The l3regex package: regular expressions in TEX*

The LATEX3 Project†

Released 2011/10/09

1 l3regex documentation

The l3regex package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on strings of characters. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX). For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\str_set:Nn \l_my_str { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_str
```

the string variable `\l_my_str` 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+) } { \1 , } \l_my_str
```

The `\w` sequence represents any "word" character, and `*` indicates that the `\w` sequence should be repeated as many times as possible, hence matching a word in the input string. The parentheses "capture" what their contents matched in the input string, and this can be used in the replacement text as `\1` (and higher numbers if several groups are used in the regular expression).

1.1 Syntax of regular expressions

Most characters match exactly themselves. Some characters are special and must be escaped with a backslash (*e.g.*, `\*` matches an explicit 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, none of them should be escaped, because most of those escape sequences have special meanings;

*This file describes v2900, last revised 2011/10/09.

†E-mail: latex-team@latex-project.org

- non-alphanumeric printable ascii characters can always be safely escaped, and for highest portability, they should always be escaped (this avoids problems if some currently "normal" characters is given a meaning in later releases);
- 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 literal string `$%^\abc#`.

TEXhackers note: When converting the regular expression to a string, the value of the escape character is set to be a backslash.

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

Characters.

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

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

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

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

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

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

`\t` Horizontal tab (hex 09).

Character types.

`.` A single period matches any character, including newlines.[1]

`\N` A character that is not the `\n` character (hex 0A).[2]

`\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 `[\^^I-\^^M]`. Note that `\^^K` is a vertical space, but not a space, for compatibility with perl.

[1] Should that be changed?
[2] Is that right?

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

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

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

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

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

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

Character classes match exactly one character in the subject string.

`[...]` Positive character class.

`[^...]` Negative character class.

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

Quantifiers.

`?` 0 or 1, greedy.

`??` 0 or 1, lazy.

`*` 0 or more, greedy.

`*?` 0 or more, lazy.

`+` 1 or more, greedy.

`+?` 1 or more, lazy.

Not implemented yet:

`{`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.

`\B` Not a word boundary.

`^\A` Start of the subject string.[3]

[3] The multiline mode is not implemented yet, so those are currently identical.

`$\Z\z` End of the subject string.[3]

`\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.

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`, `\w`, `\W` 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: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 various submatches are accessed with `\1`, `\2`, *etc.*, and the whole match is accessed using `\0`.

For instance,

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

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

Submatches with numbers higher than 10 are accessed in the same way, namely `\10`, `\11`, *etc.* To insert in the replacement text a submatch followed by a digit, the digit must be entered using the `\x` escape sequence: for instance, to get the first submatch followed by the digit `7`, use `\1\x37`, because `7` has character code `37` (in hexadecimal).

1.3 Precompiling regular expressions

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

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

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

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

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

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`.

TEXhackers note: Precompiled regular expressions can be safely written to a file and read when the LaTeX3 syntax is active (as triggered by `\ExplSyntaxOn`).

1.4 String matching

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

\regex_match:NnTF `\regex_match:nnTF {⟨regex⟩} {⟨string⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*regular expression*⟩ matches any substring of ⟨*string*⟩. For instance,

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

leaves `TRUE FALSE` in the input stream.

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

\regex_count:NnN `\regex_count:nnN {⟨regex⟩} {⟨string⟩} ⟨int var⟩`

Sets ⟨*int var*⟩ equal to the number of times ⟨*regular expression*⟩ appears in ⟨*string*⟩. 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 string. For instance,

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

results in `\l_foo_int` taking the value 5.

1.5 Submatch extraction

\regex_extract:nnNTF \regex_extract:nnNTF {⟨*regex*⟩} {⟨*string*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

\regex_extract:NnNTF \regex_extract:nnNTF {⟨*regex*⟩} {⟨*string*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

\regex_extract:nnNTF \regex_extract:nnNTF {⟨*regex*⟩} {⟨*string*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

\regex_extract:NnNTF \regex_extract:nnNTF {⟨*regex*⟩} {⟨*string*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Finds the first match of the ⟨*regular expression*⟩ in the ⟨*string*⟩. 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 left parenthesis. The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is not altered. The testing versions return ⟨*true*⟩ if a match was found, and ⟨*false*⟩ otherwise. For instance, assume that you type

```
\regex_extract:nnNTF { ^(La)?TeX(!*)$ } { LaTeX!!! }
  \l_foo_seq { true } { false }
```

Then the regular expression (anchored at the start with `^` and at the end with `$`) will match the whole string. 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.

1.6 String splitting

\regex_split:nnN \regex_split:nnN {⟨*regular expression*⟩} ⟨*string*⟩ ⟨*seq var*⟩

\regex_split:NnN \regex_split:nnN {⟨*regular expression*⟩} ⟨*string*⟩ ⟨*seq var*⟩

Searches the ⟨*string*⟩ into a sequence of substrings, delimited by matches of the ⟨*regular expression*⟩. If the ⟨*regular expression*⟩ has capturing groups, then the substrings 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 ⟨*string*⟩ as its sole item. If the ⟨*regular expression*⟩ matches the empty string, then the ⟨*string*⟩ is split into single character substrings.

1.7 String replacement

\regex_replace_once:nnNTF \regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*str var*⟩

\regex_replace_once:NnNTF \regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*str var*⟩

\regex_replace_once:nnNTF \regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*str var*⟩

\regex_replace_once:NnNTF \regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*str var*⟩

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

\regex_replace_all:nnN \regex_replace_all:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*str var*⟩

\regex_replace_all:NnN \regex_replace_all:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*str var*⟩

Replaces all occurrences of the `\regular expression` in the ⟨*string*⟩ 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. The result is assigned locally to ⟨*str var*⟩.

1.8 Bugs, misfeatures, future work, and other possibilities

The `{}` quantifiers are not implemented.

The following need discussion.

- Newline conventions are not done. In particular, `.` should not match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Caseless matching and more generally `(*..)` and `(?..)` sequences to set some options.
- General look-ahead/behind assertions.
- Idea of `#` as a synonym of `.*?`.
- Do we need a facility for balanced groups? (That's non-regular.)

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

- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`).
- Optimize groups with no alternative.

- Optimize the use of `\prg_stepwise_...` functions.
- Implement regex matching on external files.
- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- `(?|..|..)` to reset the capturing group number at the start of each alternative.
- Todo: check space's catcode in the result of a replacement.

The following features not present in PCRE nor perl (because they can be done in other ways in those languages) might be added.

- `#` as a synonym of `.*?`, in a way similar to macro parameters.
- Facility to match balanced groups: this is non-regular, but quite useful, and it can be implemented without too much performance loss. (*cf.* callout?)

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

- Other forms of escapes, and character classes. `\cx`≃`\^^x`, `\ddd` (octal `ddd`). POSIX character classes. `\p{..}` and `\P{..}` for having/not having a Unicode property. `\X` for "extended" Unicode sequence.
- Callout with `(?C...)`.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?

The following features of PCRE or perl will not be implemented.

- Comments: TEX already has its own system for comments.
- Named subpatterns: TEX programmers have lived so far without any need for named macro parameters.
- `\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.
- 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.
- `\K` for resetting the beginning of the match: tied to backtracking.
- `\C` single byte in UTF-8 mode: well, UTF-8 mode is not implemented, and that seems like a very rarely useful feature anyways.

2 l3regex implementation

<*package>

```
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\RequirePackage{l3str}
```

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

Given a regular expression of n characters, we build a non-deterministic finite automaton (NFA) with roughly n states, which accepts precisely those strings matching that regular expression. We then run the string through the NFA, and check the return value.

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

2.1 Constants and variables

`\regex_tmp:w`
`\g_regex_tmpa_tl`
`\l_regex_tmpa_tl`
`\l_regex_tmpb_tl`
`\l_regex_tmpa_int`

Temporary variables.

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

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

2.1.1 Variables used while building

`\l_regex_pattern_str`

The ⟨*pattern*⟩ is stored in a string variable.

```
\tl_new:N  \l_regex_pattern_str
```

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

`\l_regex_max_state_int`
`\l_regex_left_state_int`
`\l_regex_right_state_int`
`\l_regex_left_state_seq`
`\l_regex_right_state_seq`

The last state that was allocated is stored in `\l_regex_max_state_int`, and `\l_regex_left/right_state_int` point to both end-points of the last group (which any quantifier would repeat). For simple strings of characters, the left and right pointers only differ by one.

```
\int_new:N  \l_regex_max_state_int
\int_new:N  \l_regex_left_state_int
\int_new:N  \l_regex_right_state_int
\seq_new:N  \l_regex_left_state_seq
\seq_new:N  \l_regex_right_state_seq
```

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

`\l_regex_capturing_group_int`
`\l_regex_capturing_group_seq`

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

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

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

`\l_regex_one_or_group_tl`

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

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

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

`l_regex_repetition_int`

Used when to distinguish threads at the same state but with different repetitions of some quantifiers.

```
\int_new:N  \l_regex_repetition_int
```

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

`\l_regex_look_behind_bool`
`\l_regex_look_behind_str`

The boolean `\l_regex_look_behind_bool` indicates whether a look-behind assertion appears within the regular expression (currently only `\b` or `\B`). When matching, we will keep track of the part of the string before the current character in `\l_regex_look_behind_str`, stored backwards.

```
\bool_new:N \l_regex_look_behind_bool
\tl_new:N \l_regex_look_behind_str
```

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

2.1.2 Character classes

`\regex_build_tmp_class:n`
`\l_regex_class_bool`
`\l_regex_class_tl`

Used when building character classes.

```
\cs_new_eq:NN \regex_build_tmp_class:n \use_none:n
\bool_new:N \l_regex_class_bool
\tl_new:N   \l_regex_class_tl
```

