The l3regex package: regular expressions in TEX*

The LATEX3 Project†

Released 2012/01/19

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 we act on lists of 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 token list variable using `\regex_const:Nn`. For example,

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

stores in `\c_foo_regex_tl` 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 v3209, last revised 2012/01/19.

†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).

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 character in the subject string.

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

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

`[x-y]` Range (can be used with escaped characters).

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`⟨*character or class*⟩ Matches a token with category code `X` (any of `CBEMTPUDSLOA`) if it is also matched by the ⟨*character or class*⟩. For instance, `\cL[A-Z]` matches uppercase letters of category code letter, `\cC.` matches any control sequence, and `\cO\d` matches digits of category other.

`\c[XYZ]`⟨*character or class*⟩ Matches a token with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`), if it is also matched by the ⟨*character or class*⟩. For instance, `\c[LSO].` matches tokens of category letter, space, or other.

`\c[^XYZ]`⟨*character or class*⟩ Matches a token with category different from `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`), if it is also matched by the ⟨*character or class*⟩. For instance, `\c[LSO].` matches tokens of category letter, space, or 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.

Options can be set with `(?`⟨*option*⟩`)` and unset with `(?-`⟨*option*⟩`)`. Options are local to the group in which they are set, and revert to their previous setting upon reaching the closing parenthesis. For instance, in `(?i)a(b(?-i)c|d)e`, the `i` option applies to the letters `a`, `b` and `e`.

`(?i)` and `(?-i)` Toggle to a case insensitive/sensitive mode. This only applies to ascii letters (mapping `A`–`Z` to `a`–`z`). For instance, `(?i)[Y-\\]` matches the characters `Y`, `Z`, `[`, `\`, and the lower case letters `y` and `z`, while `(?i)[^aeiou]` matches any character which is not a vowel.

In character classes, only `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. The escape sequences `\d`, `\D`, etc. are also supported in character classes. If the first character is `^`, then the meaning of the character class is inverted. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` is equivalent to `[^6-9]`.

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 strings using for instance `\regex_extract_once:nnNTF`.

1.2 Syntax of in 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.* As an experimental feature, `\c{...}` can be nested, but the behaviour is not yet fully specified. Use at own risks.

1.3 Precompiling 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 precompiled regular expression is stored as a token list variable. All of the l3regex module's functions can be given their regular expression argument either as an explicit string or as a precompiled regular expression.

`\regex_set:Nn`
`\regex_gset:Nn`
`\regex_const:Nn`

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

Stores a precompiled version of the ⟨*regular expression*⟩ in the ⟨*tl var*⟩. For instance, this function can be used as

```
\tl_new:N \l_my_regex_tl
\regex_set:Nn \l_my_regex_tl { 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 precompiled expressions which will never change.

`\regex_to_str:N`

`\regex_to_str:N` ⟨*regex var*⟩

Converts the ⟨*regex var*⟩, previously defined using `\regex_(g)set:Nn`, to a string, which can be safely read back when the LaTeX3 syntax is active (as triggered by `\ExplSyntaxOn`). Line breaks are inserted at various places, to *try* and keep the line length short. For instance, if `\l_my_stream` is an open stream for writing, and `\l_regex_tl` is a regex variable, you can save its definition using

```
\iow_now:Nx \l_my_stream
  {
    \tl_to_str:n { \tl_set:Nn \l_my_regex_tl } { \iow_newline:
        \regex_to_str:N \l_my_regex_tl
      }
  }
```

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 precompiled expression as generated by `\regex_(g)set:Nn`.

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

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

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

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

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

`\regex_count:nnN` `\regex_count: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 string: 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.

- The {} quantifiers are only partially implemented.
- Clean up the use of messages.
- Triple check magic numbers.
- Test \par.
- Add tests for every regex and replacement feature.
- Make sure the documentation is correct, and the code comments are as well.

Code improvements to come.

- \regex_show:N to show how a given regular expression is interpreted.
- Test for the maximum register \c_max_register_int.
- Use \dimen registers rather than \l_regex_nesting_tl to build \regex_nesting:n.
- Reduce the number of epsilon-transitions in alternatives.
- Improve nesting of \c in the replacement text.
- Move the "reconstruction" part of l3regex to l3tl-analysis.
- Optimize regexes for csnames when the regex is a simple string.
- Optimize simple strings: use less states (abcade should give two states, for abc and ade).
- Optimize groups with no alternative.
- Optimize the use of \prg_stepwise_... functions.

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

- General look-ahead/behind assertions.
- Regex matching on external files.

- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- `(*..)` and `(?..)` sequences to set some options (partially implemented).
- `\K` for resetting the beginning of the match.
- 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, hence a lot of optimization ahead.

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

- `\ddd`, matching the character with code `ddd` in octal;
- POSIX character classes `[:alpha:]` *etc.*, this is redundant;
- 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

<*package>

```
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\RequirePackage{l3str, l3tl-analysis, l3flag}
```

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly. 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 build a non-deterministic finite automaton (NFA) with roughly n states, which accepts precisely those token lists matching that regular expression. We then run the token list through the NFA, and check the return value.

The code is structured as follows. Various helper functions are introduced in the next subsection, to limit the clutter in later parts. Then functions pertaining to parsing the regular expression are introduced: that part is rather long because of the many bells and whistles that we need to cater for. The next subsection takes care of running the NFA, and describes how the various TEX registers are (ab)used in this module. Finally, user functions.

2.1 Constants and variables

`\regex_tmp:w` `\l_regex_tmpa_tl` `\l_regex_tmpb_tl` `\l_regex_tmpc_tl` `\l_regex_tmpa_int` `\l_regex_tmpb_int`

Temporary variables.

```
\cs_new:Npn \regex_tmp:w { }
\tl_new:N   \l_regex_tmpa_tl
\tl_new:N   \l_regex_tmpb_tl
\tl_new:N   \l_regex_tmpc_tl
\int_new:N  \l_regex_tmpa_int
\int_new:N  \l_regex_tmpb_int
```

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

`\l_regex_group_begin_flag` `\l_regex_group_end_flag`

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

```
\flag_new:N \l_regex_group_begin_flag
\flag_new:N \l_regex_group_end_flag
```

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

2.1.1 Variables used while building

\l_regex_build_mode_int

While building, ten modes are recognized, labelled −63, −23, −6, −2, 0, 2, 3, 6, 23, 63. See section 2.3.1.

```
\int_new:N \l_regex_build_mode_int
```

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

\l_regex_max_state_int
\l_regex_left_state_int
\l_regex_right_state_int

The end-point states of the last group (which any quantifier would repeat) are stored as `\l_regex_left/right_state_int`. For simple strings of characters, the left and right pointers only differ by one. 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.

```
\int_new:N  \l_regex_max_state_int
\int_new:N  \l_regex_left_state_int
\int_new:N  \l_regex_right_state_int
```

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

\l_regex_left_state_seq
\l_regex_right_state_seq

Alternatives are implemented by branching from a state into the various choices, then merging those into another state. We store information about those states in two sequences.

```
\seq_new:N  \l_regex_left_state_seq
\seq_new:N  \l_regex_right_state_seq
```

(*End definition for* `\l_regex_left_state_seq` *and* `\l_regex_right_state_seq`. *These functions are documented on page* **??**.)

\l_regex_end_group_seq
\l_regex_end_alternation_seq

These sequences hold actions to be performed at the end of a group, and at the end of each branch of the alternation, respectively. Currently, `\l_regex_end_group_seq` is used to keep track of letter case, and `\l_regex_end_alternation_seq` is used for `(?|...)` groups.

```
\seq_new:N  \l_regex_end_group_seq
\seq_new:N  \l_regex_end_alternation_seq
```

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

\l_regex_capturing_group_int
\l_regex_capturing_group_seq
\l_regex_capturing_group_max_int

`\l_regex_capturing_group_int` is the ID number of the current capturing group, starting at 0 for a group enclosing the full regular expression, and counting in the order of their left parenthesis. This number is used when a branch of the alternation ends. Capturing groups can be arbitrarily nested, and we keep track of the stack of ID numbers in `\l_regex_capturing_group_seq`.

```
\int_new:N  \l_regex_capturing_group_int
\seq_new:N  \l_regex_capturing_group_seq
\int_new:N  \l_regex_capturing_group_max_int
```

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

\l_regex_one_or_group_tl

When looking for quantifiers, this variable holds either "one" or "group" depending on whether the object to which the quantifier applies matches one character (*i.e.*, is a character or character class), or is a group.

```
\tl_new:N   \l_regex_one_or_group_tl
```

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

`\l_regex_catcodes_int`
`\l_regex_catcodes_default_int`
`\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_catcodes_all_int`

We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, matching two tokens which are neither a begin-group nor an end-group token, followed by a token of category letter and character code in `[a-z]`, followed by two more tokens which are neither begin-group nor end-group tokens. For this to work, we need to keep track of lists of allowed category codes: `\l_regex_catcodes_int` and `\l_regex_catcodes_default_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.

```
\int_new:N \l_regex_catcodes_int
\int_new:N \l_regex_catcodes_default_int
\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_catcodes_all_int { "5515155 }
```

(*End definition for* `\l_regex_catcodes_int` *and* `\l_regex_catcodes_default_int`. *These functions are documented on page* **??**.)

`\l_regex_catcodes_bool`

Controls whether the bitmap of category codes built should be inverted or not.

```
\bool_new:N \l_regex_catcodes_bool
```

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

`\l_regex_tmpa_regex_tl`

This holds a temporary pre-compiled regular expression when matching a control sequence name.

```
\tl_new:N \l_regex_tmpa_regex_tl
```

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

2.1.2 Character classes

`\l_regex_class_bool`
`\l_regex_class_tl`
`\l_regex_class_saved_bool`
`\l_regex_class_saved_tl`

`\l_regex_class_bool` is false for negative character classes. `\l_regex_class_tl` holds the tests which should be performed to decide whether the `\l_regex_current_char_int` matches that character class. In nested class, which can only occur as `[\c[...]]`, the two variables from the outer class must be saved while processing the inner class.

```
\bool_new:N \l_regex_class_bool
\tl_new:N   \l_regex_class_tl
\bool_new:N \l_regex_class_saved_bool
\tl_new:N   \l_regex_class_saved_tl
```

(*End definition for* `\l_regex_class_bool` *and* `\l_regex_class_tl`. *These functions are documented on page* **??**.)

`\c_regex_d_tl`
`\c_regex_D_tl`
`\c_regex_h_tl`
`\c_regex_H_tl`
`\c_regex_s_tl`
`\c_regex_S_tl`
`\c_regex_v_tl`
`\c_regex_V_tl`
`\c_regex_w_tl`
`\c_regex_W_tl`
`\c_regex_N_tl`

These constant token lists encode which characters are recognized by `\d`, `\D`, `\w`, *etc.* in regular expressions. Namely, `\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 ranges appear is optimized for usual mostly lower case letter text.

```
\tl_const:Nn \c_regex_d_tl
  {
    \regex_item_range:nn { \c_forty_eight } { 57 } % 0--9
  }
\tl_const:Nn \c_regex_D_tl
  {
    \regex_item_geq:n { \c_fifty_eight } % > `9
    \regex_item_range:nn { \c_zero } { 47 } % 0
  }
\tl_const:Nn \c_regex_h_tl
  {
    \regex_item_equal:n { \c_thirty_two } % space
    \regex_item_equal:n { \c_nine } % tab
  }
\tl_const:Nn \c_regex_H_tl
  {
    \regex_item_geq:n { 33 } % > space
    \regex_item_range:nn { \c_ten } { 31 } % tab < ... < space
    \regex_item_range:nn { \c_zero } { \c_eight } % < tab
  }
\tl_const:Nn \c_regex_s_tl
  {
    \regex_item_equal:n  { \c_thirty_two } % space
    \regex_item_range:nn { \c_nine } { \c_ten } % tab, lf
    \regex_item_range:nn { \c_twelve } { \c_thirteen } % ff, cr
  }
\tl_const:Nn \c_regex_S_tl
  {
    \regex_item_geq:n    { 33 } % > space
    \regex_item_range:nn { \c_fourteen } { 31 } % tab < ... < space
    \regex_item_range:nn { \c_zero } { \c_eight } % < tab
    \regex_item_equal:n  { \c_eleven } % vtab
  }
\tl_const:Nn \c_regex_v_tl
  {
    \regex_item_range:nn { \c_ten } { \c_thirteen } % lf, vtab, ff, cr
  }
\tl_const:Nn \c_regex_V_tl
  {
    \regex_item_geq:n    { \c_fourteen } % >cr
    \regex_item_range:nn { \c_zero } { \c_nine } % < lf
```

```
  }
\tl_const:Nn \c_regex_w_tl
  {
    \regex_item_range:nn { \c_ninety_seven } { 122 } % a--z
    \regex_item_range:nn { \c_sixty_five } { 90 } % A--Z
    \regex_item_range:nn { \c_forty_eight } { 57 } % 0--9
    \regex_item_equal:n  { 95 } % _
  }
\tl_const:Nn \c_regex_W_tl
  {
    \regex_item_range:nn { \c_zero } { 47 } % <‘0
    \regex_item_range:nn { \c_fifty_eight } { 64 } % (‘9+1)--(‘A-1)
    \regex_item_range:nn { \c_ninety_one } { 94 } % (‘Z+1)--(‘_-1)
    \regex_item_equal:n  { 96 } % ‘
    \regex_item_geq:n    { \c_one_hundred_twenty_three } % z
  }
\tl_const:Nn \c_regex_N_tl
  {
    \regex_item_geq:n { \c_eleven } % > 1f
    \regex_item_range:nn { \c_zero } { \c_nine } % < 1f
  }
```

(*End definition for* `\c_regex_d_tl` *and* `\c_regex_D_tl`*. These functions are documented on page* ??.)

2.1.3 Variables used when matching

`\l_regex_nesting_int` This integer is used to keep track of begin and end-group tokens.

```
\int_new:N \l_regex_nesting_int
```

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

`\l_regex_min_step_int` `\l_regex_max_step_int` This integer holds the value of the step corresponding to the left end of the token list, *i.e.*, when the token list is stored in `\toks` registers, the `\toks` register `\l_regex_min_step_int` holds the first token in the token list. In fact, this number is always one more than `\l_regex_max_state_int`, but it is more practical to give that a name.

```
\int_new:N \l_regex_min_step_int
\int_new:N \l_regex_max_step_int
```

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

`\l_regex_current_step_int` `\l_regex_start_step_int` `\l_regex_success_step_int` While reading through the query token list, `\l_regex_current_step_int` is the position in the token list, starting at `\l_regex_min_step_int` for the left-most token. Each match begins at the position given by `\l_regex_start_step_int`. Whenever an execution thread succeeds, the corresponding step is stored into `\l_regex_success_step_int`, which will be the next starting step (except in the case of empty matches).

```
\int_new:N \l_regex_current_step_int
\int_new:N \l_regex_start_step_int
\int_new:N \l_regex_success_step_int
```

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

\l_regex_unique_id_int

In the case of repeated matches, \l_regex_current_step_int is reset to the end-position of the previous match. In contrast, \l_regex_unique_id_int is simply incremented to provide a unique number for each iteration of the matching loop. This is handy to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past iteration).

```
\int_new:N \l_regex_unique_id_int
```

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

\l_regex_current_char_int
\l_regex_current_catcode_int
\l_regex_current_token_tl
\l_regex_last_char_int
\l_regex_case_changed_char_int

The character codes of the character at the current position in the token list, and at the previous position, and the current character with its case changed (A-Z↔a-z). The \l_regex_last_char_int is used to test for word boundaries (\b and \B). The \l_regex_case_changed_char_int is only computed if the "case insensitive" option (?i) is used in the regex.

