

The `l3regex` package: regular expressions in $\text{T}_{\text{E}}\text{X}^*$

The \LaTeX 3 Project[†]

Released 2012/02/07

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 $\text{T}_{\text{E}}\text{X}$ manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

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

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

```
\regex_replace_all:nnN { \w+ } { \0 , } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word).

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

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

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

*This file describes v3330, last revised 2012/02/07.

[†]E-mail: latex-team@latex-project.org

1.1 Syntax of regular expressions

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

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

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

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

Characters.

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

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

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

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

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

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

`\t` Horizontal tab (hex 09).

Character types.

`.` A single period matches any token.

`\d` Any decimal digit.

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

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

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

`\w` Any word character, *i.e.*, alpha-numerics and underscore, equivalent to `[A-Za-z0-9_]`.

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

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

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

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

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

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

`\N` Any token not matched by `\n`.

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 *cname* 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.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{<tl var name>}` matches the exact contents of the token list *<tl var>*. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

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.

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 in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a precompiled regular expression.

`\regex_new:N` `\regex_new:N <regex var>`

Creates a new `<regex var>` or raises an error if the name is already taken. The declaration is global. The `<regex var>` will initially be such that it never matches.

`\regex_set:Nn` `\regex_set:Nn <regex var> {<regex>}`
`\regex_gset:Nn`
`\regex_const:Nn`

Stores a precompiled version of the `<regular expression>` in the `<regex var>`. For instance, this function can be used as

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

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

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`.

<code>\regex_match:nnTF</code>	<code>\regex_match:nnTF {<regex>} {<token list>} {<true code>} {<false code>}</code>
<code>\regex_match:NnTF</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.

<code>\regex_count:nnN</code>	<code>\regex_count:nnN {<regex>} {<token list>} <int var></code>
<code>\regex_count:NnN</code>	

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

<code>\regex_extract_once:nnNTF</code>	<code>\regex_extract_once:nnN {<regex>} {<token list>} <seq var></code>
<code>\regex_extract_once:NnNTF</code>	<code>\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</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.

<code>\regex_extract_all:nnNTF</code> <code>\regex_extract_all:NnNTF</code>	<code>\regex_extract_all:nnN {<regex>} {<token list>} <seq var></code> <code>\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code>
--	---

Finds all matches of the *<regular expression>* in the *<token list>*, and stores all the sub-match 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.

<code>\regex_split:nnNTF</code> <code>\regex_split:NnNTF</code>	<code>\regex_split:nnN {<regular expression>} {<token list>} <seq var></code> <code>\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<true code>} {<false code>}</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

<code>\regex_replace_once:nnNTF</code> <code>\regex_replace_once:NnNTF</code>	<code>\regex_replace_once:nnN {<regular expression>} {<replacement>} <tl var></code> <code>\regex_replace_once:nnNTF {<regular expression>} {<replacement>} <tl var> {<true code>} {<false code>}</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.*

<code>\regex_replace_all:nnNTF</code>	<code>\regex_replace_all:nnN {<regular expression>} {<replacement>} <tl var></code>
<code>\regex_replace_all:NnNTF</code>	<code>\regex_replace_all:nnNTF {<regular expression>} {<replacement>} <tl var> {<true code>} {<false code>}</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.

- Clean up the use of messages.
- Code comments.
- Rewrite the documentation in a clearer way.
- 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.
- 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`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize the use of `\prg_stepwise_...` functions.
- Decide what `\c{\c{...}}` should do in the replacement text.
- Fix the `\c{\cBx}` bug in replacement text.

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 pdf \TeX .
- 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: Xe \TeX and Lua \TeX serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

2 l3regex implementation

```
<*package>
1 \ProvidesExplPackage
2   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
3 \RequirePackage{l3str, l3tl-analysis, l3flag}
```

2.1 Plan of attack

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

Given a regular expression of n characters, we build a non-deterministic finite automaton (NFA) with roughly n states, which accepts precisely those token lists matching that regular expression. Then loop through the query token list one token (one “index”) at a time, exploring in parallel every possible path through the NFA. When performing this matching, we keep track of an array of the states currently “active”, which are to be considered in order when the next token is read.

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

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Index*: each token in the query is labelled by an integer $\langle index \rangle$, with $\backslash\text{_l_regex_min_index_int} - 1 \leq \langle index \rangle \leq \backslash\text{_l_regex_max_index_int}$. The lowest and highest indexes correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer $\langle state \rangle$ with $0 \leq \langle state \rangle < \backslash\text{_l_regex_max_state_int}$.
- *Active state*: state of the NFA that is reached when reading the query string for the matching. Those states are ordered according to the greediness of quantifiers.
- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches.

To achieve a good performance, we abuse \TeX ’s registers in two ways. We access registers directly by number rather than tying them to control sequence using $\backslash\text{_int_new:N}$ and other allocation functions. And we store integers in $\backslash\text{_dimen}$ registers in scaled points (_sp), using \TeX ’s implicit conversion from dimensions to integers in some contexts. Specifically, the registers are used as follows. When building,

- $\backslash\text{_toks}\langle state \rangle$ has two parts separated by $\backslash\text{_s_stop}$: a property list which holds the submatch information, followed by the tests and actions to perform in the $\langle state \rangle$ of the NFA.

- `\skipi` has the form $\langle group \rangle$ plus $\langle left\ state \rangle$ minus $\langle right\ state \rangle$.

When matching,

- `\dimen` $\langle state \rangle$ is equal to the last $\langle step \rangle$ in which the $\langle state \rangle$ was active.
- `\skipi` holds the $\langle state \rangle$ number corresponding to the i -th active $\langle state \rangle$ (smaller i is higher precedence) and has no shrink or stretch components.
- `\toks` $\langle index \rangle$ holds $\langle tokens \rangle$ which `o`- and `x`-expand to the $\langle index \rangle$ -th token in the query.

`\count` registers are not abused, which means that we can safely use named integers in this module.

The code is structured as follows. Variables, constants and various helper functions are introduced first, to limit the clutter in later parts. Then functions pertaining to parsing the regular expression are defined: that part is rather long because of the many bells and whistles of the regex notation. The next subsection takes care of running the NFA. Thereafter, we give tools for the replacements. Finally, user functions.

2.2 Constants and variables

```

\regex_tmp:w Temporary variables used for various purposes.
\l_regex_internal_a_int 4 \cs_new:Npn \regex_tmp:w { }
\l_regex_internal_b_int 5 \tl_new:N \l_regex_internal_a_tl
\l_regex_internal_c_int 6 \tl_new:N \l_regex_internal_b_tl
\l_regex_internal_a_tl 7 \int_new:N \l_regex_internal_a_int
\l_regex_internal_b_tl 8 \int_new:N \l_regex_internal_b_int
\l_regex_internal_c_int 9 \int_new:N \l_regex_internal_c_int
\g_regex_internal_tl 10 \tl_new:N \g_regex_internal_tl

```

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

2.2.1 Character properties

```

\c_regex_d_tl These constant token lists encode which characters are recognized by \d, \D, \w, etc. in
\c_regex_D_tl regular expressions. Namely, \d=[0-9], \w=[0-9A-Z_a-z], \s=[\_\^\^I\^\^J\^\^L\^\^M],
\c_regex_h_tl \h=[\_\^\^I], \v=[\^\^J-\^\^M], and the upper case counterparts match anything that
\c_regex_H_tl the lower case does not match. The order in which the various tests appear is optimized
\c_regex_s_tl for usual mostly lower case letter text.
\c_regex_S_tl 11 \tl_const:Nn \c_regex_d_tl
\c_regex_v_tl 12 { \regex_item_range:nn \c_forty_eight { 57 } } % 0--9
\c_regex_V_tl 13 \tl_const:Nn \c_regex_h_tl
\c_regex_w_tl 14 {
\c_regex_W_tl 15 \regex_item_equal:n \c_thirty_two % space
\c_regex_N_tl 16 \regex_item_equal:n \c_nine % tab
17 }
18 \tl_const:Nn \c_regex_s_tl
19 {
20 \regex_item_equal:n \c_thirty_two % space

```

```

21 \regex_item_equal:n \c_nine      % tab
22 \regex_item_equal:n \c_ten       % lf
23 \regex_item_equal:n \c_twelve    % ff
24 \regex_item_equal:n \c_thirteen  % cr
25 }
26 \tl_const:Nn \c_regex_v_tl
27 { \regex_item_range:nn \c_ten \c_thirteen } % lf, vtab, ff, cr
28 \tl_const:Nn \c_regex_w_tl
29 {
30   \regex_item_range:nn \c_ninety_seven { 122 } % a--z
31   \regex_item_range:nn \c_sixty_five { 90 } % A--Z
32   \regex_item_range:nn \c_forty_eight { 57 } % 0--9
33   \regex_item_equal:n { 95 } % _
34 }
35 \tl_const:Nn \c_regex_D_tl
36 { \c_regex_d_tl \regex_break_point:TF { } \regex_break_true:w }
37 \tl_const:Nn \c_regex_H_tl
38 { \c_regex_h_tl \regex_break_point:TF { } \regex_break_true:w }
39 \tl_const:Nn \c_regex_S_tl
40 { \c_regex_s_tl \regex_break_point:TF { } \regex_break_true:w }
41 \tl_const:Nn \c_regex_V_tl
42 { \c_regex_v_tl \regex_break_point:TF { } \regex_break_true:w }
43 \tl_const:Nn \c_regex_W_tl
44 { \c_regex_w_tl \regex_break_point:TF { } \regex_break_true:w }
45 \tl_const:Nn \c_regex_N_tl
46 {
47   \regex_item_equal:n \c_ten
48   \regex_break_point:TF { } { \regex_break_true:w }
49 }

```

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

`\c_regex._tl` The dot meta-character matches any character, except the end marker, whose character code is `-2`.

```

50 \tl_const:cn { c_regex._tl }
51 {
52   \if_num:w \l_regex_current_char_int > - \c_two
53   \exp_after:wN \regex_break_true:w
54   \fi:
55 }

```

(End definition for `\c_regex._tl`. This function is documented on page ??.)

2.2.2 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 2.4.1.

```

56 \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` The last state that was allocated is `\l_regex_max_state_int - 1`, so that `\l_regex_max_state_int` always points to a free state. The starting state and end state of the last group (which any quantifier would act on) are stored as `\l_regex_left/right_state_int`. In almost all cases, the left and right pointers only differ by 1.

```

57 \int_new:N \l_regex_max_state_int
58 \int_new:N \l_regex_left_state_int
59 \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` 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.

```

60 \seq_new:N \l_regex_left_state_seq
61 \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` 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.

```

62 \seq_new:N \l_regex_end_group_seq
63 \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_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`. The `max_int` variable is used in the case of the special groups `(?|...|...)`, which reset the capturing group number in each alternative: at the end of the group, we must use a number larger than every capturing group appearing in any alternative.

```

64 \int_new:N \l_regex_capturing_group_int
65 \seq_new:N \l_regex_capturing_group_seq
66 \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.

```

67 \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` 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. Errata: the `\l_regex_catcodes_default_int` is not used yet, because the idea of having `\cL(...|...)` act on the whole contents of the group is not yet implemented.

```

\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

```

```

68 \int_new:N \l_regex_catcodes_int
69 \int_new:N \l_regex_catcodes_default_int
70 \int_const:Nn \c_regex_catcode_C_int { "1 }
71 \int_const:Nn \c_regex_catcode_B_int { "4 }
72 \int_const:Nn \c_regex_catcode_E_int { "10 }
73 \int_const:Nn \c_regex_catcode_M_int { "40 }
74 \int_const:Nn \c_regex_catcode_T_int { "100 }
75 \int_const:Nn \c_regex_catcode_P_int { "1000 }
76 \int_const:Nn \c_regex_catcode_U_int { "4000 }
77 \int_const:Nn \c_regex_catcode_D_int { "10000 }
78 \int_const:Nn \c_regex_catcode_S_int { "100000 }
79 \int_const:Nn \c_regex_catcode_L_int { "400000 }
80 \int_const:Nn \c_regex_catcode_O_int { "1000000 }
81 \int_const:Nn \c_regex_catcode_A_int { "4000000 }
82 \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.

```

83 \bool_new:N \l_regex_catcodes_bool

```

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

`\l_regex_tests_bool` The tests which should be performed on an item of the token list are stored in `\l_regex_tests_tl`. In the case of character classes, `\l_regex_tests_bool` is `false` for negative character classes. In the other case, `\l_regex_tests_tl` is directly converted to a state of the NFA. The two variables are cleared (boolean set to `true`) after looking for a quantifier for the class or single character or property. It would seem cleaner to clear them instead before grabbing the next character or class, but that doesn't interact correctly with constructions like `\cL[...]` where `\cL` inserts some tests in `\l_regex_tests_tl`, which should not be eliminated before finding the class.

```

84 \bool_new:N \l_regex_tests_bool
85 \tl_new:N \l_regex_tests_tl

```

(End definition for `\l_regex_tests_bool` and `\l_regex_tests_tl`. These functions are documented on page ??.)

`\l_regex_tests_saved_bool` In nested class, which can only occur as `[\cA[...]]`, the variables `\l_regex_tests_bool`, `\l_regex_tests_tl`, and `\l_regex_catcodes_int` from the outer class must be saved while processing the inner class.

```

\l_regex_tests_saved_bool
\l_regex_tests_saved_tl
\l_regex_catcodes_saved_int

```

```

86 \bool_new:N \l_regex_tests_saved_bool
87 \tl_new:N \l_regex_tests_saved_tl
88 \int_new:N \l_regex_catcodes_saved_int

```

(End definition for `\l_regex_tests_saved_bool`, `\l_regex_tests_saved_tl`, and `\l_regex_catcodes_saved_int`. These functions are documented on page ??.)

2.2.3 Variables used when matching

`\l_regex_min_index_int` `\l_regex_max_index_int` The tokens in the query are indexed from `\l_regex_min_index_int` for the first to `\l_regex_max_index_int - 1` for the last, and their information is stored in `\muskip` and `\toks` registers with those numbers. We don't start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA.

```

89 \int_new:N \l_regex_min_index_int
90 \int_new:N \l_regex_max_index_int

```

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

`\l_regex_nesting_int` The first phase when matching is to go once through the query token list and store the information for each token as `\muskip` and `\toks` registers. During this phase, `\l_regex_nesting_int` counts the balance of begin-group and end-group character tokens which appear before a given point in the string, and is stored as the shrink component of the `\muskip` registers. This integer is also used to keep track of the balance in the replacement text.

```

91 \int_new:N \l_regex_nesting_int

```

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

`\l_regex_current_index_int` `\l_regex_start_index_int` `\l_regex_success_index_int` While reading through the query token list, `\l_regex_current_index_int` is the position in the token list, starting at `\l_regex_min_index_int` for the first token. Each match begins at the position given by `\l_regex_start_index_int`. Whenever an execution thread succeeds, the corresponding index is stored into `\l_regex_success_index_int`, which will be the next starting index (except in the case of empty matches).

```

92 \int_new:N \l_regex_current_index_int
93 \int_new:N \l_regex_start_index_int
94 \int_new:N \l_regex_success_index_int

```

(End definition for `\l_regex_current_index_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 and category codes of the token at the current position; `\l_regex_current_token_tl` which o- and x-expand to the token (as provided by `\tl_analysis:n`); the character code of the token at the previous position; and the character code of the result of changing the case of the current token (A-Z↔a-z). This last integer is only computed if the “case insensitive” option (`?i`) is used in the regex. The `\l_regex_current_char_int` variable is also used in various other phases to hold a character code.

```

95 \int_new:N \l_regex_current_char_int
96 \int_new:N \l_regex_current_catcode_int
97 \tl_new:N \l_regex_current_token_tl
98 \int_new:N \l_regex_last_char_int
99 \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.

```

100 \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 some cases of groups with quantifiers, `\l_regex_current_state_int` is shifted to a fake value for transitions to point to the correct states.

```

101 \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` 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 successful thread will remain there.

```

102 \prop_new:N \l_regex_current_submatches_prop
103 \prop_new:N \l_regex_success_submatches_prop
      (End definition for \l_regex_current_submatches_prop. This function is documented on page ??.)

```

`\l_regex_step_int` In the case of repeated matches, `\l_regex_current_index_int` is reset to the end-position of the previous match. In contrast, `\l_regex_step_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).

```

104 \int_new:N \l_regex_step_int
      (End definition for \l_regex_step_int. This function is documented on page ??.)

```

`\l_regex_max_active_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 i th item of the array is `\skip_i`. The largest index used after treating the previous character is `\l_regex_max_active_int`. At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `\l_regex_max_active_int` is reset to zero, effectively clearing the array.

```

105 \int_new:N \l_regex_max_active_int
      (End definition for \l_regex_max_active_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 empty. For multiple matching, the token list is set to repeat the matching.

```

106 \tl_new:N \l_regex_every_match_tl
      (End definition for \l_regex_every_match_tl. This function is documented on page ??.)

```

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

```

107 \bool_new:N \l_regex_fresh_thread_bool
108 \bool_new:N \l_regex_success_empty_bool
109 \cs_new_eq:NN \regex_if_two_empty_matches:F \use:n
      (End definition for \l_regex_fresh_thread_bool. This function is documented on page ??.)

```

`\g_regex_success_bool` The boolean `\g_regex_success_bool` is true if there was at least one successful 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.

```

110 \bool_new:N \g_regex_success_bool
111 \bool_new:N \l_regex_saved_success_bool
112 \bool_new:N \l_regex_success_match_bool
      (End definition for \g_regex_success_bool. This function is documented on page ??.)

```

2.2.4 Regular expression variables

`\c_regex_no_match_regex` This regular expression matches nothing, but is still a valid regular expression. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```

113 \tl_const:Nn \c_regex_no_match_regex
114 {
115   \regex_nfa:Nw \c_regex_no_match_regex
116   \l_regex_max_state_int = \c_one
117   \l_regex_capturing_group_int = \c_zero
118   \tex_toks:D \c_zero { \s_stop }
119   \s_stop
120 }
      (End definition for \c_regex_no_match_regex. This function is documented on page ??.)

```

`\regex_new:N` As for many data types deriving from the `tl` data type, creating a new one is simply a matter of checking that it does not exist yet, and setting it equal to a default value.