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

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

These constant token lists encode which characters are recognized by `\d`, `\D`, `\w`, *etc.* in regular expressions. Namely, `\d=[0-9]`, `\w=[0-9A-Z_a-z]`, `\s=[\ \^^I\^^J\^^L\^^M]`, `\h=[\ \^^I]`, `\v=[\^^J-\^^M]`, and the upper-case counterparts match anything that the lowercase does not match. The order in which the various ranges appear is optimized for usual mostly lowercase letter text.

```
\tl_const:Nn \c_regex_d_tl
  {
    \regex_item_range:nn {48} {57} % 0--9
  }
```

```
\tl_const:Nn \c_regex_D_tl
  {
    \regex_item_more:n {57} % 9
    \regex_item_range:nn {0} {47} % 0
  }
\tl_const:Nn \c_regex_h_tl
  {
    \regex_item_equal:n {32} % space
    \regex_item_equal:n {9}  % tab
  }
\tl_const:Nn \c_regex_H_tl
  {
    \regex_item_neq:n {32} % space
    \regex_item_neq:n {9}  % tab
    \regex_break_true:w
  }
\tl_const:Nn \c_regex_s_tl
  {
    \regex_item_equal:n  {32}     % space
    \regex_item_neq:n    {11}     % vtab
    \regex_item_range:nn {9} {13} % tab, lf, vtab, ff, cr
  }
\tl_const:Nn \c_regex_S_tl
  {
    \regex_item_more:n   {32}      % > space
    \regex_item_range:nn {14} {31} % tab < ... < space
    \regex_item_range:nn {0}  {8}  % < tab
    \regex_item_equal:n  {11}      % vtab
  }
\tl_const:Nn \c_regex_v_tl
  {
    \regex_item_range:nn {10} {13} % lf, vtab, ff, cr
  }
\tl_const:Nn \c_regex_V_tl
  {
    \regex_item_more:n   {13}    % cr
    \regex_item_range:nn {0} {9} % < lf
  }
\tl_const:Nn \c_regex_w_tl
  {
    \regex_item_range:nn {97} {122} % a--z
    \regex_item_range:nn {65}  {90} % A--Z
    \regex_item_range:nn {48}  {57} % 0--9
    \regex_item_equal:n  {95}       % _
  }
\tl_const:Nn \c_regex_W_tl
  {
    \regex_item_range:nn  {0} {47} % <`0
    \regex_item_range:nn {58} {64} % (`9+1)--(`A-1)
    \regex_item_range:nn {91} {94} % (`Z+1)--(`_-1)
```

```
    \regex_item_equal:n  {96}      % `
    \regex_item_more:n  {122}      % z
  }
\tl_const:Nn \c_regex_N_tl
  {
    \regex_item_neq:n {10} % lf
    \regex_break_true:w
  }
```

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

2.1.3 Variables used when matching

`\l_regex_query_str` The string that is being matched.

```
\tl_new:N \l_regex_query_str
```

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

`\l_regex_start_step_int` In the case of multiple matches, `\l_regex_start_step_int` is equal to the position where the current match attempt began.

```
\int_new:N \l_regex_start_step_int
```

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

`\l_regex_current_char_int` `\l_regex_current_step_int` The character at the current position in the string, and the current position.

```
\int_new:N \l_regex_current_char_int
\int_new:N \l_regex_current_step_int
```

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

`\l_regex_unique_step_int` This gives a unique identifier to every step in the loop over characters in the string. In the case of a single match, it is equal to `\l_regex_current_step_int`, but for multiple matches, it is not reset to the start of the next match, but incremented. A unique identifier is handy when considering whether a state of the automaton has been visited at that step or not, and when tracking submatch information.

```
\int_new:N \l_regex_unique_step_int
```

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

`\l_regex_current_state_int` For every character in the string, 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.

```
\int_new:N \l_regex_current_state_int
\prop_new:N \l_regex_current_submatches_prop
```

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

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

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

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

`\l_regex_every_match_tl` Every time a match is found, this token list is used. For single matching, the token list is set to removing the remainder of the query string. For multiple matching, the token list is set to repeat the matching.

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

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

`\regex_last_match_empty:F` This function is most often `\use:n`, unless the current regular expression can match an empty string at the current position, and has already done so at the previous match attempt. The goal is to break infinite loops.

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

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

`\l_regex_success_bool`
`\l_regex_success_step_int`
`\l_regex_success_submatches_prop`
`\l_regex_success_empty_bool`
The booleans `\l_regex_success_bool` and `\l_regex_success_empty_bool` are true if the current match attempt was successful, and if the match was empty, respectively. The other two variables hold the step number at which the match was successful and a property list of submatches.

```
\bool_new:N \l_regex_success_bool
\bool_new:N \l_regex_success_empty_bool
\int_new:N  \l_regex_success_step_int
\prop_new:N \l_regex_success_submatches_prop
```

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

`\l_regex_fresh_thread_bool` This boolean marks when the current thread has started from the beginning of the regular expression at this character, in other words, it is true if the current thread has matched an empty string so far (and we only care about this boolean when a thread succeeds, hence matching an empty string overall).

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

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

2.1.4 Variables used for user functions

`\g_regex_submatches_seq` This holds temporarily a sequence of submatches, global so that it exits the group.

```
\seq_new:N  \g_regex_submatches_seq
```

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

`\g_regex_match_count_int` The number of matches found so far is stored in `\g_regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```
\int_new:N \g_regex_match_count_int
```

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

\g_regex_split_seq The \regex_split:nnN function stores its result in that sequence variable before assigning it to the variable provided by the user.

```
\seq_new:N \g_regex_split_seq
```

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

\l_regex_replacement_tl
\g_regex_replaced_str The replacement code stores a processed version of the user's argument in \l_regex_replacement_tl. The result of the replacement is stored in \g_regex_replaced_str, global to exit the group.

```
\tl_new:N \l_regex_replacement_tl
\tl_new:N \g_regex_replaced_str
```

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

2.2 Helpers

2.2.1 Toks

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

```
\cs_new_eq:NN \s_regex_toks \scan_stop:
```

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

\regex_toks_put_left:Nx
\regex_toks_put_right:Nx During the building phase, every \toks register starts with \s_regex_toks, 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_regex_toks marker and put it back to the left of the new material.

```
\cs_new_protected:Npn \regex_toks_put_left:Nx #1#2
  {
    \cs_set_nopar:Npx \regex_tmp:w \s_regex_toks { \s_regex_toks #2 }
    \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
      { \exp_after:wN \regex_tmp:w \tex_the:D \tex_toks:D #1 }
  }
\cs_new_protected:Npn \regex_toks_put_right:Nx #1#2
  {
    \cs_set_nopar:Npx \regex_tmp:w {#2}
    \tex_toks:D #1 \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN #1 \regex_tmp:w }
  }
```

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

2.2.2 Interrupting recursions

\regex_if_tail_stop:N
\regex_if_tail_error:Nn

Test for the end of the ⟨*string*⟩, and either stop or cause an error if it is reached.

```
\cs_new_eq:NN \regex_if_tail_stop:N \quark_if_recursion_tail_stop:N
\cs_new_protected_nopar:Npn \regex_if_tail_error:Nn #1#2
  { \quark_if_recursion_tail_stop_do:Nn #1 { \regex_build_error:n {#2} } }
```

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

\regex_build_error:n

This macro is called if anything goes wrong when building the NFA corresponding to a given regular expression Negative codes are specific to the LaTeX3 implementation. Other error codes match with the PCRE codes (not all of the PCRE errors can occur, since some constructions are not supported).

```
\cs_new_protected_nopar:Npn \regex_build_error:n #1
  {
    \msg_error:nnxx { regex } { build-error } {\int_eval:n{#1}}
      {
        \prg_case_int:nnn {#1}
          {
            {-999} {File~not~found}
            {-998} {Unsupported~construct}
            {-997} {The~regular~expression~is~too~large~(32768~states).}
            {1}  {\iow_char:N\\~at~end~of~pattern }
%           {2}  {\iow_char:N\\c~at~end~of~pattern }
            {4}  {Numbers~out~of~order~in~\iow_char:N\{\iow_char\}~quantifier.}
            {6}  {Missing~terminating~\iow_char:N\]~for~character~class }
            {7}  {Invalid~escape~sequence~in~character~class}
            {8}  {Range~out~of~order~in~character~class}
            {22} {Mismatched~parentheses}
            {34} {Character~value~in~\iow_char:N\\x{...}~sequence~is~too~large}
%           {44} {Invalid~UTF-8~string}
%           {46} {Malformed~\iow_char:N\\P~or\iow_char:N\\p~sequence}
%           {47} {Unknown~property~after~\iow_char:N\\P~or\iow_char:N\\p}
%           {68} {\iow_char:N\\c~must~be~followed~by~an~ASCII~character}
          }
          { Internal~bug. }
      }
  }
\msg_new:nnn { regex } { build-error } { (error~#1): ~ #2 }
```

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

2.2.3 Testing characters

\regex_break_point:TF
\regex_break_true:w
\regex_break_false:w

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

⟨*test1*⟩ ... ⟨*test*$_n$⟩
`\regex_break_point:TF` {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\cs_new_nopar:Npn \regex_break_true:w #1 \regex_break_point:TF #2 #3 {#2}
\cs_new_nopar:Npn \regex_break_false:w #1 \regex_break_point:TF #2 #3 {#3}
\cs_new_eq:NN \regex_break_point:TF \use_ii:nn
```

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

`\regex_item_equal:n`
`\regex_item_range:nn`
`\regex_item_less:n`
`\regex_item_more:n`
`\regex_item_neq:n`

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

```
\cs_new_nopar:Npn \regex_item_equal:n #1
  {
    \if_num:w #1 = \l_regex_current_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
  }
\cs_new_nopar:Npn \regex_item_range:nn #1 #2
  {
    \reverse_if:N \if_num:w #1 > \l_regex_current_char_int
      \reverse_if:N \if_num:w #2 < \l_regex_current_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \regex_break_true:w
      \fi:
    \fi:
  }
\cs_new_nopar:Npn \regex_item_less:n #1
  {
    \if_num:w #1 > \l_regex_current_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
  }
\cs_new_nopar:Npn \regex_item_more:n #1
  {
    \if_num:w #1 < \l_regex_current_char_int
      \exp_after:wN \regex_break_true:w
    \fi:
  }
\cs_new_nopar:Npn \regex_item_neq:n #1
  {
    \if_num:w #1 = \l_regex_current_char_int
      \exp_after:wN \regex_break_false:w
    \fi:
  }
```

(*End definition for* `\regex_item_equal:n` *and* `\regex_item_range:nn`*. These functions are documented on page* **??**.)

