(*End 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_nopar:Npn \__regex_replacement_c_D:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

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

`\__regex_replacement_c_E:w` Similar to the begin-group case, the second `x`-expansion produces the bare end-group token.

```
\char_set_catcode_group_end:N \^^@
\cs_new_protected_nopar:Npn \__regex_replacement_c_E:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero
      \int_decr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \if_false: { \fi:  ^^@ } }
  }
```

(*End 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_nopar:Npn \__regex_replacement_c_L:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End 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_nopar:Npn \__regex_replacement_c_M:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End 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_nopar:Npn \__regex_replacement_c_O:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End 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 x-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_nopar:Npn \__regex_replacement_c_P:w
  {
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
  }
```

(*End 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_meaning:w \prg_do_nothing: #2
      \__msg_kernel_error:nn { regex } { replacement-catcode-end }
    \else:
      \if_int_compare:w `#2 = \c_zero
        \__msg_kernel_error:nn { regex } { replacement-null-space }
      \fi:
      \tex_lccode:D 32 = `#2 \scan_stop:
      \tl_to_lowercase:n { \__tl_build_one:n {~} }
    \fi:
  }
```

(*End 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_nopar:Npn \__regex_replacement_c_T:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End 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_nopar:Npn \__regex_replacement_c_U:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

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

Restore the catcode of the null byte.

```
\group_end:
```

2.6.7 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_kernel_error:nnx { regex } { replacement-#1 } {#3}
    #2 #3
  }
```

(*End definition for* `\__regex_replacement_error:NNN`.)

2.7 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 definition for* `\regex_new:N`. *This function is documented on page* 7.)

`\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_nopar:Npn \regex_set:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_set_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected_nopar:Npn \regex_gset:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_gset_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected_nopar:Npn \regex_const:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
  }
```

(*End definition for* `\regex_set:Nn`, `\regex_gset:Nn`, *and* `\regex_const:Nn`. *These functions are documented on page* 7.)

`\regex_show:N` `\regex_show:n` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `\__regex_show:Nx` is defined in a different section.

```
\cs_new_protected:Npn \regex_show:n #1
  {
    \__regex_compile:n {#1}
    \__regex_show:Nx \l__regex_internal_regex
```

```
      { { \tl_to_str:n {#1} } }
  }
\cs_new_protected:Npn \regex_show:N #1
  { \__regex_show:Nx #1 { variable~\token_to_str:N #1 } }
```

(*End definition for* `\regex_show:N` *and* `\regex_show:n`*. These functions are documented on page 7.*)

`\regex_match:nnTF`
`\regex_match:NnTF`

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_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:N #1 } {#2}
    \__regex_return:
  }
```

(*End definition for* `\regex_match:nnTF` *and* `\regex_match:NnTF`*. These functions are documented on page* ??.)

`\regex_count:nnN`
`\regex_count:NnN`

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 } }
```

(*End definition for* `\regex_count:nnN` *and* `\regex_count:NnN`*. These functions are documented on page* ??.)

`\regex_extract_once:nnN`
`\regex_extract_once:NnN`
`\regex_extract_all:nnN`
`\regex_extract_all:NnN`
`\regex_replace_once:nnN`
`\regex_replace_once:NnN`
`\regex_replace_all:nnN`
`\regex_replace_all:NnN`
`\regex_split:nnN`
`\regex_split:NnN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:NnNTF`
`\regex_extract_all:nnNTF`
`\regex_extract_all:NnNTF`
`\regex_replace_once:nnNTF`
`\regex_replace_once:NnNTF`
`\regex_replace_all:nnNTF`
`\regex_replace_all:NnNTF`
`\regex_split:nnNTF`
`\regex_split:NnNTF`

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: }
  }
\__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 definition for* `\regex_extract_once:nnN` *and others. These functions are documented on page* **??**.)

2.7.1 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 definition for* `\l__regex_match_count_int`. *This variable is documented on page* **??**.)

`__regex_begin` `__regex_end` Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.

```
\flag_new:n { __regex_begin }
\flag_new:n { __regex_end }
```

(*End definition for* `__regex_begin` *and* `__regex_end`. *These variables are documented on page* **??**.)