```

121 \cs_new_protected:Npn \regex_new:N #1
122 {
123   \chk_if_free_cs:N #1
124   \cs_gset_eq:NN #1 \c_regex_no_match_regex
125 }

```

(End definition for `\regex_new:N`. This function is documented on page 6.)

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

¹²⁶ `\regex_new:N \l_regex_internal_regex`

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

2.2.5 Other variables

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

¹²⁷ `\int_new:N \l_regex_replacement_csnames_int`

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

`\l_regex_replacement_int`

¹²⁸ `\int_new:N \l_regex_replacement_int`

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

`\l_regex_submatch_int`

¹²⁹ `\int_new:N \l_regex_submatch_int`

¹³⁰ `\int_new:N \l_regex_submatch_start_int`

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

`\l_regex_submatch_start_int`

`\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.

¹³¹ `\int_new:N \l_regex_match_count_int`

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

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

¹³² `\flag_new:n { regex_begin }`

¹³³ `\flag_new:n { regex_end }`

(End definition for `regex_begin` and `regex_end`. These functions are documented on page ??.)

2.3 Helpers

2.3.1 Toks

Two unrelated sets of functions for manipulating `\toks` registers.

`\regex_toks_put_left: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.

```

134 \cs_new_protected:Npn \regex_toks_put_left:Nx #1#2
135 {
136   \cs_set_nopar:Npx \regex_tmp:w \s_stop { \s_stop #2 }
137   \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
138   { \exp_after:wN \regex_tmp:w \tex_the:D \tex_toks:D #1 }
139 }
140 \cs_new_protected:Npn \regex_toks_put_right:Nx #1#2
141 {
142   \cs_set_nopar:Npx \regex_tmp:w {#2}
143   \tex_toks:D #1 \exp_after:wN
144   { \tex_the:D \tex_toks:D \exp_after:wN #1 \regex_tmp:w }
145 }
```

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

`\regex_toks_range:nn` Some non-expandable functions store pieces of their result in `\toks` registers. Those pieces are concatenated using `\regex_toks_range:nn`, which expects two integers, $\langle start \rangle$ and $\langle end \rangle$, and expands to the contents of the `\toks` registers from $\langle start \rangle$ (inclusive) to $\langle end \rangle$ (exclusive). This correctly expands to nothing if $\langle start \rangle \geq \langle end \rangle$.

`\regex_toks_range:ww`

```

146 \cs_new:Npn \regex_toks_range:nn #1#2
147 {
148   \exp_after:wN \regex_toks_range:ww
149   \int_use:N \int_eval:w #1 \exp_after:wN ;
150   \int_use:N \int_eval:w #2 ;
151   \prg_break_point:n { }
152 }
153 \cs_new:Npn \regex_toks_range:ww #1 ; #2 ;
154 {
155   \if_num:w #1 < #2 \exp_stop_f:
156   \else:
157     \exp_after:wN \prg_map_break:
158     \fi:
159     \tex_the:D \tex_toks:D #1 \exp_stop_f:
160     \exp_after:wN \regex_toks_range:ww
161     \int_use:N \int_eval:w #1 + \c_one ; #2 ;
162 }
```

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

On the one hand, 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`. On the other hand, we provide functions to unpack the contents from a range of `\toks` within an x-expanding assignment.

When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l_regex_min_index_int` inclusive to `\l_regex_max_index_int` exclusive. Furthermore, the `\skip` registers hold...

```
\regex_query_submatch:nn
\regex_query_submatch:w
163 \cs_new:Npn \regex_query_submatch:nn #1#2
164 {
165   \if_num:w #1 < \l_regex_capturing_group_int
166     \exp_after:wN \regex_query_submatch:w
167     \int_use:N \int_eval:w #1 + #2 ;
168   \fi:
169 }
170 \cs_new:Npn \regex_query_submatch:w #1 ;
171 {
172   \regex_toks_range:nn
173   { \tex_skip:D #1 }
174   { \etex_gluestretch:D \tex_skip:D #1 }
175 }
    (End definition for \regex_query_submatch:nn. This function is documented on page ??.)
```

2.3.2 Sequences

`\regex_seq_pop_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`.

```
\regex_seq_get_int:NN
\regex_seq_push_int:NN
176 \cs_new_protected:Npn \regex_seq_pop_int:NN #1#2
177 {
178   \seq_pop:NN #1 \l_regex_internal_a_tl
179   \int_set:Nn #2 \l_regex_internal_a_tl
180 }
181 \cs_new_protected:Npn \regex_seq_get_int:NN #1#2
182 {
183   \seq_get:NN #1 \l_regex_internal_a_tl
184   \int_set:Nn #2 \l_regex_internal_a_tl
185 }
186 \cs_new_protected:Npn \regex_seq_push_int:NN #1#2
187 { \seq_push:Nn #1 { \int_use:N #2 } }
    (End definition for \regex_seq_pop_int:NN. This function is documented on page ??.)
```

`\regex_seq_pop_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.

```
\regex_seq_get_use:N
188 \cs_new_protected:Npn \regex_seq_pop_use:N #1
189 {
190   \seq_pop:NN #1 \l_regex_internal_a_tl
191   \l_regex_internal_a_tl
```

```

192 }
193 \cs_new_protected:Npn \regex_seq_get_use:N #1
194 {
195   \seq_get:NN #1 \l_regex_internal_a_tl
196   \l_regex_internal_a_tl
197 }

```

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

2.3.3 Grabbing digits

\regex_get_digits:nw When parsing the { quantifier, we need a tool to grab digits until reaching the first non-digit. This \regex_get_digits:nw function places whatever digits it found as a brace group after #1.

```

198 \cs_new_protected:Npn \regex_get_digits:nw #1
199 {
200   \tex_afterassignment:D \regex_tmp:w
201   \cs_set_nopar:Npx \regex_tmp:w
202   {
203     \exp_not:n {#1}
204     { \if_false: } } \fi:
205     \regex_get_digits_aux:NN
206   }
207   \cs_new:Npn \regex_get_digits_aux:NN #1#2
208   {
209     \if_meaning:w \regex_build_raw:N #1
210     \if_num:w 9 < 1 \exp_not:N #2 \exp_stop_f:
211     #2
212     \else:
213       \regex_get_digits_end:w #1 #2
214     \fi:
215     \else:
216       \regex_get_digits_end:w #1 #2
217     \fi:
218     \regex_get_digits_aux:NN
219   }
220   \cs_new:Npn \regex_get_digits_end:w #1 \fi: #2 \regex_get_digits_aux:NN
221   {
222     \fi: #2
223     \if_false: { { \fi: } }
224     #1
225   }

```

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

2.3.4 Testing characters

\regex_break_point:TF 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> ... <testn>
    \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.

```

226 \cs_new_protected:Npn \regex_break_true:w
227     #1 \regex_break_point:TF #2 #3 {#2}
228 \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_caseful_equal:n Simple comparisons triggering \regex_break_true:w when true.

```

\regex_item_caseful_range:nn
\regex_item_caseful_geq:n
229 \cs_new_protected:Npn \regex_item_caseful_equal:n #1
230 {
231     \if_num:w #1 = \l_regex_current_char_int
232     \exp_after:wN \regex_break_true:w
233     \fi:
234 }
235 \cs_new_protected:Npn \regex_item_caseful_range:nn #1 #2
236 {
237     \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
238     \reverse_if:N \if_num:w #2 < \l_regex_current_char_int
239     \exp_after:wN \exp_after:wN \exp_after:wN \regex_break_true:w
240     \fi:
241     \fi:
242 }
243 \cs_new_protected:Npn \regex_item_caseful_geq:n #1
244 {
245     \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
246     \exp_after:wN \regex_break_true:w
247     \fi:
248 }
    (End definition for \regex_item_caseful_equal:n. This function is documented on page ??.)

```

\regex_item_caseless_equal:n For caseless matching, we perform the test both on \l_regex_current_char_int and \l_regex_case_changed_char_int.

```

\regex_item_caseless_range:nn
\regex_item_caseless_geq:n
249 \cs_new_protected:Npn \regex_item_caseless_equal:n #1
250 {
251     \if_num:w #1 = \l_regex_current_char_int
252     \exp_after:wN \regex_break_true:w
253     \fi:
254     \if_num:w #1 = \l_regex_case_changed_char_int
255     \exp_after:wN \regex_break_true:w
256     \fi:
257 }
258 \cs_new_protected:Npn \regex_item_caseless_range:nn #1 #2
259 {
260     \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
261     \reverse_if:N \if_num:w #2 < \l_regex_current_char_int

```

```

262     \exp_after:wN \exp_after:wN \exp_after:wN \regex_break_true:w
263     \fi:
264   \fi:
265   \reverse_if:N \if_num:w #1 > \l_regex_case_changed_char_int
266     \reverse_if:N \if_num:w #2 < \l_regex_case_changed_char_int
267     \exp_after:wN \exp_after:wN \exp_after:wN \regex_break_true:w
268     \fi:
269   \fi:
270 }
271 \cs_new_protected:Npn \regex_item_caseless_geq:n #1
272 {
273   \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
274   \exp_after:wN \regex_break_true:w
275   \fi:
276   \reverse_if:N \if_num:w #1 > \l_regex_case_changed_char_int
277   \exp_after:wN \regex_break_true:w
278   \fi:
279 }

```

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

`\regex_item_equal:n` By default, matching takes the letter case into account. Note that those functions are
`\regex_item_range:nn` not protected: they will expand at the building phase, hard-coding which states take care
`\regex_item_geq:n` of caseless versus careful matching.

```

280 \cs_new:Npn \regex_item_equal:n { \regex_item_careful_equal:n }
281 \cs_new:Npn \regex_item_range:nn { \regex_item_careful_range:nn }
282 \cs_new:Npn \regex_item_geq:n { \regex_item_careful_geq:n }

```

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

`\regex_build_caseless:` Switch between careful and caseless matching. This is only done during the building
`\regex_build_careful:` phase.

```

283 \cs_new_protected_nopar:Npn \regex_build_caseless:
284 {
285   \bool_set_true:N \l_regex_caseless_bool
286   \cs_set:Npn \regex_item_equal:n { \regex_item_caseless_equal:n }
287   \cs_set:Npn \regex_item_range:nn { \regex_item_caseless_range:nn }
288   \cs_set:Npn \regex_item_geq:n { \regex_item_caseless_geq:n }
289 }
290 \cs_new_protected_nopar:Npn \regex_build_careful:
291 {
292   \bool_set_false:N \l_regex_caseless_bool
293   \cs_set:Npn \regex_item_equal:n { \regex_item_careful_equal:n }
294   \cs_set:Npn \regex_item_range:nn { \regex_item_careful_range:nn }
295   \cs_set:Npn \regex_item_geq:n { \regex_item_careful_geq:n }
296 }

```

(End definition for \regex_build_caseless: and \regex_build_careful:. 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
`\regex_item_catcode_aux:` (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.

```

297 \cs_new_protected:Npn \regex_item_catcode_aux:
298 {
299   "
300   \if_case:w \l_regex_current_catcode_int
301     1          \or: 4          \or: 10         \or: 40
302   \or: 100     \or:           \or: 1000        \or: 4000
303   \or: 10000   \or:           \or: 100000       \or: 400000
304   \or: 1000000 \or: 4000000 \else: 1*\c_zero
305   \fi:
306 }
307 \cs_new_protected:Npn \regex_item_catcode:nT #1
308 {
309   \if_int_odd:w \int_eval:w #1 / \regex_item_catcode_aux: \int_eval_end:
310   \exp_after:wN \use:n
311   \else:
312   \exp_after:wN \use_none:n
313   \fi:
314 }

```

(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⟩`.

```

315 \cs_new_protected:Npn \regex_item_cs:n #1
316 {
317   \int_compare:nNnT \l_regex_current_catcode_int = \c_zero
318   {
319     \tl_set:Nn \l_regex_internal_regex {#1}
320     \bool_set_eq:NN \l_regex_saved_success_bool \g_regex_success_bool
321     \exp_args:NNx \regex_match:NnTF \l_regex_internal_regex
322     {
323       \exp_after:wN \exp_after:wN
324       \exp_after:wN \cs_to_str:N \l_regex_current_token_tl
325     }
326     {
327       \bool_gset_eq:NN \g_regex_success_bool \l_regex_saved_success_bool
328       \regex_break_true:w
329     }
330     { \bool_gset_eq:NN \g_regex_success_bool \l_regex_saved_success_bool }
331   }
332 }

```

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

2.3.5 Simple character escape

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

Usage: `\regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* *{<token list>}* The *<token list>* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *<fn1>*, and escaped characters are fed to the function *<fn2>* within an `x`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *<fn3>*. The result is then left in the input stream.

The idea is to feed unescaped characters to one function, escaped characters to another, and feed `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` converted to the appropriate character to a third function. Spaces are ignored unless escaped.

`\regex_escape_use:nnnn` Go through #4 once, applying #1, #2, or #3 to each character (after de-escaping it), then
`\regex_escape_loop:N` leave the result in the input stream. Most of the work is done within an `x`-expanding
`\regex_escape\:w` assignment, but the `\x` escape sequence cannot be done in that way. Therefore, we interrupt the assignment at each `\x` escape sequence, store the partial result in a `\toks` register, and at the end unpack all of the `\toks` registers to the left of the last chunk of `\l_regex_internal_a_tl`.

```

333 \cs_new_protected:Npn \regex_escape_use:nnnn #1#2#3#4
334 {
335   \group_begin:
336   \cs_set_nopar:Npn \regex_escape_unescaped:N ##1 { #1 }
337   \cs_set_nopar:Npn \regex_escape_escaped:N ##1 { #2 }
338   \cs_set_nopar:Npn \regex_escape_raw:N ##1 { #3 }
339   \int_set:Nn \tex_escapechar:D { 92 }
340   \str_gset_other:Nn \g_regex_internal_tl {#4}
341   \int_zero:N \l_regex_internal_a_int
342   \tl_set:Nx \l_regex_internal_a_tl
343   {
344     \exp_after:wN \regex_escape_loop:N \g_regex_internal_tl
345     { break } \prg_break_point:n { }
346   }
347   \use:x
348   {
349     \group_end:
350     \regex_toks_range:nn \c_zero \l_regex_internal_a_int
351     \exp_not:o \l_regex_internal_a_tl
352   }
353 }
```

```

354 \cs_new:Npn \regex_escape_loop:N #1
355 {
356   \cs_if_exist_use:cF { regex_escape_\token_to_str:N #1:w }
357   { \regex_escape_unescaped:N #1 }
358   \regex_escape_loop:N
359 }
360 \cs_new_nopar:cpn { regex_escape_ \c_backslash_str :w }
361   \regex_escape_loop:N #1
362 {
363   \cs_if_exist_use:cF { regex_escape_/ \token_to_str:N #1:w }
364   { \regex_escape_escaped:N #1 }
365   \regex_escape_loop:N
366 }

```

(End definition for \regex_escape_use:nnnn. This function is documented on page ??.)

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

```

\regex_escape_escaped:N
\regex_escape_raw:N
367 \cs_new_eq:NN \regex_escape_unescaped:N ?
368 \cs_new_eq:NN \regex_escape_escaped:N ?
369 \cs_new_eq:NN \regex_escape_raw:N ?

```

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

\regex_escape_q_recursion_tail:w The loop is ended upon seeing \q_recursion_tail. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

```

\regex_escape_/q_recursion_tail:w
\regex_escape_/q_recursion_tail:w
\regex_escape_ :w
\regex_escape_/a:w
\regex_escape_/e:w
\regex_escape_/f:w
\regex_escape_/n:w
\regex_escape_/r:w
\regex_escape_/t:w
370 \cs_new_eq:NN \regex_escape_break:w \prg_map_break:
371 \cs_new_nopar:cpn { regex_escape_/break:w }
372 {
373   \if_false: { \fi: }
374   \msg_kernel_error:nn { regex } { trailing-backslash }
375   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
376 }
377 \cs_new_nopar:cpn { regex_escape_~:w } { }
378 \cs_new_nopar:cpx { regex_escape_/a:w }
379   { \exp_not:N \regex_escape_raw:N \iow_char:N \^^G }
380 \cs_new_nopar:cpx { regex_escape_/t:w }
381   { \exp_not:N \regex_escape_raw:N \iow_char:N \^^I }
382 \cs_new_nopar:cpx { regex_escape_/n:w }
383   { \exp_not:N \regex_escape_raw:N \iow_char:N \^^J }
384 \cs_new_nopar:cpx { regex_escape_/f:w }
385   { \exp_not:N \regex_escape_raw:N \iow_char:N \^^L }
386 \cs_new_nopar:cpx { regex_escape_/r:w }
387   { \exp_not:N \regex_escape_raw:N \iow_char:N \^^M }
388 \cs_new_nopar:cpx { regex_escape_/e:w }
389   { \exp_not:N \regex_escape_raw:N \iow_char:N \^^[ }

```

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

\regex_escape_/x:w When \x is encountered, interrupt the assignment, and distinguish the cases of a braced or unbraced syntax. In the braced case, collect arbitrarily many hexadecimal digits,

```

\regex_escape_x_test:N
\regex_escape_x_unbraced_i:N
\regex_escape_x_unbraced_ii:N
\regex_escape_x_braced_loop:N
\regex_escape_x_braced_end:N
\regex_escape_x_end:

