The l3str package: manipulating strings of characters*

The LaTeX3 Project†

Released 2013/07/24

Contents

1 Building strings **2**

2 Accessing the contents of a string **3**

3 String conditionals **5**

4 Viewing strings **6**

5 Scratch strings **7**

6 Internal l3str functions **7**

7 Possible additions to l3str **7**

8 l3str implementation **8**

8.1 String assignments 8
8.2 String variables and constants 9
8.3 Counting characters 9
8.4 Head and tail of a string 11
8.5 Accessing specific characters in a string 12
8.6 String conditionals 16
8.7 Viewing strings 17
8.8 Deprecated string functions 17

*This file describes v4576, last revised 2013/07/24.

†E-mail: latex-team@latex-project.org

Index 18

LaTeX3 provides a set of functions to manipulate token lists as strings of characters, ignoring the category codes of those characters.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and will not treat a token list or the corresponding string representation differently.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\c_backslash_str`
`\c_left_brace_str`
`\c_right_brace_str`
`\c_hash_str`
`\c_tilde_str`
`\c_percent_str`

Constant strings, containing a single character token, with category code 12. Any character can be accessed as `\iow_char:N` \⟨*character*⟩.

\tl_to_str:N ⋆
\tl_to_str:n ⋆

\tl_to_str:N ⟨*tl var*⟩
\tl_to_str:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, leaving the resulting character tokens in the input stream.

TEXhackers note: Converting a ⟨*token list*⟩ to a ⟨*string*⟩ yields a concatenation of the string representations of every token in the ⟨*token list*⟩. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not "letter".

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character "backslash", "lower-case a", "space", but it may also produce a single "lower-case a" if the escape character is negative and `a` is currently not a letter.

\str_new:N
\str_new:c

\str_new:N ⟨*str var*⟩

Creates a new ⟨*str var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*str var*⟩ will initially be empty.

\str_const:Nn
\str_const:(Nx|cn|cx)

\str_const:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ will be set globally to the ⟨*token list*⟩, converted to a string.

\str_set:Nn
\str_set:(Nx|cn|cx)
\str_gset:Nn
\str_gset:(Nx|cn|cx)

\str_set:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩.

\str_put_left:Nn
\str_put_left:(Nx|cn|cx)
\str_gput_left:Nn
\str_gput_left:(Nx|cn|cx)

\str_put_left:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

\str_put_right:Nn
\str_put_right:(Nx|cn|cx)
\str_gput_right:Nn
\str_gput_right:(Nx|cn|cx)

\str_put_right:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

2 Accessing the contents of a string

`\str_count:N` ⋆
`\str_count:n` ⋆
`\str_count_ignore_spaces:n` ⋆
`\str_count:n {⟨token list⟩}`

Leaves in the input stream the number of characters in the string representation of ⟨*token list*⟩, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ⋆
`\str_count_spaces:n` ⋆
`\str_count_spaces:n {⟨token list⟩}`

Leaves in the input stream the number of space characters in the string representation of ⟨*token list*⟩, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

`\str_head:N` ⋆
`\str_head:n` ⋆
`\str_head_ignore_spaces:n` ⋆
`\str_head:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then left in the input stream, with category code "other". The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the ⟨*string*⟩ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

`\str_tail:N` ⋆
`\str_tail:n` ⋆
`\str_tail_ignore_spaces:n` ⋆
`\str_tail:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` will trim only that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the ⟨*token list*⟩ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

\str_item:Nn ⋆ \str_item:nn {⟨*token list*⟩} {⟨*integer expression*⟩}
\str_item:nn ⋆
\str_item_ignore_spaces:nn ⋆

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, −1 is the right-most character, *etc.*

\str_range:Nnn ⋆ \str_range:nnn {⟨*token list*⟩} {⟨*start index*⟩} {⟨*end index*⟩}
\str_range:nnn ⋆
\str_range_ignore_spaces:nnn ⋆

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Positive ⟨*indices*⟩ are counted from the start of the string, 1 being the first character, and negative ⟨*indices*⟩ are counted from the end of the string, −1 being the last character. If either of ⟨*start index*⟩ or ⟨*end index*⟩ is 0, the result is empty. For instance,

```
\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }
```

will print `bcd`, `cdef`, `ef`, and an empty line to the terminal.

