# The akshar package

Vu Van Dung

Version 0.2 — 2020/09/06

https://ctan.org/pkg/akshar
https://github.com/joulev/akshar

Abstract

This package provides tools to deal with special characters in a Devanagari string.

## Contents



# 1 Introduction

When dealing with processing strings in the Devanagari script, normal LaTeX commands usually find some difficulties in distinguishing "normal" characters, like क, and "special" characters, for example ◌ु or ◌ी. Let's consider this example code:

```
\ExplSyntaxOn
\tl_set:Nn \l_tmpa_tl { की}
\tl_count:N \l_tmpa_tl \c_space_token tokens.
\ExplSyntaxOff
```

2 tokens.

The output is 2, but the number of characters in it is only one! The reason is quite simple: the compiler treats ◌ी as a normal character, and it shouldn't do so.

To tackle that, this package provides expl3 functions to "convert" a given string, written in the Devanagari script, to a sequence of token lists. each of these token lists is a "true" Devanagari character. You can now do anything you want with this sequence; and this package does provide some front-end macros for some simple actions on the input string.

# 2 User manual

Due to the current implementation, all of these macros and functions are not expandable.

## 2.1 LaTeX 2ε macros

`\aksharStrLen` `\aksharStrLen {⟨token list⟩}`

Return the number of Devanagari characters in the ⟨token list⟩.

```
There are \aksharStrLen{ नमस्कार} characters in नमस्कार.\par
\ExplSyntaxOn
\pkg{expl3}~returns~\tl_count:n { नमस्कार},~which~is~wrong.
\ExplSyntaxOff
```

There are 4 characters in नमस्कार. expl3 returns 7, which is wrong.

`\aksharStrHead` `\aksharStrHead {⟨token list⟩} {⟨n⟩}`

Return the first character of the token list.

```
\aksharStrHead { मंळीममड}
```

मं

`\aksharStrTail` `\aksharStrTail {⟨token list⟩} {⟨n⟩}`

Return the last character of the token list.

```
\aksharStrTail { ळीममडमं}
```

मं

`\aksharStrChar` `\aksharStrChar {⟨token list⟩} {⟨n⟩}`

Return the *n*-th character of the token list.

```
3rd character of नमस्कारis \aksharStrChar{ नमस्कार}{3}.\par
\ExplSyntaxOn
It~is~not~\tl_item:nn { नमस्कार} {3}.
\ExplSyntaxOff
```

3rd character of नमस्कार is स्का. It is not स.

`\aksharStrReplace` `\aksharStrReplace*` `\aksharStrReplace {⟨tl 1⟩} {⟨tl 2⟩} {⟨tl 3⟩}`

Replace all occurences of ⟨tl 2⟩ in ⟨tl 1⟩ with ⟨tl 3⟩, and leaves the modified ⟨tl 1⟩ in the input stream.

The starred variant will replace only the first occurence of ⟨tl 2⟩, all others are left intact.

```
\ExplSyntaxOn
\pkg{expl3} ~ output:\par
\tl_set:Nn \l_tmpa_tl { मममडडमंळीममड}
\tl_replace_all:Nnnn \l_tmpa_tl { म} { स्का}
\tl_use:N \l_tmpa_tl\par
\ExplSyntaxOff
\cs{aksharStrReplace} output:\par
\aksharStrReplace { मममडडमंळीममड} { म} { स्का}
```

expl3 output: स्कास्कास्काडडस्कांळीस्कास्काड
`\aksharStrReplace` output: स्कास्कास्काडडमंळीस्कास्काड

```
\ExplSyntaxOn
\pkg{expl3} ~ output:\par
\tl_set:Nn \l_tmpa_tl { मममडडमंळीममड}
\tl_replace_once:Nnn \l_tmpa_tl { मम} { स्का}
\tl_use:N \l_tmpa_tl\par
\ExplSyntaxOff
\cs{aksharStrReplace*} output:\par
\aksharStrReplace* { मममडडमंळीममड} { मम} { स्का}
```

expl3 output: स्कांममडडमंळीममड
`\aksharStrReplace*` output: ममंस्काडडमंळीममड

`\aksharStrRemove` `\aksharStrRemove*` `\aksharStrRemove {⟨tl 1⟩} {⟨tl 2⟩}`

Remove all occurences of ⟨tl 2⟩ in ⟨tl 1⟩, and leaves the modified ⟨tl 1⟩ in the input stream.

