

l3regex 模块

T_EX 中的正则表达式

L^AT_EX 项目组* 2024 年 1 月 04 日 发布

张泓知 2024 年 1 月 14 日 【译】

目 录

1	正则表达式的语法	4
1.1	正则表达式示例	4
1.2	正则表达式中的字符	5
1.3	字符类	6
1.4	结构：替代、分组、重复	7
1.5	匹配确切的记号	9
1.6	杂项	10
2	替换文本的语法	11
3	预编译正则表达式	12
4	匹配	13
5	子匹配提取	15
6	替换	16
7	临时用的正则表达式	18
8	错误、缺陷、未来工作和其他可能性	19

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

9	l3regex implementation	21
9.1	Plan of attack	21
9.2	Helpers	23
9.2.1	Constants and variables	26
9.2.2	Testing characters	27
9.2.3	Internal auxiliaries	27
9.2.4	Character property tests	32
9.2.5	Simple character escape	34
9.3	Compiling	41
9.3.1	Variables used when compiling	42
9.3.2	Generic helpers used when compiling	44
9.3.3	Mode	45
9.3.4	Framework	48
9.3.5	Quantifiers	53
9.3.6	Raw characters	56
9.3.7	Character properties	58
9.3.8	Anchoring and simple assertions	59
9.3.9	Character classes	60
9.3.10	Groups and alternations	65
9.3.11	Catcodes and csnames	68
9.3.12	Raw token lists with \u	73
9.3.13	Other	78
9.3.14	Showing regexes	78
9.4	Building	87
9.4.1	Variables used while building	87
9.4.2	Framework	88
9.4.3	Helpers for building an NFA	92
9.4.4	Building classes	93
9.4.5	Building groups	95
9.4.6	Others	101
9.5	Matching	103
9.5.1	Variables used when matching	104
9.5.2	Matching: framework	107
9.5.3	Using states of the NFA	111
9.5.4	Actions when matching	112
9.6	Replacement	116

9.6.1	Variables and helpers used in replacement	116
9.6.2	Query and brace balance	118
9.6.3	Framework	119
9.6.4	Submatches	124
9.6.5	Csnames in replacement	125
9.6.6	Characters in replacement	127
9.6.7	An error	132
9.7	User functions	132
9.7.1	Variables and helpers for user functions	137
9.7.2	Matching	139
9.7.3	Extracting submatches	140
9.7.4	Replacement	147
9.7.5	Peeking ahead	150
9.8	Messages	158
9.9	Code for tracing	166

索引	167
-----------	------------

`l3regex` 模块提供正则表达式测试、子匹配提取、分割和替换，全部作用于记号列表。正则表达式的语法主要是 PCRE 语法的子集（并且非常接近 POSIX），但由于 \TeX 操作的是记号而不是字符，因此有一些附加的功能。由于性能原因，仅实现了有限的功能集。特别地，不支持反向引用。

让我们给出一些示例。在

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

记号列表变量 `\l_my_tl` 存储文本 “This cat.”，其中第一个 “at” 被替换为 “is”。一个更复杂的例子是用于强调每个单词并在其后添加逗号的模式：

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

`\w` 序列表示任何 “word” 字符，而 `+` 表示 `\w` 序列应重复尽可能多次（至少一次），因此匹配输入记号列表中的一个单词。在替换文本中，`\0` 表示完全匹配（这里是一个单词）。通过 `\c{emph}` 插入 `\emph` 命令，并将其参数 `\0` 放在大括号 `\cB\{` 和 `\cE\}` 之间。

如果要多次使用正则表达式，可以编译一次，并使用 `\regex_set:Nn` 将其存储在正则表达式变量中。例如，

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

在 `\l_foo_regex` 中存储一个正则表达式，该表达式匹配环境的起始标记：`\begin`，后跟一个起始组记号（`\cB.`），然后是任意数量的既不是起始组记号也不是结束组记号字符记号的记号（`\c[~BE].*`），最后是结束组记号（`\cE.`）。如下一节所述，括号 “捕获” 了 `\c[~BE].*` 的结果，从而在进行替换时让我们能够访问环境的名称。

1 正则表达式的语法

1.1 正则表达式示例

我们从一些示例开始，并鼓励读者对这些正则表达式应用 `\regex_show:n`。

- `Cat` 匹配以此方式大写的单词 “Cat”，但也匹配单词 “Cattle” 的开头：使用 `\bCat\b` 仅匹配完整的单词。
- `[abc]` 匹配字母 “a”、“b”、“c” 中的一个；模式 `(a|b|c)` 匹配相同的三个可能的字母（但请参阅下面的子匹配的讨论）。
- `[A-Za-z]*` 匹配任意数量（由于量词 `*`）的拉丁字母（没有重音）。

- `\c{[A-Za-z]*}` 匹配由拉丁字母组成的控制序列。
- `_[^_]*_` 匹配下划线，除下划线外的任意数量字符，和另一个下划线；这等效于 `_.*?_`，其中 `.` 匹配任意字符，懒惰量词 `*?` 表示尽可能匹配少的字符，从而避免匹配下划线。
- `[\+|-]?[d+]` 匹配带有最多一个符号的显式整数。
- `[\+|-|_]*[d+]_*` 匹配带有任意数量 `+` 和 `-` 符号的显式整数，允许空格，除了在尾数内，且被空格包围。
- `[\+|-|_]*(\d+|\d*\.\d+)_*` 匹配显式整数或小数；使用 `[.,]` 而不是 `\.` 允许逗号作为小数点。
- `[\+|-|_]*(\d+|\d*\.\d+)_*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)_*` 匹配任何 T_EX 知道的显式尺寸，其中 `(?i)` 表示对待小写和大写字母相同。
- `[\+|-|_]*((?i)nan|inf|(\d+|\d*\.\d+)_*(e[\+|-|_]*[d+])?)_*` 匹配显式浮点数或特殊值 `nan` 和 `inf`（允许符号和空格）。
- `[\+|-|_]*(\d+|\cC.)_*` 匹配显式整数或控制序列（不检查是否是整数变量）。
- `\G.*?K` 在正则表达式开头匹配并丢弃（由于 `\K`）前一个匹配的结尾（`\G`），以及由正则表达式的其余部分匹配的内容；在 `\regex_replace_all:nnN` 中很有用，当目标是以比 `\regex_extract_all:nnN` 更精细的方式提取匹配项或子匹配项时。

尽管不可能让正则表达式仅匹配整数表达式，但是

```
[ \+|-|-( ) * \d + \) * ( [ \+|-|/ ] [ \+|-|-( ) * \d + \) * ) *
```

匹配所有有效的整数表达式（仅由显式整数制成）。应该跟随进一步的测试。

1.2 正则表达式中的字符

大多数字符与它们自己完全匹配，具有任意类别码。一些字符是特殊的，必须用反斜杠转义（例如，`*` 匹配星号字符）。一些反斜杠-字母形式的转义序列也具有特殊含义（例如 `\d` 匹配任何数字）。一般规则是，

- 每个字母数字字符（`A-Z`、`a-z`、`0-9`）与其自身完全匹配，不应转义，因为 `\A`、`\B` 等具有特殊含义；
- 非字母数字可打印 ASCII 字符应始终（并且应该）转义：其中许多字符具有特殊含义（例如，使用 `\(`、`\)`、`\?`、`\.`、`\^`）；

- 空格应始终转义（即使在字符类中）；
- 其他任何字符可以转义，也可以不转义，都没有任何效果：两个版本都完全匹配该字符。

请注意，这些规则与许多非字母数字字符在正常类别码下很难输入到 $\text{T}_{\text{E}}\text{X}$ 中的事实相吻合。例如，`\abc\%` 匹配字符 `\abc%`（具有任意类别码），但不匹配控制序列 `\abc` 后跟一个百分号字符。可以使用 `\c{<regex>}` 语法（见下文）来匹配控制序列。

任何特殊字符出现在其特殊行为无法应用的地方时，将匹配自身（例如，在字符串开头出现的量词），并提出警告。

字符。

`\x{hh...}` 具有十六进制代码 `hh...` 的字符

`\xhh` 具有十六进制代码 `hh` 的字符

`\a` 警报（十六进制 07）

`\e` 转义（十六进制 1B）

`\f` 进纸换页（十六进制 0C）

`\n` 换行（十六进制 0A）

`\r` 回车（十六进制 0D）

`\t` 水平制表符（十六进制 09）

1.3 字符类

字符属性。

- 单个句点匹配任何记号。

`\d` 任何十进制数字。

`\h` 任何水平空白字符，等同于 `[\ \^~I]`：空格和制表符。

`\s` 任何空格字符，等同于 `[\ \^~I\^~J\^~L\^~M]`

`\v` 任何垂直空白字符，等同于 `[\^~J\^~K\^~L\^~M]`。注意 `\^~K` 是垂直空白，但不是空格，以兼容 Perl。

`\w` 任何单词字符，即字母数字和下划线，等同于明确的类 `[A-Za-z0-9_]`

`\D` 任何非 `\d` 匹配的记号。

`\H` 任何非 `\h` 匹配的记号。

`\N` 任何非 `\n` 字符（十六进制 0A）的记号。

`\S` 任何非 `\s` 匹配的记号。

`\V` 任何非 `\v` 匹配的记号。

`\W` 任何非 `\w` 匹配的记号。

其中，`.`、`\D`、`\H`、`\N`、`\S`、`\V` 和 `\W` 匹配任意控制序列。
字符类精确匹配主题中的一个记号。

`[...]` 正向字符类。匹配指定的任何记号。

`[^...]` 负向字符类。匹配指定字符之外的任何记号。

`x-y` 在字符类中，表示一个范围（可以与转义字符一起使用）。

`[:<name>:]` 在字符类中（另一组括号），表示 POSIX 字符类 `<name>`，可以是 `alnum`、`alpha`、`ascii`、`blank`、`cntrl`、`digit`、`graph`、`lower`、`print`、`punct`、`space`、`upper`、`word` 或 `xdigit`。

`[:^<name>:]` 负向 POSIX 字符类。

例如，`[a-oq-z\cC.]` 匹配任何小写拉丁字母，除了 `p`，以及控制序列（见下文对 `\c` 的描述）。

在字符类中，只有 `[`、`^`、`-`、`]`、`\` 和空格是特殊的，应该被转义。其他非字母数字字符仍可被转义而不会受到影响。在字符类中支持匹配单个字符的任何转义序列（`\d`、`\D`、等）。如果第一个字符是 `^`，则字符类的含义被反转；在范围中的任何其他位置出现的 `^` 都不是特殊的。如果第一个字符（可能是在一个领先的 `^` 之后）是 `]`，则不需要转义，因为在那里结束范围将使其为空。字符的范围可以用 `-` 表示，例如，`[\D 0-5]` 和 `[^6-9]` 是等价的。

1.4 结构：替代、分组、重复

量词（重复）。

`? 0 或 1`，贪婪模式。

`?? 0 或 1`，懒惰模式。

`* 0 或多次`，贪婪模式。

`*? 0 或多次`，懒惰模式。

+ 1 或多次，贪婪模式。

+? 1 或多次，懒惰模式。

{*n*} 恰好 *n* 次。

{*n*,} *n* 次或更多，贪婪模式。

{*n*,}? *n* 次或更多，懒惰模式。

{*n*,*m*} 至少 *n* 次，最多 *m* 次，贪婪模式。

{*n*,*m*}? 至少 *n* 次，最多 *m* 次，懒惰模式。

对于贪婪量词，正则表达式代码将首先尝试尽可能多的重复匹配，而对于懒惰量词，它将首先尝试尽可能少的重复匹配。

替代和捕获分组。

A|B|C A、B 或 C 中的任意一个，首先尝试匹配 A。

(...) 捕获分组。

(?:...) 非捕获分组。

(?|...) 非捕获分组，每个替代中都重置捕获分组编号。下一个分组将用第一个未使用的分组编号进行编号。

捕获分组是提取关于匹配的信息的一种方法。带括号的组按照其打开括号的顺序编号，从 1 开始。可以使用例如 `\regex_extract_once:nnNTF` 将这些组的内容提取并存储在一系列记号列表中。

`\K` 转义序列将匹配的起始位置重置为记号列表中的当前位置。这仅影响作为完整匹配报告的内容。例如，

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

的结果是 `\l_foo_seq` 包含项 {1} 和 {a}：真正的匹配是 {a1} 和 {aa}，但它们被使用 `\K` 截断。`\K` 命令不影响捕获分组，例如，

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

的结果是 `\l_foo_seq` 包含项 {c3} 和 {bc}：真正的匹配是 {acbc3}，其中第一个子匹配是 {bc}，但 `\K` 重置匹配的起始位置为它出现的最后位置。

1.5 匹配确切的记号

`\c` 转义序列允许测试记号的类别码，并匹配控制序列。每个字符类别由单个大写字母表示：

- C 表示控制序列；
- B 表示开始组记号；
- E 表示结束组记号；
- M 表示数学换位符；
- T 表示对齐制表符记号；
- P 表示宏参数记号；
- U 表示上标记号（上）；
- D 表示下标记号（下）；
- S 表示空格；
- L 表示字母；
- O 表示其他；以及
- A 表示活动字符。

`\c` 转义序列的使用如下。

`\c{<regex>}` 一个控制序列，其控制序列名称匹配 *<regex>*，锚定在开头和结尾，因此 `\c{begin}` 精确匹配 `\begin`，而不匹配其他任何内容。

`\cX` 适用于下一个对象，可以是字符、转义字符序列（如 `\x{0A}`）、字符类或组，并强制该对象只匹配类别为 X 的记号（其中 X 为 CBEMTPUDSLOA 中的任意一个）。例如，`\cL[A-Z\d]` 匹配大写字母和类别为字母的数字，`\cC.` 匹配任何控制序列，`\cO(abc)` 匹配类别为其他的 `abc`。¹

`\c[XYZ]` 适用于下一个对象，并强制它只匹配类别为 X、Y 或 Z 的记号（其中每个都是 CBEMTPUDSLOA 中的任意一个）。例如，`\c[LSO](..)` 匹配两个类别为字母、空格或其他记号。

`\c[~XYZ]` 适用于下一个对象，并阻止它匹配类别为 X、Y 或 Z 的任何记号（其中每个都是 CBEMTPUDSLOA 中的任意一个）。例如，`\c[~O]\d` 匹配类别为其他的数字。

¹最后一个示例还捕获了“abc”作为正则表达式组；要避免这种情况，请使用非捕获组 `\cO(?:abc)`。

类别码测试可用于类中；例如，`[\c0\d \c[L0][A-F]]` 匹配 \TeX 认为的十六进制数字，即类别为其他的数字，或类别为字母或其他的大写字母。在受到类别码测试影响的组内，嵌套测试可以覆盖外部测试；例如，`\cL(ab\c0*cd)` 匹配所有字符都是字母类别的 `ab*cd`，除了 `*` 的类别是其他。

`\u` 转义序列允许将记号列表的内容直接插入正则表达式或替换中，无需转义特殊字符。即，`\u{\var name}` 精确匹配变量 `\{\var name\}` 的内容（包括字符码和类别码），这是通过在编译正则表达式时应用 `\exp_not:v {\var name}` 获得的。在 `\c{...}` 控制序列匹配中，`\u` 转义序列仅展开其参数一次，实际上执行 `\tl_to_str:v`。支持量词。

`\ur` 转义序列允许将 `regex` 变量的内容插入较大的正则表达式。例如，`A\ur{l_tmpa_regex}D` 匹配由匹配正则表达式 `\l_tmpa_regex` 的内容分隔的记号 `A` 和 `D`。这相当于将非捕获组包围在 `\l_tmpa_regex` 周围，并且 `\l_tmpa_regex` 中包含的任何组都会转换为非捕获组。支持量词。

例如，如果 `\l_tmpa_regex` 的值为 `B|C`，那么 `A\ur{l_tmpa_regex}D` 等效于 `A(?:B|C)D`（匹配 `ABD` 或 `ACD`），而不是 `AB|CD`（匹配 `AB` 或 `CD`）。要获得后者的效果，最简单的方法是直接使用 \TeX 的展开机制：如果 `\l_mymodule_BC_tl` 包含 `B|C`，那么以下两行显示相同的结果：

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

1.6 杂项

锚点和简单断言。

`\b` 单词边界：前一个记号由 `\w` 匹配，下一个由 `\W` 匹配；或者相反。为此，将记号列表的两端视为 `\W`。

`\B` 非单词边界：在两个 `\w` 记号或两个 `\W` 记号之间（包括边界）。

`^` 或 `\A` 主题记号列表的开始。

`$`，`\Z` 或 `\z` 主题记号列表的结尾。

`\G` 当前匹配的开始。仅在多次匹配的情况下与 `^` 不同：例如，`\regex_count:nnN { \G a } { aab}` 得到 2，但将 `\G` 替换为 `^` 将导致 `\l_tmpa_int` 包含值 1。

选项 `(?i)` 使匹配不区分大小写（将 `A-Z` 和 `a-z` 视为等效，尚不支持 Unicode 大小写转换）。这适用于它所在的组直到结束，并可使用 `(?-i)` 还原。例如，在 `(?i)(a(?-i)b|c)d` 中，字母 `a` 和 `d` 受到 `i` 选项的影响。范围和类中的字符被单独

影响：(?i) [\?-B] 等效于 [\?@ABab]（与较大的类 [\?-b] 不同），(?i) [^aeiou] 匹配任何不是元音的字符。i 选项对于 \c{...}、\u{...}、字符属性或字符类等没有任何影响，例如在 (?i) \u{1_foo_tl} \d\d[[:lower:]] 中根本不起作用。

2 替换文本的语法

正则表达式中描述的大多数功能在替换文本中没有意义。反斜杠引入各种特殊结构，下面将进一步描述：

- \0 表示整个匹配；
- \1 表示由第一个（捕获）组 (...) 匹配的子匹配；类似地，\2、...、\9 和 \g{<number>}；
- _ 插入一个空格（未转义的情况下忽略空格）；
- \a、\e、\f、\n、\r、\t、\xhh、\x{hhh} 对应于正则表达式中的单个字符；
- \c{<cs name>} 插入一个控制序列；
- \c{<category>}<character>（见下文）；
- \u{<tl var name>} 将变量 <tl var> 的内容直接插入替换文本，无需转义特殊字符。

除反斜杠和空格之外的字符直接插入结果中（但由于首先将替换文本转换为字符串，因此也应转义对于 T_EX 而言是特殊的字符，例如使用 \#）。非字母数字字符始终可以安全地使用反斜杠进行转义。例如，

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

导致 \l_my_tl 包含 H(ell--el)(o,--o) w(or--o)(ld--l)!

子匹配的编号按照捕获组的开放括号在要匹配的正则表达式中出现的顺序进行。如果捕获组少于 n 个，或者捕获组出现在未用于匹配的替代方案中，则第 n 个子匹配为空。如果捕获组在匹配期间多次匹配（由于量词），则在替换文本中仅使用最后一次匹配。子匹配始终保持与原始记号列表相同的类别码。

默认情况下，替换插入的字符的类别码由替换时的主要类别码制度确定，有两个例外：

- 通过 _、\x20 或 \x{20} 插入的空格字符（字符码为 32）无论当前的类别码制度如何，其类别码始终为 10；

- 如果类别码为 0（转义）、5（换行符）、9（忽略）、14（注释）或 15（无效），则替换时将其替换为 12（其他）。

转义序列 `\c` 允许插入具有任意类别码的字符，以及控制序列。

`\cX(...)` 产生类别为 `X` 的字符“...”，其中 `X` 必须是正则表达式中的 CBEMTPUDSLOA 之一。

括号对于单个字符（可能是转义序列）是可选的。嵌套时，应用最内层的类别码，例如 `\cL(Hello\cS\ world)!` 会产生标准类别码的此文本。

`\c{<text>}` 插入 `csname` 为 `<text>` 的控制序列。`<text>` 可能包含对子匹配 `\0`、`\1` 等的引用，如下例所示。

转义序列 `\u{<var name>}` 允许将变量 `<var name>` 的内容直接插入替换文本，更容易控制类别码。在 `\c{...}` 和 `\u{...}` 结构中嵌套时，`\u` 和 `\c` 转义序列执行 `\tl_to_str:v`，即提取控制序列的值并将其转换为字符串。匹配还可在 `\c` 和 `\u` 的参数中使用。例如，

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

结果为 `\l_my_tl` 包含 `first,\emph{second},first,first`。

正则表达式替换还是一个方便的方法，用于生成具有任意类别码的记号列表。例如

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

导致 `\l_tmpa_tl` 包含类别码为 7（上标）的百分号字符和活动中划线字符。

3 预编译正则表达式

如果要多次使用正则表达式，最好是编译一次，而不是每次使用正则表达式时都编译。编译后的正则表达式存储在一个变量中。所有 `l3regex` 模块的函数都可以将其正则表达式参数作为显式字符串或编译后的正则表达式给出。

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

New: 2017-05-26

创建一个新的 `<regex var>`，如果名称已被使用则引发错误。该声明是全局的。初始时，`<regex var>` 被设置为永远不匹配。

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

`\regex_gset:Nn`

New: 2017-05-26

在 `<regex var>` 中存储 `<正则表达式>` 的编译版本。对于 `\regex_set:Nn`, 赋值是局部的; 对于 `\regex_gset:Nn`, 是全局的。例如, 此函数可以用作

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

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

New: 2017-05-26

创建一个新的常量 `<regex var>`, 如果名称已被使用则引发错误。`<regex var>` 的值被全局设置为 `<正则表达式>` 的编译版本。

`\regex_show:N` `\regex_show:n {<regex>}`

`\regex_show:n` `\regex_log:n {<regex>}`

`\regex_log:N`

`\regex_log:n`

New: 2021-04-26

Updated: 2021-04-29

在终端显示或写入日志文件 (分别) `!3regex` 如何解释 `<regex>`。例如, `\regex_show:n { \A X|Y }` 显示

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

表明锚 `\A` 仅适用于第一分支: 第二分支未锚定到匹配的开始。

4 匹配

所有正则表达式函数都有 `:n` 和 `:N` 两种变体。前者需要一个“标准”正则表达式, 而后者需要由 `\regex_set:Nn` 生成的编译表达式。

`\regex_match:nnTF` `\regex_match:nnTF {<regex>} {<token list>} {<true code>} {<false code>}`

`\regex_match:nVTF`

`\regex_match:NnTF`

`\regex_match:NVTF`

New: 2017-05-26

测试 `<正则表达式>` 是否与 `<token list>` 的任何部分匹配。例如,

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

在输入流中留下 TRUE 然后 FALSE。

`\regex_count:nnN` `\regex_count:nnN {<regex>} {<token list>} <int var>`

`\regex_count:nVN` 在当前 \TeX 组级别内, 将 `<int var>` 设置为 `<正则表达式>` 在 `<token list>` 中出现的次数。搜索从找到最左边最长的匹配开始, 尊重贪婪和懒惰 (非贪婪) 运算符。然后, `\regex_count:NnN` 搜索从前一匹配的最后一个字符之后的字符开始, 直到达到 `token list` 的末尾。在正则表达式可以匹配空 `token list` 的情况下, 防止无限循环: 在每对字符之间计数一次匹配。例如,

New: 2017-05-26

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

的结果是 `\l_foo_int` 的值为 5。

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

`\regex_match_case:nnTF` {

```
{<regex1>} {<code case1>}
{<regex2>} {<code case2>}
...
{<regexn>} {<code casen>}
} {<token list>}
{<true code>} {<false code>}
```

New: 2022-01-10

确定在 `<token list>` 中的最早位置哪个 `<正则表达式>` 匹配, 并在输入流中留下相应的 `<codei>`, 后跟 `<true code>`。如果多个 `<regex>` 在同一点开始匹配, 则选择列表中的第一个, 并丢弃其他的。如果没有任何 `<regex>` 匹配, 则在输入流中留下 `<false code>`。每个 `<regex>` 都可以是正则表达式变量或显式正则表达式。

具体而言, 对于 `<token list>` 中的每个起始位置, 依次搜索 `<regex>`。如果其中一个匹配, 则使用相应的 `<code>`, 并丢弃其他一切; 如果在给定位置没有 `<regex>` 匹配, 则尝试下一个起始位置。如果在 `<token list>` 的任何位置都没有任何 `<regex>` 匹配, 则在输入流中什么都不留下。请注意, 这与嵌套的 `\regex_match:nnTF` 语句不同, 因为在每个位置尝试匹配所有 `<regex>`, 而不是在移动到 `<regex2>` 之前尝试匹配 `<regex1>`。

5 子匹配提取

<code>\regex_extract_once:nnN</code> <code>\regex_extract_once:nVN</code> <code>\regex_extract_once:nnN\overline{TF}</code> <code>\regex_extract_once:nVN\overline{TF}</code> <code>\regex_extract_once:NnN</code> <code>\regex_extract_once:NVN</code> <code>\regex_extract_once:NnN\overline{TF}</code> <code>\regex_extract_once:NVN\overline{TF}</code>	<code>\regex_extract_once:nnN {<regex>} {<token list>} <seq var></code> <code>\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code> 在 <code><token list></code> 中找到 <code><正则表达式></code> 的第一个匹配项。如果存在匹配项，将匹配项存储为 <code><seq var></code> 的第一项，其余项是捕获组的内容，按其开括号的顺序。局部赋值给 <code><seq var></code> 。如果没有匹配项，则清除 <code><seq var></code> 。测试版本如果找到匹配项，则将 <code><true code></code> 插入输入流，否则插入 <code><false code></code> 。
--	--

New: 2017-05-26

例如，假设您键入

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

那么正则表达式(在开始处用 `\A` 锚定, 在结束处用 `\Z` 锚定)必须匹配整个 token list。第一个捕获组, `(La)?`, 匹配 `La`, 第二个捕获组, `(!*)`, 匹配 `!!!`。因此, `\l_foo_seq` 的结果包含项 `{LaTeX!!!}`, `{La}` 和 `{!!!}`, 并在输入流中留下 `true` 分支。注意, `\l_foo_seq` 的第 n 项, 使用 `\seq_item:Nn` 获取, 对应于函数 `\regex_replace_once:nnN` 等中编号为 $(n - 1)$ 的子匹配。

<code>\regex_extract_all:nnN</code> <code>\regex_extract_all:nVN</code> <code>\regex_extract_all:nnN\overline{TF}</code> <code>\regex_extract_all:nVN\overline{TF}</code> <code>\regex_extract_all:NnN</code> <code>\regex_extract_all:NVN</code> <code>\regex_extract_all:NnN\overline{TF}</code> <code>\regex_extract_all:NVN\overline{TF}</code>	<code>\regex_extract_all:nnN {<regex>} {<token list>} <seq var></code> <code>\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code> 在 <code><token list></code> 中找到 <code><正则表达式></code> 的所有匹配项, 并将所有子匹配信息存储在一个序列中(连接多个 <code>\regex_extract_once:nnN</code> 调用的结果)。局部赋值给 <code><seq var></code> 。如果没有匹配项, 则清除 <code><seq var></code> 。测试版本如果找到匹配项, 则将 <code><true code></code> 插入输入流, 否则插入 <code><false code></code> 。例如, 假设您键入
--	---

New: 2017-05-26

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

那么正则表达式匹配两次, 生成的序列包含两个项 `{Hello}` 和 `{world}`, 并在输入流中留下 `true` 分支。

<code>\regex_split:nnN</code>	<code>\regex_split:nnN {<regular expression>} {<token list>} <seq var></code>
<code>\regex_split:nVN</code>	<code>\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<true code>}</code>
<code>\regex_split:nnNTF</code>	<code>{<false code>}</code>
<code>\regex_split:nVNTF</code>	将 <code><token list></code> 拆分为一系列部分, 由 <code><正则表达式></code> 的匹配项分隔。如果 <code><正则表达式></code> 具有捕获组, 则它们匹配的 token lists 也存储为序列的项。局部赋值给 <code><seq var></code> 。
<code>\regex_split:NnN</code>	如果找不到匹配项, 生成的 <code><seq var></code> 以 <code><token list></code> 为其唯一项。如果 <code><正则表达式></code> 匹配空 token list, 则将 <code><token list></code> 拆分为单个 token。测试版本如果找到匹配项, 则将 <code><true code></code> 插入输入流, 否则插入 <code><false code></code> 。例如, 在
<code>\regex_split:NnNTF</code>	
<code>\regex_split:NVNTF</code>	

New: 2017-05-26

```

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

```

之后, 序列 `\l_path_seq` 包含项 `{the}`, `{path}`, `{for}`, `{this}` 和 `{file.tex}`, 并在输入流中留下 `true` 分支。

6 替换

<code>\regex_replace_once:nnN</code>	<code>\regex_replace_once:nnN {<regular expression>} {<replacement>} <tl var></code>
<code>\regex_replace_once:nVN</code>	<code>\regex_replace_once:nnNTF {<regular expression>} {<replacement>} <tl var> {<true code>} {<false code>}</code>
<code>\regex_replace_once:nnNTF</code>	
<code>\regex_replace_once:nVNNTF</code>	在 <code><tl var></code> 的内容中搜索 <code><正则表达式></code> , 并用 <code><替换></code> 替换第一个匹配项。在 <code><替换></code> 中, <code>\0</code> 代表完整匹配, <code>\1</code> 代表第一个捕获组的内容, <code>\2</code> 代表第二个, 以此类推。结果局部赋值给 <code><tl var></code> 。
<code>\regex_replace_once:NnN</code>	
<code>\regex_replace_once:NVN</code>	
<code>\regex_replace_once:NnNTF</code>	
<code>\regex_replace_once:NVNNTF</code>	

New: 2017-05-26

<code>\regex_replace_all:nnN</code>	<code>\regex_replace_all:nnN {<regular expression>} {<replacement>} <tl var></code>
<code>\regex_replace_all:nVN</code>	<code>\regex_replace_all:nnNTF {<regular expression>} {<replacement>} <tl var> {<true code>} {<false code>}</code>
<code>\regex_replace_all:nnNTF</code>	
<code>\regex_replace_all:nVNNTF</code>	将 <code><tl var></code> 的内容中的 <code><正则表达式></code> 的所有匹配项都替换为 <code><替换></code> , 其中 <code>\0</code> 代表完整匹配, <code>\1</code> 代表第一个捕获组的内容, <code>\2</code> 代表第二个, 以此类推。每个匹配项都是独立处理的, 匹配项之间不能重叠。结果局部赋值给 <code><tl var></code> 。
<code>\regex_replace_all:NnN</code>	
<code>\regex_replace_all:NVN</code>	
<code>\regex_replace_all:NnNTF</code>	
<code>\regex_replace_all:NVNNTF</code>	

New: 2017-05-26

```

\regex_replace_case_once:nN \regex_replace_case_once:nNTF
\regex_replace_case_once:nNTF {
    {\langle regex_1 \rangle} {\langle replacement_1 \rangle}
    {\langle regex_2 \rangle} {\langle replacement_2 \rangle}
    ...
    {\langle regex_n \rangle} {\langle replacement_n \rangle}
} \langle tl var \rangle
{\langle true code \rangle} {\langle false code \rangle}

```

New: 2022-01-10

将 $(?|\langle regex_1 \rangle|...|\langle regex_n \rangle)$ 的最早匹配项，用与之匹配的 $\langle replacement \rangle$ 替换，在输入流中留下 $\langle true code \rangle$ 。如果没有 $\langle regex \rangle$ 匹配，则不修改 $\langle tl var \rangle$ ，并在输入流中留下 $\langle false code \rangle$ 。每个 $\langle regex \rangle$ 可以作为 regex 变量或显式正则表达式给出。

具体而言，对于 $\langle token list \rangle$ 中的每个起始位置，按顺序搜索每个 $\langle regex \rangle$ 。如果其中一个匹配，则将其替换为与之对应的 $\langle replacement \rangle$ ，并从紧随此匹配（和替换）的位置开始重新搜索。这相当于使用 $\backslash regex_match_case:nn$ 检查哪个 $\langle regex \rangle$ 匹配，然后用 $\backslash regex_replace_once:nnN$ 执行替换。

```

\regex_replace_case_all:nN \regex_replace_case_all:nNTF
\regex_replace_case_all:nNTF {
    {\langle regex_1 \rangle} {\langle replacement_1 \rangle}
    {\langle regex_2 \rangle} {\langle replacement_2 \rangle}
    ...
    {\langle regex_n \rangle} {\langle replacement_n \rangle}
} \langle tl var \rangle
{\langle true code \rangle} {\langle false code \rangle}

```

New: 2022-01-10

将 $\langle token list \rangle$ 中所有 $\langle regex \rangle$ 的所有匹配项都替换为相应的 $\langle replacement \rangle$ 。每个匹配项都是独立处理的，匹配项之间不能重叠。结果局部赋值给 $\langle tl var \rangle$ ，并根据是否进行了替换留下 $\langle true code \rangle$ 或 $\langle false code \rangle$ 。

具体而言，对于 $\langle token list \rangle$ 中的每个起始位置，按顺序搜索每个 $\langle regex \rangle$ 。如果其中一个匹配，则将其替换为与之对应的 $\langle replacement \rangle$ ，并从紧随此匹配（和替换）的位置开始重新搜索。例如

```

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

```

结果是 \l_tmpa_tl 包含内容 ``Hello' '---[,][_]``world' '---[!]。请特别注意，单词边界断言 $\backslash b$ 在单词开头没有匹配，因为情况 $[A-Za-z]^+$ 在这些位置匹配。要更改这一点，可以简单地交换 $\backslash regex_replace_case_all:nN$ 参数中两个案例的顺序。

7 临时用的正则表达式

$\backslash l_tmpa_regex$ 本地赋值的 Scratch 正则表达式。这些从未被内核代码使用，因此可安全用于任何 \LaTeX 定义的函数。但可能被其他非内核代码覆盖，因此仅用于短期存储。

New: 2017-12-11

$\backslash g_tmpa_regex$ 全局赋值的 Scratch 正则表达式。这些从未被内核代码使用，因此可安全用于任何 \LaTeX 定义的函数。但可能被其他非内核代码覆盖，因此仅用于短期存储。

New: 2017-12-11

8 错误、缺陷、未来工作和其他可能性

现在需要完成以下任务。

- 以更有序的方式重写文档，或许添加一个 BNF？

更多的错误检查即将到来。

- 整理消息的使用。
- 在替换阶段进行更清晰的错误报告。
- 添加跟踪信息。
- 检测尝试使用反向引用和其他未实现语法的情况。
- 测试最大寄存器 `\c_max_register_int` 的情况。
- 弄清楚 `\w` 和类似的匹配是否导致错误。可能更新 `_regex_item_reverse:n`。
- 空 `cs` 应该由 `\c{}` 匹配，而不是由 `\c{csname.?endcsname\s?}` 匹配。

代码改进即将到来。

- 将数组移动，使有用信息从位置 1 开始。
- 仅构建一次 `\c{...}`。
- 在编译正则表达式时，使用左右状态堆栈的数组。
- `_regex_action_free_group:n` 是否仅用于贪婪的 `{n,}` 量词？（我认为不是。）
- 对于 `\u` 和断言的量词。
- 在匹配时，跟踪 `curr_state` 和 `curr_submatches` 的显式堆栈。
- 如果可能的话，在同一线程中重复使用状态时，终止其他子线程。
- 使用数组而不是 `\g_regex_balance_tl` 来构建函数 `_regex_replacement_balance_one_match:n`。
- 减少替代中的 转换数量。
- 优化简单的字符串：使用较少的状态（`abcade` 应该给出两个状态，用于 `abc` 和 `ade`）。[这真的有意义吗？]
- 优化没有替代的组。

- 优化只有一个 `_regex_action_free:n` 的状态。
- 通过直接在状态 2 中插入 `_regex_action_success:` 来优化其使用，而不是有额外的转换。
- 通过插入 `\int_step_...` 函数的使用来优化。
- 组在 `csnames` 的正则表达式中不捕获；优化并记录。
- 更好的锚定、属性和类别码测试的“show”。
- `\K`是否真的需要一个新的状态？
- 在编译时，使用布尔变量 `in_cs` 和较少的魔术数字。
- 与其使用字符代码检查字符是否为特殊字符或字母数字，不如在正则表达式中使用 `\cs_if_exist` 测试检查它是否为特殊字符。

以下功能可能在将来的某个时候实现。

- 一般的先行/后行断言。
- 在外部文件上进行正则表达式匹配。
- 具有先行/后行条件的条件子模式：“如果之后是 [...], 则 [...]”。
- `(*..)` 和 `(?..)` 序列以设置一些选项。
- pdfTeX 的 UTF-8 模式。
- 换行约定尚未完成。特别是，应该有一个选项使 `.` 不匹配换行符。此外，`\A` 应该与 `^` 不同，而 `\Z`、`\z` 和 `$` 应该不同。
- Unicode 属性：`\p{..}` 和 `\P{..}`；`\X` 应该匹配任何“扩展”的 Unicode 序列。这需要操作大量数据，可能使用树状盒子。

下面的 PCRE 或 Perl 的功能可能会或可能不会被实现。

- 使用 `(?C...)` 或其他语法的调用：一些内部代码更改使这成为可能，在替换代码中找到标记时停止正则表达式替换可能很有用；这引发了潜在的 `\regex_break:` 问题，以及在正则表达式代码中从 `\tl_map_break:` 调用的良好处理问题。还提出了正则表达式机制内嵌套调用的问题，这是一个问题，因为 `\fontdimen` 是全局的。
- 条件子模式（不是先行或后行条件的）：这是非正则的，对吧？
- 命名子模式：TeX 程序员迄今为止无需命名宏参数。

下面的 PCRE 或 Perl 的功能肯定不会被实现。

- 反向引用：非正则特性，需要回溯，速度极慢。
- 递归：这是非正则特性。
- 原子分组、贪婪量词：这些工具主要用于修复灾难性回溯，在非回溯算法中是不必要的，并且难以实现。
- 子例程调用：这种语法糖难以包含在非回溯算法中，特别是因为相应的组应该被视为原子的。
- 回溯控制动词：与回溯密切相关。
- `\ddd`，匹配八进制代码为 `ddd` 的字符：我们已经有了 `\x{...}`，并且语法与我们可以用于反向引用（`\1`，`\2`，等）的语法相似，这使得产生有用的错误消息变得更加困难。
- `\cx`，类似于 $\text{T}_{\text{E}}\text{X}$ 的 `\^x`。
- 注释： $\text{T}_{\text{E}}\text{X}$ 已经有了自己的注释系统。
- `\Q...\E` 转义：这需要逐字读取参数，不在此模块的范围内。
- 在 UTF-8 模式下的单字节 `\C`： $\text{X}_{\text{E}}\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ 和 $\text{LuaT}_{\text{E}}\text{X}$ 直接为我们提供字符，将其拆分为字节是棘手的，依赖编码，而且很可能并不实用。

9 l3regex implementation

```
1 <*package>
```

```
2 <@@=regex>
```

9.1 Plan of attack

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

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

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

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

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

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

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

- `\g__regex_state_active_intarray` holds the last $\langle step \rangle$ in which each $\langle state \rangle$ was active.
- `\g__regex_thread_info_intarray` consists of blocks for each $\langle thread \rangle$ (with $\text{min_thread} \leq \langle thread \rangle < \text{max_thread}$). Each block has $1+2\backslash 1_ \text{regex_capturing_group_in}$

entries: the $\langle state \rangle$ in which the $\langle thread \rangle$ currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The $\langle threads \rangle$ are ordered starting from the best to the least preferred.

- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks\langle position \rangle` holds $\langle tokens \rangle$ which `o`- and `e`-expand to the $\langle position \rangle$ -th token in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

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

9.2 Helpers

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

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

(`__regex_int_eval:w` 定义结束。)

`__regex_standard_escapechar:` Make the `\escapechar` into the standard backslash.

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

(*_regex_standard_escapechar*: 定义结束。)

_regex_toks_use:w Unpack a *\toks* given its number.

```
6 \cs_new:Npn \_regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(*_regex_toks_use:w* 定义结束。)

_regex_toks_clear:N Empty a *\toks* or set it to a value, given its number.

```
7 \cs_new_protected:Npn \_regex_toks_clear:N #1
8 { \_regex_toks_set:Nn #1 { } }
9 \cs_new_eq:NN \_regex_toks_set:Nn \tex_toks:D
10 \cs_new_protected:Npn \_regex_toks_set:No #1
11 { \tex_toks:D #1 \exp_after:wN }
```

(*_regex_toks_clear:N* 和 *_regex_toks_set:Nn* 定义结束。)

_regex_toks_memcpy:NNn Copy #3 *\toks* registers from #2 onwards to #1 onwards, like C’s *memcpy*.