```

building the number in `\l_regex_current_char_int` (using `\str_aux_hexadecimal_use:NTF`), and check that the run of digits was stopped by a closing brace. In the unbraced case, collect up to two hexadecimal digits, possibly less, building the character number in `\l_regex_current_char_int`. In both cases, once all digits have been collected, use the TeX primitive `\lowercase` to produce that character, and use an `\if_false:` trick to restart the assignment.

```

390 \cs_new_nopar:cpn { regex_escape_/x:w } \regex_escape_loop:N
391 {
392   \if_false: { \fi: }
393   \tex_toks:D \l_regex_internal_a_int
394   \exp_after:wN { \l_regex_internal_a_tl }
395   \int_incr:N \l_regex_internal_a_int
396   \int_zero:N \l_regex_current_char_int
397   \regex_escape_x_test:N
398 }
399 \cs_new_protected:Npx \regex_escape_x_test:N #1
400 {
401   \exp_not:N \token_if_eq_charcode:NNTF \c_space_token #1
402   { \exp_not:N \regex_escape_x_test:N }
403   {
404     \l_regex_current_char_int = "0
405     \exp_not:N \token_if_eq_charcode:NNTF \c_lbrace_str #1
406     { \exp_not:N \regex_escape_x_braced_loop:N }
407     { \exp_not:N \regex_escape_x_unbraced_i:N #1 }
408   }
409 }
410 \cs_new:Npn \regex_escape_x_unbraced_i:N #1
411 {
412   \str_aux_hexadecimal_use:NTF #1
413   { \regex_escape_x_unbraced_ii:N }
414   { \exp_stop_f: \regex_escape_x_end: #1 }
415 }
416 \cs_new:Npn \regex_escape_x_unbraced_ii:N #1
417 {
418   \token_if_eq_charcode:NNTF \c_space_token #1
419   { \regex_escape_x_unbraced_ii:N }
420   {
421     \str_aux_hexadecimal_use:NTF #1
422     { \exp_stop_f: \regex_escape_x_end: }
423     { \exp_stop_f: \regex_escape_x_end: #1 }
424   }
425 }
426 \cs_new:Npn \regex_escape_x_braced_loop:N #1
427 {
428   \token_if_eq_charcode:NNTF \c_space_token #1
429   { \regex_escape_x_braced_loop:N }
430   {
431     \str_aux_hexadecimal_use:NTF #1
432     { \regex_escape_x_braced_loop:N }

```

```

433         { \exp_stop_f: \regex_escape_x_braced_end:N #1 }
434     }
435 }
436 \cs_new_protected:Npx \regex_escape_x_braced_end:N #1
437 {
438     \exp_not:N \token_if_eq_charcode:NNTF \c_rbrace_str #1
439     { \exp_not:N \regex_escape_x_end: }
440     {
441         \msg_kernel_error:nxx { regex } { x-missing-rbrace }
442         { \int_use:N \l_regex_current_char_int }
443         \exp_not:N \regex_escape_x_end: #1
444     }
445 }
446 \group_begin:
447     \char_set_catcode_other:N \^^@
448     \cs_new_protected_nopar:Npn \regex_escape_x_end:
449     {
450         \if_num:w \l_regex_current_char_int > \c_max_char_int
451         \msg_kernel_error:nxx { regex } { x-overflow }
452         { \int_use:N \l_regex_current_char_int }
453         \exp_after:wN \use:n
454     \else:
455         \tex_lccode:D \c_zero \l_regex_current_char_int
456         \exp_after:wN \tl_to_lowercase:n
457     \fi:
458     {
459         \tl_set:Nx \l_regex_internal_a_tl
460         { \if_false: } \fi:
461         \regex_escape_raw:N \^^@
462         \regex_escape_loop:N
463     }
464 }
465 \group_end:

```

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

`\regex_char_if_alphanumeric:N`
`\regex_char_if_special:N`

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:n` 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).

```

466 \prg_new_conditional:Npnn \regex_char_if_special:N #1 { TF }
467 {
468   \if_num:w '#1 < \c_ninety_one
469     \if_num:w '#1 < \c_fifty_eight
470       \if_num:w '#1 < \c_forty_eight
471         \if_num:w '#1 < \c_thirty_two
472           \prg_return_false: \else: \prg_return_true: \fi:
473         \else: \prg_return_false: \fi:
474       \else:
475         \if_num:w '#1 < \c_sixty_five
476           \prg_return_true: \else: \prg_return_false: \fi:
477         \fi:
478       \else:
479         \if_num:w '#1 < \c_one_hundred_twenty_three
480           \if_num:w '#1 < \c_ninety_seven
481             \prg_return_true: \else: \prg_return_false: \fi:
482           \else:
483             \if_num:w '#1 < \c_one_hundred_twenty_seven
484               \prg_return_true: \else: \prg_return_false: \fi:
485             \fi:
486           \fi:
487         \fi:
488 \prg_new_conditional:Npnn \regex_char_if_alphanumeric:N #1 { TF }
489 {
490   \if_num:w '#1 < \c_ninety_one
491     \if_num:w '#1 < \c_fifty_eight
492       \if_num:w '#1 < \c_forty_eight
493         \prg_return_false: \else: \prg_return_true: \fi:
494       \else:
495         \if_num:w '#1 < \c_sixty_five
496           \prg_return_false: \else: \prg_return_true: \fi:
497         \fi:
498       \else:
499         \if_num:w '#1 < \c_one_hundred_twenty_three
500           \if_num:w '#1 < \c_ninety_seven
501             \prg_return_false: \else: \prg_return_true: \fi:
502           \else:
503             \prg_return_false:
504           \fi:
505         \fi:
506 }

```

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

2.4 Building

2.4.1 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 $m \rightarrow (m - 15)/13$, 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.

```

507 \cs_new_nopar:Npn \regex_build_if_in_class:TF
508 {
509   \if_int_odd:w \l_regex_build_mode_int
510     \exp_after:wN \use_i:nn
511   \else:
512     \exp_after:wN \use_ii:nn
513   \fi:
514 }

```

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

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

```

515 \cs_new_nopar:Npn \regex_build_if_assertions_forbidden:TF
516 {
517   \if_int_odd:w \l_regex_build_mode_int
518     \exp_after:wN \use_i:nn
519   \else:
520     \if_int_compare:w \l_regex_build_mode_int > \c_zero
521       \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
522     \else:
523       \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
524     \fi:
525   \fi:
526 }

```

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

2.4.2 Helpers for building an NFA

`\regex_build_new_state:` 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 update the left, right, and max *(states)*.

```

527 \cs_new_protected_nopar:Npn \regex_build_new_state:
528 {
529   \tex_toks:D \l_regex_max_state_int { \s_stop }
530   \int_set_eq:NN \l_regex_left_state_int \l_regex_right_state_int
531   \int_set_eq:NN \l_regex_right_state_int \l_regex_max_state_int
532   \int_incr:N \l_regex_max_state_int
533 }

```

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

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

```

534 \cs_new_protected:Npn \regex_build_transition:NN #1#2
535 {
536   \regex_build_new_state:

```



```

537     \regex_toks_put_right:Nx \l_regex_left_state_int
538     { #1 { \int_eval:n { #2 - \l_regex_left_state_int } } }
539   }
540 \cs_new_protected:Npn \regex_build_transitions:NNNN #1#2#3#4
541 {
542   \regex_build_new_state:
543   \regex_toks_put_right:Nx \l_regex_left_state_int
544   {
545     #1 { \int_eval:n { #2 - \l_regex_left_state_int } }
546     #3 { \int_eval:n { #4 - \l_regex_left_state_int } }
547   }
548 }

```

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

2.4.3 From regex to NFA: framework

\regex_build:w There are two situations where we want to convert a regular expression given as a string to an NFA. The first case is when the regex is given directly by the user, as an `n` argument to one of the `l3regex` functions, and the second case is when building the regex given as an argument of the `\c{...}` escape sequence within another regular expression. In both cases, some variables must be reset before starting to build the NFA, using `\regex_build:w`. Once building has ended (for instance when encountering the closing brace in `\c{...}`), we make sure to close any dangling class or open group, then add `\regex_action_success:` to make the match successful if it reaches the last state. The capturing group number is incremented: then its value tells us how many capturing group there are, including 0.

```

549 \cs_new_protected_nopar:Npn \regex_build:w
550 {
551   \int_set_eq:NN \l_regex_catcodes_default_int \c_regex_catcodes_all_int
552   \int_set_eq:NN \l_regex_catcodes_int \l_regex_catcodes_default_int
553   \int_set_eq:NN \l_regex_capturing_group_int \c_zero
554   \seq_clear:N \l_regex_capturing_group_seq
555   \tl_clear:N \l_regex_tests_tl
556   \bool_set_true:N \l_regex_tests_bool
557   \int_zero:N \l_regex_max_state_int
558   \regex_build_new_state:
559 }
560 \cs_new_protected_nopar:Npn \regex_build_end:
561 {
562   \regex_build_exit_class:
563   \regex_build_exit_groups:
564   \regex_toks_put_right:Nx \l_regex_right_state_int
565   { \regex_action_success: }
566   \int_incr:N \l_regex_capturing_group_int
567 }
568 \cs_new_protected_nopar:Npn \regex_build_exit_class:
569 {
570   \regex_build_if_in_class:TF

```

```

571     {
572         \msg_kernel_error:nn { regex } { missing-rbrack }
573         \use:c { regex_build_]: }
574     }
575     { }
576 }
577 \cs_new_protected_nopar:Npn \regex_build_exit_groups:
578 {
579     \seq_if_empty:NF \l_regex_capturing_group_seq
580     {
581         \msg_kernel_error:nnx { regex } { missing-rparen }
582         { \seq_length:N \l_regex_capturing_group_seq }
583         \prg_replicate:nn
584         { \seq_length:N \l_regex_capturing_group_seq }
585         { \regex_build_close_aux: \regex_build_group_: }
586     }
587 }

```

(End definition for \regex_build:w and \regex_build_end:. These functions are documented on page ??.)

`\regex_build:n` Setup some variables with `\regex_build:w`, and additionally set the build mode to 0: we are neither in a class nor a `\c` construction. Initiate the regular expression by a wildcard: the search is unanchored and should be tried at every index in the token list. We surround the regular expression by parentheses with `\regex_build_open_aux:` and `\regex_build_close_aux:`, so that any alternative appears within a group (e.g., `ab|cd` will now be within the external group); this also forms a capturing group (labelled 0), which gives us the whole match as the 0-th submatch. The regular expression itself is parsed by using the generic framework of `\regex_escape_use:nnnn` to recognize special characters, escaped ones, and escape sequences such as `\n` or `\x{02f}`. This results in successive `\regex_build_raw:N`, `\regex_build_escaped:N`, `\regex_build_special:N` followed by their arguments. 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).

```

588 \cs_new_protected:Npn \regex_build:n #1
589 {
590     \regex_build:w
591     \int_set_eq:NN \l_regex_build_mode_int \c_zero
592     \regex_build_new_state:
593     \regex_toks_put_right:Nx \l_regex_left_state_int
594     { \regex_action_start_wildcard: }
595     \regex_build_open_aux:
596     \regex_escape_use:nnnn
597     {
598         \regex_char_if_special:NTF ##1
599         \regex_build_special:N \regex_build_raw:N ##1
600     }
601     {
602         \regex_char_if_alphanumeric:NTF ##1
603         \regex_build_escaped:N \regex_build_raw:N ##1

```

```

604     }
605     { \regex_build_raw:N ##1 }
606     { #1 }
607     \prg_do_nothing: \prg_do_nothing:
608     \prg_do_nothing: \prg_do_nothing:
609     \int_compare:nNnT \l_regex_build_mode_int < \c_zero
610     {
611         \msg_kernel_error:nn { regex } { c-missing-rbrace }
612         \exp_after:wN \regex_build_special:N \c_rbrace_str
613     }
614     \seq_put_right:Nn \l_regex_capturing_group_seq {0}
615     \regex_build_close_aux: \regex_build_group:
616     \regex_build_end:
617 }

```

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

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

`\regex_build_escaped:N`

```

618 \cs_new_protected:Npn \regex_build_special:N #1
619 {
620     \cs_if_exist_use:cF { regex_build_#1: }
621     { \regex_build_raw:N #1 }
622 }
623 \cs_new_protected:Npn \regex_build_escaped:N #1
624 {
625     \cs_if_exist_use:cF { regex_build_/#1: }
626     { \regex_build_raw:N #1 }
627 }

```

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

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

`\regex_build_one:x`

```

628 \cs_new_protected:Npn \regex_build_one:n #1
629 { \regex_build_one:x { \exp_not:n {#1} } }
630 \cs_new_protected:Npn \regex_build_one:x #1
631 {
632     \tl_put_right:Nx \l_regex_tests_tl
633     {
634         \if_num:w \l_regex_catcodes_int < \c_regex_catcodes_all_int
635         \regex_item_catcode:nT { \int_use:N \l_regex_catcodes_int }
636         \else:
637         \exp_after:wN \use:n
638         \fi:
639         {#1}
640     }
641     \int_set_eq:NN \l_regex_catcodes_int \l_regex_catcodes_default_int
642     \if_num:w \l_regex_build_mode_int = \c_two

```

```

643     \l_regex_build_mode_int = \c_zero
644   \else:
645     \if_num:w \l_regex_build_mode_int = \c_six
646     \l_regex_build_mode_int = \c_three
647   \fi:
648 \fi:
649 \if_int_odd:w \l_regex_build_mode_int \else:
650   \exp_after:wN \regex_build_one_quantifier:
651 \fi:
652 }

```

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

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

```

653 \cs_new:Npn \regex_tests_action_cost:n #1
654 {
655   \exp_not:o \l_regex_tests_tl
656   \bool_if:NTF \l_regex_tests_bool
657     { \regex_break_point:TF { \regex_action_cost:n {#1} } { } }
658     { \regex_break_point:TF { } { \regex_action_cost:n {#1} } }
659 }

```

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

2.4.4 Raw characters

\regex_build_raw_error:N Within character classes, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

660 \cs_new_protected:Npn \regex_build_raw_error:N #1
661 {
662   \if_int_odd:w \l_regex_build_mode_int
663     \msg_kernel_error:nnx { regex } { class-bad-escape } {#1}
664   \else:
665     \msg_kernel_error:nnx { regex } { catcode-bad-escape } {#1}
666   \fi:
667   \regex_build_raw:N #1
668 }

```

(End definition for \regex_build_raw_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.

```

669 \cs_new_protected:Npn \regex_build_raw:N #1#2#3
670 {
671   \regex_build_if_in_class:TF
672     { \str_if_eq:nnTF {#2#3} { \regex_build_special:N - } }
673     { \use_ii:nn }
674     { \regex_class_range:Nw #1 }
675   {
676     \regex_build_one:x { \regex_item_equal:n { \int_value:w '#1 ~ } }
677     #2 #3

```

```

678     }
679 }

```

(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.

```

680 \cs_new_protected:Npn \regex_class_range:Nw #1#2#3
681 {
682   \token_if_eq_meaning:NNTF #2 \regex_build_escaped:N % [
683     { \use_i:nn } { \str_if_eq:nnTF { #2#3 } { \regex_build_special:N ] } }
684     {
685       \msg_kernel_warning:nnxx { regex } { range-missing-end } % [
686         {#1} { \token_if_eq_charcode:NNF #3 ] { \c_backslash_str } #3 }
687       \tl_put_right:Nx \l_regex_tests_tl
688         {
689           \regex_item_equal:n { \int_value:w '#1 ~ }
690           \regex_item_equal:n { \int_value:w '- ~ }
691         }
692       #2#3
693     }
694     {
695       \if_num:w '#1 > '#3 \exp_stop_f:
696       \msg_kernel_error:nnxx { regex } { range-backwards } {#1} {#3}
697       \else:
698       \tl_put_right:Nx \l_regex_tests_tl
699         {
700           \if_num:w '#1 = '#3 \exp_stop_f:
701           \regex_item_equal:n
702           \else:
703           \regex_item_range:nn { \int_value:w '#1 ~ }
704           \fi:
705           { \int_value:w '#3 ~ }
706         }
707       \fi:
708     }
709 }

```

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

2.4.5 Character properties

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

```

710 \cs_new_protected_nopar:cpx { regex_build_.: }
711 {
712   \exp_not:N \regex_build_if_in_class:TF
713     { \regex_build_raw:N . }

```

```

714     { \regex_build_one:n \exp_not:c { c_regex_.tl } }
715 }

```

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

\regex_build/d: 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.

```

\regex_build/H: 716 \tl_map_inline:nn { dDhHsSvVwWN }
\regex_build/s: 717 {
\regex_build/S: 718   \cs_new_protected_nopar:cpx { regex_build_/#1: }
\regex_build/v: 719   { \regex_build_one:n \exp_not:c { c_regex_#1_tl } }
\regex_build/V: 720 }

```

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

\regex_build/W:
\regex_build/N:

2.4.6 Anchoring and simple assertions

\regex_build_simple_assertion:nn Assertions are only allowed in modes 0, -2 and -6. In character classes, or immediately following a category code test, escaped letters cause errors, while the meaning of special characters is ignored: #1 is used instead. In modes where the assertion is allowed, we insert the test #2; if the assertion is successful, move from the left state to the right state.

```

721 \cs_new_protected:Npn \regex_build_simple_assertion:nn #1#2
722 {
723   \regex_build_if_assertions_forbidden:TF {#1}
724   {
725     \regex_build_new_state:
726     \regex_toks_put_right:Nx \l_regex_left_state_int
727     {
728       \exp_not:n {#2}
729       {
730         \regex_action_free:n
731         {
732           \int_eval:n
733           { \l_regex_right_state_int - \l_regex_left_state_int }
734         }
735       }
736     }
737     %%A todo: add \regex_assertion_quantifier:
738   }
739 }

```

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

\regex_build^: 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_index_int instead of \c_zero. End anchors match the end of the token list, marked by a character code of -2.