```
\int_new:N \l_regex_current_char_int
\int_new:N \l_regex_current_catcode_int
\tl_new:N \l_regex_current_token_tl
\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. *This function is documented on page* ??.)

\l_regex_caseless_bool

True if caseless matching is used within the regular expression. This controls whether \l_regex_case_changed_char_int is computed.

```
\bool_new:N \l_regex_caseless_bool
```

(*End definition for* \l_regex_caseless_bool. *This function is 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. In the case of groups with quantifiers, \l_regex_current_state_int is shifted to a fake value for transitions to point to the correct states.

```
\int_new:N \l_regex_current_state_int
```

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

\l_regex_current_submatches_prop
\l_regex_success_submatches_prop

The submatches for the thread which lies at the \l_regex_current_state_int are stored in a property list variable. This property list is stored by \regex_action_cost:n into the \toks register for the target state of the transition. When a thread succeeds, this property list is copied to \l_regex_success_submatches_prop and only the last sucessful 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. *This function is documented on page* ??.)

\l_regex_max_index_int

All the currently active states are kept in order of precedence in the \skip registers, which for our purpose serve as an array: the ith item of the array is \skip$i$. The largest index used after treating the previous character is \l_regex_max_index_int. At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and \l_regex_max_index_int reset to zero, effectively clearing the array.

```
\int_new:N \l_regex_max_index_int
```

(*End definition for* `\l_regex_max_index_int`. *This function is 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 set to removing the remainder of the query token list. For multiple matching, the token list is set to repeat the matching.

```
\tl_new:N \l_regex_every_match_tl
```

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

`\g_regex_success_bool` `\l_regex_saved_success_bool` `\l_regex_success_match_bool` The boolean `\g_regex_success_bool` is true if there was at least one match, and `\l_regex_success_match_bool` is true if the current match attempt was successful. The variable `\g_regex_success_bool` is the only global variable in this whole module. When nesting `\regex` functions internally, the value of `\g_regex_success_bool` is saved 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_success_match_bool
```

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

`\regex_last_match_empty:F` `\regex_last_match_empty_no:F` `\regex_last_match_empty_yes:F` When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When we detect such a situation, the next match attempt is shifted by one character. Namely, an empty match is discarded if it follows an empty match at the same position. If the previous match was non-empty, `\regex_last_match_empty:F` is simply `\use:n`, and keeps the match. If it was empty, then we test whether the new match has moved or not: if it has not, then the success is discarded.

```
\cs_new_protected:Npn \regex_last_match_empty_no:F #1 {#1}
\cs_new_protected:Npn \regex_last_match_empty_yes:F
  { \int_compare:nNnF \l_regex_start_step_int = \l_regex_current_step_int }
\cs_new_eq:NN \regex_last_match_empty:F \regex_last_match_empty_no:F
```

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

`\l_regex_success_empty_bool` `\l_regex_fresh_thread_bool` When a match succeeds, `\l_regex_success_empty_bool` records whether it is empty. This information is used to initialize `\regex_last_match_empty:F` before starting the next match attempt. The boolean `\l_regex_fresh_thread_bool` is true when the current thread has started from the beginning of the regular expression at this character. This is probably suboptimal. Improvements welcome.

```
\bool_new:N \l_regex_success_empty_bool
\bool_new:N \l_regex_fresh_thread_bool
```

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

2.1.4 Variables used when building the replacement

\l_regex_replacement_int

```
\int_new:N \l_regex_replacement_int
```

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

\l_regex_replacement_csnames_int

```
\int_new:N \l_regex_replacement_csnames_int
```

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

\l_regex_submatch_int
\l_regex_submatch_start_int

```
\int_new:N \l_regex_submatch_int
\int_new:N \l_regex_submatch_start_int
```

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

2.1.5 Variables used 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 function is documented on page* ??.)

2.2 Helpers

2.2.1 Toks

When performing the matching, the `\toks` registers hold submatch information, followed by the instruction for a given state of the NFA. The two parts are separated by `\s_stop`.

\regex_toks_put_left:Nx
\regex_toks_put_right:Nx

During the building phase, every `\toks` register starts with `\s_stop`, and we wish to add x-expanded material to those registers. The expansion is done "by hand" for optimization (these operations are used quite a lot). When adding material to the left, we define `\regex_tmp:w` to remove the `\s_stop` marker and put it back to the left of the new material.

```
\cs_new_protected:Npn \regex_toks_put_left:Nx #1#2
  {
    \cs_set_nopar:Npx \regex_tmp:w \s_stop { \s_stop #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 }
  }
```

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

2.2.2 Extracting parts of the query token list

\regex_query_substr:nn
\regex_query_submatch:nn
\regex_query_submatch:w

The token list is stored in \toks registers: the first is \l_regex_min_step_int, the last is (we don't care yet).

```
\cs_new:Npn \regex_query_substr:nn #1#2
  {
    \if_num:w \int_eval:w #1 < \int_eval:w #2 \int_eval_end:
      \str_aux_toks_range:nn {#1} {#2}
    \fi:
  }
\cs_new:Npn \regex_query_submatch:nn #1#2
  {
    \if_num:w #1 < \l_regex_capturing_group_int
      \exp_after:wN \regex_query_submatch:w
      \int_use:N \int_eval:w #1 + #2 ;
    \fi:
  }
\cs_new:Npn \regex_query_submatch:w #1 ;
  {
    \regex_query_substr:nn
      { \tex_skip:D #1 }
      { \etex_gluestretch:D \tex_skip:D #1 }
  }
```

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

2.2.3 Sequences

\regex_seq_pop_int:NN
\regex_seq_get_int:NN
\regex_seq_push_int:NN

When building the regular expression, we keep track of some integers (pointers to various states) without help from TEX's grouping. Here are variants of \seq_pop:NN and \seq_get:NN which assign using \int_set:Nn rather than \tl_set:Nn.

```
\cs_new_protected:Npn \regex_seq_pop_int:NN #1#2
  {
    \seq_pop:NN #1 \l_regex_tmpa_tl
    \int_set:Nn #2 \l_regex_tmpa_tl
  }
\cs_new_protected:Npn \regex_seq_get_int:NN #1#2
  {
    \seq_get:NN #1 \l_regex_tmpa_tl
    \int_set:Nn #2 \l_regex_tmpa_tl
  }
\cs_new_protected:Npn \regex_seq_push_int:NN #1#2
  { \seq_push:No #1 { \int_use:N #2 } }
```

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

\regex_seq_pop_use:N
\regex_seq_get_use:N

When building the regular expression, some settings are kept local to capturing groups without any help from TEX's grouping. This is done "by hand", in sequences whose items should be run immediately.

```
\cs_new_protected:Npn \regex_seq_pop_use:N #1
  {
```

```
    \seq_pop:NN #1 \l_regex_tmpa_tl
    \l_regex_tmpa_tl
  }
\cs_new_protected:Npn \regex_seq_get_use:N #1
  {
    \seq_get:NN #1 \l_regex_tmpa_tl
    \l_regex_tmpa_tl
  }
```

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

2.2.4 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`*. This function is documented on page* **??**.)

`\regex_item_dot:`

The dot meta-character matches any character, except the end marker.

```
\cs_new_protected_nopar:Npn \regex_item_dot:
  {
    \if_num:w \l_regex_current_char_int > - \c_two
      \exp_after:wN \regex_break_true:w
    \fi:
  }
```

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

`\regex_item_caseful_equal:n`
`\regex_item_caseful_range:nn`
`\regex_item_caseful_geq:n`

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

```
\cs_new_protected:Npn \regex_item_caseful_equal:n #1
  {
    \if_num: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_num:w #1 > \l_regex_current_char_int
      \reverse_if:N \if_num: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_caseful_geq:n #1
  {
    \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
  }
```

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

`\regex_item_caseless_equal:n` `\regex_item_caseless_range:nn` `\regex_item_caseless_geq:n`

For caseless matching, we perform the test both on `\l_regex_current_char_int` and on `\l_regex_case_changed_char_int`.

```
\cs_new_protected:Npn \regex_item_caseless_equal:n #1
  {
    \if_num:w #1 = \l_regex_current_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
    \if_num: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_num:w #1 > \l_regex_current_char_int
      \reverse_if:N \if_num:w #2 < \l_regex_current_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \regex_break_true:w
      \fi:
    \fi:
    \reverse_if:N \if_num:w #1 > \l_regex_case_changed_char_int
      \reverse_if:N \if_num:w #2 < \l_regex_case_changed_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \regex_break_true:w
      \fi:
    \fi:
  }
\cs_new_protected:Npn \regex_item_caseless_geq:n #1
  {
    \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
    \reverse_if:N \if_num:w #1 > \l_regex_case_changed_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
  }
```

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

`\regex_item_equal:n` `\regex_item_range:nn` `\regex_item_geq:n`

By default, matching takes the letter case into account. Note that those functions are not protected: they will expand at the building step, hard-coding which states take care of caseless versus caseful matching.

```
\cs_new:Npn \regex_item_equal:n  { \regex_item_caseful_equal:n }
```

```
\cs_new:Npn \regex_item_range:nn { \regex_item_caseful_range:nn }
\cs_new:Npn \regex_item_geq:n    { \regex_item_caseful_geq:n }
```

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

`\regex_build_caseless:`
`\regex_build_caseful:`

Switch between caseful and caseless matching. This is only done during the building step.

```
\cs_new_protected_nopar:Npn \regex_build_caseless:
  {
    \bool_set_true:N \l_regex_caseless_bool
    \cs_set:Npn \regex_item_equal:n  { \regex_item_caseless_equal:n }
    \cs_set:Npn \regex_item_range:nn { \regex_item_caseless_range:nn }
    \cs_set:Npn \regex_item_geq:n    { \regex_item_caseless_geq:n }
  }
\cs_new_protected_nopar:Npn \regex_build_caseful:
  {
    \bool_set_false:N \l_regex_caseless_bool
    \cs_set:Npn \regex_item_equal:n  { \regex_item_caseful_equal:n }
    \cs_set:Npn \regex_item_range:nn { \regex_item_caseful_range:nn }
    \cs_set:Npn \regex_item_geq:n    { \regex_item_caseful_geq:n }
  }
```

(*End definition for* `\regex_build_caseless:` *and* `\regex_build_caseful:`*. These functions are documented on page* ??.)

`\regex_item_catcode:nT`

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:nT #1
  {
    \if_int_odd:w \int_eval:w #1 / "
        \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:
      \int_eval_end:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
```

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

`\regex_item_cs:n`

Match a control sequence (the argument is a pre-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 `\l_regex_current_token_tl` (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
      {
        \tl_set:Nn \l_regex_tmpa_regex_tl {#1}
        \bool_set_eq:NN \l_regex_saved_success_bool \g_regex_success_bool
        \exp_args:NNx \regex_match:NnTF \l_regex_tmpa_regex_tl
          {
            \exp_after:wN \exp_after:wN
            \exp_after:wN \cs_to_str:N \l_regex_current_token_tl
          }
          {
            \bool_gset_eq:NN \g_regex_success_bool \l_regex_saved_success_bool
            \regex_break_true:w
          }
          { \bool_gset_eq:NN \g_regex_success_bool \l_regex_saved_success_bool }
      }
  }
```

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

2.2.5 Grabbing digits

`\regex_get_digits:nw`
`\regex_get_digits_loop:N`
`\regex_get_digits_end:w`

Grabs digits (of category code other), skipping any intervening space, until encountering a non-digit, and places the result in a brace group after `#1`. This is used when parsing the `{` quantifier.

```
\cs_new_protected:Npn \regex_get_digits:nw #1
  {
    \tex_afterassignment:D \regex_tmp:w
    \cs_set_nopar:Npx \regex_tmp:w
      {
        \exp_not:n {#1}
        { \if_false: } } \fi:
        \regex_get_digits_aux:NN
  }
\cs_new:Npn \regex_get_digits_aux:NN #1#2
  {
    \if_meaning:w \regex_build_raw:N #1
      \if_charcode:w \c_space_token \exp_not:N #2
      \else:
        \if_num:w 9 < 1 \exp_not:N #2 \exp_stop_f:
          #2
        \else:
          \regex_get_digits_end:w #1 #2
        \fi:
      \fi:
    \else:
      \regex_get_digits_end:w #1 #2
    \fi:
    \regex_get_digits_aux:NN
```

```
  }
\cs_new:Npn \regex_get_digits_end:w #1 \fi: #2 \regex_get_digits_aux:NN
  {
    \fi: #2
    \if_false: { { \fi: } }
    #1
  }
```

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

2.2.6 More character testing

`\regex_token_if_other_digit:N`

In the replacement text, `\g{`⟨*int*⟩`}` denotes the ⟨*int*⟩-th submatch. Parsing this construction robustly requires a test of whether a token is a digit or not.

```
\prg_new_conditional:Npnn \regex_token_if_other_digit:N #1 { TF }
  {
    \if_num:w \c_nine < 1 \exp_not:N #1 \exp_stop_f:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

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

`\regex_aux_char_if_alphanumeric:NTF`
`\regex_aux_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: testing for instance with `\str_if_contains_char:nN` would be much slower. 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_aux_char_if_special:N #1 { TF }
  {
    \if_num:w `#1 < \c_ninety_one
      \if_num:w `#1 < \c_fifty_eight
        \if_num:w `#1 < \c_forty_eight
          \if_num:w `#1 < \c_thirty_two
            \prg_return_false: \else: \prg_return_true: \fi:
        \else: \prg_return_false: \fi:
      \else:
        \if_num:w `#1 < \c_sixty_five
          \prg_return_true: \else: \prg_return_false: \fi:
      \fi:
```

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

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

2.3 Building

2.3.1 Build mode

When building the NFA corresponding to a given regex, 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_build_if_in_class:TF` Test whether we are currently in a character class (at the inner-most 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_build_if_in_class:TF
  {
    \if_int_odd:w \l_regex_build_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

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

2.3.2 Helpers for building an NFA

\regex_build_new_state: Here, we add a new state to the NFA. At the end of the building phase, we want every \toks register to start with \s_stop, hence initialize the new register appropriately. Then set \l_regex_left/right_state_int to their new values.

```
\cs_new_protected_nopar:Npn \regex_build_new_state:
  {
    \tex_toks:D \l_regex_max_state_int { \s_stop }
    \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:. *This function is documented on page* ??.)