2.2.4 Grabbing digits

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

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

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

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

2.2.5 More char testing

`\regex_aux_char_if_alphanumeric:NTF`
`\regex_aux_char_if_special:NTF`

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

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

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

```
\prg_new_conditional:Npnn \regex_aux_char_if_special:N #1 { TF }
```

```
  {
    \if_num:w `#1 < 97 \exp_stop_f:
      \if_num:w `#1 < 58 \exp_stop_f:
        \if_num:w \int_eval:w (`#1 - \c_eight)/\c_sixteen = \c_two
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      \else:
        \if_num:w \int_eval:w `#1 / 26 = \c_three
          \prg_return_false:
        \else:
          \prg_return_true:
        \fi:
      \fi:
    \else:
      \if_num:w \int_eval:w `#1 / \c_five = 25 \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \regex_aux_char_if_alphanumeric:N #1 { TF }
  {
    \if_num:w `#1 < 91 \exp_stop_f:
      \if_num:w `#1 < 65 \exp_stop_f:
        \if_num:w \c_nine < 1 #1 \exp_stop_f:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      \else:
        \prg_return_true:
      \fi:
    \else:
      \if_num:w \int_eval:w (`#1-\c_six)/26 = \c_four
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

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

2.3 Building

2.3.1 Helpers for building an NFA

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

```
\cs_new_protected_nopar:Npn \regex_build_new_state:
  {
    \int_compare:nNnTF \l_regex_max_state_int > {32766}
      { \regex_build_error:n {-997} }
      {
        \int_incr:N \l_regex_max_state_int
        \tex_toks:D \l_regex_max_state_int { \s_regex_toks }
      }
    \int_set_eq:NN \l_regex_left_state_int \l_regex_right_state_int
    \int_set_eq:NN \l_regex_right_state_int \l_regex_max_state_int
  }
```

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

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

```
\cs_new_protected_nopar:Npn \regex_build_transition_aux:NN #1#2
  {
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      { #1 { \int_use:N #2 } }
  }
\cs_new_protected_nopar:Npn \regex_build_transitions_aux:NNNN #1#2#3#4
  {
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        #1 { \int_use:N #2 }
        #3 { \int_use:N #4 }
      }
  }
```

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

2.3.2 From regex to NFA: framework

In order for the construction ab|cd to work, we enclose the whole pattern within parentheses. These have the added benefit to form a capturing group: hence we get the data of the whole match for free.

\regex_build:n First, reset a few variables. Then use the generic framework defined in l3str to parse the regular expression once, recognizing which characters are raw characters, and which have special meanings. The result is stored in \g_str_result_tl, and can be run directly.

The trailing `\prg_do_nothing:` ensure that the look-ahead done by some of the operations is harmless. Finally, `\regex_build_end:` adds the finishing code (checking that parentheses are properly nested, for instance).

```
\cs_new_protected:Npn \regex_build:n #1
  {
    \regex_build_setup:
    \str_aux_escape:NNNn
      \regex_build_i_unescaped:N
      \regex_build_i_escaped:N
      \regex_build_i_raw:N
      { #1 }
    \regex_build_open_aux:
      \g_str_result_tl \prg_do_nothing: \prg_do_nothing:
    \seq_push:Nn \l_regex_capturing_group_seq { 0 }
    \regex_build_close_aux: \regex_build_group_:
    \regex_build_end:
  }
```

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

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

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

```
\cs_new_nopar:Npn \regex_build_i_unescaped:N #1
  {
    \regex_aux_char_if_special:NTF #1
      { \exp_not:N \regex_build_control:N #1 }
      { \exp_not:N \regex_build_raw:N #1 }
  }
\cs_new_nopar:Npn \regex_build_i_escaped:N #1
  {
    \regex_aux_char_if_alphanumeric:NTF #1
      { \exp_not:N \regex_build_control:N #1 }
      { \exp_not:N \regex_build_raw:N #1 }
  }
\cs_new_nopar:Npn \regex_build_i_raw:N #1
  { \exp_not:N \regex_build_raw:N #1 }
```

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

`\regex_build_default_control:N`

If the control character has a particular meaning in regular expressions, the corresponding function is used. Otherwise, it is interpreted as a raw character. The `\regex_build_default_raw:N` function is defined later.

```
\cs_new_protected_nopar:Npn \regex_build_default_control:N #1
  {
    \cs_if_exist_use:cF { regex_build_#1: }
      { \regex_build_default_raw:N #1 }
  }
```

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

`\regex_build_setup:` Hopefully, we didn't forget to initialize anything here. The search is not anchored: to acheive that, we insert state(s) responsible for repeating the match attempt on every character of the string: 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.

```
\cs_new_protected_nopar:Npn \regex_build_setup:
  {
    \cs_set_eq:NN \regex_build_control:N \regex_build_default_control:N
    \cs_set_eq:NN \regex_build_raw:N \regex_build_default_raw:N
    \int_set_eq:NN \l_regex_capturing_group_int \c_zero
    \int_zero:N \l_regex_left_state_int
    \int_zero:N \l_regex_right_state_int
    \int_zero:N \l_regex_max_state_int
    \regex_build_new_state:
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \regex_action_start_wildcard:nn
          { \int_use:N \l_regex_left_state_int }
          { \int_use:N \l_regex_right_state_int }
      }
  }
```

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

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

```
\cs_new_protected_nopar:Npn \regex_build_end:
  {
    \seq_if_empty:NF \l_regex_capturing_group_seq
      {
        \regex_build_error:n {22}
        \prg_replicate:nn
          { \seq_length:N \l_regex_capturing_group_seq } % (
          { \regex_build_close_aux: \regex_build_group_: }
      }
    \regex_toks_put_right:Nx \l_regex_right_state_int
      { \regex_action_success: }
  }
```

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

2.3.3 Anchoring and simple assertions

`\regex_build_A:` `\regex_build_^:` `\regex_build_G:` `\regex_build_anchor_start:N` Anchoring at the start corresponds to checking that the current character is the first in the string. Anchoring to the beginning of the match attempt uses `\l_regex_start_step_int` instead of `\c_zero`.

```
\cs_new_protected_nopar:cpn { regex_build_^: }
```

```
  { \regex_build_anchor_start:N \c_zero }
\cs_new_protected_nopar:Npn \regex_build_A:
  { \regex_build_anchor_start:N \c_zero }
\cs_new_protected_nopar:Npn \regex_build_G:
  { \regex_build_anchor_start:N \l_regex_start_step_int }
\cs_new_protected_nopar:Npn \regex_build_anchor_start:N #1
  {
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \exp_not:N \int_compare:nNnT {#1} = \l_regex_current_step_int
          { \regex_action_free:n { \int_use:N \l_regex_right_state_int } }
      }
  }
```

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

`\regex_build_Z:` `\regex_build_z:` `\regex_build_$:` This matches the end of the string, marked by a character code of -1.

```
\cs_new_protected_nopar:cpn { regex_build_$: } % $
  {
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \exp_not:N \int_compare:nNnT
          \c_minus_one = \l_regex_current_char_int
          { \regex_action_free:n { \int_use:N \l_regex_right_state_int } }
      }
  }
\cs_new_eq:Nc \regex_build_Z: { regex_build_$: } %$
\cs_new_eq:Nc \regex_build_z: { regex_build_$: } %$
```

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

`\regex_build_b:` `\regex_build_B:` `\regex_if_word_boundary:TF` Contrarily to `^` and `$`, which could be implemented without really knowing what preceeds in the string, this requires more information. We request it by setting `\l_regex_look_behind_bool`. Then the matching code will keep store the characters that were already read, backwards, in `\l_regex_look_behind_str`, and we can analyse the first character of that string.

```
\cs_new_protected_nopar:Npn \regex_build_b:
  {
    \bool_set_true:N \l_regex_look_behind_bool
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \exp_not:N \regex_if_word_boundary:TF
          { \regex_action_free:n { \int_use:N \l_regex_right_state_int } }
          { }
      }
  }
\cs_new_protected_nopar:Npn \regex_build_B:
  {
```

```
    \bool_set_true:N \l_regex_look_behind_bool
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \exp_not:N \regex_if_word_boundary:TF
          { }
          { \regex_action_free:n { \int_use:N \l_regex_right_state_int } }
      }
  }
\cs_new_protected_nopar:Npn \regex_if_word_boundary:TF
  {
    \tl_if_empty:NTF \l_regex_look_behind_str
      { \c_regex_w_tl }
      {
        \group_begin:
          \cs_set_nopar:Npx \regex_tmp:w
            {
              \int_set:Nn \l_regex_current_char_int
                { ‘ \str_head:N \l_regex_look_behind_str }
            }
          \regex_tmp:w
          \c_regex_w_tl
          \regex_break_point:TF
            { \group_end: \c_regex_W_tl }
            { \group_end: \c_regex_w_tl }
      }
    \regex_break_point:TF
  }
```

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

2.3.4 Normal character, and simple character classes

`\regex_build_default_raw:N`

A normal alphanumeric or an escaped non-alphanumeric (actually, any unknown combination) will match itself and the thread will fail otherwise. We prepare `\regex_build_tmp_class:n` with the relevant test and commands. The program steps to be inserted in those commands will come as `##1` and `##2`: we don't know yet what those will be before checking for quantifiers.

```
\cs_new_protected_nopar:Npn \regex_build_default_raw:N #1
  {
    \cs_set:Npx \regex_build_tmp_class:n ##1
      {
        \exp_not:n { \exp_not:N \if_num:w }
            \int_value:w ‘#1 = \l_regex_current_char_int
          \regex_action_cost:n { ##1 }
        \exp_not:n { \exp_not:N \fi: }
      }
    \regex_build_one_quantifier:
  }
```

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

\regex_build_.: Similar to \regex_build_default_raw:N but accepts any character, and refuses −1, which marks the end of the string.