```

\regex_build/Z: 740 \cs_new_protected_nopar:cpn { regex_build_^: }
\regex_build/z:

```

```

741 {
742   \regex_build_simple_assertion:nn { \regex_build_raw:N ^ }
743   { \int_compare:nNnT \l_regex_min_index_int = \l_regex_current_index_int }
744 }
745 \cs_new_protected_nopar:cpn { regex_build_/A: }
746 {
747   \regex_build_simple_assertion:nn { \regex_build_raw_error:N A }
748   { \int_compare:nNnT \l_regex_min_index_int = \l_regex_current_index_int }
749 }
750 \cs_new_protected_nopar:cpn { regex_build_/G: }
751 {
752   \regex_build_simple_assertion:nn { \regex_build_raw_error:N G }
753   { \int_compare:nNnT \l_regex_start_index_int = \l_regex_current_index_int }
754 }
755 \group_begin:
756   \char_set_catcode_other:N \$
757   \cs_new_protected_nopar:cpn { regex_build_$.: } % $
758   {
759     \regex_build_simple_assertion:nn { \regex_build_raw:N $ } % $
760     { \int_compare:nNnT \l_regex_current_char_int < \c_minus_one }
761   }
762 \group_end:
763 \cs_new_protected_nopar:cpn { regex_build_/Z: }
764 {
765   \regex_build_simple_assertion:nn { \regex_build_raw_error:N Z }
766   { \int_compare:nNnT \l_regex_current_char_int < \c_minus_one }
767 }
768 \cs_new_protected_nopar:cpn { regex_build_/z: }
769 {
770   \regex_build_simple_assertion:nn { \regex_build_raw_error:N z }
771   { \int_compare:nNnT \l_regex_current_char_int < \c_minus_one }
772 }

```

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

\regex_build_/b: Contrarily to ^ and \$, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code. Case sensitivity does not change word boundaries.

\regex_build_/B:

\regex_if_word_boundary:TF

```

773 \cs_new_protected_nopar:cpn { regex_build_/b: }
774 {
775   \regex_build_simple_assertion:nn { \regex_build_raw_error:N b }
776   { \regex_if_word_boundary:T }
777 }
778 \cs_new_protected_nopar:cpn { regex_build_/B: }
779 {
780   \regex_build_simple_assertion:nn { \regex_build_raw_error:N B }
781   { \regex_if_word_boundary:TF { } }
782 }
783 \cs_new_protected_nopar:Npn \regex_if_word_boundary:TF
784 {

```

```

785 \group_begin:
786   \int_set_eq:NN \l_regex_current_char_int \l_regex_last_char_int
787   \c_regex_w_tl
788   \regex_break_point:TF
789   { \group_end: \c_regex_W_tl \regex_item_equal:n { -2 } }
790   { \group_end: \c_regex_w_tl }
791   \regex_break_point:TF
792 }
793 \cs_new_protected_nopar:Npn \regex_if_word_boundary:T
794 { \regex_if_word_boundary:TF \use:n \use_none:n }
      (End definition for \regex_build_/b:.. This function is documented on page ??.)

```

2.4.7 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.

```

795 \cs_new_protected_nopar:cpn { regex_build_[: }
796 {
797   \regex_build_if_in_class:TF
798   { \regex_build_raw:N [ }
799   { \regex_class_first:NNNN }
800 }
      (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 ($m \rightarrow (m - 15)/13$, 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_tests_tl`. Otherwise look for quantifiers.

```

801 \cs_new_protected:cpn { regex_build_] : }
802 {
803   \regex_build_if_in_class:TF
804   {
805     \if_num:w \l_regex_build_mode_int > \c_sixteen
806     \tl_set:Nx \l_regex_tests_tl
807     {
808       \exp_not:o \l_regex_tests_saved_tl
809       \if_num:w \l_regex_catcodes_saved_int < \c_regex_catcodes_all_int
810       \regex_item_catcode:nT
811       { \int_use:N \l_regex_catcodes_saved_int }
812     \else:
813       \exp_after:wN \use:n
814     \fi:
815     {
816       \exp_not:o \l_regex_tests_tl
817       \bool_if:NF \l_regex_tests_bool
818       { \regex_break_point:TF { } \regex_break_true:w }
819     }
820   }
821   \bool_set_eq:NN \l_regex_tests_bool \l_regex_tests_saved_bool

```



```

822     \fi:
823     \tex_advance:D \l_regex_build_mode_int - \c_fifteen
824     \tex_divide:D \l_regex_build_mode_int \c_thirteen
825     \if_int_odd:w \l_regex_build_mode_int \else:
826     \exp_after:wN \regex_build_one_quantifier:
827     \fi:
828   }
829   { \regex_build_raw:N }
830 }

```

(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_tests_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_tests_bool` and `\l_regex_tests_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_tests_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).

```

831 \cs_new_protected:Npn \regex_class_first:NNNN #1#2#3#4
832 {
833   \l_regex_build_mode_int = \int_value:w \l_regex_build_mode_int 3 ~ %
834   \if_num:w \l_regex_build_mode_int > \c_sixteen
835     \tl_set_eq:NN \l_regex_tests_saved_tl \l_regex_tests_tl
836     \bool_set_eq:NN \l_regex_tests_saved_bool \l_regex_tests_bool
837     \int_set_eq:NN \l_regex_catcodes_saved_int \l_regex_catcodes_int
838     \int_set_eq:NN \l_regex_catcodes_int \c_regex_catcodes_all_int
839   \fi:
840   \token_if_eq_meaning:NNTF #1 \regex_build_special:N
841   {
842     \token_if_eq_charcode:NNTF #2 ^
843     {
844       \bool_set_false:N \l_regex_tests_bool % [
845       \str_if_eq:nnTF {#3#4} { \regex_build_special:N } {
846         { \regex_build_raw:N }
847       }
848       { % [
849         \token_if_eq_charcode:NNTF #2 ]
850         { \regex_build_raw:N #2 }
851         { #1 #2 }
852       }
853     }
854     { #1 #2 }
855   #3 #4
856 }

```

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

2.4.8 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.

```

857 \cs_new_protected:cpn { regex_build_/c: }
858 {
859   \if_case:w \l_regex_build_mode_int
860     \exp_after:wN \regex_build_c_aux:wNN
861   \or: \or: \or: \exp_after:wN \regex_build_c_aux:wNN
862   \fi:
863   \msg_kernel_error:nn { regex } { c-bad-mode }
864   \s_stop
865 }

```

(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.

```

866 \cs_new_protected:Npn \regex_build_c_aux:wNN #1 \s_stop #2#3
867 {
868   \token_if_eq_meaning:NNTF #2 \regex_build_raw:N
869   {
870     \cs_if_exist:cTF { c_regex_catcode_#3_int }
871     {
872       \int_set_eq:Nc \l_regex_catcodes_int { c_regex_catcode_#3_int }
873       \l_regex_build_mode_int
874       = \if_case:w \l_regex_build_mode_int \c_two \else: \c_six \fi:
875     }
876   }
877   { \cs_if_exist_use:cF { regex_build_c_#3: } }
878   {
879     \msg_kernel_error:nnx { regex } { c-missing-category } {#3}
880     #2 #3
881   }
882 }

```

(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`.

```

883 \cs_new_protected:cpn { regex_build_c_ \c_lbrace_str : }
884 {
885   \group_begin:
886   \l_regex_build_mode_int
887   = - \if_case:w \l_regex_build_mode_int \c_two \else: \c_six \fi:
888   \regex_build:w
889 }

```

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

\regex_build_c[: When encountering \c[, the task is to collect uppercase letters representing character categories.

```

\regex_build_c_lbrack_loop:NN
\regex_build_c_lbrack_end:
\regex_build_add:N
890 \cs_new_protected:cpn { regex_build_c[: } #1#2
891 {
892   \int_zero:N \l_regex_catcodes_int
893   \str_if_eq:nnTF { #1 #2 } { \regex_build_special:N ^ }
894   {
895     \bool_set_false:N \l_regex_catcodes_bool
896     \regex_build_c_lbrack_loop:NN
897   }
898   {
899     \bool_set_true:N \l_regex_catcodes_bool
900     \regex_build_c_lbrack_loop:NN
901     #1#2
902   }
903 }
904 \cs_new_protected:Npn \regex_build_c_lbrack_loop:NN #1#2
905 {
906   \token_if_eq_meaning:NNTF #1 \regex_build_raw:N
907   {
908     \cs_if_exist:CTF { c_regex_catcode_#2_int }
909     {
910       \exp_args:Nc \regex_build_c_lbrack_add:N
911       { c_regex_catcode_#2_int }
912       \regex_build_c_lbrack_loop:NN
913     }
914   }
915   { % [
916     \token_if_eq_charcode:NNTF #2 ]
917     { \regex_build_c_lbrack_end: }
918   }
919   {
920     \msg_kernel_error:nxx { regex } { c-missing-rbrack } {#2}
921     \regex_build_c_lbrack_end:
922   }
923 }
924 \cs_new_protected_nopar:Npn \regex_build_c_lbrack_end:
925 {
926   \l_regex_build_mode_int
927   = \if_case:w \l_regex_build_mode_int \c_two \else: \c_six \fi:
928   \if_meaning:w \c_false_bool \l_regex_catcodes_bool
929   \int_set:Nn \l_regex_catcodes_int
930   { \c_regex_catcodes_all_int - \l_regex_catcodes_int }
931   \fi:
932 }
933 \cs_new_protected:Npn \regex_build_c_lbrack_add:N #1
934 {
935   \if_int_odd:w \int_eval:w \l_regex_catcodes_int / #1 \int_eval_end:
936   \else:

```

```

937     \tex_advance:D \l_regex_catcodes_int #1
938     \fi:
939 }

```

(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. In that case, make sure that any character class and group is closed, anchor the match at the end, add the code for making the match successful, and expand the contents of the regular expression we just built. Otherwise, replace the brace with an escaped brace.

```

940 \cs_new_protected:cpn { regex_build_ \c_rbrace_str : }
941 {
942     \int_compare:nNnTF \l_regex_build_mode_int < \c_zero
943     {
944         \regex_build_exit_class:
945         \regex_build_exit_groups:
946         \regex_build_escaped:N Z
947         \regex_build_end:
948         \use:x
949         {
950             \group_end:
951             \regex_build_one:n
952             {
953                 \regex_item_cs:n
954                 { \regex_set_aux:N \l_regex_internal_regex }
955             }
956         }
957     }
958     { \exp_after:wN \regex_build_raw:N \c_rbrace_str }
959 }

```

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

2.4.9 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.

```

960 \cs_new_protected:Npn \regex_build_quantifier:w #1#2
961 {
962     \token_if_eq_meaning:NNTF #1 \regex_build_special:N
963     {
964         \cs_if_exist_use:cF { regex_build_quantifier_#2:w }
965         {
966             \regex_build_quantifier_end:nn { } { }
967             #1 #2
968         }
969     }
970     {
971         \regex_build_quantifier_end:nn { } { }
972         #1 #2

```

```

973     }
974 }
(End definition for \regex_build_quantifier:w. This function is documented on page ??.)

```

\regex_build_quantifier?:w For each “basic” quantifier, ?, *, +, feed the correct arguments to \regex_build_quantifier_aux:nnNN.

```

\regex_build_quantifier*:w
\regex_build_quantifier+:w
975 \cs_new_protected_nopar:cpn { regex_build_quantifier?:w }
976   { \regex_build_quantifier_aux:nnNN { } { ? } }
977 \cs_new_protected_nopar:cpn { regex_build_quantifier*:w }
978   { \regex_build_quantifier_aux:nnNN { } { * } }
979 \cs_new_protected_nopar:cpn { regex_build_quantifier+:w }
980   { \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.

```

981 \cs_new_protected:Npn \regex_build_quantifier_aux:nnNN #1#2#3#4
982 {
983   \str_if_eq:nnTF { #3 #4 } { \regex_build_special:N ? }
984     { \regex_build_quantifier_end:nn { #2 #4 } {#1} }
985     {
986       \regex_build_quantifier_end:nn { #2 } {#1}
987       #3 #4
988     }
989 }
(End definition for \regex_build_quantifier_aux:nnNN. This function is documented on page ??.)

```

\regex_build_quantifier_{:w Three possible syntaxes: {<int>}, {<int>}, or {<int>,<int>}.

```

\regex_build_quantifier_lbrace:n
\regex_build_quantifier_lbrace:nw
\regex_build_quantifier_lbrace:nnw
990 \cs_new_protected_nopar:cpn { regex_build_quantifier_ \c_lbrace_str :w }
991   { \regex_get_digits:nw { \regex_build_quantifier_lbrace:n } }
992 \cs_new_protected:Npn \regex_build_quantifier_lbrace:n #1
993 {
994   \tl_if_empty:nTF {#1}
995     {
996       \regex_build_quantifier_end:nn { } { }
997       \exp_after:wN \regex_build_raw:N \c_lbrace_str
998     }
999     { \regex_build_quantifier_lbrace:nw {#1} }
1000 }
1001 \cs_new_protected:Npx \regex_build_quantifier_lbrace:nw #1#2#3
1002 {
1003   \exp_not:N \prg_case_str:nnn { #2 #3 }
1004   {
1005     { \exp_not:N \regex_build_special:N , }
1006     {
1007       \exp_not:N \regex_get_digits:nw
1008       { \exp_not:N \regex_build_quantifier_lbrace:nnw {#1} }

```

```

1009     }
1010     { \exp_not:N \regex_build_special:N \c_rbrace_str }
1011     { \exp_not:N \regex_build_quantifier_end:nn {n} { {#1} } }
1012   }
1013   {
1014     \exp_not:N \regex_build_quantifier_end:nn { } { }
1015     \exp_not:N \regex_build_raw:N \c_lbrace_str #1#2
1016   }
1017 }
1018 \cs_new_protected:Npn \regex_build_quantifier_lbrace:nnw #1#2#3#4
1019 {
1020   \str_if_eq:xxTF
1021   { \exp_not:n {#3#4} }
1022   { \exp_not:N \regex_build_special:N \c_rbrace_str }
1023   {
1024     \tl_if_empty:nTF {#2}
1025     { \regex_build_quantifier_aux:nnNN { {#1} } { n* } }
1026     { \regex_build_quantifier_aux:nnNN { {#1} {#2} } { nn } }
1027   }
1028   {
1029     \regex_build_quantifier_end:nn { } { }
1030     \use:x
1031     {
1032       \exp_args:No \tl_map_function:nN
1033       { \c_lbrace_str #1 , #2 }
1034       \regex_build_raw:N
1035     }
1036     #3 #4
1037   }
1038 }

```

(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.

```

1039 \cs_new_protected:Npn \regex_build_quantifier_end:nn #1#2
1040 {
1041   \use:c { regex_build_ \l_regex_one_or_group_tl _ #1 : } #2
1042   \tl_clear:N \l_regex_tests_tl
1043   \bool_set_true:N \l_regex_tests_bool
1044 }

```

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

2.4.10 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.

```

1045 \cs_new_protected_nopar:Npn \regex_build_one_quantifier:
1046 {

```

```

1047 \tl_set:Nx \l_regex_one_or_group_tl { one }
1048 \regex_build_quantifier:w
1049 }

```

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

\regex_build_one_n: This function is called in case the syntax is $\{\langle int \rangle\}$. Greedy and lazy operators are identical, since the number of repetitions is fixed. Build one state for each repetition, with a transition controlled by the tests that we have collected. The case where we find no quantifier is the special case $n = 1$.

```

1050 \cs_new_protected:Npn \regex_build_one_n: #1
1051 {
1052   \prg_replicate:nn {#1}
1053   {
1054     \regex_build_transition:NN
1055     \regex_tests_action_cost:n \l_regex_right_state_int
1056   }
1057 }
1058 \cs_new_eq:cN { regex_build_one_n?: } \regex_build_one_n:
1059 \cs_new_protected_nopar:Npn \regex_build_one_:
1060 { \regex_build_one_n: \c_one }

```

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

\regex_build_one_n*: Those functions are called in case the syntax is $\{\langle int \rangle, \}$. The $*$ and $+$ quantifiers (greedy or lazy) are special cases for $n = 0$ or $n = 1$. In the case $n = 0$, build a costly transition going from the current state to itself, and a free transition moving to a new state. In the case $n \geq 1$, reuse \regex_build_one_n: to match n repetition, then add free transitions from the last state to the previous one, and from the last state to a new one. The greedy and lazy versions only differ in the order of transitions.

```

1061 \cs_new_protected:cpn { regex_build_one_n*: } #1
1062 {
1063   \int_compare:nNnTF {#1} = \c_zero
1064   {
1065     \regex_build_transitions:NNNN
1066     \regex_tests_action_cost:n \l_regex_left_state_int
1067     \regex_action_free:n \l_regex_right_state_int
1068   }
1069   {
1070     \regex_build_one_n: {#1}
1071     \int_set_eq:NN \l_regex_internal_a_int \l_regex_left_state_int
1072     \regex_build_transitions:NNNN
1073     \regex_action_free:n \l_regex_internal_a_int
1074     \regex_action_free:n \l_regex_right_state_int
1075   }
1076 }
1077 \cs_new_protected:cpn { regex_build_one_n*?: } #1
1078 {
1079   \int_compare:nNnTF {#1} = \c_zero
1080   {

```

```

1081     \regex_build_transitions:NNNN
1082     \regex_action_free:n      \l_regex_right_state_int
1083     \regex_tests_action_cost:n \l_regex_left_state_int
1084 }
1085 {
1086     \regex_build_one_n: {#1}
1087     \int_set_eq:NN \l_regex_internal_a_int \l_regex_left_state_int
1088     \regex_build_transitions:NNNN
1089     \regex_action_free:n \l_regex_right_state_int
1090     \regex_action_free:n \l_regex_internal_a_int
1091 }
1092 }
1093 \cs_new_protected_nopar:cpx { regex_build_one_*: }
1094 { \exp_not:c { regex_build_one_n*: } \c_zero }
1095 \cs_new_protected_nopar:cpx { regex_build_one_*?: }
1096 { \exp_not:c { regex_build_one_n*?: } \c_zero }
1097 \cs_new_protected_nopar:cpx { regex_build_one_+: }
1098 { \exp_not:c { regex_build_one_n*: } \c_one }
1099 \cs_new_protected_nopar:cpx { regex_build_one_+?: }
1100 { \exp_not:c { regex_build_one_n*?: } \c_one }

```

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

\regex_build_one_nn: Those functions are called when the syntax is $\{ \langle int \rangle, \langle int \rangle \}$, or for the ? quantifier,
 \regex_build_one_nn?: with $m = 0$ and $n = 1$.
 \regex_build_one_?:
 \regex_build_one_??:
 \regex_build_one_nn_aux:nnnn

```

1101 \cs_new_protected_nopar:Npn \regex_build_one_nn:
1102 {
1103     \regex_build_one_nn_aux:nnnn { }
1104     {
1105         \regex_build_transitions:NNNN
1106         \regex_tests_action_cost:n \l_regex_right_state_int
1107         \regex_action_free:n      \l_regex_internal_a_int
1108     }
1109 }
1110 \cs_new_protected_nopar:cpn { regex_build_one_nn?: }
1111 {
1112     \regex_build_one_nn_aux:nnnn {?}
1113     {
1114         \regex_build_transitions:NNNN
1115         \regex_action_free:n      \l_regex_internal_a_int
1116         \regex_tests_action_cost:n \l_regex_right_state_int
1117     }
1118 }
1119 \cs_new_protected:Npn \regex_build_one_nn_aux:nnnn #1#2#3#4
1120 {
1121     \regex_build_one_n: {#3}
1122     \int_compare:nNnTF {#3} > {#4}
1123     {
1124         \msg_kernel_error:nnxxx
1125         { regex } { quantifier-backwards } {#3} {#4} {#1}

```