The starred variant will remove only the first occurence of ⟨tl 2⟩, all others are left intact.

expl3 output:
डडंळीड
\aksharStrRemove output:
डडमंळीड

```
\ExplSyntaxOn
\pkg{expl3} ~ output:\par
\tl_set:Nn \l_tmpa_tl { मममडडमंळीममड}
\tl_remove_all:Nn \l_tmpa_tl { म}
\tl_use:N \l_tmpa_tl\par
\ExplSyntaxOff
\cs{aksharStrRemove} output:\par
\aksharStrRemove { मममडडमंळीममड} { म}
```

expl3 output:
◌ंममडडमंळीममड
\aksharStrRemove* output:
ममंडडमंळीममड

```
\ExplSyntaxOn
\pkg{expl3} ~ output:\par
\tl_set:Nn \l_tmpa_tl { ममंममडडमंळीममड}
\tl_remove_once:Nn \l_tmpa_tl { मम}
\tl_use:N \l_tmpa_tl\par
\ExplSyntaxOff
\cs{aksharStrRemove*} output:\par
\aksharStrRemove* { ममंममडडमंळीममड} { मम}
```

## 2.2 expl3 functions

This section assumes that you have a basic knowledge in LaTeX3 programming. All macros in 2.1 directly depend on the following function, so it is much more powerful than all features we have described above.

`\akshar_convert:Nn`
`\akshar_convert:(cn|Nx|cx)`

`\akshar_convert:Nn` ⟨seq var⟩ {⟨token list⟩}

This function converts ⟨token list⟩ to a sequence of characters, that sequence is stored in ⟨seq var⟩.

न, म, स्का, and र

```
\ExplSyntaxOn
\akshar_convert:Nn \l_tmpa_seq { नमस्कार}
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
\ExplSyntaxOff
```

# 3 Implementation

```
⟨@@=akshar⟩
⟨*package⟩
```

Declare the package. By loading fontspec, xparse, and in turn, expl3, are also loaded.

```
\RequirePackage{fontspec}
\ProvidesExplPackage {\aksharPackageName}
  {\aksharPackageDate} {\aksharPackageVersion} {\aksharPackageDescription}
```

## 3.1 Variable declarations

`\c__akshar_joining_tl`
`\c__akshar_diacritics_tl`

These variables store the special characters we need to take into account:

- \c__akshar_joining_tl is the “connecting” character ◌्.
- \c__akshar_diacritics_tl is the list of all diacritics: ◌ा, ◌ि, ◌ी, ◌ु, ◌ू, ◌े, ◌ै, ◌ो, ◌ौ, ◌ं, ◌ः, ◌ॢ, ◌ृ, ◌ॅ, ◌ँ, ◌़, ◌ॆ, ◌ॊ, ◌ॉ, ◌ॎ, ◌ॏ, ◌॑, ◌॒, ◌॓, ◌॔, ◌ॕ, ◌ॖ, ◌ॗ, ◌ॄ, ◌ऺ, ◌ऻ, ◌꣠, ◌꣡, ◌꣢, ◌꣣, ◌꣤, ◌꣥, ◌꣦, ◌꣧, ◌꣨, ◌꣩, ◌꣪, ◌꣫, ◌꣬, ◌꣭, ◌꣮, ◌꣯, ◌꣰, ◌꣱.

```
\tl_const:Nn \c__akshar_joining_tl { ◌्}
\tl_const:Nn \c__akshar_diacritics_tl
  {
    ◌ा, ◌ि, ◌ी, ◌ु, ◌ू, ◌े, ◌ै, ◌ो, ◌ौ, ◌ं, ◌ः, ◌ॢ, ◌ृ,
    ◌ॅ, ◌ँ, ◌़, ◌ॆ, ◌ॊ, ◌ॉ, ◌ॎ, ◌ॏ, ◌॑, ◌॒, ◌॓, ◌॔, ◌ॕ, ◌ॖ, ◌ॗ,
    ◌ॄ, ◌ऺ, ◌ऻ, ◌꣠, ◌꣡, ◌꣢, ◌꣣, ◌꣤, ◌꣥, ◌꣦, ◌꣧, ◌꣨, ◌꣩, ◌꣪,
```

```
    উ   ক   ন   প   র   ৰি   ঽ
    ◌,  ◌,  ◌,  ◌,  ◌,  ◌,  ◌,  ◌
  }
```

(End definition for `\c__akshar_joining_tl` and `\c__akshar_diacritics_tl`.)