`\l__regex_submatch_int` `\l__regex_zeroth_submatch_int` The end-points of each submatch are stored as main and stretch components of `\skip`⟨*submatch*⟩, where ⟨*submatch*⟩ ranges from `\l__regex_max_state_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`. Additionally, the shrink component of this 0-th submatch is the position at which that match attempt started: this is used for splitting and replacements.

```
\int_new:N \l__regex_submatch_int
\int_new:N \l__regex_zeroth_submatch_int
```

(*End definition for* `\l__regex_submatch_int` *and* `\l__regex_zeroth_submatch_int`. *These variables are documented on page* **??**.)

`\__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_nopar:Npn \__regex_return:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__regex_return:`.)

2.7.2 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 definition for* `\__regex_if_match: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 definition for* `\__regex_count:nnN`.)

2.7.3 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_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_group_end_extract_seq:N #3
  }
```

(*End 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 `\skip` registers 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:
            \tex_skip:D \l__regex_zeroth_submatch_int
              = \l__regex_start_pos_int sp
                plus \tex_skip:D \l__regex_zeroth_submatch_int \scan_stop:
          \fi:
        }
      #1
      \__regex_match:n {#2}
⟨assert⟩\assert_int:n { \l__regex_current_pos_int = \l__regex_max_pos_int }
      \tex_skip:D \l__regex_submatch_int
        = \l__regex_start_pos_int sp plus \l__regex_max_pos_int sp \scan_stop:
      \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 definition for* `\__regex_split:nnN`.)

`\__regex_group_end_extract_seq:N` The end-points of submatches are stored as the main and stretch components of `\skip` registers from `\l__regex_max_state_int` to `\l__regex_submatch_int` (exclusive). Extract the relevant ranges into `\l__regex_internal_a_tl`. We detect unbalanced results using the two flags `@@_begin` and `@@_end`, raised whenever we see too many begin-group or end-group tokens in a submatch. We disable `\__seq_item:n` to prevent two `x`-expansions.

```
\cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
  {
      \cs_set_eq:NN \__seq_item:n \scan_stop:
      \flag_clear:n { __regex_begin }
      \flag_clear:n { __regex_end }
      \tl_set:Nx \l__regex_internal_a_tl
        {
          \s__seq
          \int_step_function:nnnN
            { \c_two * \l__regex_max_state_int }
            \c_one
            { \l__regex_submatch_int - \c_one }
            \__regex_extract_seq_aux:n
        }
      \int_compare:nNnF
        { \flag_height:n { __regex_begin } + \flag_height:n { __regex_end } }
        = \c_zero
        {
          \__msg_kernel_error:nnxxx { regex } { result-unbalanced }
            { splitting~or~extracting~submatches }
            { \flag_height:n { __regex_end } }
            { \flag_height:n { __regex_begin } }
        }
      \use:x
        {
          \group_end:
          \tl_set:Nn \exp_not:N #1 { \l__regex_internal_a_tl }
        }
  }
```

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

`\__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
  {
    \__seq_item:n
      {
        \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
      \flag_raise:n { __regex_end }
      \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
    \fi:
    \__regex_query_submatch:n {#2}
```

```
    \if_int_compare:w #1 > \c_zero
      \flag_raise:n { __regex_begin }
      \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
    \fi:
  }
```

(*End definition for* `\__regex_extract_seq_aux:n` *and* `\__regex_extract_seq_aux:ww`.)