```
12 \cs_new_protected:Npn \_regex_toks_memcpy:NNn #1#2#3
13 {
14   \prg_replicate:nn {#3}
15   {
16     \tex_toks:D #1 = \tex_toks:D #2
17     \int_incr:N #1
18     \int_incr:N #2
19   }
20 }
```

(*_regex_toks_memcpy:NNn* 定义结束。)

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

```
21 \cs_new_protected:Npn \_regex_toks_put_left:Ne #1#2
22 {
23   \cs_set_nopar:Npe \_regex_tmp:w { #2 }
24   \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
25   { \exp_after:wN \_regex_tmp:w \tex_the:D \tex_toks:D #1 }
26 }
27 \cs_new_protected:Npn \_regex_toks_put_right:Ne #1#2
28 {
29   \cs_set_nopar:Npe \_regex_tmp:w {#2}
30   \tex_toks:D #1 \exp_after:wN
```



```

31      { \tex_the:D \tex_toks:D \exp_after:wN #1 \_\_regex_tmp:w }
32    }
33 \cs_new_protected:Npn \_\_regex_toks_put_right:Nn #1#2
34 { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }

```

(*__regex_toks_put_left:Ne* 和 *__regex_toks_put_right:Ne* 定义结束。)

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

```

35 \cs_new:Npn \_\_regex_curr_cs_to_str:
36 {
37   \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
38   \l_\_regex_curr_token_tl
39 }

```

(*__regex_curr_cs_to_str:* 定义结束。)

__regex_intarray_item:NnF Item of intarray, with a default value.

```

\_\_regex_intarray_item_aux:nNF
40 \cs_new:Npn \_\_regex_intarray_item:NnF #1#2
41 { \exp_args:Nf \_\_regex_intarray_item_aux:nNF { \int_eval:n {#2} } #1 }
42 \cs_new:Npn \_\_regex_intarray_item_aux:nNF #1#2
43 {
44   \if_int_compare:w #1 > \c_zero_int
45     \exp_after:wN \use_i:nn
46   \else:
47     \exp_after:wN \use_ii:nn
48   \fi:
49   { \_\_kernel_intarray_item:Nn #2 {#1} }
50 }

```

(*__regex_intarray_item:NnF* 和 *__regex_intarray_item_aux:nNF* 定义结束。)

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

```

51 \cs_new:Npn \_\_regex_maplike_break:
52 { \prg_map_break:Nn \_\_regex_maplike_break: { } }

```

(*__regex_maplike_break:* 定义结束。)

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

```

\__regex_tl_even_items:n
  \__regex_tl_even_items_loop:nn
53 \cs_new:Npn \__regex_tl_odd_items:n #1 { \__regex_tl_even_items:n { ? #1 } }
54 \cs_new:Npn \__regex_tl_even_items:n #1
55 {
56   \__regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
57   \prg_break_point:
58 }
59 \cs_new:Npn \__regex_tl_even_items_loop:nn #1#2
60 {
61   \__regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
62   { \exp_not:n {#2} }
63   \__regex_tl_even_items_loop:nn
64 }

```

(`__regex_tl_odd_items:n`, `__regex_tl_even_items:n`, 和 `__regex_tl_even_items_loop:nn` 定义结束。)

9.2.1 Constants and variables

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

```

65 \cs_new:Npn \__regex_tmp:w { }

```

(`__regex_tmp:w` 定义结束。)

`\l__regex_internal_a_tl` Temporary variables used for various purposes.

```

\l__regex_internal_b_tl
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl
66 \tl_new:N \l__regex_internal_a_tl
67 \tl_new:N \l__regex_internal_b_tl
68 \int_new:N \l__regex_internal_a_int
69 \int_new:N \l__regex_internal_b_int
70 \int_new:N \l__regex_internal_c_int
71 \bool_new:N \l__regex_internal_bool
72 \seq_new:N \l__regex_internal_seq
73 \tl_new:N \g__regex_internal_tl

```

(`\l__regex_internal_a_tl` 以及其它的定义结束。)

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

```

74 \tl_new:N \l__regex_build_tl

```

(`\l__regex_build_tl` 定义结束。)

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

```

75 \tl_const:Nn \c__regex_no_match_regex
76 {
77   \__regex_branch:n
78   { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
79 }

```

(`\c__regex_no_match_regex` 定义结束。)

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

```

80 \int_new:N \l__regex_balance_int

```

(`\l__regex_balance_int` 定义结束。)

9.2.2 Testing characters

`\c__regex_ascii_min_int`

`\c__regex_ascii_max_control_int`

`\c__regex_ascii_max_int`

```

81 \int_const:Nn \c__regex_ascii_min_int { 0 }
82 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
83 \int_const:Nn \c__regex_ascii_max_int { 127 }

```

(`\c__regex_ascii_min_int`, `\c__regex_ascii_max_control_int`, 和 `\c__regex_ascii_max_int` 定义结束。)

`\c__regex_ascii_lower_int`

```

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

```

(`\c__regex_ascii_lower_int` 定义结束。)

9.2.3 Internal auxiliaries

`\q__regex_recursion_stop` Internal recursion quarks.

```

85 \quark_new:N \q__regex_recursion_stop

```

(`\q__regex_recursion_stop` 定义结束。)

`\q__regex_nil` Internal quarks.

```

86 \quark_new:N \q__regex_nil

```

(`\q__regex_nil` 定义结束。)

`regex_use_none_delimit_by_q_recursion_stop:w` Functions to gobble up to a quark.

```

87 \cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
88   #1 \q__regex_recursion_stop { }
89 \cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
90   #1 #2 \q__regex_recursion_stop {#1}
91 \cs_new:Npn \__regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }

(\__regex_use_none_delimit_by_q_recursion_stop:w, \__regex_use_i_delimit_by_q_recursion_stop:nw, 和
\__regex_use_none_delimit_by_q_nil:w 定义结束。)
```

`__regex_quark_if_nil_p:n` Branching quark conditional.

`__regex_quark_if_nil:nTF`

```

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

(`__regex_quark_if_nil:nTF` 定义结束。)

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

`__regex_break_true:w`

```

<test1> ... <testn>
\__regex_break_point:TF {<true code>} {<false code>}
```

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

```

93 \cs_new_protected:Npn \__regex_break_true:w
94   #1 \__regex_break_point:TF #2 #3 {#2}
95 \cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }

(\__regex_break_point:TF 和 \__regex_break_true:w 定义结束。)
```

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

```

96 \cs_new_protected:Npn \__regex_item_reverse:n #1
97   {
98     #1
99     \__regex_break_point:TF { } \__regex_break_true:w
100  }

(\__regex_item_reverse:n 定义结束。)
```

`_regex_item_caseful_equal:n` Simple comparisons triggering `_regex_break_true:w` when true.

```
\_regex_item_caseful_range:nn 101 \cs_new_protected:Npn \_regex_item_caseful_equal:n #1
102 {
103   \if_int_compare:w #1 = \l__regex_curr_char_int
104     \exp_after:wN \_regex_break_true:w
105   \fi:
106 }
107 \cs_new_protected:Npn \_regex_item_caseful_range:nn #1 #2
108 {
109   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
110   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
111   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
112   \fi:
113   \fi:
114 }
```

(`_regex_item_caseful_equal:n` 和 `_regex_item_caseful_range:nn` 定义结束。)

`_regex_item_caseless_equal:n` For caseless matching, we perform the test both on the `curr_char` and on the `case_`
`_regex_item_caseless_range:nn` `changed_char`. Before doing the second set of tests, we make sure that `case_`
`changed_char` has been computed.

```
115 \cs_new_protected:Npn \_regex_item_caseless_equal:n #1
116 {
117   \if_int_compare:w #1 = \l__regex_curr_char_int
118     \exp_after:wN \_regex_break_true:w
119   \fi:
120   \_regex_maybe_compute_ccc:
121   \if_int_compare:w #1 = \l__regex_case_changed_char_int
122     \exp_after:wN \_regex_break_true:w
123   \fi:
124 }
125 \cs_new_protected:Npn \_regex_item_caseless_range:nn #1 #2
126 {
127   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
128   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
129   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
130   \fi:
131   \fi:
132   \_regex_maybe_compute_ccc:
133   \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
134   \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
135   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
```

```

136     \fi:
137     \fi:
138 }

```

(`_regex_item_caseless_equal:n` 和 `_regex_item_caseless_range:nn` 定义结束。)

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

```

139 \cs_new_protected:Npn \_regex_compute_case_changed_char:
140 {
141     \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
142     \if_int_compare:w \l__regex_curr_char_int > `Z \exp_stop_f:
143     \if_int_compare:w \l__regex_curr_char_int > `z \exp_stop_f: \else:
144     \if_int_compare:w \l__regex_curr_char_int < `a \exp_stop_f: \else:
145     \int_sub:Nn \l__regex_case_changed_char_int
146     { \c__regex_ascii_lower_int }
147     \fi:
148     \fi:
149     \else:
150     \if_int_compare:w \l__regex_curr_char_int < `A \exp_stop_f: \else:
151     \int_add:Nn \l__regex_case_changed_char_int
152     { \c__regex_ascii_lower_int }
153     \fi:
154     \fi:
155     \cs_set_eq:NN \_regex_maybe_compute_ccc: \prg_do_nothing:
156 }
157 \cs_new_eq:NN \_regex_maybe_compute_ccc: \_regex_compute_case_changed_char:

```

(`_regex_compute_case_changed_char:` 定义结束。)

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

`_regex_item_range:nn`

```

158 \cs_new_eq:NN \_regex_item_equal:n ?
159 \cs_new_eq:NN \_regex_item_range:nn ?

```

(`_regex_item_equal:n` 和 `_regex_item_range:nn` 定义结束。)

`_regex_item_catcode:nT` The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and
`_regex_item_catcode_reverse:nT`
`_regex_item_catcode:`

only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

160 \cs_new_protected:Npn \__regex_item_catcode:
161 {
162   "
163   \if_case:w \l__regex_curr_catcode_int
164     1      \or: 4      \or: 10      \or: 40
165   \or: 100  \or:      \or: 1000  \or: 4000
166   \or: 10000 \or:      \or: 100000 \or: 400000
167   \or: 1000000 \or: 4000000 \else: 1*0
168   \fi:
169 }
170 \cs_new_protected:Npn \__regex_item_catcode:nT #1
171 {
172   \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
173   \exp_after:wN \use:n
174   \else:
175   \exp_after:wN \use_none:n
176   \fi:
177 }
178 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
179 { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

```

(`__regex_item_catcode:nT`, `__regex_item_catcode_reverse:nT`, 和 `__regex_item_catcode:` 定义结束。)

`__regex_item_exact:nn` This matches an exact *<category>-<character code>* pair, or an exact control sequence,
`__regex_item_exact_cs:n` more precisely one of several possible control sequences, separated by `\scan_stop:`.

```

180 \cs_new_protected:Npn \__regex_item_exact:nn #1#2
181 {
182   \if_int_compare:w #1 = \l__regex_curr_catcode_int
183   \if_int_compare:w #2 = \l__regex_curr_char_int
184   \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
185   \fi:
186   \fi:
187 }
188 \cs_new_protected:Npn \__regex_item_exact_cs:n #1
189 {
190   \int_compare:nNnTF \l__regex_curr_catcode_int = 0
191   {
192     \__kernel_tl_set:Ne \l__regex_internal_a_tl
193     { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
194     \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }

```

```

195         \l__regex_internal_a_tl
196         { \__regex_break_true:w } { }
197     }
198     { }
199 }

```

(__regex_item_exact:nn 和 __regex_item_exact_cs:n 定义结束。)

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

```

200 \cs_new_protected:Npn \__regex_item_cs:n #1
201 {
202     \int_compare:nNnT \l__regex_curr_catcode_int = 0
203     {
204         \group_begin:
205             \__regex_single_match:
206             \__regex_disable_submatches:
207             \__regex_build_for_cs:n {#1}
208             \bool_set_eq:NN \l__regex_saved_success_bool
209             \g__regex_success_bool
210             \exp_args:Ne \__regex_match_cs:n { \__regex_curr_cs_to_str: }
211             \if_meaning:w \c_true_bool \g__regex_success_bool
212             \group_insert_after:N \__regex_break_true:w
213             \fi:
214             \bool_gset_eq:NN \g__regex_success_bool
215             \l__regex_saved_success_bool
216         \group_end:
217     }
218 }

```

(__regex_item_cs:n 定义结束。)

9.2.4 Character property tests

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

```

219 \cs_new_protected:Npn \__regex_prop_d:

```



```

220 { \_regex_item_caseful_range:nn { `0 } { `9 } }
221 \cs_new_protected:Npn \_regex_prop_h:
222 {
223   \_regex_item_caseful_equal:n { ``\ }
224   \_regex_item_caseful_equal:n { ``^^I }
225 }
226 \cs_new_protected:Npn \_regex_prop_s:
227 {
228   \_regex_item_caseful_equal:n { ``\ }
229   \_regex_item_caseful_equal:n { ``^^I }
230   \_regex_item_caseful_equal:n { ``^^J }
231   \_regex_item_caseful_equal:n { ``^^L }
232   \_regex_item_caseful_equal:n { ``^^M }
233 }
234 \cs_new_protected:Npn \_regex_prop_v:
235 { \_regex_item_caseful_range:nn { ``^^J } { ``^^M } } % lf, vtab, ff, cr
236 \cs_new_protected:Npn \_regex_prop_w:
237 {
238   \_regex_item_caseful_range:nn { `a } { `z }
239   \_regex_item_caseful_range:nn { `A } { `Z }
240   \_regex_item_caseful_range:nn { `0 } { `9 }
241   \_regex_item_caseful_equal:n { `_ }
242 }
243 \cs_new_protected:Npn \_regex_prop_N:
244 {
245   \_regex_item_reverse:n
246   { \_regex_item_caseful_equal:n { ``^^J } }
247 }

```

(*_regex_prop_d:* 以及其它的定义结束。)

_regex_posix_alnum: POSIX properties. No surprise.

```

\_regex_posix_alpha: 248 \cs_new_protected:Npn \_regex_posix_alnum:
\_regex_posix_ascii: 249 { \_regex_posix_alpha: \_regex_posix_digit: }
\_regex_posix_blank: 250 \cs_new_protected:Npn \_regex_posix_alpha:
\_regex_posix_ctrl: 251 { \_regex_posix_lower: \_regex_posix_upper: }
\_regex_posix_digit: 252 \cs_new_protected:Npn \_regex_posix_ascii:
\_regex_posix_graph: 253 {
\_regex_posix_lower: 254   \_regex_item_caseful_range:nn
\_regex_posix_print: 255   \c__regex_ascii_min_int
\_regex_posix_punct: 256   \c__regex_ascii_max_int
\_regex_posix_space: 257 }
\_regex_posix_upper: 258 \cs_new_eq:NN \_regex_posix_blank: \_regex_prop_h:
\_regex_posix_word:
\_regex_posix_xdigit:

```

```

259 \cs_new_protected:Npn \__regex_posix_ctrl:
260 {
261   \__regex_item_caseful_range:nn
262     \c__regex_ascii_min_int
263     \c__regex_ascii_max_control_int
264   \__regex_item_caseful_equal:n \c__regex_ascii_max_int
265 }
266 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
267 \cs_new_protected:Npn \__regex_posix_graph:
268 { \__regex_item_caseful_range:nn { `! } { `~ } }
269 \cs_new_protected:Npn \__regex_posix_lower:
270 { \__regex_item_caseful_range:nn { `a } { `z } }
271 \cs_new_protected:Npn \__regex_posix_print:
272 { \__regex_item_caseful_range:nn { `\ } { `~ } }
273 \cs_new_protected:Npn \__regex_posix_punct:
274 {
275   \__regex_item_caseful_range:nn { `! } { `/ }
276   \__regex_item_caseful_range:nn { `: } { `@ }
277   \__regex_item_caseful_range:nn { `[ } { `` }
278   \__regex_item_caseful_range:nn { `\{ } { `~ }
279 }
280 \cs_new_protected:Npn \__regex_posix_space:
281 {
282   \__regex_item_caseful_equal:n { `\ }
283   \__regex_item_caseful_range:nn { `^~I } { `^~M }
284 }
285 \cs_new_protected:Npn \__regex_posix_upper:
286 { \__regex_item_caseful_range:nn { `A } { `Z } }
287 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
288 \cs_new_protected:Npn \__regex_posix_xdigit:
289 {
290   \__regex_posix_digit:
291   \__regex_item_caseful_range:nn { `A } { `F }
292   \__regex_item_caseful_range:nn { `a } { `f }
293 }

```

(`__regex_posix_alnum:` 以及其它的定义结束。)

9.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, *etc.*) or escaped alphanumeric character into a

marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `_regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* {*<token list>*}

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

The conversion is done within an e-expanding assignment.

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

```

294 \cs_new_protected:Npn \_regex_escape_use:nnnn #1#2#3#4
295 {
296   \group_begin:
297   \tl_clear:N \l__regex_internal_a_tl
298   \cs_set:Npn \_regex_escape_unescaped:N ##1 { #1 }
299   \cs_set:Npn \_regex_escape_escaped:N ##1 { #2 }
300   \cs_set:Npn \_regex_escape_raw:N ##1 { #3 }
301   \__regex_standard_escapechar:
302   \__kernel_tl_gset:Ne \g__regex_internal_tl
303     { \__kernel_str_to_other_fast:n {#4} }
304   \tl_put_right:Ne \l__regex_internal_a_tl
305     {
306       \exp_after:wN \_regex_escape_loop:N \g__regex_internal_tl
307       \scan_stop: \prg_break_point:
308     }
309   \exp_after:wN
310   \group_end:
311   \l__regex_internal_a_tl
312 }

```

(`_regex_escape_use:nnnn` 定义结束。)

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

```

313 \cs_new:Npn __regex_escape_loop:N #1
314 {
315   \cs_if_exist_use:cF { __regex_escape\_token_to_str:N #1:w }
316   { __regex_escape_unescaped:N #1 }
317   __regex_escape_loop:N
318 }
319 \cs_new:cpn { __regex_escape\_c_backslash_str :w }
320   __regex_escape_loop:N #1
321 {
322   \cs_if_exist_use:cF { __regex_escape\_token_to_str:N #1:w }
323   { __regex_escape_escaped:N #1 }
324   __regex_escape_loop:N
325 }

```

(`__regex_escape_loop:N` 和 `__regex_escape_w` 定义结束。)

`__regex_escape_unescaped:N` Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

__regex_escape_escaped:N
__regex_escape_raw:N
326 \cs_new_eq:NN __regex_escape_unescaped:N ?
327 \cs_new_eq:NN __regex_escape_escaped:N ?
328 \cs_new_eq:NN __regex_escape_raw:N ?

```

(`__regex_escape_unescaped:N`, `__regex_escape_escaped:N`, 和 `__regex_escape_raw:N` 定义结束。)

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

```

__regex_escape\_scan_stop:w
__regex_escape\_a:w
__regex_escape\_e:w
__regex_escape\_f:w
__regex_escape\_n:w
__regex_escape\_r:w
__regex_escape\_t:w
__regex_escape\_ :w
329 \cs_new_eq:cN { __regex_escape\_iow_char:N\scan_stop: :w } \prg_break:
330 \cs_new:cpn { __regex_escape\_iow_char:N\scan_stop: :w }
331 {
332   \msg_expandable_error:nn { regex } { trailing-backslash }
333   \prg_break:
334 }
335 \cs_new:cpn { __regex_escape\_~:w } { }
336 \cs_new:cpe { __regex_escape\_a:w }
337 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^G }
338 \cs_new:cpe { __regex_escape\_t:w }
339 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^I }
340 \cs_new:cpe { __regex_escape\_n:w }

```

```

341 { \exp_not:N \__regex_escape_raw:N \iow_char:N ^^J }
342 \cs_new:cpe { __regex_escape_/f:w }
343 { \exp_not:N \__regex_escape_raw:N \iow_char:N ^^L }
344 \cs_new:cpe { __regex_escape_/r:w }
345 { \exp_not:N \__regex_escape_raw:N \iow_char:N ^^M }
346 \cs_new:cpe { __regex_escape_/e:w }
347 { \exp_not:N \__regex_escape_raw:N \iow_char:N ^^[ }

```

(`__regex_escape_scan_stop:w` 以及其它的定义结束。)

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

```

348 \cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
349 {
350   \exp_after:wN \__regex_escape_x_end:w
351   \int_value:w "0 \__regex_escape_x_test:N
352 }
353 \cs_new:Npn \__regex_escape_x_end:w #1 ;
354 {
355   \int_compare:nNnTF {#1} > \c_max_char_int
356   {
357     \msg_expandable_error:nnff { regex } { x-overflow }
358     {#1} { \int_to_Hex:n {#1} }
359   }
360   {
361     \exp_last_unbraced:Nf \__regex_escape_raw:N
362     { \char_generate:nn {#1} { 12 } }
363   }
364 }

```

(`__regex_escape_/x:w`, `__regex_escape_x_end:w`, 和 `__regex_escape_x_large:n` 定义结束。)

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

```

365 \cs_new:Npn \__regex_escape_x_test:N #1
366 {
367   \if_meaning:w \scan_stop: #1
368   \exp_after:wN \use_i:nnn \exp_after:wN ;

```

```

369     \fi:
370     \use:n
371     {
372         \if_charcode:w \c_space_token #1
373         \exp_after:wN \__regex_escape_x_test:N
374         \else:
375         \exp_after:wN \__regex_escape_x_testii:N
376         \exp_after:wN #1
377         \fi:
378     }
379 }
380 \cs_new:Npn \__regex_escape_x_testii:N #1
381 {
382     \if_charcode:w \c_left_brace_str #1
383     \exp_after:wN \__regex_escape_x_loop:N
384     \else:
385     \__regex_hexadecimal_use:NTF #1
386     { \exp_after:wN \__regex_escape_x:N }
387     { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
388     \fi:
389 }

```

(*__regex_escape_x_test:N* 和 *__regex_escape_x_testii:N* 定义结束。)

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

```

390 \cs_new:Npn \__regex_escape_x:N #1
391 {
392     \if_meaning:w \scan_stop: #1
393     \exp_after:wN \use_i:nnn \exp_after:wN ;
394     \fi:
395     \use:n
396     {
397         \__regex_hexadecimal_use:NTF #1
398         { ; \__regex_escape_loop:N }
399         { ; \__regex_escape_loop:N #1 }
400     }
401 }

```

(*__regex_escape_x:N* 定义结束。)

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

__regex_escape_x_loop_error:

```

402 \cs_new:Npn \__regex_escape_x_loop:N #1

```

```

403 {
404   \if_meaning:w \scan_stop: #1
405   \exp_after:wN \use_ii:nnn
406   \fi:
407   \use_ii:nn
408   { ; \__regex_escape_x_loop_error:n { } {#1} }
409   {
410     \__regex_hexadecimal_use:NTF #1
411     { \__regex_escape_x_loop:N }
412     {
413       \token_if_eq_charcode:NNTF \c_space_token #1
414       { \__regex_escape_x_loop:N }
415       {
416         ;
417         \exp_after:wN
418         \token_if_eq_charcode:NNTF \c_right_brace_str #1
419         { \__regex_escape_loop:N }
420         { \__regex_escape_x_loop_error:n {#1} }
421       }
422     }
423   }
424 }
425 \cs_new:Npn \__regex_escape_x_loop_error:n #1
426 {
427   \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
428   \__regex_escape_loop:N #1
429 }

```

(`__regex_escape_x_loop:N` 和 `__regex_escape_x_loop_error:` 定义结束。)

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

```

430 \prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
431 {
432   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
433   #1 \prg_return_true:
434   \else:
435     \if_case:w
436       \int_eval:n { \exp_after:wN ` \token_to_str:N #1 - `a }
437       A
438     \or: B
439     \or: C
440     \or: D

```

```

441     \or: E
442     \or: F
443     \else:
444         \prg_return_false:
445         \exp_after:wN \use_none:n
446     \fi:
447     \prg_return_true:
448 \fi:
449 }

```

(`_regex_hexadecimal_use:NTF` 定义结束。)

`_regex_char_if_alphanumeric: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,

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

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

```

450 \prg_new_conditional:Npnn \_regex_char_if_special:N #1 { TF }
451 {
452     \if_int_compare:w `#1 > `Z \exp_stop_f:
453     \if_int_compare:w `#1 > `z \exp_stop_f:
454     \if_int_compare:w `#1 < \c__regex_ascii_max_int
455         \prg_return_true: \else: \prg_return_false: \fi:
456     \else:
457         \if_int_compare:w `#1 < `a \exp_stop_f:
458         \prg_return_true: \else: \prg_return_false: \fi:
459     \fi:
460 \else:
461     \if_int_compare:w `#1 > `9 \exp_stop_f:

```



```

462         \if_int_compare:w `#1 < `A \exp_stop_f:
463         \prg_return_true: \else: \prg_return_false: \fi:
464     \else:
465         \if_int_compare:w `#1 < `0 \exp_stop_f:
466         \if_int_compare:w `#1 < `` \exp_stop_f:
467         \prg_return_false: \else: \prg_return_true: \fi:
468     \else: \prg_return_false: \fi:
469     \fi:
470 \fi:
471 }
472 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
473 {
474     \if_int_compare:w `#1 > `Z \exp_stop_f:
475     \if_int_compare:w `#1 > `z \exp_stop_f:
476     \prg_return_false:
477     \else:
478         \if_int_compare:w `#1 < `a \exp_stop_f:
479         \prg_return_false: \else: \prg_return_true: \fi:
480     \fi:
481     \else:
482         \if_int_compare:w `#1 > `9 \exp_stop_f:
483         \if_int_compare:w `#1 < `A \exp_stop_f:
484         \prg_return_false: \else: \prg_return_true: \fi:
485     \else:
486         \if_int_compare:w `#1 < `0 \exp_stop_f:
487         \prg_return_false: \else: \prg_return_true: \fi:
488     \fi:
489 \fi:
490 }

```

(`__regex_char_if_alphanumeric:N`TF 和 `__regex_char_if_special:N`TF 定义结束。)

9.3 Compiling

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

- `__regex_class:NnnnN` $\langle \text{boolean} \rangle$ $\{ \langle \text{tests} \rangle \}$ $\{ \langle \text{min} \rangle \}$ $\{ \langle \text{more} \rangle \}$ $\langle \text{lazyness} \rangle$
- `__regex_group:nnnN` $\{ \langle \text{branches} \rangle \}$ $\{ \langle \text{min} \rangle \}$ $\{ \langle \text{more} \rangle \}$ $\langle \text{lazyness} \rangle$, also `__regex_group_no_capture:nnnN` and `__regex_group_resetting:nnnN` with the same

syntax.

- `__regex_branch:n {⟨contents⟩}`
- `__regex_command_K:`
- `__regex_assertion:Nn ⟨boolean⟩ {⟨assertion test⟩}`, where the `⟨assertion test⟩` is `__regex_b_test:` or `__regex_Z_test:` or `__regex_A_test:` or `__regex_G_test:`

Tests can be the following:

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

9.3.1 Variables used when compiling

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

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

(`\l__regex_group_level_int` 定义结束。)

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

```
\c__regex_cs_in_class_mode_int 492 \int_new:N \l__regex_mode_int
\c__regex_cs_mode_int           493 \int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\c__regex_outer_mode_int       494 \int_const:Nn \c__regex_cs_mode_int { -2 }
\c__regex_catcode_mode_int     495 \int_const:Nn \c__regex_outer_mode_int { 0 }
\c__regex_class_mode_int
\c__regex_catcode_in_class_mode_int
```

```

496 \int_const:Nn \c__regex_catcode_mode_int { 2 }
497 \int_const:Nn \c__regex_class_mode_int { 3 }
498 \int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }

```

(*\l__regex_mode_int* 以及其它的定义结束。)

\l__regex_catcodes_int We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test.

\l__regex_default_catcodes_int

\l__regex_catcodes_bool For this to work, we need to keep track of lists of allowed category codes: *\l__regex_catcodes_int* and *\l__regex_default_catcodes_int* are bitmaps, sums of 4^c , for all allowed catcodes c . The latter is local to each capturing group, and we reset *\l__regex_catcodes_int* to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```

499 \int_new:N \l__regex_catcodes_int
500 \int_new:N \l__regex_default_catcodes_int
501 \bool_new:N \l__regex_catcodes_bool

```

(*\l__regex_catcodes_int*, *\l__regex_default_catcodes_int*, 和 *\l__regex_catcodes_bool* 定义结束。)

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

\c__regex_catcode_B_int

\c__regex_catcode_E_int

\c__regex_catcode_M_int

\c__regex_catcode_T_int

\c__regex_catcode_P_int

\c__regex_catcode_U_int

\c__regex_catcode_D_int

\c__regex_catcode_S_int

\c__regex_catcode_L_int

\c__regex_catcode_O_int

\c__regex_catcode_A_int

\c__regex_all_catcodes_int

```

502 \int_const:Nn \c__regex_catcode_C_int { "1 }
503 \int_const:Nn \c__regex_catcode_B_int { "4 }
504 \int_const:Nn \c__regex_catcode_E_int { "10 }
505 \int_const:Nn \c__regex_catcode_M_int { "40 }
506 \int_const:Nn \c__regex_catcode_T_int { "100 }
507 \int_const:Nn \c__regex_catcode_P_int { "1000 }
508 \int_const:Nn \c__regex_catcode_U_int { "4000 }
509 \int_const:Nn \c__regex_catcode_D_int { "10000 }
510 \int_const:Nn \c__regex_catcode_S_int { "100000 }
511 \int_const:Nn \c__regex_catcode_L_int { "400000 }
512 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
513 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
514 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }

```

(*\c__regex_catcode_C_int* 以及其它的定义结束。)

\l__regex_internal_regex The compilation step stores its result in this variable.

```

515 \cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex

```

(*\l__regex_internal_regex* 定义结束。)

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

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

(`\l__regex_show_prefix_seq` 定义结束。)

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

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

(`\l__regex_show_lines_int` 定义结束。)

9.3.2 Generic helpers used when compiling

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

```
518 \prg_new_conditional:Npnn \__regex_two_if_eq:NNNN #1#2#3#4 { TF }
519 {
520   \if_meaning:w #1 #3
521   \if:w #2 #4
522     \prg_return_true:
523   \else:
524     \prg_return_false:
525   \fi:
526 \else:
527   \prg_return_false:
528 \fi:
529 }
```

(`__regex_two_if_eq:NNNTF` 定义结束。)

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

`__regex_get_digits_loop:w`

```
530 \cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
531 {
532   \__regex_if_raw_digit:NNTF #4 #5
533   { #1 = #5 \__regex_get_digits_loop:nw {#2} }
534   { #3 #4 #5 }
535 }
```

```

536 \cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
537 {
538   \__regex_if_raw_digit:NNTF #2 #3
539   { #3 \__regex_get_digits_loop:nw {#1} }
540   { \scan_stop: #1 #2 #3 }
541 }

```

(*__regex_get_digits:NNTFw* 和 *__regex_get_digits_loop:w* 定义结束。)

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

```

542 \prg_new_conditional:Npnn \__regex_if_raw_digit:NN #1#2 { TF }
543 {
544   \if_meaning:w \__regex_compile_raw:N #1
545   \if_int_compare:w 1 < 1 #2 \exp_stop_f:
546   \prg_return_true:
547   \else:
548   \prg_return_false:
549   \fi:
550   \else:
551   \prg_return_false:
552   \fi:
553 }

```

(*__regex_if_raw_digit:NNTF* 定义结束。)

9.3.3 Mode

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

- 6 [\c{...}] control sequence in a class,
- 2 \c{...} control sequence,
- 0 ... outer,
- 2 \c... catcode test,
- 6 [\c...] catcode test in a class,
- 63 [\c{[...]}] class inside mode -6,
- 23 \c{[...]} class inside mode -2,

3 [...] class inside mode 0,
 23 \c[...] class inside mode 2,
 63 [\c[...]] class inside mode 6.

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

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \rightarrow (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the \d character class) can appear in any mode.

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

```

554 \cs_new:Npn \__regex_if_in_class:TF
555   {
556     \if_int_odd:w \l__regex_mode_int
557       \exp_after:wN \use_i:nn
558     \else:
559       \exp_after:wN \use_ii:nn
560     \fi:
561   }
```

(`__regex_if_in_class:TF` 定义结束。)

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

```

562 \cs_new:Npn \__regex_if_in_cs:TF
563 {
564   \if_int_odd:w \l__regex_mode_int
565     \exp_after:wN \use_ii:nn
566   \else:
567     \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
568       \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
569     \else:
570       \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
571     \fi:
572   \fi:
573 }
```

(`__regex_if_in_cs:TF` 定义结束。)

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

```

574 \cs_new:Npn \__regex_if_in_class_or_catcode:TF
575 {
576   \if_int_odd:w \l__regex_mode_int
577     \exp_after:wN \use_i:nn
578   \else:
579     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
580       \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
581     \else:
582       \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
583     \fi:
584   \fi:
585 }
```

(`__regex_if_in_class_or_catcode:TF` 定义结束。)

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

```

586 \cs_new:Npn \__regex_if_within_catcode:TF
587 {
588   \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
589     \exp_after:wN \use_i:nn
590   \else:
591     \exp_after:wN \use_ii:nn
```

```

592     \fi:
593 }

```

(`_regex_if_within_catcode:TF` 定义结束。)

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

```

594 \cs_new_protected:Npn \_regex_chk_c_allowed:T
595 {
596   \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
597     \exp_after:wN \use:n
598   \else:
599     \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
600       \exp_after:wN \exp_after:wN \exp_after:wN \use:n
601     \else:
602       \msg_error:nn { regex } { c-bad-mode }
603       \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
604     \fi:
605   \fi:
606 }

```

(`_regex_chk_c_allowed:T` 定义结束。)

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

```

607 \cs_new_protected:Npn \_regex_mode_quit_c:
608 {
609   \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
610     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
611   \else:
612     \if_int_compare:w \l__regex_mode_int =
613       \c__regex_catcode_in_class_mode_int
614     \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
615   \fi:
616   \fi:
617 }

```

(`_regex_mode_quit_c:` 定义结束。)

9.3.4 Framework

`_regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with `e`-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch.

`_regex_compile_end:`

At the end, make sure there are no dangling classes nor groups, close the last branch:
we are done building `\l__regex_internal_regex`.

```

618 \cs_new_protected:Npn \__regex_compile:w
619 {
620   \group_begin:
621     \tl_build_begin:N \l__regex_build_tl
622     \int_zero:N \l__regex_group_level_int
623     \int_set_eq:NN \l__regex_default_catcodes_int
624       \c_regex_all_catcodes_int
625     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
626     \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
627     \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
628     \tl_build_put_right:Nn \l__regex_build_tl
629       { \__regex_branch:n { \if_false: } \fi: }
630   }
631 \cs_new_protected:Npn \__regex_compile_end:
632 {
633   \__regex_if_in_class:TF
634   {
635     \msg_error:nn { regex } { missing-rbrack }
636     \use:c { __regex_compile_]: }
637     \prg_do_nothing: \prg_do_nothing:
638   }
639   { }
640   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
641     \msg_error:nne { regex } { missing-rparen }
642     { \int_use:N \l__regex_group_level_int }
643     \prg_replicate:nn
644       { \l__regex_group_level_int }
645     {
646       \tl_build_put_right:Nn \l__regex_build_tl
647       {
648         \if_false: { \fi: }
649         \if_false: { \fi: } { 1 } { 0 } \c_true_bool
650       }
651       \tl_build_end:N \l__regex_build_tl
652       \exp_args:NNNo
653       \group_end:
654       \tl_build_put_right:Nn \l__regex_build_tl
655         { \l__regex_build_tl }
656     }
657   \fi:

```

```

658     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
659     \tl_build_end:N \l__regex_build_tl
660     \exp_args:NNNe
661     \group_end:
662     \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
663 }

```

(`__regex_compile:w` 和 `__regex_compile_end:` 定义结束。)

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

```

664 \cs_new_protected:Npn \__regex_compile:n #1
665 {
666     \__regex_compile:w
667     \__regex_standard_escapechar:
668     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
669     \__regex_escape_use:nnnn
670     {
671         \__regex_char_if_special:NTF ##1
672         \__regex_compile_special:N \__regex_compile_raw:N ##1
673     }
674     {
675         \__regex_char_if_alphanumeric:NTF ##1
676         \__regex_compile_escaped:N \__regex_compile_raw:N ##1
677     }
678     { \__regex_compile_raw:N ##1 }
679     { #1 }
680     \prg_do_nothing: \prg_do_nothing:
681     \prg_do_nothing: \prg_do_nothing:
682     \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
683     { \msg_error:nn { regex } { c-trailing } }
684     \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
685     {
686         \msg_error:nn { regex } { c-missing-rbrace }
687         \__regex_compile_end_cs:
688         \prg_do_nothing: \prg_do_nothing:
689         \prg_do_nothing: \prg_do_nothing:

```

```

690     }
691     \__regex_compile_end:
692 }

```

(__regex_compile:n 定义结束。)

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

```

693 \cs_new_protected:Npn \__regex_compile_use:n #1
694 {
695     \tl_if_single_token:nT {#1}
696     {
697         \exp_after:wN \__regex_compile_use_aux:w
698         \token_to_meaning:N #1 ~ \q__regex_nil
699     }
700     \__regex_compile:n {#1} \l__regex_internal_regex
701 }
702 \cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
703 {
704     \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
705     { \use_ii:nnn }
706 }

```

(__regex_compile_use:n 定义结束。)

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

```

707 \cs_new_protected:Npn \__regex_compile_special:N #1
708 {
709     \cs_if_exist_use:cF { __regex_compile_#1: }
710     { \__regex_compile_raw:N #1 }
711 }
712 \cs_new_protected:Npn \__regex_compile_escaped:N #1
713 {
714     \cs_if_exist_use:cF { __regex_compile_/#1: }
715     { \__regex_compile_raw:N #1 }
716 }

```

(__regex_compile_escaped:N 和 __regex_compile_special:N 定义结束。)

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

```

717 \cs_new_protected:Npn \__regex_compile_one:n #1
718 {
719   \__regex_mode_quit_c:
720   \__regex_if_in_class:TF { }
721   {
722     \tl_build_put_right:Nn \l__regex_build_tl
723     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
724   }
725   \tl_build_put_right:Nn \l__regex_build_tl
726   {
727     \if_int_compare:w \l__regex_catcodes_int <
728     \c__regex_all_catcodes_int
729     \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
730     { \exp_not:N \exp_not:n {#1} }
731     \else:
732     \exp_not:N \exp_not:n {#1}
733     \fi:
734   }
735   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
736   \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
737 }

```

(`__regex_compile_one:n` 定义结束。)

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

```

\__regex_compile_abort_tokens:e
738 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
739 {
740   \use:e
741   {
742     \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
743     \__regex_compile_raw:N
744   }
745 }
746 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { e }

```

(`__regex_compile_abort_tokens:n` 定义结束。)

9.3.5 Quantifiers

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

```
747 \cs_new_protected:Npn \_regex_compile_if_quantifier:TFw #1#2#3#4
748 {
749   \token_if_eq_meaning:NNTF #3 \_regex_compile_special:N
750   { \cs_if_exist:cTF { \_regex_compile_quantifier_#4:w } }
751   { \use_i:nn }
752   {#1} {#2} #3 #4
753 }
```

(`_regex_compile_if_quantifier:TFw` 定义结束。)

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

```
754 \cs_new_protected:Npn \_regex_compile_quantifier:w #1#2
755 {
756   \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
757   {
758     \cs_if_exist_use:cF { \_regex_compile_quantifier_#2:w }
759     { \_regex_compile_quantifier_none: #1 #2 }
760   }
761   { \_regex_compile_quantifier_none: #1 #2 }
762 }
```

(`_regex_compile_quantifier:w` 定义结束。)

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

`_regex_compile_quantifier_abort:eNN`

```
763 \cs_new_protected:Npn \_regex_compile_quantifier_none:
764 {
765   \tl_build_put_right:Nn \l__regex_build_tl
766   { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
767 }
768 \cs_new_protected:Npn \_regex_compile_quantifier_abort:eNN #1#2#3
769 {
770   \_regex_compile_quantifier_none:
771   \msg_warning:nnee { regex } { invalid-quantifier } {#1} {#3}
772   \_regex_compile_abort_tokens:e {#1}
773   #2 #3
774 }
```

(`_regex_compile_quantifier_none:` 和 `_regex_compile_quantifier_abort:eNN` 定义结束。)

`_regex_compile_quantifier_lazyness:nnNN`

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

```

775 \cs_new_protected:Npn \_regex_compile_quantifier_lazyness:nnNN #1#2#3#4
776 {
777   \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ?
778   {
779     \tl_build_put_right:Nn \l__regex_build_tl
780     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
781   }
782   {
783     \tl_build_put_right:Nn \l__regex_build_tl
784     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
785     #3 #4
786   }
787 }
```

(`_regex_compile_quantifier_lazyness:nnNN` 定义结束。)

`_regex_compile_quantifier?:w`

`_regex_compile_quantifier*:w`

`_regex_compile_quantifier+:w`

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

```

788 \cs_new_protected:cpn { \_regex_compile_quantifier?:w }
789 { \_regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
790 \cs_new_protected:cpn { \_regex_compile_quantifier*:w }
791 { \_regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
792 \cs_new_protected:cpn { \_regex_compile_quantifier+:w }
793 { \_regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(`_regex_compile_quantifier?:w`, `_regex_compile_quantifier*:w`, 和 `_regex_compile_quantifier+:w` 定义结束。)