`\l__akshar_prev_joining_bool`

When we get to a normal character, we need to know whether it is joined, i.e. whether the previous character is the joining character. This boolean variable takes care of that.

```
\bool_new:N \l__akshar_prev_joining_bool
```

(End definition for `\l__akshar_prev_joining_bool`.)

`\l__akshar_char_seq`

This local sequence stores the output of the converter.

```
\seq_new:N \l__akshar_char_seq
```

(End definition for `\l__akshar_char_seq`.)

`\c__akshar_str_g_tl`
`\c__akshar_str_seq_tl`
`\c__akshar_str_comma_tl`

Some self-descriptive constant variables.

```
\tl_const:Nx \c__akshar_str_g_tl { \tl_to_str:n {g} }
\tl_const:Nx \c__akshar_str_seq_tl { \tl_to_str:n {seq} }
\tl_const:Nx \c__akshar_str_comma_tl { \tl_to_str:n {,} }
```

(End definition for `\c__akshar_str_g_tl`, `\c__akshar_str_seq_tl`, and `\c__akshar_str_comma_tl`.)

`\l__akshar_tmpa_tl`
`\l__akshar_tmpb_tl`
`\l__akshar_tmpa_seq`
`\l__akshar_tmpb_seq`
`\l__akshar_tmpc_seq`
`\l__akshar_tmpd_seq`
`\l__akshar_tmpe_seq`
`\l__akshar_tmpa_int`
`\l__akshar_tmpb_int`

Some temporary variables.

```
\tl_new:N \l__akshar_tmpa_tl
\tl_new:N \l__akshar_tmpb_tl
\seq_new:N \l__akshar_tmpa_seq
\seq_new:N \l__akshar_tmpb_seq
\seq_new:N \l__akshar_tmpc_seq
\seq_new:N \l__akshar_tmpd_seq
\seq_new:N \l__akshar_tmpe_seq
\int_new:N \l__akshar_tmpa_int
\int_new:N \l__akshar_tmpb_int
```

(End definition for `\l__akshar_tmpa_tl` and others.)

## 3.2 Messages

In `\akshar_convert:Nn` and friends, the argument needs to be a sequence variable. There will be an error if it isn't.

```
\msg_new:nnnn { akshar } { err_not_a_sequence_variable }
  { #1 ~ is ~ not ~ a ~ valid ~ LaTeX3 ~ sequence ~ variable. }
  {
    You ~ have ~ requested ~ me ~ to ~ assign ~ some ~ value ~ to ~
    the ~ control ~ sequence ~ #1, ~ but ~ it ~ is ~ not ~ a ~ valid ~
    sequence ~ variable. ~ Read ~ the ~ documentation ~ of ~ expl3 ~
    for ~ more ~ information. ~ Proceed ~ and ~ I ~ will ~ pretend ~
    that ~ #1 ~ is ~ a ~ local ~ sequence ~ variable ~ (beware ~ that ~
    unexpected ~ behaviours ~ may ~ occur).
  }
```

In `\aksharStrChar`, we need to guard against accessing an 'out-of-bound' character (like trying to get the 8th character in a 5-character string.)

```
\msg_new:nnnn { akshar } { err_character_out_of_bound }
  { Character ~ index ~ out ~ of ~ bound. }
  {
    You ~ are ~ trying ~ to ~ get ~ the ~ #2 ~ character ~ of ~ the ~
    string ~ #1. ~ However ~ that ~ character ~ doesn't ~ exist. ~
    Make ~ sure ~ that ~ you ~ use ~ a ~ number ~ between ~ and ~ not ~
    including ~ 0 ~ and ~ #3, ~ so ~ that ~ I ~ can ~ return ~ a ~
    good ~ output. ~ Proceed ~ and ~ I ~ will ~ return ~
    \token_to_str:N \scan_stop:.
  }
```