```

1126     }
1127     {
1128         \int_set:Nn \l_regex_internal_a_int
1129         { \l_regex_right_state_int + #4 - #3 }
1130         \prg_replicate:nn { #4 - #3 } { #2 }
1131     }
1132 }
1133 \cs_new_protected_nopar:cpn { regex_build_one?: }
1134 { \regex_build_one_nn: \c_zero \c_one }
1135 \cs_new_protected_nopar:cpx { regex_build_one_?: }
1136 { \exp_not:c { regex_build_one_nn?: } \c_zero \c_one }

```

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

2.4.11 Groups and alternation

We support the syntax $\langle expr_1 \rangle \dots \langle expr_n \rangle$ for alternations.

<pre> \regex_build_(\regex_build_) \regex_build_open_aux: \regex_build_ \regex_build_begin_alternation: \regex_build_end_alternation: </pre>	<p>Grouping and alternation go together.</p> <ul style="list-style-type: none"> • 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.
--	---

```

1137 \cs_new_protected:cpn { regex_build_( } #1#2
1138 {
1139     \regex_build_if_in_class:TF
1140     {
1141         \regex_build_raw:N (
1142         #1 #2
1143     }
1144     {
1145         \str_if_eq:nnTF { #1 #2 } { \regex_build_special:N ? }
1146         { \regex_build_special_group:NN }
1147         {
1148             \int_incr:N \l_regex_capturing_group_int
1149             \regex_seq_push_int:NN
1150             \l_regex_capturing_group_seq \l_regex_capturing_group_int
1151             \regex_build_open_aux:
1152             #1 #2
1153         }
1154     }
1155 }
1156 \cs_new_protected_nopar:Npn \regex_build_open_aux:
1157 {
1158     \regex_build_new_state:

```

```

1159 \regex_seq_push_int:NN \l_regex_left_state_seq \l_regex_left_state_int
1160 \regex_seq_push_int:NN \l_regex_right_state_seq \l_regex_right_state_int
1161 \bool_if:NTF \l_regex_caseless_bool
1162 { \seq_push:Nn \l_regex_end_group_seq \regex_build_caseless: }
1163 { \seq_push:Nn \l_regex_end_group_seq \regex_build_caseful: }
1164 \seq_push:Nn \l_regex_end_alteration_seq { }
1165 \regex_build_begin_alteration:
1166 }
1167 \cs_new_protected_nopar:cpn { regex_build_|: }
1168 {
1169   \regex_build_if_in_class:TF { \regex_build_raw:N | }
1170   {
1171     \regex_build_end_alteration:
1172     \regex_build_begin_alteration:
1173   }
1174 }
1175 \cs_new_protected_nopar:cpn { regex_build_): }
1176 {
1177   \regex_build_if_in_class:TF { \regex_build_raw:N ) }
1178   {
1179     \seq_if_empty:NTF \l_regex_capturing_group_seq
1180     { \msg_kernel_error:nn { regex } { extra-rparen } }
1181     {
1182       \regex_build_close_aux:
1183       \regex_build_group_quantifier:
1184     }
1185   }
1186 }
1187 \cs_new_protected_nopar:Npn \regex_build_close_aux:
1188 {
1189   \regex_build_end_alteration:
1190   \regex_seq_pop_int:NN \l_regex_left_state_seq \l_regex_left_state_int
1191   \regex_seq_pop_int:NN \l_regex_right_state_seq \l_regex_right_state_int
1192   \regex_seq_pop_use:N \l_regex_end_group_seq
1193   \seq_pop:NN \l_regex_end_alteration_seq \l_regex_internal_a_tl
1194 }

```

Building each branch.

```

1195 \cs_new_protected_nopar:Npn \regex_build_begin_alteration:
1196 {
1197   \regex_build_new_state:
1198   \regex_seq_get_int:NN \l_regex_left_state_seq \l_regex_left_state_int
1199   \regex_toks_put_right:Nx \l_regex_left_state_int
1200   {
1201     \regex_action_free:n
1202     {
1203       \int_eval:n
1204       { \l_regex_right_state_int - \l_regex_left_state_int }
1205     }
1206   }

```

```

1207 }
1208 \cs_new_protected_nopar:Npn \regex_build_end_alteration:
1209 {
1210   \int_set_eq:NN \l_regex_left_state_int \l_regex_right_state_int
1211   \regex_seq_get_int:NN \l_regex_right_state_seq \l_regex_right_state_int
1212   \regex_toks_put_right:Nx \l_regex_left_state_int
1213   {
1214     \regex_action_free:n
1215     {
1216       \int_eval:n
1217       { \l_regex_right_state_int - \l_regex_left_state_int }
1218     }
1219   }
1220   \regex_seq_get_use:N \l_regex_end_alteration_seq
1221 }

```

(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.

```

1222 \cs_new_protected:Npn \regex_build_special_group:NN #1#2
1223 {
1224   \cs_if_exist_use:cF { regex_build_special_group\_token_to_str:N #2 : }
1225   {
1226     \msg_kernel_error:nxx { regex } { special-group-unknown }
1227     { (? \token_to_str:N #2 } %)
1228     \regex_build_special:N ( % )
1229     \regex_build_raw:N ?
1230     #1 #2
1231   }
1232 }

```

(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.

```

1233 \cs_new_protected_nopar:cpn { regex_build_special_group:: }
1234 {
1235   \regex_seq_push_int:NN \l_regex_capturing_group_seq \c_minus_one
1236   \regex_build_open_aux:
1237 }

```

(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.

```

1238 \cs_new_protected_nopar:cpn { regex_build_special_group_|: }
1239 {
1240   \regex_seq_push_int:NN \l_regex_capturing_group_seq \c_minus_one
1241   \regex_build_new_state:
1242   \regex_seq_push_int:NN \l_regex_left_state_seq \l_regex_left_state_int
1243   \regex_seq_push_int:NN \l_regex_right_state_seq \l_regex_right_state_int
1244   \seq_push:Nx \l_regex_end_alternation_seq
1245   {
1246     \exp_not:N \int_compare:nNnT
1247       \l_regex_capturing_group_int
1248       > \l_regex_capturing_group_max_int
1249     {
1250       \int_set_eq:NN
1251         \l_regex_capturing_group_max_int
1252         \l_regex_capturing_group_int
1253     }
1254     \int_set:Nn \l_regex_capturing_group_int
1255       { \int_use:N \l_regex_capturing_group_int }
1256   }
1257   \seq_push:Nx \l_regex_end_group_seq
1258   {
1259     \bool_if:NTF \l_regex_caseless_bool
1260       \regex_build_caseless:
1261       \regex_build_caseful:
1262     \int_set_eq:NN
1263       \l_regex_capturing_group_int
1264       \l_regex_capturing_group_max_int
1265   }
1266   \regex_build_begin_alternation:
1267 }

```

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

\regex_build_special_group_i: The match can be made case-insensitive by setting the option with (?i).
 \regex_build_special_group_-: 1268 \cs_new_protected_nopar:Npn \regex_build_special_group_i:
 \regex_build_options:NNN 1269 {
 \regex_build_option_+i: 1270 \regex_build_options:NNN +
 \regex_build_option_-i: 1271 \regex_build_raw:N i
 1272 }

```

1273 \cs_new_protected_nopar:cpn { regex_build_special_group_-: }
1274 {
1275   \regex_build_options:NNN -
1276 }
1277 \cs_new_protected:Npn \regex_build_options:NNN #1#2#3
1278 {
1279   \token_if_eq_meaning:NNTF \regex_build_raw:N #2
1280   {
1281     \cs_if_exist_use:cF { regex_build_option_#1#3: }
1282     { \msg_kernel_error:nnx { regex } { unknown-option } { #3 } }
1283     \regex_build_options:NNN #1
1284   }

```

```

1285     {
1286         \prg_case_str:nnn { #3 }
1287         { % (
1288             { ) } { }
1289             { - } { \regex_build_options:NNN - }
1290         }
1291         { \msg_kernel_error:nxx { regex } { invalid-in-option } { #3 } }
1292     }
1293 }
1294 \cs_new_protected_nopar:cpn { regex_build_option_+i: }
1295 {
1296     \regex_build_caseless:
1297     \cs_set_eq:NN \regex_match_loop_case_hook:
1298     \regex_match_loop_caseless_hook:
1299 }
1300 \cs_new_protected_nopar:cpn { regex_build_option_-i: }
1301 { \regex_build_caseful: }

```

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

2.4.12 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.

```

1302 \cs_new_protected_nopar:Npn \regex_build_group_quantifier:
1303 {
1304     \tl_set:Nn \l_regex_one_or_group_tl { group }
1305     \regex_build_quantifier:w
1306 }

```

(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.

```

1307 \cs_new_protected:Npn \regex_build_group_submatches:NN #1#2
1308 {
1309     \seq_pop:NN \l_regex_capturing_group_seq \l_regex_internal_a_tl
1310     \int_compare:nNnF { \l_regex_internal_a_tl } < \c_zero
1311     {
1312         \regex_toks_put_left:Nx #1
1313         { \regex_action_submatch:n { \l_regex_internal_a_tl < } }
1314         \regex_toks_put_left:Nx #2
1315         { \regex_action_submatch:n { \l_regex_internal_a_tl > } }
1316     }
1317 }

```

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

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_internal_a_int to a new state.

`\regex_build_group_qs_aux:NN` 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_internal_a_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. We shift the left state to avoid interactions with `action_submatch`.

`\regex_build_group_?:`

`\regex_build_group_??:`

```

1318 \cs_new_protected_nopar:Npn \regex_build_group_qs_aux:
1319 {
1320   \int_set_eq:NN \l_regex_internal_a_int \l_regex_left_state_int
1321   \regex_build_new_state:
1322   \tex_toks:D \l_regex_right_state_int = \tex_toks:D \l_regex_internal_a_int
1323   \regex_toks_put_left:Nx \l_regex_right_state_int
1324   {
1325     \int_sub:Nn \l_regex_current_state_int
1326     {
1327       \int_eval:n % ^^A here we lie!
1328       { \l_regex_right_state_int - \l_regex_internal_a_int }
1329     }
1330   }
1331   \use:x
1332   {
1333     \tex_toks:D \l_regex_internal_a_int
1334     {
1335       \s_stop
1336       \regex_action_free:n
1337       {
1338         \int_eval:n
1339         { \l_regex_right_state_int - \l_regex_internal_a_int }
1340       }
1341     }
1342   }
1343   \regex_build_group_submatches:NN
1344   \l_regex_right_state_int \l_regex_left_state_int
1345 }
1346 \cs_new_protected:Npn \regex_build_group_qs_aux:NN #1#2
1347 {
1348   \regex_build_group_qs_aux:
1349   \int_set_eq:NN \l_regex_right_state_int \l_regex_left_state_int
1350   \regex_build_transition:NN \regex_action_free:n #1
1351   #2 \l_regex_internal_a_int
1352   {
1353     \regex_action_free:n
1354     { \int_eval:n { \l_regex_right_state_int - \l_regex_internal_a_int } }
1355   }
1356 }
1357 \cs_new_protected_nopar:cpn { regex_build_group?: }
1358 {
1359   \regex_build_group_qs_aux:NN

```

```

1360     \l_regex_right_state_int \regex_toks_put_right:Nx
1361   }
1362   \cs_new_protected_nopar:cpn { regex_build_group_??: }
1363   {
1364     \regex_build_group_qs_aux:NN
1365     \l_regex_right_state_int \regex_toks_put_left:Nx
1366   }

```

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

\regex_build_group_n_aux:n
\regex_build_group_transition_right_max:

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_internal_a_int points to the last copy of the initial left-most state, \l_regex_left/right_state_int have their initial values. Furthermore, \l_regex_max_state_int is set appropriately to the largest allocated \toks register plus 1.

```

1367   \cs_new_protected:Npn \regex_build_group_transition_right_max:
1368   {
1369     \regex_toks_put_right:Nx \l_regex_right_state_int
1370     {
1371       \regex_action_free:n
1372       { \int_eval:n { \l_regex_max_state_int - \l_regex_right_state_int } }
1373     }
1374   }
1375   \cs_new_protected:Npn \regex_build_group_n_aux:n #1
1376   {
1377     \int_set_eq:NN \l_regex_internal_a_int \l_regex_left_state_int
1378     \int_set_eq:NN \l_regex_internal_b_int \l_regex_max_state_int
1379     \int_add:Nn \l_regex_right_state_int
1380     { ( #1 - 1 ) * ( \l_regex_max_state_int - \l_regex_left_state_int ) }
1381     \int_set:Nn \l_regex_max_state_int
1382     {
1383       \l_regex_left_state_int
1384       + ( #1 ) * ( \l_regex_max_state_int - \l_regex_left_state_int )
1385     }
1386     \int_while_do:nNnn \l_regex_internal_b_int < \l_regex_max_state_int
1387     {
1388       \tex_toks:D \l_regex_internal_b_int
1389       = \tex_toks:D \l_regex_internal_a_int
1390       \int_incr:N \l_regex_internal_a_int
1391       \int_incr:N \l_regex_internal_b_int
1392     }
1393   }

```

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

\regex_build_group_n:
\regex_build_group_n?:
\regex_build_group_:

These functions are called in case the syntax is $\{ \langle int \rangle \}$, or in the absence of quantifier (with $n = 1$). Greedy and lazy operators are identical, since the number of repetitions is fixed. We only record the submatch information at the last repetition; and of course, in the case $n = 0$, we record nothing.

```

1394 \cs_new_protected:Npn \regex_build_group_n: #1
1395 {
1396   \regex_build_group_transition_right_max:
1397   \regex_build_group_n_aux:n {#1}
1398   \regex_build_new_state:
1399   \int_compare:nNnF {#1} = \c_zero
1400   {
1401     \regex_build_group_submatches:NN
1402     \l_regex_internal_a_int \l_regex_left_state_int
1403   }
1404 }
1405 \cs_new_eq:cN { regex_build_group_n?: } \regex_build_group_n:
1406 \cs_new_protected_nopar:Npn \regex_build_group_:
1407 { \regex_build_group_n: \c_one }

```

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

\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.

```

\regex_build_group_*:
\regex_build_group_*?:
\regex_build_group_+:
\regex_build_group_+?:
\regex_build_group_n_star_aux:NNn
1408 \cs_new_protected_nopar:cpn { regex_build_group_n*: }
1409 {
1410   \regex_build_group_n_star_aux:NNn
1411   \regex_toks_put_right:Nx \regex_toks_put_left:Nx
1412 }
1413 \cs_new_protected_nopar:cpn { regex_build_group_n*?: }
1414 {
1415   \regex_build_group_n_star_aux:NNn
1416   \regex_toks_put_left:Nx \regex_toks_put_right:Nx
1417 }
1418 \cs_new_protected:Npn \regex_build_group_n_star_aux:NNn #1#2#3
1419 {
1420   \int_compare:nNnTF {#3} = \c_zero
1421   { \regex_build_group_qs_aux:NN \l_regex_internal_a_int #1 }
1422   {
1423     \regex_build_group_transition_right_max:
1424     \regex_build_group_n_aux:n {#3}
1425     \regex_build_new_state:
1426     #2 \l_regex_left_state_int
1427     {
1428       \regex_action_free:n
1429       {
1430         \int_eval:n
1431         { \l_regex_internal_a_int - \l_regex_left_state_int }
1432       }
1433     }
1434     \regex_build_group_submatches:NN
1435     \l_regex_internal_a_int \l_regex_left_state_int

```



```

1436     }
1437   }
1438   \cs_new_protected_nopar:cpx { regex_build_group_*: }
1439   { \exp_not:c { regex_build_group_n*: } \c_zero }
1440   \cs_new_protected_nopar:cpx { regex_build_group_*?: }
1441   { \exp_not:c { regex_build_group_n*?: } \c_zero }
1442   \cs_new_protected_nopar:cpx { regex_build_group_+: }
1443   { \exp_not:c { regex_build_group_n*: } \c_one }
1444   \cs_new_protected_nopar:cpx { regex_build_group_+?: }
1445   { \exp_not:c { regex_build_group_n*?: } \c_one }

```

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

\regex_build_group_nn: These functions are called when the syntax is either {<int>}, or {<int>,<int>}.