3 String conditionals

\str_if_eq_p:nn ⋆ \str_if_eq_p:nn {⟨tl_1⟩} {⟨tl_2⟩}
\str_if_eq_p:(Vn|on|no|nV|VV) ⋆ \str_if_eq:nnTF {⟨tl_1⟩} {⟨tl_2⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\str_if_eq:nnTF ⋆
\str_if_eq:(Vn|on|no|nV|VV)TF ⋆

Compares the string representations of the two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically `true`.

\str_if_eq_x_p:nn ⋆
\str_if_eq_x:nnTF ⋆

New: 2012-06-05

```
\str_if_eq_x_p:nn {⟨tl1⟩} {⟨tl2⟩}
\str_if_eq_x:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_x_p:nn { abc } { \tl_to_str:n { abc } }
```

is logically `true`.

\str_case:nnTF ⋆
\str_case:onTF ⋆

New: 2013-07-24

```
\str_case:nnTF {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩. If the two are equal (as described for `\str_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

\str_case_x:nnTF ⋆

New: 2013-07-24

```
\str_case_x:nnn {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the full expansion of the ⟨*test string*⟩ in turn with the full expansion of the ⟨*string cases*⟩. If the two full expansions are equal (as described for `\str_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case_x:nn`, which does nothing if there is no match, is also available. The ⟨*test string*⟩ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

4 Viewing strings

\str_show:N
\str_show:(c|n)

\str_show:N ⟨*tl var*⟩

Displays the content of the ⟨*str var*⟩ on the terminal.

5 Scratch strings

\l_tmpa_str
\l_tmpb_str

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_str
\g_tmpb_str

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

6 Internal l3str functions

__str_to_other:n ⋆

__str_to_other:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*other string*⟩, where spaces have category code "other". This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string, but there exist non-expandable ways to reach linear time.

__str_count_unsafe:n ⋆

__str_count_unsafe:n {⟨*other string*⟩}

This function expects an argument that is entirely made of characters with category "other", as produced by __str_to_other:n. It leaves in the input stream the number of character tokens in the ⟨*other string*⟩, faster than the analogous \str_count:n function.

__str_range_unsafe:nnn ⋆

__str_range_unsafe:nnn {⟨*other string*⟩} {⟨*start index*⟩} {⟨*end index*⟩}

Identical to \str_range:nnn except that the first argument is expected to be entirely made of characters with category "other", as produced by __str_to_other:n, and the result is also an ⟨*other string*⟩.

7 Possible additions to l3str

Semantically correct copies of some tl functions.

- \c_space_str
- \str_clear:N, \str_gclear:N, \str_clear_new:N, \str_gclear_new:N.

- `\str_concat:NNN`, `\str_gconcat:NNN`
- `\str_set_eq:NN`, `\str_gset_eq:NN`
- `\str_if_empty:NTF`, `\str_if_empty_p:N`
- `\str_if_exist:NTF`, `\str_if_exist_p:N`
- `\str_use:N`

Some functions that are not copies of `tl` functions.

- `\str_if_blank:NTF`, `\str_if_blank_p:N`.
- `\str_map_inline:Nn`, `\str_map_function:NN`, `\str_map_variable:NNn`, and `:n` analogs.
- Expandable `\str_if_in:nnTF`?
- `\str_if_head_eq:nNTF`, `\str_if_head_eq_p:nN`
- `\str_if_numeric/decimal/integer:n`, perhaps in l3fp?

8 l3str implementation

```
1 ⟨*initex | package⟩
2 ⟨@@=str⟩
3 \ProvidesExplPackage
4   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

The following string-related functions are currently defined in l3kernel.

- `\str_if_eq:nn`[pTF] and variants,
- `\str_if_eq_x_return:on`, `\str_if_eq_x_return:nn`
- `\tl_to_str:n`, `\tl_to_str:N`, `\tl_to_str:c`,
- `\token_to_str:N`, `\cs_to_str:N`
- `\str_head:n`, `\__str_head:w`, (copied here)
- `\str_tail:n`, `\__str_tail:w`, (copied here)
- `\__str_count_ignore_spaces` (unchanged)
- `\__str_count_loop:NNNNNNNNN` (unchanged)

8.1 String assignments

\str_new:N
\str_new:c

A string is simply a token list.