In \aksharStrHead and \aksharStrTail, the string must not be blank.

```
\msg_new:nnnn { akshar } { err_string_empty }
  { The ~ input ~ string ~ is ~ empty. }
  {
    To ~ get ~ the ~ #1 ~ character ~ of ~ a ~ string, ~ that ~ string ~
    must ~ not ~ be ~ empty, ~ but ~ the ~ input ~ string ~ is ~ empty.
    Make ~ sure ~ the ~ string ~ contains ~ something, ~ or ~ proceed ~
    and ~ I ~ will ~ use ~ \token_to_str:N \scan_stop:.
  }
```

## 3.3 Utilities

\tl_if_in:NoTF When we get to a character which is not the joining one, we need to know if it is a diacritic. The current character is stored in a variable, so an expanded variant is needed. We only need it to expand only once.

```
\prg_generate_conditional_variant:Nnn \tl_if_in:Nn { No } { TF }
```

(End definition for *\tl_if_in:NoTF*.)

\seq_set_split:Nxx A variant we will need in __akshar_var_if_global.

```
\cs_generate_variant:Nn \seq_set_split:Nnn { Nxx }
```

(End definition for *\seq_set_split:Nxx*.)

\msg_error:nnx Some variants of l3msg functions that we will need when issuing error mes-
\msg_error:nnnxx sages.

```
\cs_generate_variant:Nn \msg_error:nnn { nnx }
\cs_generate_variant:Nn \msg_error:nnnnn { nnnxx }
```

(End definition for *\msg_error:nnx* and *\msg_error:nnnxx*.)

__akshar_tl_if_in_ncomma:NNTF This conditional is essentially \tl_if_in:Nn, but if #2 is a comma this conditional always return false.

```
\prg_new_conditional:Npnn \__akshar_tl_if_in_ncomma:NN #1 #2 { T, F, TF }
  {
    \tl_if_eq:NNTF \c__akshar_str_comma_tl #2 { \prg_return_false: }
      { \tl_if_in:NoTF #1 {#2} { \prg_return_true: } { \prg_return_false: } }
  }
```

(End definition for *__akshar_tl_if_in_ncomma:NNTF*.)

__akshar_var_if_global:NTF This conditional checks if #1 is a global sequence variable or not. In other words, it returns true iff #1 is a control sequence in the format \g_⟨name⟩_seq. If it is not a sequence variable, this function will (TODO) issue an error message.

```
\prg_new_conditional:Npnn \__akshar_var_if_global:N #1 { T, F, TF }
  {
    \bool_if:nTF
      { \exp_last_unbraced:Nf \use_iii:nnn { \cs_split_function:N #1 } }
      {
        \msg_error:nnx { akshar } { err_not_a_sequence_variable }
          { \token_to_str:N #1 }
        \prg_return_false:
      }
      {
        \seq_set_split:Nxx \l__akshar_tmpb_seq { \token_to_str:N _ }
          { \exp_last_unbraced:Nf \use_i:nnn { \cs_split_function:N #1 } }
        \seq_get_left:NN  \l__akshar_tmpb_seq \l__akshar_tmpa_tl
        \seq_get_right:NN \l__akshar_tmpb_seq \l__akshar_tmpb_tl
        \tl_if_eq:NNTF \c__akshar_str_seq_tl \l__akshar_tmpb_tl
          {
            \tl_if_eq:NNTF \c__akshar_str_g_tl \l__akshar_tmpa_tl
              { \prg_return_true: } { \prg_return_false: }
```

```
      }
      {
        \msg_error:nnx { akshar } { err_not_a_sequence_variable }
          { \token_to_str:N #1 }
        \prg_return_false:
      }
    }
  }
```

(End definition for `\__akshar_var_if_global:NTF`.)