```
\cs_new_protected_nopar:cpn { regex_build_.: }
  {
    \cs_set:Npn \regex_build_tmp_class:n ##1
      {
        \exp_not:N \if_num:w \c_minus_one < \l_regex_current_char_int
          \regex_action_cost:n {##1}
        \exp_not:N \fi:
      }
    \regex_build_one_quantifier:
  }
```

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

\regex_build_d: \regex_build_D \regex_build_h: \regex_build_H \regex_build_s: \regex_build_S \regex_build_v: \regex_build_V \regex_build_w: \regex_build_W \regex_build_N: \regex_build_char_type:N

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

```
\cs_new_protected_nopar:Npn \regex_build_char_type:N #1
  {
    \cs_set:Npn \regex_build_tmp_class:n ##1
      {
        \exp_not:N #1
        \exp_not:N \regex_break_point:TF
          { \regex_action_cost:n {##1} }
          { }
      }
    \regex_build_one_quantifier:
  }
\tl_map_inline:nn { dDhHsSvVwWN }
  {
    \cs_new_protected_nopar:cpx { regex_build_#1: }
      {
        \exp_not:N \regex_build_char_type:N
        \exp_not:c { c_regex_#1_tl }
      }
  }
```

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

2.3.5 Character classes

\regex_build_[: This starts a class. The code for the class is collected in \l_regex_class_tl. The first character is special.

```
\cs_new_protected_nopar:cpn { regex_build_[: }
  {
    \tl_clear:N \l_regex_class_tl
    \cs_set_eq:NN \regex_build_control:N \regex_class_control:N
    \cs_set_eq:NN \regex_build_raw:N \regex_class_raw:N
    \regex_class_first:NN
```

```
}
```

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

`\regex_class_control:N` This function is similar to `\regex_build_control:N`. If the control character has a meaning in character classes, call the corresponding function, otherwise, treat it as a raw character, with the `\regex_class_raw:N` function, defined later.

```
\cs_new_protected_nopar:Npn \regex_class_control:N #1
  {
    \cs_if_exist_use:cF { regex_class_#1: }
      { \regex_class_raw:N #1 }
  }
```

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

`\regex_class_]:` If `]` appears as the first item of a class, then it doesn't end the class. Otherwise, it's the end, act just as for a single character, but with a more complicated test. And restore `\regex_build_control:N` and `\regex_build_raw:N`.

```
\cs_new_protected_nopar:cpn { regex_class_]: }
  {
    \tl_if_empty:NTF \l_regex_class_tl %[
      { \regex_class_raw:N ] }
      {
        \cs_set_eq:NN \regex_build_control:N \regex_build_default_control:N
        \cs_set_eq:NN \regex_build_raw:N  \regex_build_default_raw:N
        \cs_set:Npn \regex_build_tmp_class:n ##1
          {
            \exp_not:o \l_regex_class_tl
            \bool_if:NTF \l_regex_class_bool
              {
                \exp_not:N \regex_break_point:TF
                  { \regex_action_cost:n {##1} }
                  { }
              }
              {
                \exp_not:N \regex_break_point:TF
                  { }
                  { \regex_action_cost:n {##1} }
              }
          }
        \regex_build_one_quantifier:
      }
  }
```

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

`\regex_class_first:NN` If the first non-space character is `^`, then the class is inverted. We keep track of this in `\l_regex_class_bool`.

```
\cs_new_protected_nopar:Npn \regex_class_first:NN #1#2
  {
    \str_if_eq:nnTF {#1#2} { \regex_build_control:N ^ }
```

```
      { \bool_set_false:N \l_regex_class_bool }
      {
        \bool_set_true:N \l_regex_class_bool
        #1 #2
      }
  }
```

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

`\regex_class_raw:N`
`\regex_class_single:N`

Most characters are treated here. We look ahead for an unescaped dash. If there is none, then the character matches itself.

```
\cs_new_protected_nopar:Npn \regex_class_raw:N #1#2#3
  {
    \str_if_eq:nnTF {#2#3} { \regex_build_control:N - }
      { \regex_class_range:Nw #1 }
      {
        \regex_class_single:N #1
        #2 #3
      }
  }
\cs_new_protected_nopar:Npn \regex_class_single:N #1
  {
    \tl_put_right:Nx \l_regex_class_tl
      { \exp_not:N \regex_item_equal:n { \int_value:w `#1 } }
  }
```

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

`\regex_class_range:Nw`
`\regex_class_range_put:NN`

If the character is followed by a dash, we look for the end-point of the range. For "raw" characters, that's simply `#3`. Most "control" characters also have no meaning, and can serve as an end-point, but those with a meaning interrupt the range. In the case of a true range, check whether the end-points are in the right order, and optimize in the case of equal end-points.

```
\cs_new_protected_nopar:Npn \regex_class_range:Nw #1#2#3
  {
    \token_if_eq_meaning:NNTF #2 \regex_build_control:N
      {
        \cs_if_exist:cTF { regex_class_#3: }
          {
            \regex_class_single:N #1
            \regex_class_single:N -
            #2#3
          }
          { \regex_class_range_put:NN #1#3 }
      }
      { \regex_class_range_put:NN #1#3 }
  }
\cs_new_protected_nopar:Npn \regex_class_range_put:NN #1#2
  {
    \if_num:w `#1 > `#2 \exp_stop_f:
      \regex_build_error:n {8}
```

```
    \else:
      \tl_put_right:Nx \l_regex_class_tl
        {
          \if_num:w `#1 = `#2 \exp_stop_f:
            \exp_not:N \regex_item_equal:n
          \else:
            \exp_not:N \regex_item_range:nn { \int_value:w `#1 }
          \fi:
          { \int_value:w `#2 }
        }
    \fi:
  }
```

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

`\regex_class_d:` `\regex_class_D:` `\regex_class_h:` `\regex_class_H:` `\regex_class_s:` `\regex_class_S:` `\regex_class_v:` `\regex_class_V:` `\regex_class_w:` `\regex_class_W:`

Similar to `\regex_class_single:N`, adding the appropriate ranges of characters to the class. The token lists are not expanded because it is more memory efficient, with a tiny overhead on execution.

```
\tl_map_inline:nn { dDhHsSvVwWN }
  {
    \cs_new_protected_nopar:cpx { regex_class_#1: }
      {
        \tl_put_right:Nn \exp_not:N \l_regex_class_tl
          { \exp_not:c { c_regex_#1_tl } }
      }
  }
```

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

2.3.6 Quantifiers

`\regex_build_quantifier:w`

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

```
\cs_new_protected_nopar:Npn \regex_build_quantifier:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \regex_build_control:N
      {
        \cs_if_exist_use:cF { regex_build_quantifier_#2:w }
          {
            \regex_build_quantifier_end:n { }
            #1 #2
          }
      }
      {
        \regex_build_quantifier_end:n { }
        #1 #2
      }
  }
```

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

\regex_build_quantifier_?:w
\regex_build_quantifier_*:w
\regex_build_quantifier_+:w

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

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

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

\regex_build_quantifier_aux:nnNN

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

```
\cs_new_protected_nopar:Npn \regex_build_quantifier_aux:nnNN #1#2#3#4
  {
    \str_if_eq:nnTF { #3 #4 } { \regex_build_control:N ? }
      { \regex_build_quantifier_end:n { #2 #4 } #1 }
      {
        \regex_build_quantifier_end:n { #2 } #1
        #3 #4
      }
  }
```

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

\regex_build_quantifier_{:w
\regex_build_quantifier_lbrace:n
\regex_build_quantifier_lbrace:nw
\regex_build_quantifier_lbrace:nnw

Three possible syntaxes: {⟨*int*⟩}, {⟨*int*⟩,}, or {⟨*int*⟩,⟨*int*⟩}.

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

```
      {
        \exp_not:N \regex_build_quantifier_end:n { }
        \exp_not:N \regex_build_raw:N \c_lbrace_str #1#2
      }
  }
\cs_new_protected_nopar:Npn \regex_build_quantifier_lbrace:nnw #1#2#3
  {
    \str_if_eq:xxTF { \exp_not:N #3 } { \c_rbrace_str }
      {
        \tl_if_empty:nTF {#2}
          { \regex_build_quantifier_aux:nnN { {#1} {\c_max_int} } {nn} }
          {
            \int_compare:nNnT {#1} > {#2}
              { \regex_build_error:n {4} }
            \regex_build_quantifier_aux:nnN { {#1} {#2} } {nn}
          }
      }
      {
        \regex_build_quantifier_end:n { }
        \use:x
          {
            \exp_not:n { \exp_args:No \tl_map_function:nN }
              { \c_lbrace_str #1 #2 , }
            \regex_build_raw:N
          }
      }
  }
```

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

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

```
\cs_new_protected_nopar:Npn \regex_build_quantifier_end:n #1
  { \use:c { regex_build_ \l_regex_one_or_group_tl _ #1 : } }
```

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

2.3.7 Quantifiers for one character or character class

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

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

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

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

```
\cs_new_protected_nopar:Npn \regex_build_one_:
  {
    \regex_build_transition_aux:NN
      \regex_build_tmp_class:n \l_regex_right_state_int
  }
```

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

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

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

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

`\regex_build_one_*:` `\regex_build_one_*?:` Build a costly transition going from the current state to itself, and a free transition moving to a new state.

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

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

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

```
\cs_new_protected_nopar:cpn { regex_build_one_+: }
  {
    \regex_build_one_:
```

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

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

`\regex_build_one_n:`
`\regex_build_one_n?:`

This function is called in case the syntax is {⟨*int*⟩}. Greedy and lazy operators are identical, since the number of repetitions is fixed.