```
\cs_new_eq:NN \str_new:N \tl_new:N
\cs_generate_variant:Nn \str_new:N { c }
```

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

\str_set:Nn
\str_set:Nx
\str_set:cn
\str_set:cx
\str_gset:Nn
\str_gset:Nx
\str_gset:cn
\str_gset:cx
\str_const:Nn
\str_const:Nx
\str_const:cn
\str_const:cx
\str_put_left:Nn
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cx
\str_gput_left:Nn
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cx
\str_put_right:Nn
\str_put_right:Nx
\str_put_right:cn
\str_put_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx

Simply convert the token list inputs to ⟨*strings*⟩.

```
\tl_map_inline:nn
  {
    { set }
    { gset }
    { const }
    { put_left }
    { gput_left }
    { put_right }
    { gput_right }
  }
  {
    \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
      { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
    \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
  }
```

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

8.2 String variables and constants

\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_hash_str
\c_tilde_str
\c_percent_str

For all of those strings, use `\cs_to_str:N` to get characters with the correct category code.

```
\str_const:Nx \c_backslash_str { \cs_to_str:N \\ }
\str_const:Nx \c_left_brace_str  { \cs_to_str:N \{ }
\str_const:Nx \c_right_brace_str { \cs_to_str:N \} }
\str_const:Nx \c_hash_str      { \cs_to_str:N \# }
\str_const:Nx \c_tilde_str     { \cs_to_str:N \~ }
\str_const:Nx \c_percent_str   { \cs_to_str:N \% }
```

(*End definition for* `\c_backslash_str` *and others. These variables are documented on page* 2.)

\l_tmpa_str
\l_tmpb_str
\g_tmpa_str
\g_tmpb_str

Scratch strings.

```
\str_new:N \l_tmpa_str
\str_new:N \l_tmpb_str
\str_new:N \g_tmpa_str
\str_new:N \g_tmpb_str
```

(*End definition for* `\l_tmpa_str` *and others. These variables are documented on page* 7.)

8.3 Counting characters

\str_count_spaces:N
\str_count_spaces:n
__str_count_spaces_loop:wwwwwwwww

To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing `X`⟨*number*⟩, and that ⟨*number*⟩ is added to the sum of 9 that precedes, to adjust the result.

```
\cs_new_nopar:Npn \str_count_spaces:N
  { \exp_args:No \str_count_spaces:n }
\cs_new:Npn \str_count_spaces:n #1
  {
    \int_eval:n
      {
        \exp_after:wN \__str_count_spaces_loop:wwwwwwwww
        \tl_to_str:n {#1} ~
        X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
        \q_stop
      }
  }
\cs_new:Npn \__str_count_spaces_loop:wwwwwwwww #1~#2~#3~#4~#5~#6~#7~#8~#9~
  {
    \if_meaning:w X #9
      \use_i_delimit_by_q_stop:nw
    \fi:
    \c_nine + \__str_count_spaces_loop:wwwwwwwww
  }
```

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

\str_count:N
\str_count:n
\str_count_ignore_spaces:n
__str_count_unsafe:n
__str_count:n
__str_count_loop:NNNNNNNNN

To count characters in a string we could first escape all spaces using `\__str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The `_unsafe` variant expects a token list already converted to category code 12 characters, and is used by `\str_item:nn` and `\str_range:nnn`.

```
\cs_new_nopar:Npn \str_count:N { \exp_args:No \str_count:n }
\cs_new:Npn \str_count:n #1
  {
    \__str_count:n
      {
        \str_count_spaces:n {#1}
        + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
      }
  }
\cs_new:Npn \__str_count_unsafe:n #1
  {
    \__str_count:n
```

```
      { \__str_count_loop:NNNNNNNNN #1 }
  }
\cs_new:Npn \str_count_ignore_spaces:n #1
  {
    \__str_count:n
      { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
  }
\cs_new:Npn \__str_count:n #1
  {
    \int_eval:n
      {
        #1
        { X \c_eight } { X \c_seven } { X \c_six   }
        { X \c_five  } { X \c_four  } { X \c_three }
        { X \c_two   } { X \c_one   } { X \c_zero  }
        \q_stop
      }
  }
\cs_set:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_meaning:w X #9
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    \c_nine + \__str_count_loop:NNNNNNNNN
  }
```

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

8.4 Head and tail of a string

`\str_head:N` `\str_head:n` `\str_head_ignore_spaces:n` `\__str_head:w`

The `_ignore_spaces` variant is almost identical to `\tl_head:n`. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`. To circumvent the fact that TEX skips spaces when grabbing undelimited macro parameters, `\__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `\use_i_delimit_by_q_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `\__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `\__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `\use_i_delimit_by_q_stop:nw`.