`\__akshar_int_append_ordinal:n` Append st, nd, rd or th to interger #1. Will be needed in error messages.

```
\cs_new:Npn \__akshar_int_append_ordinal:n #1
  {
    #1
    \int_case:nnF { #1 }
      {
        { 11 } { th }
        { 12 } { th }
        { 13 } { th }
        { -11 } { th }
        { -12 } { th }
        { -13 } { th }
      }
      {
        \int_compare:nNnTF { #1 } > { -1 }
          {
            \int_case:nnF { #1 - 10 * (#1 / 10) }
              {
                { 1 } { st }
                { 2 } { nd }
                { 3 } { rd }
              } { th }
          }
          {
            \int_case:nnF { (- #1) - 10 * ((- #1) / 10) }
              {
                { 1 } { st }
                { 2 } { nd }
                { 3 } { rd }
              } { th }
          }
      }
  }
```

(End definition for `\__akshar_int_append_ordinal:n`.)

## 3.4 The `\akshar_convert:Nn` function and its variants

`\akshar_convert:Nn` `\akshar_convert:cn` `\akshar_convert:Nx` `\akshar_convert:cx` This converts #2 to a sequence of true Devanagari characters. The sequence is set to #1, which should be a sequence variable.

```
\cs_new:Npn \akshar_convert:Nn #1 #2
  {
```

Clear anything stored in advance. We don't want different calls of the function to conflict with each other.

```
    \seq_clear:N \l__akshar_char_seq
    \bool_set_false:N \l__akshar_prev_joining_bool
```

Loop through every token of the input.

```
    \tl_map_variable:NNn {#2} \l__akshar_map_tl
      {
        \__akshar_tl_if_in_ncomma:NNTF
          \c__akshar_diacritics_tl \l__akshar_map_tl
          {
```

It is a diacritic. We append the current diacritic to the last item of the sequence instead of pushing the diacritic to a new sequence item.

```
            \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
            \seq_put_right:Nx \l__akshar_char_seq
              { \l__akshar_tmpa_tl \l__akshar_map_tl }
          }
          {
            \tl_if_eq:NNTF \l__akshar_map_tl \c__akshar_joining_tl
              {
```

In this case, the character is the joining character, ◌. What we do is similar to the above case, but `\l__akshar_prev_joining_bool` is set to true so that the next character is also appended to this item.

```
                \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
                \seq_put_right:Nx \l__akshar_char_seq
                  { \l__akshar_tmpa_tl \l__akshar_map_tl }
                \bool_set_true:N \l__akshar_prev_joining_bool
              }
              {
```

Now the character is normal. We see if we can push to a new item or not. It depends on the boolean variable.

```
                \bool_if:NTF \l__akshar_prev_joining_bool
                  {
                    \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
                    \seq_put_right:Nx \l__akshar_char_seq
                      { \l__akshar_tmpa_tl \l__akshar_map_tl }
                    \bool_set_false:N \l__akshar_prev_joining_bool
                  }
                  {
                    \seq_put_right:Nx
                      \l__akshar_char_seq { \l__akshar_map_tl }
                  }
              }
          }
      }
```

Set #1 to `\l__akshar_char_seq`. The package automatically determines whether the variable is a global one or a local one.

```
    \__akshar_var_if_global:NTF #1
      { \seq_gset_eq:NN #1 \l__akshar_char_seq }
      { \seq_set_eq:NN #1 \l__akshar_char_seq }
}
```

Generate variants that might be helpful for some.

```
\cs_generate_variant:Nn \akshar_convert:Nn { cn, Nx, cx }
```

(End definition for *\akshar_convert:Nn*. This function is documented on page 3.)

## 3.5 Other internal functions

`\__akshar_seq_push_seq:NN` Append sequence #1 to the end of sequence #2. A simple loop will do.

```
\cs_new:Npn \__akshar_seq_push_seq:NN #1 #2
  { \seq_map_inline:Nn #2 { \seq_put_right:Nn #1 { ##1 } } }
```

(End definition for *__akshar_seq_push_seq:NN*.)

`\__akshar_replace:NnnnN` If #5 is `\c_false_bool`, this function replaces all occurences of #3 in #2 by #4 and stores the output sequence to #1. If #5 is `\c_true_bool`, the replacement only happens once.

The algorithm used in this function: We will use `\l__akshar_tmpa_int` to store the "current position" in the sequence of #3. At first it is set to 1.

We will store any subsequence of #2 that may match #3 to a temporary sequence. If it doesn't match, we push this temporary sequence to the output, but if it matches, #4 is pushed instead.