```
\cs_new_protected_nopar:Npn \regex_build_one_n: #1
  {
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_right_state_int
    \regex_build_new_state:
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_tmpa_int
      { %^^A safe \if nesting?
        \exp_not:N \if_num:w #1 > \l_regex_repetition_int
          \regex_action_repeat_move:n
            { \int_use:N \l_regex_left_state_int }
        \exp_not:N \else:
          \regex_action_no_repeat_move:n
            { \int_use:N \l_regex_right_state_int }
        \exp_not:N \fi:
      }
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \regex_build_tmp_class:n
          { \int_use:N \l_regex_tmpa_int }
      }
  }
\cs_new_eq:cN { regex_build_one_n?: } \regex_build_one_n:
```

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

`\regex_build_one_nn:`
`\regex_build_one_nn?:`
`\regex_build_one_nn_aux:nn`

This function is called when the syntax is either {⟨*int*⟩,} or {⟨*int*⟩,⟨*int*⟩}.

```
\cs_new_protected_nopar:Npn \regex_build_one_nn: #1#2
  {
    \regex_build_one_nn_aux:nn {#1}
      {
        \exp_not:N \if_num:w #2 > \l_regex_repetition_int
          \regex_action_repeat_copy:n
            { \int_use:N \l_regex_left_state_int }
```

```
        \exp_not:N \fi:
        \regex_action_no_repeat_move:n
          { \int_use:N \l_regex_right_state_int }
      }
  }
\cs_new_protected_nopar:cpn { regex_build_one_nn?: } #1#2
  {
    \regex_build_one_nn_aux:nn {#1}
      {
        \regex_action_no_repeat_copy:n
          { \int_use:N \l_regex_right_state_int }
        \exp_not:N \if_num:w #2 > \l_regex_repetition_int
          \regex_action_repeat_move:n
            { \int_use:N \l_regex_left_state_int }
        \exp_not:N \fi:
      }
  }
\cs_new_protected_nopar:Npn \regex_build_one_nn_aux:nn #1#2
  {
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_right_state_int
    \regex_build_new_state:
    \regex_build_new_state:
    \regex_toks_put_right:Nx \l_regex_tmpa_int
      {
        \exp_not:N \if_num:w #1 > \l_regex_repetition_int
          \regex_action_repeat_move:n
            { \int_use:N \l_regex_left_state_int }
        \exp_not:N \else:
          #2
        \exp_not:N \fi:
      }
    \regex_toks_put_right:Nx \l_regex_left_state_int
      {
        \regex_build_tmp_class:n
          { \int_use:N \l_regex_tmpa_int }
      }
  }
```

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

2.3.8 Groups and alternation

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

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

Grouping and alternation go together.

- Allocate the next available number for the end vertex of the alternation/group and store it on a stack (so that nested alternations work).
- Put free transitions to separate all cases of the alternation.

- Build each branch separately, and merge them to the common end-node.
- Test for a quantifier, and if needed, transfer the initial vertex to a new vertex.

```
\cs_new_protected_nopar:cpn { regex_build_(: } #1#2
  {
    \str_if_eq:nnTF { #1 #2 } { \regex_build_control:N ? }
      { \regex_build_special_group:NN }
      {
        \int_incr:N \l_regex_capturing_group_int
        \seq_push:Nx \l_regex_capturing_group_seq
          { \int_use:N \l_regex_capturing_group_int }
        \regex_build_open_aux:
        #1 #2
      }
  }
\cs_new_protected_nopar:Npn \regex_build_open_aux:
  {
    \regex_build_new_state:
    \seq_push:Nx \l_regex_left_state_seq
      { \int_use:N \l_regex_left_state_int }
    \seq_push:Nx \l_regex_right_state_seq
      { \int_use:N \l_regex_right_state_int }
    \regex_build_begin_alternation:
  }
\cs_new_protected_nopar:cpn { regex_build_|: }
  {
    \regex_build_end_alternation:
    \regex_build_begin_alternation:
  }
\cs_new_protected_nopar:cpn { regex_build_): }
  {
    \seq_if_empty:NTF \l_regex_capturing_group_seq
      { \regex_build_error:n {22} }
      {
        \regex_build_close_aux:
        \regex_build_group_quantifier:
      }
  }
\cs_new_protected_nopar:Npn \regex_build_close_aux:
  {
    \regex_build_end_alternation:
    \seq_pop:NN \l_regex_left_state_seq  \l_regex_tmpa_tl
    \int_set:Nn \l_regex_left_state_int  \l_regex_tmpa_tl
    \seq_pop:NN \l_regex_right_state_seq \l_regex_tmpa_tl
    \int_set:Nn \l_regex_right_state_int \l_regex_tmpa_tl
  }
```

Building each branch.

```
\cs_new_protected_nopar:Npn \regex_build_begin_alternation:
  {
```

```
    \regex_build_new_state:
    \seq_get:NN \l_regex_left_state_seq \l_regex_tmpa_tl
    \int_set:Nn \l_regex_left_state_int \l_regex_tmpa_tl
    \regex_toks_put_right:Nx \l_regex_left_state_int
      { \regex_action_free:n { \int_use:N \l_regex_right_state_int } }
  }
\cs_new_protected_nopar:Npn \regex_build_end_alternation:
  {
    \seq_get:NN \l_regex_right_state_seq \l_regex_tmpa_tl
    \regex_toks_put_right:Nx \l_regex_right_state_int
      { \regex_action_free:n { \l_regex_tmpa_tl } }
  }
```

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

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

```
\cs_new_protected_nopar:Npn \regex_build_special_group:NN #1#2
  {
    \cs_if_exist_use:cF { regex_build_special_group_\token_to_str:N #2 : }
      {
        \regex_build_error:n { -998 }
        \regex_build_control:N ( % )
        \regex_build_raw:N ?
        #1 #2
      }
  }
```

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

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

```
\cs_new_protected_nopar:cpn { regex_build_special_group_:: }
  {
    \seq_push:Nx \l_regex_capturing_group_seq { * }
    \regex_build_open_aux:
  }
```

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

2.3.9 Quantifiers for groups

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

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

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

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

```
\cs_new_protected_nopar:Npn \regex_build_group_submatches:NN #1#2
  {
    \seq_pop:NN \l_regex_capturing_group_seq \l_regex_tmpa_tl
    \str_if_eq:xxF { \l_regex_tmpa_tl } { * }
      {
        \regex_toks_put_left:Nx #1
          { \regex_action_submatch:n { \l_regex_tmpa_tl < } }
        \regex_toks_put_left:Nx #2
          { \regex_action_submatch:n { \l_regex_tmpa_tl > } }
      }
  }
```

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

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

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

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

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

```
\cs_new_protected_nopar:Npn \regex_build_group_shift:N #1
  {
    \int_set_eq:NN \l_regex_tmpa_int \l_regex_left_state_int
    \regex_build_new_state:
    \tex_toks:D \l_regex_right_state_int = \tex_toks:D \l_regex_tmpa_int
    \use:x
      {
        \tex_toks:D \l_regex_tmpa_int
          { \s_regex_toks #1 { \int_use:N \l_regex_right_state_int } }
      }
    \regex_build_group_submatches:NN
      \l_regex_right_state_int \l_regex_left_state_int
    \int_set_eq:NN \l_regex_right_state_int \l_regex_left_state_int
  }
```

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

`\regex_build_group_qs_aux:NN`
`\regex_build_group_?:`
`\regex_build_group_??:`
`\regex_build_group_*:`
`\regex_build_group_*?:`
Shift the state at which the group begins using `\regex_build_group_shift:N`, then add two transitions. The first transition is taken once the group has been traversed: in the case of `?` and `??`, we should exit by going to `\l_regex_right_state_int`, while for `*`

and `*?` we loop by going to `\l_regex_tmpa_int`. The second transition corresponds to skipping the group; it has lower priority (`put_right`) for greedy operators, and higher priority (`put_left`) for lazy operators.

```
\cs_new_protected_nopar:Npn \regex_build_group_qs_aux:NN #1#2
  {
    \regex_build_group_shift:N \regex_action_free:n
    \regex_build_transition_aux:NN \regex_action_free:n #1
    #2 \l_regex_tmpa_int
      { \regex_action_free:n { \int_use:N \l_regex_right_state_int } }
  }
\cs_new_protected_nopar:cpn { regex_build_group_?: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_right_state_int \regex_toks_put_right:Nx
  }
\cs_new_protected_nopar:cpn { regex_build_group_??: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_right_state_int \regex_toks_put_left:Nx
  }
\cs_new_protected_nopar:cpn { regex_build_group_*: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_tmpa_int \regex_toks_put_right:Nx
  }
\cs_new_protected_nopar:cpn { regex_build_group_*?: }
  {
    \regex_build_group_qs_aux:NN
      \l_regex_tmpa_int \regex_toks_put_left:Nx
  }
```

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

`\regex_build_group_+:`
`\regex_build_group_+?:`

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

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

```
    \regex_build_transitions_aux:NNNN
      \regex_action_free:n \l_regex_right_state_int
      \regex_action_free:n \l_regex_tmpa_int
  }
```

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

`\regex_build_group_n:`
`\regex_build_group_n?:`

These functions are called in case the syntax is `{`⟨*int*⟩`}`. Greedy and lazy operators are identical, since the number of repetitions is fixed.