`\__regex_extract:`
`\__regex_extract_b:wn`
`\__regex_extract_e:wn`

Our task here is to extract from the property list `\l__regex_success_submatches_prop` the list of end-points of submatches, and store them in `\skip` registers, from `\l__regex_zeroth_submatch_int` upwards. We begin by emptying those `\skip` registers. Then for each ⟨*key*⟩–⟨*value*⟩ pair in the property list update the appropriate `\skip` component. This is somewhat a hack: the ⟨*key*⟩ is a non-negative integer followed by `<` or `>`, which we use in a comparison to −1. At the end, store the information about the position at which the match attempt started, as a shrink component.

```
\cs_new_protected_nopar: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
        {
          \tex_skip:D \l__regex_submatch_int \c_zero sp \scan_stop:
          \int_incr:N \l__regex_submatch_int
        }
      \prop_map_inline:Nn \l__regex_success_submatches_prop
        {
          \if_int_compare:w ##1 \c_minus_one
            \exp_after:wN \__regex_extract_e:wn \__int_value:w
          \else:
            \exp_after:wN \__regex_extract_b:wn \__int_value:w
          \fi:
          \__int_eval:w \l__regex_zeroth_submatch_int + ##1 {##2}
        }
      \tex_skip:D \l__regex_zeroth_submatch_int
        = \tex_the:D \tex_skip:D \l__regex_zeroth_submatch_int
          minus \l__regex_start_pos_int sp \scan_stop:
    \fi:
  }
\cs_new_protected:Npn \__regex_extract_b:wn #1 < #2
  {
    \tex_skip:D #1 = #2 sp
      plus \etex_gluestretch:D \tex_skip:D #1 \scan_stop:
  }
\cs_new_protected:Npn \__regex_extract_e:wn #1 > #2
  {
    \tex_skip:D #1
      = 1 \tex_skip:D #1 plus #2 sp \scan_stop:
  }
```

(*End definition for* `\__regex_extract:`, `\__regex_extract_b:wn`, *and* `\__regex_extract_e:wn`.)

2.7.4 Replacement

__regex_replace_once: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 x-expansion, and checks that braces are balanced in the final result.

```
\cs_new_protected:Npn \__regex_replace_once:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \__regex_replacement:n {#2}
      \exp_args:No \__regex_match:n { #3 }
      \if_meaning:w \c_false_bool \g__regex_success_bool
        \group_end:
      \else:
        \__regex_extract:
        \int_set:Nn \l__regex_balance_int
          {
            \__regex_replacement_balance_one_match:n
              { \l__regex_zeroth_submatch_int }
          }
        \tl_set:Nx \l__regex_internal_a_tl
          {
            \__regex_replacement_do_one_match:n { \l__regex_zeroth_submatch_int }
            \__regex_query_range:nn
              { \etex_gluestretch:D \tex_skip:D \l__regex_zeroth_submatch_int }
              { \l__regex_max_pos_int }
          }
        \__regex_group_end_replace:N #3
      \fi:
  }
```

(*End definition for* `\__regex_replace_once: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 (as the shrink component of a `\skip` register). The `\skip` registers from `\l__regex_max_state_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 `\skip` registers. 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#3
  {
```

```
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \__regex_replacement:n {#2}
      \exp_args:No \__regex_match:n {#3}
      \int_set:Nn \l__regex_balance_int
        {
          0
          \int_step_function:nnnN
            { \c_two * \l__regex_max_state_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - \c_one }
            \__regex_replacement_balance_one_match:n
        }
      \tl_set:Nx \l__regex_internal_a_tl
        {
          \int_step_function:nnnN
            { \c_two * \l__regex_max_state_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - \c_one }
            \__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 definition for* `\__regex_replace_all:nnN`.)

`\__regex_group_end_replace:N` If the brace balance is not 0, raise an error. Then set the user's variable `#1` to the `x`-expansion of `\l__regex_internal_a_tl`, adding the appropriate braces to produce a balanced result. And end the group.

```
\cs_new_protected_nopar:Npn \__regex_group_end_replace:N #1
  {
    \if_int_compare:w \l__regex_balance_int = \c_zero
    \else:
      \__msg_kernel_error:nnxxx { regex } { result-unbalanced }
        { replacing }
        { \int_max:nn { - \l__regex_balance_int } { \c_zero } }
        { \int_max:nn { \l__regex_balance_int } { \c_zero } }
    \fi:
    \use:x
      {
        \group_end:
        \tl_set:Nn \exp_not:N #1
          {
            \if_int_compare:w \l__regex_balance_int < \c_zero
              \prg_replicate:nn { - \l__regex_balance_int }
                { { \if_false: } \fi: }
            \fi:
```

```
            \l__regex_internal_a_tl
            \if_int_compare:w \l__regex_balance_int > \c_zero
              \prg_replicate:nn { \l__regex_balance_int }
                { \if_false: { \fi: } }
            \fi:
          }
      }
  }
```