\regex_build_transition:NN
\regex_build_transitions:NNNN These functions create a new state, and put one or two transitions starting from the old current state.

```
\cs_new_protected:Npn \regex_build_transition:NN #1#2
  {
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      { #1 { \int_eval:n { #2 - \l_regex_left_state_int } } }
  }
\cs_new_protected:Npn \regex_build_transitions:NNNN #1#2#3#4
  {
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        #1 { \int_eval:n { #2 - \l_regex_left_state_int } }
        #3 { \int_eval:n { #4 - \l_regex_left_state_int } }
      }
  }
```

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

2.3.3 From regex to NFA: framework

\regex_build:n First, reset a few variables. Then use the generic framework defined in l3str to parse the regular expression once, recognizing which characters are raw characters, and which have special meanings. The search is not anchored: to achieve that, we insert state(s) responsible for repeating the match attempt on every token in the token list. The trailing \prg_do_nothing: ensure that the look-ahead done by some of the operations is harmless. Finally, \regex_build_end: adds the finishing code (checking that parentheses are properly nested, for instance).

```
\cs_new_protected:Npn \regex_build:n #1
  {
    \int_set_eq:NN \l_regex_build_mode_int \c_zero
    \regex_build:w
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
```

```
      { \regex_action_start_wildcard: }
    \regex_build_open_aux:
      \str_escape_use:NNNn
        \regex_build_i_unescaped:N
        \regex_build_i_escaped:N
        \regex_build_i_raw:N
        { #1 }
      \prg_do_nothing: \prg_do_nothing:
      \prg_do_nothing: \prg_do_nothing:
    \regex_build_end:
  }
```

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

`\regex_build_i_unescaped:N`
`\regex_build_i_escaped:N`
`\regex_build_i_raw:N`

The l3str function `\str_escape_use:NNNn` goes through the regular expression and finds the `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, and `\x` escape sequences, then distinguishes three cases: non-escaped characters, escaped characters, and "raw" characters coming from one of the escape sequences. In the particular case of regular expressions, escaped alphanumerics and non-escaped non-alphanumeric printable ascii characters may have special meanings, while everything else should be treated as a raw character.

```
\cs_new:Npn \regex_build_i_unescaped:N #1
  {
    \regex_aux_char_if_special:NTF #1
      { \exp_not:N \regex_build_special:N #1 }
      { \exp_not:N \regex_build_raw:N #1 }
  }
\cs_new:Npn \regex_build_i_escaped:N #1
  {
    \regex_aux_char_if_alphanumeric:NTF #1
      { \exp_not:N \regex_build_escaped:N #1 }
      { \exp_not:N \regex_build_raw:N #1 }
  }
\cs_new:Npn \regex_build_i_raw:N #1
  { \exp_not:N \regex_build_raw:N #1 }
```

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

`\regex_build_special:N`
`\regex_build_escaped:N`

If the control character 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 in character classes, unknown escaped alphanumeric characters raise an error, while special charcters are silently converted to raw characters.

```
\cs_new_protected:Npn \regex_build_special:N #1
  {
    \cs_if_exist_use:cF { regex_build_#1: }
      { \regex_build_raw:N #1 }
  }
\cs_new_protected:Npn \regex_build_escaped:N #1
  {
    \cs_if_exist_use:cF { regex_build_/#1: }
```

```
      { \regex_build_raw:N #1 }
  }
```

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

`\regex_build:w` Hopefully, we didn't forget to initialize anything here.

```
\cs_new_protected_nopar:Npn \regex_build:w
  {
    \int_set_eq:NN \l_regex_catcodes_default_int \c_regex_catcodes_all_int
    \int_set_eq:NN \l_regex_capturing_group_int  \c_zero
    \int_zero:N \l_regex_max_state_int
    \regex_build_new_state:
    \tl_clear:N \l_regex_class_tl
    \bool_set_true:N \l_regex_class_bool
    \int_set_eq:NN \l_regex_catcodes_int \l_regex_catcodes_default_int
  }
```

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

`\regex_build_end:` If parentheses are not nested properly, an error is raised, and the correct number of parentheses is closed. After that, we insert an instruction for the match to succeed.

```
\cs_new_protected_nopar:Npn \regex_build_end:
  {
    \regex_seq_push_int:NN \l_regex_capturing_group_seq \c_zero
    \regex_build_close_aux: \regex_build_group_:
    \seq_if_empty:NF \l_regex_capturing_group_seq
      {
        \msg_kernel_error:nnx { regex } { missing-rparen }
          { \seq_length:N \l_regex_capturing_group_seq }
        \prg_replicate:nn
          { \seq_length:N \l_regex_capturing_group_seq }
          { \regex_build_close_aux: \regex_build_group_: }
      }
    \regex_toks_put_right:Nx \l_regex_right_state_int
      { \regex_action_success: }
    \int_incr:N \l_regex_capturing_group_int
  }
```

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

`\regex_build_one:n` `\regex_build_one:x` In a class, add the argument to the current class. Outside a class, this argument is the whole "class", and we look for quantifiers.

```
\cs_new_protected:Npn \regex_build_one:n #1
  { \regex_build_one:x { \exp_not:n {#1} } }
\cs_new_protected:Npn \regex_build_one:x #1
  {
    \tl_put_right:Nx \l_regex_class_tl
      {
        \if_num:w \l_regex_catcodes_int = \c_regex_catcodes_all_int
          \exp_after:wN \use:n
        \else:
          \regex_item_catcode:nT { \int_use:N \l_regex_catcodes_int }
```

```
        \fi:
        {#1}
      }
    \if_num:w \l_regex_build_mode_int = \c_two
      \l_regex_build_mode_int = \c_zero
    \else:
      \if_num:w \l_regex_build_mode_int = \c_six
        \l_regex_build_mode_int = \c_three
      \fi:
    \fi:
    \if_int_odd:w \l_regex_build_mode_int \else:
      \exp_after:wN \regex_build_one_quantifier:
    \fi:
  }
```

(*End definition for* `\regex_build_one:n` *and* `\regex_build_one:x`*. These functions are documented on page* ??.)

`\regex_build_tmp_class:n` The argument is the target state if the test succeeds.

```
\cs_new:Npn \regex_build_tmp_class:n #1
  {
    \exp_not:o \l_regex_class_tl
    \bool_if:NTF \l_regex_class_bool
      { \regex_break_point:TF { \regex_action_cost:n {#1} } { } }
      { \regex_break_point:TF { } { \regex_action_cost:n {#1} } }
  }
```

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

`\regex_class_put:N` `\regex_class_put:NN` This function is used to rapidly put a character in a character class being built, avoiding some tests implied by `\regex_build_one:x`.

```
\cs_new_protected:Npn \regex_class_put:N #1
  {
    \tl_put_right:Nx \l_regex_class_tl
      { \regex_item_equal:n { \int_value:w `#1 ~ } }
  }
\cs_new_protected:Npn \regex_class_put:NN #1#2
  {
    \tl_put_right:Nx \l_regex_class_tl
      { \regex_item_range:nn { \int_value:w `#1 ~ } { \int_value:w `#2 ~ } }
  }
```

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

`\regex_build_raw_alphanum_is_error:N` Within character classes, some special character and escape sequences do not have any meaning. Special characters are interpreted as their "raw" counterpart, but escape sequences without a meaning, *e.g.*, `\A`, are most likely an error: this should be `A`.

```
\cs_new_protected:Npn \regex_build_raw_alphanum_is_error:N #1
  {
    \regex_aux_char_if_alphanumeric:NTF #1
      { \msg_kernel_error:nnx { regex } { class-bad-escape } { #1 } }
      { \regex_build_raw:N #1 }
  }
```

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

`\regex_build_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_build_raw:N #1#2#3
  {
    \regex_build_if_in_class:TF
      { \str_if_eq:nnTF {#2#3} { \regex_build_special:N - } }
      { \use_ii:nn }
      { \regex_class_range:Nw #1 }
      {
        \regex_build_one:x { \regex_item_equal:n { \int_value:w `#1 ~ } }
        #2 #3
      }
  }
```

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

`\regex_class_range:Nw` We have just read a raw character followed by a dash; this should be followed by an end-point for the range. If the following character is an escaped alphanumeric, or if it is an unescaped right bracket, then we have an error, so we put the initial character and the dash back, as raw characters. Otherwise, build the range, checking that it is in the right order, and optimizing for equal end-points.

```
\cs_new_protected:Npn \regex_class_range:Nw #1#2#3
  {
    \token_if_eq_meaning:NNTF #2 \regex_build_escaped:N % [
      { \use_i:nn } { \str_if_eq:nnTF { #2#3 } { \regex_build_special:N ] } }
      {
        \msg_kernel_warning:nnx { regex } { end-range-missing } % [
          { #1 - \if_meaning:w #3 ] \else: \c_backslash_str \fi: #3 }
        \regex_class_put:N #1
        \regex_class_put:N -
        #2#3
      }
      {
        \if_num:w `#1 > `#3 \exp_stop_f:
          \msg_kernel_error:nnxx { regex } { backwards-range } {#1} {#3}
        \else:
          \if_num:w `#1 = `#3 \exp_stop_f:
            \regex_class_put:N #3
          \else:
            \regex_class_put:NN #1#3
          \fi:
        \fi:
      }
  }
```

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

2.3.4 Character properties

\regex_build_.: In a class, the dot has no special meaning. Outside, insert \regex_item_dot:, which matches any character or control sequence, and refuses −2, which marks the end of the token list.

```
\cs_new_protected_nopar:cpn { regex_build_.: }
  {
    \regex_build_if_in_class:TF
      { \regex_build_raw:N . }
      { \regex_build_one:x \regex_item_dot: }
  }
```

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

\regex_build_/d: \regex_build_/D: \regex_build_/h: \regex_build_/H: \regex_build_/s: \regex_build_/S: \regex_build_/v: \regex_build_/V: \regex_build_/w: \regex_build_/W: \regex_build_/N: The constants \c_regex_d_tl, *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.

```
\tl_map_inline:nn { dDhHsSvVwWN }
  {
    \cs_new_protected_nopar:cpx { regex_build_/#1: }
      { \regex_build_one:n \exp_not:c { c_regex_#1_tl } }
  }
```

(*End definition for* \regex_build_/d: *and* \regex_build_/D:. *These functions are documented on page* ??.)

2.3.5 Anchoring and simple assertions

\regex_build_simple_assertion:Nn Assertions are not allowed within character classes: the raw character #1 is inserted instead. Otherwise, use the test #2; if the assertion is successful, move from the left state to the right state.

```
\cs_new_protected:Npn \regex_build_simple_assertion:Nn #1#2
  {
    \regex_build_if_in_class:TF
      { \regex_build_raw_alphanum_is_error:N #1 }
      {
        \regex_build_new_state:
        \regex_toks_put_right:Nx \l_regex_left_state_int
          {
            \exp_not:n {#2}
              {
                \regex_action_free:n
                  {
                    \int_eval:n
                      { \l_regex_right_state_int - \l_regex_left_state_int }
                  }
              }
          }
        %^^A \regex_assertion_quantifier:
      }
  }
```

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

`\regex_build_^:`
`\regex_build_/A:`
`\regex_build_/G:`
`\regex_build_$:`
`\regex_build_/Z:`
`\regex_build_/z:`

Anchoring at the start corresponds to checking that the current character is the first in the token list. Anchoring to the beginning of the match attempt uses `\l_regex_start_step_int` instead of `\c_zero`. End anchors match the end of the token list, marked by a character code of -2.

```
\cs_new_protected_nopar:cpn { regex_build_^: }
  {
    \regex_build_simple_assertion:Nn ^
      { \int_compare:nNnT \l_regex_min_step_int = \l_regex_current_step_int }
  }
\cs_new_protected_nopar:cpn { regex_build_/A: }
  {
    \regex_build_simple_assertion:Nn A
      { \int_compare:nNnT \l_regex_min_step_int = \l_regex_current_step_int }
  }
\cs_new_protected_nopar:cpn { regex_build_/G: }
  {
    \regex_build_simple_assertion:Nn G
      { \int_compare:nNnT \l_regex_start_step_int = \l_regex_current_step_int }
  }
\cs_new_protected_nopar:cpn { regex_build_$: } % $
  {
    \regex_build_simple_assertion:Nn $ % $
      { \int_compare:nNnT \l_regex_current_char_int < \c_minus_one }
  }
\cs_new_protected_nopar:cpn { regex_build_/Z: }
  {
    \regex_build_simple_assertion:Nn Z
      { \int_compare:nNnT \l_regex_current_char_int < \c_minus_one }
  }
\cs_new_protected_nopar:cpn { regex_build_/z: }
  {
    \regex_build_simple_assertion:Nn z
      { \int_compare:nNnT \l_regex_current_char_int < \c_minus_one }
  }
```

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

`\regex_build_/b:`
`\regex_build_/B:`
`\regex_if_word_boundary:TF`

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. Case sensitivity does not change word boundaries.

```
\cs_new_protected_nopar:cpn { regex_build_/b: }
  {
    \regex_build_simple_assertion:Nn b
      { \regex_if_word_boundary:TF \use:n \use_none:n }
  }
\cs_new_protected_nopar:cpn { regex_build_/B: }
  {
    \regex_build_simple_assertion:Nn B
```

```
      { \regex_if_word_boundary:TF { } }
  }
\cs_new_protected_nopar:Npn \regex_if_word_boundary:TF
  {
    \group_begin:
      \int_set_eq:NN \l_regex_current_char_int \l_regex_last_char_int
      \c_regex_w_tl
      \regex_break_point:TF
        { \group_end: \c_regex_W_tl \regex_item_equal:n { -2 } }
        { \group_end: \c_regex_w_tl }
    \regex_break_point:TF
  }
```

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

2.3.6 Entering and exiting character classes

`\regex_build_[:` In a class, left brackets mean nothing. Outside a class, this starts a class, whose first characters may have a special meaning.

```
\cs_new_protected_nopar:cpn { regex_build_[: }
  {
    \regex_build_if_in_class:TF
      { \regex_build_raw:N [ }
      { \regex_class_first:NNNN }
  }
```

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

`\regex_build_]:` 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. If we are still in a class after leaving one, then this is the case `[...\cL[...]...]`, and we insert the relevant closing material in `\l_regex_class_tl`. Otherwise look for quantifiers.

```
\cs_new_protected:cpn { regex_build_]: }
  {
    \regex_build_if_in_class:TF
      {
        \if_num:w \l_regex_build_mode_int > \c_sixteen
          \tl_set:Nx \l_regex_class_tl
            {
              \exp_not:o \l_regex_class_saved_tl
              \if_num:w \l_regex_catcodes_int < \c_regex_catcodes_all_int
                \regex_item_catcode:nT { \int_use:N \l_regex_catcodes_int }
              \else:
                \exp_after:wN \use:n
              \fi:
                {
                  \exp_not:o \l_regex_class_tl
                  \bool_if:NF \l_regex_class_bool
                    { \regex_break_point:TF { } { \regex_break_true:w } }
                }
            }
```

```
          \bool_set_eq:NN \l_regex_class_bool \l_regex_class_saved_bool
        \fi:
        \tex_advance:D \l_regex_build_mode_int - \c_fifteen
        \tex_divide:D \l_regex_build_mode_int \c_thirteen
        \if_int_odd:w \l_regex_build_mode_int \else:
          \exp_after:wN \regex_build_one_quantifier:
        \fi:
      }
      { \regex_build_raw:N ] }
  }
```

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

`\regex_class_first:NNNN` This starts a class. Change the mode by appending 3 to it, and reset the variables `\l_regex_class_tl` and `\l_regex_bool_tl`. In the special case of mode 63 (`[\c[...]]`), we open a group, to avoid overriding the setting of `\l_regex_class_bool` and `\l_regex_class_tl`; the group ends at the matching right bracket. If the first character is `^`, then the class is inverted. We keep track of this in `\l_regex_class_bool`. If the next character is a right bracket, then it should be changed to a raw one (dirty hack here; the `F` argument of `\str_if_eq:nnTF` is the trailing `#3`).