`_regex_compile_quantifier{:w`

`_regex_compile_quantifier_braced_auxi:w`

`_regex_compile_quantifier_braced_auxii:w`

`_regex_compile_quantifier_braced_auxiii:w`

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

```

794 \cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
795 {
796   \__regex_get_digits:NTFw \l__regex_internal_a_int
797   { \__regex_compile_quantifier_braced_auxi:w }
798   { \__regex_compile_quantifier_abort:eNN { \c_left_brace_str } }
799 }
800 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
801 {
802   \str_case_e:nnF { #1 #2 }
803   {
804     { \__regex_compile_special:N \c_right_brace_str }
805     {
806       \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
807       { \int_use:N \l__regex_internal_a_int } { 0 }
808     }
809     { \__regex_compile_special:N , }
810     {
811       \__regex_get_digits:NTFw \l__regex_internal_b_int
812       { \__regex_compile_quantifier_braced_auxiii:w }
813       { \__regex_compile_quantifier_braced_auxii:w }
814     }
815   }
816   {
817     \__regex_compile_quantifier_abort:eNN
818     { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
819     #1 #2
820   }
821 }
822 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
823 {
824   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
825   {
826     \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
827     { \int_use:N \l__regex_internal_a_int } { -1 }
828   }
829   {
830     \__regex_compile_quantifier_abort:eNN
831     { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
832     #1 #2
833   }
834 }
835 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2

```

```

836 {
837     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
838     {
839         \if_int_compare:w \l__regex_internal_a_int >
840         \l__regex_internal_b_int
841         \msg_error:nnee { regex } { backwards-quantifier }
842         { \int_use:N \l__regex_internal_a_int }
843         { \int_use:N \l__regex_internal_b_int }
844         \int_zero:N \l__regex_internal_b_int
845     \else:
846         \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
847     \fi:
848     \exp_args:Noo \_regex_compile_quantifier_lazyness:nnNN
849     { \int_use:N \l__regex_internal_a_int }
850     { \int_use:N \l__regex_internal_b_int }
851 }
852 {
853     \_regex_compile_quantifier_abort:eNN
854     {
855         \c_left_brace_str
856         \int_use:N \l__regex_internal_a_int ,
857         \int_use:N \l__regex_internal_b_int
858     }
859     #1 #2
860 }
861 }

```

(_regex_compile_quantifier_f:w 以及其它的定义结束。)

9.3.6 Raw characters

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

```

862 \cs_new_protected:Npn \_regex_compile_raw_error:N #1
863 {
864     \msg_error:nne { regex } { bad-escape } {#1}
865     \_regex_compile_raw:N #1
866 }

```

(_regex_compile_raw_error:N 定义结束。)

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

```

867 \cs_new_protected:Npn \\_regex_compile_raw:N #1#2#3
868 {
869   \\_regex_if_in_class:TF
870   {
871     \\_regex_two_if_eq:NNNTF #2 #3 \\_regex_compile_special:N -
872     { \\_regex_compile_range:Nw #1 }
873     {
874       \\_regex_compile_one:n
875       { \\_regex_item_equal:n { \\int_value:w `#1 } }
876       #2 #3
877     }
878   }
879   {
880     \\_regex_compile_one:n
881     { \\_regex_item_equal:n { \\int_value:w `#1 } }
882     #2 #3
883   }
884 }

```

(_regex_compile_raw:N 定义结束。)

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

`_regex_if_end_range:NTTF`

```

885 \prg_new_protected_conditional:Npnn \\_regex_if_end_range:NN #1#2 { TF }
886 {
887   \\if_meaning:w \\_regex_compile_raw:N #1
888   \\prg_return_true:
889   \\else:
890     \\if_meaning:w \\_regex_compile_special:N #1
891     \\if_charcode:w ] #2
892     \\prg_return_false:
893     \\else:
894       \\prg_return_true:
895     \\fi:
896     \\else:
897       \\prg_return_false:
898     \\fi:
899   \\fi:
900 }

```

```

901 \cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
902 {
903   \__regex_if_end_range:NNTF #2 #3
904   {
905     \if_int_compare:w `#1 > `#3 \exp_stop_f:
906     \msg_error:nnee { regex } { range-backwards } {#1} {#3}
907     \else:
908     \tl_build_put_right:Ne \l__regex_build_tl
909     {
910       \if_int_compare:w `#1 = `#3 \exp_stop_f:
911       \__regex_item_equal:n
912       \else:
913       \__regex_item_range:nn { \int_value:w `#1 }
914       \fi:
915       { \int_value:w `#3 }
916     }
917     \fi:
918   }
919   {
920     \msg_warning:nnee { regex } { range-missing-end }
921     {#1} { \c_backslash_str #3 }
922     \tl_build_put_right:Ne \l__regex_build_tl
923     {
924       \__regex_item_equal:n { \int_value:w `#1 \exp_stop_f: }
925       \__regex_item_equal:n { \int_value:w `~ \exp_stop_f: }
926     }
927     #2#3
928   }
929 }

```

(`__regex_compile_range:Nw` 和 `__regex_if_end_range:NNTF` 定义结束。)

9.3.7 Character properties

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

```

930 \cs_new_protected:cpe { __regex_compile_.: }
931 {
932   \exp_not:N \__regex_if_in_class:TF
933   { \__regex_compile_raw:N . }
934   { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
935 }
936 \cs_new_protected:cpn { __regex_prop_.: }

```

```

937 {
938     \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
939     \exp_after:wN \__regex_break_true:w
940     \fi:
941 }

```

(`__regex_compile_.`: 和 `__regex_prop_.`: 定义结束。)

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

```

\__regex_compile_/H: 942 \cs_set_protected:Npn \__regex_tmp:w #1#2
\__regex_compile_/s: 943 {
\__regex_compile_/S: 944     \cs_new_protected:cpe { __regex_compile_/#1: }
\__regex_compile_/v: 945     { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
\__regex_compile_/V: 946     \cs_new_protected:cpe { __regex_compile_/#2: }
\__regex_compile_/w: 947     {
\__regex_compile_/W: 948         \__regex_compile_one:n
\__regex_compile_/N: 949         { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#1: } } }
950     }
951 }
952 \__regex_tmp:w d D
953 \__regex_tmp:w h H
954 \__regex_tmp:w s S
955 \__regex_tmp:w v V
956 \__regex_tmp:w w W
957 \cs_new_protected:cpn { __regex_compile_/N: }
958 { \__regex_compile_one:n \__regex_prop_N: }

```

(`__regex_compile_/d:` 以及其它的定义结束。)

9.3.8 Anchoring and simple assertions

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

```

\__regex_compile_/b: 959 \cs_new_protected:Npn \__regex_compile_anchor_letter:NNN #1#2#3
\__regex_compile_/B: 960 {
\__regex_compile_/^: 961     \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw_error:N #1 }
\__regex_compile_/$: 962     {

```

```

963         \tl_build_put_right:Nn \l__regex_build_tl
964         { \__regex_assertion:Nn #2 {#3} }
965     }
966 }
967 \cs_new_protected:cpn { __regex_compile_/A: }
968 { \__regex_compile_anchor_letter:NNN A \c_true_bool \__regex_A_test: }
969 \cs_new_protected:cpn { __regex_compile_/G: }
970 { \__regex_compile_anchor_letter:NNN G \c_true_bool \__regex_G_test: }
971 \cs_new_protected:cpn { __regex_compile_/Z: }
972 { \__regex_compile_anchor_letter:NNN Z \c_true_bool \__regex_Z_test: }
973 \cs_new_protected:cpn { __regex_compile_/z: }
974 { \__regex_compile_anchor_letter:NNN z \c_true_bool \__regex_Z_test: }
975 \cs_new_protected:cpn { __regex_compile_/b: }
976 { \__regex_compile_anchor_letter:NNN b \c_true_bool \__regex_b_test: }
977 \cs_new_protected:cpn { __regex_compile_/B: }
978 { \__regex_compile_anchor_letter:NNN B \c_false_bool \__regex_b_test: }
979 \cs_set_protected:Npn \__regex_tmp:w #1#2
980 {
981     \cs_new_protected:cpn { __regex_compile_#1: }
982     {
983         \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw:N #1 }
984         {
985             \tl_build_put_right:Nn \l__regex_build_tl
986             { \__regex_assertion:Nn \c_true_bool {#2} }
987         }
988     }
989 }
990 \exp_args:Ne \__regex_tmp:w { \iow_char:N ^ } { \__regex_A_test: }
991 \exp_args:Ne \__regex_tmp:w { \iow_char:N $ } { \__regex_Z_test: }

```

(*__regex_compile_anchor_letter:NNN* 以及其它的定义结束。)

9.3.9 Character classes

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

```

992 \cs_new_protected:cpn { __regex_compile_[: }
993 {
994     \__regex_if_in_class:TF
995     {

```

```

996         \if_int_compare:w \l__regex_mode_int >
997         \c__regex_catcode_in_class_mode_int
998         \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
999         \fi:
1000         \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
1001         \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
1002         \if_int_odd:w \l__regex_mode_int \else:
1003         \exp_after:wN \__regex_compile_quantifier:w
1004         \fi:
1005     }
1006     { \__regex_compile_raw:N ] }
1007 }

```

(`__regex_compile_]`: 定义结束。)

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

```

1008 \cs_new_protected:cpn { __regex_compile_[: }
1009 {
1010     \__regex_if_in_class:TF
1011     { \__regex_compile_class_posix_test:w }
1012     {
1013         \__regex_if_within_catcode:TF
1014         {
1015             \exp_after:wN \__regex_compile_class_catcode:w
1016             \int_use:N \l__regex_catcodes_int ;
1017         }
1018         { \__regex_compile_class_normal:w }
1019     }
1020 }

```

(`__regex_compile_[:` 定义结束。)

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

```

1021 \cs_new_protected:Npn \__regex_compile_class_normal:w
1022 {
1023     \__regex_compile_class:TFNN

```

```

1024      { \_regex\_class:NnnnN \c\_true\_bool }
1025      { \_regex\_class:NnnnN \c\_false\_bool }
1026    }

```

(_regex_compile_class_normal:w 定义结束。)

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

```

1027 \cs\_new\_protected:Npn \_regex\_compile\_class\_catcode:w #1;
1028 {
1029   \if\_int\_compare:w \l\_regex\_mode\_int = \c\_regex\_catcode\_mode\_int
1030     \tl\_build\_put\_right:Nn \l\_regex\_build\_tl
1031       { \_regex\_class:NnnnN \c\_true\_bool { \if\_false: } \fi: }
1032   \fi:
1033   \int\_set\_eq:NN \l\_regex\_catcodes\_int \l\_regex\_default\_catcodes\_int
1034   \_regex\_compile\_class:TFNN
1035     { \_regex\_item\_catcode:nT {#1} }
1036     { \_regex\_item\_catcode\_reverse:nT {#1} }
1037 }

```

(_regex_compile_class_catcode:w 定义结束。)

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

```

1038 \cs\_new\_protected:Npn \_regex\_compile\_class:TFNN #1#2#3#4
1039 {
1040   \l\_regex\_mode\_int = \int\_value:w \l\_regex\_mode\_int 3 \exp\_stop\_f:
1041   \_regex\_two\_if\_eq:NNNTF #3 #4 \_regex\_compile\_special:N ^
1042   {
1043     \tl\_build\_put\_right:Nn \l\_regex\_build\_tl { #2 { \if\_false: } \fi: }
1044     \_regex\_compile\_class:NN
1045   }
1046   {
1047     \tl\_build\_put\_right:Nn \l\_regex\_build\_tl { #1 { \if\_false: } \fi: }
1048     \_regex\_compile\_class:NN #3 #4
1049   }
1050 }
1051 \cs\_new\_protected:Npn \_regex\_compile\_class:NN #1#2

```

```

1052 {
1053     \token_if_eq_charcode:NNTF #2 ]
1054     { \__regex_compile_raw:N #2 }
1055     { #1 #2 }
1056 }

```

(__regex_compile_class:TFNN 和 __regex_compile_class:NN 定义结束。)

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

```

1057 \cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
1058 {
1059     \token_if_eq_meaning:NNT \__regex_compile_special:N #1
1060     {
1061         \str_case:nn { #2 }
1062         {
1063             : { \__regex_compile_class_posix:NNNNw }
1064             = {
1065                 \msg_warning:nne { regex }
1066                 { posix-unsupported } { = }
1067             }
1068             . {
1069                 \msg_warning:nne { regex }
1070                 { posix-unsupported } { . }
1071             }
1072         }
1073     }
1074     \__regex_compile_raw:N [ #1 #2
1075 }
1076 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
1077 {
1078     \__regex_two_if_eq:NNNTF #5 #6 \__regex_compile_special:N ^
1079     {
1080         \bool_set_false:N \l__regex_internal_bool
1081         \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
1082         \__regex_compile_class_posix_loop:w
1083     }

```

```

1084     {
1085         \bool_set_true:N \l__regex_internal_bool
1086         \__kernel_tl_set:N \l__regex_internal_a_tl { \if_false: } \fi:
1087         \__regex_compile_class_posix_loop:w #5 #6
1088     }
1089 }
1090 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
1091 {
1092     \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
1093     { #2 \__regex_compile_class_posix_loop:w }
1094     { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
1095 }
1096 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
1097 {
1098     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N :
1099     { \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ] }
1100     { \use_ii:nn }
1101     {
1102         \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
1103         {
1104             \__regex_compile_one:n
1105             {
1106                 \bool_if:NTF \l__regex_internal_bool \use:n \__regex_item_reverse:n
1107                 { \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : } }
1108             }
1109         }
1110         {
1111             \msg_warning:nne { regex } { posix-unknown }
1112             { \l__regex_internal_a_tl }
1113             \__regex_compile_abort_tokens:e
1114             {
1115                 [: \bool_if:NF \l__regex_internal_bool { ^ }
1116                 \l__regex_internal_a_tl :]
1117             }
1118         }
1119     }
1120 }
1121 {
1122     \msg_error:nnee { regex } { posix-missing-close }
1123     { [: \l__regex_internal_a_tl } { #2 #4 }
1124     \__regex_compile_abort_tokens:e { [: \l__regex_internal_a_tl }
1125     #1 #2 #3 #4
1126 }

```



```
1126 }
```

(`_regex_compile_class_posix_test:w` 以及其它的定义结束。)

9.3.10 Groups and alternations

`_regex_compile_group_begin:N`

`_regex_compile_group_end:`

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

```
1127 \cs_new_protected:Npn \_regex_compile_group_begin:N #1
1128 {
1129   \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
1130   \_regex_mode_quit_c:
1131   \group_begin:
1132     \tl_build_begin:N \l__regex_build_tl
1133     \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
1134     \int_incr:N \l__regex_group_level_int
1135     \tl_build_put_right:Nn \l__regex_build_tl
1136       { \_regex_branch:n { \if_false: } \fi: }
1137   }
1138 \cs_new_protected:Npn \_regex_compile_group_end:
1139 {
1140   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
1141     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
1142     \tl_build_end:N \l__regex_build_tl
1143     \exp_args:NNNe
1144     \group_end:
1145     \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
1146     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
1147     \exp_after:wN \_regex_compile_quantifier:w
1148   \else:
1149     \msg_warning:nn { regex } { extra-rparen }
1150     \exp_after:wN \_regex_compile_raw:N \exp_after:wN )
1151   \fi:
1152 }
```

(`_regex_compile_group_begin:N` 和 `_regex_compile_group_end:` 定义结束。)

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

```
1153 \cs_new_protected:cpn { \_regex_compile_(: }
1154 {
1155   \_regex_if_in_class:TF { \_regex_compile_raw:N ( }
1156   {
1157     \if_int_compare:w \l__regex_mode_int =
1158       \c__regex_catcode_in_class_mode_int
1159     \msg_error:nn { regex } { c-lparen-in-class }
1160     \exp_after:wN \_regex_compile_raw:N \exp_after:wN (
1161     \else:
1162       \exp_after:wN \_regex_compile_lparen:w
1163     \fi:
1164   }
1165 }
1166 \cs_new_protected:Npn \_regex_compile_lparen:w #1#2#3#4
1167 {
1168   \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N ?
1169   {
1170     \cs_if_exist_use:cF
1171     { \_regex_compile_special_group_\token_to_str:N #4 :w }
1172     {
1173       \msg_warning:nne { regex } { special-group-unknown }
1174       { (? #4 }
1175       \_regex_compile_group_begin:N \_regex_group:nnnN
1176       \_regex_compile_raw:N ? #3 #4
1177     }
1178   }
1179   {
1180     \_regex_compile_group_begin:N \_regex_group:nnnN
1181     #1 #2 #3 #4
1182   }
1183 }
```

(`_regex_compile_(:` 定义结束。)

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

```
1184 \cs_new_protected:cpn { \_regex_compile_|: }
```

```

1185 {
1186     \__regex_if_in_class:TF { \__regex_compile_raw:N | }
1187     {
1188         \tl_build_put_right:Nn \l__regex_build_tl
1189         { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
1190     }
1191 }

```

(__regex_compile_/: 定义结束。)

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

```

1192 \cs_new_protected:cpn { __regex_compile_): }
1193 {
1194     \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
1195     { \__regex_compile_group_end: }
1196 }

```

(__regex_compile_): 定义结束。)

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

```

1197 \cs_new_protected:cpn { __regex_compile_special_group::w }
1198 { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
1199 \cs_new_protected:cpn { __regex_compile_special_group|:w }
1200 { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }

```

(__regex_compile_special_group::w 和 __regex_compile_special_group|:w 定义结束。)

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

```

1201 \cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
1202 {
1203     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N )
1204     {
1205         \cs_set:Npn \__regex_item_equal:n
1206         { \__regex_item_caseless_equal:n }
1207         \cs_set:Npn \__regex_item_range:nn
1208         { \__regex_item_caseless_range:nn }
1209     }
1210     {
1211         \msg_warning:nne { regex } { unknown-option } { (?i #2 }
1212         \__regex_compile_raw:N (
1213         \__regex_compile_raw:N ?

```

```

1214     \_regex_compile_raw:N i
1215     #1 #2
1216   }
1217 }
1218 \cs_new_protected:cpn { \_regex_compile_special_group-:w } #1#2#3#4
1219 {
1220   \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_raw:N i
1221   { \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ) }
1222   { \use_ii:nn }
1223   {
1224     \cs_set:Npn \_regex_item_equal:n
1225       { \_regex_item_caseful_equal:n }
1226     \cs_set:Npn \_regex_item_range:nn
1227       { \_regex_item_caseful_range:nn }
1228   }
1229   {
1230     \msg_warning:nne { regex } { unknown-option } { (?-#2#4 }
1231     \_regex_compile_raw:N (
1232     \_regex_compile_raw:N ?
1233     \_regex_compile_raw:N -
1234     #1 #2 #3 #4
1235   }
1236 }

```

(`_regex_compile_special_group_i:w` 和 `_regex_compile_special_group-:w` 定义结束。)

9.3.11 Catcodes and csnames

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

```

1237 \cs_new_protected:cpn { \_regex_compile_/c: }
1238 { \_regex_chk_c_allowed:T { \_regex_compile_c_test:NN } }
1239 \cs_new_protected:Npn \_regex_compile_c_test:NN #1#2
1240 {
1241   \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
1242   {
1243     \int_if_exist:cTF { c__regex_catcode_#2_int }
1244     {
1245       \int_set_eq:Nc \l__regex_catcodes_int
1246       { c__regex_catcode_#2_int }
1247       \l__regex_mode_int

```

```

1248         = \if_case:w \l__regex_mode_int
1249         \c__regex_catcode_mode_int
1250         \else:
1251         \c__regex_catcode_in_class_mode_int
1252         \fi:
1253         \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
1254     }
1255 }
1256 { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
1257 {
1258     \msg_error:nne { regex } { c-missing-category } {#2}
1259     #1 #2
1260 }
1261 }

```

(__regex_compile_/c: 和 __regex_compile_c_test:NN 定义结束。)

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

```

1262 \cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
1263 {
1264     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
1265     {
1266         \token_if_eq_charcode:NNTF #2 .
1267         { \use_none:n }
1268         { \token_if_eq_charcode:NNF #2 ( } % )
1269     }
1270     { \use:n }
1271     { \msg_error:nnn { regex } { c-C-invalid } {#2} }
1272     #1 #2
1273 }

```

(__regex_compile_c_C:NN 定义结束。)

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

__regex_compile_c_lbrack_loop:NN
__regex_compile_c_lbrack_add:N
__regex_compile_c_lbrack_end:

```

1274 \cs_new_protected:cpn { __regex_compile_c[:w } #1#2
1275 {
1276     \l__regex_mode_int
1277     = \if_case:w \l__regex_mode_int
1278     \c__regex_catcode_mode_int
1279     \else:
1280     \c__regex_catcode_in_class_mode_int

```

```

1281     \fi:
1282     \int_zero:N \l__regex_catcodes_int
1283     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ^
1284     {
1285         \bool_set_false:N \l__regex_catcodes_bool
1286         \__regex_compile_c_lbrack_loop:NN
1287     }
1288     {
1289         \bool_set_true:N \l__regex_catcodes_bool
1290         \__regex_compile_c_lbrack_loop:NN
1291         #1 #2
1292     }
1293 }
1294 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
1295 {
1296     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
1297     {
1298         \int_if_exist:cTF { c__regex_catcode_#2_int }
1299         {
1300             \exp_args:Nc \__regex_compile_c_lbrack_add:N
1301             { c__regex_catcode_#2_int }
1302             \__regex_compile_c_lbrack_loop:NN
1303         }
1304     }
1305     {
1306         \token_if_eq_charcode:NNTF #2 ]
1307         { \__regex_compile_c_lbrack_end: }
1308     }
1309     {
1310         \msg_error:nne { regex } { c-missing-rbrack } {#2}
1311         \__regex_compile_c_lbrack_end:
1312         #1 #2
1313     }
1314 }
1315 \cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
1316 {
1317     \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
1318     \else:
1319         \int_add:Nn \l__regex_catcodes_int {#1}
1320     \fi:
1321 }
1322 \cs_new_protected:Npn \__regex_compile_c_lbrack_end:

```

```

1323 {
1324     \if_meaning:w \c_false_bool \l__regex_catcodes_bool
1325     \int_set:Nn \l__regex_catcodes_int
1326     { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
1327     \fi:
1328 }

```

(__regex_compile_c_l:w 以及其它的定义结束。)

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

```

1329 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
1330 {
1331     \__regex_compile:w
1332     \__regex_disable_submatches:
1333     \l__regex_mode_int
1334     = \if_case:w \l__regex_mode_int
1335         \c__regex_cs_mode_int
1336     \else:
1337         \c__regex_cs_in_class_mode_int
1338     \fi:
1339 }

```

(__regex_compile_c_{: 定义结束。)

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

```

1340 \cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
1341 {
1342     \__regex_if_in_cs:TF
1343     { \msg_error:nnn { regex } { cu-lbrace } { c } }
1344     { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
1345 }

```

(__regex_compile_{: 定义结束。)

__regex_cs Non-escaped right braces are only special if they appear when compiling the regular

__regex_compile_{: expression for a csname, but not within a class: \c{[{}]} matches the control

__regex_compile_end_cs: sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class

__regex_compile_cs_aux:Nn or group). Then insert the corresponding test in the outer regex. As an optimization,

__regex_compile_cs_aux:NNnnN

if the control sequence test simply consists of several explicit possibilities (branches) then use `__regex_item_exact_cs:n` with an argument consisting of all possibilities separated by `\scan_stop:`.

```

1346 \flag_new:n { __regex_cs }
1347 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
1348 {
1349   \__regex_if_in_cs:TF
1350     { \__regex_compile_end_cs: }
1351     { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
1352 }
1353 \cs_new_protected:Npn \__regex_compile_end_cs:
1354 {
1355   \__regex_compile_end:
1356   \flag_clear:n { __regex_cs }
1357   \__kernel_tl_set:Ne \l__regex_internal_a_tl
1358   {
1359     \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
1360     \q__regex_nil \q__regex_nil \q__regex_recursion_stop
1361   }
1362   \exp_args:Ne \__regex_compile_one:n
1363   {
1364     \flag_if_raised:nTF { __regex_cs }
1365       { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
1366       {
1367         \__regex_item_exact_cs:n
1368           { \tl_tail:N \l__regex_internal_a_tl }
1369       }
1370   }
1371 }
1372 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
1373 {
1374   \cs_if_eq:NNTF #1 \__regex_branch:n
1375     {
1376       \scan_stop:
1377       \__regex_compile_cs_aux:NNnnN #2
1378       \q__regex_nil \q__regex_nil \q__regex_nil
1379       \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
1380       \__regex_compile_cs_aux:Nn
1381     }
1382     {
1383       \__regex_quark_if_nil:NF #1 { \flag_ensure_raised:n { __regex_cs } }
1384       \__regex_use_none_delimit_by_q_recursion_stop:w

```



```

1385     }
1386 }
1387 \cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
1388 {
1389     \bool_lazy_all:nTF
1390     {
1391         { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
1392         {#2}
1393         { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
1394         { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
1395         { \int_compare_p:nNn {#5} = { 0 } }
1396     }
1397     {
1398         \prg_replicate:nn {#4}
1399         { \char_generate:nn { \use_ii:nn #3 } {12} }
1400         \__regex_compile_cs_aux:NNnnnN
1401     }
1402     {
1403         \__regex_quark_if_nil:NF #1
1404         {
1405             \flag_ensure_raised:n { __regex_cs }
1406             \__regex_use_i_delimit_by_q_recursion_stop:nw
1407         }
1408         \__regex_use_none_delimit_by_q_recursion_stop:w
1409     }
1410 }

```

(*__regex_cs* 以及其它的定义结束。)

9.3.12 Raw token lists with \u

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

```

1411 \cs_new_protected:cpn { __regex_compile_/u: } #1#2
1412 {
1413     \__regex_if_in_class_or_catcode:TF
1414     { \__regex_compile_raw_error:N u #1 #2 }
1415     {
1416         \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_raw:N r
1417         { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
1418         { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }

```

```

1419     }
1420 }

```

(`_regex_compile_u`: 定义结束。)

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

```

1421 \cs_new:Npn \_regex_compile_u_brace:NNN #1#2#3
1422 {
1423   \_regex_two_if_eq:NNNTF #2 #3 \_regex_compile_special:N \c_left_brace_str
1424   {
1425     \tl_set:Nn \l__regex_internal_b_tl {#1}
1426     \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
1427     \_regex_compile_u_loop:NN
1428   }
1429   {
1430     \msg_error:nn { regex } { u-missing-lbrace }
1431     \token_if_eq_meaning:NNTF #1 \_regex_compile_ur_end:
1432     { \_regex_compile_raw:N u \_regex_compile_raw:N r }
1433     { \_regex_compile_raw:N u }
1434     #2 #3
1435   }
1436 }

```

(`_regex_compile_u_brace:NNN` 定义结束。)

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

```

1437 \cs_new:Npn \_regex_compile_u_loop:NN #1#2
1438 {
1439   \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
1440   { #2 \_regex_compile_u_loop:NN }
1441   {
1442     \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
1443     {
1444       \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
1445       { \if_false: { \fi: } \l__regex_internal_b_tl }
1446       {

```

```

1447         \if_charcode:w \c_left_brace_str #2
1448         \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
1449         \else:
1450             #2
1451         \fi:
1452         \__regex_compile_u_loop:NN
1453     }
1454 }
1455 {
1456     \if_false: { \fi: }
1457     \msg_error:nne { regex } { u-missing-rbrace } {#2}
1458     \l__regex_internal_b_tl
1459     #1 #2
1460 }
1461 }
1462 }

```

(`__regex_compile_u_loop:NN` 定义结束。)

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

```

1463 \cs_new_protected:Npn \__regex_compile_ur_end:
1464 {
1465     \group_begin:
1466     \cs_set:Npn \__regex_group:nnnN { \__regex_group_no_capture:nnnN }
1467     \cs_set:Npn \__regex_group_resetting:nnnN { \__regex_group_no_capture:nnnN }
1468     \exp_args:NNe
1469     \group_end:
1470     \__regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
1471 }
1472 \cs_new_protected:Npn \__regex_compile_ur:n #1
1473 {
1474     \tl_if_empty:oTF { \__regex_compile_ur_aux:w #1 {} } ? ? \q__regex_nil }
1475     { \__regex_compile_if_quantifier:TFw }
1476     { \use_i:nn }
1477     {
1478         \tl_build_put_right:Nn \l__regex_build_tl

```

```

1479         { \__regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
1480         \__regex_compile_quantifier:w
1481     }
1482     { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
1483 }
1484 \cs_new:Npn \__regex_compile_ur_aux:w \__regex_branch:n #1#2#3 \q__regex_nil {#2}

```

(*__regex_compile_ur_end:*, *__regex_compile_ur:n*, 和 *__regex_compile_ur_aux:w* 定义结束。)

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

```

1485 \cs_new_protected:Npn \__regex_compile_u_end:
1486 {
1487     \__regex_compile_if_quantifier:TFw
1488     {
1489         \tl_build_put_right:Nn \l__regex_build_tl
1490         {
1491             \__regex_group_no_capture:nnnN { \if_false: } \fi:
1492             \__regex_branch:n { \if_false: } \fi:
1493         }
1494         \__regex_compile_u_payload:
1495         \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
1496         \__regex_compile_quantifier:w
1497     }
1498     { \__regex_compile_u_payload: }
1499 }
1500 \cs_new_protected:Npn \__regex_compile_u_payload:
1501 {
1502     \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
1503     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
1504         \__regex_compile_u_not_cs:
1505     \else:
1506         \__regex_compile_u_in_cs:
1507     \fi:
1508 }

```

(*__regex_compile_u_end:* 和 *__regex_compile_u_payload:* 定义结束。)

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

```
1509 \cs_new_protected:Npn \__regex_compile_u_in_cs:
1510 {
1511   \__kernel_tl_gset:Nc \g__regex_internal_tl
1512   {
1513     \exp_args:No \__kernel_str_to_other_fast:n
1514     { \l__regex_internal_a_tl }
1515   }
1516   \tl_build_put_right:Ne \l__regex_build_tl
1517   {
1518     \tl_map_function:NN \g__regex_internal_tl
1519     \__regex_compile_u_in_cs_aux:n
1520   }
1521 }
1522 \cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
1523 {
1524   \__regex_class:NnnnN \c_true_bool
1525   { \__regex_item_caseful_equal:n { \int_value:w `#1 } }
1526   { 1 } { 0 } \c_false_bool
1527 }
```

(`__regex_compile_u_in_cs:` 定义结束。)

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

```
1528 \cs_new_protected:Npn \__regex_compile_u_not_cs:
1529 {
1530   \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
1531   {
1532     \tl_build_put_right:Ne \l__regex_build_tl
1533     {
1534       \__regex_class:NnnnN \c_true_bool
1535       {
1536         \if_int_compare:w "##3 = \c_zero_int
1537         \__regex_item_exact_cs:n
1538         { \exp_after:wN \cs_to_str:N ##1 }
1539       }
1540       \else:
1541         \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
```

```

1541         \fi:
1542     }
1543     { 1 } { 0 } \c_false_bool
1544 }
1545 }
1546 }

```

(*_regex_compile_u_not_cs*: 定义结束。)

9.3.13 Other

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

```

1547 \cs_new_protected:cpn { \_regex_compile_/K: }
1548 {
1549     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
1550     { \tl_build_put_right:Nn \l__regex_build_tl { \_regex_command_K: } }
1551     { \_regex_compile_raw_error:N K }
1552 }

```

(*_regex_compile_/K*: 定义结束。)

9.3.14 Showing regexes

_regex_clean_bool:n Before showing a regex we check that it is “clean” in the sense that it has the correct internal structure. We do this (in the implementation of *\regex_show:N* and *\regex_log:N*) by comparing it with a cleaned-up version of the same regex.
_regex_clean_int:n
_regex_clean_int_aux:N Along the way we also need similar functions for other types: all *_regex_clean_<type>:n* functions produce valid *<type>* tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.
_regex_clean_regex:n
_regex_clean_regex_loop:w
_regex_clean_branch:n
_regex_clean_branch_loop:n
_regex_clean_assertion:Nn
_regex_clean_class:NnnnN
_regex_clean_group:nnnN
_regex_clean_class:n
_regex_clean_class_loop:nnn
_regex_clean_exact_cs:n
_regex_clean_exact_cs:w

```

1553 \cs_new:Npn \_regex_clean_bool:n #1
1554 {
1555     \tl_if_single:nTF {#1}
1556     { \bool_if:NTF #1 \c_true_bool \c_false_bool }
1557     { \c_true_bool }
1558 }
1559 \cs_new:Npn \_regex_clean_int:n #1
1560 {
1561     \tl_if_head_eq_meaning:nNTF {#1} -
1562     { - \exp_args:No \_regex_clean_int:n { \use_none:n #1 } }
1563     { \int_eval:n { 0 \str_map_function:nN {#1} \_regex_clean_int_aux:N } }

```

```

1564     }
1565     \cs_new:Npn \__regex_clean_int_aux:N #1
1566     {
1567         \if_int_compare:w 1 < 1 #1 ~
1568         #1
1569         \else:
1570             \exp_after:wN \str_map_break:
1571         \fi:
1572     }
1573     \cs_new:Npn \__regex_clean_regex:n #1
1574     {
1575         \__regex_clean_regex_loop:w #1
1576         \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
1577     }
1578     \cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
1579     {
1580         \quark_if_recursion_tail_stop:n {#2}
1581         \__regex_branch:n { \__regex_clean_branch:n {#2} }
1582         \__regex_clean_regex_loop:w
1583     }
1584     \cs_new:Npn \__regex_clean_branch:n #1
1585     {
1586         \__regex_clean_branch_loop:n #1
1587         ? ? ? ? ? \prg_break_point:
1588     }
1589     \cs_new:Npn \__regex_clean_branch_loop:n #1
1590     {
1591         \tl_if_single:nF {#1} { \prg_break: }
1592         \token_case_meaning:NnF #1
1593         {
1594             \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
1595             \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
1596             \__regex_class:NnnN { #1 \__regex_clean_class:NnnN }
1597             \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
1598             \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
1599             \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
1600         }
1601         { \prg_break: }
1602     }
1603     \cs_new:Npn \__regex_clean_assertion:Nn #1#2
1604     {
1605         \__regex_clean_bool:n {#1}

```

```

1606     \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
1607     \token_case_meaning:NnTF #2
1608     {
1609         \__regex_A_test: { }
1610         \__regex_G_test: { }
1611         \__regex_Z_test: { }
1612         \__regex_b_test: { }
1613     }
1614     { {#2} }
1615     { { \__regex_A_test: } \prg_break: }
1616     \__regex_clean_branch_loop:n
1617 }
1618 \cs_new:Npn \__regex_clean_class:NnnnN #1#2#3#4#5
1619 {
1620     \__regex_clean_bool:n {#1}
1621     { \__regex_clean_class:n {#2} }
1622     { \int_max:nn { 0 } { \__regex_clean_int:n {#3} } }
1623     { \int_max:nn { -1 } { \__regex_clean_int:n {#4} } }
1624     \__regex_clean_bool:n {#5}
1625     \__regex_clean_branch_loop:n
1626 }
1627 \cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4
1628 {
1629     { \__regex_clean_regex:n {#1} }
1630     { \int_max:nn { 0 } { \__regex_clean_int:n {#2} } }
1631     { \int_max:nn { -1 } { \__regex_clean_int:n {#3} } }
1632     \__regex_clean_bool:n {#4}
1633     \__regex_clean_branch_loop:n
1634 }
1635 \cs_new:Npn \__regex_clean_class:n #1
1636 { \__regex_clean_class_loop:nnn #1 ????? \prg_break_point: }

```

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

```

1637 \cs_new:Npn \__regex_clean_class_loop:nnn #1#2#3
1638 {
1639     \tl_if_single:nF {#1} { \prg_break: }
1640     \token_case_meaning:NnTF #1
1641     {
1642         \__regex_item_cs:n { #1 { \__regex_clean_regex:n {#2} } }

```



```

1643     \_regex_item_exact_cs:n { #1 { \_regex_clean_exact_cs:n {#2} } }
1644     \_regex_item_caseful_equal:n { #1 { \_regex_clean_int:n {#2} } }
1645     \_regex_item_caseless_equal:n { #1 { \_regex_clean_int:n {#2} } }
1646     \_regex_item_reverse:n { #1 { \_regex_clean_class:n {#2} } }
1647 }
1648 { \_regex_clean_class_loop:nnn {#3} }
1649 {
1650     \token_case_meaning:NnTF #1
1651     {
1652         \_regex_item_caseful_range:nn { }
1653         \_regex_item_caseless_range:nn { }
1654         \_regex_item_exact:nn { }
1655     }
1656     {
1657         #1 { \_regex_clean_int:n {#2} } { \_regex_clean_int:n {#3} }
1658         \_regex_clean_class_loop:nnn
1659     }
1660     {
1661         \token_case_meaning:NnTF #1
1662         {
1663             \_regex_item_catcode:nT { }
1664             \_regex_item_catcode_reverse:nT { }
1665         }
1666         {
1667             #1 { \_regex_clean_int:n {#2} } { \_regex_clean_class:n {#3} }
1668             \_regex_clean_class_loop:nnn
1669         }
1670         {
1671             \exp_args:Nf \str_case:nnTF
1672             {
1673                 \exp_args:Nf \str_range:nnn
1674                 { \cs_to_str:N #1 } { 1 } { 13 }
1675             }
1676             {
1677                 { __regex_prop_ } { }
1678                 { __regex_posix } { }
1679             }
1680             {
1681                 #1
1682                 \_regex_clean_class_loop:nnn {#2} {#3}
1683             }
1684             { \prg_break: }

```

```

1685         }
1686     }
1687 }
1688 }
1689 \cs_new:Npn \__regex_clean_exact_cs:n #1
1690 {
1691     \exp_last_unbraced:Nf \use_none:n
1692     {
1693         \__regex_clean_exact_cs:w #1
1694         \scan_stop: \q_recursion_tail \scan_stop:
1695         \q_recursion_stop
1696     }
1697 }
1698 \cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
1699 {
1700     \quark_if_recursion_tail_stop:n {#1}
1701     \scan_stop: \tl_to_str:n {#1}
1702     \__regex_clean_exact_cs:w
1703 }

```

(`__regex_clean_bool:n` 以及其它的定义结束。)

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

```

1704 \cs_new_protected:Npn \__regex_show:N #1
1705 {
1706     \group_begin:
1707     \tl_build_begin:N \l__regex_build_tl
1708     \cs_set_protected:Npn \__regex_branch:n
1709     {
1710         \seq_pop_right:NN \l__regex_show_prefix_seq
1711         \l__regex_internal_a_tl
1712         \__regex_show_one:n { +-branch }
1713         \seq_put_right:No \l__regex_show_prefix_seq
1714         \l__regex_internal_a_tl
1715         \use:n
1716     }
1717     \cs_set_protected:Npn \__regex_group:nnnN
1718     { \__regex_show_group_aux:nnnnN { } }
1719     \cs_set_protected:Npn \__regex_group_no_capture:nnnN
1720     { \__regex_show_group_aux:nnnnN { ~(no~capture) } }

```

```

1721 \cs_set_protected:Npn \__regex_group_resetting:nnnN
1722   { \__regex_show_group_aux:nnnnN { ~(resetting) } }
1723 \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
1724 \cs_set_protected:Npn \__regex_command_K:
1725   { \__regex_show_one:n { reset~match~start~(\iow_char:N\K) } }
1726 \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
1727   {
1728     \__regex_show_one:n
1729     { \bool_if:NF ##1 { negative~ } assertion:~##2 }
1730   }
1731 \cs_set:Npn \__regex_b_test: { word~boundary }
1732 \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\Z) }
1733 \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\A) }
1734 \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\G) }
1735 \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
1736   { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
1737 \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
1738   {
1739     \__regex_show_one:n
1740     { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
1741   }
1742 \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
1743   { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
1744 \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
1745   {
1746     \__regex_show_one:n
1747     { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
1748   }
1749 \cs_set_protected:Npn \__regex_item_catcode:nT
1750   { \__regex_show_item_catcode:NnT \c_true_bool }
1751 \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
1752   { \__regex_show_item_catcode:NnT \c_false_bool }
1753 \cs_set_protected:Npn \__regex_item_reverse:n
1754   { \__regex_show_scope:nn { Reversed~match } }
1755 \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
1756   { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
1757 \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
1758 \cs_set_protected:Npn \__regex_item_cs:n
1759   { \__regex_show_scope:nn { control~sequence } }
1760 \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
1761 \seq_clear:N \l__regex_show_prefix_seq
1762 \__regex_show_push:n { ~ }

```

```

1763     \cs_if_exist_use:N #1
1764     \tl_build_end:N \l__regex_build_tl
1765     \exp_args:NNNo
1766     \group_end:
1767     \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
1768   }

```

(*__regex_show:N* 定义结束。)

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

```

1769 \cs_new:Npn \__regex_show_char:n #1
1770 {
1771   \int_eval:n {#1}
1772   \int_compare:nT { 32 <= #1 <= 126 }
1773     { ~ ( \char_generate:nn {#1} {12} ) }
1774 }

```

(*__regex_show_char:n* 定义结束。)

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

```

1775 \cs_new_protected:Npn \__regex_show_one:n #1
1776 {
1777   \int_incr:N \l__regex_show_lines_int
1778   \tl_build_put_right:Ne \l__regex_build_tl
1779     {
1780       \exp_not:N \iow_newline:
1781       \seq_map_function:NN \l__regex_show_prefix_seq \use:n
1782       #1
1783     }
1784 }

```

(*__regex_show_one:n* 定义结束。)

__regex_show_push:n Enter and exit levels of nesting. The **scope** function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.

