Unicode algorithms for LuaTEX*

Marcel Krüger†

August 31, 2025

Dealing with general Unicode encoded data comes with many challenges because it has to respect individual concerns of many different scripts and languages. The Unicode consortium maintains multiple useful algorithms which can sometimes make this task much easier.

`lua-uni-algos` tries to make the most fundamental algorithms available for authors of Lua-based packages to aid in handling Unicode data.

Currently this package implements:

Unicode normalization Normalize a given Lua string into any of the normalization forms NFC, NFD, NFKC, or NFKD as specified in the Unicode standard, section 2.12.

Case folding Fold Unicode codepoints into a form which eliminates all case distinctions. This can be used for case-independent matching of strings. Not to be confused with case mapping which maps all characters to lower/upper/titlecase: In contrast to case mapping, case folding is mostly locale independent but does not give results which should be shown to users.

Grapheme cluster segmentation Identify a grapheme cluster, a unit of text which is perceived as a single character by typical users, according to the rules in UAX #29, section 3.

Word boundary segmentation Identify word boundaries according to the rules in UAX #29, section 4.

1 Normalization

Unicode normalization is handled by the Lua module `lua-uni-normalize`. You can either load it directly with

```
local normalize = require'lua-uni-normalize'
```

*This document corresponds to `lua-uni-algos` v0.5.

†E-Mail: tex@2krueger.de

or if you need access to all implemented algorithms you can use

```
local uni_algos = require'lua-uni-algos'
local normalize = uni_algos.normalize
```

Then, four functions are available: `normalize.NFC`, `normalize.NFD`, `normalize.NFKC`, and `normalize.NFKD`. If you do not know which of these you need, then you should probably `normalize.NFC`. All functions are used in the same way:

```
local str = "Äpfel…"
print("Original:", str)
print("NFC:", normalize.NFC(str))
print("NFD:", normalize.NFD(str))
print("NFKC:", normalize.NFKC(str))
print("NFKD:", normalize.NFKD(str))
```

This results in

```
Original: Äpfel…
NFC: Äpfel…
NFD: Äpfel…
NFKC: Äpfel...
NFKD: Äpfel...
```

(This example is shown in Latin Modern Mono which has the (for this purpose) very useful property of not handling combining character very well. In a well-behaving font, the '...C' and '...D' lines should look the same.)

Additionally for direct normalization of LuaTEX node lists is supported. There are two functions `normalize.node.NFC` and `normalize.direct.NFC` taking upto four parameters: The first parameter is the head of the node list to be converted. The second parameter is the font id of the affected character nodes. Only non-protected glyph nodes of the specified font will be normalized. Pass `nil` for the font to normalize without respecting the font in the process. The third parameter is an optional table. If it is not `nil`, normalization is supressed if it might add glyph which map to `false` (or `nil`) in this table. If the forth argument is `true`, normalization will never join two glyph nodes with different attributes.

For NFD and NFKD equivalent functions exists without the last parameter (since they never compose nodes, they never have to deal with composing nodes with different attributes.

NFKC is not supported for node list normalization since the author is not convinced that there is any usecase for it. (Probably there isn't any usecase for node list NFKD normalization either, but that was easy to implement while NFKC would need separate data tables.

2 Case folding

For case folding load the Lua module `lua-uni-case`. You can either load it directly with

```
local uni_case = require'lua-uni-case'
```

or if you need access to all implemented algorithms you can use

```
local uni_algos = require'lua-uni-algos'
local uni_case = uni_algos.case
```

The main function is `uni_case.casefold(str, full, special)`. It accepts three parameters: A Lua string `str` to be case folded, a boolean `full` to specify if the number of codepoints is allowed to change in the progress (This should normally be set to `true`.) and a boolean `special` which enables special handling for Turkish languages (In most cases, this should be set to `false`.) The function returns the case-folded string:

```
local str = "Straße…"
print("Original:", str)
print("Case folded (full=false):", uni_case.casefold(str, false, false))
print("Case folded (full=true):", uni_case.casefold(str, true, false))
```

This results in
```
Original:Straße…
Case folded (full=false):straße…
Case folded (full=true):strasse…
```

In most cases, you will want to normalize the string after casefolding.

For cases where you want to casefold something which is not given as a Lua string, you can use the function `uni_case.casefold_lookup(cp, full, special)`. Instead of a string, it accepts a codepoint as first parameter and returns a table of codepoints. A string can be casefolded by replacing every codepoints with the sequence of codepoints returned by `uni_case.casefold_lookup`. If `casefold_lookup` returns `false` or `nil`, the codepoint should not be changed.

3 Grapheme clusters

Grapheme cluster handling is handled by the Lua module `lua-uni-graphemes`. You can either load it directly with

```
local graphemes = require'lua-uni-graphemes'
```

or if you need access to all implemented algorithms you can use

```
local uni_algos = require'lua-uni-algos'
local graphemes = uni_algos.graphemes
```

Sometimes we want to look at a single character of a string, but identifying what a character is is not that easy in Unicode. A simple example is the character from the previous section: "Ä" The NFD form is certainly a single character, but is encoded using two codepoints: U+0041 (LATIN CAPITAL LETTER A) and U+0308 (COMBINING DIAERESIS). Or the Tamil letter Ni which is encoded as U+0BA8 (TAMIL LETTER NA) followed by U+0BBF

(TAMIL VOWEL SIGN I). But sometimes it can be useful to identify characters, e.g. for letterspacing or letterines.

There are two main interfaces for this: One iterator for iterating over grapheme clusters and one direct interface to the underlying state machine:

```
for final, first, grapheme in graphemes.graphemes'Äpfel' do
  print(grapheme)
end
```

```
Ä
p
f
e
l
```

The more powerful state machine interface `graphemes.read_codepoint` takes two parameters: A new codepoint and a state. At the beginning, the state can be omitted. For every codepoint in your input, call the function with the new codepoint and the last state. Then there are two return values: The first one is a boolean telling you if the current codepoint is the beginning of a new cluster, the second is a new state you have to pass with the next codepoint.

So e.g. to find cluster boundaries in the Unicode codepoint sequence U+0041 U+0308 U+0BA8 U+0BBF you could use

```
local graphemes = require'lua-uni-graphemes'
local new_cluster, state
new_cluster, state = graphemes.read_codepoint(0x0041, state)
print(new_cluster)
new_cluster, state = graphemes.read_codepoint(0x0308, state)
print(new_cluster)
new_cluster, state = graphemes.read_codepoint(0x0BA8, state)
print(new_cluster)
new_cluster, state = graphemes.read_codepoint(0x0BBF, state)
print(new_cluster)
```

resulting in

```
true
nil
true
nil
```

meaning the first and third codepoint start a new cluster.

Do not try to interpret the `state`, it has no defined values and might change at any point.

4 Word boundaries

Word segmentation is handled by the Lua module `lua-uni-words`. You can either load it directly with

```
local words = require'lua-uni-words'
```

or if you need access to all implemented algorithms you can use

```
local uni_algos = require'lua-uni-algos'
local words = uni_algos.words
```

This is used to identify word boundaries. Unicode describes these as the boundaries users would expect when searching in a text for whole words.

The UAX also suggests that for some use-cases useful words can be determined from these by taking any segment between word boundaries and filtering out all segments "containing only whitespace, punctuation and similar characters".

Currently only the string interface is recommended:

```
for final, first, word_segment in words.word_boundaries'This text will be split into words segments!'
  print('"' .. word_segment .. '"')
end
```

```
"This"
" "
"text"
" "
"will"
" "
"be"
" "
"split"
" "
"into"
" "
"words"
" "
"segments"
"!"
```