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

2.7.5 Storing and showing compiled patterns

2.8 Messages

Messages for the preparsing phase.

```
\__msg_kernel_new:nnnn { regex } { trailing-backslash }
  { Trailing~escape~character~\iow_char:N\\. }
  {
    A~regular~expression~or~its~replacement~text~ends~with~
    the~escape~character~\iow_char:N\\.~It~will~be~ignored.
  }
\__msg_kernel_new:nnnn { regex } { x-missing-rbrace }
  { Missing~closing~brace~in~\iow_char:N\\x~hexadecimal~sequence. }
  {
    You~wrote~something~like~
    `\iow_char:N\\x\{\int_to_hexadecimal:n{#1}'.~
    The~closing~brace~is~missing.
  }
\__msg_kernel_new:nnnn { regex } { x-overflow }
  { Character~code~'#1'~too~large~in~\iow_char:N\\x~hexadecimal~sequence. }
  {
    You~wrote~something~like~
    `\iow_char:N\\x\{\int_to_hexadecimal:n{#1}\}'.~
    The~character~code~'#1'~is~larger~than~\int_use:N \c_max_char_int.
  }
```

Invalid quantifier.

```
\__msg_kernel_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>}',~followed~or~not~by~'?'.
  }
```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```
\__msg_kernel_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_kernel_new:nnnn { regex } { missing-rparen }
  {
    Missing~right~parenthes\int_compare:nTF{#1=1}{i}{e}s~
    inserted~in~regular~expression.
  }
  {
    LaTeX~was~given~a~regular~expression~with~\int_eval:n{#1}~
    more~left~parenthes\int_compare:nTF{#1=1}{i}{e}s~than~right~
    parenthes\int_compare:nTF{#1=1}{i}{e}s.
  }
\__msg_kernel_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_kernel_new:nnnn { regex } { bad-escape }
  {
    Invalid~escape~\c_backslash_str #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_kernel_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_kernel_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_kernel_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{...}’.~
    To~combine~several~category~tests,~use~’\iow_char:N\\c[...]’.
  }
\__msg_kernel_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_kernel_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_kernel_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_kernel_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_kernel_new:nnnn { regex } { u-missing-rbrace }
  { Missing~right~brace~inserted~for~\iow_char:N\\u~escape. }
  {
```

```
    LaTeX~
    \str_if_eq_x:nnTF { } {#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_kernel_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_kernel_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_kernel_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_kernel_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_kernel_new:nnnn { regex } { unknown-option }
  { Unknown~option~‘#1’~for~regular~expressions. }
  {
    The~only~available~option~is~’case-insensitive’,~toggled~by~
    ’(?i)’~and~’(?-i)’.
  }
```

Errors in the replacement text.

```
\__msg_kernel_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_kernel_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_kernel_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_kernel_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_kernel_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_kernel_new:nnnn { regex } { replacement-missing-rbrace }
  { Missing~right~brace~inserted~in~replacement~text. }
  {
    There~were~\int_use:N \l__regex_replacement_csnames_int \
    missing~right~braces.
  }
```

`\__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_x:nnF { #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 definition for* `\__regex_msg_repeated:nnN`.)

2.9 Code for tracing

The tracing code is still very experimental, and is meant to be used with the l3trace package, currently in `l3trial`.

`\__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).

```
⟨*trace⟩
\cs_new_protected:Npn \__regex_trace_states:n #1
  {
    \int_step_inline:nnnn
      \l__regex_min_state_int
      \c_one
      { \l__regex_max_state_int - 1 }
      {
        \trace:nnx { regex } { #1 }
          { \iow_char:N \\toks ##1 = { \tex_the:D \tex_toks:D ##1 } }
      }
  }
⟨/trace⟩
```

(*End definition for* `\__regex_trace_states:n`.)

```
⟨/initex | package⟩
```

Index

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

Symbols

A

B

C

F

G

I

L

O

P

U