__regex_show_pop:

```

__regex_show_scope:nn 1785 \cs_new_protected:Npn \__regex_show_push:n #1
1786 { \seq_put_right:Ne \l__regex_show_prefix_seq { #1 ~ } }
1787 \cs_new_protected:Npn \__regex_show_pop:
1788 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
1789 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
1790 {

```

```

1791     \_regex_show_one:n {#1}
1792     \_regex_show_push:n { ~ }
1793     #2
1794     \_regex_show_pop:
1795 }

```

(_regex_show_push:n, _regex_show_pop:, 和 _regex_show_scope:nn 定义结束。)

_regex_show_group_aux:nnnnN

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

```

1796 \cs_new_protected:Npn \_regex_show_group_aux:nnnnN #1#2#3#4#5
1797 {
1798     \_regex_show_one:n { ,-group~begin #1 }
1799     \_regex_show_push:n { | }
1800     \use_ii:nn #2
1801     \_regex_show_pop:
1802     \_regex_show_one:n
1803     { ~-group~end \_regex_msg_repeated:nnN {#3} {#4} #5 }
1804 }

```

(_regex_show_group_aux:nnnnN 定义结束。)

_regex_show_class:NnnnN

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

```

1805 \cs_set:Npn \_regex_show_class:NnnnN #1#2#3#4#5
1806 {
1807     \group_begin:
1808     \tl_build_begin:N \l__regex_build_tl
1809     \int_zero:N \l__regex_show_lines_int
1810     \_regex_show_push:n {~}
1811     #2
1812     \int_compare:nTF { \l__regex_show_lines_int = 0 }
1813     {
1814         \group_end:
1815         \_regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
1816     }
1817     {

```

```

1818     \bool_if:nTF
1819     { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
1820     {
1821         \group_end:
1822         #2
1823         \tl_build_put_right:Nn \l__regex_build_tl
1824         { \__regex_msg_repeated:nnN {#3} {#4} #5 }
1825     }
1826     {
1827         \tl_build_end:N \l__regex_build_tl
1828         \exp_args:NNNo
1829         \group_end:
1830         \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
1831         \__regex_show_one:n
1832         {
1833             \bool_if:NTF #1 { Match } { Don't~match }
1834             \__regex_msg_repeated:nnN {#3} {#4} #5
1835         }
1836         \tl_build_put_right:Ne \l__regex_build_tl
1837         { \exp_not:o \l__regex_internal_a_tl }
1838     }
1839 }
1840 }

```

(`__regex_show_class:NnnnN` 定义结束。)

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

```

1841 \cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
1842 {
1843     \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
1844     \seq_set_filter:Nnn \l__regex_internal_seq \l__regex_internal_seq
1845     { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
1846     \__regex_show_scope:nn
1847     {
1848         categories~
1849         \seq_map_function:NN \l__regex_internal_seq \use:n
1850         , ~
1851         \bool_if:NF #1 { negative~ } class
1852     }
1853 }

```

(`__regex_show_item_catcode:NnT` 定义结束。)

`_regex_show_item_exact_cs:n`

```
1854 \cs_new_protected:Npn \_regex_show_item_exact_cs:n #1
1855 {
1856   \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
1857   \seq_set_map_e:NNn \l__regex_internal_seq
1858     \l__regex_internal_seq { \iow_char:N\\##1 }
1859   \_regex_show_one:n
1860     { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
1861 }
```

(`_regex_show_item_exact_cs:n` 定义结束。)

9.4 Building

9.4.1 Variables used while building

`\l__regex_min_state_int`

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

`\l__regex_max_state_int`

```
1862 \int_new:N \l__regex_min_state_int
1863 \int_set:Nn \l__regex_min_state_int { 1 }
1864 \int_new:N \l__regex_max_state_int
```

(`\l__regex_min_state_int` 和 `\l__regex_max_state_int` 定义结束。)

`\l__regex_left_state_int`

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

`\l__regex_right_state_int`

`\l__regex_left_state_seq`

`\l__regex_right_state_seq`

```
1865 \int_new:N \l__regex_left_state_int
1866 \int_new:N \l__regex_right_state_int
1867 \seq_new:N \l__regex_left_state_seq
1868 \seq_new:N \l__regex_right_state_seq
```

(`\l__regex_left_state_int` 以及其它的定义结束。)

`\l__regex_capturing_group_int`

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

```
1869 \int_new:N \l__regex_capturing_group_int
```

(`\l__regex_capturing_group_int` 定义结束。)

9.4.2 Framework

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

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

We strive to preserve the following properties while building.

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

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

```

1870 \cs_new_protected:Npn \__regex_build:n
1871   { \__regex_build_aux:Nn \c_true_bool }
1872 \cs_new_protected:Npn \__regex_build:N
1873   { \__regex_build_aux:NN \c_true_bool }
1874 \cs_new_protected:Npn \__regex_build_aux:Nn #1#2
1875   {
1876     \__regex_compile:n {#2}
1877     \__regex_build_aux:NN #1 \l__regex_internal_regex
1878   }
1879 \cs_new_protected:Npn \__regex_build_aux:NN #1#2
1880   {
1881     \__regex_standard_escapechar:
1882     \int_zero:N \l__regex_capturing_group_int
1883     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
1884     \__regex_build_new_state:
1885     \__regex_build_new_state:
1886     \__regex_toks_put_right:Nn \l__regex_left_state_int
1887     { \__regex_action_start_wildcard:N #1 }
1888     \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
1889     \__regex_toks_put_right:Nn \l__regex_right_state_int
1890     { \__regex_action_success: }
1891   }

```

(`__regex_build:n` 以及其它的定义结束。)

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

```

1892 \int_new:N \g__regex_case_int

```

(`\g__regex_case_int` 定义结束。)

`\l__regex_case_max_group_int` The largest group number appearing in any of the $\langle regex \rangle$ in the argument of `\regex_match_case:nn` and related functions.

```

1893 \int_new:N \l__regex_case_max_group_int

```

(`\l__regex_case_max_group_int` 定义结束。)

```

__regex_case_build:n See __regex_build:n, but with a loop.
__regex_case_build:e 1894 \cs_new_protected:Npn __regex_case_build:n #1
__regex_case_build_aux:Nn 1895 {
__regex_case_build_loop:n 1896   __regex_case_build_aux:Nn \c_true_bool {#1}
1897   \int_gzero:N \g__regex_case_int
1898 }
1899 \cs_generate_variant:Nn __regex_case_build:n { e }
1900 \cs_new_protected:Npn __regex_case_build_aux:Nn #1#2
1901 {
1902   __regex_standard_escapechar:
1903   \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
1904   __regex_build_new_state:
1905   __regex_build_new_state:
1906   __regex_toks_put_right:Nn \l__regex_left_state_int
1907   { __regex_action_start_wildcard:N #1 }
1908   %
1909   __regex_build_new_state:
1910   __regex_toks_put_left:Ne \l__regex_left_state_int
1911   { __regex_action_submatch:nN { 0 } < }
1912   __regex_push_lr_states:
1913   \int_zero:N \l__regex_case_max_group_int
1914   \int_gzero:N \g__regex_case_int
1915   \tl_map_inline:nn {#2}
1916   {
1917     \int_gincr:N \g__regex_case_int
1918     __regex_case_build_loop:n {##1}
1919   }
1920   \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
1921   __regex_pop_lr_states:
1922 }
1923 \cs_new_protected:Npn __regex_case_build_loop:n #1
1924 {
1925   \int_set:Nn \l__regex_capturing_group_int { 1 }
1926   __regex_compile_use:n {#1}
1927   \int_set:Nn \l__regex_case_max_group_int
1928   {
1929     \int_max:nn { \l__regex_case_max_group_int }
1930     { \l__regex_capturing_group_int }
1931   }
1932   \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
1933   \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
1934   __regex_toks_put_left:Ne \l__regex_right_state_int

```

```

1935     {
1936         \__regex_action_submatch:nN { 0 } >
1937         \int_gset:Nn \g__regex_case_int
1938         { \int_use:N \g__regex_case_int }
1939         \__regex_action_success:
1940     }
1941     \__regex_toks_clear:N \l__regex_max_state_int
1942     \seq_push:No \l__regex_right_state_seq
1943     { \int_use:N \l__regex_max_state_int }
1944     \int_incr:N \l__regex_max_state_int
1945 }

```

(`__regex_case_build:n`, `__regex_case_build_aux:Nn`, 和 `__regex_case_build_loop:n` 定义结束。)

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

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

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

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

```

1946 \cs_new_protected:Npn \__regex_build_for_cs:n #1
1947 {
1948     \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
1949     \__regex_build_new_state:
1950     \__regex_build_new_state:
1951     \__regex_push_lr_states:
1952     #1
1953     \__regex_pop_lr_states:
1954     \__regex_toks_put_right:Nn \l__regex_right_state_int
1955     {
1956         \if_int_compare:w -2 = \l__regex_curr_char_int
1957         \exp_after:wN \__regex_action_success:
1958         \fi:
1959     }

```

```
1960 }
```

(`_regex_build_for_cs:n` 定义结束。)

9.4.3 Helpers for building an nfa

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

```
1961 \cs_new_protected:Npn \_regex_push_lr_states:
1962 {
1963   \seq_push:No \l__regex_left_state_seq
1964   { \int_use:N \l__regex_left_state_int }
1965   \seq_push:No \l__regex_right_state_seq
1966   { \int_use:N \l__regex_right_state_int }
1967 }
1968 \cs_new_protected:Npn \_regex_pop_lr_states:
1969 {
1970   \seq_pop:NN \l__regex_left_state_seq \l__regex_internal_a_tl
1971   \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
1972   \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
1973   \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
1974 }
```

(`_regex_push_lr_states:` 和 `_regex_pop_lr_states:` 定义结束。)

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

`_regex_build_transition_right:nNn`

```
1975 \cs_new_protected:Npn \_regex_build_transition_left:NNN #1#2#3
1976 { \_regex_toks_put_left:Ne #2 { #1 { \int_eval:n { #3 - #2 } } } }
1977 \cs_new_protected:Npn \_regex_build_transition_right:nNn #1#2#3
1978 { \_regex_toks_put_right:Ne #2 { #1 { \int_eval:n { #3 - #2 } } } }
```

(`_regex_build_transition_left:NNN` 和 `_regex_build_transition_right:nNn` 定义结束。)

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

```
1979 \cs_new_protected:Npn \_regex_build_new_state:
1980 {
1981   \_regex_toks_clear:N \l__regex_max_state_int
```

```

1982     \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
1983     \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
1984     \int_incr:N \l__regex_max_state_int
1985 }

```

(`__regex_build_new_state`: 定义结束。)

`__regex_build_transitions_lazy:NNNN`

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

```

1986 \cs_new_protected:Npn \__regex_build_transitions_lazy:NNNN #1#2#3#4#5
1987 {
1988     \__regex_build_new_state:
1989     \__regex_toks_put_right:Ne \l__regex_left_state_int
1990     {
1991         \if_meaning:w \c_true_bool #1
1992         #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
1993         #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
1994         \else:
1995         #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
1996         #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
1997         \fi:
1998     }
1999 }

```

(`__regex_build_transitions_lazy:NNNN` 定义结束。)

9.4.4 Building classes

`__regex_class:NnnnN`
`__regex_tests_action_cost:n`

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

```

2000 \cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
2001 {
2002     \cs_set:Npe \__regex_tests_action_cost:n ##1
2003     {
2004         \exp_not:n { \exp_not:n {#2} }
2005         \bool_if:NTF #1
2006         { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
2007         { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }

```

```

2008     }
2009     \if_case:w - #4 \exp_stop_f:
2010         \__regex_class_repeat:n {#3}
2011     \or: \__regex_class_repeat:nN {#3} #5
2012     \else: \__regex_class_repeat:nnN {#3} {#4} #5
2013     \fi:
2014 }
2015 \cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }

```

(`__regex_class:NnnN` 和 `__regex_tests_action_cost:n` 定义结束。)

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

```

2016 \cs_new_protected:Npn \__regex_class_repeat:n #1
2017 {
2018     \prg_replicate:nn {#1}
2019     {
2020         \__regex_build_new_state:
2021         \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
2022         \l__regex_left_state_int \l__regex_right_state_int
2023     }
2024 }

```

(`__regex_class_repeat:n` 定义结束。)

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

```

2025 \cs_new_protected:Npn \__regex_class_repeat:nN #1#2
2026 {
2027     \if_int_compare:w #1 = \c_zero_int
2028         \__regex_build_transitions_lazyness:NNNN #2
2029         \__regex_action_free:n \l__regex_right_state_int
2030         \__regex_tests_action_cost:n \l__regex_left_state_int
2031     \else:
2032         \__regex_class_repeat:n {#1}
2033         \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int

```

```

2034     \__regex_build_transitions_lazyness:NNNN #2
2035     \__regex_action_free:n \l__regex_right_state_int
2036     \__regex_action_free:n \l__regex_internal_a_int
2037     \fi:
2038 }

```

(__regex_class_repeat:nN 定义结束。)

__regex_class_repeat:nnN

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

```

2039 \cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
2040 {
2041     \__regex_class_repeat:n {#1}
2042     \int_set:Nn \l__regex_internal_a_int
2043         { \l__regex_max_state_int + #2 - 1 }
2044     \prg_replicate:nn { #2 }
2045     {
2046         \__regex_build_transitions_lazyness:NNNN #3
2047         \__regex_action_free:n         \l__regex_internal_a_int
2048         \__regex_tests_action_cost:n \l__regex_right_state_int
2049     }
2050 }

```

(__regex_class_repeat:nnN 定义结束。)

9.4.5 Building groups

__regex_group_aux:nnnnN

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

```

2051 \cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
2052 {
2053     \if_int_compare:w #3 = \c_zero_int
2054         \__regex_build_new_state:
2055         \__regex_build_transition_right:nNn \__regex_action_free_group:n
2056         \l__regex_left_state_int \l__regex_right_state_int
2057     \fi:
2058     \__regex_build_new_state:
2059     \__regex_push_lr_states:
2060     #2
2061     \__regex_pop_lr_states:
2062     \if_case:w - #4 \exp_stop_f:
2063         \__regex_group_repeat:nn {#1} {#3}
2064     \or: \__regex_group_repeat:nnN {#1} {#3} #5
2065     \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
2066     \fi:
2067 }

```

(`__regex_group_aux:nnnnN` 定义结束。)

`__regex_group:nnnN`

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

`__regex_group_no_capture:nnnN`

```

2068 \cs_new_protected:Npn \__regex_group:nnnN #1
2069 {
2070     \exp_args:No \__regex_group_aux:nnnnN
2071     { \int_use:N \l__regex_capturing_group_int }
2072     {
2073         \int_incr:N \l__regex_capturing_group_int
2074         #1
2075     }
2076 }
2077 \cs_new_protected:Npn \__regex_group_no_capture:nnnN
2078 { \__regex_group_aux:nnnnN { -1 } }

```

(`__regex_group:nnnN` 和 `__regex_group_no_capture:nnnN` 定义结束。)

`__regex_group_resetting:nnnN`

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

`__regex_group_resetting_loop:nnNn`

```

2079 \cs_new_protected:Npn \__regex_group_resetting:nnnN #1
2080 {

```



```

2081     \__regex_group_aux:nnnnN { -1 }
2082     {
2083         \exp_args:Noo \__regex_group_resetting_loop:nnNn
2084         { \int_use:N \l__regex_capturing_group_int }
2085         { \int_use:N \l__regex_capturing_group_int }
2086         #1
2087         { ?? \prg_break:n } { }
2088         \prg_break_point:
2089     }
2090 }
2091 \cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
2092 {
2093     \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
2094     \int_set:Nn \l__regex_capturing_group_int {#2}
2095     #3 {#4}
2096     \exp_args:Nf \__regex_group_resetting_loop:nnNn
2097     { \int_max:nn {#1} { \l__regex_capturing_group_int } }
2098     {#2}
2099 }

```

(*__regex_group_resetting:nnnN* 和 *__regex_group_resetting_loop:nnNn* 定义结束。)

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

```

2100 \cs_new_protected:Npn \__regex_branch:n #1
2101 {
2102     \__regex_build_new_state:
2103     \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
2104     \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
2105     \__regex_build_transition_right:nNn \__regex_action_free:n
2106     \l__regex_left_state_int \l__regex_right_state_int
2107     #1
2108     \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
2109     \__regex_build_transition_right:nNn \__regex_action_free:n
2110     \l__regex_right_state_int \l__regex_internal_a_tl
2111 }

```

(*__regex_branch:n* 定义结束。)

__regex_group_repeat:nn This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label).

Otherwise, the auxiliary `__regex_group_repeat_aux:n` copies `#2` times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

2112 \cs_new_protected:Npn \__regex_group_repeat:nn #1#2
2113 {
2114   \if_int_compare:w #2 = \c_zero_int
2115     \int_set:Nn \l__regex_max_state_int
2116       { \l__regex_left_state_int - 1 }
2117     \__regex_build_new_state:
2118   \else:
2119     \__regex_group_repeat_aux:n {#2}
2120     \__regex_group_submatches:nNN {#1}
2121     \l__regex_internal_a_int \l__regex_right_state_int
2122     \__regex_build_new_state:
2123   \fi:
2124 }

```

(`__regex_group_repeat:nn` 定义结束。)

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

```

2125 \cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
2126 {
2127   \if_int_compare:w #1 > - \c_one_int
2128     \__regex_toks_put_left:Ne #2 { \__regex_action_submatch:nN {#1} < }
2129     \__regex_toks_put_left:Ne #3 { \__regex_action_submatch:nN {#1} > }
2130   \fi:
2131 }

```

(`__regex_group_submatches:nNN` 定义结束。)

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

```

2132 \cs_new_protected:Npn \__regex_group_repeat_aux:n #1
2133 {
2134   \__regex_build_transition_right:nNn \__regex_action_free:n
2135     \l__regex_right_state_int \l__regex_max_state_int
2136   \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int

```

```

2137     \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
2138     \if_int_compare:w \int_eval:n {#1} > \c_one_int
2139         \int_set:Nn \l__regex_internal_c_int
2140         {
2141             ( #1 - 1 )
2142             * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
2143         }
2144     \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
2145     \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
2146     \__regex_toks_memcpy:NNn
2147         \l__regex_internal_b_int
2148         \l__regex_internal_a_int
2149         \l__regex_internal_c_int
2150     \fi:
2151 }

```

(`__regex_group_repeat_aux:n` 定义结束。)

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

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

```

2152 \cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
2153 {
2154     \if_int_compare:w #2 = \c_zero_int
2155         \__regex_group_submatches:nnN {#1}
2156         \l__regex_left_state_int \l__regex_right_state_int
2157         \int_set:Nn \l__regex_internal_a_int
2158         { \l__regex_left_state_int - 1 }
2159         \__regex_build_transition_right:nNn \__regex_action_free:n
2160         \l__regex_right_state_int \l__regex_internal_a_int
2161         \__regex_build_new_state:

```

```

2162     \if_meaning:w \c_true_bool #3
2163     \__regex_build_transition_left:NNN \__regex_action_free:n
2164     \l__regex_internal_a_int \l__regex_right_state_int
2165     \else:
2166     \__regex_build_transition_right:nNn \__regex_action_free:n
2167     \l__regex_internal_a_int \l__regex_right_state_int
2168     \fi:
2169     \else:
2170     \__regex_group_repeat_aux:n {#2}
2171     \__regex_group_submatches:nNN {#1}
2172     \l__regex_internal_a_int \l__regex_right_state_int
2173     \if_meaning:w \c_true_bool #3
2174     \__regex_build_transition_right:nNn \__regex_action_free_group:n
2175     \l__regex_right_state_int \l__regex_internal_a_int
2176     \else:
2177     \__regex_build_transition_left:NNN \__regex_action_free_group:n
2178     \l__regex_right_state_int \l__regex_internal_a_int
2179     \fi:
2180     \__regex_build_new_state:
2181     \fi:
2182 }

```

(`__regex_group_repeat:nN` 定义结束。)

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

```

2183 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
2184 {
2185     \__regex_group_submatches:nNN {#1}
2186     \l__regex_left_state_int \l__regex_right_state_int
2187     \__regex_group_repeat_aux:n { #2 + #3 }

```

```

2188     \if_meaning:w \c_true_bool #4
2189     \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
2190     \prg_replicate:nn { #3 }
2191     {
2192         \int_sub:Nn \l__regex_left_state_int
2193         { \l__regex_internal_b_int - \l__regex_internal_a_int }
2194         \__regex_build_transition_left:NNN \__regex_action_free:n
2195         \l__regex_left_state_int \l__regex_max_state_int
2196     }
2197 \else:
2198     \prg_replicate:nn { #3 - 1 }
2199     {
2200         \int_sub:Nn \l__regex_right_state_int
2201         { \l__regex_internal_b_int - \l__regex_internal_a_int }
2202         \__regex_build_transition_right:nNn \__regex_action_free:n
2203         \l__regex_right_state_int \l__regex_max_state_int
2204     }
2205     \if_int_compare:w #2 = \c_zero_int
2206     \int_set:Nn \l__regex_right_state_int
2207     { \l__regex_left_state_int - 1 }
2208     \else:
2209         \int_sub:Nn \l__regex_right_state_int
2210         { \l__regex_internal_b_int - \l__regex_internal_a_int }
2211     \fi:
2212     \__regex_build_transition_right:nNn \__regex_action_free:n
2213     \l__regex_right_state_int \l__regex_max_state_int
2214 \fi:
2215 \__regex_build_new_state:
2216 }

```

(`__regex_group_repeat:nnnN` 定义结束。)

9.4.6 Others

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

```

2217 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
2218 {

```

```

2219     \__regex_build_new_state:
2220     \__regex_toks_put_right:Ne \l__regex_left_state_int
2221     {
2222         \exp_not:n {#2}
2223         \__regex_break_point:TF
2224         \bool_if:NF #1 { { } }
2225         {
2226             \__regex_action_free:n
2227             {
2228                 \int_eval:n
2229                 { \l__regex_right_state_int - \l__regex_left_state_int }
2230             }
2231         }
2232         \bool_if:NT #1 { { } }
2233     }
2234 }
2235 \cs_new_protected:Npn \__regex_b_test:
2236 {
2237     \group_begin:
2238     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
2239     \__regex_prop_w:
2240     \__regex_break_point:TF
2241     { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
2242     { \group_end: \__regex_prop_w: }
2243 }
2244 \cs_new_protected:Npn \__regex_Z_test:
2245 {
2246     \if_int_compare:w -2 = \l__regex_curr_char_int
2247     \exp_after:wN \__regex_break_true:w
2248     \fi:
2249 }
2250 \cs_new_protected:Npn \__regex_A_test:
2251 {
2252     \if_int_compare:w -2 = \l__regex_last_char_int
2253     \exp_after:wN \__regex_break_true:w
2254     \fi:
2255 }
2256 \cs_new_protected:Npn \__regex_G_test:
2257 {
2258     \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
2259     \exp_after:wN \__regex_break_true:w
2260     \fi:

```

```
2261 }
```

(`_regex_assertion:Nn` 以及其它的定义结束。)

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

```
2262 \cs_new_protected:Npn \_regex_command_K:
2263 {
2264   \_regex_build_new_state:
2265   \_regex_toks_put_right:Ne \l__regex_left_state_int
2266   {
2267     \_regex_action_submatch:nN { 0 } <
2268     \bool_set_true:N \l__regex_fresh_thread_bool
2269     \_regex_action_free:n
2270     {
2271       \int_eval:n
2272       { \l__regex_right_state_int - \l__regex_left_state_int }
2273     }
2274     \bool_set_false:N \l__regex_fresh_thread_bool
2275   }
2276 }
```

(`_regex_command_K:` 定义结束。)

9.5 Matching

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

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

that higher priority actions always come before lower priority actions, which makes things work.

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

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

9.5.1 Variables used when matching

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

```

2277 \int_new:N \l__regex_min_pos_int
2278 \int_new:N \l__regex_max_pos_int
2279 \int_new:N \l__regex_curr_pos_int
2280 \int_new:N \l__regex_start_pos_int
2281 \int_new:N \l__regex_success_pos_int

```

(`\l__regex_min_pos_int` 以及其它的定义结束。)

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


```

2282 \int_new:N \l__regex_curr_char_int
2283 \int_new:N \l__regex_curr_catcode_int
2284 \tl_new:N \l__regex_curr_token_tl
2285 \int_new:N \l__regex_last_char_int
2286 \int_new:N \l__regex_last_char_success_int
2287 \int_new:N \l__regex_case_changed_char_int

```

(*\l__regex_curr_char_int* 以及其它的定义结束。)

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

```

2288 \int_new:N \l__regex_curr_state_int

```

(*\l__regex_curr_state_int* 定义结束。)

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

\l__regex_success_submatches_tl

```

2289 \tl_new:N \l__regex_curr_submatches_tl
2290 \tl_new:N \l__regex_success_submatches_tl

```

(*\l__regex_curr_submatches_tl* 和 *\l__regex_success_submatches_tl* 定义结束。)

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

```

2291 \int_new:N \l__regex_step_int

```

(*\l__regex_step_int* 定义结束。)

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

```
2292 \int_new:N \l__regex_min_thread_int
```

```
2293 \int_new:N \l__regex_max_thread_int
```

(`\l_regex_min_thread_int` 和 `\l_regex_max_thread_int` 定义结束。)

`\g_regex_state_active_intarray` stores the last $\langle step \rangle$ in which each $\langle state \rangle$ was active. `\g_regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
2294 \intarray_new:Nn \g_regex_state_active_intarray { 65536 }
```

```
2295 \intarray_new:Nn \g_regex_thread_info_intarray { 65536 }
```

(\q_regex_state_active_intarray 和 \q_regex_thread_info_intarray 定义结束。)

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

```
2296 \tl_new:N \l_regex_matched_analysis_tl
```

```
2297 \tl_new:N \l__regex_curr_analysis_tl
```

(`\l regex matched analysis tl` 和 `\l regex curr analysis tl` 定义结束。)

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

```
2298 \tl_new:N \l__regex_every_match_tl
```

(\l_regex_every_match_tl 定义结束。)

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

```
2299 \bool_new:N \l__regex_fresh_thread_bool
2300 \bool_new:N \l__regex_empty_success_bool
2301 \cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(`\l__regex_fresh_thread_bool`, `\l__regex_empty_success_bool`, 和 `__regex_if_two_empty_matches:F` 定义结束。)

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

```
2302 \bool_new:N \g__regex_success_bool
2303 \bool_new:N \l__regex_saved_success_bool
2304 \bool_new:N \l__regex_match_success_bool
```

(`\g__regex_success_bool`, `\l__regex_saved_success_bool`, 和 `\l__regex_match_success_bool` 定义结束。)

9.5.2 Matching: framework

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

```
2305 \cs_new_protected:Npn \__regex_match:n #1
2306 {
2307   \__regex_match_init:
```

```

2308     \__regex_match_once_init:
2309     \tl_analysis_map_inline:nn {#1}
2310         { \__regex_match_one_token:nnN {##1} {##2} ##3 }
2311     \__regex_match_one_token:nnN { } { -2 } F
2312     \prg_break_point:Nn \__regex_maplike_break: { }
2313 }
2314 \cs_new_protected:Npn \__regex_match_cs:n #1
2315 {
2316     \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
2317     \__regex_match_init:
2318     \__regex_match_once_init:
2319     \str_map_inline:nn {#1}
2320     {
2321         \tl_if_blank:nTF {##1}
2322             { \__regex_match_one_token:nnN {##1} {`##1} A }
2323             { \__regex_match_one_token:nnN {##1} {`##1} C }
2324     }
2325     \__regex_match_one_token:nnN { } { -2 } F
2326     \prg_break_point:Nn \__regex_maplike_break: { }
2327 }
2328 \cs_new_protected:Npn \__regex_match_init:
2329 {
2330     \bool_gset_false:N \g__regex_success_bool
2331     \int_step_inline:nnn
2332         \l__regex_min_state_int { \l__regex_max_state_int - 1 }
2333     {
2334         \__kernel_intarray_gset:Nnn
2335             \g__regex_state_active_intarray {##1} { 1 }
2336     }
2337     \int_zero:N \l__regex_step_int
2338     \int_set:Nn \l__regex_min_pos_int { 2 }
2339     \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
2340     \int_set:Nn \l__regex_last_char_success_int { -2 }
2341     \tl_build_begin:N \l__regex_matched_analysis_tl
2342     \tl_clear:N \l__regex_curr_analysis_tl
2343     \int_set:Nn \l__regex_min_submatch_int { 1 }
2344     \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
2345     \bool_set_false:N \l__regex_empty_success_bool
2346 }

```

(*__regex_match:n*, *__regex_match_cs:n*, 和 *__regex_match_init:* 定义结束。)

__regex_match_once_init: This function resets various variables used when finding one match. It is called before

the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

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

```

2347 \cs_new_protected:Npn \__regex_match_once_init:
2348 {
2349   \if_meaning:w \c_true_bool \l__regex_empty_success_bool
2350     \cs_set:Npn \__regex_if_two_empty_matches:F
2351     {
2352       \int_compare:nNf
2353         \l__regex_start_pos_int = \l__regex_curr_pos_int
2354     }
2355   \else:
2356     \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
2357   \fi:
2358   \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
2359   \bool_set_false:N \l__regex_match_success_bool
2360   \tl_set:Nx \l__regex_curr_submatches_tl
2361     { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
2362   \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
2363   \__regex_store_state:n { \l__regex_min_state_int }
2364   \int_set:Nn \l__regex_curr_pos_int
2365     { \l__regex_start_pos_int - 1 }
2366   \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
2367   \tl_build_get_intermediate:NN \l__regex_matched_analysis_tl \l__regex_internal_a_tl
2368   \exp_args:NNf \__regex_match_once_init_aux:
2369   \tl_map_inline:nn
2370     { \exp_after:wN \l__regex_internal_a_tl \l__regex_curr_analysis_tl }
2371     { \__regex_match_one_token:nnN ##1 }
2372   \prg_break_point:Nn \__regex_maplike_break: { }
2373 }
2374 \cs_new_protected:Npn \__regex_match_once_init_aux:
2375 {
2376   \tl_build_begin:N \l__regex_matched_analysis_tl
2377   \tl_clear:N \l__regex_curr_analysis_tl
2378 }

```

(`_regex_match_once_init`: 定义结束。)

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

`_regex_multi_match:n`

```
2379 \cs_new_protected:Npn \_regex_single_match:
2380 {
2381   \tl_set:Nn \l__regex_every_match_tl
2382   {
2383     \bool_gset_eq:NN
2384     \g__regex_success_bool
2385     \l__regex_match_success_bool
2386     \_regex_maplike_break:
2387   }
2388 }
2389 \cs_new_protected:Npn \_regex_multi_match:n #1
2390 {
2391   \tl_set:Nn \l__regex_every_match_tl
2392   {
2393     \if_meaning:w \c_false_bool \l__regex_match_success_bool
2394     \exp_after:wN \_regex_maplike_break:
2395     \fi:
2396     \bool_gset_true:N \g__regex_success_bool
2397     #1
2398     \_regex_match_once_init:
2399   }
2400 }
```

(`_regex_single_match`: 和 `_regex_multi_match:n` 定义结束。)

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

`_regex_match_one_active:n`

```
2401 \cs_new_protected:Npn \_regex_match_one_token:nnN #1#2#3
2402 {
2403   \int_add:Nn \l__regex_step_int { 2 }
```

```

2404 \int_incr:N \l__regex_curr_pos_int
2405 \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
2406 \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
2407 \tl_set:Nn \l__regex_curr_token_tl {#1}
2408 \int_set:Nn \l__regex_curr_char_int {#2}
2409 \int_set:Nn \l__regex_curr_catcode_int { "#3 }
2410 \tl_build_put_right:Ne \l__regex_matched_analysis_tl
2411 { \exp_not:o \l__regex_curr_analysis_tl }
2412 \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
2413 \use:e
2414 {
2415 \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
2416 \int_step_function:nnN
2417 { \l__regex_min_thread_int }
2418 { \l__regex_max_thread_int - 1 }
2419 \__regex_match_one_active:n
2420 }
2421 \prg_break_point:
2422 \bool_set_false:N \l__regex_fresh_thread_bool
2423 \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
2424 \if_int_compare:w -2 < \l__regex_curr_char_int
2425 \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
2426 \fi:
2427 \fi:
2428 \l__regex_every_match_tl
2429 }
2430 \cs_new:Npn \__regex_match_one_active:n #1
2431 {
2432 \__regex_use_state_and_submatches:w
2433 \__kernel_intarray_range_to_clist:Nnn
2434 \g__regex_thread_info_intarray
2435 { 1 + #1 * (\l__regex_capturing_group_int * 2 + 1) }
2436 { (1 + #1) * (\l__regex_capturing_group_int * 2 + 1) }
2437 ;
2438 }

```

(`__regex_match_one_token:nnN` 和 `__regex_match_one_active:n` 定义结束。)

9.5.3 Using states of the nfa

`__regex_use_state:` Use the current NFA instruction. The state is initially marked as belonging to the current **step**: this allows normal free transition to repeat, but group-repeating tran-

sitions won't. Once we are done exploring all the branches it spawned, the state is marked as `step + 1`: any thread hitting it at that point will be terminated.

```

2439 \cs_new_protected:Npn \__regex_use_state:
2440 {
2441   \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
2442     { \l__regex_curr_state_int } { \l__regex_step_int }
2443   \__regex_toks_use:w \l__regex_curr_state_int
2444   \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
2445     { \l__regex_curr_state_int }
2446     { \int_eval:n { \l__regex_step_int + 1 } }
2447 }

```

(`__regex_use_state`: 定义结束。)

`__regex_use_state_and_submatches:w`

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

```

2448 \cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
2449 {
2450   \int_set:Nn \l__regex_curr_state_int {#1}
2451   \if_int_compare:w
2452     \__kernel_intarray_item:Nn \g__regex_state_active_intarray
2453       { \l__regex_curr_state_int }
2454       < \l__regex_step_int
2455     \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
2456     \exp_after:wN \__regex_use_state:
2457   \fi:
2458   \scan_stop:
2459 }

```

(`__regex_use_state_and_submatches:w` 定义结束。)

9.5.4 Actions when matching

`__regex_action_start_wildcard:N`

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

```

2460 \cs_new_protected:Npn \__regex_action_start_wildcard:N #1
2461 {
2462   \bool_set_true:N \l__regex_fresh_thread_bool

```



```

2463     \_regex_action_free:n {1}
2464     \bool_set_false:N \l__regex_fresh_thread_bool
2465     \bool_if:NT #1 { \_regex_action_cost:n {0} }
2466 }

```

(_regex_action_start_wildcard:N 定义结束。)

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

```

2467 \cs_new_protected:Npn \_regex_action_free:n
2468 { \_regex_action_free_aux:nn { > \l__regex_step_int \else: } }
2469 \cs_new_protected:Npn \_regex_action_free_group:n
2470 { \_regex_action_free_aux:nn { < \l__regex_step_int } }
2471 \cs_new_protected:Npn \_regex_action_free_aux:nn #1#2
2472 {
2473   \use:e
2474   {
2475     \int_add:Nn \l__regex_curr_state_int {#2}
2476     \exp_not:n
2477     {
2478       \if_int_compare:w
2479         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
2480         { \l__regex_curr_state_int }
2481         #1
2482         \exp_after:wN \_regex_use_state:
2483         \fi:
2484     }
2485     \int_set:Nn \l__regex_curr_state_int
2486     { \int_use:N \l__regex_curr_state_int }
2487     \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
2488     { \exp_not:o \l__regex_curr_submatches_tl }
2489   }
2490 }

```

(_regex_action_free:n, _regex_action_free_group:n, 和 _regex_action_free_aux:nn 定义结束。)

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

```
2491 \cs_new_protected:Npn \__regex_action_cost:n #1
2492 {
2493   \exp_args:Ne \__regex_store_state:n
2494     { \int_eval:n { \l__regex_curr_state_int + #1 } }
2495 }
```

(`__regex_action_cost:n` 定义结束。)

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

```
\__regex_store_submatches:
2496 \cs_new_protected:Npn \__regex_store_state:n #1
2497 {
2498   \exp_args:No \__regex_store_submatches:nn
2499     \l__regex_curr_submatches_tl {#1}
2500   \int_incr:N \l__regex_max_thread_int
2501 }
2502 \cs_new_protected:Npn \__regex_store_submatches:nn #1#2
2503 {
2504   \__kernel_intarray_gset_range_from_clist:Nnn
2505     \g__regex_thread_info_intarray
2506     {
2507       \__regex_int_eval:w
2508         1 + \l__regex_max_thread_int *
2509         (\l__regex_capturing_group_int * 2 + 1)
2510     }
2511     { #2 , #1 }
2512 }
```

(`__regex_store_state:n` 和 `__regex_store_submatches:` 定义结束。)

`__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.

```
2513 \cs_new_protected:Npn \__regex_disable_submatches:
2514 {
2515   \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
2516   \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
2517 }
```

(*_regex_disable_submatches*: 定义结束。)

_regex_action_submatch:nN Update the current submatches with the information from the current position.

_regex_action_submatch_aux:w Maybe a bottleneck.