```

\regex_build_group_nn?: 1446 \cs_new_protected_nopar:Npn \regex_build_group_nn:
  \regex_build_group_nn_aux:nnnn 1447 {
    \regex_build_group_nn_aux:nnnn { } 1448
    \regex_build_group_transition_right_max: 1449
  } 1450
  \cs_new_protected_nopar:cpn { regex_build_group_nn?: } 1451
  { 1452
    \regex_build_group_nn_aux:nnnn { } 1453
    { 1454
      \use:x 1455
      { 1456
        \tex_toks:D \l_regex_right_state_int 1457
        { 1458
          \exp_after:wN \regex_build_group_nn_aux_ii:w 1459
          \tex_the:D \tex_toks:D \l_regex_right_state_int 1460
        } 1461
      } 1462
    } 1463
  } 1464
  \cs_new_protected:Npn \regex_build_group_nn_aux:nnnn #1#2#3#4 1465
  { 1466
    \int_compare:nNnTF {#3} > {#4} 1467
    { 1468
      \msg_kernel_error:nnxxx 1469
      { regex } { quantifier-backwards } {#3} {#4} {#1} 1470
    } 1471
    { 1472
      \int_compare:nNnTF {#3} = \c_zero 1473
      { 1474
        \regex_build_group_qs_aux: 1475
        \int_set_eq:NN \l_regex_right_state_int \l_regex_left_state_int 1476
        \int_set_eq:NN \l_regex_left_state_int \l_regex_internal_a_int 1477
        \regex_build_group_transition_right_max: 1478
        \regex_build_group_n_aux:n { #4 - #3 } 1479
        \int_set_eq:NN \l_regex_internal_c_int \l_regex_right_state_int 1480
        \int_set_eq:NN \l_regex_right_state_int \l_regex_left_state_int 1481
      }
    }
  }

```

```

1482         \prg_replicate:nn { #4 - #3 }
1483         {
1484             #2
1485             \int_add:Nn \l_regex_right_state_int
1486             { \l_regex_internal_b_int - \l_regex_internal_a_int }
1487         }
1488         \int_set_eq:NN \l_regex_right_state_int \l_regex_internal_c_int
1489         \int_set_eq:NN \l_regex_left_state_int \l_regex_internal_a_int
1490         \regex_build_new_state:
1491     }
1492     {
1493         \regex_build_group_transition_right_max:
1494         \regex_build_group_n_aux:n {#3}
1495         \int_set_eq:NN \l_regex_left_state_int \l_regex_internal_a_int
1496         \regex_build_group_submatches:NN
1497         \l_regex_left_state_int \l_regex_right_state_int
1498         \regex_build_group_n_aux:n { #4 - #3 + \c_one }
1499         \int_sub:Nn \l_regex_right_state_int
1500         { \l_regex_internal_a_int - \l_regex_left_state_int }
1501         \prg_replicate:nn { #4 - #3 }
1502         {
1503             #2
1504             \int_add:Nn \l_regex_right_state_int
1505             { \l_regex_internal_b_int - \l_regex_internal_a_int }
1506         }
1507         \int_set_eq:NN \l_regex_left_state_int \l_regex_internal_a_int
1508         \regex_build_new_state:
1509     }
1510 }
1511 }
1512 \cs_new_protected:Npn \regex_build_group_nn_aux_ii:w #1 \regex_action_free:n
1513 {
1514     #1
1515     \regex_action_free:n
1516     {
1517         \int_eval:n
1518         { \l_regex_max_state_int - \l_regex_right_state_int }
1519     }
1520     \regex_action_free:n
1521 }

```

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

2.4.13 Matching raw token lists with `\u`

`\regex_build_u:` The `\u` escape is invalid in classes nad directly following a catcode test. Otherwise, it
`\regex_build_u_loop:NN` must be followed by a left brace. We then collect the characters for the argument of `\u` within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace is missing, then we will reach the end-markers

of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace.

```

1522 \cs_new_protected:cpn { regex_build_/u: } #1#2
1523 {
1524   \regex_build_if_assertions_forbidden:TF
1525   { \regex_build_raw_error:N u #1#2 }
1526   {
1527     \str_if_eq:xxTF {#1#2} { \regex_build_special:N \c_lbrace_str }
1528     {
1529       \tl_set:Nx \l_regex_internal_a_tl { \if_false: } \fi:
1530       \regex_build_u_loop:NN
1531     }
1532     {
1533       \msg_kernel_error:nn { regex } { u-missing-lbrace }
1534       #1#2
1535     }
1536   }
1537 }
1538 \cs_new:Npn \regex_build_u_loop:NN #1#2
1539 {
1540   \token_if_eq_meaning:NNTF #1 \regex_build_raw:N
1541   { #2 \regex_build_u_loop:NN }
1542   {
1543     \token_if_eq_meaning:NNTF #1 \regex_build_special:N
1544     {
1545       \exp_after:wN \token_if_eq_charcode:NNTF \c_rbrace_str #2
1546       { \if_false: { \fi: } \regex_build_u_end: }
1547       { #2 \regex_build_u_loop:NN }
1548     }
1549     {
1550       \if_false: { \fi: }
1551       \msg_kernel_error:nnx { regex } { u-missing-rbrace } {#2}
1552       \regex_build_u_end:
1553       #1#2
1554     }
1555   }
1556 }

```

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

\regex_build_u_end: Once we have extracted the variable's name, we store the contents of that variable in `\l_regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in this case, the variable is turned to a string), or not. Afterwards, clean up the value of `\l_regex_tests_tl`, but not `\l_regex_tests_bool`, because this is guaranteed to be true at the start, and is not affected by anything we do here.

```

1557 \cs_new_protected:Npn \regex_build_u_end:
1558 {
1559   \tl_set:Nv \l_regex_internal_a_tl \l_regex_internal_a_tl
1560   \if_num:w \l_regex_build_mode_int = \c_zero
1561     \regex_build_u_not_cs:

```

```

1562     \else:
1563         \regex_build_u_in_cs:
1564     \fi:
1565     \tl_clear:N \l_regex_tests_tl
1566 }

```

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

\regex_build_u_in_cs: When \u appears within a control sequence, the variable is converted to a string, and we add to the NFA one state per character. We use \regex_build_one_: directly, as this avoids having to search for a quantifier.

```

1567 \cs_new_protected:Npn \regex_build_u_in_cs:
1568 {
1569     \exp_args:NNo \str_gset_other:Nn \g_regex_internal_tl
1570     { \l_regex_internal_a_tl }
1571     \tl_map_inline:Nn \g_regex_internal_tl
1572     {
1573         \tl_set:Nx \l_regex_tests_tl { \regex_item_equal:n { ##1 } }
1574         \regex_build_one_:
1575     }
1576 }

```

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

\regex_build_u_not_cs: In mode 0, the \u escape adds one state to the NFA for each token in \l_regex_internal_a_tl. The tests corresponding to a given *<token>* are: first test the category, then, if the *<token>* is a control sequence, compare the string representations, otherwise compare the character codes. We could compare the string representations in all cases, but that doesn't interact well with caseless matching.

```

1577 \cs_new_protected:Npn \regex_build_u_not_cs:
1578 {
1579     \exp_args:No \tl_analysis_map_inline:nm { \l_regex_internal_a_tl }
1580     {
1581         \tl_set:Nx \l_regex_tests_tl
1582         {
1583             \exp_not:N \int_compare:nNnT
1584             \l_regex_current_catcode_int = { \int_value:w "##2 }
1585             {
1586                 \int_compare:nNnTF { "##2 } = \c_zero
1587                 {
1588                     \exp_not:n
1589                     { \str_if_eq:xxT { \l_regex_current_token_tl } {##1} }
1590                     { \regex_break_true:w }
1591                 }
1592                 { \regex_item_equal:n {##3} }
1593             }
1594         }
1595         \regex_build_one_:
1596     }
1597 }

```

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

2.5 Matching

At every index, 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.

2.5.1 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_active_int`, and converting the integer to a dimension (suitable for a `\skip` assignment) in scaled points.

```
1598 \cs_new_protected:Npn \regex_store_state:n #1
1599 {
1600   \int_incr:N \l_regex_max_active_int
1601   \tex_skip:D \l_regex_max_active_int #1 sp \scan_stop:
1602   \regex_store_submatches:n {#1}
1603 }
```

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

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

```

1604 \cs_new_protected_nopar:Npn \regex_state_use_with_submatches:
1605   { \regex_state_use_aux:n { } }
1606 \cs_new_protected_nopar:Npn \regex_state_use:
1607   { \regex_state_use_aux:n { \exp_after:wN \use_none_delimit_by_s_stop:w } }
1608 \cs_new_protected:Npn \regex_state_use_aux:n #1
1609   {
1610     \if_num:w \tex_dimen:D \l_regex_current_state_int < \l_regex_step_int
1611       \tex_dimen:D \l_regex_current_state_int
1612       = \l_regex_step_int sp \scan_stop:
1613       #1 \tex_the:D \tex_toks:D \exp_after:wN \l_regex_current_state_int
1614     \fi:
1615     \scan_stop:
1616   }

```

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

2.5.2 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.

```

1617 \cs_new_protected_nopar:Npn \regex_disable_submatches:
1618   {
1619     \cs_set_eq:NN \regex_state_use_with_submatches: \regex_state_use:
1620     \cs_set_eq:NN \regex_store_submatches:n
1621     \regex_protected_use_none:n
1622     \cs_set_eq:NN \regex_action_submatches:n
1623     \regex_protected_use_none:n
1624   }
1625 \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` The submatch information pertaining to one given thread is moved from state to state
`\regex_store_submatches_aux:w` as we execute the NFA. We make sure that most of the `\toks` register is not read before
`\regex_store_submatches_aux_ii:Nnnw` being assigned again to that same register.

```

1626 \cs_new_protected:Npn \regex_store_submatches:n #1
1627   {
1628     \tex_toks:D #1 \exp_after:wN
1629     {
1630       \tex_romannumeral:D
1631       \exp_after:wN \regex_store_submatches_aux:w
1632       \tex_the:D \tex_toks:D #1

```

```

1633     }
1634 }
1635 \cs_new_protected:Npn \regex_store_submatches_aux:w #1 \s_stop
1636 {
1637     \regex_store_submatches_aux_ii:Nnnw
1638     #1
1639     \regex_state_submatches:nn \c_minus_one \q_prop
1640     \s_stop
1641 }
1642 \cs_new_protected:Npn \regex_store_submatches_aux_ii:Nnnw
1643     \regex_state_submatches:nn #1 #2 #3 \s_stop
1644 {
1645     \exp_after:wN \c_zero
1646     \exp_after:wN \regex_state_submatches:nn \exp_after:wN
1647     { \int_value:w \int_eval:w \l_regex_step_int + \c_one \exp_after:wN }
1648     \exp_after:wN { \l_regex_current_submatches_prop }
1649     \regex_state_submatches:nn {#1} {#2}
1650     \s_stop
1651 }

```

(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.

```

1652 \cs_new_protected:Npn \regex_state_submatches:nn #1#2
1653 {
1654     \if_num:w #1 = \l_regex_step_int
1655     \tl_set:Nn \l_regex_current_submatches_prop { #2 }
1656     \fi:
1657 }

```

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

2.5.3 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).

```

1658 \cs_new_protected:Npn \regex_match:n #1
1659 {
1660     \int_zero:N \l_regex_nesting_int
1661     \int_set_eq:NN \l_regex_current_index_int \l_regex_max_state_int
1662     \regex_query_set:nnn { } { -1 } { -2 }
1663     \int_set_eq:NN \l_regex_min_index_int \l_regex_current_index_int
1664     \tl_analysis_map_inline:nn {#1}
1665     {
1666         \regex_query_set:nnn {##1} {"##2"} {##3}
1667         \if_case:w "##2 \exp_stop_f:
1668         \or: \int_incr:N \l_regex_nesting_int
1669         \or: \int_decr:N \l_regex_nesting_int
1670         \fi:

```

```

1671     }
1672     \int_set_eq:NN \l_regex_max_index_int \l_regex_current_index_int
1673     \regex_query_set:nnn { } { -1 } { -2 }
1674     \regex_match_initial_setup:
1675     \regex_match_once:
1676   }
1677   \cs_new_protected:Npn \regex_query_set:nnn #1#2#3
1678   {
1679     \tex_muskip:D \l_regex_current_index_int
1680     = \etex_glutomu:D
1681       #3 sp
1682     plus #2 sp
1683     minus \l_regex_nesting_int sp
1684     \scan_stop:
1685     \tex_toks:D \l_regex_current_index_int {#1}
1686     \int_incr:N \l_regex_current_index_int
1687   }
1688   \cs_new_protected_nopar:Npn \regex_query_get:
1689   {
1690     \tl_set:Nx \l_regex_current_token_tl
1691     { \tex_the:D \tex_toks:D \l_regex_current_index_int }
1692     \l_regex_current_char_int
1693     = \etex_mutoglue:D \tex_muskip:D \l_regex_current_index_int
1694     \l_regex_current_catcode_int = \etex_gluestretch:D
1695     \etex_mutoglue:D \tex_muskip:D \l_regex_current_index_int
1696   }

```

(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.

```

1697   \cs_new_protected_nopar:Npn \regex_match_once:
1698   {
1699     \regex_match_setup:
1700     \regex_query_get:
1701     \regex_match_loop:
1702     \prg_break_point:n { }
1703     \bool_if:NT \l_regex_success_match_bool
1704     {
1705       \bool_gset_true:N \g_regex_success_bool
1706       \l_regex_every_match_tl
1707     }
1708   }

```

(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.

```

1709   \cs_new_protected_nopar:Npn \regex_match_initial_setup:

```



```

1710 {
1711   \int_set_eq:NN \l_regex_step_int \c_minus_one
1712   \prg_stepwise_inline:nnnn
1713   \c_zero \c_one { \l_regex_max_state_int - \c_one }
1714   { \tex_dimen:D ##1 \l_regex_step_int sp }
1715   \int_set:Nn \l_regex_start_index_int
1716   { \l_regex_min_index_int - \c_one }
1717   \int_set_eq:NN \l_regex_current_index_int \l_regex_min_index_int
1718   \int_set_eq:NN \l_regex_success_index_int \l_regex_min_index_int
1719   \int_set_eq:NN \l_regex_submatch_int \l_regex_max_state_int
1720   \bool_set_false:N \l_regex_success_empty_bool
1721   \bool_gset_false:N \g_regex_success_bool
1722 }

```

(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.

```

1723 \cs_new_protected_nopar:Npn \regex_match_setup:
1724 {
1725   \prop_clear:N \l_regex_current_submatches_prop
1726   \bool_if:NTF \l_regex_success_empty_bool
1727   { \cs_set_eq:NN \regex_if_two_empty_matches:F \regex_if_match_empty:F }
1728   { \cs_set_eq:NN \regex_if_two_empty_matches:F \use:n }
1729   \int_set_eq:NN \l_regex_start_index_int \l_regex_success_index_int
1730   \int_set:Nn \l_regex_current_index_int
1731   { \l_regex_start_index_int - \c_one }
1732   \bool_set_false:N \l_regex_success_match_bool
1733   \int_zero:N \l_regex_max_active_int
1734   \regex_store_state:n {0} %^A _state_int!
1735 }

```

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

\regex_match_loop: 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 \regex_match_one_active:n 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.

```

1736 \cs_new_protected_nopar:Npn \regex_match_loop:
1737 {
1738   \int_incr:N \l_regex_current_index_int
1739   \int_incr:N \l_regex_step_int
1740   \bool_set_false:N \l_regex_fresh_thread_bool
1741   \int_set_eq:NN \l_regex_last_char_int \l_regex_current_char_int
1742   \regex_query_get:
1743   \regex_match_loop_case_hook:
1744   \use:x
1745   {
1746     \int_zero:N \l_regex_max_active_int
1747     \regex_match_one_active:w 1 ; \prg_break_point:n { }

```

```

1748     \exp_not:N \prg_break_point:n { }
1749   }
1750   \if_num:w \l_regex_current_char_int < \c_minus_one
1751     \exp_after:wN \prg_map_break:
1752   \fi:
1753   \if_num:w \l_regex_max_active_int = \c_zero
1754     \exp_after:wN \prg_map_break:
1755   \fi:
1756   \regex_match_loop:
1757 }
1758 \cs_new:Npn \regex_match_one_active:w #1;
1759 {
1760   \if_num:w #1 > \l_regex_max_active_int
1761     \exp_after:wN \prg_map_break:
1762   \fi:
1763   \regex_match_one_active_aux:n
1764     { \int_value:w \tex_skip:D #1 }
1765   \exp_after:wN \regex_match_one_active:w
1766   \int_use:N \int_eval:w #1 + \c_one ;
1767 }
1768 \cs_new_protected:Npn \regex_match_one_active_aux:n #1
1769 {
1770   \int_set:Nn \l_regex_current_state_int {#1}
1771   \prop_clear:N \l_regex_current_submatches_prop
1772   \regex_state_use_with_submatches:
1773 }

```

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

`\regex_match_loop_case_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.

```

1774 \cs_new_protected_nopar:Npn \regex_match_loop_case_hook: { }
1775 \cs_new_protected_nopar:Npn \regex_match_loop_caseless_hook:
1776 {
1777   \int_set_eq:NN \l_regex_case_changed_char_int \l_regex_current_char_int
1778   \if_num:w \l_regex_current_char_int < \c_ninety_one
1779     \if_num:w \l_regex_current_char_int < \c_sixty_five
1780     \else:
1781       \int_add:Nn \l_regex_case_changed_char_int { \c_thirty_two }
1782     \fi:
1783   \else:
1784     \if_num:w \l_regex_current_char_int < \c_one_hundred_twenty_three
1785     \if_num:w \l_regex_current_char_int < \c_ninety_seven
1786     \else:
1787       \int_sub:Nn \l_regex_case_changed_char_int { \c_thirty_two }
1788     \fi:
1789   \fi:
1790 }
1791 }

```

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

2.5.4 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`.

```

1792 \cs_new_protected_nopar:Npn \regex_action_start_wildcard:
1793 {
1794   \bool_set_true:N \l_regex_fresh_thread_bool
1795   \regex_action_free:n {1}
1796   \bool_set_false:N \l_regex_fresh_thread_bool
1797   \regex_action_cost:n {0}
1798 }

```

(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.

```

1799 \cs_new_protected:Npn \regex_action_cost:n #1
1800 {
1801   \exp_args:Nf \regex_store_state:n
1802   { \int_eval:n { \l_regex_current_state_int + #1 } }
1803 }

```

(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.

```

1804 \cs_new_protected_nopar:Npn \regex_action_success:
1805 {
1806   \regex_if_two_empty_matches:F
1807   {
1808     \bool_set_true:N \l_regex_success_match_bool
1809     \bool_set_eq:NN \l_regex_success_empty_bool
1810     \l_regex_fresh_thread_bool
1811     \int_set_eq:NN \l_regex_success_index_int \l_regex_current_index_int
1812     \prop_set_eq:NN \l_regex_success_submatches_prop
1813     \l_regex_current_submatches_prop
1814     \prg_map_break:
1815   }
1816 }

```

(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.

```

1817 \cs_new_protected:Npn \regex_action_free:n #1
1818 {
1819   \use:x
1820   {
1821     \int_add:Nn \l_regex_current_state_int {#1}
1822     \regex_state_use:
1823     \int_set:Nn \l_regex_current_state_int
1824       { \int_use:N \l_regex_current_state_int }
1825     \tl_set:Nn \exp_not:N \l_regex_current_submatches_prop
1826       { \exp_not:o \l_regex_current_submatches_prop }
1827   }
1828 }

```

(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 index.

```

1829 \cs_new_protected:Npn \regex_action_submatch:n #1
1830 {
1831   \prop_put:Nno \l_regex_current_submatches_prop {#1}
1832   { \int_use:N \l_regex_current_index_int }
1833 }

```

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

2.5.5 ??