```
\cs_new_protected:Npn \regex_class_first:NNNN #1#2#3#4
  {
    \l_regex_build_mode_int = \int_value:w \l_regex_build_mode_int 3 ~ %
    \if_num:w \l_regex_build_mode_int > \c_sixteen
      \tl_set_eq:NN \l_regex_class_saved_tl \l_regex_class_tl
      \bool_set_eq:NN \l_regex_class_saved_bool \l_regex_class_bool
    \fi:
    \token_if_eq_meaning:NNTF #1 \regex_build_special:N
      {
        \token_if_eq_charcode:NNTF #2 ^
          {
            \bool_set_false:N \l_regex_class_bool % [
            \str_if_eq:nnTF {#3#4} { \regex_build_special:N ] }
              { \regex_build_raw:N }
          }
          { % [
            \token_if_eq_charcode:NNTF #2 ]
              { \regex_build_raw:N #2 }
              { #1 #2 }
          }
      }
      { #1 #2 }
    #3 #4
  }
```

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

2.3.7 Catcodes and csnames

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

```
\cs_new_protected:cpn { regex_build_/c: }
  {
    \if_case:w \l_regex_build_mode_int
      \exp_after:wN \regex_build_c_aux:wNN
    \or: \or: \or: \exp_after:wN \regex_build_c_aux:wNN
    \fi:
    \msg_kernel_error:nn { regex } { c-bad-mode }
    \s_stop
  }
```

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

`\regex_build_c_aux:wNN` 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:Npn \regex_build_c_aux:wNN #1 \s_stop #2#3
  {
    \token_if_eq_meaning:NNTF #2 \regex_build_raw:N
      {
        \cs_if_exist:cTF { c_regex_catcode_#3_int }
          {
            \int_set_eq:Nc \l_regex_catcodes_int { c_regex_catcode_#3_int }
            \l_regex_build_mode_int
              = \if_case:w \l_regex_build_mode_int \c_two \else: \c_six \fi:
          }
      }
      { \cs_if_exist_use:cF { regex_build_c_#3: } }
          {
            \msg_kernel_error:nnxx { regex } { c-command }
              { regular~expression } { #3 }
            #2 #3
          }
  }
```

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

`\regex_build_c_{:` The case of a left brace is easy, based on what we have done so far: in a group, build the regular expression, after changing the mode to forbid nesting `\c`.

```
\cs_new_protected:cpn { regex_build_c_ \c_lbrace_str : }
  {
    \group_begin:
      \l_regex_build_mode_int
        = - \if_case:w \l_regex_build_mode_int \c_two \else: \c_six \fi:
      \regex_build:w
  }
```

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

`\regex_build_c_[:`
`\regex_build_c_lbrack_loop:NN`
`\regex_build_c_lbrack_end:`
`\regex_build_add:N`

```
\cs_new_protected:cpn { regex_build_c_[: } #1#2
  {
    \int_zero:N \l_regex_catcodes_int
```

```
    \str_if_eq:nnTF { #1 #2 } { \regex_build_special:N ^ }
      {
        \bool_set_false:N \l_regex_catcodes_bool
        \regex_build_c_lbrack_loop:NN
      }
      {
        \bool_set_true:N \l_regex_catcodes_bool
        \regex_build_c_lbrack_loop:NN
        #1#2
      }
  }
\cs_new_protected:Npn \regex_build_c_lbrack_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \regex_build_raw:N
      {
        \cs_if_exist:cTF { c_regex_catcode_#2_int }
          {
            \exp_args:Nc \regex_build_c_lbrack_add:N
              { c_regex_catcode_#2_int }
            \regex_build_c_lbrack_loop:NN
          }
      }
      { % [
        \token_if_eq_charcode:NNTF #2 ]
          { \regex_build_c_lbrack_end: }
      }
          {
            \msg_kernel_error:nnxx { regex } { c-command }
              { regular-expression } { #2 }
            \regex_build_c_lbrack_end:
          }
  }
\cs_new_protected_nopar:Npn \regex_build_c_lbrack_end:
  {
    \l_regex_build_mode_int
      = \if_case:w \l_regex_build_mode_int \c_two \else: \c_six \fi:
    \if_meaning:w \c_false_bool \l_regex_catcodes_bool
      \int_set:Nn \l_regex_catcodes_int
        { \c_regex_catcodes_all_int - \l_regex_catcodes_int }
    \fi:
  }
\cs_new_protected:Npn \regex_build_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:
  }
```

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

\regex_build}: Non-escaped right braces are only special if they appear when building the regular expression for a csname. Otherwise, replace the brace with an escaped brace.

```
\cs_new_protected:cpn { regex_build_ \c_rbrace_str : }
  { %^^A incorrect recovery for \c{...(...[...} with missing ) or ].
    \int_compare:nNnTF \l_regex_build_mode_int < \c_zero
      {
        \use:c { regex_build_/Z: }
        \regex_toks_put_right:Nx \l_regex_right_state_int
          { \regex_action_success: }
        \int_incr:N \l_regex_capturing_group_int
        \use:x
          {
            \group_end:
              \regex_build_one:n
                { \regex_item_cs:n { \regex_set_aux:N ? } }
          }
      }
      { \exp_after:wN \regex_build_raw:N \c_rbrace_str }
  }
```

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

2.3.8 Quantifiers

\regex_build_quantifier:w This looks ahead and finds any quantifier (control character equal to either of `?+*{`). When all characters for the quantifier are found, the corresponding function is called.

```
\cs_new_protected:Npn \regex_build_quantifier:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \regex_build_special:N
      {
        \cs_if_exist_use:cF { regex_build_quantifier_#2:w }
          {
            \regex_build_quantifier_end:nn { } { }
            #1 #2
          }
      }
      {
        \regex_build_quantifier_end:nn { } { }
        #1 #2
      }
  }
```

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

\regex_build_quantifier_?:w
\regex_build_quantifier_*:w
\regex_build_quantifier_+:w
For each "basic" quantifier, `?`, `*`, `+`, feed the correct arguments to `\regex_build_quantifier_aux:nnNN`.

```
\cs_new_protected_nopar:cpn { regex_build_quantifier_?:w }
  { \regex_build_quantifier_aux:nnNN { } { ? } }
\cs_new_protected_nopar:cpn { regex_build_quantifier_*:w }
  { \regex_build_quantifier_aux:nnNN { } { * } }
```

```
\cs_new_protected_nopar:cpn { regex_build_quantifier_+:w }
  { \regex_build_quantifier_aux:nnNN { } { + } }
```

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

`\regex_build_quantifier_aux:nnNN` Once the "main" quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark), and calls the appropriate function. Here `#1` holds some extra arguments that the final function needs in the case of braced constructions, and is empty otherwise.

```
\cs_new_protected:Npn \regex_build_quantifier_aux:nnNN #1#2#3#4
  {
    \str_if_eq:nnTF { #3 #4 } { \regex_build_special:N ? }
      { \regex_build_quantifier_end:nn { #2 #4 } {#1} }
      {
        \regex_build_quantifier_end:nn { #2 } {#1}
        #3 #4
      }
  }
```

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

`\regex_build_quantifier_{:w` `\regex_build_quantifier_lbrace:n` `\regex_build_quantifier_lbrace:nw` `\regex_build_quantifier_lbrace:nnw` Three possible syntaxes: {⟨*int*⟩}, {⟨*int*⟩,}, or {⟨*int*⟩,⟨*int*⟩}.

```
\cs_new_protected_nopar:cpn { regex_build_quantifier_ \c_lbrace_str :w }
  { \regex_get_digits:nw { \regex_build_quantifier_lbrace:n } }
\cs_new_protected:Npn \regex_build_quantifier_lbrace:n #1
  {
    \tl_if_empty:nTF {#1}
      {
        \regex_build_quantifier_end:nn { } { }
        \exp_after:wN \regex_build_raw:N \c_lbrace_str
      }
      { \regex_build_quantifier_lbrace:nw {#1} }
  }
\cs_new_protected:Npx \regex_build_quantifier_lbrace:nw #1#2#3
  {
    \exp_not:N \prg_case_str:nnn { #2 #3 }
      {
        { \exp_not:N \regex_build_special:N , }
          {
            \exp_not:N \regex_get_digits:nw
              { \exp_not:N \regex_build_quantifier_lbrace:nnw {#1} }
          }
        { \exp_not:N \regex_build_special:N \c_rbrace_str }
          { \exp_not:N \regex_build_quantifier_end:nn {n} { {#1} } }
      }
      {
        \exp_not:N \regex_build_quantifier_end:nn { } { }
        \exp_not:N \regex_build_raw:N \c_lbrace_str #1#2
      }
  }
\cs_new_protected:Npn \regex_build_quantifier_lbrace:nnw #1#2#3#4
```

```
  {
    \str_if_eq:xxTF
      { \exp_not:n {#3#4} }
      { \exp_not:N \regex_build_special:N \c_rbrace_str }
      {
        \tl_if_empty:nTF {#2}
          { \regex_build_quantifier_aux:nnNN { {#1} } { n* } }
          { \regex_build_quantifier_aux:nnNN { {#1} {#2} } { nn } }
      } %^^A todo: catch the case m>n
      {
        \regex_build_quantifier_end:nn { } { }
        \use:x
          {
            \exp_args:No \tl_map_function:nN
              { \c_lbrace_str #1 , #2 }
              \regex_build_raw:N
          }
        #3 #4
      }
  }
```

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

`\regex_build_quantifier_end:nn` When all quantifiers are found, we will call the relevant `\regex_build_one/group_-⟨quantifiers⟩:` function.

```
\cs_new_protected:Npn \regex_build_quantifier_end:nn #1#2
  {
    \use:c { regex_build_ \l_regex_one_or_group_tl _ #1 : } #2
    \tl_clear:N \l_regex_class_tl
    \bool_set_true:N \l_regex_class_bool
    \int_set_eq:NN \l_regex_catcodes_int \l_regex_catcodes_default_int
  }
```

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

2.3.9 Quantifiers for one character or character class

`\regex_build_one_quantifier:` Used for one single character, or a character class. Contrarily to `\regex_build_group_-quantifier:`, we don't need to keep track of submatches, and no thread can be created within one repetition, so things are relatively easy.

```
\cs_new_protected_nopar:Npn \regex_build_one_quantifier:
  {
    \tl_set:Nx \l_regex_one_or_group_tl { one }
    \regex_build_quantifier:w
  }
```

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

`\regex_build_one_:` If no quantifier is found, then the character or character class should just be built into a transition from the current "right" state to a new state.

```
\cs_new_protected_nopar:Npn \regex_build_one_:
```

```
  {
    \regex_build_transition:NN
      \regex_build_tmp_class:n \l_regex_right_state_int
  }
```

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

`\regex_build_one_?:`
`\regex_build_one_??:`

The two transitions are a costly transition controlled by the character class, and a free transition, both going to a common new state. The only difference between the greedy and lazy operators is the order of transitions.

```
\cs_new_protected_nopar:cpn { regex_build_one_?: }
  {
    \regex_build_transitions:NNNN
      \regex_build_tmp_class:n \l_regex_right_state_int
      \regex_action_free:n     \l_regex_right_state_int
  }
\cs_new_protected_nopar:cpn { regex_build_one_??: }
  {
    \regex_build_transitions:NNNN
      \regex_action_free:n     \l_regex_right_state_int
      \regex_build_tmp_class:n \l_regex_right_state_int
  }
```

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

`\regex_build_one_*:`
`\regex_build_one_*?:`

Build a costly transition going from the current state to itself, and a free transition moving to a new state.

```
\cs_new_protected_nopar:cpn { regex_build_one_*: }
  {
    \regex_build_transitions:NNNN
      \regex_build_tmp_class:n \l_regex_left_state_int
      \regex_action_free:n     \l_regex_right_state_int
  }
\cs_new_protected_nopar:cpn { regex_build_one_*?: }
  {
    \regex_build_transitions:NNNN
      \regex_action_free:n     \l_regex_right_state_int
      \regex_build_tmp_class:n \l_regex_left_state_int
  }
```

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

`\regex_build_one_+:`
`\regex_build_one_+?:`

Build a transition from the current state to a new state, controlled by the character class, then build two transitions from this new state to the original state (for repetition) and to another new state (to move on to the rest of the pattern).

```
\cs_new_protected_nopar:cpn { regex_build_one_+: }
  {
    \regex_build_one_:
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \regex_build_transitions:NNNN
      \regex_action_free:n \l_regex_tmpa_int
```

```
      \regex_action_free:n \l_regex_right_state_int
  }
\cs_new_protected_nopar:cpn { regex_build_one_+?: }
  {
    \regex_build_one_:
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \regex_build_transitions:NNNN
      \regex_action_free:n \l_regex_right_state_int
      \regex_action_free:n \l_regex_tmpa_int
  }
```

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

`\regex_build_one_n:` `\regex_build_one_n?:` This function is called in case the syntax is {⟨*int*⟩}. Greedy and lazy operators are identical, since the number of repetitions is fixed. Simply repeat `#1` times the effect of `\regex_build_one_:`.

```
\cs_new_protected:Npn \regex_build_one_n: #1
  { \prg_replicate:nn {#1} { \regex_build_one_: } }
\cs_new_eq:cN { regex_build_one_n?: } \regex_build_one_n:
```

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

`\regex_build_one_n*:` `\regex_build_one_n*?:` This function is called in case the syntax is {⟨*int*⟩,}.

```
\cs_new_protected:cpx { regex_build_one_n*: } #1
  {
    \exp_not:N \prg_replicate:nn {#1} { \exp_not:N \regex_build_one_: }
    \exp_not:c { regex_build_one_*: }
  }
\cs_new_protected:cpx { regex_build_one_n*?: } #1
  {
    \exp_not:N \prg_replicate:nn {#1} { \exp_not:N \regex_build_one_: }
    \exp_not:c { regex_build_one_*?: }
  }
```

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

`\regex_build_one_nn:` `\regex_build_one_nn?:` `\regex_build_one_nn_aux:Nnn` This function is called when the syntax is {⟨*int*⟩,⟨*int*⟩}.

```
\cs_new_protected:Npn \regex_build_one_nn_aux:Nnn #1#2#3
  {
    \prg_replicate:nn {#2} { \regex_build_one_: }
    \prg_replicate:nn {#3-#2} {#1}
  }
\cs_new_protected_nopar:Npx \regex_build_one_nn:
  { \regex_build_one_nn_aux:Nnn \exp_not:c { regex_build_one_?: } }
\cs_new_protected_nopar:cpx { regex_build_one_nn?: }
  { \regex_build_one_nn_aux:Nnn \exp_not:c { regex_build_one_??: } }
```

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

2.3.10 Groups and alternation

We support the syntax $\langle expr1 \rangle \ldots \langle expr_n \rangle$ for alternations.

\regex_build_(:
\regex_build_):
\regex_build_open_aux:
\regex_build_|:
\regex_build_begin_alternation:
\regex_build_end_alternation:

Grouping and alternation go together.