```
2518 \cs_new_protected:Npn \_regex_action_submatch:nN #1#2
2519 {
2520   \exp_after:wN \_regex_action_submatch_aux:w
2521   \l__regex_curr_submatches_tl ; {#1} #2
2522 }
2523 \cs_new_protected:Npn \_regex_action_submatch_aux:w #1 ; #2#3
2524 {
2525   \tl_set:Nx \l__regex_curr_submatches_tl
2526   {
2527     \prg_replicate:nn
2528     { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }
2529     { \_regex_action_submatch_auxii:w }
2530     \_regex_action_submatch_auxiii:w
2531     #1
2532   }
2533 }
2534 \cs_new:Npn \_regex_action_submatch_auxii:w
2535   #1 \_regex_action_submatch_auxiii:w #2 ,
2536   { #2 , #1 \_regex_action_submatch_auxiii:w }
2537 \cs_new:Npn \_regex_action_submatch_auxiii:w #1 ,
2538   { \int_use:N \l__regex_curr_pos_int , }
```

(*_regex_action_submatch:nN* 以及其它的定义结束。)

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

```
2539 \cs_new_protected:Npn \_regex_action_success:
2540 {
2541   \_regex_if_two_empty_matches:F
2542   {
2543     \bool_set_true:N \l__regex_match_success_bool
2544     \bool_set_eq:NN \l__regex_empty_success_bool
2545     \l__regex_fresh_thread_bool
2546     \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
```

```

2547         \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
2548         \tl_build_begin:N \l__regex_matched_analysis_tl
2549         \tl_set_eq:NN \l__regex_success_submatches_tl
2550             \l__regex_curr_submatches_tl
2551         \prg_break:
2552     }
2553 }

```

(`__regex_action_success`: 定义结束。)

9.6 Replacement

9.6.1 Variables and helpers used in replacement

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

```

2554 \int_new:N \l__regex_replacement_csnames_int

```

(`\l__regex_replacement_csnames_int` 定义结束。)

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

`\l__regex_replacement_category_seq`

```

2555 \tl_new:N \l__regex_replacement_category_tl
2556 \seq_new:N \l__regex_replacement_category_seq

```

(`\l__regex_replacement_category_tl` 和 `\l__regex_replacement_category_seq` 定义结束。)

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

```

2557 \tl_new:N \g__regex_balance_tl

```

(`\g__regex_balance_tl` 定义结束。)

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

important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
2558 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
2559 { - \__regex_submatch_balance:n {#1} }
```

(*__regex_replacement_balance_one_match:n* 定义结束。)

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

```
2560 \cs_new:Npn \__regex_replacement_do_one_match:n #1
2561 {
2562   \__regex_query_range:nn
2563     { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
2564     { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
2565 }
```

(*__regex_replacement_do_one_match:n* 定义结束。)

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

```
2566 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(*__regex_replacement_exp_not:N* 定义结束。)

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

```
2567 \cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(*__regex_replacement_exp_not:V* 定义结束。)

9.6.2 Query and brace balance

`_regex_query_range:nn`

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

```
2568 \cs_new:Npn \_regex_query_range:nn #1#2
2569 {
2570   \exp_after:wN \_regex_query_range_loop:ww
2571   \int_value:w \_regex_int_eval:w #1 \exp_after:wN ;
2572   \int_value:w \_regex_int_eval:w #2 ;
2573   \prg_break_point:
2574 }
2575 \cs_new:Npn \_regex_query_range_loop:ww #1 ; #2 ;
2576 {
2577   \if_int_compare:w #1 < #2 \exp_stop_f:
2578   \else:
2579     \exp_after:wN \prg_break:
2580   \fi:
2581   \_regex_toks_use:w #1 \exp_stop_f:
2582   \exp_after:wN \_regex_query_range_loop:ww
2583   \int_value:w \_regex_int_eval:w #1 + 1 ; #2 ;
2584 }
```

(`_regex_query_range:nn` 和 `_regex_query_range_loop:ww` 定义结束。)

`_regex_query_submatch:n`

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

```
2585 \cs_new:Npn \_regex_query_submatch:n #1
2586 {
2587   \_regex_query_range:nn
2588   { \_kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
2589   { \_kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
2590 }
```

(`_regex_query_submatch:n` 定义结束。)

`_regex_submatch_balance:n`

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the

brace balances at the $\langle max\ pos \rangle$ and $\langle min\ pos \rangle$. These two positions are found in the corresponding “submatch” arrays.

```

2591 \cs_new_protected:Npn \__regex_submatch_balance:n #1
2592 {
2593   \int_eval:n
2594   {
2595     \__regex_intarray_item:NnF \g__regex_balance_intarray
2596     {
2597       \__kernel_intarray_item:Nn
2598       \g__regex_submatch_end_intarray {#1}
2599     }
2600     { 0 }
2601   -
2602     \__regex_intarray_item:NnF \g__regex_balance_intarray
2603     {
2604       \__kernel_intarray_item:Nn
2605       \g__regex_submatch_begin_intarray {#1}
2606     }
2607     { 0 }
2608   }
2609 }

```

(`__regex_submatch_balance:n` 定义结束。)

9.6.3 Framework

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

```

2610 \cs_new_protected:Npn \__regex_replacement:n
2611 { \__regex_replacement_apply:Nn \__regex_replacement_set:n }
2612 \cs_new_protected:Npn \__regex_replacement_apply:Nn #1#2
2613 {
2614   \group_begin:
2615     \tl_build_begin:N \l__regex_build_tl
2616     \int_zero:N \l__regex_balance_int
2617     \tl_gclear:N \g__regex_balance_tl

```

```

2618 \__regex_escape_use:nnnn
2619 {
2620     \if_charcode:w \c_right_brace_str ##1
2621     \__regex_replacement_rbrace:N
2622 \else:
2623     \if_charcode:w \c_left_brace_str ##1
2624     \__regex_replacement_lbrace:N
2625 \else:
2626     \__regex_replacement_normal:n
2627 \fi:
2628 \fi:
2629 ##1
2630 }
2631 { \__regex_replacement_escaped:N ##1 }
2632 { \__regex_replacement_normal:n ##1 }
2633 {#2}
2634 \prg_do_nothing: \prg_do_nothing:
2635 \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
2636 \msg_error:nne { regex } { replacement-missing-rbrace }
2637 { \int_use:N \l__regex_replacement_csnames_int }
2638 \tl_build_put_right:Ne \l__regex_build_tl
2639 { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
2640 \fi:
2641 \seq_if_empty:NF \l__regex_replacement_category_seq
2642 {
2643     \msg_error:nne { regex } { replacement-missing-rparen }
2644     { \seq_count:N \l__regex_replacement_category_seq }
2645     \seq_clear:N \l__regex_replacement_category_seq
2646 }
2647 \tl_gput_right:Ne \g__regex_balance_tl
2648 { + \int_use:N \l__regex_balance_int }
2649 \tl_build_end:N \l__regex_build_tl
2650 \exp_args:NNo
2651 \group_end:
2652 #1 \l__regex_build_tl
2653 }
2654 \cs_generate_variant:Nn \__regex_replacement:n { e }
2655 \cs_new_protected:Npn \__regex_replacement_set:n #1
2656 {
2657     \cs_set:Npn \__regex_replacement_do_one_match:n ##1
2658     {
2659         \__regex_query_range:nn

```



```

2660         {
2661             \__kernel_intarray_item:Nn
2662             \g__regex_submatch_prev_intarray {##1}
2663         }
2664         {
2665             \__kernel_intarray_item:Nn
2666             \g__regex_submatch_begin_intarray {##1}
2667         }
2668         #1
2669     }
2670 \exp_args:Nno \use:n
2671 { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
2672 {
2673     \g__regex_balance_tl
2674     - \__regex_submatch_balance:n {##1}
2675 }
2676 }

```

(`__regex_replacement:n`, `__regex_replacement_apply:Nn`, 和 `__regex_replacement_set:n` 定义结束。)

`__regex_case_replacement:n`

`__regex_case_replacement:e`

```

2677 \tl_new:N \g__regex_case_replacement_tl
2678 \tl_new:N \g__regex_case_balance_tl
2679 \cs_new_protected:Npn \__regex_case_replacement:n #1
2680 {
2681     \tl_gset:Nn \g__regex_case_balance_tl
2682     {
2683         \if_case:w
2684             \__kernel_intarray_item:Nn
2685             \g__regex_submatch_case_intarray {##1}
2686         }
2687     \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
2688     \tl_map_tokens:nn {#1}
2689     { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
2690     \tl_gset:No \g__regex_balance_tl
2691     { \g__regex_case_balance_tl \fi: }
2692     \exp_args:No \__regex_replacement_set:n
2693     { \g__regex_case_replacement_tl \fi: }
2694 }
2695 \cs_generate_variant:Nn \__regex_case_replacement:n { e }
2696 \cs_new_protected:Npn \__regex_case_replacement_aux:n #1
2697 {
2698     \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }

```

```

2699     \tl_gput_right:No \g__regex_case_balance_tl
2700     { \exp_after:wN \or: \g__regex_balance_tl }
2701   }

```

(__regex_case_replacement:n 定义结束。)

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

```

2702 \cs_new_protected:Npn \__regex_replacement_put:n
2703   { \tl_build_put_right:Nn \l__regex_build_tl }

```

(__regex_replacement_put:n 定义结束。)

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

```

2704 \cs_new_protected:Npn \__regex_replacement_normal:n #1
2705   {
2706     \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
2707       { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
2708       {
2709         \tl_if_empty:NTF \l__regex_replacement_category_tl
2710           { \__regex_replacement_normal_aux:N #1 }
2711         { % (
2712           \token_if_eq_charcode:NNTF #1 )
2713           {
2714             \seq_pop:NN \l__regex_replacement_category_seq
2715               \l__regex_replacement_category_tl
2716           }
2717           {
2718             \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
2719             ? #1
2720           }
2721         }
2722       }
2723   }

```

```

2724 \cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
2725 {
2726   \token_if_eq_charcode:NNTF #1 \c_space_token
2727   { \__regex_replacement_c_S:w }
2728   {
2729     \exp_after:wN \exp_after:wN
2730     \if_case:w \tex_catcode:D `#1 \exp_stop_f:
2731       \__regex_replacement_c_O:w
2732       \or: \__regex_replacement_c_B:w
2733       \or: \__regex_replacement_c_E:w
2734       \or: \__regex_replacement_c_M:w
2735       \or: \__regex_replacement_c_T:w
2736       \or: \__regex_replacement_c_O:w
2737       \or: \__regex_replacement_c_P:w
2738       \or: \__regex_replacement_c_U:w
2739       \or: \__regex_replacement_c_D:w
2740       \or: \__regex_replacement_c_O:w
2741       \or: \__regex_replacement_c_S:w
2742       \or: \__regex_replacement_c_L:w
2743       \or: \__regex_replacement_c_O:w
2744       \or: \__regex_replacement_c_A:w
2745       \else: \__regex_replacement_c_O:w
2746     \fi:
2747   }
2748   ? #1
2749 }

```

(`__regex_replacement_normal:n` 和 `__regex_replacement_normal_aux:N` 定义结束。)

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

```

2750 \cs_new_protected:Npn \__regex_replacement_escaped:N #1
2751 {
2752   \cs_if_exist_use:cF { __regex_replacement_#1:w }
2753   {
2754     \if_int_compare:w 1 < 1#1 \exp_stop_f:
2755       \__regex_replacement_put_submatch:n {#1}
2756     \else:
2757       \__regex_replacement_normal:n {#1}
2758     \fi:
2759   }
2760 }

```

(`_regex_replacement_escaped:N` 定义结束。)

9.6.4 Submatches

`_regex_replacement_put_submatch:n` Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

`_regex_replacement_put_submatch_aux:n`

```
2761 \cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
2762 {
2763   \if_int_compare:w #1 < \l__regex_capturing_group_int
2764     \__regex_replacement_put_submatch_aux:n {#1}
2765   \else:
2766     \msg_expandable_error:nnff { regex } { submatch-too-big }
2767     {#1} { \int_eval:n { \l__regex_capturing_group_int - 1 } }
2768   \fi:
2769 }
2770 \cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
2771 {
2772   \tl_build_put_right:Nn \l__regex_build_tl
2773   { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
2774   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
2775     \tl_gput_right:Nn \g__regex_balance_tl
2776     { + \__regex_submatch_balance:n { \int_eval:n { #1 + ##1 } } }
2777   \fi:
2778 }
```

(`_regex_replacement_put_submatch:n` 和 `_regex_replacement_put_submatch_aux:n` 定义结束。)

`__regex_replacement_g:w` Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

`__regex_replacement_g_digits:NN`

```
2779 \cs_new_protected:Npn \__regex_replacement_g:w #1#2
2780 {
2781   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
2782   { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
2783   { \__regex_replacement_error:NNN g #1 #2 }
2784 }
2785 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
2786 {
2787   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
2788   {
```

```

2789     \if_int_compare:w 1 < 1#2 \exp_stop_f:
2790     #2
2791     \exp_after:wN \use_i:nnn
2792     \exp_after:wN \_regex_replacement_g_digits:NN
2793 \else:
2794     \exp_stop_f:
2795     \exp_after:wN \_regex_replacement_error:NNN
2796     \exp_after:wN g
2797 \fi:
2798 }
2799 {
2800     \exp_stop_f:
2801     \if_meaning:w \_regex_replacement_rbrace:N #1
2802     \exp_args:No \_regex_replacement_put_submatch:n
2803     { \int_use:N \l__regex_internal_a_int }
2804     \exp_after:wN \use_none:nn
2805 \else:
2806     \exp_after:wN \_regex_replacement_error:NNN
2807     \exp_after:wN g
2808 \fi:
2809 }
2810 #1 #2
2811 }

```

(`_regex_replacement_g:w` 和 `_regex_replacement_g_digits:NN` 定义结束。)

9.6.5 Csames in replacement

`_regex_replacement_c:w` `\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```

2812 \cs_new_protected:Npn \_regex_replacement_c:w #1#2
2813 {
2814     \token_if_eq_meaning:NNTF #1 \_regex_replacement_normal:n
2815     {
2816         \cs_if_exist:cTF { \_regex_replacement_c_#2:w }
2817         { \_regex_replacement_cat:NNN #2 }
2818         { \_regex_replacement_error:NNN c #1#2 }
2819     }
2820     {
2821         \token_if_eq_meaning:NNTF #1 \_regex_replacement_lbrace:N
2822         { \_regex_replacement_cu_aux:Nw \_regex_replacement_exp_not:N }

```

```

2823         { \_regex_replacement_error:NNN c #1#2 }
2824     }
2825 }

```

(_regex_replacement_c:w 定义结束。)

_regex_replacement_cu_aux:Nw Start a control sequence with \cs:w, protected from expansion by #1 (either _regex_replacement_exp_not:N or \exp_not:V), or turned to a string by \tl_to_str:V if inside another csname construction \c or \u. We use \tl_to_str:V rather than \tl_to_str:N to deal with integers and other registers.

```

2826 \cs_new_protected:Npn \_regex_replacement_cu_aux:Nw #1
2827 {
2828     \if_case:w \l__regex_replacement_csnames_int
2829         \tl_build_put_right:Nn \l__regex_build_tl
2830         { \exp_not:n { \exp_after:wN #1 \cs:w } }
2831     \else:
2832         \tl_build_put_right:Nn \l__regex_build_tl
2833         { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
2834     \fi:
2835     \int_incr:N \l__regex_replacement_csnames_int
2836 }

```

(_regex_replacement_cu_aux:Nw 定义结束。)

_regex_replacement_u:w Check that \u is followed by a left brace. If so, start a control sequence with \cs:w, which is then unpacked either with \exp_not:V or \tl_to_str:V depending on the current context.

```

2837 \cs_new_protected:Npn \_regex_replacement_u:w #1#2
2838 {
2839     \token_if_eq_meaning:NNTF #1 \_regex_replacement_lbrace:N
2840     { \_regex_replacement_cu_aux:Nw \_regex_replacement_exp_not:V }
2841     { \_regex_replacement_error:NNN u #1#2 }
2842 }

```

(_regex_replacement_u:w 定义结束。)

_regex_replacement_rbrace:N Within a \c{...} or \u{...} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

2843 \cs_new_protected:Npn \_regex_replacement_rbrace:N #1
2844 {
2845     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
2846         \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }

```

```

2847     \int_decr:N \l__regex_replacement_csnames_int
2848   \else:
2849     \__regex_replacement_normal:n {#1}
2850   \fi:
2851 }

```

(__regex_replacement_rbrace:N 定义结束。)

__regex_replacement_lbrace:N Within a `\c{...}` or `\u{...}` construction, this is forbidden. Otherwise, this is a raw left brace.

```

2852 \cs_new_protected:Npn \__regex_replacement_lbrace:N #1
2853 {
2854   \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
2855     \msg_error:nnn { regex } { cu-lbrace } { u }
2856   \else:
2857     \__regex_replacement_normal:n {#1}
2858   \fi:
2859 }

```

(__regex_replacement_lbrace:N 定义结束。)

9.6.6 Characters in replacement

__regex_replacement_cat:NNN Here, #1 is a letter among BEMTPUDSLOA and #2#3 denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\c{...}` or `\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

2860 \cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
2861 {
2862   \token_if_eq_meaning:NNTF \prg_do_nothing: #3
2863   { \msg_error:nn { regex } { replacement-catcode-end } }
2864   {
2865     \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
2866     {
2867       \msg_error:nnnn
2868       { regex } { replacement-catcode-in-cs } {#1} {#3}
2869       #2 #3
2870     }
2871     {
2872       \__regex_two_if_eq:NNNNTF #2 #3 \__regex_replacement_normal:n (
2873       {
2874         \seq_push:NV \l__regex_replacement_category_seq

```

```

2875         \l__regex_replacement_category_tl
2876         \tl_set:Nn \l__regex_replacement_category_tl {#1}
2877     }
2878     {
2879         \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
2880         {
2881             \__regex_char_if_alphanumeric:NTF #3
2882             {
2883                 \msg_error:nnnn
2884                 { regex } { replacement-catcode-escaped }
2885                 {#1} {#3}
2886             }
2887             { }
2888         }
2889         \use:c { __regex_replacement_c_#1:w } #2 #3
2890     }
2891 }
2892 }
2893 }

```

(`__regex_replacement_cat:NNN` 定义结束。)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
2894 \group_begin:
```

`__regex_replacement_char:nNN` The only way to produce an arbitrary character-catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

2895     \cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
2896     {
2897         \tex_lccode:D 0 = `#3 \scan_stop:
2898         \tex_lowercase:D { \__regex_replacement_put:n {#1} }
2899     }

```

(`__regex_replacement_char:nNN` 定义结束。)

`_regex_replacement_c_A:w` For an active character, expansion must be avoided, twice because we later do two e-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```
2900 \char_set_catcode_active:N \^^@
2901 \cs_new_protected:Npn \_regex_replacement_c_A:w
2902 { \_regex_replacement_char:nNN { \exp_not:n { \exp_not:N \^^@ } } }
```

(`_regex_replacement_c_A:w` 定义结束。)

`_regex_replacement_c_B:w` An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually e-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with l3tl-analysis.

```
2903 \char_set_catcode_group_begin:N \^^@
2904 \cs_new_protected:Npn \_regex_replacement_c_B:w
2905 {
2906   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
2907     \int_incr:N \l__regex_balance_int
2908   \fi:
2909   \_regex_replacement_char:nNN
2910     { \exp_not:n { \exp_after:wN \^^@ \if_false: } \fi: } }
2911 }
```

(`_regex_replacement_c_B:w` 定义结束。)

`_regex_replacement_c_C:w` This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two e-expansions.

```
2912 \cs_new_protected:Npn \_regex_replacement_c_C:w #1#2
2913 {
2914   \tl_build_put_right:Nn \l__regex_build_tl
2915     { \exp_not:N \_regex_replacement_exp_not:N \exp_not:c {#2} }
2916 }
```

(`_regex_replacement_c_C:w` 定义结束。)

`_regex_replacement_c_D:w` Subscripts fit the mould: `\lowercase` the null byte with the correct category.

```
2917 \char_set_catcode_math_subscript:N \^^@
2918 \cs_new_protected:Npn \_regex_replacement_c_D:w
2919 { \_regex_replacement_char:nNN { \^^@ } }
```

(*_regex_replacement_c_D:w* 定义结束。)

_regex_replacement_c_E:w Similar to the begin-group case, the second **e**-expansion produces the bare end-group token.

```
2920 \char_set_catcode_group_end:N \^^@
2921 \cs_new_protected:Npn \_regex\_replacement\_c\_E:w
2922 {
2923   \if_int_compare:w \l__regex\_replacement\_csnames\_int = \c_zero_int
2924     \int_decr:N \l__regex\_balance\_int
2925   \fi:
2926   \_regex\_replacement\_char:nNN
2927   { \exp_not:n { \if_false: { \fi: ^^@ } }
2928 }
```

(*_regex_replacement_c_E:w* 定义结束。)

_regex_replacement_c_L:w Simply **\lowercase** a letter null byte to produce an arbitrary letter.

```
2929 \char_set_catcode_letter:N \^^@
2930 \cs_new_protected:Npn \_regex\_replacement\_c\_L:w
2931 { \_regex\_replacement\_char:nNN { ^^@ } }
```

(*_regex_replacement_c_L:w* 定义结束。)

_regex_replacement_c_M:w No surprise here, we lowercase the null math toggle.

```
2932 \char_set_catcode_math_toggle:N \^^@
2933 \cs_new_protected:Npn \_regex\_replacement\_c\_M:w
2934 { \_regex\_replacement\_char:nNN { ^^@ } }
```

(*_regex_replacement_c_M:w* 定义结束。)

_regex_replacement_c_O:w Lowercase an other null byte.

```
2935 \char_set_catcode_other:N \^^@
2936 \cs_new_protected:Npn \_regex\_replacement\_c\_O:w
2937 { \_regex\_replacement\_char:nNN { ^^@ } }
```

(*_regex_replacement_c_O:w* 定义结束。)

_regex_replacement_c_P:w For macro parameters, expansion is a tricky issue. We need to prepare for two **e**-expansions and passing through various macro definitions. Note that we cannot replace one **\exp_not:n** by doubling the macro parameter characters because this would misbehave if a mischievous user asks for **\c{\cP\#}**, since that macro parameter character would be doubled.

```
2938 \char_set_catcode_parameter:N \^^@
```

```

2939 \cs_new_protected:Npn \__regex_replacement_c_P:w
2940 {
2941   \__regex_replacement_char:nNN
2942   { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
2943 }

```

(*__regex_replacement_c_P:w* 定义结束。)

__regex_replacement_c_S:w Spaces are normalized on input by T_EX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```

2944 \cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
2945 {
2946   \if_int_compare:w `#2 = \c_zero_int
2947     \msg_error:nn { regex } { replacement-null-space }
2948   \fi:
2949   \tex_lccode:D `\< = `#2 \scan_stop:
2950   \tex_lowercase:D { \__regex_replacement_put:n {~} }
2951 }

```

(*__regex_replacement_c_S:w* 定义结束。)

__regex_replacement_c_T:w No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```

2952 \char_set_catcode_alignment:N \^^@
2953 \cs_new_protected:Npn \__regex_replacement_c_T:w
2954 { \__regex_replacement_char:nNN { ^^@ } }

```

(*__regex_replacement_c_T:w* 定义结束。)

__regex_replacement_c_U:w Simple call to *__regex_replacement_char:nNN* which lowercases the math superscript *^^@*.

```

2955 \char_set_catcode_math_superscript:N \^^@
2956 \cs_new_protected:Npn \__regex_replacement_c_U:w
2957 { \__regex_replacement_char:nNN { ^^@ } }

```

(*__regex_replacement_c_U:w* 定义结束。)

Restore the catcode of the null byte.

```

2958 \group_end:

```

9.6.7 An error

`_regex_replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
2959 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
2960 {
2961     \msg_error:nne { regex } { replacement-#1 } {#3}
2962     #2 #3
2963 }
```

(`_regex_replacement_error:NNN` 定义结束。)

9.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
2964 \cs_new_protected:Npn \regex_new:N #1
2965 { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(`\regex_new:N` 定义结束。这个函数被记录在第 12 页。)

`\l_tmpa_regex` The usual scratch space.

```
\l_tmpb_regex 2966 \regex_new:N \l_tmpa_regex
\g_tmpa_regex 2967 \regex_new:N \l_tmpb_regex
\g_tmpb_regex 2968 \regex_new:N \g_tmpa_regex
2969 \regex_new:N \g_tmpb_regex
```

(`\l_tmpa_regex` 以及其它的定义结束。这些变量被记录在第 18 页。)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment
`\regex_gset:Nn` function.

```
\regex_const:Nn 2970 \cs_new_protected:Npn \regex_set:Nn #1#2
2971 {
2972     \__regex_compile:n {#2}
2973     \tl_set_eq:NN #1 \l__regex_internal_regex
2974 }
2975 \cs_new_protected:Npn \regex_gset:Nn #1#2
2976 {
2977     \__regex_compile:n {#2}
2978     \tl_gset_eq:NN #1 \l__regex_internal_regex
2979 }
2980 \cs_new_protected:Npn \regex_const:Nn #1#2
2981 {
2982     \__regex_compile:n {#2}
```

```

2983     \tl_const:Ne #1 { \exp_not:o \l__regex_internal_regex }
2984 }

```

(`\regex_set:Nn`, `\regex_gset:Nn`, 和 `\regex_const:Nn` 定义结束。这些函数被记录在第13页。)

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

```

\regex_show:N 2985 \cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nneeee }
\regex_log:N   2986 \cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nneeee }
\__regex_show:Nn 2987 \cs_new_protected:Npn \__regex_show:Nn #1#2
                2988 {
                2989     \__regex_compile:n {#2}
                2990     \__regex_show:N \l__regex_internal_regex
                2991     #1 { regex } { show }
                2992     { \tl_to_str:n {#2} } { }
                2993     { \l__regex_internal_a_tl } { }
                2994 }
                2995 \cs_new_protected:Npn \regex_show:N { \__regex_show:Nn \msg_show:nneeee }
                2996 \cs_new_protected:Npn \regex_log:N { \__regex_show:Nn \msg_log:nneeee }
                2997 \cs_new_protected:Npn \__regex_show:Nn #1#2
                2998 {
                2999     \__kernel_chk_tl_type:NnnT #2 { regex }
                3000     { \exp_args:No \__regex_clean_regex:n {#2} }
                3001     {
                3002         \__regex_show:N #2
                3003         #1 { regex } { show }
                3004         { } { \token_to_str:N #2 }
                3005         { \l__regex_internal_a_tl } { }
                3006     }
                3007 }

```

(`\regex_show:n` 以及其它的定义结束。这些函数被记录在第13页。)

`\regex_match:nnTF` Those conditionals are based on a common auxiliary defined later. Its first argument
`\regex_match:nVTF` builds the NFA corresponding to the regex, and the second argument is the query
`\regex_match:NnTF` token list. Once we have performed the match, convert the resulting boolean to
`\regex_match:NVTF` `\prg_return_true:` or `false`.

```

3008 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
3009 {
3010     \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
3011     \__regex_return:

```

```

3012 }
3013 \prg_generate_conditional_variant:Nnn \regex_match:nn { nV } { T , F , TF }
3014 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
3015 {
3016   \__regex_if_match:nn { \__regex_build:N #1 } {#2}
3017   \__regex_return:
3018 }
3019 \prg_generate_conditional_variant:Nnn \regex_match:Nn { NV } { T , F , TF }

```

(*\regex_match:nnTF* 和 *\regex_match:NnTF* 定义结束。这些函数被记录在第13页。)

\regex_count:nnN Again, use an auxiliary whose first argument builds the NFA.

```

\regex_count:nVN 3020 \cs_new_protected:Npn \regex_count:nnN #1
\regex_count:NnN 3021 { \__regex_count:nnN { \__regex_build:n {#1} } }
\regex_count:NVN 3022 \cs_new_protected:Npn \regex_count:NnN #1
3023 { \__regex_count:nnN { \__regex_build:N #1 } }
3024 \cs_generate_variant:Nn \regex_count:nnN { nV }
3025 \cs_generate_variant:Nn \regex_count:NnN { NV }

```

(*\regex_count:nnN* 和 *\regex_count:NnN* 定义结束。这些函数被记录在第14页。)

\regex_match_case:nn The auxiliary errors if #1 has an odd number of items, and otherwise it sets *\g__-regex_case_int* according to which case was found (zero if not found). The **true** branch leaves the corresponding code in the input stream.

\regex_match_case:nnTF

```

3026 \cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
3027 {
3028   \__regex_match_case:nnTF {#1} {#2}
3029   {
3030     \tl_item:nn {#1} { 2 * \g__regex_case_int }
3031     #3
3032   }
3033 }
3034 \cs_new_protected:Npn \regex_match_case:nn #1#2
3035 { \regex_match_case:nnTF {#1} {#2} { } { } }
3036 \cs_new_protected:Npn \regex_match_case:nnT #1#2#3
3037 { \regex_match_case:nnTF {#1} {#2} {#3} { } }
3038 \cs_new_protected:Npn \regex_match_case:nnF #1#2
3039 { \regex_match_case:nnTF {#1} {#2} { } { } }

```

(*\regex_match_case:nnTF* 定义结束。这个函数被记录在第14页。)

\regex_extract_once:nnN We define here 40 user functions, following a common pattern in terms of *:nnN* auxiliaries, defined in the coming subsections. The auxiliary is handed *__regex_build:n*

\regex_extract_once:nVN

\regex_extract_once:nnN

\regex_extract_once:nVN

\regex_extract_once:NnN

\regex_extract_once:NVN

\regex_extract_once:NnN

\regex_extract_once:NVN

\regex_extract_all:nnN

or `__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.* The conditionals call `__regex_return:` to return either `true` or `false` once matching has been performed.

```

3040 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
3041 {
3042   \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
3043   \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
3044   \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
3045     { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
3046   \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
3047     { #1 { \__regex_build:N ##1 } {##2} ##3 \__regex_return: }
3048   \cs_generate_variant:Nn #2 { nV }
3049   \prg_generate_conditional_variant:Nnn #2 { nV } { T , F , TF }
3050   \cs_generate_variant:Nn #3 { NV }
3051   \prg_generate_conditional_variant:Nnn #3 { NV } { T , F , TF }
3052
3053 }
3054 \__regex_tmp:w \__regex_extract_once:nnN
3055 \regex_extract_once:nnN \regex_extract_once:NnN
3056 \__regex_tmp:w \__regex_extract_all:nnN
3057 \regex_extract_all:nnN \regex_extract_all:NnN
3058 \__regex_tmp:w \__regex_replace_once:nnN
3059 \regex_replace_once:nnN \regex_replace_once:NnN
3060 \__regex_tmp:w \__regex_replace_all:nnN
3061 \regex_replace_all:nnN \regex_replace_all:NnN
3062 \__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN

```

(`\regex_extract_once:nnNTF` 以及其它的定义结束。这些函数被记录在第15页。)

`\regex_replace_case_once:nN` If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_once:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use. The `\tl_item:nn` is only expanded once we know the value of `\g__regex_case_int`, namely which case matched.

`\regex_replace_case_once:nNTF`

```

3063 \cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
3064 {
3065   \int_if_odd:nTF { \tl_count:n {#1} }
3066     {
3067       \msg_error:nneeee { regex } { case-odd }
3068       { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
3069       { \tl_count:n {#1} } { \tl_to_str:n {#1} }
3070     } \use_ii:nn

```

```

3071     }
3072     {
3073         \__regex_replace_once_aux:nnN
3074         { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
3075         { \__regex_replacement:e { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
3076         #2
3077         \bool_if:NTF \g__regex_success_bool
3078     }
3079 }
3080 \cs_new_protected:Npn \regex_replace_case_once:nN #1#2
3081 { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
3082 \cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
3083 { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
3084 \cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
3085 { \regex_replace_case_once:nNTF {#1} {#2} { } }

```

(*\regex_replace_case_once:nNTF* 定义结束。这个函数被记录在第17页。)

\regex_replace_case_all:nN If the input is bad (odd number of items) then take the false branch. Otherwise, use
\regex_replace_case_all:nNTF the same auxiliary as **\regex_replace_all:nnN**, but with more complicated code to build the automaton, and to find what replacement text to use.

```

3086 \cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
3087 {
3088     \int_if_odd:nTF { \tl_count:n {#1} }
3089     {
3090         \msg_error:nneeee { regex } { case-odd }
3091         { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
3092         { \tl_count:n {#1} } { \tl_to_str:n {#1} }
3093         \use_ii:nn
3094     }
3095     {
3096         \__regex_replace_all_aux:nnN
3097         { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
3098         { \__regex_case_replacement:e { \__regex_tl_even_items:n {#1} } }
3099         #2
3100         \bool_if:NTF \g__regex_success_bool
3101     }
3102 }
3103 \cs_new_protected:Npn \regex_replace_case_all:nN #1#2
3104 { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
3105 \cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
3106 { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }

```



```

3107 \cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
3108 { \regex_replace_case_all:nNTF {#1} {#2} { } }

```

(`\regex_replace_case_all:nNTF` 定义结束。这个函数被记录在第18页。)

9.7.1 Variables and helpers for user functions

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

```

3109 \int_new:N \l__regex_match_count_int

```

(`\l__regex_match_count_int` 定义结束。)

`__regex_begin` Those flags are raised to indicate begin-group or end-group tokens that had to be added when extracting submatches.

```

3110 \flag_new:n { __regex_begin }
3111 \flag_new:n { __regex_end }

```

(`__regex_begin` 和 `__regex_end` 定义结束。)

`\l__regex_min_submatch_int` The end-points of each submatch are stored in two arrays whose index $\langle submatch \rangle$
`\l__regex_submatch_int` ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int`
`\l__regex_zeroth_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```

3112 \int_new:N \l__regex_min_submatch_int
3113 \int_new:N \l__regex_submatch_int
3114 \int_new:N \l__regex_zeroth_submatch_int

```

(`\l__regex_min_submatch_int`, `\l__regex_submatch_int`, 和 `\l__regex_zeroth_submatch_int` 定义结束。)

`\g__regex_submatch_prev_intarray` Hold the place where the match attempt begun, the end-points of each submatch,
`\g__regex_submatch_begin_intarray` and which regex case the match corresponds to, respectively.

```

\g__regex_submatch_end_intarray 3115 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\g__regex_submatch_case_intarray 3116 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
3117 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
3118 \intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }

```

(`\g__regex_submatch_prev_intarray` 以及其它的定义结束。)

`\g__regex_balance_intarray` The first thing we do when matching is to store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
3119 \intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(`\g__regex_balance_intarray` 定义结束。)

`\l__regex_added_begin_int` `\l__regex_added_end_int` Keep track of the number of left/right braces to add when performing a regex operation such as a replacement.

```
3120 \int_new:N \l__regex_added_begin_int
```

```
3121 \int_new:N \l__regex_added_end_int
```

(`\l__regex_added_begin_int` 和 `\l__regex_added_end_int` 定义结束。)

`__regex_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```
3122 \cs_new_protected:Npn \__regex_return:
```

```
3123 {
```

```
3124   \if_meaning:w \c_true_bool \g__regex_success_bool
```

```
3125     \prg_return_true:
```

```
3126   \else:
```

```
3127     \prg_return_false:
```

```
3128   \fi:
```

```
3129 }
```

(`__regex_return:` 定义结束。)

`__regex_query_set:n` `__regex_query_set_aux:nN` To easily extract subsets of the input once we found the positions at which to cut, store the input tokens one by one into successive `\toks` registers. Also store the brace balance (used to check for overall brace balance) in an array.

```
3130 \cs_new_protected:Npn \__regex_query_set:n #1
```

```
3131 {
```

```
3132   \int_zero:N \l__regex_balance_int
```

```
3133   \int_zero:N \l__regex_curr_pos_int
```

```
3134   \__regex_query_set_aux:nN { } F
```

```
3135   \tl_analysis_map_inline:nn {#1}
```

```
3136     { \__regex_query_set_aux:nN {##1} ##3 }
```

```
3137   \__regex_query_set_aux:nN { } F
```

```
3138   \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
```

```
3139 }
```

```
3140 \cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
```

```
3141 {
```

```
3142   \int_incr:N \l__regex_curr_pos_int
```

```

3143     \_regex_toks_set:Nn \l__regex_curr_pos_int {#1}
3144     \_kernel_intarray_gset:Nnn \g__regex_balance_intarray
3145         { \l__regex_curr_pos_int } { \l__regex_balance_int }
3146     \if_case:w "#2 \exp_stop_f:
3147     \or: \int_incr:N \l__regex_balance_int
3148     \or: \int_decr:N \l__regex_balance_int
3149     \fi:
3150 }

```

(`_regex_query_set:n` 和 `_regex_query_set_aux:nN` 定义结束。)

9.7.2 Matching

`_regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```

3151 \cs_new_protected:Npn \_regex_if_match:nn #1#2
3152 {
3153     \group_begin:
3154         \_regex_disable_submatches:
3155         \_regex_single_match:
3156         #1
3157         \_regex_match:n {#2}
3158     \group_end:
3159 }

```

(`_regex_if_match:nn` 定义结束。)

`_regex_match_case:nnTF` The code would get badly messed up if the number of items in #1 were not even, so we catch this case, then follow the same code as `\regex_match:nnTF` but using `_regex_case_build:n` and without returning a result.

`_regex_match_case_aux:nn`

```

3160 \cs_new_protected:Npn \_regex_match_case:nnTF #1#2
3161 {
3162     \int_if_odd:nTF { \tl_count:n {#1} }
3163     {
3164         \msg_error:nneeee { regex } { case-odd }
3165         { \token_to_str:N \regex_match_case:nn(TF) } { code }
3166         { \tl_count:n {#1} } { \tl_to_str:n {#1} }
3167         \use_ii:nn
3168     }
3169     {
3170         \_regex_if_match:nn
3171         { \_regex_case_build:e { \_regex_tl_odd_items:n {#1} } }
3172         {#2}

```

```

3173         \bool_if:NTF \g__regex_success_bool
3174     }
3175 }
3176 \cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }

(\__regex_match_case:nnTF 和 \__regex_match_case_aux:nn 定义结束。)

```

`__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first “longest match” is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```

3177 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
3178 {
3179     \group_begin:
3180         \__regex_disable_submatches:
3181         \int_zero:N \l__regex_match_count_int
3182         \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
3183         #1
3184         \__regex_match:n {#2}
3185         \exp_args:NNNo
3186     \group_end:
3187     \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
3188 }

(\__regex_count:nnN 定义结束。)

```

9.7.3 Extracting submatches

`__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `__regex_extract:.` At the end, store the sequence containing all the submatches into the user variable `#3` after closing the group.

`__regex_extract_all:nnN`

```

3189 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
3190 {
3191     \group_begin:
3192         \__regex_single_match:
3193         #1
3194         \__regex_match:n {#2}
3195         \__regex_extract:
3196         \__regex_query_set:n {#2}
3197         \__regex_group_end_extract_seq:N #3
3198     }
3199 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3

```

```

3200 {
3201     \group_begin:
3202     \__regex_multi_match:n { \__regex_extract: }
3203     #1
3204     \__regex_match:n {#2}
3205     \__regex_query_set:n {#2}
3206     \__regex_group_end_extract_seq:N #3
3207 }

```

(`__regex_extract_once:nnN` 和 `__regex_extract_all:nnN` 定义结束。)

`__regex_split:nnN` Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```

3208 \cs_new_protected:Npn \__regex_split:nnN #1#2#3
3209 {
3210     \group_begin:
3211     \__regex_multi_match:n
3212     {
3213         \if_int_compare:w
3214             \l__regex_start_pos_int < \l__regex_success_pos_int
3215             \__regex_extract:
3216             \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
3217             { \l__regex_zeroth_submatch_int } { 0 }
3218             \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
3219             { \l__regex_zeroth_submatch_int }
3220             {
3221                 \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
3222                 { \l__regex_zeroth_submatch_int }
3223             }
3224             \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
3225             { \l__regex_zeroth_submatch_int }
3226             { \l__regex_start_pos_int }
3227         \fi:
3228     }
3229     #1
3230     \__regex_match:n {#2}

```

```

3231     \__regex_query_set:n {#2}
3232     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
3233         { \l__regex_submatch_int } { 0 }
3234     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
3235         { \l__regex_submatch_int }
3236         { \l__regex_max_pos_int }
3237     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
3238         { \l__regex_submatch_int }
3239         { \l__regex_start_pos_int }
3240     \int_incr:N \l__regex_submatch_int
3241     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
3242         \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
3243             \int_decr:N \l__regex_submatch_int
3244         \fi:
3245     \fi:
3246     \__regex_group_end_extract_seq:N #3
3247 }

```

(__regex_split:nnN 定义结束。)

`__regex_group_end_extract_seq:N` The end-points of submatches are stored as entries of two arrays from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` (exclusive). Extract the relevant
`__regex_extract_seq:N` ranges into `\g__regex_internal_tl`, separated by `__regex_tmp:w {}`. We keep
`__regex_extract_seq:NNn` track in the two flags `__regex_begin` and `__regex_end` of the number of begin-group
`__regex_extract_seq_loop:Nw` or end-group tokens added to make each of these items overall balanced. At this
step, `}{` is counted as being balanced (same number of begin-group and end-group
tokens). This problem is caught by `__regex_extract_check:w`, explained later.
After complaining about any begin-group or end-group tokens we had to add, we
are ready to construct the user's sequence outside the group.

```

3248 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
3249 {
3250     \flag_clear:n { __regex_begin }
3251     \flag_clear:n { __regex_end }
3252     \cs_set_eq:NN \__regex_tmp:w \scan_stop:
3253     \__kernel_tl_gset:Ne \g__regex_internal_tl
3254     {
3255         \int_step_function:nnN { \l__regex_min_submatch_int }
3256             { \l__regex_submatch_int - 1 } \__regex_extract_seq_aux:n
3257         \__regex_tmp:w
3258     }
3259     \int_set:Nn \l__regex_added_begin_int

```

```

3260     { \flag_height:n { __regex_begin } }
3261 \int_set:Nn \l__regex_added_end_int
3262     { \flag_height:n { __regex_end } }
3263 \tex_afterassignment:D \__regex_extract_check:w
3264 \__kernel_tl_gset:Ne \g__regex_internal_tl
3265     { \g__regex_internal_tl \if_false: { \fi: } }
3266 \int_compare:nNnT
3267     { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
3268     {
3269         \msg_error:nneee { regex } { result-unbalanced }
3270         { splitting~or~extracting~submatches }
3271         { \int_use:N \l__regex_added_begin_int }
3272         { \int_use:N \l__regex_added_end_int }
3273     }
3274 \group_end:
3275 \__regex_extract_seq:N #1
3276 }
3277 \cs_gset_protected:Npn \__regex_extract_seq:N #1
3278 {
3279     \seq_clear:N #1
3280     \cs_set_eq:NN \__regex_tmp:w \__regex_extract_seq_loop:Nw
3281     \exp_after:wN \__regex_extract_seq:NNn
3282     \exp_after:wN #1
3283     \g__regex_internal_tl \use_none:nnn
3284 }
3285 \cs_new_protected:Npn \__regex_extract_seq:NNn #1#2#3
3286     { #3 #2 #1 \prg_do_nothing: }
3287 \cs_new_protected:Npn \__regex_extract_seq_loop:Nw #1#2 \__regex_tmp:w #3
3288     {
3289         \seq_put_right:No #1 {#2}
3290         #3 \__regex_extract_seq_loop:Nw #1 \prg_do_nothing:
3291     }

```

(`__regex_group_end_extract_seq:N` 以及其它的定义结束。)

`__regex_extract_seq_aux:n` The `:n` auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

3292 \cs_new:Npn \__regex_extract_seq_aux:n #1
3293 {
3294     \__regex_tmp:w { }
3295     \exp_after:wN \__regex_extract_seq_aux:ww

```

```

3296     \int_value:w \__regex_submatch_balance:n {#1} ; #1;
3297   }
3298   \cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
3299   {
3300     \if_int_compare:w #1 < \c_zero_int
3301       \prg_replicate:nn {-#1}
3302       {
3303         \flag_raise:n { __regex_begin }
3304         \exp_not:n { { \if_false: } \fi: }
3305       }
3306     \fi:
3307     \__regex_query_submatch:n {#2}
3308     \if_int_compare:w #1 > \c_zero_int
3309       \prg_replicate:nn {#1}
3310       {
3311         \flag_raise:n { __regex_end }
3312         \exp_not:n { \if_false: { \fi: } }
3313       }
3314     \fi:
3315   }

```

(`__regex_extract_seq_aux:n` 和 `__regex_extract_seq_aux:ww` 定义结束。)

`__regex_extract_check:w` In `__regex_group_end_extract_seq:N` we had to expand `\g__regex_internal_tl` to turn `\if_false:` constructions into actual begin-group and end-group tokens. This is done with a `__kernel_tl_gset:Ne` assignment, and `__regex_extract_check:w` is run immediately after this assignment ends, thanks to the `\afterassignment` primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so `}{` is not) then `__regex_extract_check:w` is called just before the closing brace of the `__kernel_tl_gset:Ne` (thanks to our sneaky `\if_false: { \fi: }` construction), and finds that there is nothing left to expand. If any of the items is unbalanced, the assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `__kernel_tl_gset:Ne` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```

3316   \cs_new_protected:Npn \__regex_extract_check:w
3317   {
3318     \exp_after:wN \__regex_extract_check:n
3319     \exp_after:wN { \if_false: } \fi:
3320   }

```



```

3321 \cs_new_protected:Npn \__regex_extract_check:n #1
3322 {
3323   \tl_if_empty:nF {#1}
3324   {
3325     \int_incr:N \l__regex_added_begin_int
3326     \int_incr:N \l__regex_added_end_int
3327     \tex_afterassignment:D \__regex_extract_check:w
3328     \__kernel_tl_gset:Ne \g__regex_internal_tl
3329     {
3330       \exp_after:wN \__regex_extract_check_loop:w
3331       \g__regex_internal_tl
3332       \__regex_tmp:w \__regex_extract_check_end:w
3333       #1
3334     }
3335   }
3336 }
3337 \cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
3338 {
3339   #2
3340   \exp_not:o {#1}
3341   \__regex_tmp:w { }
3342   \__regex_extract_check_loop:w \prg_do_nothing:
3343 }

```

Arguments of `__regex_extract_check_end:w` are: #1 is the part of the item before the extra end-group token; #2 is junk; #3 is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like #3), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```

3344 \cs_new:Npn \__regex_extract_check_end:w
3345   \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
3346 {
3347   { \exp_not:o {#1} }
3348   #3
3349   \if_false: { \fi: }
3350   \__regex_tmp:w
3351 }

```

(`__regex_extract_check:w` 以及其它的定义结束。)

`__regex_extract:` Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from `l__regex_zeroth_submatch_int` upwards. First, we store in `g__regex_submatch_prev_intarray` the position at which the match attempt started. We extract the rest from the comma list `l__regex_success_submatches_tl`, which starts with entries to be stored in `g__regex_submatch_begin_intarray` and continues with entries for `g__regex_submatch_end_intarray`.