```
\cs_new_nopar:Npn \str_head:N { \exp_args:No \str_head:n }
\cs_set:Npn \str_head:n #1
  {
    \exp_after:wN \__str_head:w
    \tl_to_str:n {#1}
    { { } } ~ \q_stop
  }
\cs_set:Npn \__str_head:w #1 ~ %
  { \use_i_delimit_by_q_stop:nw #1 { ~ } }
```

```
\cs_new:Npn \str_head_ignore_spaces:n #1
  {
    \exp_after:wN \use_i_delimit_by_q_stop:nw
    \tl_to_str:n {#1} { } \q_stop
  }
```

(*End definition for* `\str_head:N`, `\str_head:n`, *and* `\str_head_ignore_spaces:n`. *These functions are documented on page 4.*)

`\str_tail:N`
`\str_tail:n`
`\str_tail_ignore_spaces:n`
`\__str_tail_auxi:w`
`\__str_tail_auxii:w`

Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:`. This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:`. The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `\__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```
\cs_new_nopar:Npn \str_tail:N { \exp_args:No \str_tail:n }
\cs_set:Npn \str_tail:n #1
  {
    \exp_after:wN \__str_tail_auxi:w
    \reverse_if:N \if_charcode:w
        \scan_stop: \tl_to_str:n {#1} X X \q_stop
  }
\cs_set:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
\cs_new:Npn \str_tail_ignore_spaces:n #1
  {
    \exp_after:wN \__str_tail_auxii:w
    \tl_to_str:n {#1} \q_mark \q_mark \q_stop
  }
\cs_new:Npn \__str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }
```

(*End definition for* `\str_tail:N`, `\str_tail:n`, *and* `\str_tail_ignore_spaces:n`. *These functions are documented on page 4.*)

8.5 Accessing specific characters in a string

`\__str_to_other:n`
`\__str_to_other_loop:w`
`\__str_to_other_end:w`

First apply `\tl_to_str:n`, then replace all spaces by "other" spaces, 8 at a time, storing the converted part of the string between the `\q_mark` and `\q_stop` markers. The end is detected when `\__str_to_other_loop:w` finds one of the trailing `A`, distinguished from any contents of the initial token list by their category. Then `\__str_to_other_end:w` is called, and finds the result between `\q_mark` and the first `A` (well, there is also the need to remove a space).

```
\group_begin:
\char_set_lccode:nn { `\* } { `\  }
\char_set_lccode:nn { `\A } { `\A }
\tl_to_lowercase:n
  {
```

```
    \group_end:
    \cs_new:Npn \__str_to_other:n #1
      {
        \exp_after:wN \__str_to_other_loop:w \tl_to_str:n {#1} ~ %
        A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
      }
    \cs_new:Npn \__str_to_other_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
      {
        \if_meaning:w A #8
          \__str_to_other_end:w
        \fi:
        \__str_to_other_loop:w
        #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
      }
    \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
      { \fi: #2 }
  }
```

(*End definition for* `\__str_to_other:n`. *This function is documented on page* 7.)

`\__str_skip_c_zero:w`
`\__str_skip_loop:wNNNNNNNN`
`\__str_skip_end:w`
`\__str_skip_end:NNNNNNNN`

Removes `max(#1,0)` characters from the input stream, and then leaves `\c_zero`. This should be expanded using `\tex_romannumeral:D`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `\__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `\__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\c_zero` (see places where `\__str_skip_c_zero:w` is called).

```
\cs_new:Npn \__str_skip_c_zero:w #1;
  {
    \if_int_compare:w #1 > \c_eight
      \exp_after:wN \__str_skip_loop:wNNNNNNNN
    \else:
      \exp_after:wN \__str_skip_end:w
      \int_use:N \__int_eval:w
    \fi:
    #1 ;
  }
\cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
  { \exp_after:wN \__str_skip_c_zero:w \int_use:N \__int_eval:w #1 - \c_eight ; }
\cs_new:Npn \__str_skip_end:w #1 ;
  {
    \exp_after:wN \__str_skip_end:NNNNNNNN
    \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
  }
\cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \c_zero }
```

(*End definition for* `\__str_skip_c_zero:w`. *This function is documented on page* 7.)