- Allocate the next available number for the end vertex of the alternation/group and store it on a stack (so that nested alternations work).
- Put free transitions to separate all cases of the alternation.
- Build each branch separately, and merge them to the common end-node.
- Test for a quantifier, and if needed, transfer the initial vertex to a new vertex.

```
\cs_new_protected:cpn { regex_build_(: } #1#2
  {
    \regex_build_if_in_class:TF
      {
        \regex_build_raw:N (
        #1 #2
      }
      {
        \str_if_eq:nnTF { #1 #2 } { \regex_build_special:N ? }
          { \regex_build_special_group:NN }
          {
            \int_incr:N \l_regex_capturing_group_int
            \regex_seq_push_int:NN
              \l_regex_capturing_group_seq \l_regex_capturing_group_int
            \regex_build_open_aux:
            #1 #2
          }
      }
  }
\cs_new_protected_nopar:Npn \regex_build_open_aux:
  {
    \regex_build_new_state:
    \regex_seq_push_int:NN \l_regex_left_state_seq  \l_regex_left_state_int
    \regex_seq_push_int:NN \l_regex_right_state_seq \l_regex_right_state_int
    \bool_if:NTF \l_regex_caseless_bool
      { \seq_push:Nn \l_regex_end_group_seq \regex_build_caseless: }
      { \seq_push:Nn \l_regex_end_group_seq \regex_build_caseful: }
    \seq_push:Nn \l_regex_end_alternation_seq { }
    \regex_build_begin_alternation:
  }
\cs_new_protected_nopar:cpn { regex_build_|: }
  {
    \regex_build_if_in_class:TF { \regex_build_raw:N | }
      {
        \regex_build_end_alternation:
        \regex_build_begin_alternation:
```

```
      }
  }
\cs_new_protected_nopar:cpn { regex_build_): }
  {
    \regex_build_if_in_class:TF { \regex_build_raw:N ) }
      {
        \seq_if_empty:NTF \l_regex_capturing_group_seq
          { \msg_kernel_error:nn { regex } { extra-rparen } }
          {
            \regex_build_close_aux:
            \regex_build_group_quantifier:
          }
      }
  }
\cs_new_protected_nopar:Npn \regex_build_close_aux:
  {
    \regex_build_end_alternation:
    \regex_seq_pop_int:NN \l_regex_left_state_seq  \l_regex_left_state_int
    \regex_seq_pop_int:NN \l_regex_right_state_seq \l_regex_right_state_int
    \regex_seq_pop_use:N \l_regex_end_group_seq
    \seq_pop:NN \l_regex_end_alternation_seq \l_regex_tmpa_tl
  }
```

Building each branch.

```
\cs_new_protected_nopar:Npn \regex_build_begin_alternation:
  {
    \regex_build_new_state:
    \regex_seq_get_int:NN \l_regex_left_state_seq \l_regex_left_state_int
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \regex_action_free:n
          {
            \int_eval:n
              { \l_regex_right_state_int - \l_regex_left_state_int }
          }
      }
  }
\cs_new_protected_nopar:Npn \regex_build_end_alternation:
  {
    \int_set_eq:NN \l_regex_left_state_int \l_regex_right_state_int
    \regex_seq_get_int:NN \l_regex_right_state_seq \l_regex_right_state_int
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \regex_action_free:n
          {
            \int_eval:n
              { \l_regex_right_state_int - \l_regex_left_state_int }
          }
      }
    \regex_seq_get_use:N \l_regex_end_alternation_seq
```

```
  }
```

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

`\regex_build_special_group:NN` Same method as elsewhere: if the combination `(?#1` is known, then use that. Otherwise, treat the question mark as if it had been escaped.

```
\cs_new_protected:Npn \regex_build_special_group:NN #1#2
  {
    \cs_if_exist_use:cF { regex_build_special_group_\token_to_str:N #2 : }
      {
        \msg_kernel_error:nnx { regex } { unsupported }
          { (? \token_to_str:N #2 } %)
        \regex_build_special:N ( % )
        \regex_build_raw:N ?
        #1 #2
      }
  }
```

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

`\regex_build_special_group_::` Non-capturing groups are like capturing groups, except that we set the group id to `-1`, which will then inhibit submatching in `\regex_build_group_submatches:NN`. The group number is not increased.

```
\cs_new_protected_nopar:cpn { regex_build_special_group_:: }
  {
    \regex_seq_push_int:NN \l_regex_capturing_group_seq \c_minus_one
    \regex_build_open_aux:
  }
```

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

`\regex_build_special_group_|:` The special group `(?|..|..)` is non-capturing (hence we set the `capturing_group` to -1), and resets the group number in each branch of the alternation. We use a variant of `\regex_build_open_aux:`, adding some code to be performed at every alternation, and at the end of the group. Namely, we keep track of the maximal value that `\l_regex_capturing_group_int` takes, and restore that value when the group end, and in every branch, we reset the capturing group number.

```
\cs_new_protected_nopar:cpn { regex_build_special_group_|: }
  {
    \regex_seq_push_int:NN \l_regex_capturing_group_seq \c_minus_one
    \regex_build_new_state:
    \regex_seq_push_int:NN \l_regex_left_state_seq  \l_regex_left_state_int
    \regex_seq_push_int:NN \l_regex_right_state_seq \l_regex_right_state_int
    \seq_push:Nx \l_regex_end_alternation_seq
      {
        \exp_not:N \int_compare:nNnT
          \l_regex_capturing_group_int
          > \l_regex_capturing_group_max_int
          {
            \int_set_eq:NN
```

```
              \l_regex_capturing_group_max_int
              \l_regex_capturing_group_int
          }
        \int_set:Nn \l_regex_capturing_group_int
          { \int_use:N \l_regex_capturing_group_int }
      }
    \seq_push:Nx \l_regex_end_group_seq
      {
        \bool_if:NTF \l_regex_caseless_bool
          \regex_build_caseless:
          \regex_build_caseful:
        \int_set_eq:NN
          \l_regex_capturing_group_int
          \l_regex_capturing_group_max_int
      }
    \regex_build_begin_alternation:
  }
```

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

`\regex_build_special_group_i:`
`\regex_build_special_group_-:`
`\regex_build_options:NNN`
`\regex_build_option_+i:`
`\regex_build_option_-i:`

The match can be made case-insensitive by setting the option with `(?i)`.

```
\cs_new_protected_nopar:Npn \regex_build_special_group_i:
  {
    \regex_build_options:NNN +
    \regex_build_raw:N i
  }
\cs_new_protected_nopar:cpn { regex_build_special_group_-: }
  {
    \regex_build_options:NNN -
  }
\cs_new_protected:Npn \regex_build_options:NNN #1#2#3
  {
    \token_if_eq_meaning:NNTF \regex_build_raw:N #2
      {
        \cs_if_exist_use:cF { regex_build_option_#1#3: }
          { \msg_error:nnx { regex } { unknown-option } { #3 } }
        \regex_build_options:NNN #1
      }
      {
        \prg_case_str:nnn { #3 }
          { % (
            { ) } { }
            { - } { \regex_build_options:NNN - }
          }
          { \msg_error:nnx { regex } { invalid-in-option } { #3 } }
      }
  }
\cs_new_protected_nopar:cpn { regex_build_option_+i: }
  {
    \regex_build_caseless:
    \cs_set_eq:NN \regex_match_loop_case_hook:
```

```
      \regex_match_loop_caseless_hook:
  }
\cs_new_protected_nopar:cpn { regex_build_option_-i: }
  { \regex_build_caseful: }
```

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

2.3.11 Quantifiers for groups

`\regex_build_group_quantifier:` Used for one group. We need to keep track of submatches, threads can be created within one repetition, so things are hard. The code for the group that was just built starts at `\l_regex_left_state_int` and ends at `\l_regex_right_state_int`.

```
\cs_new_protected_nopar:Npn \regex_build_group_quantifier:
  {
    \tl_set:Nn \l_regex_one_or_group_tl { group }
    \regex_build_quantifier:w
  }
```

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

`\regex_build_group_submatches:NN` Once the quantifier is found by `\regex_build_quantifier:w`, we insert the code for tracking submatches.

```
\cs_new_protected:Npn \regex_build_group_submatches:NN #1#2
  {
    \seq_pop:NN \l_regex_capturing_group_seq \l_regex_tmpa_tl
    \int_compare:nNnF { \l_regex_tmpa_tl } < \c_zero
      {
        \regex_toks_put_left:Nx #1
          { \regex_action_submatch:n { \l_regex_tmpa_tl < } }
        \regex_toks_put_left:Nx #2
          { \regex_action_submatch:n { \l_regex_tmpa_tl > } }
      }
  }
```

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

`\regex_build_group_:` When there is no quantifier, the group is simply inserted as is, and we only need to track submatches, and move to a new state.

```
\cs_new_protected_nopar:Npn \regex_build_group_:
  {
    \regex_build_group_submatches:NN
      \l_regex_left_state_int \l_regex_right_state_int
    \regex_build_transition:NN
      \regex_action_free:n \l_regex_right_state_int
  }
```

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

`\regex_build_group_shift:N` Most quantifiers require to add an extra state before the group. This is done by shifting the current contents of the `\tex_toks:D \l_regex_tmpa_int` to a new state.

```
\cs_new_protected:Npn \regex_build_group_shift:N #1
  {
```

```
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \regex_build_new_state:
    \tex_toks:D \l_regex_right_state_int = \tex_toks:D \l_regex_tmpa_int
    \regex_toks_put_left:Nx \l_regex_right_state_int
      {
        \int_set:Nn \l_regex_current_state_int
          { \int_use:N \l_regex_tmpa_int } % ^^A here we lie!
      }
    \cs_set:Npx \regex_tmp:w
      {
        \tex_toks:D \l_regex_tmpa_int
          {
            \s_stop
            #1 { \int_eval:n { \l_regex_right_state_int - \l_regex_tmpa_int } }
          }
      }
    \regex_tmp:w
    \regex_build_group_submatches:NN
      \l_regex_right_state_int \l_regex_left_state_int
  }
```

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

`\regex_build_group_qs_aux:NN`
`\regex_build_group_?:`
`\regex_build_group_??:`
`\regex_build_group_*:`
`\regex_build_group_*?:`

Shift the state at which the group begins using `\regex_build_group_shift:N`, then add two transitions. The first transition is taken once the group has been traversed: in the case of `?` and `??`, we should exit by going to `\l_regex_right_state_int`, while for `*` and `*?` we loop by going to `\l_regex_tmpa_int`. The second transition corresponds to skipping the group; it has lower priority (`put_right`) for greedy operators, and higher priority (`put_left`) for lazy operators.

```
\cs_new_protected:Npn \regex_build_group_qs_aux:NN #1#2
  {
    \regex_build_group_shift:N \regex_action_free:n
    \int_set_eq:NN \l_regex_right_state_int \l_regex_left_state_int
    \regex_build_transition:NN \regex_action_free:n #1
    #2 \l_regex_tmpa_int
      {
        \regex_action_free:n
          { \int_eval:n { \l_regex_right_state_int - \l_regex_tmpa_int } }
      }
  }
\cs_new_protected_nopar:cpn { regex_build_group_?: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_right_state_int \regex_toks_put_right:Nx
  }
\cs_new_protected_nopar:cpn { regex_build_group_??: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_right_state_int \regex_toks_put_left:Nx
  }
```

```
\cs_new_protected_nopar:cpn { regex_build_group_*: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_tmpa_int \regex_toks_put_right:Nx
  }
\cs_new_protected_nopar:cpn { regex_build_group_*?: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_tmpa_int \regex_toks_put_left:Nx
  }
```

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

`\regex_build_group_+:` `\regex_build_group_+?:` Insert the submatch tracking code, then add two transitions from the current state to the left end of the group (repeating the group), and to a new state (to carry on with the rest of the regular expression).

```
\cs_new_protected_nopar:cpn { regex_build_group_+: }
  {
    \regex_build_group_submatches:NN
      \l_regex_left_state_int \l_regex_right_state_int
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \regex_build_transitions:NNNN
      \regex_action_free:n \l_regex_tmpa_int
      \regex_action_free:n \l_regex_right_state_int
  }
\cs_new_protected_nopar:cpn { regex_build_group_+?: }
  {
    \regex_build_group_submatches:NN
      \l_regex_left_state_int \l_regex_right_state_int
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \regex_build_transitions:NNNN
      \regex_action_free:n \l_regex_right_state_int
      \regex_action_free:n \l_regex_tmpa_int
  }
```

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

`\regex_build_group_n_aux:n` The braced quantifiers rely on replicating the states corresponding to the group that has just been built, and joining the right state of each copy to the left state of the next copy. Once this function has been run, `\l_regex_tmpa_int` points to the last copy of the initial left-most state, `\l_regex_left_state_int` has its initial value, and `\l_regex_right_state_int` points to the last copy of the initial right-most state. Furthermore, `\l_regex_max_state_int` is set appropriately to the largest allocated `\toks` register.

```
\cs_new_protected:Npn \regex_build_group_n_aux:n #1
  {
    \regex_toks_put_right:Nx \l_regex_right_state_int
      {
        \regex_action_free:n
          {
            \int_eval:n %^^A todo: document why that value.
              { \l_regex_max_state_int - \c_one - \l_regex_left_state_int }
```

```
          }
      }
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \int_set_eq:NN \l_regex_tmpb_int \l_regex_max_state_int
    \int_set:Nn \l_regex_max_state_int
      {
        \l_regex_left_state_int
        + #1 * ( \l_regex_max_state_int - \l_regex_left_state_int )
      }
    \int_while_do:nNnn \l_regex_tmpb_int < \l_regex_max_state_int
      {
        \tex_toks:D \l_regex_tmpb_int = \tex_toks:D \l_regex_tmpa_int
        \int_incr:N \l_regex_tmpa_int
        \int_incr:N \l_regex_tmpb_int
      }
  }
```

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

`\regex_build_group_n:` `\regex_build_group_n?:` These functions are called in case the syntax is `{`⟨*int*⟩`}`. Greedy and lazy operators are identical, since the number of repetitions is fixed. We only record the submatch information at the last repetition.

```
\cs_new_protected:Npn \regex_build_group_n: #1
  { % ^^A todo: catch case #1 <= 0.
    \regex_build_group_n_aux:n {#1}
    \regex_build_transition:NN
      \regex_action_free:n \l_regex_right_state_int
    \regex_build_group_submatches:NN
      \l_regex_tmpa_int \l_regex_left_state_int
  }
\cs_new_eq:cN { regex_build_group_n?: } \regex_build_group_n:
```

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

`\regex_build_group_n*:` `\regex_build_group_n*?:` These functions are called in case the syntax is `{`⟨*int*⟩`,}`. They are somewhat hybrid between the `{`⟨*int*⟩`}` and the `*` quantifiers. Contrarily to the `*` quantifier, for which we had to be careful not to overwrite the submatch information in case no iteration was made, here, we know that the submatch information is overwritten in any case.

```
\cs_new_protected:cpn { regex_build_group_n*: } #1
  { % ^^A todo: catch case #1 <= 0.
    \regex_build_group_n_aux:n {#1}
    \regex_build_transitions:NNNN
      \regex_action_free:n \l_regex_tmpa_int
      \regex_action_free:n \l_regex_right_state_int
    \regex_build_group_submatches:NN
      \l_regex_tmpa_int \l_regex_left_state_int
  }
\cs_new_protected:cpn { regex_build_group_n*?: } #1
  { % ^^A todo: catch case #1 <= 0.
    \regex_build_group_n_aux:n {#1}
    \regex_build_transitions:NNNN
```

```
      \regex_action_free:n \l_regex_right_state_int
      \regex_action_free:n \l_regex_tmpa_int
    \regex_build_group_submatches:NN
      \l_regex_tmpa_int \l_regex_left_state_int
  }
```

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

`\regex_build_group_nn:`
`\regex_build_group_nn?:`

These functions are called when the syntax is either {⟨*int*⟩,} or {⟨*int*⟩,⟨*int*⟩}.

```
\cs_new_protected:Npn \regex_build_group_nn: #1#2
  { % ^^A Not Implemented Yet!
    \msg_expandable_error:n { Quantifier~{m,n}~not~implemented~yet }
    \use:c { regex_build_group_n*: } {#1}
  }
\cs_new_protected:cpn { regex_build_group_nn?: } #1#2
  { % ^^A Not Implemented Yet!
    \msg_expandable_error:n { Quantifier~{m,n}~not~implemented~yet }
    \use:c { regex_build_group_n*?: } {#1}
  }
```

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

2.4 Matching

2.4.1 Use of TEX registers when matching

The first step in matching a regular expression is to build the corresponding NFA and store its states in the `\toks` registers. Then loop through the query token list one character (one "step") at a time, exploring in parallel every possible path through the NFA. We keep track of an array of the states currently "active". More precisely, `\skip` registers hold the state numbers to be considered when the next token is read.