```

3352 \cs_new_protected:Npn \__regex_extract:
3353 {
3354   \if_meaning:w \c_true_bool \g__regex_success_bool
3355   \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
3356   \prg_replicate:nn \l__regex_capturing_group_int
3357   {
3358     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
3359     { \l__regex_submatch_int } { 0 }
3360     \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
3361     { \l__regex_submatch_int } { 0 }
3362     \int_incr:N \l__regex_submatch_int
3363   }
3364   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
3365   { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
3366   \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
3367   { \l__regex_zeroth_submatch_int } { \g__regex_case_int }
3368   \int_zero:N \l__regex_internal_a_int
3369   \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
3370   \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
3371   \q__regex_recursion_stop
3372   \fi:
3373 }
3374 \cs_new_protected:Npn \__regex_extract_aux:w #1 ,
3375 {
3376   \prg_break: #1 \prg_break_point:
3377   \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
3378     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
3379     { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#1
3380   \else:
3381     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
3382     { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int - \l
3383   \fi:
3384   \int_incr:N \l__regex_internal_a_int
3385   \__regex_extract_aux:w

```

```
3386 }
```

(`_regex_extract:` 和 `_regex_extract_aux:w` 定义结束。)

9.7.4 Replacement

```
\_regex_replace_once:nnN  
  \_regex_replace_once_aux:nnN
```

Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional `e`-expansion, and checks that braces are balanced in the final result.

```
3387 \cs_new_protected:Npn \_regex_replace_once:nnN #1#2  
3388   { \_regex_replace_once_aux:nnN {#1} { \_regex_replacement:n {#2} } }  
3389 \cs_new_protected:Npn \_regex_replace_once_aux:nnN #1#2#3  
3390   {  
3391     \group_begin:  
3392     \_regex_single_match:  
3393     #1  
3394     \exp_args:No \_regex_match:n {#3}  
3395     \bool_if:NTF \g__regex_success_bool  
3396     {  
3397       \_regex_extract:  
3398       \exp_args:No \_regex_query_set:n {#3}  
3399       #2  
3400       \int_set:Nn \l__regex_balance_int  
3401       {  
3402         \_regex_replacement_balance_one_match:n  
3403         { \l__regex_zeroth_submatch_int }  
3404       }  
3405       \__kernel_tl_set:Ne \l__regex_internal_a_tl  
3406       {  
3407         \_regex_replacement_do_one_match:n  
3408         { \l__regex_zeroth_submatch_int }  
3409         \_regex_query_range:nn  
3410         {  
3411           \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray  
3412           { \l__regex_zeroth_submatch_int }
```

```

3413         }
3414         { \l__regex_max_pos_int }
3415     }
3416     \__regex_group_end_replace:N #3
3417 }
3418 { \group_end: }
3419 }

```

(`__regex_replace_once:nnN` 和 `__regex_replace_once_aux:nnN` 定义结束。)

`__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

3420 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2
3421 { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
3422 \cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
3423 {
3424     \group_begin:
3425     \__regex_multi_match:n { \__regex_extract: }
3426     #1
3427     \exp_args:No \__regex_match:n {#3}
3428     \exp_args:No \__regex_query_set:n {#3}
3429     #2
3430     \int_set:Nn \l__regex_balance_int
3431     {
3432         0
3433         \int_step_function:nnnN
3434         { \l__regex_min_submatch_int }
3435         \l__regex_capturing_group_int
3436         { \l__regex_submatch_int - 1 }
3437         \__regex_replacement_balance_one_match:n
3438     }
3439     \__kernel_tl_set:Ne \l__regex_internal_a_tl
3440     {
3441         \int_step_function:nnnN
3442         { \l__regex_min_submatch_int }
3443         \l__regex_capturing_group_int

```

```

3444         { \l__regex_submatch_int - 1 }
3445         \__regex_replacement_do_one_match:n
3446         \__regex_query_range:nn
3447         \l__regex_start_pos_int \l__regex_max_pos_int
3448     }
3449     \__regex_group_end_replace:N #3
3450 }

```

(__regex_replace_all:nnN 定义结束。)

__regex_group_end_replace:N At this stage \l__regex_internal_a_tl (e-expands to the desired result). Guess from \l__regex_balance_int the number of braces to add before or after the result then try expanding. The simplest case is when \l__regex_internal_a_tl together with the braces we insert via \prg_replicate:nn give a balanced result, and the assignment ends at the \if_false: { \fi: } construction: then __regex_group_end_replace_check:w sees that there is no material left and we successfully found the result. The harder case is that expanding \l__regex_internal_a_tl may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has not yet been expanded so that __regex_group_end_replace_check:n grabs everything until the last brace in __regex_group_end_replace_try:, letting us try again with an extra surrounding pair of braces.

```

3451 \cs_new_protected:Npn \__regex_group_end_replace:N #1
3452 {
3453     \int_set:Nn \l__regex_added_begin_int
3454     { \int_max:nn { - \l__regex_balance_int } { 0 } }
3455     \int_set:Nn \l__regex_added_end_int
3456     { \int_max:nn { \l__regex_balance_int } { 0 } }
3457     \__regex_group_end_replace_try:
3458     \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
3459     {
3460         \msg_error:nneee { regex } { result-unbalanced }
3461         { replacing } { \int_use:N \l__regex_added_begin_int }
3462         { \int_use:N \l__regex_added_end_int }
3463     }
3464     \group_end:
3465     \tl_set_eq:NN #1 \g__regex_internal_tl
3466 }
3467 \cs_new_protected:Npn \__regex_group_end_replace_try:
3468 {
3469     \tex_afterassignment:D \__regex_group_end_replace_check:w

```

```

3470     \__kernel_tl_gset:Ne \g__regex_internal_tl
3471     {
3472         \prg_replicate:nn { \l__regex_added_begin_int } { { \if_false: } \fi: }
3473         \l__regex_internal_a_tl
3474         \prg_replicate:nn { \l__regex_added_end_int } { { \if_false: { \fi: } }
3475         \if_false: { \fi: }
3476     }
3477 }
3478 \cs_new_protected:Npn \__regex_group_end_replace_check:w
3479 {
3480     \exp_after:wN \__regex_group_end_replace_check:n
3481     \exp_after:wN { \if_false: } \fi:
3482 }
3483 \cs_new_protected:Npn \__regex_group_end_replace_check:n #1
3484 {
3485     \tl_if_empty:nF {#1}
3486     {
3487         \int_incr:N \l__regex_added_begin_int
3488         \int_incr:N \l__regex_added_end_int
3489         \__regex_group_end_replace_try:
3490     }
3491 }

```

(*__regex_group_end_replace:n* 以及其它的定义结束。)

9.7.5 Peeking ahead

\l__regex_peek_true_tl True/false code arguments of *\peek_regex:nTF* or similar.

```

\l__regex_peek_false_tl 3492 \tl_new:N \l__regex_peek_true_tl
3493 \tl_new:N \l__regex_peek_false_tl

```

(*\l__regex_peek_true_tl* 和 *\l__regex_peek_false_tl* 定义结束。)

\l__regex_replacement_tl When peeking in *\peek_regex_replace_once:nnTF* we need to store the replacement text.

```

3494 \tl_new:N \l__regex_replacement_tl

```

(*\l__regex_replacement_tl* 定义结束。)

\l__regex_input_tl Stores each token found as *__regex_input_item:n {⟨tokens⟩}*, where the *⟨tokens⟩* o-expand to the token found, as for *\tl_analysis_map_inline:nn*.

__regex_input_item:n

```

3495 \tl_new:N \l__regex_input_tl
3496 \cs_new_eq:NN \__regex_input_item:n ?

```

(`_regex_input_tl` 和 `_regex_input_item:n` 定义结束。)

`\peek_regex:nTF` The T and F functions just call the corresponding TF function. The four TF functions differ along two axes: whether to remove the token or not, distinguished by using `\peek_regex:NTF` `_regex_peek_end:` or `_regex_peek_remove_end:n` (the latter case needs an argument, as we will see), and whether the regex has to be compiled or is already in an N-type variable, distinguished by calling `_regex_build_aux:Nn` or `_regex_build_aux:NN`. The first argument of these functions is `\c_false_bool` to indicate that there should be no implicit insertion of a wildcard at the start of the pattern: otherwise the code would keep looking further into the input stream until matching the regex.

```
3497 \cs_new_protected:Npn \peek_regex:nTF #1
3498 {
3499   \_regex_peek:nnTF
3500   { \_regex_build_aux:Nn \c_false_bool {#1} }
3501   { \_regex_peek_end: }
3502 }
3503 \cs_new_protected:Npn \peek_regex:nT #1#2
3504 { \peek_regex:nTF {#1} {#2} { } }
3505 \cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
3506 \cs_new_protected:Npn \peek_regex:NTF #1
3507 {
3508   \_regex_peek:nnTF
3509   { \_regex_build_aux:NN \c_false_bool #1 }
3510   { \_regex_peek_end: }
3511 }
3512 \cs_new_protected:Npn \peek_regex:NT #1#2
3513 { \peek_regex:NTF #1 {#2} { } }
3514 \cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
3515 \cs_new_protected:Npn \peek_regex_remove_once:nTF #1
3516 {
3517   \_regex_peek:nnTF
3518   { \_regex_build_aux:Nn \c_false_bool {#1} }
3519   { \_regex_peek_remove_end:n {##1} }
3520 }
3521 \cs_new_protected:Npn \peek_regex_remove_once:nT #1#2
3522 { \peek_regex_remove_once:nTF {#1} {#2} { } }
3523 \cs_new_protected:Npn \peek_regex_remove_once:nF #1
3524 { \peek_regex_remove_once:nTF {#1} { } }
3525 \cs_new_protected:Npn \peek_regex_remove_once:NTF #1
3526 {
```

```

3527     \_regex_peek:nnTF
3528     { \_regex_build_aux:NN \c_false_bool #1 }
3529     { \_regex_peek_remove_end:n {##1} }
3530 }
3531 \cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
3532 { \peek_regex_remove_once:NTF #1 {#2} { } }
3533 \cs_new_protected:Npn \peek_regex_remove_once:NF #1
3534 { \peek_regex_remove_once:NTF #1 { } }

```

(`\peek_regex:nTF` 以及其它的定义结束。这些函数被记录在第??页。)

`_regex_peek:nnTF` Store the user's true/false codes (plus `\group_end:`) into two token lists. Then
`_regex_peek_aux:nnTF` build the automaton with #1, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like `\regex_match:nnTF`, with the addition of `\l__regex_input_tl` that keeps track of the tokens seen, to reinsert them at the end. Instead of `\tl_analysis_map_inline:nn` on the input, we call `\peek_analysis_map_inline:n` to go through tokens in the input stream. Since `_regex_match_one_token:nnN` calls `_regex_maplike_break:` we need to catch that and break the `\peek_analysis_map_inline:n` loop instead.

```

3535 \cs_new_protected:Npn \_regex_peek:nnTF #1
3536 {
3537     \_regex_peek_aux:nnTF
3538     {
3539         \_regex_disable_submatches:
3540         #1
3541     }
3542 }
3543 \cs_new_protected:Npn \_regex_peek_aux:nnTF #1#2#3#4
3544 {
3545     \group_begin:
3546     \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
3547     \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
3548     \_regex_single_match:
3549     #1
3550     \_regex_match_init:
3551     \tl_build_begin:N \l__regex_input_tl
3552     \_regex_match_once_init:
3553     \peek_analysis_map_inline:n
3554     {
3555         \tl_build_put_right:Nn \l__regex_input_tl

```



```

3556         { \_regex_input_item:n {##1} }
3557     \_regex_match_one_token:nnN {##1} {##2} ##3
3558     \use_none:nnn
3559     \prg_break_point:Nn \_regex_maplike_break:
3560         { \peek_analysis_map_break:n {#2} }
3561     }
3562 }

```

(`_regex_peek:nnTF` 和 `_regex_peek_aux:nnTF` 定义结束。)

`_regex_peek_end:` Once the regex matches (or permanently fails to match) we call `_regex_peek_end:`, or `_regex_peek_remove_end:n` with argument the last token seen. For `\peek_regex:nTF` we reinsert tokens seen by calling `_regex_peek_reinsert:N` regardless of the result of the match. For `\peek_regex_remove_once:nTF` we reinsert the tokens seen only if the match failed; otherwise we just reinsert the tokens #1, with one expansion. To be more precise, #1 consists of tokens that o-expand and e-expand to the last token seen, for example it is `\exp_not:N <cs>` for a control sequence. This means that just doing `\exp_after:wN \l__regex_peek_true_tl #1` would be unsafe because the expansion of `<cs>` would be suppressed.

```

3563 \cs_new_protected:Npn \_regex_peek_end:
3564 {
3565     \bool_if:NTF \g__regex_success_bool
3566         { \_regex_peek_reinsert:N \l__regex_peek_true_tl }
3567         { \_regex_peek_reinsert:N \l__regex_peek_false_tl }
3568 }
3569 \cs_new_protected:Npn \_regex_peek_remove_end:n #1
3570 {
3571     \bool_if:NTF \g__regex_success_bool
3572         { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
3573         { \_regex_peek_reinsert:N \l__regex_peek_false_tl }
3574 }

```

(`_regex_peek_end:` 和 `_regex_peek_remove_end:n` 定义结束。)

`_regex_peek_reinsert:N` Insert the true/false code #1, followed by the tokens found, which were stored in `\l__regex_input_tl`. For this, loop through that token list using `_regex_reinsert_item:n`, which expands #1 once to get a single token, and jumps over it to expand what follows, with suitable `\exp:w` and `\exp_end:.` We cannot just use `\use:e` on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```

3575 \cs_new_protected:Npn \_regex_peek_reinsert:N #1

```

```

3576 {
3577     \tl_build_end:N \l__regex_input_tl
3578     \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
3579     \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
3580 }
3581 \cs_new_protected:Npn \__regex_reinsert_item:n #1
3582 {
3583     \exp_after:wN \exp_after:wN
3584     \exp_after:wN \exp_end:
3585     \exp_after:wN \exp_after:wN
3586     #1
3587     \exp:w
3588 }

```

(*__regex_peek_reinsert:N* 和 *__regex_reinsert_item:n* 定义结束。)

\peek_regex_replace_once:nn Similar to *\peek_regex:nTF* above.

```

\peek_regex_replace_once:nnTF 3589 \cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
\peek_regex_replace_once:Nn 3590 { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
\peek_regex_replace_once:NnTF 3591 \cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
3592 { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
3593 \cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2
3594 { \peek_regex_replace_once:nnTF {#1} {#2} { } }
3595 \cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
3596 { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
3597 \cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
3598 { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
3599 \cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
3600 { \peek_regex_replace_once:NnTF #1 {#2} {#3} { } }
3601 \cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
3602 { \peek_regex_replace_once:NnTF #1 {#2} { } }
3603 \cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
3604 { \peek_regex_replace_once:NnTF #1 {#2} { } { } }

```

(*\peek_regex_replace_once:nnTF* 和 *\peek_regex_replace_once:NnTF* 定义结束。这些函数被记录在第??页。)

__regex_peek_replace:nnTF Same as *__regex_peek:nnTF* (used for *\peek_regex:nTF* above), but without disabling submatches, and with a different end. The replacement text #2 is stored, to be analyzed later.

```

3605 \cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
3606 {
3607     \tl_set:Nn \l__regex_replacement_tl {#2}
3608     \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }

```

3609 }

(__regex_peek_replace:nnTF 定义结束。)

__regex_peek_replace_end: If the match failed __regex_peek_reinsert:N reinserts the tokens found. Otherwise, finish storing the submatch information using __regex_extract:, and store the input into \toks. Redefine a few auxiliaries to change slightly their expansion behaviour as explained below. Analyse the replacement text with __regex_replacement:n, which as usual defines __regex_replacement_do_one_match:n to insert the tokens from the start of the match attempt to the beginning of the match, followed by the replacement text. The \use:e expands for instance the trailing __regex_query_range:nn down to a sequence of __regex_reinsert_item:n {<tokens>} where <tokens> o-expand to a single token that we want to insert. After e-expansion, \use:e does \use:n, so we have \exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:. This is set up such as to obtain \l__regex_peek_true_tl followed by the replaced tokens (possibly unbalanced) in the input stream.

```
3610 \cs_new_protected:Npn \__regex_peek_replace_end:
3611 {
3612   \bool_if:NTF \g__regex_success_bool
3613   {
3614     \__regex_extract:
3615     \__regex_query_set_from_input_tl:
3616     \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
3617     \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
3618     \__regex_peek_replacement_put_submatch_aux:n
3619     \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
3620     \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
3621     \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
3622     \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
3623     \use:e
3624     {
3625       \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
3626       \__regex_replacement_do_one_match:n
3627       { \l__regex_zeroth_submatch_int }
3628       \__regex_query_range:nn
3629       {
3630         \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
3631         { \l__regex_zeroth_submatch_int }
3632       }
3633       { \l__regex_max_pos_int }
3634     } \exp_end:
```

```

3635     }
3636   }
3637   { \_regex_peek_reinsert:N \l__regex_peek_false_tl }
3638 }

```

(_regex_peek_replace_end: 定义结束。)

_regex_query_set_from_input_tl: The input was stored into \l__regex_input_tl as successive items _regex_input_item:n {tokens}. Store that in successive \toks. It's not clear whether the empty entries before and after are both useful.

```

3639 \cs_new_protected:Npn \_regex_query_set_from_input_tl:
3640 {
3641   \tl_build_end:N \l__regex_input_tl
3642   \int_zero:N \l__regex_curr_pos_int
3643   \cs_set_eq:NN \_regex_input_item:n \_regex_query_set_item:n
3644   \_regex_query_set_item:n { }
3645   \l__regex_input_tl
3646   \_regex_query_set_item:n { }
3647   \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
3648 }
3649 \cs_new_protected:Npn \_regex_query_set_item:n #1
3650 {
3651   \int_incr:N \l__regex_curr_pos_int
3652   \_regex_toks_set:Nn \l__regex_curr_pos_int { \_regex_input_item:n {#1} }
3653 }

```

(_regex_query_set_from_input_tl: 和 _regex_query_set_item:n 定义结束。)

_regex_peek_replacement_put:n While building the replacement function _regex_replacement_do_one_match:n, we often want to put simple material, given as #1, whose e-expansion o-expands to a single token. Normally we can just add the token to \l__regex_build_tl, but for \peek_regex_replace_once:nnTF we eventually want to do some strange expansion that is basically using \exp_after:wN to jump through numerous tokens (we cannot use e-expansion like for \regex_replace_once:nnNTF because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because \cs:w ... \cs_end: does all the expansion we need.

```

3654 \cs_new_protected:Npn \_regex_peek_replacement_put:n #1
3655 {
3656   \if_case:w \l__regex_replacement_csnames_int
3657     \tl_build_put_right:Nn \l__regex_build_tl

```

```

3658         { \exp_not:N \__regex_reinsert_item:n {#1} }
3659     \else:
3660         \tl_build_put_right:Nn \l__regex_build_tl {#1}
3661     \fi:
3662 }

```

(__regex_peek_replacement_put:n 定义结束。)

`__regex_peek_replacement_token:n` When hit with `\exp:w`, `__regex_peek_replacement_token:n {⟨token⟩}` stops `\exp_end:` and does `\exp_after:wN ⟨token⟩ \exp:w` to continue expansion after it.

```

3663 \cs_new_protected:Npn \__regex_peek_replacement_token:n #1
3664 { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }

```

(__regex_peek_replacement_token:n 定义结束。)

`__regex_peek_replacement_put_submatch_aux:n` While analyzing the replacement we also have to insert submatches found in the query. Since query items `__regex_input_item:n {⟨tokens⟩}` expand correctly only when surrounded by `\exp:w ... \exp_end:`, and since these expansion controls are not there within csnames (because `\cs:w ... \cs_end:` make them unnecessary in most cases), we have to put `\exp:w` and `\exp_end:` by hand here.

```

3665 \cs_new_protected:Npn \__regex_peek_replacement_put_submatch_aux:n #1
3666 {
3667     \if_case:w \l__regex_replacement_csnames_int
3668         \tl_build_put_right:Nn \l__regex_build_tl
3669             { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
3670     \else:
3671         \tl_build_put_right:Nn \l__regex_build_tl
3672             { \exp:w \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } \exp_end: }
3673     \fi:
3674 }

```

(__regex_peek_replacement_put_submatch_aux:n 定义结束。)

`__regex_peek_replacement_var:N` This is used for `\u` outside csnames. It makes sure to continue expansion with `\exp:w` before expanding the variable `#1` and stopping the `\exp:w` that precedes.

```

3675 \cs_new_protected:Npn \__regex_peek_replacement_var:N #1
3676 {
3677     \exp_after:wN \exp_last_unbraced:NV
3678     \exp_after:wN \exp_end:
3679     \exp_after:wN #1
3680     \exp:w
3681 }

```

(`_regex_peek_replacement_var:N` 定义结束。)

9.8 Messages

Messages for the preparsing phase.

```
3682 \use:e
3683 {
3684   \msg_new:nnn { regex } { trailing-backslash }
3685   { Trailing~'\iow_char:N\\'~in~regex~or~replacement. }
3686   \msg_new:nnn { regex } { x-missing-rbrace }
3687   {
3688     Missing-brace~'\iow_char:N\}'~in~regex~
3689     '...\iow_char:N\x\iow_char:N\{...##1'.
3690   }
3691   \msg_new:nnn { regex } { x-overflow }
3692   {
3693     Character~code~##1~too~large~in~
3694     \iow_char:N\x\iow_char:N\{##2\iow_char:N\}~regex.
3695   }
3696 }
```

Invalid quantifier.

```
3697 \msg_new:nnnn { regex } { invalid-quantifier }
3698 { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
3699 {
3700   The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
3701   The~only~valid~quantifiers~are~'*,~'?',~'+',~'{<int>}',~
3702   '{<min>},~'~and~'{<min>,<max>}',~optionally~followed~by~'?''.
3703 }
```

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

```
3704 \msg_new:nnnn { regex } { missing-rbrack }
3705 { Missing~right~bracket~inserted~in~regular~expression. }
3706 {
3707   LaTeX~was~given~a~regular~expression~where~a~character~class~
3708   was~started~with~'[',~but~the~matching~']'~is~missing.
3709 }
3710 \msg_new:nnnn { regex } { missing-rparen }
3711 {
3712   Missing~right~
3713   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
3714   inserted~in~regular~expression.
```

```

3715 }
3716 {
3717     LaTeX~was~given~a~regular~expression~with~\int_eval:n {#1} ~
3718     more~left~parentheses~than~right~parentheses.
3719 }
3720 \msg_new:nnnn { regex } { extra-rparen }
3721 { Extra~right~parenthesis~ignored~in~regular~expression. }
3722 {
3723     LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
3724     was~open.~The~parenthesis~will~be~ignored.
3725 }

```

Some escaped alphanumerics are not allowed everywhere.

```

3726 \msg_new:nnnn { regex } { bad-escape }
3727 {
3728     Invalid~escape~'\iow_char:N\\#1'~
3729     \__regex_if_in_cs:TF { within~a~control~sequence. }
3730     {
3731         \__regex_if_in_class:TF
3732         { in~a~character~class. }
3733         { following~a~category~test. }
3734     }
3735 }
3736 {
3737     The~escape~sequence~'\iow_char:N\\#1'~may~not~appear~
3738     \__regex_if_in_cs:TF
3739     {
3740         within~a~control~sequence~test~introduced~by~
3741         '\iow_char:N\\c\iow_char:N\{' .
3742     }
3743     {
3744         \__regex_if_in_class:TF
3745         { within~a~character~class~ }
3746         { following~a~category~test~such~as~'\iow_char:N\\cL'~ }
3747         because~it~does~not~match~exactly~one~character.
3748     }
3749 }

```

Range errors.

```

3750 \msg_new:nnnn { regex } { range-missing-end }
3751 { Invalid~end~point~for~range~'#1-#2'~in~character~class. }
3752 {
3753     The~end~point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~

```

```

3754     end-point~for~a~range:~alphanumeric~characters~should~not~be~
3755     escaped,~and~non-alphanumeric~characters~should~be~escaped.
3756 }
3757 \msg_new:nnnn { regex } { range-backwards }
3758 { Range~'[#1-#2]~out-of-order-in-character-class. }
3759 {
3760     In~ranges~of~characters~'[x-y]~appearing~in~character~classes,~
3761     the~first~character~code~must~not~be~larger~than~the~second.~
3762     Here,~'#1'~has~character~code~\int_eval:n {`#1},~while~
3763     '#2'~has~character~code~\int_eval:n {`#2}.
3764 }

```

Errors related to \c and \u.

```

3765 \msg_new:nnnn { regex } { c-bad-mode }
3766 { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
3767 {
3768     The~'\iow_char:N\\c'~escape~cannot~be~used~within~
3769     a~control~sequence~test~'\iow_char:N\\c{...}'~
3770     nor~another~category~test.~
3771     To~combine~several~category~tests,~use~'\iow_char:N\\c[...]' .
3772 }
3773 \msg_new:nnnn { regex } { c-C-invalid }
3774 { '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'(',~not~'#1'. }
3775 {
3776     The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
3777     control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
3778     It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
3779 }
3780 \msg_new:nnnn { regex } { cu-lbrace }
3781 { Left~braces~must~be~escaped~in~'\iow_char:N\\#1{...}' . }
3782 {
3783     Constructions~such~as~'\iow_char:N\\#1{...~\iow_char:N\\{...}'~are~
3784     not~allowed~and~should~be~replaced~by~
3785     '\iow_char:N\\#1{...~\token_to_str:N\\{...}' .
3786 }
3787 \msg_new:nnnn { regex } { c-lparen-in-class }
3788 { Catcode~test~cannot~apply~to~group~in~character~class }
3789 {
3790     Construction~such~as~'\iow_char:N\\cL(abc)~are~not~allowed~inside~a~
3791     class~'[...]'~because~classes~do~not~match~multiple~characters~at~once.
3792 }
3793 \msg_new:nnnn { regex } { c-missing-rbrace }
3794 { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }

```



```

3795 {
3796     LaTeX~was~given~a~regular~expression~where~a~
3797     '\iow_char:N\c\iow_char:N\{...\}'~construction~was~not~ended~
3798     with~a~closing~brace~'\iow_char:N\}'.
3799 }
3800 \msg_new:nnnn { regex } { c-missing-rbrack }
3801 { Missing~right~bracket~inserted~for~'\iow_char:N\c'~escape. }
3802 {
3803     A~construction~'\iow_char:N\c[...\}'~appears~in~a~
3804     regular~expression,~but~the~closing~'\}'~is~not~present.
3805 }
3806 \msg_new:nnnn { regex } { c-missing-category }
3807 { Invalid~character~'#1'~following~'\iow_char:N\c'~escape. }
3808 {
3809     In~regular~expressions,~the~'\iow_char:N\c'~escape~sequence~
3810     may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
3811     capital~letter~representing~a~character~category,~namely~
3812     one~of~'ABCDELMOPTU'.
3813 }
3814 \msg_new:nnnn { regex } { c-trailing }
3815 { Trailing~category~code~escape~'\iow_char:N\c'... }
3816 {
3817     A~regular~expression~ends~with~'\iow_char:N\c'~followed~
3818     by~a~letter.~It~will~be~ignored.
3819 }
3820 \msg_new:nnnn { regex } { u-missing-lbrace }
3821 { Missing~left~brace~following~'\iow_char:N\u'~escape. }
3822 {
3823     The~'\iow_char:N\u'~escape~sequence~must~be~followed~by~
3824     a~brace~group~with~the~name~of~the~variable~to~use.
3825 }
3826 \msg_new:nnnn { regex } { u-missing-rbrace }
3827 { Missing~right~brace~inserted~for~'\iow_char:N\u'~escape. }
3828 {
3829     LaTeX~
3830     \str_if_eq:eeTF { } {#2}
3831     { reached~the~end~of~the~string~ }
3832     { encountered~an~escaped~alphanumeric~character '\iow_char:N\#2'~ }
3833     when~parsing~the~argument~of~an~
3834     '\iow_char:N\u\iow_char:N\{...\}'~escape.
3835 }

```

Errors when encountering the POSIX syntax [:...:].

```

3836 \msg_new:nnnn { regex } { posix-unsupported }
3837 { POSIX-collating-element~'[#1 ~ #1]~not~supported. }
3838 {
3839   The~'[.foo.]~and~'[=bar=]~syntaxes~have~a~special~meaning~
3840   in~POSIX-regular~expressions.~This~is~not~supported~by~LaTeX.~
3841   Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
3842 }
3843 \msg_new:nnnn { regex } { posix-unknown }
3844 { POSIX~class~'[:#1:]~unknown. }
3845 {
3846   '[:#1:]~is~not~among~the~known~POSIX~classes~
3847   '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
3848   '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
3849   '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
3850   '[:word:]',~and~'[:xdigit:]'.
3851 }
3852 \msg_new:nnnn { regex } { posix-missing-close }
3853 { Missing~closing~':'~for~POSIX~class. }
3854 { The~POSIX~syntax~'#1'~must~be~followed~by~':'',~not~'#2'. }

```

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

```

3855 \msg_new:nnnn { regex } { result-unbalanced }
3856 { Missing~brace~inserted~when~#1. }
3857 {
3858   LaTeX~was~asked~to~do~some~regular~expression~operation,~
3859   and~the~resulting~token~list~would~not~have~the~same~number~
3860   of~begin-group~and~end-group~tokens.~Braces~were~inserted:~
3861   #2~left,~#3~right.
3862 }

```

Error message for unknown options.

```

3863 \msg_new:nnnn { regex } { unknown-option }
3864 { Unknown~option~'#1'~for~regular~expressions. }
3865 {
3866   The~only~available~option~is~'case-insensitive',~toggled~by~
3867   '(?i)~and~'(?-i)'.
3868 }
3869 \msg_new:nnnn { regex } { special-group-unknown }
3870 { Unknown~special~group~'#1~...~in~a~regular~expression. }
3871 {
3872   The~only~valid~constructions~starting~with~'('?~are~

```

```

3873     '(:~...~)',~'(|~...~)',~'(?i)',~and~'(?-i)'.
3874 }

```

Errors in the replacement text.

```

3875 \msg_new:nnnn { regex } { replacement-c }
3876 { Misused~'\iow_char:N\c'~command-in~a~replacement~text. }
3877 {
3878     In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
3879     can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
3880     or~a~brace~group,~not~by~'#1'.
3881 }
3882 \msg_new:nnnn { regex } { replacement-u }
3883 { Misused~'\iow_char:N\u'~command-in~a~replacement~text. }
3884 {
3885     In~a~replacement~text,~the~'\iow_char:N\u'~escape~sequence~
3886     must~be~followed~by~a~brace~group~holding~the~name~of~the~
3887     variable~to~use.
3888 }
3889 \msg_new:nnnn { regex } { replacement-g }
3890 {
3891     Missing~brace~for~the~'\iow_char:N\g'~construction~
3892     in~a~replacement~text.
3893 }
3894 {
3895     In~the~replacement~text~for~a~regular~expression~search,~
3896     submatches~are~represented~either~as~'\iow_char:N \g{dd..d}',~
3897     or~'\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
3898 }
3899 \msg_new:nnnn { regex } { replacement-catcode-end }
3900 {
3901     Missing~character~for~the~'\iow_char:N\c<category><character>'~
3902     construction~in~a~replacement~text.
3903 }
3904 {
3905     In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
3906     can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
3907     the~character~category.~Then,~a~character~must~follow.~LaTeX~
3908     reached~the~end~of~the~replacement~when~looking~for~that.
3909 }
3910 \msg_new:nnnn { regex } { replacement-catcode-escaped }
3911 {
3912     Escaped~letter~or~digit~after~category~code~in~replacement~text.
3913 }

```

```

3914 {
3915     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
3916     can~be~followed~by~one~of~the~letters~'ABCDELMOPTU'~representing~
3917     the~character~category.~Then,~a~character~must~follow,~not~
3918     '\iow_char:N\\#2'.
3919 }
3920 \msg_new:nnnn { regex } { replacement-catcode-in-cs }
3921 {
3922     Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
3923     '\iow_char:N\\c\{...\}'~in~a~replacement~text.
3924 }
3925 {
3926     In~a~replacement~text,~the~category~codes~of~the~argument~of~
3927     '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control~
3928     sequence~name.
3929 }
3930 \msg_new:nnnn { regex } { replacement-null-space }
3931 { TeX~cannot~build~a~space~token~with~character~code~0. }
3932 {
3933     You~asked~for~a~character~token~with~category~space,~
3934     and~character~code~0,~for~instance~through~
3935     '\iow_char:N\\cS\iow_char:N\\x00'.~
3936     This~specific~case~is~impossible~and~will~be~replaced~
3937     by~a~normal~space.
3938 }
3939 \msg_new:nnnn { regex } { replacement-missing-rbrace }
3940 { Missing~right~brace~inserted~in~replacement~text. }
3941 {
3942     There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
3943     missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
3944 }
3945 \msg_new:nnnn { regex } { replacement-missing-rparen }
3946 { Missing~right~parenthesis~inserted~in~replacement~text. }
3947 {
3948     There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
3949     missing~right~
3950     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
3951 }
3952 \msg_new:nnn { regex } { submatch-too-big }
3953 { Submatch~#1~used~but~regex~only~has~#2~group(s) }

    Some escaped alphanumerics are not allowed everywhere.
3954 \msg_new:nnnn { regex } { backwards-quantifier }

```

```

3955 { Quantifier~"{#1,#2}"~is~backwards. }
3956 { The~values~given~in~a~quantifier~must~be~in~order. }

```

Used in user commands, and when showing a regex.

```

3957 \msg_new:nnnn { regex } { case-odd }
3958 { #1~with~odd~number~of~items }
3959 {
3960   There~must~be~a~#2~part~for~each~regex:~
3961   found~odd~number~of~items~(#3)~in\\
3962   \iow_indent:n {#4}
3963 }
3964 \msg_new:nnn { regex } { show }
3965 {
3966   >~Compiled~regex~
3967   \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
3968   #3
3969 }
3970 \prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
3971 \prop_gput:Nnn \g_msg_module_type_prop { regex } { }

```

`__regex_msg_repeated:nnN` This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (−1 for infinite number), and #3 tells us about laziness.