`\regex_if_match_empty:T`

`\regex_if_match_empty:F`

```

1834 \cs_new:Npn \regex_if_match_empty:T
1835 { \int_compare:nNnT \l_regex_start_index_int = \l_regex_current_index_int }
1836 \cs_new:Npn \regex_if_match_empty:F
1837 { \int_compare:nNnF \l_regex_start_index_int = \l_regex_current_index_int }

```

(End definition for \regex_if_match_empty:T and \regex_if_match_empty:F. These functions are documented on page ??.)

2.6 Replacement

`\regex_submatch_nesting_aux:n`

```

1838 \cs_new_protected:Npn \regex_submatch_nesting_aux:n #1
1839 {
1840   + \etex_glueshrink:D \etex_mutogluue:D \etex_muexpr:D
1841     \tex_muskip:D \etex_gluestretch:D \tex_skip:D #1
1842   - \tex_muskip:D \tex_skip:D #1
1843   \scan_stop:
1844 }

```

(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 `\regex_escape_use:nnnn` with three protected arguments. This inserts in the input stream a token list of the form $\langle fn\ 1 \rangle \langle char\ 1 \rangle \dots \langle fn\ N \rangle \langle char\ N \rangle$, where $\langle fn\ i \rangle$ is one of the three functions, and $\langle char\ i \rangle$ a character in the string #1.

```

1845 \cs_new:Npn \regex_nesting:n #1 { } %^^A move. Rename?
1846 \tl_new:N \l_regex_nesting_tl
1847 \cs_new_protected:Npn \regex_replacement:n #1
1848 {
1849   \int_zero:N \l_regex_replacement_int
1850   \int_zero:N \l_regex_nesting_int
1851   \tl_clear:N \l_regex_nesting_tl
1852   \regex_escape_use:nnnn
1853   { \regex_replacement_unescaped:N ##1 }
1854   { \regex_replacement_escaped:N ##1 }
1855   { \regex_replacement_raw:N ##1 }
1856   {#1}
1857   \prg_do_nothing: \prg_do_nothing:
1858   \cs_set:Npx \regex_nesting:n ##1
1859   {
1860     + \int_use:N \l_regex_nesting_int
1861     \l_regex_nesting_tl
1862     - \regex_submatch_nesting_aux:n {##1}
1863   }
1864   \use:x
1865   {
1866     \exp_not:n { \cs_set:Npn \regex_replacement_tl:n ##1 }
1867     { \regex_toks_range:nn \c_zero \l_regex_replacement_int }
1868   }
1869   % ^^A rename!
1870 }

```

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

`\regex_replacement_raw:N`

```

1871 \cs_new_protected:Npn \regex_replacement_raw:N #1
1872 { \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.

```

1873 \cs_new_protected:Npn \regex_replacement_put:n #1
1874 {
1875   \tex_toks:D \l_regex_replacement_int {#1}
1876   \int_incr:N \l_regex_replacement_int
1877 }

```

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

`\regex_replacement_escaped:N`

```

\regex_replacement_submatch:w 1878 \cs_new_protected:Npn \regex_replacement_unescaped:N #1
\regex_replacement_submatch_aux:nN 1879 {

```

```

1880 \if_charcode:w \c_rbrace_str #1
1881 \if_num:w \l_regex_replacement_csnames_int > \c_zero
1882 \regex_replacement_put:n \cs_end:
1883 \else:
1884 \regex_replacement_put:n #1
1885 \fi:
1886 \else:
1887 \regex_replacement_put:n #1
1888 \fi:
1889 }
1890 \cs_new_protected:Npn \regex_replacement_escaped:N #1
1891 {
1892 \cs_if_exist_use:cF { regex_replacement_#1:w }
1893 {
1894 \if_num:w \c_one < 1#1 \exp_stop_f:
1895 \regex_replacement_put_submatch:n {#1}
1896 \else:
1897 \regex_replacement_put:n #1
1898 \fi:
1899 }
1900 }
1901 \cs_new_protected:Npn \regex_replacement_put_submatch:n #1
1902 {
1903 \regex_replacement_put:n
1904 { \regex_query_submatch:nn {#1} {##1} }
1905 \if_num:w \l_regex_replacement_csnames_int = \c_zero
1906 \tl_put_right:Nn \l_regex_nesting_tl
1907 {
1908 \exp_not:N \if_num:w #1 < \l_regex_capturing_group_int
1909 \regex_submatch_nesting_aux:n { \int_eval:w #1+##1 \int_eval_end: }
1910 \exp_not:N \fi:
1911 }
1912 \fi:
1913 }
1914 \cs_new_protected:Npn \regex_replacement_error:NNN #1#2#3
1915 {
1916 \msg_kernel_error:nxxx { regex } { #1-command }
1917 { replacement-text } {#3}
1918 #2 #3
1919 }
1920 \cs_new_protected:Npn \regex_replacement_g:w #1#2
1921 {
1922 \str_if_eq:xxTF
1923 { \exp_not:n { #1#2 } }
1924 { \regex_replacement_unescaped:N \c_lbrace_str }
1925 {
1926 \int_zero:N \l_regex_internal_a_int
1927 \regex_replacement_g_digits:NN
1928 }
1929 { \regex_replacement_error:NNN g #1 #2 }

```

```

1930 }
1931 \cs_new_protected:Npn \regex_replacement_g_digits:NN #1#2
1932 {
1933   \token_if_eq_meaning:NNTF #1 \regex_replacement_unescaped:N
1934   {
1935     \if_num:w \c_one < 1#2 \exp_stop_f:
1936     \int_set:Nn \l_regex_internal_a_int { \c_ten * \l_regex_internal_a_int + #2 }
1937     \exp_after:wN \use_i:nnn
1938     \exp_after:wN \regex_replacement_g_digits:NN
1939   \else:
1940     \if_charcode:w \c_rbrace_str #2
1941     \exp_args:No \regex_replacement_put_submatch:n
1942     { \int_use:N \l_regex_internal_a_int }
1943     \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
1944   \else:
1945     \exp_after:wN \exp_after:wN
1946     \exp_after:wN \regex_replacement_error:NNN
1947     \exp_after:wN \exp_after:wN \exp_after:wN g
1948   \fi:
1949   \fi:
1950 }
1951 { \regex_replacement_error:NNN g }
1952 #1 #2
1953 }
1954 \cs_new_protected:Npn \regex_replacement_u:w #1#2
1955 {
1956   \str_if_eq:xxTF { #1#2 } { \regex_replacement_unescaped:N \c_lbrace_str }
1957   {
1958     \int_incr:N \l_regex_replacement_csnames_int
1959     \regex_replacement_put:n
1960     { \exp_not:n { \exp_after:wN \exp_not:V \cs:w } }
1961   }
1962   { \regex_replacement_error:NNN u #1#2 }
1963 }
1964 \cs_new_protected:Npn \regex_replacement_c:w #1#2
1965 {
1966   \token_if_eq_meaning:NNTF #1 \regex_replacement_unescaped:N
1967   {
1968     \cs_if_exist_use:cF { regex_replacement_c_#2:w }
1969     { \regex_replacement_error:NNN c #1#2 }
1970   }
1971   { \regex_replacement_error:NNN c #1#2 }
1972 }
1973 \cs_new_protected_nopar:cpn { regex_replacement_c_ \c_lbrace_str :w }
1974 {
1975   \int_incr:N \l_regex_replacement_csnames_int
1976   \regex_replacement_put:n
1977   { \exp_not:n { \exp_after:wN \regex_replacement_exp_not:N \cs:w } }
1978 }
1979 \cs_new:Npn \regex_replacement_exp_not:N #1 { \exp_not:n {#1} }

```

```

1980 \group_begin:
1981
1982 \char_set_catcode_math_superscript:N \^^@
1983 \cs_new_protected_nopar:Npn \regex_replacement_c_U:w
1984 { \regex_replacement_char:nNN { ^^@ } }
1985
1986 \char_set_catcode_alignment:N \^^@
1987 \cs_new_protected_nopar:Npn \regex_replacement_c_T:w
1988 { \regex_replacement_char:nNN { ^^@ } }
1989
1990 \cs_new_protected:Npn \regex_replacement_c_S:w #1#2
1991 {
1992   \int_compare:nNnTF { '#2 } = \c_zero
1993   { \regex_replacement_error:NNN c #1#2 }
1994   {
1995     \char_set_lccode:nn {32} { '#2 }
1996     \tl_to_lowercase:n { \regex_replacement_put:n {~} }
1997   }
1998 }
1999
2000 \char_set_catcode_parameter:N \^^@
2001 \cs_new_protected_nopar:Npn \regex_replacement_c_P:w
2002 { \regex_replacement_char:nNN { ^^@^^@^^@^^@^^@^^@^^@ } }
2003
2004 \char_set_catcode_other:N \^^@
2005 \cs_new_protected_nopar:Npn \regex_replacement_c_O:w
2006 { \regex_replacement_char:nNN { ^^@ } }
2007
2008 \char_set_catcode_math_toggle:N \^^@
2009 \cs_new_protected_nopar:Npn \regex_replacement_c_M:w
2010 { \regex_replacement_char:nNN { ^^@ } }
2011
2012 \char_set_catcode_letter:N \^^@
2013 \cs_new_protected_nopar:Npn \regex_replacement_c_L:w
2014 { \regex_replacement_char:nNN { ^^@ } }
2015
2016 \char_set_catcode_group_end:N \^^@
2017 \cs_new_protected_nopar:Npn \regex_replacement_c_E:w
2018 {
2019   \int_decr:N \l_regex_nesting_int
2020   \regex_replacement_char:nNN { \if_false: { \fi: ^^@ }
2021 }
2022
2023 \char_set_catcode_math_subscript:N \^^@
2024 \cs_new_protected_nopar:Npn \regex_replacement_c_D:w
2025 { \regex_replacement_char:nNN { ^^@ } }
2026
2027 \char_set_catcode_group_begin:N \^^@
2028 \cs_new_protected_nopar:Npn \regex_replacement_c_B:w
2029 {

```



```

2030     \int_incr:N \l_regex_nesting_int
2031     \regex_replacement_char:nNN { \exp_after:wN ^^@ \if_false: } \fi: }
2032   }
2033
2034   \char_set_catcode_active:N \^^@
2035   \cs_new_protected_nopar:Npn \regex_replacement_c_A:w
2036     { \regex_replacement_char:nNN { \exp_not:N ^^@ } }
2037
2038   \group_end:
2039
2040   \cs_new_protected:Npn \regex_replacement_char:nNN #1#2#3
2041     {
2042       \char_set_lccode:nn \c_zero { '#3 }
2043       \tl_to_lowercase:n
2044         { \regex_replacement_put:n { \exp_not:n {#1} } }
2045     }

```

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

2.7 User commands

2.7.1 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 $\langle nfa\ var \rangle$ should only be accessed through dedicated functions. This rather drastic approach is taken because assignments triggered by the contents of $\langle nfa\ var \rangle$ may overwrite data which is used elsewhere, unless everything is done carefully in a group.

<pre> \regex_gset:Nn \regex_const:Nn \regex_set:Nn \regex_set_aux:NNn </pre>	<p>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 <code>\toks</code> registers in the user's variable.</p>
--	--

```

2046 \cs_new_protected_nopar:Npn \regex_set:Nn
2047   { \regex_set_aux:NNn \tl_set:Nn }
2048 \cs_new_protected_nopar:Npn \regex_gset:Nn
2049   { \regex_set_aux:NNn \tl_gset:Nn }
2050 \cs_new_protected_nopar:Npn \regex_const:Nn

```

```

2051 { \regex_set_aux:NNn \tl_const:Nn }
2052 \cs_new_protected:Npn \regex_set_aux:NNn #1#2#3
2053 {
2054   \group_begin:
2055     \regex_build:n {#3}
2056     \use:x
2057     {
2058       \group_end:
2059       #1 \exp_not:N #2 { \regex_set_aux:N #2 }
2060     }
2061 }

```

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

\regex_set_aux:N 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 **\regex_set_aux:n** 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.

```

2062 \cs_new:Npn \regex_set_aux:N #1
2063 {
2064   \exp_not:n { \regex_nfa:Nw #1 }
2065   \l_regex_max_state_int
2066   = \int_use:N \l_regex_max_state_int
2067   \l_regex_capturing_group_int
2068   = \int_use:N \l_regex_capturing_group_int
2069   \token_if_eq_meaning:NNT
2070   \regex_match_loop_case_hook:
2071   \regex_match_loop_caseless_hook:
2072   {
2073     \cs_set_eq:NN \regex_match_loop_case_hook:
2074     \regex_match_loop_caseless_hook:
2075   }
2076   \prg_stepwise_function:nnnN
2077   \c_zero \c_one {\l_regex_max_state_int - \c_one }
2078   \regex_set_aux:n
2079   \s_stop
2080 }
2081 \cs_new:Npn \regex_set_aux:n #1
2082 { \tex_toks:D #1 { \tex_the:D \tex_toks:D #1 } }
2083 \cs_new:Npn \regex_nfa:Nw #1
2084 {
2085   \msg_expandable_kernel_error:nnn { regex } { nfa-misused } {#1}
2086   \use_none_delimit_by_s_stop:w
2087 }

```

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

\regex_use:N No error-checking.

```

2088 \cs_new_protected_nopar:Npn \regex_use:N
2089 { \exp_last_unbraced:No \use_none:nn }

```

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

2.7.2 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.

```
2090 \cs_new_protected_nopar:Npn \regex_aux_return:
2091 {
2092   \if_meaning:w \c_true_bool \g_regex_success_bool
2093   \prg_return_true:
2094   \else:
2095     \prg_return_false:
2096   \fi:
2097 }
```

(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 $\langle regex \rangle$ 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.

```
2098 \cs_new_protected:Npn \regex_aux_build_match:nn #1#2
2099 {
2100   \regex_build:n {#1}
2101   \regex_match:n {#2}
2102 }
```

(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 $\langle regex \rangle$ 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.

```
2103 \cs_new_protected:Npn \regex_aux_use_match:Nn #1#2
2104 {
2105   \exp_args:No \tl_if_head_eq_meaning:nNTF {#1} \regex_nfa:Nw
2106   {
2107     \regex_use:N #1
2108     \regex_match:n {#2}
2109   }
2110   {
2111     \msg_kernel_error:nnx { regex } { not-nfa } { \token_to_str:N #1 }
2112     \bool_gset_false:N \g_regex_success_bool
2113   }
2114 }
```

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

`\regex_extract_once: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_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_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.

```

2115 \cs_set_protected:Npn \regex_tmp:w #1#2#3
2116 {
2117   \cs_new_protected_nopar:Npn #1
2118     { #3 \regex_aux_build_match:nn }
2119   \cs_new_protected_nopar:Npn #2
2120     { #3 \regex_aux_use_match:Nn }
2121   \prg_new_protected_conditional:Npnn #1 ##1##2##3 { T , F , TF }
2122     {
2123       #3 \regex_aux_build_match:nn {##1} {##2} ##3
2124       \regex_aux_return:
2125     }
2126   \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
2127     {
2128       #3 \regex_aux_use_match:Nn {##1} {##2} ##3
2129       \regex_aux_return:
2130     }
2131 }
2132 \tl_map_inline:nn
2133 {
2134   { extract_once } { extract_all }
2135   { replace_once } { replace_all }
2136   { split }
2137 }
2138 {
2139   \exp_args:Nccc \regex_tmp:w
2140     { regex_#1:nnN } { regex_#1:NnN } { regex_#1_aux:NnnN }
2141 }

```

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

2.7.3 Submatches, once the correct match is found

```

\regex_extract:
\regex_extract_aux_b:wn 2142 \cs_new_protected_nopar:Npn \regex_extract:
\regex_extract_aux_e:wn 2143 {
2144   \int_set_eq:NN \l_regex_submatch_start_int \l_regex_submatch_int
2145   \if_meaning:w \c_true_bool \g_regex_success_bool
2146     \prg_replicate:nn \l_regex_capturing_group_int
2147     {
2148       \tex_skip:D \l_regex_submatch_int \c_zero sp \scan_stop:
2149       \int_incr:N \l_regex_submatch_int
2150     }
2151   \prop_map_inline:Nn \l_regex_success_submatches_prop
2152     {
2153       \if_num:w ##1 \c_max_int
2154       \exp_after:wN \regex_extract_aux_b:wn \int_use:N

```

```

2155         \else:
2156             \exp_after:wN \regex_extract_aux_e:wn \int_use:N
2157             \fi:
2158             \int_eval:w \l_regex_submatch_start_int + ##1 {##2}
2159         }
2160     \fi:
2161 }
2162 \cs_new_protected:Npn \regex_extract_aux_b:wn #1 < #2
2163 {
2164     \tex_skip:D #1 = #2 sp
2165     plus \etex_gluestretch:D \tex_skip:D #1 \scan_stop:
2166 }
2167 \cs_new_protected:Npn \regex_extract_aux_e:wn #1 > #2
2168 {
2169     \tex_skip:D #1
2170     = 1 \tex_skip:D #1 plus #2 sp \scan_stop:
2171 }

```

(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
2172 \cs_new_protected:Npn \regex_group_end_extract_seq:N #1
2173 {
2174     \cs_set_eq:NN \seq_item:n \scan_stop:
2175     \flag_clear:n { regex_begin }
2176     \flag_clear:n { regex_end }
2177     \tl_set:Nx \l_regex_internal_a_tl
2178     {
2179         \prg_stepwise_function:nnnN
2180         \l_regex_max_state_int
2181         \c_one
2182         { \l_regex_submatch_int - \c_one }
2183         \regex_extract_seq_aux:n
2184     }
2185     \int_compare:nNnF
2186     { \flag_height:n { regex_begin } + \flag_height:n { regex_end } }
2187     = \c_zero
2188     { \msg_kernel_error:nn { regex } { sequence-unbalanced } }
2189     \tl_set:Nx \l_regex_internal_a_tl { \l_regex_internal_a_tl }
2190     \exp_args:NNNo \group_end:
2191     \tl_set:Nn #1 \l_regex_internal_a_tl
2192 }
2193 \cs_new:Npn \regex_extract_seq_aux:n #1
2194 {
2195     \seq_item:n
2196     {
2197         \exp_after:wN \regex_extract_seq_aux:ww
2198         \int_value:w \regex_submatch_nesting_aux:n {#1} ; #1;
2199     }
2200 }
2201 \cs_new:Npn \regex_extract_seq_aux:ww #1; #2;