At every step, we unpack that array of active states and empty it. Then loop over all active states, and perform the instruction at that state of the NFA. This can involve "free" transitions to other states, or transitions which "consume" the current character. For free transitions, the instruction at the new state of the NFA is performed. When a transition consumes a character, the new state is put in the array of `\skip` registers: it will be active again when the next character is read.

If two paths through the NFA "collide" in the sense that they reach the same state when reading a given character, then any future execution will be identical for both. Hence, it is indeed enough to keep track of which states are active. [In the presence of back-references, the future execution is affected by how the previous match took place; this is why we cannot support those non-regular features.]

Many of the functions require extracting the submatches for the "best" match. Execution paths through the NFA are ordered by precedence: for instance, the regular expression `a?` creates two paths, matching either an empty token list or a single `a`; the path matching an `a` has higher precedence. When two paths collide, the path with the highest precedence is kept, and the other one is discarded. The submatch information for a given path is stored at the start of the `\toks` register which holds the state at which that path currently is.

Deciding to store the submatch information in \toks registers alongside with states of the NFA unfortunately implies some shuffling around. The two other options are to store the submatch information in one control sequence per path, which wastes csnames, or to store all of the submatch information in one property list, which turns out to be too slow. A tricky aspect of submatch tracking is to know when to get rid of submatch information. This naturally happens when submatch information is stored in \toks registers: if the information is not moved, it will be overwritten later.

The presence of ϵ-transitions (transitions which consume no character) leads to potential infinite loops; for instance the regular expression (a??)* could lead to an infinite recursion, where a?? matches no character, * loops back to the start of the group, and a?? matches no character again. Therefore, we need to keep track of the states of the NFA visited at the current step. More precisely, a state is marked as "visited" if the instructions for that state have been inserted in the input stream, by setting the corresponding \dimen register to a value which uniquely identifies at which step it was last inserted.

The current approach means that stretch and shrink components of \skip registers, as well as all \muskip registers are unused. It could seem that \count registers are also free for use, but we still want to be able to safely use integers, which are implemented as \count registers.

2.4.2 Helpers for running the NFA

\regex_store_state:n

Put the given state in the array of \skip registers. This is done by increasing the pointer \l_regex_max_index_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
  {
    \int_incr:N \l_regex_max_index_int
    \tex_skip:D \l_regex_max_index_int #1 sp \scan_stop:
    \regex_store_submatches:n {#1}
  }
```

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

\regex_state_use:
\regex_state_use_with_submatches:
\regex_state_use_aux_ii:w
\regex_state_use_aux:n

Use a given program instruction, unless it has already been executed at this step. The \toks registers begin with some submatch information, ignored by \regex_state_use:, but not by \regex_state_use_with_submatches:. A state is free if it is not marker as taken, namely if the corresponding \dimen register is not \l_regex_unique_id_int in sp. The primitive conditional is ended before unpacking the \toks register.

```
\cs_new_protected_nopar:Npn \regex_state_use_with_submatches:
  { \regex_state_use_aux:n { } }
\cs_new_protected_nopar:Npn \regex_state_use:
  { \regex_state_use_aux:n { \exp_after:wN \use_none_delimit_by_s_stop:w } }
\cs_new_protected:Npn \regex_state_use_aux:n #1
  {
    \if_num:w \tex_dimen:D \l_regex_current_state_int
        < \l_regex_unique_id_int
      \tex_dimen:D \l_regex_current_state_int
```

```
        = \l_regex_unique_id_int sp \scan_stop:
      #1 \tex_the:D \tex_toks:D \exp_after:wN \l_regex_current_state_int
    \fi:
    \scan_stop:
  }
```

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

2.4.3 Submatch tracking when running the NFA

`\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_eq:NN \regex_state_use_with_submatches: \regex_state_use:
    \cs_set_eq:NN \regex_store_submatches:n
      \regex_protected_use_none:n
    \cs_set_eq:NN \regex_action_submatches:n
      \regex_protected_use_none:n
  }
\cs_new_protected:Npn \regex_protected_use_none:n #1 { }
```

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

`\regex_store_submatches:n`
`\regex_store_submatches_aux:w`
`\regex_store_submatches_aux_ii:Nnnw`

The submatch information pertaining to one given thread is moved from state to state as we execute the NFA. We make sure that most of the `\toks` register is not read before being assigned again to that same register.

```
\cs_new_protected:Npn \regex_store_submatches:n #1
  {
    \tex_toks:D #1 \exp_after:wN
      {
        \tex_romannumeral:D
        \exp_after:wN \regex_store_submatches_aux:w
        \tex_the:D \tex_toks:D #1
      }
  }
\cs_new_protected:Npn \regex_store_submatches_aux:w #1 \s_stop
  {
    \regex_store_submatches_aux_ii:Nnnw
      #1
      \regex_state_submatches:nn \c_minus_one \q_prop
    \s_stop
  }
\cs_new_protected:Npn \regex_store_submatches_aux_ii:Nnnw
    \regex_state_submatches:nn #1 #2 #3 \s_stop
  {
    \exp_after:wN \c_zero
    \exp_after:wN \regex_state_submatches:nn \exp_after:wN
      {
        \int_value:w \int_eval:w
```

```
          \l_regex_unique_id_int + \c_one
        \exp_after:wN
      }
      \exp_after:wN { \l_regex_current_submatches_prop }
    \regex_state_submatches:nn {#1} {#2}
    \s_stop
  }
```

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

`\regex_state_submatches:nn` This function is inserted by `\regex_store_submatches:n` in the `\toks` register holding a given state, and it is performed when the state is used.

```
\cs_new_protected:Npn \regex_state_submatches:nn #1#2
  {
    \if_num:w #1 = \l_regex_unique_id_int
      \tl_set:Nn \l_regex_current_submatches_prop { #2 }
    \fi:
  }
```

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

2.4.4 Matching: framework

`\regex_match:n` Then reset a few variables which should be set only once, before the first match, even in the case of multiple matches. Then run the NFA (`\regex_match_once:` matches multiple times when appropriate).

```
\cs_new_protected:Npn \regex_match:n #1
  {
    \tl_set_analysis:Nn \l_regex_tmpa_tl {#1}

    \int_zero:N \l_regex_nesting_int
    \int_set_eq:NN \l_regex_current_step_int \l_regex_max_state_int
    \regex_query_set:nnn { } { -1 } { -2 }
    \int_set_eq:NN \l_regex_min_step_int \l_regex_current_step_int
    \exp_after:wN \regex_query_set_loop:ww
      \l_regex_tmpa_tl
      \s_tl { ? \prg_map_break: } \s_tl
    \prg_break_point:n { }
    \int_set_eq:NN \l_regex_max_step_int \l_regex_current_step_int
    \regex_query_set:nnn { } { -1 } { -2 }

    \regex_match_initial_setup:
    \regex_match_once:
  }
\cs_new:Npn \regex_query_set_loop:ww #1 \s_tl #2#3 \s_tl
  {
    \use_none:n #2
    \regex_query_set:nnn {#1} {"#2} {#3}
    \if_case:w "#2 \exp_stop_f:
    \or: \int_incr:N \l_regex_nesting_int
    \or: \int_decr:N \l_regex_nesting_int
```

```
    \fi:
    \regex_query_set_loop:ww
  }
\cs_new_protected:Npn \regex_query_set:nnn #1#2#3
  {
    \tex_muskip:D \l_regex_current_step_int
      = \etex_gluetomu:D
        #3 sp
        plus #2 sp
        minus \l_regex_nesting_int sp
      \scan_stop:
    \tex_toks:D \l_regex_current_step_int {#1}
    \int_incr:N \l_regex_current_step_int
  }
\cs_new_protected_nopar:Npn \regex_query_get:
  {
    \tl_set:Nx \l_regex_current_token_tl
      { \tex_the:D \tex_toks:D \l_regex_current_step_int }
    \l_regex_current_char_int
      = \etex_mutoglue:D \tex_muskip:D \l_regex_current_step_int
    \l_regex_current_catcode_int = \etex_gluestretch:D
      \etex_mutoglue:D \tex_muskip:D \l_regex_current_step_int
  }
```

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

`\regex_match_once:` Set up more variables in `\regex_match_setup:`. If there was a match, use the token list `\l_regex_every_match_tl`, which may call `\regex_match_once:` again to achieve multiple matches.

```
\cs_new_protected_nopar:Npn \regex_match_once:
  {
    \regex_match_setup:
    \regex_query_get:
    \regex_match_loop:
    \prg_break_point:n { }
    \bool_if:NT \l_regex_success_match_bool
      {
        \bool_gset_true:N \g_regex_success_bool
        \l_regex_every_match_tl
      }
  }
```

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

`\regex_match_initial_setup:` This function holds the setup that should be done only once for one given pattern matching. It is called only once for the whole token list. On the other hand, `\regex_match_setup:` is called for every match in the token list in case of repeated matches.

```
\cs_new_protected_nopar:Npn \regex_match_initial_setup:
  {
    \prg_stepwise_inline:nnnn {0} {1} { \l_regex_max_state_int - \c_one }
      { \tex_dimen:D ##1 \c_minus_one sp \scan_stop: }
```

```
    \int_set_eq:NN \l_regex_unique_id_int  \c_minus_one
    \int_set:Nn \l_regex_start_step_int
      { \l_regex_min_step_int - \c_one }
    \int_set_eq:NN \l_regex_current_step_int \l_regex_min_step_int
    \int_set_eq:NN \l_regex_success_step_int \l_regex_min_step_int
    \int_set_eq:NN \l_regex_submatch_int \l_regex_max_state_int
    \bool_set_false:N \l_regex_success_empty_bool
    \bool_gset_false:N \g_regex_success_bool
  }
```

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

`\regex_match_setup:` Every time a match starts, `\regex_match_setup:` resets a few variables.

```
\cs_new_protected_nopar:Npn \regex_match_setup:
  {
    \prop_clear:N \l_regex_current_submatches_prop
    \bool_if:NTF \l_regex_success_empty_bool
      { \cs_set_eq:NN \regex_last_match_empty:F \regex_last_match_empty_yes:F }
      { \cs_set_eq:NN \regex_last_match_empty:F \regex_last_match_empty_no:F }
    \int_set_eq:NN \l_regex_start_step_int \l_regex_success_step_int
    \int_set:Nn \l_regex_current_step_int
      { \l_regex_start_step_int - \c_one }
    \bool_set_false:N \l_regex_success_match_bool
    \int_zero:N \l_regex_max_index_int
    \regex_store_state:n {0} %^^A _state_int!
  }
```

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

`\regex_match_loop:`
`\regex_match_one_index:n`
`\regex_match_one_index_aux:n`

Setup what needs to be reset at every character, then set `\l_regex_current_char_int` to the character code of the token that is read (and -1 for the end of the token list), and loop over the elements of the `\skip` array. Then repeat. There are a couple of tests to stop reading the token list when no active state is left, or when the end is reached. At every step in reading the token list, we store the character code of the current character in `\l_regex_current_char_int`, unless the end was reached: then we store -1.

```
\cs_new_protected_nopar:Npn \regex_match_loop:
  {
    \int_incr:N \l_regex_current_step_int
    \int_incr:N \l_regex_unique_id_int
    \bool_set_false:N \l_regex_fresh_thread_bool
    \int_set_eq:NN \l_regex_last_char_int \l_regex_current_char_int
    \regex_query_get:
    \regex_match_loop_case_hook:
    \cs_set_nopar:Npx \regex_tmp:w
      {
        \int_zero:N \l_regex_max_index_int
        \regex_match_one_index:w 1 ; \prg_break_point:n { }
        \exp_not:N \prg_break_point:n { }
      }
    \regex_tmp:w
    \if_num:w \l_regex_current_char_int < \c_minus_one
```

```
      \exp_after:wN \prg_map_break:
    \fi:
    \if_num:w \l_regex_max_index_int = \c_zero
      \exp_after:wN \prg_map_break:
    \fi:
    \regex_match_loop:
  }
\cs_new:Npn \regex_match_one_index:w #1;
  {
    \if_num:w #1 > \l_regex_max_index_int
      \exp_after:wN \prg_map_break:
    \fi:
    \regex_match_one_index_aux:n
      { \int_value:w \tex_skip:D #1 }
    \exp_after:wN \regex_match_one_index:w
    \int_use:N \int_eval:w #1 + \c_one ;
  }
\cs_new_protected:Npn \regex_match_one_index_aux:n #1
  {
    \int_set:Nn \l_regex_current_state_int {#1}
    \prop_clear:N \l_regex_current_submatches_prop
    \regex_state_use_with_submatches:
  }
```

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

`\regex_match_loop_case_hook:`
`\regex_match_loop_caseless_hook:`

In the case where the regular expression contains caseless matching, the `\regex_match_loop_case_hook:` (normally empty) is redefined to set `\l_regex_case_changed_char_int` properly.

```
\cs_new_protected_nopar:Npn \regex_match_loop_case_hook: { }
\cs_new_protected_nopar:Npn \regex_match_loop_caseless_hook:
  {
    \int_set_eq:NN \l_regex_case_changed_char_int \l_regex_current_char_int
    \if_num:w \l_regex_current_char_int < \c_ninety_one
      \if_num: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_num:w \l_regex_current_char_int < \c_one_hundred_twenty_three
        \if_num: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_match_loop_case_hook:`*. This function is documented on page* ??.)

2.4.5 Actions when matching

\regex_action_start_wildcard: : the first state has a free transition to the second state, where the regular expression really begins, and a costly transition to itself, to try again at the next character. The search is made unanchored at the start by putting a free transition to the real start of the NFA, and a costly transition to the same state, waiting for the next token in the query. This combination could be reused (with some changes). We sometimes need to know that the match for a given thread starts at this character. For that, we use the boolean \l_regex_fresh_thread_bool.

```
\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:. *This function is documented on page* ??.)

\regex_action_cost:n A transition which consumes the current character and moves to state #1.

```
\cs_new_protected:Npn \regex_action_cost:n #1
  {
    \exp_args:Nf \regex_store_state:n
      { \int_eval:n { \l_regex_current_state_int + #1 } }
  }
```

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

\regex_action_success: There is a successful match when an execution path reaches the end of the regular expression. Then store the current step and submatches. The current step is then interrupted with \prg_map_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_last_match_empty:F
      {
        \bool_set_true:N \l_regex_success_match_bool
        \bool_set_eq:NN \l_regex_success_empty_bool
          \l_regex_fresh_thread_bool
        \int_set_eq:NN \l_regex_success_step_int \l_regex_current_step_int
        \prop_set_eq:NN \l_regex_success_submatches_prop
          \l_regex_current_submatches_prop
        \prg_map_break:
      }
  }
```

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

\regex_action_free:n To copy a thread, check whether the program state has already been used at this character. 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.

```
\cs_new_protected:Npn \regex_action_free:n #1
  {
    \cs_set_nopar:Npx \regex_tmp:w
      {
        \int_add:Nn \l_regex_current_state_int {#1}
        \regex_state_use:
        \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 }
      }
    \regex_tmp:w
  }
```

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

`\regex_action_submatch:n` Update the current submatches with the information from the current step.