We loop over #2. For each of these loops, we need to make sure the `\l__akshar_tmpa_int`-th item must indeed appear in #3. So we need to compare that with the length of #3.

- If now `\l__akshar_tmpa_int` is greater than the length of #3, the whole of #3 has been matched somewhere, so we reinitialize the integer to `1` and push #4 to the output.

  Note that it is possible that the current character might be the start of another match, so we have to compare it to the first character of #3. If they are not the same, we may now push the current mapping character to the output and proceed; otherwise the current character is pushed to the temporary variable.

- Otherwise, we compare the current loop character of #2 with the `\l__akshar_tmpa_int`-th character of #3.

  – If they are the same, we still have a chance that it will match, so we increase the "iterator" `\l__akshar_tmpa_int` by `1` and push the current mapping character to the temporary sequence.

  – If they are the same, the temporary sequence won't match. Let's push that sequence to the output and set the iterator back to `1`.

    Note that now the iterator has changed. Who knows whether the current character may start a match? Let's compare it to the first character of #3, and do as in the case of `\l__akshar_tmpa_int` is greater than the length of #3.

The complexity of this algorithm is $O(m \max(n, p))$, where $m, n, p$ are the lengths of the sequences created from #2, #3 and #4. As #3 and #4 are generally short strings, this is (almost) linear to the length of the original sequence #2.

```
\cs_new:Npn \__akshar_replace:NnnnN #1 #2 #3 #4 #5
  {
    \akshar_convert:Nn \l__akshar_tmpc_seq {#2}
    \akshar_convert:Nn \l__akshar_tmpd_seq {#3}
    \akshar_convert:Nn \l__akshar_tmpe_seq {#4}
    \seq_clear:N \l__akshar_tmpa_seq
    \seq_clear:N \l__akshar_tmpb_seq
    \int_set:Nn \l__akshar_tmpa_int { 1 }
    \int_set:Nn \l__akshar_tmpb_int { 0 }
    \seq_map_variable:NNn \l__akshar_tmpc_seq \l__akshar_map_tl
      {
        \int_compare:nNnTF { \l__akshar_tmpb_int } > { 0 }
          { \seq_put_right:NV \l__akshar_tmpb_seq \l__akshar_map_tl }
          {
            \int_compare:nNnTF
              {\l__akshar_tmpa_int} = {1 + \seq_count:N \l__akshar_tmpd_seq}
              {
                \bool_if:NT {#5}
                  { \int_incr:N \l__akshar_tmpb_int }
                \seq_clear:N \l__akshar_tmpb_seq
                \__akshar_seq_push_seq:NN
                  \l__akshar_tmpa_seq \l__akshar_tmpe_seq
                \int_set:Nn \l__akshar_tmpa_int { 1 }
                \tl_set:Nx \l__akshar_tmpa_tl
                  { \seq_item:Nn \l__akshar_tmpd_seq { 1 } }
                \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpa_tl
                  {
                    \int_incr:N \l__akshar_tmpa_int
                    \seq_put_right:NV \l__akshar_tmpb_seq \l__akshar_map_tl
                  }
                  {
                    \seq_put_right:NV \l__akshar_tmpa_seq \l__akshar_map_tl
                  }
              }
```

```
            {
              \tl_set:Nx \l__akshar_tmpa_tl
                {
                  \seq_item:Nn \l__akshar_tmpd_seq { \l__akshar_tmpa_int }
                }
              \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpa_tl
                {
                  \int_incr:N \l__akshar_tmpa_int
                  \seq_put_right:NV \l__akshar_tmpb_seq \l__akshar_map_tl
                }
                {
                  \int_set:Nn \l__akshar_tmpa_int { 1 }
                  \__akshar_seq_push_seq:NN
                    \l__akshar_tmpa_seq \l__akshar_tmpb_seq
                  \seq_clear:N \l__akshar_tmpb_seq
                  \tl_set:Nx \l__akshar_tmpa_tl
                    { \seq_item:Nn \l__akshar_tmpd_seq { 1 } }
                  \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpa_tl
                    {
                      \int_incr:N \l__akshar_tmpa_int
                      \seq_put_right:NV
                        \l__akshar_tmpb_seq \l__akshar_map_tl
                    }
                    {
                      \seq_put_right:NV
                        \l__akshar_tmpa_seq \l__akshar_map_tl
                    }
                }
            }
        }
    }
  \__akshar_seq_push_seq:NN \l__akshar_tmpa_seq \l__akshar_tmpb_seq
  \__akshar_var_if_global:NTF #1
    { \seq_gset_eq:NN #1 \l__akshar_tmpa_seq }
    { \seq_set_eq:NN #1 \l__akshar_tmpa_seq }
}
```

(End definition for *__akshar_replace:NnnnN*.)