```

```

2202 {
2203   \if_num:w #1 < \c_zero
2204     \flag_raise:n { regex_end }
2205     \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
2206   \fi:
2207   \regex_query_submatch:w #2;
2208   \if_num:w #1 > \c_zero
2209     \flag_raise:n { regex_begin }
2210     \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
2211   \fi:
2212 }

```

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

2.7.4 Matching

\regex_match:nn 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.

```

2213 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
2214 {
2215   \regex_match_aux:n
2216   { \regex_aux_build_match:nn {#1} {#2} }
2217 }
2218 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
2219 {
2220   \regex_match_aux:n
2221   { \regex_aux_use_match:Nn #1 {#2} }
2222 }
2223 \cs_new_protected:Npn \regex_match_aux:n #1
2224 {
2225   \group_begin:
2226   \tl_clear:N \l_regex_every_match_tl
2227   \regex_disable_submatches:
2228   #1
2229   \group_end:
2230   \regex_aux_return:
2231 }

```

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

\regex_count:nnN 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.

```

2232 \cs_new_protected_nopar:Npn \regex_count:nnN
2233 { \regex_count_aux:NnnN \regex_aux_build_match:nn }
2234 \cs_new_protected_nopar:Npn \regex_count:NnN
2235 { \regex_count_aux:NnnN \regex_aux_use_match:Nn }
2236 \cs_new_protected:Npn \regex_count_aux:NnnN #1#2#3#4
2237 {
2238   \group_begin:

```

```

2239 \regex_disable_submatches:
2240 \int_zero:N \l_regex_match_count_int
2241 \tl_set:Nn \l_regex_every_match_tl
2242 {
2243   \int_incr:N \l_regex_match_count_int
2244   \regex_match_once:
2245 }
2246 #1 {#2} {#3}
2247 \exp_args:NNNo
2248 \group_end:
2249 \int_set:Nn #4 { \int_use:N \l_regex_match_count_int }
2250 }

```

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

2.7.5 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.

```

2251 \cs_new_protected:Npn \regex_extract_once_aux:NnnN #1#2#3#4
2252 {
2253   \group_begin:
2254   \tl_set:Nn \l_regex_every_match_tl { \regex_extract: }
2255   #1 {#2} {#3}
2256   \regex_group_end_extract_seq:N #4
2257 }

```

(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.

```

2258 \cs_new_protected:Npn \regex_extract_all_aux:NnnN #1#2#3#4
2259 {
2260   \group_begin:
2261   \tl_set:Nn \l_regex_every_match_tl
2262   { \regex_extract: \regex_match_once: }
2263   #1 {#2} {#3}
2264   \regex_group_end_extract_seq:N #4
2265 }

```

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

2.7.6 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.

```

2266 \cs_new_protected:Npn \regex_split_aux:NnnN #1#2#3#4
2267 {
2268   \group_begin:
2269   \tl_set:Nn \l_regex_every_match_tl
2270   {
2271     \if_num:w \l_regex_start_index_int < \l_regex_success_index_int
2272     \regex_extract:
2273     \tex_skip:D \l_regex_submatch_start_int
2274     = \l_regex_start_index_int sp
2275     plus \tex_skip:D \l_regex_submatch_start_int \scan_stop:
2276     \fi:
2277     \regex_match_once:
2278   }
2279   #1 {#2} {#3}
2280   \tex_skip:D \l_regex_submatch_int
2281   = \l_regex_start_index_int sp
2282   plus \l_regex_current_index_int sp \scan_stop:
2283   \int_incr:N \l_regex_submatch_int
2284   \if_num:w \l_regex_start_index_int = \l_regex_current_index_int
2285   \if_meaning:w \c_true_bool \l_regex_success_empty_bool
2286   \int_decr:N \l_regex_submatch_int
2287   \fi:
2288   \fi:
2289   \regex_group_end_extract_seq:N #4
2290 }

```

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

2.7.7 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.

```

2291 \cs_new_protected:Npn \regex_replace_once_aux:NnnN #1#2#3#4
2292 {
2293   \group_begin:
2294   \tl_clear:N \l_regex_every_match_tl
2295   \regex_replacement:n {#3}
2296   \exp_args:Nno #1 {#2} #4
2297   \if_meaning:w \c_true_bool \g_regex_success_bool
2298   \regex_extract:
2299   \int_set:Nn \l_regex_internal_a_int
2300   { \regex_nesting:n { \l_regex_submatch_start_int } }

```



```

2301 \if_num:w \l_regex_internal_a_int = \c_zero
2302 \else:
2303   \msg_kernel_error:nxx { regex } { replace-unbalanced }
2304   { \l_regex_internal_a_int }
2305 \fi:
2306 \tl_set:Nx \l_regex_internal_a_tl
2307 {
2308   \if_num:w \l_regex_internal_a_int < \c_zero
2309     \prg_replicate:nn { - \l_regex_internal_a_int }
2310     { \exp_not:n { { \if_false: } \fi: } }
2311   \fi:
2312   \regex_toks_range:nn
2313   { \l_regex_min_index_int }
2314   { \tex_skip:D \l_regex_submatch_start_int }
2315   \regex_replacement_tl:n { \l_regex_submatch_start_int }
2316   \regex_toks_range:nn
2317   { \etex_gluestretch:D \tex_skip:D \l_regex_submatch_start_int }
2318   { \l_regex_max_index_int }
2319   \if_num:w \l_regex_internal_a_int > \c_zero
2320     \prg_replicate:nn { \l_regex_internal_a_int }
2321     { \exp_not:n { \if_false: { \fi: } } }
2322   \fi:
2323 }
2324 \tl_set:Nx \l_regex_internal_a_tl { \l_regex_internal_a_tl }
2325 \exp_args:NNNo \group_end:
2326 \tl_set:Nn #4 \l_regex_internal_a_tl
2327 \else:
2328   \group_end:
2329 \fi:
2330 }

```

(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.

```

2331 \cs_new_protected:Npn \regex_replace_all_aux:NnnN #1#2#3#4
2332 {
2333   \group_begin:
2334   \tl_set:Nn \l_regex_every_match_tl
2335   {
2336     \regex_extract:
2337     \tex_skip:D \l_regex_submatch_start_int
2338     = \tex_the:D \tex_skip:D \l_regex_submatch_start_int
2339     minus \l_regex_start_index_int sp \scan_stop:
2340     \regex_match_once:
2341   }
2342   \regex_replacement:n {#3}
2343   \exp_args:Nno #1 {#2} #4
2344   \int_set:Nn \l_regex_internal_a_int

```

```

2345     {
2346     0
2347     \prg_stepwise_function:nnnN
2348     \l_regex_max_state_int
2349     \l_regex_capturing_group_int
2350     { \l_regex_submatch_int - \c_one }
2351     \regex_nesting:n
2352     }
2353     \if_num:w \l_regex_internal_a_int = \c_zero
2354     \else:
2355     \msg_kernel_error:nnx { regex } { replace-unbalanced }
2356     { \l_regex_internal_a_int }
2357     \fi:
2358     \tl_set:Nx \l_regex_internal_a_tl
2359     {
2360     \if_num:w \l_regex_internal_a_int < \c_zero
2361     \prg_replicate:nn { - \l_regex_internal_a_int }
2362     { \exp_not:n { { \if_false: } \fi: } }
2363     \fi:
2364     \prg_stepwise_function:nnnN
2365     \l_regex_max_state_int
2366     \l_regex_capturing_group_int
2367     { \l_regex_submatch_int - \c_one }
2368     \regex_replace_all_aux:n
2369     \regex_toks_range:nn
2370     \l_regex_start_index_int \l_regex_max_index_int
2371     \if_num:w \l_regex_internal_a_int > \c_zero
2372     \prg_replicate:nn { \l_regex_internal_a_int }
2373     { \exp_not:n { \if_false: { \fi: } } }
2374     \fi:
2375     }
2376     \tl_set:Nx \l_regex_internal_a_tl { \l_regex_internal_a_tl }
2377     \exp_args:NNNo \group_end:
2378     \tl_set:Nn #4 \l_regex_internal_a_tl
2379     }
2380     \cs_new:Npn \regex_replace_all_aux:n #1
2381     {
2382     \regex_toks_range:nn
2383     { \etex_glueshrink:D \tex_skip:D #1 } { \tex_skip:D #1 }
2384     \regex_replacement_tl:n {#1}
2385     }

```

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

2.8 Messages

Messages for the preparsing phase.

```

2386 \msg_kernel_new:nnnn { regex } { trailing-backslash }
2387 { Trailing-escape-character~\iow_char:N\\. }
2388 {

```

```

2389 A~regular~expression~or~its~replacement~text~ends~with~
2390 the~escape~character~\iow_char:N\\.~It~will~be~ignored.
2391 }
2392 \msg_kernel_new:nnnn { regex } { x-missing-rbrace }
2393 { Missing~closing~brace~in~\iow_char:N\\x~hexadecimal~sequence. }
2394 {
2395   You~wrote~something~like~
2396   '\iow_char:N\\x\{\int_to_hexadecimal:n{#1}'~
2397   The~closing~brace~is~missing.
2398 }
2399 \msg_kernel_new:nnnn { regex } { x-overflow }
2400 { Character~code~'#1'~too~large~in~\iow_char:N\\x~hexadecimal~sequence. }
2401 {
2402   You~wrote~something~like~
2403   '\iow_char:N\\x\{\int_to_hexadecimal:n{#1}\}'~
2404   The~character~code~'#1'~is~larger~than~\int_use:N \c_max_char_int.
2405 }

```

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

```

2406 \msg_kernel_new:nnnn { regex } { missing-rbrack }
2407 { Missing~right~bracket~inserted~in~regular~expression. }
2408 {
2409   LaTeX~was~given~a~regular~expression~where~a~character~class~
2410   was~started~with~'['~but~the~matching~']'~is~missing.
2411 }
2412 \msg_kernel_new:nnnn { regex } { missing-rparen }
2413 {
2414   Missing~right~parenthes\int_compare:nTF{#1=1}{i}{e}s~
2415   inserted~in~regular~expression.
2416 }
2417 {
2418   LaTeX~was~given~a~regular~expression~with~\int_eval:n{#1}~
2419   more~left~parenthes\int_compare:nTF{#1=1}{i}{e}s~than~right~
2420   parenthes\int_compare:nTF{#1=1}{i}{e}s.
2421 }
2422 \msg_kernel_new:nnnn { regex } { extra-rparen }
2423 { Extra~right~parenthesis~ignored~in~regular~expression. }
2424 {
2425   LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
2426   was~open.~The~parenthesis~will~be~ignored.
2427 }

```

Sometimes escaped alphanumerics are not allowed everywhere.

```

2428 \msg_kernel_new:nnnn { regex } { class-bad-escape }
2429 { Invalid~escape~\c_backslash_str #1~in~character~class. }
2430 {
2431   The~escape~sequence~\iow_char:N\\#1~may~not~appear~within~
2432   a~character~class.~For~instance,~assertions~(which~match~zero~
2433   characters)~would~not~make~sense~there.

```

```

2434 }
2435 \msg_kernel_new:nnnn { regex } { catcode-bad-escape }
2436 { Invalid-escape~\c_backslash_str #1~following~category~test. }
2437 {
2438   The~escape~sequence~\iow_char:N\|#1~may~not~appear~following~
2439   a~category~test~such~as~\iow_char:N\cL~or~
2440   \iow_char:N\c[\iow_char:N^BE].
2441 }

```

Range errors.

```

2442 \msg_kernel_new:nnnn { regex } { range-missing-end }
2443 { Invalid-end-point~for~range~'#1-#2'~in~character~class. }
2444 {
2445   The~end~point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
2446   end~point~for~a~range:~alphanumeric~characters~should~not~be~
2447   escaped,~and~non-alphanumeric~characters~should~be~escaped.
2448 }
2449 \msg_kernel_new:nnnn { regex } { range-backwards }
2450 { Range~[#1-#2]~out~of~order~in~character~class. }
2451 {
2452   In~ranges~of~characters~[x-y]~appearing~in~character~classes,~
2453   the~first~character~code~must~not~be~larger~than~the~second.~
2454   Here,~#1~has~character~code~\int_eval:n {'#1},~while~#2~has~
2455   character~code~\int_eval:n {'#2}.
2456 }

```

Errors related to \c and \u.

```

2457 \msg_kernel_new:nnnn { regex } { c-bad-mode }
2458 { Invalid-nested~\iow_char:N\c~escape~in~regular~expression. }
2459 {
2460   The~\iow_char:N\c~escape~cannot~be~used~within~
2461   a~control~sequence~test~'\iow_char:N\c{...}'.~
2462   To~combine~several~category~tests,~use~'\iow_char:N\c[...}'.
2463 }
2464 \msg_kernel_new:nnnn { regex } { c-missing-rbrace }
2465 { Missing-right~brace~inserted~for~\iow_char:N\c~escape. }
2466 {
2467   LaTeX~was~given~a~regular~expression~where~a~
2468   '\iow_char:N\c\iow_char:N\{...}'~construction~was~not~ended~
2469   with~a~closing~brace~'\iow_char:N\}'.
2470 }
2471 \msg_kernel_new:nnnn { regex } { c-missing-rbrack }
2472 { Missing-right~bracket~inserted~for~\iow_char:N\c~escape. }
2473 {
2474   A~construction~'\iow_char:N\c[...]'~appears~in~a~
2475   regular~expression,~but~the~closing~']'~is~not~present.
2476 }
2477 \msg_kernel_new:nnnn { regex } { c-missing-category }
2478 { Invalid~character~'#1'~following~\iow_char\c~escape. }
2479 {
2480   In~regular~expressions,~the~\iow_char:N\c~escape~sequence~

```

```

2481 may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
2482 capital~letter~representing~a~character~category,~namely~
2483 one~of~ABCDELMOPTU.
2484 }
2485 \msg_kernel_new:nnnn { regex } { u-missing-lbrace }
2486 { Missing~left~brace~following~\iow_char:N\\u~escape. }
2487 {
2488   The~\iow_char:N\\u~escape~sequence~must~be~followed~by~
2489   a~brace~group~with~the~name~of~the~variable~to~use.
2490 }
2491 \msg_kernel_new:nnnn { regex } { u-missing-rbrace }
2492 { Missing~right~brace~inserted~for~\iow_char:N\\u~escape. }
2493 {
2494   LaTeX~
2495   \tl_if_empty:xTF {#2}
2496     { reached~the~end~of~the~string~ }
2497     { encountered~an~escaped~alphanumeric~character '~\iow_char:N\\#2'~ }
2498     when~parsing~the~argument~of~an~'\iow_char:N\\u\iow_char:N\{...\}'~escape.
2499 }

```

In various cases, the result of a `l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

2500 \msg_kernel_new:nnnn { regex } { sequence-unbalanced }
2501 {
2502   Missing~
2503   \flag_if_raised:nTF { regex_end }
2504     {
2505       left~
2506       \flag_if_raised:nTF { regex_begin }
2507         { and~right~braces } { brace }
2508     }
2509     { right~brace }
2510   \ inserted~in~extracted~match.
2511 }
2512 {
2513   LaTeX~was~asked~to~extract~submatches~or~split~a~token~list~
2514   according~to~a~given~regular~expression,~but~some~of~the~resulting~
2515   items~were~not~balanced.
2516 }
2517 \msg_kernel_new:nnnn { regex } { replace-unbalanced }
2518 { The~result~of~a~replacement~does~not~have~balanced~braces. }
2519 {
2520   LaTeX~was~asked~to~do~some~regular~expression~replacement,~
2521   and~the~resulting~token~list~would~not~have~the~same~number~
2522   of~begin~group~and~end~group~tokens. \\ \\
2523   \\ \\ \\
2524   \prg_case_int:nnn {#1}
2525     {
2526       { -1 } { A~left~brace~was }

```

```

2527         { 1 } { A~right~brace~was }
2528     }
2529     {
2530         \int_abs:n {#1} ~
2531         \int_compare:nNnTF {#1} < \c_zero { left } { right } ~
2532         braces ~ were
2533     }
2534     \ inserted.
2535 }

Messages related to NFA variables.

2536 \msg_kernel_new:nnnn { regex } { not-nfa }
2537 { This~is~not~a~regular~expression~variable. }
2538 {
2539     LaTeX~was~expecting~‘#1’~to~be~a~regular~expression~variable.\\
2540     This~control~sequence~is~not~a~regex~variable.~It’s~current~meaning~
2541     is~\\\\
2542     \ \ \ \ \token_to_str:N #1 = \token_to_meaning:N #1 .
2543 }
2544 \msg_kernel_new:nnn { regex } { nfa-misused }
2545 { Automaton~#1~used~incorrectly. }

2546 \msg_kernel_new:nnnn { regex } { c-command }
2547 { Misused~\iow_char:N\c~or~\iow_char:N\C~command~in~a~#1. }
2548 {
2549     In~a~#1,~the~\iow_char:N\C~escape~sequence~
2550     can~be~followed~by~one~of~the~letters~ABCDELMOPTU~
2551     or~a~brace~group,~not~by~‘#2’.
2552 }
2553 \msg_kernel_new:nnnn { regex } { unknown-option }
2554 { Unknown~option~‘#1’~for~regular~expressions. }
2555 {
2556     LaTeX~came~across~something~like~‘(?#1)’~in~a~regular~expression,~
2557     but~the~option~‘#1’~is~not~known.~It~will~be~ignored.
2558 }
2559 \msg_kernel_new:nnnn { regex } { invalid-in-option }
2560 { Invalid~character~in~option~of~a~regular~expression. }
2561 {
2562     The~character~or~escape~sequence~‘#1’~is~not~defined~
2563     as~an~option~within~regular~expressions.
2564 }
2565 \msg_kernel_new:nnnn { regex } { g-command }
2566 { Missing~brace~for~the~\iow_char:N\g~construction~in~a~replacement~text. }
2567 {
2568     In~the~replacement~text~for~a~regular~expression~search,~
2569     submatches~are~represented~either~as~\iow_char:N \g{dd..d},~
2570     or~\d,~where~‘d’~are~single~digits.~Here,~a~brace~is~missing.
2571 }

2572 </package>

```