```
\cs_new_protected_nopar:Npn \regex_build_group_n: #1
  {
    \regex_build_group_shift:N \regex_action_repeat_move:n
    \regex_build_transition_aux:NN
      \regex_action_free:n \l_regex_tmpa_int
    \use:x
      {
        \tex_toks:D \l_regex_tmpa_int
          {
            \s_regex_toks
            \exp_not:N \if_num:w #1 > \l_regex_repetition_int
              \tex_the:D \tex_toks:D \l_regex_tmpa_int
            \exp_not:N \else:
              \regex_action_no_repeat_move:n
                { \int_use:N \l_regex_right_state_int }
            \exp_not:N \fi:
          }
      }
  }
\cs_new_eq:cN { regex_build_group_n?: } \regex_build_group_n:
```

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

`\regex_build_group_nn:`
`\regex_build_group_nn?:`

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

```
\cs_new_protected_nopar:Npn \regex_build_group_nn: #1#2
  {
    \regex_build_group_shift:N \regex_action_repeat_move:n
    \regex_build_transition_aux:NN
      \regex_action_free:n \l_regex_tmpa_int
    \use:x
      {
        \tex_toks:D \l_regex_tmpa_int
          {
            \s_regex_toks
            \exp_not:N \if_num:w #1 > \l_regex_repetition_int
              \tex_the:D \tex_toks:D \l_regex_tmpa_int
            \exp_not:N \else:
              \exp_not:N \if_num:w #2 > \l_regex_repetition_int
                \tex_the:D \tex_toks:D \l_regex_tmpa_int
              \exp_not:N \fi:
              \regex_action_no_repeat_copy:n
                { \int_use:N \l_regex_right_state_int }
```

```
            \exp_not:N \fi:
          }
      }
  }
\cs_new_protected_nopar:cpn { regex_build_group_nn?: } #1#2
  {
    \regex_build_group_shift:N \regex_action_repeat_move:n
    \regex_build_transition_aux:NN
      \regex_action_free:n \l_regex_tmpa_int
    \use:x
      {
        \tex_toks:D \l_regex_tmpa_int
          {
            \s_regex_toks
            \exp_not:N \if_num:w #1 > \l_regex_repetition_int
              \tex_the:D \tex_toks:D \l_regex_tmpa_int
            \exp_not:N \else:
              \regex_action_no_repeat_copy:n
                { \int_use:N \l_regex_right_state_int }
              \exp_not:N \if_num:w #2 > \l_regex_repetition_int
                \tex_the:D \tex_toks:D \l_regex_tmpa_int
              \exp_not:N \fi:
            \exp_not:N \fi:
          }
      }
  }
```

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

2.4 Matching

2.4.1 Use of TEX registers when matching

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

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

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

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

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

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

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

2.4.2 Helpers for running the NFA

`\regex_if_state_free:nT` A state is free if it is not marker as taken, namely if the corresponding `\dimen` register is 0`sp` rather than 1`sp`.

```
\cs_new_protected:Npn \regex_if_state_free:nT #1
  {
    \if_num:w \tex_dimen:D #1 = \l_regex_unique_step_int
      \exp_after:wN \use_none:n
    \else:
      \exp_after:wN \use:n
    \fi:
  }
```

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

`\regex_store_state:n` Put the given state in the array of `\skip` registers. This is done by increasing the pointer `\l_regex_max_index_int`, and converting the integer to a dimension (suitable for a `\skip` assignment) in scaled points.

```
\cs_new_protected:Npn \regex_store_state:n #1
  {
    \int_incr:N \l_regex_max_index_int
```

```
    \tex_skip:D \l_regex_max_index_int #1 sp \scan_stop:
    \regex_store_submatches:n {#1}
  }
```

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

`\regex_state_use:`
`\regex_state_use_submatches:`
`\regex_state_use_aux:n`
`\regex_state_use_aux_ii:w`

Use a given program instruction, unless it has already been executed at this step. The `\toks` registers begin with some submatch information, ignored by `\regex_state_use:`, but not by `\regex_state_use_submatches:`.

```
\cs_new_protected_nopar:Npn \regex_state_use_submatches:
  { \regex_state_use_aux:n { } }
\cs_new_protected_nopar:Npn \regex_state_use:
  { \regex_state_use_aux:n { \exp_after:wN \regex_state_use_aux_ii:w } }
\cs_new:Npn \regex_state_use_aux_ii:w #1 \s_regex_toks { }
\cs_new_protected:Npn \regex_state_use_aux:n #1
  {
    \regex_if_state_free:nT {  \l_regex_current_state_int }
      {
        \tex_dimen:D \l_regex_current_state_int
          = \l_regex_unique_step_int sp \scan_stop:
        #1 \tex_the:D \tex_toks:D \l_regex_current_state_int \scan_stop:
      }
  }
```

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

2.4.3 Submatch tracking when running the NFA

`\regex_disable_submatches:`

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

```
\cs_new_protected_nopar:Npn \regex_disable_submatches:
  {
    \cs_set_eq:NN \regex_state_use_submatches: \regex_state_use:
    \cs_set_eq:NN \regex_store_submatches:n
      \regex_protected_use_none:n
    \cs_set_eq:NN \regex_action_submatches:n
      \regex_protected_use_none:n
  }
\cs_new_protected:Npn \regex_protected_use_none:n #1 { }
```

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

`\regex_store_submatches:n`
`\regex_store_submatches_aux:Nw`
`\regex_store_submatches_aux_ii:wnnwwn`

The submatch information pertaining to one given thread is moved from state to state as we execute the NFA.

```
\cs_new_protected:Npn \regex_store_submatches:n #1
  {
    \exp_after:wN \regex_store_submatches_aux:Nw
    \exp_after:wN \l_regex_current_submatches_prop
    \tex_the:D \tex_toks:D #1 \scan_stop: \q_stop
    { #1 }
```

```
  }
\cs_new_protected:Npn \regex_store_submatches_aux:Nw #1 #2 \s_regex_toks
  {
    \exp_args:No \regex_store_submatches_aux_ii:wnnwwn { #1 }
      #2
      \regex_state_submatches:nn { \c_minus_one } { \q_prop }
    \s_regex_toks
  }
\cs_new_protected:Npn \regex_store_submatches_aux_ii:wnnwwn
    #1 \regex_state_submatches:nn #2#3 #4 \s_regex_toks #5 \q_stop #6
  {
    \tex_toks:D #6 \exp_after:wN
      {
        \exp_after:wN \regex_state_submatches:nn \exp_after:wN
          { \int_value:w \int_eval:w \l_regex_unique_step_int + \c_one }
          { #1 }
        \regex_state_submatches:nn {#2} {#3}
        \s_regex_toks
        #5
      }
  }
```

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

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

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

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

2.4.4 Matching: framework

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

```
\cs_new_protected:Npn \regex_match:n #1
  {
    \tl_set:Nx \l_regex_query_str { \tl_to_other_str:n {#1} }
    \regex_match_initial_setup:
    \regex_match_once:
  }
```

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

`\regex_match_once:` `\regex_match_once_aux:` After setting up more variables in `\regex_match_setup:`, skip the `\l_regex_start_step_int` first characters of the query string, and loop over it. If there was a match,

use the token list \l_regex_every_match_tl, which may call \regex_match_once: to achieve multiple matches.

```
\cs_new_protected_nopar:Npn \regex_match_once:
  {
    \regex_match_setup:
    \exp_after:wN \regex_match_once_aux: \l_regex_query_str
      \q_recursion_tail \q_recursion_stop
    \bool_if:NT \l_regex_success_bool { \l_regex_every_match_tl }
  }
\cs_new_protected_nopar:Npn \regex_match_once_aux:
  { \str_skip_do:nn { \l_regex_start_step_int } { \regex_match_loop:N } }
```

(*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 on a given string. It is called only once for the whole string. On the other hand, \regex_match_setup: is called for every match in the string in case of repeated matches, and \regex_match_loop_setup: is called at every step.

```
\cs_new_protected_nopar:Npn \regex_match_initial_setup:
  {
    \tl_clear:N \l_regex_look_behind_str
    \prg_stepwise_inline:nnnn {1} {1} { \l_regex_max_state_int }
      { \tex_dimen:D ##1 \c_minus_one sp \scan_stop: }
    \int_set_eq:NN \l_regex_unique_step_int  \c_minus_one
    \int_set_eq:NN \l_regex_start_step_int   \c_minus_one
    \int_set_eq:NN \l_regex_current_step_int \c_zero
    \int_set_eq:NN \l_regex_success_step_int \c_zero
    \bool_set_false:N \l_regex_success_empty_bool
  }
```

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

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

```
\cs_new_protected_nopar:Npn \regex_match_setup:
  {
    \prop_clear:N \l_regex_current_submatches_prop
    \bool_if:NTF \l_regex_success_empty_bool
      { \cs_set_eq:NN \regex_last_match_empty:F \regex_last_match_empty_yes:F }
      { \cs_set_eq:NN \regex_last_match_empty:F \regex_last_match_empty_no:F }
    \int_set_eq:NN \l_regex_start_step_int \l_regex_success_step_int
    \int_set_eq:NN \l_regex_current_step_int \l_regex_start_step_int
    \int_decr:N \l_regex_current_step_int
    \bool_set_false:N \l_regex_success_bool
    \int_zero:N \l_regex_max_index_int
    \regex_store_state:n {1}
  }
```

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

\regex_match_loop_setup: This is called for every character in the string.

```
\cs_new_protected_nopar:Npn \regex_match_loop_setup:
```

```
  {
    \int_incr:N \l_regex_current_step_int
    \int_incr:N \l_regex_unique_step_int
    \bool_set_false:N \l_regex_fresh_thread_bool
  }
```

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

`\s_regex_step`
`\regex_break_step:w`

When a thread succeeds, all threads with lower precedence can be ignored, and the next character should be read. This is done by skipping to `\s_regex_step`. This marker does nothing if no thread succeeded at this step.

```
\cs_new_eq:NN \s_regex_step \scan_stop:
\cs_new:Npn \regex_break_step:w #1 \s_regex_step { }
```

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

`\regex_match_loop:N`
`\regex_match_one_index:n`
`\regex_match_one_index_aux:nn`

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

```
\cs_new_protected_nopar:Npn \regex_match_loop:N #1
  {
    \regex_match_loop_setup:
    \token_if_eq_meaning:NNTF #1 \q_recursion_tail
      { \int_set_eq:NN \l_regex_current_char_int \c_minus_one }
      { \int_set:Nn \l_regex_current_char_int {`#1} }
    \cs_set_nopar:Npx \regex_tmp:w
      {
        \int_zero:N \l_regex_max_index_int
        \prg_stepwise_function:nnnN
          {1} {1} { \l_regex_max_index_int }
          \regex_match_one_index:n
      }
    \regex_tmp:w
    \s_regex_step
    \if_num:w \l_regex_max_index_int = \c_zero
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \quark_if_recursion_tail_stop:N #1
    \bool_if:NT \l_regex_look_behind_bool
      { \tl_put_left:Nx \l_regex_look_behind_str {#1} }
    \regex_match_loop:N
  }
\cs_new_nopar:Npn \regex_match_one_index:n #1
  {
    \regex_match_one_index_aux:n { \int_value:w \tex_skip:D #1 }
  }
\cs_new_protected_nopar:Npn \regex_match_one_index_aux:n #1
  {
    \int_set:Nn \l_regex_current_state_int {#1}
    \prop_clear:N \l_regex_current_submatches_prop
```

```
    \regex_state_use_submatches:
  }
```

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

2.4.5 Actions when matching

`\regex_action_start_wildcard:nn` 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 character in the query string. This combination could be reused (with some changes). We sometimes need to know that the match for a given thread starts at this character. For that, we use the boolean `\l_regex_fresh_thread_bool`.