```

3972 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
3973 {
3974   \str_if_eq:eeF { #1 #2 } { 1 0 }
3975   {
3976     , ~ repeated ~
3977     \int_case:nnF {#2}
3978     {
3979       { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
3980       { 0 } { #1~times }
3981     }
3982     {
3983       between~#1~and~\int_eval:n {#1+#2}~times,~
3984       \bool_if:NTF #3 { lazy } { greedy }
3985     }
3986   }
3987 }

```

(`__regex_msg_repeated:nnN` 定义结束。)

9.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

```
\__regex_trace_push:nnN Here #1 is the module name (regex) and #2 is typically 1. If the module's current
\__regex_trace_pop:nnN tracing level is less than #2 show nothing, otherwise write #3 to the terminal.
\__regex_trace:nne
3988 \cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
3989 { \__regex_trace:nne {#1} {#2} { entering~ \token_to_str:N #3 } }
3990 \cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
3991 { \__regex_trace:nne {#1} {#2} { leaving~ \token_to_str:N #3 } }
3992 \cs_new_protected:Npn \__regex_trace:nne #1#2#3
3993 {
3994   \int_compare:nNnF
3995     { \int_use:c { g__regex_trace_#1_int } } < {#2}
3996     { \iow_term:e { Trace:~#3 } }
3997 }

(\__regex_trace_push:nnN, \__regex_trace_pop:nnN, 和 \__regex_trace:nne 定义结束。)
```

```
\g__regex_trace_regex_int No tracing when that is zero.
3998 \int_new:N \g__regex_trace_regex_int

(\g__regex_trace_regex_int 定义结束。)
```

```
\__regex_trace_states:n This function lists the contents of all states of the NFA, stored in \toks from 0 to
\l__regex_max_state_int (excluded).
3999 \cs_new_protected:Npn \__regex_trace_states:n #1
4000 {
4001   \int_step_inline:nnn
4002     \l__regex_min_state_int
4003     { \l__regex_max_state_int - 1 }
4004     {
4005       \__regex_trace:nne { regex } {#1}
4006       { \iow_char:N \toks ##1 = { \__regex_toks_use:w ##1 } }
4007     }
4008 }

(\__regex_trace_states:n 定义结束。)
```

```
4009 </package>
```

索引

斜体数字指向相应条目描述的页面，下划线数字指向定义的代码行，其它的都指向使用条目的页面。

Symbols	
$\backslash \$$	991
$\backslash \backslash$.	5, 329, 330, 1725, 1732, 1733, 1734, 1858, 3685, 3689, 3694, 3728, 3737, 3741, 3746, 3766, 3768, 3769, 3771, 3774, 3776, 3781, 3783, 3785, 3790, 3794, 3797, 3801, 3803, 3807, 3809, 3815, 3817, 3821, 3823, 3827, 3832, 3834, 3876, 3878, 3883, 3885, 3891, 3896, 3897, 3901, 3905, 3915, 3918, 3922, 3923, 3927, 3935, 3961, 4006
$\backslash \{$	278, 3689, 3694, 3741, 3783, 3785, 3797, 3834, 3923, 3927
$\backslash \}$	3688, 3694, 3798, 3834, 3923, 3927
$\backslash \sqcup$	223, 228, 272, 282, 466, 2949
$\backslash ^$	224, 229, 230, 231, 232, 235, 246, 283, 337, 339, 341, 343, 345, 347, 990, 2900, 2903, 2917, 2920, 2929, 2932, 2935, 2938, 2952, 2955
$\backslash \sim$	268, 272, 278
B	
bool commands:	
$\backslash \text{bool_gset_eq:N}$	214, 2383
$\backslash \text{bool_gset_false:N}$	2330
$\backslash \text{bool_gset_true:N}$	2396
$\backslash \text{bool_if:NTF}$	1106, 1115, 1556, 1729, 1815, 1833, 1851, 2005, 2224, 2232, 2465, 3077, 3100, 3173, 3395, 3565, 3571, 3612, 3979, 3984
$\backslash \text{bool_if:nTF}$	1818
$\backslash \text{bool_lazy_all:nTF}$	1389
$\backslash \text{bool_new:N}$	71, 501, 2299, 2300, 2302, 2303, 2304
$\backslash \text{bool_set_eq:N}$	208, 2544
$\backslash \text{bool_set_false:N}$	1080, 1285, 2274, 2345, 2359, 2422, 2464
$\backslash \text{bool_set_true:N}$	1085, 1289, 2268, 2462, 2543
$\backslash \text{c_false_bool}$	151, 766, 784, 978, 1025, 1324, 1526, 1543, 1556, 1752, 1888, 2393, 3500, 3509, 3518, 3528, 3590, 3598
$\backslash \text{c_true_bool}$	78, 211, 649, 723, 780, 968, 970, 972, 974, 976, 986, 1024, 1031, 1524, 1534, 1556, 1557, 1750, 1871, 1873, 1896, 1991, 2162, 2173, 2188, 2349, 3124, 3241, 3354
C	
char commands:	
$\backslash \text{char_generate:nn}$	128, 362, 1399, 1773
$\backslash \text{char_set_catcode_active:N}$	2900
$\backslash \text{char_set_catcode_alignment:N}$	2952
$\backslash \text{char_set_catcode_group_begin:N}$	2903
$\backslash \text{char_set_catcode_group_end:N}$	2920
$\backslash \text{char_set_catcode_letter:N}$	2929
$\backslash \text{char_set_catcode_math_subscript:N}$	2917
$\backslash \text{char_set_catcode_math_superscript:N}$	2955
$\backslash \text{char_set_catcode_math_toggle:N}$	2932
$\backslash \text{char_set_catcode_other:N}$	2935
$\backslash \text{char_set_catcode_parameter:N}$	2938
cs commands:	
$\backslash \text{cs:w}$	126, 156, 157, 2830, 2833
$\backslash \text{cs_end:}$	156, 157, 2639, 2846
$\backslash \text{cs_generate_variant:Nn}$	746, 1899, 2654, 2695, 3024, 3025, 3048, 3050
$\backslash \text{cs_gset:Npn}$	2671
$\backslash \text{cs_gset_protected:Npn}$	3277
$\backslash \text{cs_if_eq:NNTF}$	1374
$\backslash \text{cs_if_eq_p:NN}$	1391
$\backslash \text{cs_if_exist:NTF}$	750, 1102, 2816

\cs_if_exist_use:N	1763	2079, 2091, 2100, 2112, 2125, 2132,
\cs_if_exist_use:NTF	315,	2152, 2183, 2217, 2235, 2244, 2250,
322, 709, 714, 758, 1170, 1256, 2752		2256, 2262, 2305, 2314, 2328, 2347,
\cs_new:Npe 336, 338, 340, 342, 344, 346		2374, 2379, 2389, 2401, 2439, 2448,
\cs_new:Npn	6,	2460, 2467, 2469, 2471, 2491, 2496,
35, 40, 42, 51, 53, 54, 59, 65, 87,		2502, 2513, 2518, 2523, 2539, 2591,
89, 91, 313, 319, 330, 335, 348, 353,		2610, 2612, 2655, 2679, 2696, 2702,
365, 380, 390, 402, 425, 536, 554,		2704, 2724, 2750, 2761, 2770, 2779,
562, 574, 586, 1090, 1372, 1387,		2812, 2826, 2837, 2843, 2852, 2860,
1421, 1437, 1484, 1522, 1553, 1559,		2895, 2901, 2904, 2912, 2918, 2921,
1565, 1573, 1578, 1584, 1589, 1603,		2930, 2933, 2936, 2939, 2944, 2953,
1618, 1627, 1635, 1637, 1689, 1698,		2956, 2959, 2964, 2970, 2975, 2980,
1769, 2015, 2430, 2534, 2537, 2558,		2985, 2986, 2987, 2995, 2996, 2997,
2560, 2566, 2568, 2575, 2585, 2785,		3020, 3022, 3026, 3034, 3036, 3038,
3176, 3292, 3298, 3337, 3344, 3972		3042, 3043, 3063, 3080, 3082, 3084,
\cs_new_eq:NN	3, 9, 157,	3086, 3103, 3105, 3107, 3122, 3130,
158, 159, 258, 266, 287, 326, 327,		3140, 3151, 3160, 3177, 3189, 3199,
328, 329, 515, 2301, 2567, 2965, 3496		3208, 3248, 3285, 3287, 3316, 3321,
\cs_new_protected:Npe . .	930, 944, 946	3352, 3374, 3387, 3389, 3420, 3422,
\cs_new_protected:Npn		3451, 3467, 3478, 3483, 3497, 3503,
.	4, 7, 10, 12, 21, 27,	3505, 3506, 3512, 3514, 3515, 3521,
33, 93, 95, 96, 101, 107, 115, 125,		3523, 3525, 3531, 3533, 3535, 3543,
139, 160, 170, 178, 180, 188, 200,		3563, 3569, 3575, 3581, 3589, 3591,
219, 221, 226, 234, 236, 243, 248,		3593, 3595, 3597, 3599, 3601, 3603,
250, 252, 259, 267, 269, 271, 273,		3605, 3610, 3639, 3649, 3654, 3663,
280, 285, 288, 294, 530, 594, 607,		3665, 3675, 3988, 3990, 3992, 3999
618, 631, 664, 693, 702, 707, 712,		
717, 738, 747, 754, 763, 768, 775,		\cs_set:Npe
788, 790, 792, 794, 800, 822, 835,		2002
862, 867, 901, 936, 957, 959, 967,		
969, 971, 973, 975, 977, 981, 992,		\cs_set:Npn
1008, 1021, 1027, 1038, 1051, 1057,		298, 299, 300, 626, 627, 1205, 1207,
1076, 1096, 1127, 1138, 1153, 1166,		1224, 1226, 1466, 1467, 1731, 1732,
1184, 1192, 1197, 1199, 1201, 1218,		1733, 1734, 1760, 1805, 2350, 2657
1237, 1239, 1262, 1274, 1294, 1315,		
1322, 1329, 1340, 1347, 1353, 1411,		\cs_set_eq:NN
1463, 1472, 1485, 1500, 1509, 1528,		155, 1723,
1547, 1704, 1775, 1785, 1787, 1789,		1757, 2356, 2406, 3252, 3280, 3578,
1796, 1841, 1854, 1870, 1872, 1874,		3616, 3617, 3619, 3620, 3621, 3643
1879, 1894, 1900, 1923, 1946, 1961,		
1968, 1975, 1977, 1979, 1986, 2000,		\cs_set_nopar:Npe
2016, 2025, 2039, 2051, 2068, 2077,		23, 29
		\cs_set_protected:Npn . .
		942, 979,
		1708, 1717, 1719, 1721, 1724, 1726,
		1735, 1737, 1742, 1744, 1749, 1751,
		1753, 1755, 1758, 2515, 2516, 3040
		\cs_to_str:N
		37, 1538, 1674

E

else commands:

`\else:` 46, 143, 144, 149, 150,
167, 174, 374, 384, 434, 443, 455,
456, 458, 460, 463, 464, 467, 468,
477, 479, 481, 484, 485, 487, 523,
526, 547, 550, 558, 566, 569, 578,
581, 590, 598, 601, 611, 731, 845,
889, 893, 896, 907, 912, 1002, 1148,
1161, 1250, 1279, 1318, 1336, 1449,
1505, 1539, 1569, 1994, 2012, 2031,
2065, 2118, 2165, 2169, 2176, 2197,
2208, 2355, 2468, 2578, 2622, 2625,
2745, 2756, 2765, 2793, 2805, 2831,
2848, 2856, 3126, 3380, 3659, 3670

exp commands:

`\exp:w` 153, 155,
157, 3579, 3587, 3625, 3664, 3672, 3680
`\exp_after:wN` 129, 153,
155–157, 11, 24, 25, 30, 31, 34, 37,
45, 47, 104, 111, 118, 122, 129, 135,
173, 175, 184, 306, 309, 350, 368,
373, 375, 376, 383, 386, 387, 393,
405, 417, 436, 445, 557, 559, 565,
568, 570, 577, 580, 582, 589, 591,
597, 600, 603, 697, 939, 1003, 1015,
1147, 1150, 1160, 1162, 1344, 1351,
1359, 1444, 1538, 1570, 1957, 2247,
2253, 2259, 2370, 2394, 2425, 2456,
2482, 2520, 2570, 2571, 2579, 2582,
2700, 2729, 2791, 2792, 2795, 2796,
2804, 2806, 2807, 2830, 2833, 2910,
3281, 3282, 3295, 3318, 3319, 3330,
3369, 3480, 3481, 3579, 3583, 3584,
3585, 3625, 3664, 3677, 3678, 3679
`\exp_args:Nc` 1300
`\exp_args:Ne` . 210, 990, 991, 1362, 2493
`\exp_args:Nf` 41, 1671, 1673, 2096
`\exp_args:NNe` 1468
`\exp_args:NNf` 2368
`\exp_args:NNNe` 660, 1143
`\exp_args:NNNo` . . 652, 1765, 1828, 3185

`\exp_args:NNo` 2650, 3572
`\exp_args:Nno` 2670
`\exp_args:No` . . 742, 806, 826, 1513,
1562, 2070, 2498, 2692, 2707, 2802,
3000, 3394, 3398, 3427, 3428, 3622
`\exp_args:Noo` 848, 2083
`\exp_end:` 153, 155,
157, 3579, 3584, 3634, 3664, 3672, 3678
`\exp_last_unbraced:Nf` 361, 1691
`\exp_last_unbraced:NV` 3677
`\exp_not:N`
. . . . 117, 153, 337, 339, 341, 343,
345, 347, 730, 732, 932, 934, 945,
949, 1107, 1780, 2487, 2902, 2915, 3658
`\exp_not:n` 10,
24, 117, 126, 130, 145, 62, 730,
732, 1365, 1837, 2004, 2222, 2411,
2476, 2488, 2566, 2567, 2830, 2833,
2902, 2910, 2927, 2942, 2983, 3176,
3304, 3312, 3340, 3345, 3347, 3625
`\exp_stop_f:`
. . . . 142, 143, 144, 150, 172, 432,
452, 453, 457, 461, 462, 465, 466,
474, 475, 478, 482, 483, 486, 545,
905, 910, 924, 925, 938, 1000, 1001,
1040, 1317, 2009, 2062, 2577, 2581,
2730, 2754, 2789, 2794, 2800, 3146

F

fi commands:

`\fi:` 144, 145, 149, 48,
105, 112, 113, 119, 123, 130, 131,
136, 137, 147, 148, 153, 154, 168,
176, 185, 186, 213, 369, 377, 388,
394, 406, 446, 448, 455, 458, 459,
463, 467, 468, 469, 470, 479, 480,
484, 487, 488, 489, 525, 528, 549,
552, 560, 571, 572, 583, 584, 592,
604, 605, 615, 616, 629, 648, 649,
657, 658, 723, 733, 766, 780, 784,
847, 895, 898, 899, 914, 917, 940,
998, 999, 1004, 1031, 1032, 1043,
1047, 1081, 1086, 1094, 1129, 1136,

1141, 1151, 1163, 1189, 1252, 1281, 1320, 1327, 1338, 1426, 1445, 1451, 1456, 1479, 1491, 1492, 1495, 1507, 1541, 1571, 1958, 1997, 2013, 2037, 2057, 2066, 2123, 2130, 2150, 2168, 2179, 2181, 2211, 2214, 2248, 2254, 2260, 2357, 2395, 2426, 2427, 2457, 2483, 2528, 2580, 2627, 2628, 2640, 2691, 2693, 2746, 2758, 2768, 2777, 2797, 2808, 2834, 2850, 2858, 2908, 2910, 2925, 2927, 2948, 3128, 3149, 3227, 3244, 3245, 3265, 3304, 3306, 3312, 3314, 3319, 3349, 3372, 3383, 3472, 3474, 3475, 3481, 3661, 3673	<code>\if_false:</code> 129, 144, 145, 149, 629, 648, 649, 658, 723, 766, 780, 784, 998, 1031, 1043, 1047, 1081, 1086, 1094, 1129, 1136, 1141, 1189, 1426, 1445, 1456, 1479, 1491, 1492, 1495, 2910, 2927, 3265, 3304, 3312, 3319, 3349, 3472, 3474, 3475, 3481
flag commands:	<code>\if_int_compare:w</code> 44, 103, 109, 110, 117, 121, 127, 128, 133, 134, 142, 143, 144, 150, 182, 183, 432, 452, 453, 454, 457, 461, 462, 465, 466, 474, 475, 478, 482, 483, 486, 545, 567, 579, 588, 596, 599, 609, 612, 640, 727, 839, 905, 910, 938, 996, 1029, 1140, 1157, 1503, 1536, 1567, 1956, 2027, 2053, 2114, 2127, 2138, 2154, 2205, 2246, 2252, 2258, 2423, 2424, 2451, 2478, 2577, 2635, 2754, 2763, 2774, 2789, 2845, 2854, 2906, 2923, 2946, 3213, 3242, 3300, 3308, 3377
<code>\flag_clear:n</code> 1356, 3250, 3251 <code>\flag_ensure_raised:n</code> 1383, 1405 <code>\flag_height:n</code> 3260, 3262 <code>\flag_if_raised:nTF</code> 1364 <code>\flag_new:n</code> 1346, 3110, 3111 <code>\flag_raise:n</code> 3303, 3311	<code>\if_int_odd:w</code> 172, 556, 564, 576, 1002, 1317
G	<code>\if_meaning:w</code> 211, 367, 392, 404, 520, 544, 887, 890, 1324, 1991, 2162, 2173, 2188, 2349, 2393, 2528, 2801, 3124, 3241, 3354
group commands:	int commands:
<code>\group_begin:</code> 204, 296, 620, 1131, 1465, 1706, 1807, 2237, 2614, 2894, 3153, 3179, 3191, 3201, 3210, 3391, 3424, 3545 <code>\group_end:</code> 152, 216, 310, 653, 661, 1144, 1469, 1766, 1814, 1821, 1829, 2241, 2242, 2651, 2958, 3158, 3186, 3274, 3418, 3464, 3546, 3547 <code>\group_insert_after:N</code> 212	<code>\int_add:Nn</code> 151, 1319, 2144, 2145, 2403, 2475 <code>\int_case:nnTF</code> 3977 <code>\int_compare:nNnTF</code> 190, 202, 355, 682, 684, 1549, 2352, 2706, 2865, 3266, 3458, 3994 <code>\int_compare:nTF</code> 1772, 1812, 3713, 3942, 3943, 3948, 3950 <code>\int_compare_p:n</code> 1819 <code>\int_compare_p:nNn</code> 1394, 1395 <code>\int_const:Nn</code> 81, 82, 83, 84, 493, 494, 495, 496, 497, 498, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514
I	
if commands:	
<code>\if:w</code> 44, 521 <code>\if_case:w</code> 163, 435, 1248, 1277, 1334, 2009, 2062, 2683, 2730, 2828, 3146, 3656, 3667 <code>\if_charcode:w</code> 372, 382, 891, 1447, 2620, 2623	

```

\int_decr:N . . . . 2847, 2924, 3148, 3243
\int_eval:n . . . . . 23, 41, 172, 436,
    1317, 1563, 1771, 1976, 1978, 1992,
    1993, 1995, 1996, 2138, 2228, 2271,
    2446, 2494, 2593, 2767, 2773, 2776,
    3669, 3672, 3717, 3762, 3763, 3983
\int_gincr:N . . . . . 1917
\int_gset:Nn . . . . . 1937
\int_gzero:N . . . . . 1897, 1914
\int_if_exist:NTF . . . . . 1243, 1298
\int_if_odd:nTF . . . . 3065, 3088, 3162
\int_if_odd_p:n . . . . . 1845
\int_incr:N . . . . 17, 18, 1134, 1777,
    1944, 1984, 2073, 2404, 2500, 2835,
    2907, 3142, 3147, 3182, 3240, 3325,
    3326, 3362, 3384, 3487, 3488, 3651
\int_max:nn . . . . . 1622, 1623,
    1630, 1631, 1929, 2097, 3454, 3456
\int_new:N . . . . .
    68, 69, 70, 80, 491, 492, 499, 500,
    517, 1862, 1864, 1865, 1866, 1869,
    1892, 1893, 2277, 2278, 2279, 2280,
    2281, 2282, 2283, 2285, 2286, 2287,
    2288, 2291, 2292, 2293, 2554, 3109,
    3112, 3113, 3114, 3120, 3121, 3998
\int_set:Nn . . . . . 5,
    1325, 1863, 1925, 1927, 1933, 1971,
    1973, 2042, 2093, 2094, 2104, 2115,
    2139, 2157, 2206, 2338, 2340, 2343,
    2364, 2408, 2409, 2450, 2485, 3187,
    3259, 3261, 3400, 3430, 3453, 3455
\int_set_eq:NN . . . . .
    . . . . . 141, 610, 614, 623, 625,
    668, 735, 1033, 1133, 1146, 1245,
    1883, 1903, 1920, 1948, 1982, 1983,
    2033, 2136, 2137, 2189, 2238, 2316,
    2339, 2344, 2358, 2362, 2366, 2405,
    2415, 2546, 2547, 3138, 3355, 3647
\int_step_function:nnN . . . 2416, 3255
\int_step_function:nnnN . . 3433, 3441
\int_step_inline:nnn . . . . 2331, 4001
\int_sub:Nn . 145, 846, 2192, 2200, 2209
\int_to_Hex:n . . . . . 358
\int_use:N . . . . .
    642, 729, 807, 818, 827, 831, 842,
    843, 849, 850, 856, 857, 1016, 1845,
    1938, 1943, 1964, 1966, 2071, 2084,
    2085, 2486, 2538, 2637, 2648, 2803,
    3187, 3271, 3272, 3461, 3462, 3995
\int_value:w . . . . . 351,
    875, 881, 913, 915, 924, 925, 1040,
    1525, 1540, 2571, 2572, 2583, 3296
\int_zero:N . . . . . 622,
    844, 1282, 1809, 1882, 1913, 2337,
    2616, 3132, 3133, 3181, 3368, 3642
\c_max_char_int . . . . . 355
\c_max_int . . . . . 104
\c_one_int . . . . . 2127, 2138
\l_tmpa_int . . . . . 10
\c_zero_int . . . . .
    . . 44, 640, 1140, 1536, 2027, 2053,
    2114, 2154, 2205, 2635, 2774, 2845,
    2854, 2906, 2923, 2946, 3300, 3308

intarray commands:
  \intarray_new:Nn . . . . . 2294,
    2295, 3115, 3116, 3117, 3118, 3119

iow commands:
  \iow_char:N . . . . . 329,
    330, 337, 339, 341, 343, 345, 347,
    990, 991, 1725, 1732, 1733, 1734,
    1858, 3685, 3688, 3689, 3694, 3728,
    3737, 3741, 3746, 3766, 3768, 3769,
    3771, 3774, 3776, 3781, 3783, 3785,
    3790, 3794, 3797, 3798, 3801, 3803,
    3807, 3809, 3815, 3817, 3821, 3823,
    3827, 3832, 3834, 3876, 3878, 3883,
    3885, 3891, 3896, 3901, 3905, 3915,
    3918, 3922, 3923, 3927, 3935, 4006
  \iow_indent:n . . . . . 3962
  \iow_newline: . . . . . 1780
  \iow_term:n . . . . . 3996

K

kernel internal commands:
  \__kernel_chk_tl_type:NnnTF . . . 2999

```

```

\__kernel_intarray_gset:Nnn ...
    2334, 2441, 2444, 3144,
    3216, 3218, 3224, 3232, 3234, 3237,
    3358, 3360, 3364, 3366, 3378, 3381
\__kernel_intarray_gset_range_-
    from_clist:Nnn ..... 2504
\__kernel_intarray_item:Nn ....
    49, 2452, 2479,
    2563, 2564, 2588, 2589, 2597, 2604,
    2661, 2665, 2684, 3221, 3411, 3630
\__kernel_intarray_range_to_-
    clist:Nnn ..... 2433
\__kernel_quark_new_conditional:Nn
    ..... 92
\__kernel_str_to_other_fast:n ..
    303, 1513
\__kernel_tl_gset:Nn .....
    144, 302, 1511, 3253, 3264, 3328, 3470
\__kernel_tl_set:Nn .....
    192, 1081, 1086, 1357, 1426, 3405, 3439

M

msg commands:
\msg_error:nn ..... 602,
    635, 683, 686, 1159, 1430, 2863, 2947
\msg_error:nnnn .....
    641, 864, 1258, 1271, 1310,
    1343, 1457, 2636, 2643, 2855, 2961
\msg_error:nnnnn .....
    841, 906, 1121, 2867, 2883
\msg_error:nnnnnn ..... 3269, 3460
\msg_error:nnnnnnn .. 3067, 3090, 3164
\msg_expandable_error:nn ..... 332
\msg_expandable_error:nnn 427, 1448
\msg_expandable_error:nnnn 357, 2766
\msg_log:nnnnnn ..... 2986, 2996
\g_msg_module_name_prop ..... 3970
\g_msg_module_type_prop ..... 3971
\msg_new:nnn .....
    3684, 3686, 3691, 3952, 3964
\msg_new:nnnn 3697, 3704, 3710, 3720,
    3726, 3750, 3757, 3765, 3773, 3780,
    3787, 3793, 3800, 3806, 3814, 3820,
    3826, 3836, 3843, 3852, 3855, 3863,
    3869, 3875, 3882, 3889, 3899, 3910,
    3920, 3930, 3939, 3945, 3954, 3957
\msg_show:nnnnnn ..... 2985, 2995
\msg_warning:nn ..... 1149
\msg_warning:nnnn .....
    1065, 1069, 1111, 1173, 1211, 1230
\msg_warning:nnnnn ..... 771, 920

O

or commands:
\or: ..... 164, 165, 166,
    167, 438, 439, 440, 441, 442, 2011,
    2064, 2698, 2700, 2732, 2733, 2734,
    2735, 2736, 2737, 2738, 2739, 2740,
    2741, 2742, 2743, 2744, 3147, 3148

P

peek commands:
\peek_analysis_map_break:n ... 3560
\peek_analysis_map_inline:n 152, 3553
\peek_regex:NTF .....
    3497, 3506, 3512, 3513, 3514
\peek_regex:nTF ..... 89, 150,
    153, 154, 3497, 3497, 3503, 3504, 3505
\peek_regex_remove_once:NTF ....
    3497, 3525, 3531, 3532, 3533, 3534
\peek_regex_remove_once:nTF ....
    153, 3497, 3515, 3521, 3522, 3523, 3524
\peek_regex_replace_once:Nn ....
    3589, 3603
\peek_regex_replace_once:nn ....
    3589, 3595
\peek_regex_replace_once:NnTF ..
    3589,
    3597, 3599, 3600, 3601, 3602, 3604
\peek_regex_replace_once:nnTF ..
    117, 122, 150, 156, 3589,
    3589, 3591, 3592, 3593, 3594, 3596

prg commands:
\prg_break: .....
    115, 61, 329, 333, 1591, 1601, 1606,
    1615, 1639, 1684, 2551, 2579, 3376

```

`\prg_break:n` 2087
`\prg_break_point:` . 57, 307, 1587,
 1636, 2088, 2421, 2573, 3370, 3376
`\prg_break_point:Nn`
 25, 2312, 2326, 2372, 3559
`\prg_do_nothing:`
 . 50, 119, 145, 155, 637, 680, 681,
 688, 689, 2634, 2862, 3286, 3290, 3342
`\prg_generate_conditional_-`
 variant:Nnn 3013, 3019, 3049, 3051
`\prg_map_break:Nn` 52
`\prg_new_conditional:Npnn`
 430, 450, 472, 518, 542
`\prg_new_protected_conditional:Npnn`
 885, 3008, 3014, 3044, 3046
`\prg_replicate:nn`
 149, 14, 643, 1398,
 2018, 2044, 2190, 2198, 2361, 2527,
 2639, 3301, 3309, 3356, 3472, 3474
`\prg_return_false:` 138, 444, 455,
 458, 463, 467, 468, 476, 479, 484,
 487, 524, 527, 548, 551, 892, 897, 3127
`\prg_return_true:` 133,
 138, 433, 447, 455, 458, 463, 467,
 479, 484, 487, 522, 546, 888, 894, 3125

prop commands:

`\prop_gput:Nnn` 3970, 3971

Q

quark commands:

`\quark_if_recursion_tail_stop:n`
 1580, 1700
`\quark_new:N` 85, 86
`\q_recursion_stop` 1576, 1695
`\q_recursion_tail` 1576, 1694

quark internal commands:

`\q_regex_nil` 56, 61, 86, 91,
 698, 702, 1360, 1378, 1379, 1474, 1484
`\q_regex_recursion_stop`
 85, 88, 90, 1360, 1379, 3371

R

regex commands:

`\regex_const:Nn` 13, 2970, 2980
`\regex_count:NnN` 14, 3020, 3022, 3025
`\regex_count:nnN`
 14, 137, 3020, 3020, 3024
`\regex_extract_all:NnN` 15, 3040, 3057
`\regex_extract_all:nnN`
 5, 15, 23, 3040, 3057
`\regex_extract_all:NnNTF` .. 15, 3040
`\regex_extract_all:nnNTF` .. 15, 3040
`\regex_extract_once:NnN` 15, 3040, 3055
`\regex_extract_once:nnN` 15, 3040, 3055
`\regex_extract_once:NnNTF` . 15, 3040
`\regex_extract_once:nnNTF` 8, 15, 3040
`\regex_gset:Nn` 13, 2970, 2975
`\regex_log:N` 13, 78, 2985, 2996
`\regex_log:n` 13, 2985, 2986
`\regex_match:Nn` 3014, 3019
`\regex_match:nn` 3008, 3013
`\regex_match:NnTF` 13, 3008
`\regex_match:nnTF` 13, 14, 139, 152, 3008
`\regex_match_case:nn`
 14, 17, 51, 89, 3026, 3034, 3165
`\regex_match_case:nnTF` . 14, 3026,
 3026, 3035, 3036, 3037, 3038, 3039
`\regex_new:N` 12,
 27, 2964, 2964, 2966, 2967, 2968, 2969
`\regex_replace_all:NnN` 16, 3040, 3061
`\regex_replace_all:nnN`
 5, 16, 136, 3040, 3061
`\regex_replace_all:NnNTF` .. 16, 3040
`\regex_replace_all:nnNTF` .. 16, 3040
`\regex_replace_case_all:nN`
 18, 3086, 3091, 3103
`\regex_replace_case_all:nNTF` ...
 18, 3086,
 3086, 3104, 3105, 3106, 3107, 3108
`\regex_replace_case_once:nN`
 17, 3063, 3068, 3080
`\regex_replace_case_once:nNTF` ..
 17, 3063,

[3063](#), [3081](#), [3082](#), [3083](#), [3084](#), [3085](#)
[\regex_replace_once:NnN](#) [16](#), [3040](#), [3059](#)
[\regex_replace_once:nnN](#)
[15–17](#), [135](#), [3040](#), [3059](#)
[\regex_replace_once:NnNTF](#) . [16](#), [3040](#)
[\regex_replace_once:nnNTF](#)
[16](#), [156](#), [3040](#)
[\regex_set:Nn](#) [4](#), [13](#), [2970](#), [2970](#)
[\regex_show:N](#) .. [13](#), [63](#), [78](#), [2985](#), [2995](#)
[\regex_show:n](#) ... [4](#), [10](#), [13](#), [2985](#), [2985](#)
[\regex_split:NnN](#) [16](#), [3040](#), [3062](#)
[\regex_split:nnN](#) [16](#), [3040](#), [3062](#)
[\regex_split:NnNTF](#) [16](#), [3040](#)
[\regex_split:nnNTF](#) [16](#), [3040](#)
[\g_tmpa_regex](#) [18](#), [2966](#)
[\l_tmpa_regex](#) [18](#), [2966](#)
[\g_tmpb_regex](#) [18](#), [2966](#)
[\l_tmpb_regex](#) [18](#), [2966](#)
regex internal commands:
[__regex_A_test:](#) [42](#), [968](#),
[990](#), [1606](#), [1609](#), [1615](#), [1733](#), [2217](#), [2250](#)
[__regex_action_cost:n](#) [88](#),
[93](#), [2006](#), [2007](#), [2015](#), [2465](#), [2491](#), [2491](#)
[__regex_action_free:n](#) .. [88](#), [104](#),
[2029](#), [2035](#), [2036](#), [2047](#), [2105](#), [2109](#),
[2134](#), [2159](#), [2163](#), [2166](#), [2194](#), [2202](#),
[2212](#), [2226](#), [2269](#), [2463](#), [2467](#), [2467](#)
[__regex_action_free_aux:nn](#)
[2467](#), [2468](#), [2470](#), [2471](#)
[__regex_action_free_group:n](#) ...
[88](#), [104](#), [2055](#), [2174](#), [2177](#), [2467](#), [2469](#)
[__regex_action_start_wildcard:N](#)
[88](#), [1887](#), [1907](#), [2460](#), [2460](#)
[__regex_action_submatch:nN](#)
[88](#), [1911](#), [1936](#),
[2128](#), [2129](#), [2267](#), [2516](#), [2518](#), [2518](#)
[__regex_action_submatch_aux:w](#) .
[2518](#), [2520](#), [2523](#)
[__regex_action_submatch_auxii:w](#)
[2518](#), [2529](#), [2534](#)
[__regex_action_submatch_](#)
[auxiii:w](#) [2518](#), [2530](#), [2535](#), [2536](#), [2537](#)
[__regex_action_submatch_auxiv:w](#)
[2518](#)
[__regex_action_success:](#)
[88](#), [1890](#), [1939](#), [1957](#), [2539](#), [2539](#)
[__regex_action_wildcard:](#) [110](#)
[\l__regex_added_begin_int](#)
[3120](#), [3259](#), [3267](#), [3271](#),
[3325](#), [3453](#), [3458](#), [3461](#), [3472](#), [3487](#)
[\l__regex_added_end_int](#)
[3120](#), [3261](#), [3267](#), [3272](#),
[3326](#), [3455](#), [3458](#), [3462](#), [3474](#), [3488](#)
[\c__regex_all_catcodes_int](#)
[502](#), [624](#), [728](#), [1326](#)
[\c__regex_ascii_lower_int](#) [84](#), [146](#), [152](#)
[\c__regex_ascii_max_control_int](#)
[81](#), [263](#)
[\c__regex_ascii_max_int](#)
[81](#), [256](#), [264](#), [454](#)
[\c__regex_ascii_min_int](#) . [81](#), [255](#), [262](#)
[__regex_assertion:Nn](#) [42](#), [59](#),
[101](#), [964](#), [986](#), [1595](#), [1726](#), [2217](#), [2217](#)
[__regex_b_test:](#) [42](#),
[101](#), [976](#), [978](#), [1612](#), [1731](#), [2217](#), [2235](#)
[\l__regex_balance_int](#)
[27](#), [119](#), [149](#), [80](#),
[2616](#), [2648](#), [2907](#), [2924](#), [3132](#), [3145](#),
[3147](#), [3148](#), [3400](#), [3430](#), [3454](#), [3456](#)
[\g__regex_balance_intarray](#)
[23](#), [138](#), [2595](#), [2602](#), [3119](#), [3144](#)
[\g__regex_balance_tl](#) .. [119](#), [2557](#),
[2617](#), [2647](#), [2673](#), [2690](#), [2700](#), [2775](#)
[__regex_begin](#) [3110](#)
[__regex_branch:n](#)
[42](#), [65](#), [96](#), [77](#), [629](#),
[704](#), [1136](#), [1189](#), [1374](#), [1484](#), [1492](#),
[1576](#), [1578](#), [1581](#), [1708](#), [2100](#), [2100](#)
[__regex_break_point:TF](#) [28](#), [59](#), [93](#),
[93](#), [94](#), [95](#), [99](#), [2006](#), [2007](#), [2223](#), [2240](#)
[__regex_break_true:w](#) . [28](#), [29](#), [93](#),
[93](#), [99](#), [104](#), [111](#), [118](#), [122](#), [129](#), [135](#),
[184](#), [196](#), [212](#), [939](#), [2247](#), [2253](#), [2259](#)

```

\__regex_build:n .....
    135, 1870, 1872, 3016, 3023, 3043, 3047
\__regex_build:n ..... 90,
    134, 1870, 1870, 3010, 3021, 3042, 3045
\__regex_build_aux:NN . 151, 1870,
    1873, 1877, 1879, 3509, 3528, 3598
\__regex_build_aux:Nn .....
    151, 1870, 1871, 1874, 3500, 3518, 3590
\__regex_build_for_cs:n .....
    ..... 207, 1946, 1946
\__regex_build_new_state: .....
    ..... 1884, 1885, 1904,
    1905, 1909, 1949, 1950, 1979, 1979,
    1988, 2020, 2054, 2058, 2102, 2117,
    2122, 2161, 2180, 2215, 2219, 2264
\l__regex_build_tl .....
    ..... 65, 156, 74, 621, 628, 646,
    651, 654, 655, 658, 659, 662, 722,
    725, 765, 779, 783, 908, 922, 963,
    985, 998, 1030, 1043, 1047, 1129,
    1132, 1135, 1141, 1142, 1145, 1188,
    1478, 1482, 1489, 1495, 1516, 1532,
    1550, 1707, 1764, 1767, 1778, 1808,
    1823, 1827, 1830, 1836, 2615, 2638,
    2649, 2652, 2703, 2772, 2829, 2832,
    2846, 2914, 3657, 3660, 3668, 3671
\__regex_build_transition_-
    left:NNN 1975, 1975, 2163, 2177, 2194
\__regex_build_transition_-
    right:nNn ..... 1975,
    1977, 2021, 2055, 2105, 2109,
    2134, 2159, 2166, 2174, 2202, 2212
\__regex_build_transitions_-
    lazyness:NNNNN .....
    ..... 1986, 1986, 2028, 2034, 2046
\l__regex_capturing_group_int ..
    ..... 22, 23, 87, 148,
    1869, 1882, 1920, 1925, 1930, 2071,
    2073, 2084, 2085, 2093, 2094, 2097,
    2361, 2435, 2436, 2509, 2528, 2763,
    2767, 3356, 3377, 3382, 3435, 3443
\g__regex_case_balance_tl .....
    ..... 2678, 2681, 2687, 2691, 2699
\__regex_case_build:n .....
    139, 1894, 1894, 1899, 3074, 3097, 3171
\__regex_case_build_aux:Nn ....
    ..... 1894, 1896, 1900
\__regex_case_build_loop:n ....
    ..... 1894, 1918, 1923
\l__regex_case_changed_char_int
    30, 121, 133, 134, 141, 145, 151, 2282
\g__regex_case_int .....
    ..... 134, 135, 1892, 1897, 1914,
    1917, 1937, 1938, 3030, 3075, 3367
\l__regex_case_max_group_int ...
    ..... 1893, 1913, 1920, 1927, 1929
\__regex_case_replacement:n ....
    ..... 2677, 2679, 2695, 3098
\__regex_case_replacement_aux:n
    ..... 2689, 2696
\g__regex_case_replacement_tl ..
    ..... 2677, 2687, 2693, 2698
\c__regex_catcode_A_int ..... 502
\c__regex_catcode_B_int ..... 502
\c__regex_catcode_C_int ..... 502
\c__regex_catcode_D_int ..... 502
\c__regex_catcode_E_int ..... 502
\c__regex_catcode_in_class_mode_-
    int . 492, 613, 997, 1158, 1251, 1280
\c__regex_catcode_L_int ..... 502
\c__regex_catcode_M_int ..... 502
\c__regex_catcode_mode_int ....
    ..... 492, 609, 682, 1029, 1249, 1278
\c__regex_catcode_O_int ..... 502
\c__regex_catcode_P_int ..... 502
\c__regex_catcode_S_int ..... 502
\c__regex_catcode_T_int ..... 502
\c__regex_catcode_U_int ..... 502
\l__regex_catcodes_bool .....
    ..... 499, 1285, 1289, 1324
\l__regex_catcodes_int .....
    ..... 43, 499, 625, 727,
    729, 735, 1016, 1033, 1133, 1146,
    1245, 1282, 1317, 1319, 1325, 1326

```

<code>__regex_char_if_alphanumeric:N</code>	472	<code>__regex_clean_int_aux:N</code>
<code>__regex_char_if_alphanumeric:NTF</code>	<code>__regex_clean_regex:n</code>
.....	450, 675, 2881	1553, 1573, 1629, 1642, 3000
<code>__regex_char_if_special:N</code>	<code>__regex_clean_regex_loop:w</code>
<code>__regex_char_if_special:NTF</code>	450, 671	1553, 1575, 1578, 1582
<code>__regex_chk_c_allowed:TF</code>	<code>__regex_command_K:</code>
.....	594, 594, 1238	42, 1550, 1594, 1724, 2262, 2262
<code>__regex_class:NnnnN</code>	..	<code>__regex_compile:n</code>
.....	41, 52, 54, 61, 78, 723, 1024, 1025, 1031, 1391, 1524, 1534, 1596, 1723, 2000, 2000	664, 664, 700, 1876, 2972, 2977, 2982, 2989
<code>\c__regex_class_mode_int</code>	492, 599, 614	<code>__regex_compile:w</code>
<code>__regex_class_repeat:n</code>	50, 618, 618, 666, 1331
....	94, 2010, 2016, 2016, 2032, 2041	<code>__regex_compile_\$:</code>
<code>__regex_class_repeat:nN</code>	959
.....	2011, 2025, 2025	<code>__regex_compile(:</code>
<code>__regex_class_repeat:nnN</code>	1153
.....	2012, 2039, 2039	<code>__regex_compile_):</code>
<code>__regex_clean_assertion:Nn</code>	1192
.....	1553, 1595, 1603	<code>__regex_compile_:</code>
<code>__regex_clean_bool:n</code>	930
..	1553, 1553, 1605, 1620, 1624, 1632	<code>__regex_compile_/A:</code>
<code>__regex_clean_branch:n</code>	959
.....	1553, 1581, 1584	<code>__regex_compile_/B:</code>
<code>__regex_clean_branch_loop:n</code>	959
.....	1553, 1586, 1589, 1594, 1616, 1625, 1633	<code>__regex_compile_/b:</code>
<code>__regex_clean_class:n</code>	1237
.....	1553, 1621, 1635, 1646, 1667	<code>__regex_compile_/c:</code>
<code>__regex_clean_class:NnnnN</code>	942
.....	1553, 1596, 1618	<code>__regex_compile_/D:</code>
<code>__regex_clean_class_loop:nnn</code>	942
.....	1553, 1636, 1637, 1648, 1658, 1668, 1682	<code>__regex_compile_/d:</code>
<code>__regex_clean_exact_cs:n</code>	942
.....	1553, 1643, 1689	<code>__regex_compile_/G:</code>
<code>__regex_clean_exact_cs:w</code>	959
.....	1553, 1693, 1698, 1702	<code>__regex_compile_/H:</code>
<code>__regex_clean_group:nnnN</code>	942
.....	1553, 1597, 1598, 1599, 1627	<code>__regex_compile_/h:</code>
<code>__regex_clean_int:n</code>	942
...	1553, 1559, 1562, 1622, 1623, 1630, 1631, 1644, 1645, 1657, 1667	<code>__regex_compile_/K:</code>
		1547
		<code>__regex_compile_/N:</code>
		942
		<code>__regex_compile_/S:</code>
		942
		<code>__regex_compile_/s:</code>
		942
		<code>__regex_compile_/u:</code>
		1411
		<code>__regex_compile_/V:</code>
		942
		<code>__regex_compile_/v:</code>
		942
		<code>__regex_compile_/W:</code>
		942
		<code>__regex_compile_/w:</code>
		942
		<code>__regex_compile_/Z:</code>
		959
		<code>__regex_compile_/z:</code>
		959
		<code>__regex_compile_[:</code>
		1008
		<code>__regex_compile_]:</code>
		992
		<code>__regex_compile_^:</code>
		959
		<code>__regex_compile_abort_tokens:n</code>
		738, 738, 746, 772, 1113, 1123

<code>__regex_compile_anchor_letter:NNN</code>	<code>__regex_compile_group_end: . . .</code>
959 , 959 , 968 , 970 , 972 , 974 , 976 , 978 1127 , 1138 , 1195
<code>__regex_compile_c[:w</code> 1274	<code>__regex_compile_if_quantifier:TFw</code>
<code>__regex_compile_c_C:NN</code> 747 , 747 , 1475 , 1487
. 1253 , 1262 , 1262	<code>__regex_compile_lparen:w</code> 1162 , 1166
<code>__regex_compile_c_lbrack_add:N</code>	<code>__regex_compile_one:n .</code> 717 , 717 ,
. 1274 , 1300 , 1315	874 , 880 , 934 , 945 , 948 , 958 , 1104 , 1362
<code>__regex_compile_c_lbrack_end: .</code>	<code>__regex_compile_quantifier:w . .</code>
. 1274 , 1307 , 1311 , 1322	736 , 754 , 754 , 1003 , 1147 , 1480 , 1496
<code>__regex_compile_c_lbrack_-</code>	<code>__regex_compile_quantifier*:w</code> 788
loop:NN 1274 , 1286 , 1290 , 1294 , 1302	<code>__regex_compile_quantifier+:w</code> 788
<code>__regex_compile_c_test:NN</code>	<code>__regex_compile_quantifier?:w</code> 788
. 1237 , 1238 , 1239	<code>__regex_compile_quantifier_-</code>
<code>__regex_compile_class:NN</code>	abort:nNN 763 , 768 , 798 , 817 , 830 , 853
. 1038 , 1044 , 1048 , 1051	<code>__regex_compile_quantifier_-</code>
<code>__regex_compile_class:TFNN</code>	braced_auxi:w 794 , 797 , 800
. 61 , 1023 , 1034 , 1038 , 1038	<code>__regex_compile_quantifier_-</code>
<code>__regex_compile_class_catcode:w</code>	braced_auxii:w 794 , 813 , 822
. 1015 , 1027 , 1027	<code>__regex_compile_quantifier_-</code>
<code>__regex_compile_class_normal:w</code>	braced_auxiii:w 794 , 812 , 835
. 1018 , 1021 , 1021	<code>__regex_compile_quantifier_-</code>
<code>__regex_compile_class_posix:NNNw</code>	lazyness:nnNN 54 ,
. 1057 , 1063 , 1076	775 , 775 , 789 , 791 , 793 , 806 , 826 , 848
<code>__regex_compile_class_posix_-</code>	<code>__regex_compile_quantifier_-</code>
end:w 1057 , 1094 , 1096	none: 759 , 761 , 763 , 763 , 770
<code>__regex_compile_class_posix_-</code>	<code>__regex_compile_range:Nw</code>
loop:w . 1057 , 1082 , 1087 , 1090 , 1093 872 , 885 , 901
<code>__regex_compile_class_posix_-</code>	<code>__regex_compile_raw:N</code>
test:w 1011 , 1057 , 1057	544 , 672 , 676 , 678 , 710 , 715 , 743 ,
<code>__regex_compile_cs_aux:Nn</code>	865 , 867 , 867 , 887 , 933 , 983 , 1006 ,
. 1346 , 1359 , 1372 , 1380	1054 , 1074 , 1092 , 1150 , 1155 , 1160 ,
<code>__regex_compile_cs_aux:NNnnN .</code>	1176 , 1186 , 1194 , 1212 , 1213 , 1214 ,
. 1346 , 1377 , 1387 , 1400	1220 , 1231 , 1232 , 1233 , 1241 , 1296 ,
<code>__regex_compile_end:</code>	1344 , 1351 , 1416 , 1432 , 1433 , 1439
. 50 , 618 , 631 , 691 , 1355	<code>__regex_compile_raw_error:N . . .</code>
<code>__regex_compile_end_cs:</code> 862 , 862 , 961 , 1414 , 1551
. 687 , 1346 , 1350 , 1353	<code>__regex_compile_special:N</code>
<code>__regex_compile_escaped:N</code> 44 , 672 , 707 , 707 , 749 , 756 ,
. 676 , 707 , 712	777 , 804 , 809 , 824 , 837 , 871 , 890 ,
<code>__regex_compile_group_begin:N .</code>	1041 , 1059 , 1078 , 1098 , 1099 , 1168 ,
. . 1127 , 1127 , 1175 , 1180 , 1198 , 1200	1203 , 1221 , 1264 , 1283 , 1423 , 1442

__regex_compile_special_group_-	\l__regex_curr_catcode_int
-:w 163, 182, 190, 202, 2282 , 2409
__regex_compile_special_group_-	\l__regex_curr_char_int
::w	109, 103,
__regex_compile_special_group_-	109, 110, 117, 127, 128, 141, 142,
i:w	143, 144, 150, 183, 938, 1956, 2238,
__regex_compile_special_group_-	2246, 2282 , 2366, 2405, 2408, 2424
l:w	__regex_curr_cs_to_str:
__regex_compile_u_brace:NNN 35 , 35, 193, 210
.....	\l__regex_curr_pos_int
1417, 1418, 1421 , 1421	. 25, 109, 2258, 2277 , 2353, 2364,
__regex_compile_u_end:	2404, 2538, 2546, 3133, 3138, 3142,
.....	3143, 3145, 3642, 3647, 3651, 3652
1418, 1485 , 1485	\l__regex_curr_state_int
__regex_compile_u_in_cs:	... 105,
.....	113, 2288 , 2442, 2443, 2445, 2450,
1506, 1509 , 1509	2453, 2475, 2480, 2485, 2486, 2494
__regex_compile_u_in_cs_aux:n	\l__regex_curr_submatches_tl
..... 2289 , 2360, 2455,
1519, 1522	2487, 2488, 2499, 2521, 2525, 2550
__regex_compile_u_loop:NN	\l__regex_curr_token_tl
.....	38, 2282 , 2407
1427, 1437 , 1437, 1440, 1452	\l__regex_default_catcodes_int
__regex_compile_u_not_cs:	43, 499 , 623, 625, 735, 1033, 1133, 1146
.....	__regex_disable_submatches:
1504, 1528 , 1528	206, 1332, 2513 , 2513, 3154, 3180, 3539
__regex_compile_u_payload:	\l__regex_empty_success_bool
..... 2299 , 2345, 2349, 2544, 3241
76, 1485 , 1494, 1498, 1500	__regex_end
__regex_compile_ur:n 3110
.....	__regex_escape_␣:w
75, 1463 , 1470, 1472 329
__regex_compile_ur_aux:w	__regex_escape_/\scan_stop:w
..... 329
1463, 1474, 1484	__regex_escape_/a:w
__regex_compile_ur_end: 329
.....	__regex_escape_/e:w
1417, 1431, 1463 , 1463 329
__regex_compile_use:n	__regex_escape_/f:w
693, 693, 1926 329
__regex_compile_use_aux:w	__regex_escape_/n:w
697, 702 329
__regex_compile_ :	__regex_escape_/r:w
..... 1184 329
__regex_compute_case_changed_-	__regex_escape_/t:w
char: 329
.....	__regex_escape_/x:w
139, 139, 157, 2406 348
__regex_count:nnN	__regex_escape_\\:w
..... 313
3021, 3023, 3177 , 3177	__regex_escape_\\scan_stop:w
__regex_cs 329
..... 1346	__regex_escape_escaped:N
\c__regex_cs_in_class_mode_int
.....	299, 323, 326 , 327
492, 1337	__regex_escape_loop:N
\c__regex_cs_mode_int
..... 492 , 1335 36, 306, 313 , 313, 317,
\l__regex_curr_analysis_tl	320, 324, 348, 387, 398, 399, 419, 428
.....	
106, 2296 , 2342, 2370, 2377, 2411, 2412	

<code>__regex_escape_raw:N</code> 37 , 300 , 326 , 328 , 337 , 339 , 341 , 343 , 345 , 347 , 361	<code>__regex_extract_seq_aux:n</code> 3256 , 3292 , 3292
<code>__regex_escape_unescaped:N</code> 298 , 316 , 326 , 326	<code>__regex_extract_seq_aux:ww</code> 3292 , 3295 , 3298
<code>__regex_escape_use:nnnn</code> 35 , 50 , 294 , 294 , 669 , 2618	<code>__regex_extract_seq_loop:Nw</code> 3248 , 3280 , 3287 , 3290
<code>__regex_escape_x:N</code> 37 , 386 , 390 , 390	<code>\l__regex_fresh_thread_bool</code> 107 , 112 , 2268 , 2274 , 2299 , 2422 , 2462 , 2464 , 2545
<code>__regex_escape_x_end:w</code> 37 , 348 , 350 , 353	<code>__regex_G_test:</code> 42 , 970 , 1610 , 1734 , 2217 , 2256
<code>__regex_escape_x_large:n</code> 348	<code>__regex_get_digits:NtFw</code> 530 , 530 , 796 , 811
<code>__regex_escape_x_loop:N</code> 37 , 383 , 402 , 402 , 411 , 414	<code>__regex_get_digits_loop:nw</code> 533 , 536 , 539
<code>__regex_escape_x_loop_error:</code> 402	<code>__regex_get_digits_loop:w</code> 530
<code>__regex_escape_x_loop_error:n</code> 408 , 420 , 425	<code>__regex_group:nnnN</code> 41 , 65 , 1175 , 1180 , 1466 , 1597 , 1717 , 1888 , 2068 , 2068
<code>__regex_escape_x_test:N</code> 37 , 351 , 365 , 365 , 373	<code>__regex_group_aux:nnnnN</code> 96 , 2051 , 2051 , 2070 , 2078 , 2081
<code>__regex_escape_x_testii:N</code> 365 , 375 , 380	<code>__regex_group_aux:nnnnnN</code> 96
<code>\l__regex_every_match_tl</code> 2298 , 2381 , 2391 , 2428	<code>__regex_group_end_extract_seq:N</code> 144 , 3197 , 3206 , 3246 , 3248 , 3248
<code>__regex_extract:</code> 140 , 155 , 3195 , 3202 , 3215 , 3352 , 3352 , 3397 , 3425 , 3614	<code>__regex_group_end_replace:N</code> 3416 , 3449 , 3451 , 3451
<code>__regex_extract_all:nnN</code> 3056 , 3189 , 3199	<code>__regex_group_end_replace_- check:n</code> 149 , 3451 , 3480 , 3483
<code>__regex_extract_aux:w</code> 3352 , 3369 , 3374 , 3385	<code>__regex_group_end_replace_- check:w</code> 149 , 3451 , 3469 , 3478
<code>__regex_extract_check:n</code> 3316 , 3318 , 3321	<code>__regex_group_end_replace_try:</code> 149 , 3451 , 3457 , 3467 , 3489
<code>__regex_extract_check:w</code> 142 , 144 , 3263 , 3316 , 3316 , 3327	<code>\l__regex_group_level_int</code> 491 , 622 , 640 , 642 , 644 , 1134 , 1140
<code>__regex_extract_check_end:w</code> 145 , 3316 , 3332 , 3344	<code>__regex_group_no_capture:nnnN</code> 41 , 1198 , 1466 , 1467 , 1479 , 1491 , 1598 , 1719 , 2068 , 2077
<code>__regex_extract_check_loop:w</code> 3316 , 3330 , 3337 , 3342 , 3345	<code>__regex_group_repeat:nn</code> 2063 , 2112 , 2112
<code>__regex_extract_once:nnN</code> 3054 , 3189 , 3189	<code>__regex_group_repeat:nnN</code> 2064 , 2152 , 2152
<code>__regex_extract_seq:N</code> 3248 , 3275 , 3277	
<code>__regex_extract_seq:NNn</code> 3248 , 3281 , 3285	