```
\cs_new_protected:Npn \regex_action_submatch:n #1
  {
    \prop_put:Nno \l_regex_current_submatches_prop {#1}
      { \int_use:N \l_regex_current_step_int }
  }
```

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

2.5 Replacement

`\regex_submatch_nesting_aux:n`

```
\cs_new_protected:Npn \regex_submatch_nesting_aux: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_nesting_aux:n`*. This function is documented on page* **??**.)

`\regex_replacement:n` Our goal here is to analyse the replacement text. First take care of detecting escaped and non-escaped characters using `\str_escape_use:NNNn` with three protected arguments. This inserts in the input stream a token list of the form $\langle fn\ 1\rangle\ \langle char\ 1\rangle\ \ldots\ \langle fn\ N\rangle\ \langle char\ N\rangle$, where $\langle fn\ i\rangle$ is one of the three functions, and $\langle char\ i\rangle$ a character in the string `#1`.

```
\cs_new:Npn \regex_nesting:n #1 { } %^^A move. Rename?
\tl_new:N \l_regex_nesting_tl
\cs_new_protected:Npn \regex_replacement:n #1
  {
    \int_zero:N \l_regex_replacement_int
    \int_zero:N \l_regex_nesting_int
    \tl_clear:N \l_regex_nesting_tl
```

```
    \str_escape_use:NNNn
      \regex_replacement_unescaped:N
      \regex_replacement_escaped:N
      \regex_replacement_raw:N
      {#1}
    \prg_do_nothing: \prg_do_nothing:
    \cs_set:Npx \regex_nesting:n ##1
      {
        + \int_use:N \l_regex_nesting_int
        \l_regex_nesting_tl
        - \regex_submatch_nesting_aux:n {##1}
      }
    \use:x
      {
        \exp_not:n { \cs_set:Npn \regex_replacement_tl:n ##1 }
          { \str_aux_toks_range:nn \c_zero \l_regex_replacement_int }
      }
      % ^^A rename! Careful with \cP\#.
  }
```

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

`\regex_replacement_raw:N`

```
\cs_new_protected:Npn \regex_replacement_raw:N #1
  { \regex_replacement_put:n {#1} }
```

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

`\regex_replacement_put:n` Raw characters are stored in a toks register.

```
\cs_new_protected:Npn \regex_replacement_put:n #1
  {
    \tex_toks:D \l_regex_replacement_int {#1}
    \int_incr:N \l_regex_replacement_int
  }
```

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

`\regex_replacement_escaped:N`
`\regex_replacement_submatch:w`
`\regex_replacement_submatch_aux:nN`

```
\cs_new_protected:Npn \regex_replacement_unescaped:N #1
  {
    \if_charcode:w \c_rbrace_str #1
      \if_num:w \l_regex_replacement_csnames_int > \c_zero
        \regex_replacement_put:n \cs_end:
      \else:
        \regex_replacement_put:n #1
      \fi:
    \else:
      \regex_replacement_put:n #1
    \fi:
  }
\cs_new_protected:Npn \regex_replacement_escaped:N #1
  {
```

```
    \if_charcode:w c #1
      \exp_after:wN \regex_replacement_c:w
    \else:
      \if_charcode:w g #1
        \exp_after:wN \exp_after:wN \exp_after:wN \regex_replacement_g:w
      \else:
        \if_num:w \c_one < 1#1 \exp_stop_f:
          \regex_replacement_put_submatch:n {#1}
        \else:
          \regex_replacement_put:n #1
        \fi:
      \fi:
    \fi:
  }
\cs_new_protected:Npn \regex_replacement_put_submatch:n #1
  {
    \regex_replacement_put:n
      { \regex_query_submatch:nn {#1} {##1} }
    \if_num:w \l_regex_replacement_csnames_int = \c_zero
      \tl_put_right:Nn \l_regex_nesting_tl
        {
          \exp_not:N \if_num:w #1 < \l_regex_capturing_group_int
            \regex_submatch_nesting_aux:n { \int_eval:w #1+##1 \int_eval_end: }
          \exp_not:N \fi:
        }
    \fi:
  }
\cs_new_protected:Npn \regex_replacement_error:NNN #1#2#3
  {
    \msg_kernel_error:nnxx { regex } { #1-command }
      { replacement~text } {#3}
    #2 #3
  }
\cs_new_protected:Npn \regex_replacement_g:w #1#2
  {
    \str_if_eq:xxTF
      { \exp_not:n { #1#2 } }
      { \regex_replacement_unescaped:N \c_lbrace_str }
      {
        \int_zero:N \l_regex_tmpa_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 \regex_replacement_unescaped:N
      {
        \if_num:w \c_one < 1#2 \exp_stop_f:
          \int_set:Nn \l_regex_tmpa_int { \c_ten * \l_regex_tmpa_int + #2 }
```

```
          \exp_after:wN \use_i:nnn
          \exp_after:wN \regex_replacement_g_digits:NN
        \else:
          \if_charcode:w \c_rbrace_str #2
            \exp_args:No \regex_replacement_put_submatch:n
              { \int_use:N \l_regex_tmpa_int }
            \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
          \else:
            \exp_after:wN \exp_after:wN
            \exp_after:wN \regex_replacement_error:NNN
            \exp_after:wN \exp_after:wN \exp_after:wN g
          \fi:
        \fi:
      }
      { \regex_replacement_error:NNN g }
    #1 #2
  }
\cs_new_protected:Npn \regex_replacement_c:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \regex_replacement_unescaped: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_lbrace_str :w }
  {
    \int_incr:N \l_regex_replacement_csnames_int
    \regex_replacement_put:n
      { \exp_not:n { \exp_after:wN \regex_replacement_exp_not:N \cs:w } }
  }
\cs_new_eq:cc { regex_replacement_c_C:w }
  { regex_replacement_c_ \c_lbrace_str :w }
\cs_new:Npn \regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
\group_begin:

  \char_set_catcode_math_superscript:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_U:w
    { \regex_replacement_char:nNN { ^^@ } }

  \char_set_catcode_alignment:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_T:w
    { \regex_replacement_char:nNN { ^^@ } }

  \cs_new_protected:Npn \regex_replacement_c_S:w #1#2
    {
      \int_compare:nNnTF { `#2 } = \c_zero
        { \regex_replacement_error:NNN c #1#2 }
        {
```

```
          \char_set_lccode:nn {32} { `#2 }
          \tl_to_lowercase:n { \regex_replacement_put:n {~} }
        }
    }

  \char_set_catcode_parameter:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_P:w
    { \regex_replacement_char:nNN { ^^@^^@^^@^^@^^@^^@^^@^^@ } }

  \char_set_catcode_other:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_O:w
    { \regex_replacement_char:nNN { ^^@ } }

  \char_set_catcode_math_toggle:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_M:w
    { \regex_replacement_char:nNN { ^^@ } }

  \char_set_catcode_letter:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_L:w
    { \regex_replacement_char:nNN { ^^@ } }

  \char_set_catcode_group_end:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_E:w
    {
      \int_decr:N \l_regex_nesting_int
      \regex_replacement_char:nNN { \if_false: { \fi:  ^^@ }
    }

  \char_set_catcode_math_subscript:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_D:w
    { \regex_replacement_char:nNN { ^^@ } }

  \char_set_catcode_group_begin:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_B:w
    {
      \int_incr:N \l_regex_nesting_int
      \regex_replacement_char:nNN { \exp_after:wN ^^@ \if_false: } \fi: }
    }

  \char_set_catcode_active:N \^^@
  \cs_new_protected_nopar:Npn \regex_replacement_c_A:w
    { \regex_replacement_char:nNN { \exp_not:N ^^@ } }

\group_end:

\cs_new_protected:Npn \regex_replacement_char:nNN #1#2#3
  {
    \char_set_lccode:nn \c_zero { `#3 }
    \tl_to_lowercase:n
      { \regex_replacement_put:n { \exp_not:n {#1} } }
```

```
}
```

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

2.6 User commands

2.6.1 Precompiled pattern

A given pattern is often reused to match many different query token lists. We thus give a means of storing the NFA corresponding to a given pattern in a token list variable of the form

```
\regex_nfa:Nw ⟨variable name⟩
⟨assignments⟩
\tex_toks:D 0 { ⟨instruction0⟩ }
...
\tex_toks:D n { ⟨instruction_n⟩ }
\s_stop
```

where n is the number of states in the NFA, and the various $\langle instruction_i \rangle$ control how the NFA behaves in state i. The `\regex_nfa:Nw` function removes the whole NFA from the input stream and produces an error: the ⟨*nfa var*⟩ should only be accessed through dedicated functions. This rather drastic approach is taken because assignments triggered by the contents of ⟨*nfa var*⟩ may overwrite data which is used elsewhere, unless everything is done carefully in a group.

`\regex_gset:Nn` `\regex_const:Nn` `\regex_set:Nn` `\regex_set_aux:NNn`

The three user functions only differ with which function is used to assign the pre-compiled regular expression to the user's variable. Internally, they all first build the NFA corresponding to the regex, then store the contents of all the necessary `\toks` registers in the user's variable.

```
\cs_new_protected_nopar:Npn \regex_set:Nn
  { \regex_set_aux:NNn \tl_set:Nn }
\cs_new_protected_nopar:Npn \regex_gset:Nn
  { \regex_set_aux:NNn \tl_gset:Nn }
\cs_new_protected_nopar:Npn \regex_const:Nn
  { \regex_set_aux:NNn \tl_const:Nn }
\cs_new_protected:Npn \regex_set_aux:NNn #1#2#3
  {
    \group_begin:
      \regex_build:n {#3}
      \cs_set_nopar:Npx \regex_tmp:w
        { #1 \exp_not:N #2 { \regex_set_aux:N #2 } }
      \exp_after:wN
    \group_end:
    \regex_tmp:w
  }
```

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

\regex_set_aux:N
\regex_set_aux:n
\regex_nfa:Nw

Within a group, build the NFA corresponding to the given regular expression, with submatch tracking. Then save the contents of all relevant \toks registers into the variable outside the group. The auxiliary \regex_nfa:Nw is not protected: this ensures that the NFA will properly be replaced by an error message in expansion contexts.

```
\cs_new:Npn \regex_set_aux:N #1
  {
    \exp_not:n { \regex_nfa:Nw #1 }
    \l_regex_max_state_int
      = \int_use:N \l_regex_max_state_int
    \l_regex_capturing_group_int
      = \int_use:N \l_regex_capturing_group_int
    \token_if_eq_meaning:NNT
      \regex_match_loop_case_hook:
      \regex_match_loop_caseless_hook:
      {
        \cs_set_eq:NN \regex_match_loop_case_hook:
          \regex_match_loop_caseless_hook:
      }
    \prg_stepwise_function:nnnN
      {0} {1} {\l_regex_max_state_int - \c_one }
      \regex_set_aux:n
    \s_stop
  }
\cs_new:Npn \regex_set_aux:n #1
  { \tex_toks:D #1 { \tex_the:D \tex_toks:D #1 } }
\cs_new:Npn \regex_nfa:Nw #1
  {
    \msg_expandable_kernel_error:nnn { regex } { nfa-misused } {#1}
    \use_none_delimit_by_s_stop:w
  }
```

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

\regex_use:N

No error-checking.

```
\cs_new_protected_nopar:Npn \regex_use:N
  { \exp_last_unbraced:No \use_none:nn }
```

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

\regex_to_str:N
\regex_to_str_aux:w

Of course, we could simply set \regex_to_str:N equal to \tl_to_str:N. After all, regex variables are in particular token list variables. We do some more processing to start each line with \tex_toks:D instead of a single very long line.

```
\cs_new:Npn \regex_to_str:N #1
  {
    \exp_after:wN \regex_to_str_aux:Nw #1
    \tex_toks:D \q_stop \prg_map_break: \tex_toks:D
    \prg_break_point:n { }
  }
\cs_new:Npn \regex_to_str_aux:Nw #1#2\tex_toks:D
  {
```

```
    \use_none_delimit_by_q_stop:w #2 \q_stop
    \tl_to_str:n {#1#2} \iow_newline:
    \regex_to_str_aux:Nw \tex_toks:D
  }
```

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

2.6.2 Generic auxiliary functions

Most of l3regex's work is done within a group.

`\regex_aux_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not.

```
\cs_new_protected_nopar:Npn \regex_aux_return:
  {
    \if_meaning:w \c_true_bool \g_regex_success_bool
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

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

`\regex_aux_build_match:nn` This auxiliary is used by user functions whose ⟨*regex*⟩ argument is given as an explicit regular expression within braces. In that case, we need to build the automaton corresponding to that regular expression, then perform the matching on the given token list #2.

```
\cs_new_protected:Npn \regex_aux_build_match:nn #1#2
  {
    \regex_build:n {#1}
    \regex_match:n {#2}
  }
```

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

`\regex_aux_use_match:Nn` This auxiliary is used by user functions whose ⟨*regex*⟩ argument is given as a pre-compiled regex variable. We make sure that the token list variable indeed is an automaton (by testing the first token). If not, the match is deemed unsuccessful, after raising an error. If we have an automaton, "use" it, then perform the matching on the given token list #2.