```
\cs_new_protected_nopar:Npn \regex_action_start_wildcard:nn #1#2
  {
    \bool_set_true:N \l_regex_fresh_thread_bool
    \regex_action_free:n {#2}
    \bool_set_false:N \l_regex_fresh_thread_bool
    \regex_action_cost:n {#1}
  }
```

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

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

```
\cs_new_protected_nopar:Npn \regex_action_cost:n #1
  { \regex_store_state:n {#1} }
```

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

`\regex_action_success:` There is a successful match when an execution path reaches the end of the regular expression. Then store the current step and submatches. The current step is then interrupted, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```
\cs_new_protected_nopar:Npn \regex_action_success:
  {
    \regex_last_match_empty:F
      {
        \bool_set_true:N \l_regex_success_bool
        \bool_set_eq:NN \l_regex_success_empty_bool
          \l_regex_fresh_thread_bool
        \int_set_eq:NN \l_regex_success_step_int
          \l_regex_current_step_int
        \prop_set_eq:NN \l_regex_success_submatches_prop
          \l_regex_current_submatches_prop
        \regex_break_step:w
      }
  }
```

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

\regex_action_free:n

To copy a thread, check whether the program state has already been used at this character. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l_regex_current_state_int` and of the current submatches.

```
\cs_new_protected_nopar:Npn \regex_action_free:n #1
  {
    \regex_if_state_free:nT {#1}
      {
        \use:x
          {
            \int_set:Nn \l_regex_current_state_int {#1}
            \regex_store_submatches:n { \l_regex_current_state_int }
            \regex_state_use:
            \int_set:Nn \l_regex_current_state_int
              { \int_use:N \l_regex_current_state_int }
            \tl_set:Nn \exp_not:N \l_regex_current_submatches_prop
              { \exp_not:o \l_regex_current_submatches_prop }
          }
      }
  }
```

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

\regex_action_submatch:n

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

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

```
\cs_new_protected_nopar:Npn \regex_action_repeat_copy:n { \ERROR }
\cs_new_protected_nopar:Npn \regex_action_repeat_move:n { \ERROR }
\cs_new_protected_nopar:Npn \regex_action_no_repeat_copy:n { \ERROR }
\cs_new_protected_nopar:Npn \regex_action_no_repeat_move:n { \ERROR }
```

2.5 Submatches, once the correct match is found

\regex_extract:
\regex_extract_aux:nTF

```
\cs_new_protected_nopar:Npn \regex_extract:
  {
    \seq_gclear:N \g_regex_submatches_seq
    \prg_stepwise_inline:nnnn
      {0} {1} { \l_regex_capturing_group_int }
      {
        \regex_extract_aux:nTF { ##1 }
          {
            \seq_gput_right:Nx \g_regex_submatches_seq
              {
```

```
                \str_from_to:Nnn \l_regex_query_str
                  { \l_regex_tmpa_tl }
                  { \l_regex_tmpb_tl }
              }
          }
          { \seq_gput_right:Nn \g_regex_submatches_seq { } }
      }
  }
\cs_new_protected_nopar:Npn \regex_extract_aux:nTF #1#2#3
  {
    \prop_get:NnNTF \l_regex_success_submatches_prop
      { #1 < } \l_regex_tmpa_tl
      {
        \prop_get:NnNTF \l_regex_success_submatches_prop
          { #1 > } \l_regex_tmpb_tl
          {#2}
          {#3}
      }
      {#3}
  }
```

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

2.6 User commands

2.6.1 Precompiled pattern

A given pattern is often reused to match many different strings. 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*$_n$⟩ }
`\regex_nfa_end:`

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

`\regex_set:Nn`
`\regex_gset:Nn`
`\regex_set_aux:NNn`
`\regex_set_aux:n`
`\regex_nfa:Nw`
`\regex_nfa_end:`

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 `\g_regex_tmpa_tl`, then transferred to the user's tl variable. The auxiliary `\regex_nfa:Nw` is not protected: this ensures that the NFA will properly be replaced by an error message in expansion contexts.

```
\cs_new_protected_nopar:Npn \regex_set:Nn
```

```
  { \regex_set_aux:NNn \tl_set_eq:NN }
\cs_new_protected_nopar:Npn \regex_gset:Nn
  { \regex_set_aux:NNn \tl_gset_eq:NN }
\cs_new_protected:Npn \regex_set_aux:NNn #1#2#3
  {
    \group_begin:
      \regex_build:n {#3}
      \tl_gset:Nx \g_regex_tmpa_tl
        {
          \exp_not:N \regex_nfa:Nw \exp_not:N #2
          \l_regex_max_state_int
            = \int_use:N \l_regex_max_state_int
          \l_regex_capturing_group_int
            = \int_use:N \l_regex_capturing_group_int
          \bool_if:NTF \l_regex_look_behind_bool
            { \bool_set_true:N \l_regex_look_behind_bool }
            { \bool_set_false:N \l_regex_look_behind_bool }
          \prg_stepwise_function:nnnN
            {1} {1} {\l_regex_max_state_int}
            \regex_set_aux:n
          \regex_nfa_end:
        }
    \group_end:
    #1 #2 \g_regex_tmpa_tl
  }
\cs_new_nopar:Npn \regex_set_aux:n #1
  { \tex_toks:D #1 { \tex_the:D \tex_toks:D #1 } }
\cs_new:Npn \regex_nfa:Nw #1 #2 \regex_nfa_end:
  { \msg_expandable_error:n { Automaton~#1 used~incorrectly. } }
\cs_new_eq:NN \regex_nfa_end: \scan_stop:
```

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

`\regex_check_nfa:N` If a token list variable starts with `\regex_nfa:Nw`, then it most likely holds the data for a precompiled pattern.

```
\prg_new_protected_conditional:Npnn \regex_check_nfa:N #1 { TF }
  { \exp_after:wN \regex_check_nfa_aux:Nw #1 \q_stop }
\cs_new:Npn \regex_check_nfa_aux:Nw #1 #2 \q_stop
  {
    \if_meaning:w \regex_nfa:Nw #1
      \prg_return_true:
    \else:
      \msg_error:nnx { regex } { not-nfa } { \token_to_str:N #1 }
      \prg_return_false:
    \fi:
  }
\msg_new:nnn { regex } { not-nfa }
  {
    I~was~expecting~a~regular~expression~variable.\\
    Instead,~I~got~#1.
  }
```

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

`\regex_use:N` No error-checking.

```
\cs_new_protected_nopar:Npn \regex_use:N #1
  { \exp_after:wN \use_none:nn #1 }
```

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

2.6.2 Generic auxiliary functions

`\regex_user_aux:n` This is an auxiliary used by most user functions. The first and second arguments control whether we should track submatches, and whether we track one or multiple submatches. Everything is done within a group, so that `#1` can perform "unsafe" assignments. Most user functions return a result using `\group_insert_after:N`.

```
\cs_new_protected:Npn \regex_user_aux:n #1
  {
    \group_begin:
      \tl_clear:N \l_regex_every_match_tl
      #1
    \group_end:
  }
```

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

`\regex_return_after_group:` Most of l3regex's work is done within a group. This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not.

```
\cs_new_protected_nopar:Npn \regex_return_after_group:
  {
    \if_bool:N \l_regex_success_bool
      \group_insert_after:N \prg_return_true:
    \else:
      \group_insert_after:N \prg_return_false:
    \fi:
  }
```

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

`\regex_extract_after_group:N` Extract submatches, and store them in the user-given variable after the group has ended.

```
\cs_new_protected_nopar:Npn \regex_extract_after_group:N #1
  {
    \if_bool:N \l_regex_success_bool
      \regex_extract:
      \group_insert_after:N \seq_set_eq:NN
      \group_insert_after:N #1
      \group_insert_after:N \g_regex_submatches_seq
    \fi:
  }
```

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

\regex_count_after_group:N Same procedure as \regex_extract_after_group:N, but simpler since the match counting has already taken place.

```
\cs_new_protected_nopar:Npn \regex_count_after_group:N #1
  {
    \group_insert_after:N \int_set_eq:NN
    \group_insert_after:N #1
    \group_insert_after:N \g_regex_match_count_int
  }
```

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

2.6.3 Matching

\regex_match:nn
\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.

```
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
  {
    \regex_user_aux:n
      {
        \regex_disable_submatches:
        \regex_build:n {#1}
        \regex_match:n {#2}
        \regex_return_after_group:
      }
  }
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \regex_check_nfa:NTF #1
      {
        \regex_user_aux:n
          {
            \regex_disable_submatches:
            \regex_use:N #1
            \regex_match:n {#2}
            \regex_return_after_group:
          }
      }
      { \prg_return_false: }
  }
```

(*End definition for* `\regex_match:nn`. *This function is documented on page* 5.)

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

```
\cs_new_protected:Npn \regex_count:nnN #1#2#3
  {
    \regex_user_aux:n
      {
        \regex_disable_submatches:
```

```
        \int_gzero:N \g_regex_match_count_int
        \tl_set:Nn \l_regex_every_match_tl
          {
            \int_gincr:N \g_regex_match_count_int
            \regex_match_once:
          }
        \regex_build:n {#1}
        \regex_match:n {#2}
        \regex_count_after_group:N #3
      }
  }
\cs_new_protected:Npn \regex_count:NnN #1#2#3
  {
    \regex_check_nfa:NTF #1
      {
        \regex_user_aux:n
          {
            \regex_disable_submatches:
            \int_gzero:N \g_regex_match_count_int
            \tl_set:Nn \l_regex_every_match_tl
              {
                \int_gincr:N \g_regex_match_count_int
                \regex_match_once:
              }
            \regex_use:N #1
            \regex_match:n {#2}
            \regex_count_after_group:N #3
          }
      }
      { }
  }
```

(*End definition for* `\regex_count:nnN`*. This function is documented on page* 5.)