```

\\__regex_group_repeat:nnnN . . . .
    . . . . . 2065, 2183, 2183
\\__regex_group_repeat_aux:n . . . .
    . 98, 99, 2119, 2132, 2132, 2170, 2187
\\__regex_group_resetting:nnnN . .
    41, 1200, 1467, 1599, 1721, 2079, 2079
\\__regex_group_resetting_
    loop:nnNn . . 2079, 2083, 2091, 2096
\\__regex_group_submatches:nNN . .
    . . 2120, 2125, 2125, 2155, 2171, 2185
\\__regex_hexadecimal_use:N . . . . 430
\\__regex_hexadecimal_use:NNTF . . .
    . . . . . 385, 397, 410, 430
\\__regex_if_end_range:NN . . . . . 885
\\__regex_if_end_range:NNTF 885, 903
\\__regex_if_in_class:TF . . . . .
    554, 554, 633, 720, 736, 869, 932,
    994, 1010, 1155, 1186, 1194, 3731, 3744
\\__regex_if_in_class_or_catcode:TF
    . . . . . 574, 574, 961, 983, 1413
\\__regex_if_in_cs:TF . . . . .
    . . . 562, 562, 1342, 1349, 3729, 3738
\\__regex_if_match:nn . . . . .
    . . . . . 3010, 3016, 3151, 3151, 3170
\\__regex_if_raw_digit:NN . . . . . 542
\\__regex_if_raw_digit:NNTF . . . .
    . . . . . 532, 538, 542
\\__regex_if_two_empty_matches:TF
    . . . 107, 2299, 2301, 2350, 2356, 2541
\\__regex_if_within_catcode:TF . .
    . . . . . 586, 586, 1013
\\__regex_input_item:n . . . . .
    . . . . . 150, 156, 157, 3495,
    3496, 3556, 3578, 3619, 3643, 3652
\\l__regex_input_tl . . . . .
    . . . . . 152, 153, 156, 3495,
    3551, 3555, 3577, 3579, 3641, 3645
\\__regex_int_eval:w . . . . . 3,
    3, 2507, 2571, 2572, 2583, 3379, 3382
\\__regex_intarray_item:NnTF . . . .
    . . . . . 40, 40, 2595, 2602

\\__regex_intarray_item_aux:nNTF
    . . . . . 40, 41, 42
\\l__regex_internal_a_int . . . . .
    . . 54, 124, 66, 796, 807, 818, 827,
    831, 839, 842, 846, 849, 856, 2033,
    2036, 2042, 2047, 2121, 2136, 2142,
    2148, 2157, 2160, 2164, 2167, 2172,
    2175, 2178, 2193, 2201, 2210, 2782,
    2803, 3368, 3377, 3379, 3382, 3384
\\l__regex_internal_a_tl . . . . .
    . . . . . 35, 76, 77, 82, 149, 66,
    192, 195, 297, 304, 311, 1081, 1086,
    1102, 1107, 1112, 1116, 1122, 1123,
    1357, 1368, 1426, 1470, 1502, 1514,
    1530, 1711, 1714, 1767, 1788, 1830,
    1837, 1932, 1933, 1970, 1971, 1972,
    1973, 2103, 2104, 2108, 2110, 2367,
    2370, 2993, 3005, 3405, 3439, 3473
\\l__regex_internal_b_int . . . . .
    . 66, 811, 840, 843, 844, 846, 850,
    857, 2137, 2142, 2147, 2193, 2201, 2210
\\l__regex_internal_b_tl . . . . .
    . . . . . 66, 1425, 1445, 1458
\\l__regex_internal_bool . . . . .
    . . . . . 66, 1080, 1085, 1106, 1115
\\l__regex_internal_c_int . . . . .
    . . . . . 66, 2139, 2144, 2145, 2149
\\l__regex_internal_regex . . . . .
    . . . . . 49, 515, 662, 700, 1359,
    1365, 1877, 2973, 2978, 2983, 2990
\\l__regex_internal_seq . 66, 1843,
    1844, 1849, 1856, 1857, 1858, 1860
\\g__regex_internal_tl 142, 144, 66,
    302, 306, 1511, 1518, 3253, 3264,
    3265, 3283, 3328, 3331, 3465, 3470
\\__regex_item_caseful_equal:n . .
    . . . . 42, 101, 101, 223, 224, 228,
    229, 230, 231, 232, 241, 246, 264,
    282, 626, 1225, 1393, 1525, 1644, 1735
\\__regex_item_caseful_range:nn .
    . . . . . 42, 101, 107, 220,
    235, 238, 239, 240, 254, 261, 268,

```

270, 272, 275, 276, 277, 278, 283,
 286, 291, 292, 627, 1227, 1652, 1737
 _regex_item_caseless_equal:n .
 42, 115, 115, 1206, 1645, 1742
 _regex_item_caseless_range:nn
 42, 115, 125, 1208, 1653, 1744
 _regex_item_catcode: . 160, 160, 172
 _regex_item_catcode:nTF ... 42,
 62, 160, 170, 179, 729, 1035, 1663, 1749
 _regex_item_catcode_reverse:nTF
 42, 160, 178, 1036, 1664, 1751
 _regex_item_cs:n
 42, 200, 200, 1365, 1642, 1758
 _regex_item_equal:n .. 158, 158,
 626, 875, 881, 911, 924, 925, 1205, 1224
 _regex_item_exact:nn
 ... 42, 77, 180, 180, 1540, 1654, 1755
 _regex_item_exact_cs:n 42,
 72, 180, 188, 1367, 1537, 1643, 1757
 _regex_item_range:nn
 158, 159, 627, 913, 1207, 1226
 _regex_item_reverse:n
 42, 63, 96, 96,
 179, 245, 949, 1106, 1646, 1753, 2241
 \l_regex_last_char_int
 2238, 2252, 2282, 2405, 2547
 \l_regex_last_char_success_int
 2282, 2340, 2366, 2547
 \l_regex_left_state_int .. 1865,
 1886, 1906, 1910, 1964, 1971, 1982,
 1989, 1992, 1993, 1995, 1996, 2022,
 2030, 2033, 2056, 2104, 2106, 2116,
 2136, 2156, 2158, 2186, 2189, 2192,
 2195, 2207, 2220, 2229, 2265, 2272
 \l_regex_left_state_seq
 1865, 1963, 1970, 2103
 _regex_maplike_break:
 25, 152, 51, 51,
 52, 2312, 2326, 2372, 2386, 2394, 3559
 _regex_match:n 2305, 2305, 3157,
 3184, 3194, 3204, 3230, 3394, 3427
 _regex_match_case:nnTF
 3028, 3160, 3160
 _regex_match_case_aux:nn 3160, 3176
 \l_regex_match_count_int
 137, 140, 3109, 3181, 3182, 3187
 _regex_match_cs:n .. 210, 2305, 2314
 _regex_match_init:
 2305, 2307, 2317, 2328, 3550
 _regex_match_once_init:
 .. 2308, 2318, 2347, 2347, 2398, 3552
 _regex_match_once_init_aux: ..
 2368, 2374
 _regex_match_one_active:n
 2401, 2419, 2430
 _regex_match_one_token:nnN ...
 . 109, 112, 152, 2310, 2311, 2322,
 2323, 2325, 2371, 2401, 2401, 3557
 \l_regex_match_success_bool ...
 ... 107, 2302, 2359, 2385, 2393, 2543
 \l_regex_matched_analysis_tl ...
 106, 2296, 2341, 2367, 2376, 2410, 2548
 \l_regex_max_pos_int
 118, 2277, 3138,
 3236, 3242, 3414, 3447, 3633, 3647
 \l_regex_max_state_int
 ... 87, 91, 166, 1862, 1883, 1903,
 1941, 1943, 1944, 1948, 1981, 1983,
 1984, 2043, 2115, 2135, 2137, 2145,
 2189, 2195, 2203, 2213, 2332, 4003
 \l_regex_max_thread_int
 2292, 2316,
 2362, 2415, 2418, 2423, 2500, 2508
 _regex_maybe_compute_ccc:
 120, 132, 155, 157, 2406
 \l_regex_min_pos_int
 118, 2277, 2338, 2339
 \l_regex_min_state_int 91, 1862,
 1883, 1903, 1948, 2332, 2363, 4002
 \l_regex_min_submatch_int
 137, 142,
 148, 2343, 2344, 3112, 3255, 3434, 3442

<code>\l__regex_min_thread_int</code>	<code>\l__regex_peek_true_tl</code>
.. 2292 , 2316 , 2362 , 2415 , 2417 , 2423	153 , 155 , 3492 , 3546 , 3566 , 3572 , 3625
<code>\l__regex_mode_int</code>	<code>__regex_pop_lr_states:</code>
..... 492 , 556 , 564 , 567 , 576 , 1921 , 1953 , 1961 , 1968 , 2061
579 , 588 , 596 , 599 , 609 , 610 , 612 ,	<code>__regex_posix_alnum:</code> 248 , 248
614 , 668 , 682 , 684 , 996 , 1000 , 1001 ,	<code>__regex_posix_alpha:</code> 80 , 248 , 249 , 250
1002 , 1029 , 1040 , 1157 , 1247 , 1248 ,	<code>__regex_posix_ascii:</code> 248 , 252
1276 , 1277 , 1333 , 1334 , 1503 , 1549	<code>__regex_posix_blank:</code> 248 , 258
<code>__regex_mode_quit_c:</code>	<code>__regex_posix_cntrl:</code> 248 , 259
..... 607 , 607 , 719 , 1130	<code>__regex_posix_digit:</code>
<code>__regex_msg_repeated:nnN</code> 248 , 249 , 266 , 290
..... 1803 , 1824 , 1834 , 3972 , 3972	<code>__regex_posix_graph:</code> 248 , 267
<code>__regex_multi_match:n</code>	<code>__regex_posix_lower:</code> .. 248 , 251 , 269
106 , 2379 , 2389 , 3182 , 3202 , 3211 , 3425	<code>__regex_posix_print:</code> 248 , 271
<code>\c__regex_no_match_regex</code> 75 , 515 , 2965	<code>__regex_posix_punct:</code> 248 , 273
<code>\c__regex_outer_mode_int</code> 492 , 567 ,	<code>__regex_posix_space:</code> 248 , 280
579 , 588 , 596 , 610 , 668 , 684 , 1503 , 1549	<code>__regex_posix_upper:</code> .. 248 , 251 , 285
<code>__regex_peek:nnTF</code>	<code>__regex_posix_word:</code> 248 , 287
154 , 3499 , 3508 , 3517 , 3527 , 3535 , 3535	<code>__regex_posix_xdigit:</code> 248 , 288
<code>__regex_peek_aux:nnTF</code>	<code>__regex_prop_:</code>
..... 3535 , 3537 , 3543 , 3608 58 , 930
<code>__regex_peek_end:</code>	<code>__regex_prop_d:</code> 59 , 80 , 219 , 219 , 266
.... 151 , 153 , 3501 , 3510 , 3563 , 3563	<code>__regex_prop_h:</code> 219 , 221 , 258
<code>\l__regex_peek_false_tl</code>	<code>__regex_prop_N:</code> 219 , 243 , 958
..... 3492 , 3547 , 3567 , 3573 , 3637	<code>__regex_prop_s:</code>
<code>__regex_peek_reinsert:N</code> ... 153 ,	<code>__regex_prop_v:</code>
155 , 3566 , 3567 , 3573 , 3575 , 3575 , 3637	<code>__regex_prop_w:</code>
<code>__regex_peek_remove_end:n</code> 219 , 236 , 287 , 2239 , 2241 , 2242
.... 151 , 153 , 3519 , 3529 , 3563 , 3569	<code>__regex_push_lr_states:</code>
<code>__regex_peek_replace:nnTF</code> 1912 , 1951 , 1961 , 1961 , 2059
..... 3590 , 3598 , 3605 , 3605	<code>__regex_quark_if_nil:N</code>
<code>__regex_peek_replace_end:</code> 92
..... 3608 , 3610 , 3610	<code>__regex_quark_if_nil:nTF</code>
<code>__regex_peek_replacement_put:n</code> 92
..... 3616 , 3654 , 3654	<code>__regex_query_range:nn</code>
<code>__regex_peek_replacement_put_-</code> 118 , 155 , 2562 , 2568 ,
<code>submatch_aux:n</code> ... 3618 , 3665 , 3665	2568 , 2587 , 2659 , 3409 , 3446 , 3628
<code>__regex_peek_replacement_-</code>	<code>__regex_query_range_loop:ww</code> ...
<code>token:n</code> 157 , 3620 , 3663 , 3663 2568 , 2570 , 2575 , 2582
<code>__regex_peek_replacement_var:N</code>	<code>__regex_query_set:n</code>
..... 3621 , 3675 , 3675 3130 ,
	3130 , 3196 , 3205 , 3231 , 3398 , 3428
	<code>__regex_query_set_aux:nN</code>
 3130 , 3134 , 3136 , 3137 , 3140

```

\\__regex_query_set_from_input_-
    tl: ..... 3615, 3639, 3639
\\__regex_query_set_item:n .....
    ..... 3639, 3643, 3644, 3646, 3649
\\__regex_query_submatch:n .....
    .. 2585, 2585, 2773, 3307, 3669, 3672
\\__regex_reinsert_item:n .....
    153, 155, 3575, 3578, 3581, 3619, 3658
\\__regex_replace_all:nnN .....
    ..... 3060, 3420, 3420
\\__regex_replace_all_aux:nnN ...
    ..... 3096, 3421, 3422
\\__regex_replace_once:nnN .....
    ..... 3058, 3387, 3387
\\__regex_replace_once_aux:nnN ..
    ..... 3073, 3387, 3388, 3389
\\__regex_replacement:n 155, 2610,
    2610, 2654, 3075, 3388, 3421, 3622
\\__regex_replacement_apply:Nn ..
    ..... 2610, 2611, 2612, 2689
\\__regex_replacement_balance_-
    one_match:n ..... 116,
    117, 2558, 2558, 2671, 3402, 3437
\\__regex_replacement_c:w . 2812, 2812
\\__regex_replacement_c_A:w ....
    ..... 122, 2744, 2900, 2901
\\__regex_replacement_c_B:w ....
    ..... 2732, 2903, 2904
\\__regex_replacement_c_C:w 2912, 2912
\\__regex_replacement_c_D:w ....
    ..... 2739, 2917, 2918
\\__regex_replacement_c_E:w ....
    ..... 2733, 2920, 2921
\\__regex_replacement_c_L:w ....
    ..... 2742, 2929, 2930
\\__regex_replacement_c_M:w ....
    ..... 2734, 2932, 2933
\\__regex_replacement_c_O:w 2731,
    2736, 2740, 2743, 2745, 2935, 2936
\\__regex_replacement_c_P:w ....
    ..... 2737, 2938, 2939
\\__regex_replacement_c_S:w ....
    ..... 2727, 2741, 2944, 2944
\\__regex_replacement_c_T:w ....
    ..... 2735, 2952, 2953
\\__regex_replacement_c_U:w ....
    ..... 2738, 2955, 2956
\\__regex_replacement_cat:NNN ...
    ..... 2817, 2860, 2860
\\l__regex_replacement_category_-
    seq 2555, 2641, 2644, 2645, 2714, 2874
\\l__regex_replacement_category_-
    tl ..... 122,
    2555, 2709, 2715, 2718, 2875, 2876
\\__regex_replacement_char:nnN ..
    ..... 131,
    2895, 2895, 2902, 2909, 2919, 2926,
    2931, 2934, 2937, 2941, 2954, 2957
\\l__regex_replacement_csnames_-
    int 116, 2554, 2635, 2637, 2639,
    2706, 2774, 2828, 2835, 2845, 2847,
    2854, 2865, 2906, 2923, 3656, 3667
\\__regex_replacement_cu_aux:Nw .
    ..... 2822, 2826, 2826, 2840
\\__regex_replacement_do_one_-
    match:n ..... 155,
    156, 2560, 2560, 2657, 3407, 3445, 3626
\\__regex_replacement_error:NNN .
    ..... 2783, 2795,
    2806, 2818, 2823, 2841, 2959, 2959
\\__regex_replacement_escaped:N .
    ..... 2631, 2750, 2750, 2879
\\__regex_replacement_exp_not:N .
    ... 126, 2566, 2566, 2822, 2915, 3620
\\__regex_replacement_exp_not:n .
    ..... 2567, 2567, 2840, 3621
\\__regex_replacement_g:w . 2779, 2779
\\__regex_replacement_g_digits:NN
    ..... 2779, 2782, 2785, 2792
\\__regex_replacement_lbrace:N ..
    .. 2624, 2781, 2821, 2839, 2852, 2852
\\__regex_replacement_normal:n ..
    ..... 2626, 2632, 2704, 2704,

```

[2757](#), [2787](#), [2814](#), [2849](#), [2857](#), [2872](#)
`__regex_replacement_normal_-`
 [aux:N](#) [2704](#), [2710](#), [2724](#)
`__regex_replacement_put:n`
 .. [2702](#), [2702](#), [2707](#), [2898](#), [2950](#), [3616](#)
`__regex_replacement_put_-`
 [submatch:n](#) . [2755](#), [2761](#), [2761](#), [2802](#)
`__regex_replacement_put_-`
 [submatch_aux:n](#)
 [2761](#), [2764](#), [2770](#), [3617](#)
`__regex_replacement_rbrace:N` ..
 [2621](#), [2801](#), [2843](#), [2843](#)
`__regex_replacement_set:n`
 [2610](#), [2611](#), [2655](#), [2692](#)
`\l__regex_replacement_tl`
 [3494](#), [3607](#), [3622](#)
`__regex_replacement_u:w` . [2837](#), [2837](#)
`__regex_return:`
 [135](#), [3011](#), [3017](#), [3045](#), [3047](#), [3122](#), [3122](#)
`\l__regex_right_state_int`
 [1865](#), [1889](#), [1933](#), [1934](#),
 [1954](#), [1966](#), [1973](#), [1982](#), [1983](#), [2022](#),
 [2029](#), [2035](#), [2048](#), [2056](#), [2106](#), [2110](#),
 [2121](#), [2135](#), [2144](#), [2156](#), [2160](#), [2164](#),
 [2167](#), [2172](#), [2175](#), [2178](#), [2186](#), [2200](#),
 [2203](#), [2206](#), [2209](#), [2213](#), [2229](#), [2272](#)
`\l__regex_right_state_seq`
 .. [1865](#), [1932](#), [1942](#), [1965](#), [1972](#), [2108](#)
`\l__regex_saved_success_bool` ...
 [107](#), [208](#), [215](#), [2302](#)
`__regex_show:N`
 [133](#), [1704](#), [1704](#), [2990](#), [3002](#)
`__regex_show:NN` [2985](#), [2995](#), [2996](#), [2997](#)
`__regex_show:Nn` [2985](#), [2985](#), [2986](#), [2987](#)
`__regex_show_char:n` [1736](#),
 [1740](#), [1743](#), [1747](#), [1756](#), [1769](#), [1769](#)
`__regex_show_class:NnnnN`
 [1723](#), [1805](#), [1805](#)
`__regex_show_group_aux:nnnnN` ..
 [1718](#), [1720](#), [1722](#), [1796](#), [1796](#)
`__regex_show_item_catcode:NnTF`
 [1750](#), [1752](#), [1841](#), [1841](#)

`__regex_show_item_exact_cs:n` ..
 [1757](#), [1854](#), [1854](#)
`\l__regex_show_lines_int`
 [517](#), [1777](#), [1809](#), [1812](#), [1819](#)
`__regex_show_one:n`
 ... [1712](#), [1725](#), [1728](#), [1736](#), [1739](#),
 [1743](#), [1746](#), [1756](#), [1760](#), [1775](#), [1775](#),
 [1791](#), [1798](#), [1802](#), [1815](#), [1831](#), [1859](#)
`__regex_show_pop:`
 [1785](#), [1787](#), [1794](#), [1801](#)
`\l__regex_show_prefix_seq`
 [516](#), [1710](#), [1713](#), [1761](#), [1781](#), [1786](#), [1788](#)
`__regex_show_push:n`
 .. [1762](#), [1785](#), [1785](#), [1792](#), [1799](#), [1810](#)
`__regex_show_scope:nn`
 [1754](#), [1759](#), [1785](#), [1789](#), [1846](#)
`__regex_single_match:` [106](#),
 [205](#), [2379](#), [2379](#), [3155](#), [3192](#), [3392](#), [3548](#)
`__regex_split:nnN` . [3062](#), [3208](#), [3208](#)
`__regex_standard_escapechar:` ..
 [4](#), [4](#), [301](#), [667](#), [1881](#), [1902](#)
`\l__regex_start_pos_int`
 ... [2258](#), [2277](#), [2353](#), [2358](#), [2365](#),
 [3214](#), [3226](#), [3239](#), [3242](#), [3365](#), [3447](#)
`\g__regex_state_active_intarray`
 [22](#), [91](#), [105](#)-
 [107](#), [2294](#), [2335](#), [2441](#), [2444](#), [2452](#), [2479](#)
`\l__regex_step_int` [22](#), [2291](#), [2337](#),
 [2403](#), [2442](#), [2446](#), [2454](#), [2468](#), [2470](#)
`__regex_store_state:n`
 [105](#), [2363](#), [2493](#), [2496](#), [2496](#)
`__regex_store_submatches:` ... [2496](#)
`__regex_store_submatches:n` ... [2515](#)
`__regex_store_submatches:nn` ...
 [2498](#), [2502](#)
`__regex_submatch_balance:n`
 .. [2559](#), [2591](#), [2591](#), [2674](#), [2776](#), [3296](#)
`\g__regex_submatch_begin_-`
 intarray
 .. [23](#), [116](#), [146](#), [2564](#), [2588](#), [2605](#),
 [2666](#), [3115](#), [3221](#), [3224](#), [3237](#), [3378](#)


```

\g__regex_submatch_case_intarray
    ..... 2685, 3115, 3360, 3366
\g__regex_submatch_end_intarray
    ..... 23, 146, 2589, 2598,
    ..... 3115, 3218, 3234, 3381, 3411, 3630
\l__regex_submatch_int .....
    ..... 23, 137, 141, 142, 148, 2344, 3112,
    ..... 3233, 3235, 3238, 3240, 3243, 3256,
    ..... 3355, 3359, 3361, 3362, 3436, 3444
\g__regex_submatch_prev_intarray
    ..... 23, 137, 146, 2563,
    ..... 2662, 3115, 3216, 3232, 3358, 3364
\g__regex_success_bool .....
    ..... 107, 209, 211, 214, 2302,
    ..... 2330, 2384, 2396, 3077, 3100, 3124,
    ..... 3173, 3354, 3395, 3565, 3571, 3612
\l__regex_success_pos_int .....
    ..... 2277, 2339, 2358, 2546, 3214
\l__regex_success_submatches_tl
    ..... 105, 146, 2289, 2549, 3369
\__regex_tests_action_cost:n ...
    ... 2000, 2002, 2015, 2021, 2030, 2048
\g__regex_thread_info_intarray .
    ... 22, 103, 106, 114, 2294, 2434, 2505
\__regex_tl_even_items:n .....
    ..... 53, 53, 54, 3098
\__regex_tl_even_items_loop:nn .
    ..... 53, 56, 59, 63
\__regex_tl_odd_items:n .....
    ..... 53, 53, 3074, 3097, 3171
\__regex_tmp:w .....
    ..... 142, 23, 25, 29, 31, 65, 65,
    ..... 942, 952, 953, 954, 955, 956, 979,
    ..... 990, 991, 3040, 3054, 3056, 3058,
    ..... 3060, 3062, 3252, 3257, 3280, 3287,
    ..... 3294, 3332, 3337, 3341, 3345, 3350
\__regex_toks_clear:N 7, 7, 1941, 1981
\__regex_toks_memcpy:NNn 12, 12, 2146
\__regex_toks_put_left:Nn .....
    . 21, 21, 1910, 1934, 1976, 2128, 2129
\__regex_toks_put_right:Nn .....
    ..... 24, 21, 27, 33, 1886, 1889,
    ..... 1906, 1954, 1978, 1989, 2220, 2265
\__regex_toks_set:Nn .....
    ..... 7, 8, 9, 10, 3143, 3652
\__regex_toks_use:w .....
    ..... 6, 6, 2443, 2581, 4006
\__regex_trace:nnn .....
    ..... 3988, 3989, 3991, 3992, 4005
\__regex_trace_pop:nnN ... 3988, 3990
\__regex_trace_push:nnN .. 3988, 3988
\g__regex_trace_regex_int .... 3998
\__regex_trace_states:n .. 3999, 3999
\__regex_two_if_eq:NNNN ..... 518
\__regex_two_if_eq:NNNNTF .....
    ..... 518, 777, 824, 837, 871,
    ..... 1041, 1078, 1098, 1099, 1168, 1203,
    ..... 1220, 1221, 1283, 1416, 1423, 2872
\__regex_use_i_delimit_by_q-
    recursion_stop:nw ... 87, 89, 1406
\__regex_use_none_delimit_by_q-
    nil:w ..... 61, 87, 91
\__regex_use_none_delimit_by_q-
    recursion_stop:w .....
    ..... 87, 87, 1384, 1408, 3370
\__regex_use_state: .....
    ..... 2439, 2439, 2456, 2482
\__regex_use_state_and_submatches:w
    ..... 110, 2432, 2448, 2448
\__regex_Z_test: ..... 42,
    ..... 972, 974, 991, 1611, 1732, 2217, 2244
\l__regex_zeroth_submatch_int ..
    ..... 137, 146, 3112, 3217, 3219,
    ..... 3222, 3225, 3355, 3365, 3367, 3379,
    ..... 3382, 3403, 3408, 3412, 3627, 3631
reverse commands:
\reverse_if:N .....
    ..... 109, 110, 127, 128, 133, 134

S
scan commands:
\scan_stop: .....
    ... 31, 42, 72, 193, 194, 307, 367,
    ..... 392, 404, 540, 1376, 1694, 1698,
    ..... 1701, 1856, 2458, 2897, 2949, 3252

```

seq commands:

<code>\seq_clear:N</code>	1761, 2645, 3279
<code>\seq_count:N</code>	2644
<code>\seq_get:NN</code>	2103, 2108
<code>\seq_if_empty:NTF</code>	2641
<code>\seq_item:Nn</code>	15
<code>\seq_map_function:NN</code>	1781, 1849
<code>\seq_new:N</code> ..	72, 516, 1867, 1868, 2556
<code>\seq_pop:NN</code>	1932, 1970, 1972, 2714
<code>\seq_pop_right:NN</code>	1710, 1788
<code>\seq_push:Nn</code> ...	1942, 1963, 1965, 2874
<code>\seq_put_right:Nn</code> ..	1713, 1786, 3289
<code>\seq_set_filter:NNn</code>	1844
<code>\seq_set_map_e:NNn</code>	1857
<code>\seq_set_split:Nnn</code>	1843, 1856
<code>\seq_use:Nn</code>	1860

str commands:

<code>\c_backslash_str</code>	319, 921
<code>\c_left_brace_str</code>	44, 382, 794, 798, 818, 831, 855, 1329, 1340, 1344, 1423, 1447, 2623
<code>\c_right_brace_str</code>	418, 804, 824, 837, 1347, 1351, 1444, 2620
<code>\str_case:nn</code>	1061
<code>\str_case:nnTF</code>	1671
<code>\str_case_e:nnTF</code>	802
<code>\str_if_eq:nnTF</code>	704, 3830, 3974
<code>\str_map_break:</code>	1570
<code>\str_map_function:nN</code>	1563
<code>\str_map_inline:nn</code>	2319
<code>\str_range:nnn</code>	1673

T

T_EX and L^AT_EX 2_ε commands:

<code>\afterassignment</code>	144
<code>\escapechar</code>	23
<code>\fontdimen</code>	20
<code>\lowercase</code>	128–130
<code>\newtoks</code>	22
<code>\toks</code>	22–24, 88, 98, 104, 105, 118, 129, 138, 155, 156, 166
<code>\uppercase</code>	128

tex commands:

<code>\tex_advance:D</code>	1000
<code>\tex_afterassignment:D</code>	3263, 3327, 3469
<code>\tex_catcode:D</code>	2730
<code>\tex_divide:D</code>	1001
<code>\tex_escapechar:D</code>	5
<code>\tex_lccode:D</code>	2897, 2949
<code>\tex_lowercase:D</code>	2898, 2950
<code>\tex_numexpr:D</code>	3
<code>\tex_the:D</code>	6, 25, 31, 34
<code>\tex_toks:D</code>	6, 9, 11, 16, 24, 25, 30, 31, 34

tl commands:

<code>\tl_analysis_map_inline:Nn</code> ...	1530
<code>\tl_analysis_map_inline:nn</code>	106, 150, 152, 2309, 3135
<code>\tl_build_begin:N</code>	82, 621, 1132, 1707, 1808, 2341, 2376, 2548, 2615, 3551
<code>\tl_build_end:N</code>	82, 651, 659, 1142, 1764, 1827, 2649, 3577, 3641
<code>\tl_build_get_intermediate:NN</code> .	2367
<code>\tl_build_put_right:Nn</code> .	122, 628, 646, 654, 658, 722, 725, 765, 779, 783, 908, 922, 963, 985, 998, 1030, 1043, 1047, 1129, 1135, 1141, 1145, 1188, 1478, 1482, 1489, 1495, 1516, 1532, 1550, 1778, 1823, 1836, 2410, 2638, 2703, 2772, 2829, 2832, 2846, 2914, 3555, 3657, 3660, 3668, 3671
<code>\tl_clear:N</code>	297, 2342, 2377
<code>\tl_const:Nn</code>	75, 2983
<code>\tl_count:n</code>	1394, 3065, 3069, 3088, 3092, 3162, 3166
<code>\tl_gclear:N</code>	2617
<code>\tl_gput_right:Nn</code>	2647, 2698, 2699, 2775
<code>\tl_gset:Nn</code>	2681, 2690
<code>\tl_gset_eq:NN</code>	2687, 2978
<code>\tl_if_blank:nTF</code>	2321
<code>\tl_if_empty:NTF</code>	2709

<code>\tl_if_empty:nTF</code>	<code>\token_case_meaning:NnTF</code>
..... 75, 1474, 3323, 3485, 3967 1592, 1607, 1640, 1650, 1661
<code>\tl_if_head_eq_meaning:nNTF</code> ...	<code>\token_if_eq_charcode:NNTF</code>
1561 413, 418, 1053, 1253,
<code>\tl_if_head_eq_meaning_p:nN</code> ...	1266, 1268, 1306, 1444, 2712, 2726
1393	<code>\token_if_eq_meaning:NNTF</code>
<code>\tl_if_in:nnTF</code> 749, 756, 1059, 1092, 1241,
194	1264, 1296, 1431, 1439, 1442, 2781,
<code>\tl_if_single:nTF</code>	2787, 2814, 2821, 2839, 2862, 2879
..... 1555, 1591, 1606, 1639	<code>\token_to_meaning:N</code>
<code>\tl_if_single_token:nTF</code>	698
695	<code>\token_to_str:N</code>
<code>\tl_item:nn</code>	315,
135, 3030, 3075	322, 432, 436, 1171, 2707, 3004,
<code>\tl_map_break:</code>	3068, 3091, 3165, 3785, 3989, 3991
20, 25	
<code>\tl_map_function:NN</code>	
1518	
<code>\tl_map_function:nN</code>	
742	
<code>\tl_map_inline:nn</code>	
25, 1915, 2369	
<code>\tl_map_tokens:nn</code>	
2688	
<code>\tl_new:N</code> 66, 67, 73, 74, 2284, 2289,	
2290, 2296, 2297, 2298, 2555, 2557,	
2677, 2678, 3492, 3493, 3494, 3495	
<code>\tl_put_right:Nn</code>	
304	
<code>\tl_set:Nn</code>	
..... 662, 1425, 1502, 1767, 1830,	
2360, 2381, 2391, 2407, 2412, 2455,	
2487, 2525, 2876, 3546, 3547, 3607	
<code>\tl_set_eq:NN</code>	
2549, 2973, 3465	
<code>\tl_tail:N</code>	
1368	
<code>\tl_to_str:N</code>	
126	
<code>\tl_to_str:n</code>	
10, 12, 126,	
742, 1701, 2833, 2992, 3069, 3092, 3166	
<code>\l_tmpa_tl</code>	
18	
token commands:	
<code>\c_parameter_token</code>	
117	
<code>\c_space_token</code>	
372, 413, 2726	