__str_collect_delimit_by_q_stop:w
__str_collect_loop:wn
__str_collect_loop:wnNNNNNNN
__str_collect_end:wn
__str_collect_end:nnnnnnnnw

Collects `max(#1,0)` characters, and removes everything else until `\q_stop`. This is somewhat similar to `\__str_skip_c_zero:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `\__str_collect_end:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by `#1` characters from the input stream. Simply leaving this in the input stream will close the conditional properly and the `\or:` disappear.

```
\cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
  { \__str_collect_loop:wn #1 ; { } }
\cs_new:Npn \__str_collect_loop:wn #1 ;
  {
    \if_int_compare:w #1 > \c_seven
      \exp_after:wN \__str_collect_loop:wnNNNNNNN
    \else:
      \exp_after:wN \__str_collect_end:wn
    \fi:
    #1 ;
  }
\cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
  {
    \exp_after:wN \__str_collect_loop:wn
    \int_use:N \__int_eval:w #1 - \c_seven ;
    { #2 #3#4#5#6#7#8#9 }
  }
\cs_new:Npn \__str_collect_end:wn #1 ;
  {
    \exp_after:wN \__str_collect_end:nnnnnnnnw
    \if_case:w \if_int_compare:w #1 > \c_zero #1 \else: 0 \fi: \exp_stop_f:
    \or: \or: \or: \or: \or: \or: \fi:
  }
\cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
  { #1#2#3#4#5#6#7#8 }
```

(*End definition for* `\__str_collect_delimit_by_q_stop:w`. *This function is documented on page* 7.)

\str_item:Nn
\str_item:nn
\str_item_ignore_spaces:nn
__str_item_unsafe:nn
__str_item:ww

The `\str_item:nn` hands its argument with spaces escaped to `\__str_item_unsafe:nn`, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function cheats a little bit in that it doesn't hand to `\__str_item_unsafe:nn` an "other string". This is safe, as everything else is done with undelimited arguments. Then evaluate the ⟨*index*⟩ argument `#2` and count characters in the string, passing those two numbers to `\__str_item:ww` for further analysis. If the ⟨*index*⟩ is negative, shift by `#2`, and remove that number of characters before returning the next item in the input stream (and if `#1` is smaller than $-$`#2`, nothing is returned). If the ⟨*index*⟩ is positive, ignore that number (minus one) of characters before returning the next one. The shift by one is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the ⟨*index*⟩ is zero.

```
\cs_new_nopar:Npn \str_item:Nn { \exp_args:No \str_item:nn }
\cs_new:Npn \str_item:nn #1#2
  {
```

```
    \exp_args:Nf \tl_to_str:n
      {
        \exp_args:Nf \__str_item_unsafe:nn
          { \__str_to_other:n {#1} } {#2}
      }
  }
\cs_new:Npn \str_item_ignore_spaces:nn #1
  { \exp_args:No \__str_item_unsafe:nn { \tl_to_str:n {#1} } }
\cs_new:Npn \__str_item_unsafe:nn #1#2
  {
    \exp_after:wN \__str_item:ww
    \int_use:N \__int_eval:w #2 \exp_after:wN ;
    \__int_value:w \__str_count_unsafe:n {#1} ;
    #1 \q_stop
  }
\cs_new:Npn \__str_item:ww #1; #2;
  {
    \int_compare:nNnTF {#1} < \c_zero
      {
        \int_compare:nNnTF {#1} < {-#2}
          { \use_none_delimit_by_q_stop:w }
          {
            \exp_after:wN \use_i_delimit_by_q_stop:nw
            \tex_romannumeral:D \exp_after:wN \__str_skip_c_zero:w
              \int_use:N \__int_eval:w #1 + #2 ;
          }
      }
      {
        \int_compare:nNnTF {#1} > {#2}
          { \use_none_delimit_by_q_stop:w }
          {
            \exp_after:wN \use_i_delimit_by_q_stop:nw
            \tex_romannumeral:D \__str_skip_c_zero:w #1 ; { }
          }
      }
  }
```

(*End definition for* `\str_item:Nn`, `\str_item:nn`, *and* `\str_item_ignore_spaces:nn`. *These functions are documented on page* 4.)