2.6.4 Submatch extraction

`\regex_extract:nnN`
`\regex_extract:NnN`
`\regex_extract:nnN`
`\regex_extract:NnN`

```
\cs_new_protected:Npn \regex_extract:nnN #1#2#3
  {
    \regex_user_aux:n
      {
        \regex_build:n {#1}
        \regex_match:n {#2}
        \regex_extract_after_group:N #3
      }
  }
\prg_new_protected_conditional:Npnn \regex_extract:nnN #1#2#3 { T , F , TF }
  {
    \regex_user_aux:n
      {
```

```
        \regex_build:n {#1}
        \regex_match:n {#2}
        \regex_extract_after_group:N #3
        \regex_return_after_group:
      }
  }
\cs_new_protected:Npn \regex_extract:NnN #1#2#3
  {
    \regex_check_nfa:NTF #1
      {
        \regex_user_aux:n
          {
            \regex_use:N #1
            \regex_match:n {#2}
            \regex_extract_after_group:N #3
          }
      }
      { }
  }
\prg_new_protected_conditional:Npnn \regex_extract:NnN #1#2#3 { T , F , TF }
  {
    \regex_check_nfa:NTF #1
      {
        \regex_user_aux:n
          {
            \regex_use:N #1
            \regex_match:n {#2}
            \regex_extract_after_group:N #3
            \regex_return_after_group:
          }
      }
      { \prg_return_false: }
  }
```

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

2.6.5 Splitting a string by matches of a regex

`\regex_split:nnN`
`\regex_split:NnN`
`\regex_split_aux:`
`\regex_split_after_group:N`

Similarly to `\regex_count:nnN` functions, recurse through matches of the pattern. Then we do something slightly different, extracting submatches. Submatches are not extracted if the pattern matched an empty string at the start of the match attempt (to avoid adding spurious empty items to the resulting sequence).

```
\cs_new_protected:Npn \regex_split:nnN #1#2#3
  {
    \regex_user_aux:n
      {
        \seq_gclear:N \g_regex_split_seq
        \tl_set:Nn \l_regex_every_match_tl { \regex_split_aux: }
        \regex_build:n {#1}
        \regex_match:n {#2}
```

```
        \regex_split_after_group:N #3
      }
  }
\cs_new_protected:Npn \regex_split:NnN #1#2#3
  {
    \regex_check_nfa:NTF #1
      {
        \regex_user_aux:n
          {
            \seq_gclear:N \g_regex_split_seq
            \tl_set:Nn \l_regex_every_match_tl { \regex_split_aux: }
            \regex_use:N #1
            \regex_match:n {#2}
            \regex_split_after_group:N #3
          }
      }
      { }
  }
\cs_new_protected_nopar:Npn \regex_split_aux:
  {
    \int_compare:nNnF \l_regex_start_step_int = \l_regex_success_step_int
      {
        \regex_extract:
        \seq_pop:NN \g_regex_submatches_seq \l_regex_tmpa_tl
        \regex_extract_aux:nTF {0}
          {
            \seq_gput_left:Nx \g_regex_submatches_seq
              {
                \str_from_to:Nnn \l_regex_query_str
                  { \int_use:N \l_regex_start_step_int }
                  { \l_regex_tmpa_tl }
              }
          }
          { \msg_error:nn { regex } { internal } }
        \seq_gconcat:NNN \g_regex_split_seq
          \g_regex_split_seq \g_regex_submatches_seq
      }
    \regex_match_once:
  }
\cs_new_protected_nopar:Npn \regex_split_after_group:N #1
  {
    \int_compare:nNnTF \l_regex_start_step_int = \l_regex_current_step_int
      {
        \bool_if:NF \l_regex_success_empty_bool
          { \seq_gput_right:Nn \g_regex_split_seq { } }
      }
      {
        \seq_gput_right:Nx \g_regex_split_seq
          {
            \str_from_to:Nnn \l_regex_query_str
```

```
              { \int_use:N \l_regex_start_step_int }
              { \int_use:N \l_regex_current_step_int }
          }
      }
    \group_insert_after:N \seq_set_eq:NN
    \group_insert_after:N #1
    \group_insert_after:N \g_regex_split_seq
  }
```

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

2.6.6 String replacement

`\regex_replacement:n`
`\regex_replacement_loop:N`
`\regex_replacement_normal:N`

```
\cs_new_protected:Npn \regex_replacement:n #1
  {
    \str_aux_escape:NNNn
      \prg_do_nothing:
      \regex_replacement_escaped:N
      \regex_replacement_raw:N
      {#1}
    \tl_set_eq:NN \l_regex_replacement_tl \g_str_result_tl
    \tl_set:Nx \l_regex_replacement_tl
      { \l_regex_replacement_tl \prg_do_nothing: }
  }
\cs_new_nopar:Npn \regex_replacement_raw:N { \exp_not:N \prg_do_nothing: }
\cs_new_nopar:Npn \regex_replacement_escaped:N #1
  {
    \if_num:w 9 < 1 #1 \exp_stop_f:
      \exp_not:N \regex_replacement_escaped_ii:nN {#1}
    \else:
      #1
    \fi:
  }
\cs_new_nopar:Npn \regex_replacement_escaped_ii:nN #1#2
  {
    \regex_token_if_other_digit:NTF #2
      { \regex_replacement_escaped_ii:nN {#1#2} }
      { \exp_not:N \seq_item:Nn \exp_not:N \g_regex_submatches_seq {#1} #2 }
  }
\prg_new_conditional:Npnn \regex_token_if_other_digit:N #1 { TF }
  {
    \if_num:w 9 < 1 \exp_not:N #1 \exp_stop_f:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

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

\regex_replace_after_group:N

```
\cs_new_protected_nopar:Npn \regex_replace_after_group:N #1
  {
    \group_insert_after:N \tl_set_eq:NN
    \group_insert_after:N #1
    \group_insert_after:N \g_regex_replaced_str
  }
```

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

\regex_replace_once:nnN
\regex_replace_once:NnN
\regex_replace_once:nnN
\regex_replace_once:NnN
\regex_replace_once_aux:Nn

```
\cs_new_protected:Npn \regex_replace_once_aux:Nn #1#2
  {
    \group_begin:
      \regex_replace_after_group:N #1
      \tl_clear:N \l_regex_every_match_tl
      #2
      \exp_args:No \regex_match:n {#1}
      \regex_extract:
      \regex_extract_aux:nTF {0}
        {
          \tl_gset:Nx \g_regex_replaced_str
            {
              \str_from_to:Nnn \l_regex_query_str {0} { \l_regex_tmpa_tl }
              \l_regex_replacement_tl
              \str_from_to:Nnn \l_regex_query_str
                { \l_regex_tmpb_tl } { \c_max_int }
            }
        }
        { \tl_gset_eq:NN \g_regex_replaced_str \l_regex_query_str }
    \group_end:
  }
\cs_new_protected:Npn \regex_replace_once:nnN #1#2#3
  {
    \regex_replace_once_aux:Nn #3
      {
        \regex_build:n {#1}
        \regex_replacement:n {#2}
      }
  }
\cs_new_protected:Npn \regex_replace_once:NnN #1#2#3
  {
    \regex_replace_once_aux:Nn #3
      {
        \regex_use:N #1
        \regex_replacement:n {#2}
      }
  }
\prg_new_protected_conditional:Npnn \regex_replace_once:nnN #1#2#3 {T,F,TF}
  {
```

```
    \regex_replace_once_aux:Nn #3
      {
        \regex_build:n {#1}
        \regex_replacement:n {#2}
        \regex_return_after_group:
      }
  }
\prg_new_protected_conditional:Npnn \regex_replace_once:NnN #1#2#3 {T,F,TF}
  {
    \regex_replace_once_aux:Nn #3
      {
        \regex_use:N #1
        \regex_replacement:n {#2}
        \regex_return_after_group:
      }
  }
```

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

`\regex_replace_all:nnN`
`\regex_replace_all:NnN`
`\regex_replace_all_aux:Nn`
`\regex_replace_all_aux:`

```
\cs_new_protected:Npn \regex_replace_all_aux:Nn #1#2
  {
    \group_begin:
      \regex_replace_after_group:N #1
      \tl_set:Nn \l_regex_every_match_tl { \regex_replace_all_aux: }
      \tl_gclear:N \g_regex_replaced_str
      #2
      \exp_args:No \regex_match:n {#1}
      \tl_gput_right:Nx \g_regex_replaced_str
        {
          \str_from_to:Nnn \l_regex_query_str
            { \int_use:N \l_regex_start_step_int }
            { \c_max_int }
        }
    \group_end:
  }
\cs_new_protected_nopar:Npn \regex_replace_all_aux:
  {
    \regex_extract:
    \regex_extract_aux:nTF {0}
      {
        \tl_gput_right:Nx \g_regex_replaced_str
          {
            \str_from_to:Nnn \l_regex_query_str
              { \int_use:N \l_regex_start_step_int }
              { \l_regex_tmpa_tl }
            \l_regex_replacement_tl
          }
      }
      { \msg_error:nn { regex } { internal } }
    \regex_match_once:
```

```
  }
\cs_new_protected:Npn \regex_replace_all:nnN #1#2#3
  {
    \regex_replace_all_aux:Nn #3
      {
        \regex_build:n {#1}
        \regex_replacement:n {#2}
      }
  }
\cs_new_protected:Npn \regex_replace_all:NnN #1#2#3
  {
    \regex_replace_all_aux:Nn #3
      {
        \regex_use:N #1
        \regex_replacement:n {#2}
      }
  }
```

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

⟨/package⟩