## 3.6 Front-end LaTeX 2ε macros

\aksharStrLen Expands to the length of the string.

```
\NewDocumentCommand \aksharStrLen {m}
  {
    \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
    \seq_count:N \l__akshar_tmpa_seq
  }
```

(End definition for *\aksharStrLen*. This function is documented on page 2.)

\aksharStrChar Returns the *n*-th character of the string.

```
\NewDocumentCommand \aksharStrChar {mm}
  {
    \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
    \bool_if:nTF
      {
        \int_compare_p:nNn {#2} > {0} &&
        \int_compare_p:nNn {#2} < {1 + \seq_count:N \l__akshar_tmpa_seq}
      }
      { \seq_item:Nn \l__akshar_tmpa_seq { #2 } }
      {
        \msg_error:nnnxx { akshar } { err_character_out_of_bound }
          { #1 } { \__akshar_int_append_ordinal:n { #2 } }
          { \int_eval:n { 1 + \seq_count:N \l__akshar_tmpa_seq } }
        \scan_stop:
      }
  }
```

(End definition for *\aksharStrChar*. This function is documented on page 2.)

\aksharStrHead Return the first character of the string.

```
\NewDocumentCommand \aksharStrHead {m}
  {
    \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
    \int_compare:nNnTF { \seq_count:N \l__akshar_tmpa_seq } = {0}
      {
        \msg_error:nnn { akshar } { err_character_out_of_bound }
          { first }
        \scan_stop:
      }
      { \seq_item:Nn \l__akshar_tmpa_seq { 1 } }
  }
```

(End definition for *\aksharStrHead*. This function is documented on page 2.)

\aksharStrTail Return the last character of the string.

```
\NewDocumentCommand \aksharStrTail {m}
  {
    \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
    \int_compare:nNnTF { \seq_count:N \l__akshar_tmpa_seq } = {0}
      {
        \msg_error:nnn { akshar } { err_character_out_of_bound }
          { last }
        \scan_stop:
      }
      { \seq_item:Nn \l__akshar_tmpa_seq {\seq_count:N \l__akshar_tmpa_seq} }
  }
```

(End definition for *\aksharStrTail*. This function is documented on page 2.)

\aksharStrReplace
\aksharStrReplace* Replace occurences of #3 of a string #2 with another string #4.

```
\NewDocumentCommand \aksharStrReplace {smmm}
  {
    \IfBooleanTF {#1}
      {
        \__akshar_replace:NnnnN \l__akshar_tmpa_seq
          {#2} {#3} {#4} \c_true_bool
      }
      {
        \__akshar_replace:NnnnN \l__akshar_tmpa_seq
          {#2} {#3} {#4} \c_false_bool
      }
    \seq_use:Nn \l__akshar_tmpa_seq {}
  }
```

(End definition for *\aksharStrReplace* and *\aksharStrReplace**. These functions are documented on page 2.)

\aksharStrRemove
\aksharStrRemove* Remove occurences of #3 in #2. This is just a special case of \aksharStrReplace.

```
\NewDocumentCommand \aksharStrRemove {smm}
  {
    \IfBooleanTF {#1}
      {
        \__akshar_replace:NnnnN \l__akshar_tmpa_seq
          {#2} {#3} {} \c_true_bool
      }
      {
        \__akshar_replace:NnnnN \l__akshar_tmpa_seq
          {#2} {#3} {} \c_false_bool
      }
    \seq_use:Nn \l__akshar_tmpa_seq {}
  }
```

(End definition for `\aksharStrRemove` and `\aksharStrRemove*`. These functions are documented on page 2.)

```
305 ⟨/package⟩
```

# Index

Underlined page numbers point to the definition, all others indicate the places where it is used or described.

T



U