`\__str_range_normalize:nn`

This function converts an ⟨*index*⟩ argument into an explicit position in the string (a result of 0 denoting "out of bounds"). Expects two explicit integer arguments: the ⟨*index*⟩ `#1` and the string count `#2`. If `#1` is negative, replace it by `#1 + #2 + 1`, then limit to the range $[0, \#2]$.

```
\cs_new:Npn \__str_range_normalize:nn #1#2
  {
    \int_eval:n
      {
        \if_int_compare:w #1 < \c_zero
          \if_int_compare:w #1 < -#2 \exp_stop_f:
```

```
            \c_zero
          \else:
            #1 + #2 + \c_one
          \fi:
        \else:
          \if_int_compare:w #1 < #2 \exp_stop_f:
            #1
          \else:
            #2
          \fi:
        \fi:
      }
  }
```

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

`\str_range:Nnn`
`\str_range:nnn`
`\str_range_ignore_spaces:nnn`
`\__str_range_unsafe:nnn`
`\__str_range:www`
`\__str_range:nnw`

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the ⟨*start index*⟩, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```
\cs_new_nopar:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
\cs_new:Npn \str_range:nnn #1#2#3
  {
    \exp_args:Nf \tl_to_str:n
      {
        \exp_args:Nf \__str_range_unsafe:nnn
          { \__str_to_other:n {#1} } {#2} {#3}
      }
  }
\cs_new:Npn \str_range_ignore_spaces:nnn #1
  { \exp_args:No \__str_range_unsafe:nnn { \tl_to_str:n {#1} } }
\cs_new:Npn \__str_range_unsafe:nnn #1#2#3
  {
    \exp_after:wN \__str_range:www
    \__int_value:w \__str_count_unsafe:n {#1} \exp_after:wN ;
    \int_use:N \__int_eval:w #2 - \c_one \exp_after:wN ;
    \int_use:N \__int_eval:w #3 ;
    #1 \q_stop
  }
\cs_new:Npn \__str_range:www #1; #2; #3;
  {
    \exp_args:Nf \__str_range:nnw
      { \__str_range_normalize:nn {#2} {#1} }
      { \__str_range_normalize:nn {#3} {#1} }
  }
\cs_new:Npn \__str_range:nnw #1#2
  {
    \exp_after:wN \__str_collect_delimit_by_q_stop:w
```

```
    \int_use:N \__int_eval:w #2 - #1 \exp_after:wN ;
    \tex_romannumeral:D \__str_skip_c_zero:w #1 ;
  }
```

(*End definition for* `\str_range:Nnn`, `\str_range:nnn`, *and* `\str_range_ignore_spaces:nnn`. *These functions are documented on page* 7.)

8.6 String conditionals

`\str_if_eq_p:NN`
`\str_if_eq:NNTF`
`\str_if_eq_p:nn`
`\str_if_eq_x_p:nn`
`\str_if_eq:nnTF`
`\str_if_eq_x:nnTF`

Note that `\str_if_eq:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.

```
\prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
  {
    \if_int_compare:w \pdftex_strcmp:D { \tl_to_str:N #1 } { \tl_to_str:N #2 }
      = \c_zero \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\str_if_eq:NN`. *These functions are documented on page* 5.)

`\str_case:nnTF`
`\str_case:onTF`
`\str_case_x:nnTF`

Defined in l3basics at present.

(*End definition for* `\str_case:nnTF`, `\str_case:onTF`, *and* `\str_case_x:nnTF`. *These functions are documented on page* 6.)

8.7 Viewing strings

`\str_show:n`
`\str_show:N`
`\str_show:c`

Displays a string on the terminal.

```
\cs_new_eq:NN \str_show:n \tl_show:n
\cs_new_eq:NN \str_show:N \tl_show:N
\cs_generate_variant:Nn \str_show:N { c }
```

(*End definition for* `\str_show:n`, `\str_show:N`, *and* `\str_show:c`. *These functions are documented on page* ??.)

8.8 Deprecated string functions

Deprecated 2013-01-20 for removal by 2013-04-30

`\str_substr:Nnn`
`\str_substr:nnn`
`\str_substr_ignore_spaces:nnn`
`\__str_substr:nnn`

These functions used to allow for an empty argument to denote the start/end of the string. We reimplement them here by first checking for an empty argument, then only calling the appropriate version of the `\str_range:nnn` function.

```
\cs_new:Npn \str_substr:Nnn #1 { \__str_substr:nnn { \str_range:Nnn #1 } }
\cs_new:Npn \str_substr:nnn #1 { \__str_substr:nnn { \str_range:nnn {#1} } }
\cs_new:Npn \str_substr_ignore_spaces:nnn #1
  { \__str_substr:nnn { \str_range_ignore_spaces:nnn {#1} } }
\cs_new:Npn \__str_substr:nnn #1#2#3
  {
    \tl_if_empty:nTF {#2}
      { \tl_if_empty:nTF {#3} { #1 { 1} { -1 } } { #1 { 1} {#3} } }
      { \tl_if_empty:nTF {#3} { #1 {#2} { -1 } } { #1 {#2} {#3} } }
  }
```

(*End definition for* `\str_substr:Nnn`, `\str_substr:nnn`, *and* `\str_substr_ignore_spaces:nnn`. *These functions are documented on page* **??**.)