```
\cs_new_protected:Npn \regex_aux_use_match:Nn #1#2
  {
    \exp_args:No \tl_if_head_eq_meaning:nNTF {#1} \regex_nfa:Nw
      {
        \regex_use:N #1
        \regex_match:n {#2}
      }
      {
        \msg_kernel_error:nnx { regex } { not-nfa } { \token_to_str:N #1 }
        \bool_gset_false:N \g_regex_success_bool
      }
  }
```

(*End definition for* `\regex_aux_use_match:Nn`*. This function is 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: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`

We define here 40 user functions, following a common pattern in terms of an auxiliary such as `\regex_extract_once_aux:NnnN` (those auxiliaries are defined in the coming sections). The arguments handed to the auxiliary are `\regex_aux_build_match:nn` or `\regex_aux_use_match:Nn`, followed by the three arguments of the user function. The conditionals call `\regex_aux_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_nopar:Npn #1
      { #3 \regex_aux_build_match:nn }
    \cs_new_protected_nopar:Npn #2
      { #3 \regex_aux_use_match:Nn }
    \prg_new_protected_conditional:Npnn #1 ##1##2##3 { T , F , TF }
      {
        #3 \regex_aux_build_match:nn {##1} {##2} ##3
        \regex_aux_return:
      }
    \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
      {
        #3 \regex_aux_use_match:Nn {##1} {##2} ##3
        \regex_aux_return:
      }
  }
\tl_map_inline:nn
  {
    { extract_once } { extract_all }
    { replace_once } { replace_all }
    { split }
  }
  {
    \exp_args:Nccc \regex_tmp:w
      { regex_#1:nnN } { regex_#1:NnN } { regex_#1_aux:NnnN }
  }
```

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

2.6.3 Submatches, once the correct match is found

`\regex_extract:`
`\regex_extract_aux_b:wn`
`\regex_extract_aux_e:wn`

```
\cs_new_protected_nopar:Npn \regex_extract:
  {
    \int_set_eq:NN \l_regex_submatch_start_int \l_regex_submatch_int
    \if_meaning:w \c_true_bool \g_regex_success_bool
      \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_num:w ##1 \c_max_int
            \exp_after:wN \regex_extract_aux_b:wn \int_use:N
          \else:
            \exp_after:wN \regex_extract_aux_e:wn \int_use:N
          \fi:
          \int_eval:w \l_regex_submatch_start_int + ##1 {##2}
        }
    \fi:
  }
\cs_new_protected:Npn \regex_extract_aux_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_aux_e:wn #1 > #2
  {
    \tex_skip:D #1
      = 1 \tex_skip:D #1 plus #2 sp \scan_stop:
  }
```

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

`\regex_group_end_extract_seq:N`
`\regex_extract_seq_aux:n`
`\regex_extract_seq_aux:ww`

```
\cs_new_protected:Npn \regex_group_end_extract_seq:N #1
  {
      \cs_set_eq:NN \seq_item:n \scan_stop:
      \flag_lower:N \l_regex_group_begin_flag
      \flag_lower:N \l_regex_group_end_flag
      \tl_set:Nx \l_regex_tmpa_tl
        {
          \prg_stepwise_function:nnnN
            \l_regex_max_state_int
            \c_one
            { \l_regex_submatch_int - \c_one }
            \regex_extract_seq_aux:n
        }
      \flag_test:NF \l_regex_group_begin_flag
        { \flag_test:NF \l_regex_group_end_flag { \use_none:nnn } }
      \msg_kernel_error:nn { regex } { sequence-unbalanced }
      \tl_set:Nx \l_regex_tmpa_tl { \l_regex_tmpa_tl }
    \exp_args:NNNo \group_end:
    \tl_set:Nn #1 \l_regex_tmpa_tl
  }
\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_nesting_aux:n {#1} ; #1;
      }
  }
\cs_new:Npn \regex_extract_seq_aux:ww #1; #2;
  { %^^A todo: use flags
    \if_num:w #1 < \c_zero
      \flag_raise:N \l_regex_group_end_flag
      \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
    \fi:
    \regex_query_submatch:w #2;
    \if_num:w #1 > \c_zero
      \flag_raise:N \l_regex_group_begin_flag
      \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
    \fi:
  }
```

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

2.6.4 Matching

`\regex_match:nn` `\regex_match:Nn` `\regex_match_aux:n`

We don't track submatches. Then either build the NFA corresponding to the regular expression, or use a precompiled pattern. Then match, using the internal `\regex_match:n`. Finally return the result after closing the group.

```
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
  {
    \regex_match_aux:n
      { \regex_aux_build_match:nn {#1} {#2} }
  }
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \regex_match_aux:n
      { \regex_aux_use_match:Nn #1 {#2} }
  }
\cs_new_protected:Npn \regex_match_aux:n #1
  {
    \group_begin:
      \tl_clear:N \l_regex_every_match_tl
      \regex_disable_submatches:
      #1
    \group_end:
    \regex_aux_return:
  }
```

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

`\regex_count:nnN` `\regex_count:NnN` `\regex_count_aux:n`

Instead of aborting once the first "longest match" is found, we repeat the search. The code is such that the search will not start on the same character, hence avoiding infinite loops.

```
\cs_new_protected_nopar:Npn \regex_count:nnN
  { \regex_count_aux:NnnN \regex_aux_build_match:nn }
\cs_new_protected_nopar:Npn \regex_count:NnN
```

```
  { \regex_count_aux:NnnN \regex_aux_use_match:Nn }
\cs_new_protected:Npn \regex_count_aux:NnnN #1#2#3#4
  {
    \group_begin:
      \regex_disable_submatches:
      \int_zero:N \l_regex_match_count_int
      \tl_set:Nn \l_regex_every_match_tl
        {
          \int_incr:N \l_regex_match_count_int
          \regex_match_once:
        }
      #1 {#2} {#3}
      \exp_args:NNNo
    \group_end:
    \int_set:Nn #4 { \int_use:N \l_regex_match_count_int }
  }
```

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

2.6.5 Submatch extraction

`\regex_extract_once_aux:NnnN`

As announced, here comes the auxiliary for extracting one match. Since we only want one match, `\l_regex_every_match_tl` is empty, and does not trigger the matching code again. After matching, `\regex_extract:` extracts submatches into various `\skip` registers, and those are then concatenated into a sequence by `\regex_group_end_extract_seq:N`. That function is also responsible for closing the group.

```
\cs_new_protected:Npn \regex_extract_once_aux:NnnN #1#2#3#4
  {
    \group_begin:
      \tl_set:Nn \l_regex_every_match_tl { \regex_extract: }
      #1 {#2} {#3}
      \regex_group_end_extract_seq:N #4
  }
```

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

`\regex_extract_all_aux:NnnN`

The set of submatches will be built progressively in `\l_regex_result_seq`. For each match, extract the submatches, and concatenate that to the right of the result sequence, then start matching again. Finally, copy the result in the user's sequence variable.

```
\cs_new_protected:Npn \regex_extract_all_aux:NnnN #1#2#3#4
  {
    \group_begin:
      \tl_set:Nn \l_regex_every_match_tl
        { \regex_extract: \regex_match_once: }
      #1 {#2} {#3}
      \regex_group_end_extract_seq:N #4
  }
```

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

2.6.6 Splitting a token list by matches of a regex

\regex_split_aux:NnnN

Recurse through the matches, and for each, do the following. Extract the submatches into various \skip registers, then replace the match \0, which should not be kept in the final result, and replace it by the part of the token list before the match. This process must be inhibited to avoid creating empty items if the regex matched an empty token list at the place where the match attempt started. After the last successful match, we need to add to the result the part of the token list after the last match, unless the last match was empty and at the very end. Finally, \regex_group_end_extract_seq:N builds a sequence from all the \skip registers, and assigns it to #4 after closing the group.

```
\cs_new_protected:Npn \regex_split_aux:NnnN #1#2#3#4
  {
    \group_begin:
      \tl_set:Nn \l_regex_every_match_tl
        {
          \if_num:w \l_regex_start_step_int < \l_regex_success_step_int
            \regex_extract:
            \tex_skip:D \l_regex_submatch_start_int
              = \l_regex_start_step_int sp
                plus \tex_skip:D \l_regex_submatch_start_int \scan_stop:
          \fi:
          \regex_match_once:
        }
      #1 {#2} {#3}
      \tex_skip:D \l_regex_submatch_int
        = \l_regex_start_step_int sp
          plus \l_regex_current_step_int sp \scan_stop:
      \int_incr:N \l_regex_submatch_int
      \if_num:w \l_regex_start_step_int = \l_regex_current_step_int
        \if_meaning:w \c_true_bool \l_regex_success_empty_bool
          \int_decr:N \l_regex_submatch_int
        \fi:
      \fi:
      \regex_group_end_extract_seq:N #4
  }
```

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

2.6.7 Replacement

\regex_replace_once_aux:NnnN

The replacement text is analysed by \regex_replacement:n, which defines \regex_replacement_tl:n to expand to the replaced token list, assuming that submatches are stored in various \skip registers, as done by \regex_extract:. If there is a match, we grab the parts before and after it, and get the result by x-expanding twice.

```
\cs_new_protected:Npn \regex_replace_once_aux:NnnN #1#2#3#4
  {
    \group_begin:
      \tl_clear:N \l_regex_every_match_tl
      \regex_replacement:n {#3}
```

```
      \exp_args:Nno #1 {#2} #4
      \if_meaning:w \c_true_bool \g_regex_success_bool
        \regex_extract:
        \int_set:Nn \l_regex_tmpa_int
          { \regex_nesting:n { \l_regex_submatch_start_int } }
        \if_num:w \l_regex_tmpa_int = \c_zero
        \else:
          \msg_kernel_error:nnx { regex } { replace-unbalanced }
            { \l_regex_tmpa_int }
        \fi:
        \tl_set:Nx \l_regex_tmpa_tl
          {
            \if_num:w \l_regex_tmpa_int < \c_zero
              \prg_replicate:nn { - \l_regex_tmpa_int }
                { \exp_not:n { { \if_false: } \fi: } }
            \fi:
            \regex_query_substr:nn
              { \l_regex_min_step_int }
              { \tex_skip:D \l_regex_submatch_start_int }
            \regex_replacement_tl:n { \l_regex_submatch_start_int }
            \regex_query_substr:nn
              { \etex_gluestretch:D \tex_skip:D \l_regex_submatch_start_int }
              { \l_regex_max_step_int }
            \if_num:w \l_regex_tmpa_int > \c_zero
              \prg_replicate:nn { \l_regex_tmpa_int }
                { \exp_not:n { \if_false: { \fi: } } }
            \fi:
          }
        \tl_set:Nx \l_regex_tmpa_tl { \l_regex_tmpa_tl }
        \exp_args:NNNo \group_end:
        \tl_set:Nn #4 \l_regex_tmpa_tl
      \else:
        \group_end:
      \fi:
  }
\cs_new_protected:Npn \regex_replace_once_aux:w #1;
  {
  }
```

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

`\regex_replace_all_aux:NnnN` For every match, extract submatches, and add the part before the beginning of the match, as well as the replacement, to the result. After the last match, extract the end of the token list, and add it to the replaced token list.

```
\cs_new_protected:Npn \regex_replace_all_aux:NnnN #1#2#3#4
  {
    \group_begin:
      \tl_set:Nn \l_regex_every_match_tl
        {
          \regex_extract:
```

```
          \tex_skip:D \l_regex_submatch_start_int
            = \tex_the:D \tex_skip:D \l_regex_submatch_start_int
              minus \l_regex_start_step_int sp \scan_stop:
          \regex_match_once:
        }
      \regex_replacement:n {#3}
      \exp_args:Nno #1 {#2} #4
      \int_set:Nn \l_regex_tmpa_int
        {
          0
          \prg_stepwise_function:nnnN
            \l_regex_max_state_int
            \l_regex_capturing_group_int
            { \l_regex_submatch_int - \c_one }
            \regex_nesting:n
        }
      \if_num:w \l_regex_tmpa_int = \c_zero
      \else:
        \msg_kernel_error:nnx { regex } { replace-unbalanced }
          { \l_regex_tmpa_int }
      \fi:
      \tl_set:Nx \l_regex_tmpa_tl
        {
          \if_num:w \l_regex_tmpa_int < \c_zero
            \prg_replicate:nn { - \l_regex_tmpa_int }
              { \exp_not:n { { \if_false: } \fi: } }
          \fi:
          \prg_stepwise_function:nnnN
            \l_regex_max_state_int
            \l_regex_capturing_group_int
            { \l_regex_submatch_int - \c_one }
            \regex_replace_all_aux:n
          \regex_query_substr:nn
            \l_regex_start_step_int \l_regex_max_step_int
          \if_num:w \l_regex_tmpa_int > \c_zero
            \prg_replicate:nn { \l_regex_tmpa_int }
              { \exp_not:n { \if_false: { \fi: } } }
          \fi:
        }
      \tl_set:Nx \l_regex_tmpa_tl { \l_regex_tmpa_tl }
    \exp_args:NNNo \group_end:
    \tl_set:Nn #4 \l_regex_tmpa_tl
  }
\cs_new:Npn \regex_replace_all_aux:n #1
  {
    \regex_query_substr:nn
      { \etex_glueshrink:D \tex_skip:D #1 } { \tex_skip:D #1 }
    \regex_replacement_tl:n {#1}
  }
```

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

2.7 Messages

```
\msg_kernel_new:nnnn { regex } { sequence-unbalanced }
  {
    Missing~
    \flag_test:NTF \l_regex_group_end_flag
      {
        left~
        \flag_test:NTF \l_regex_group_begin_flag
          { and~right~braces } { brace }
      }
      { right~brace }
    \ inserted~in~extracted~match.
  }
  {
    LaTeX~was~asked~to~extract~submatches~or~split~a~token~list~
    according~to~a~given~regular~expression,~but~some~of~the~resulting~
    items~were~not~balanced.
  }
\msg_kernel_new:nnnn { regex } { replace-unbalanced }
  { The~result~of~a~replacement~does~not~have~balanced~braces. }
  {
    LaTeX~was~asked~to~do~some~regular~expression~replacement,~
    and~the~resulting~token~list~would~not~have~the~same~number~
    of~begin-group~and~end-group~tokens. \\ \\
    \ \ \ \
    \prg_case_int:nnn {#1}
      {
        { -1 } { A~left~brace~was }
        {  1 } { A~right~brace~was }
      }
      {
        \int_abs:n {#1} ~
        \int_compare:nNnTF {#1} < \c_zero { left } { right } ~
        braces ~ were
      }
    \ inserted.
  }
\msg_kernel_new:nnnn { regex } { missing-rparen }
  { Missing~right~parenthesis~added~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.
```

```
  }
\msg_kernel_new:nnnn { regex } { backwards-range }
  { 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}.
  }
\msg_kernel_new:nnnn { regex } { unsupported }
  { Unsupported~construction~‘#1’. }
  {
    The~construction~‘#1’~is~not~supported~by~the~LaTeX~
    regular~expression~module.~Perhaps~some~character~should~
    have~been~escaped?
  }
\msg_kernel_new:nnnn { regex } { not-nfa }
  { This~is~not~a~regular~expression~variable. }
  {
    LaTeX~was~expecting~‘#1’~to~be~a~regular~expression~variable.\\
    This~control~sequence~is~not~a~regex~variable.~It’s~current~meaning~
    is~\\\\
    \ \ \ \ \token_to_str:N #1 = \token_to_meaning:N #1 .
  }
% \msg_new:nnn { regex } { 1 } { \iow_char:N\\~at~end~of~pattern }
% \msg_new:nnn { regex } { 2 } { \iow_char:N\\c~at~end~of~pattern }
% \msg_new:nnn { regex } { 4 }
%   { Numbers~out~of~order~in~\iow_char:N\{\iow_char\}~quantifier. }
% \msg_new:nnn { regex } { 6 }
%   { Missing~terminating~\iow_char:N\]~for~character~class }
% \msg_new:nnn { regex } { 7 }
%   { Invalid~escape~sequence~in~character~class }
% \msg_new:nnn { regex } { 34 }
%   { Character~value~in~\iow_char:N\\x{...}~sequence~is~too~large }
% \msg_new:nnn { regex } { 44 } { Invalid~UTF-8~string }
% \msg_new:nnn { regex } { 46 }
%   { Malformed~\iow_char:N\\P~or\iow_char:N\\p~sequence }
% \msg_new:nnn { regex } { 47 }
%   { Unknown~property~after~\iow_char:N\\P~or\iow_char:N\\p }
% \msg_new:nnn { regex } { 68 }
%   { \iow_char:N\\c~must~be~followed~by~an~ASCII~character }
\msg_kernel_new:nnnn { regex } { c-command }
  { Misused~\iow_char:N\\c~or\iow_char:N\\C~command~in~a~#1. }
  {
    In~a~#1,~the~\iow_char:N\\C~escape~sequence~
    can~be~followed~by~one~of~the~letters~ABCDELMOPSTU~
    or~a~brace~group,~not~by~’#2’.
  }
\msg_kernel_new:nnnn { regex } { unknown-option }
```

```
  { Unknown~option~‘#1’~for~regular~expressions. }
  {
    LaTeX~came~across~something~like~‘(?#1)’~in~a~regular~expression,~
    but~the~option~‘#1’~is~not~known.~It~will~be~ignored.
  }
\msg_kernel_new:nnnn { regex } { invalid-in-option }
  { Invalid~character~in~option~of~a~regular~expression. }
  {
    The~character~or~escape~sequence~‘#1’~is~not~defined~
    as~an~option~within~regular~expressions.
  }
\msg_kernel_new:nnnn { regex } { g-command }
  { 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:nnn { regex } { nfa-misused }
  { Automaton~#1 used~incorrectly. }
⟨/package⟩
```