Deprecated 2013-01-20 for removal by 2013-04-30

`\c_lbrace_str`
`\c_rbrace_str`

```
\cs_new_eq:NN \c_lbrace_str \c_left_brace_str
\cs_new_eq:NN \c_rbrace_str \c_right_brace_str
```

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

⟨/initex | package⟩

Index

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

Symbols

\# 25
\% 27
* 117
\\ 22
\{ 23
\} 24
__int_eval:w 145, 150, 171, 196, 209, 255, 256, 268
__int_value:w 197, 254
__str_collect_delimit_by_q_stop:w 157, 157, 267
__str_collect_end:nnnnnnnnw 157, 176, 180
__str_collect_end:wn 157, 164, 174
__str_collect_loop:wn 157, 158, 159, 170
__str_collect_loop:wnNNNNNNN 157, 162, 168
__str_count:n 51, 54, 62, 67, 70
__str_count_loop:NNNNNNNNN 51, 57, 63, 68, 81, 86
__str_count_spaces_loop:wwwwwwwww 32, 38, 44, 49
__str_count_unsafe:n *6*, 51, 60, 197, 254
__str_head:w 88, 91, 95
__str_item:ww 182, 195, 200
__str_item_unsafe:nn . 182, 187, 192, 193
__str_range:nnw 240, 261, 265
__str_range:www 240, 253, 259
__str_range_normalize:nn 221, 221, 262, 263
__str_range_unsafe:nnn *6*, 240, 245, 250, 251
__str_skip_c_zero:w 139, 139, 150, 208, 217, 269
__str_skip_end:NNNNNNNN . . 139, 153, 156
__str_skip_end:w 139, 144, 151
__str_skip_loop:wNNNNNNNN 139, 142, 149
__str_substr:nnn 279, 279, 280, 282, 283
__str_tail_auxi:w 102, 105, 109
__str_tail_auxii:w 102, 112, 115
__str_to_other:n . . *6*, 116, 122, 188, 246
__str_to_other_end:w 116, 131, 136
__str_to_other_loop:w 116, 124, 127, 133
\~ 26

\␣ 117

A

\A 118

C

\c_backslash_str *1*, 22, 22
\c_eight 75, 141, 150
\c_five 76
\c_four 76
\c_hash_str *1*, 22, 25
\c_lbrace_str 289, 289
\c_left_brace_str *1*, 22, 23, 289
\c_nine 49, 86
\c_one 77, 229, 255
\c_percent_str *1*, 22, 27

\c_rbrace_str 289, 290
\c_right_brace_str 1, 22, 24, 290
\c_seven 75, 161, 171
\c_six 75
\c_three 76
\c_tilde_str 1, 22, 26
\c_two 77
\c_zero . . . 77, 156, 177, 202, 225, 227, 274
\char_set_lccode:nn 117, 118
\cs_generate_variant:Nn 6, 20, 278
\cs_new:Npn 34, 44, 52, 60, 65, 70, 97, 110, 115, 122, 127, 136, 139, 149, 151, 156, 157, 159, 168, 174, 180, 183, 191, 193, 200, 221, 241, 249, 251, 259, 265, 279, 280, 281, 283
\cs_new_eq:NN 5, 276, 277, 289, 290
\cs_new_nopar:Npn 32, 51, 88, 102, 182, 240
\cs_new_protected:cpx 18
\cs_set:Npn 81, 89, 95, 103, 109
\cs_to_str:N 22, 23, 24, 25, 26, 27

E

\else: . . . 143, 163, 177, 228, 231, 234, 274
\exp_after:wN 38, 57, 68, 84, 91, 99, 105, 112, 124, 142, 144, 150, 153, 162, 164, 170, 176, 195, 196, 207, 208, 216, 253, 254, 255, 267, 268
\exp_args:Nc 20
\exp_args:Nf 185, 187, 243, 245, 261
\exp_args:No 33, 51, 88, 102, 182, 192, 240, 250
\exp_not:c 19
\exp_not:N 19
\exp_stop_f: 154, 177, 226, 232
\ExplFileDate 4
\ExplFileDescription 4
\ExplFileName 4
\ExplFileVersion 4

F

\fi: 48, 85, 109, 132, 136, 137, 146, 156, 165, 177, 178, 230, 236, 237, 274

G

\g_tmpa_str 6, 28, 30
\g_tmpb_str 6, 28, 31
\group_begin: 116
\group_end: 121

I

\if_case:w 154, 177
\if_charcode:w 106
\if_int_compare:w 141, 161, 177, 225, 226, 232, 273
\if_meaning:w 46, 83, 130
\int_compare:nNnTF 202, 204, 213
\int_eval:n 36, 72, 223
\int_use:N 145, 150, 171, 196, 209, 255, 256, 268

L

\l_tmpa_str 6, 28, 28
\l_tmpb_str 6, 28, 29

O

\or: 154, 178

P

\pdftex_strcmp:D 273
\prg_new_conditional:Npnn 271
\prg_return_false: 274
\prg_return_true: 274
\ProvidesExplPackage 3

Q

\q_mark 113, 115, 125, 136
\q_stop . 41, 78, 93, 100, 107, 109, 113, 115, 125, 128, 134, 136, 180, 198, 257

R

\reverse_if:N 106

S

\scan_stop: 107
\str_case:nnTF 5, 276
\str_case:onTF 276
\str_case_x:nnTF 5, 276
\str_const:cn 7
\str_const:cx 7
\str_const:Nn 2, 7
\str_const:Nx 7, 22, 23, 24, 25, 26, 27
\str_count:N 3, 51, 51
\str_count:n 51, 51, 52
\str_count_ignore_spaces:n 3, 51, 65
\str_count_spaces:N 3, 32, 32
\str_count_spaces:n 32, 33, 34, 56
\str_gput_left:cn 7
\str_gput_left:cx 7
\str_gput_left:Nn 2, 7
\str_gput_left:Nx 7
\str_gput_right:cn 7
\str_gput_right:cx 7
\str_gput_right:Nn 2, 7

\str_gput_right:Nx 7
\str_gset:cn 7
\str_gset:cx 7
\str_gset:Nn 2, 7
\str_gset:Nx 7
\str_head:N 3, 88, 88
\str_head:n 88, 88, 89
\str_head_ignore_spaces:n 3, 88, 97
\str_if_eq:NN 271
\str_if_eq:NNTF 271
\str_if_eq:nnTF 4, 271
\str_if_eq_p:NN 271
\str_if_eq_p:nn 4, 271
\str_if_eq_x:nnTF 5, 271
\str_if_eq_x_p:nn 5, 271
\str_item:Nn 4, 182, 182
\str_item:nn 182, 182, 183
\str_item_ignore_spaces:nn . . 4, 182, 191
\str_new:c 5
\str_new:N 2, 5, 5, 6, 28, 29, 30, 31
\str_put_left:cn 7
\str_put_left:cx 7
\str_put_left:Nn 2, 7
\str_put_left:Nx 7
\str_put_right:cn 7
\str_put_right:cx 7
\str_put_right:Nn 2, 7
\str_put_right:Nx 7
\str_range:Nnn 4, 240, 240, 279
\str_range:nnn 240, 240, 241, 280
\str_range_ignore_spaces:nnn 4, 240, 249, 282
\str_set:cn 7
\str_set:cx 7
\str_set:Nn 2, 7
\str_set:Nx 7
\str_show:c 276
\str_show:N 6, 276, 277, 278
\str_show:n 276, 276
\str_substr:Nnn 279, 279
\str_substr:nnn 279, 280
\str_substr_ignore_spaces:nnn . 279, 281
\str_tail:N 3, 102, 102
\str_tail:n 102, 102, 103
\str_tail_ignore_spaces:n . . . 3, 102, 110

T

\tex_romannumeral:D 208, 217, 269
\tl_if_empty:nTF 285, 286, 287
\tl_map_inline:nn 7
\tl_new:N 5
\tl_show:N 277
\tl_show:n 276
\tl_to_lowercase:n 119
\tl_to_str:N 2, 273
\tl_to_str:n 19, 39, 57, 68, 92, 100, 107, 113, 124, 185, 192, 243, 250

U

\use_i_delimit_by_q_stop:nw 47, 96, 99, 207, 216
\use_none_delimit_by_q_stop:w 84, 205, 214