++ed by:
23 PAUSE users
10 non-PAUSE users

SBURKE
Sean M. Burke

Contents

NAME

Text::Unidecode -- plain ASCII transliterations of Unicode text

SYNOPSIS

```
use utf8;
use Text::Unidecode;
print unidecode(
  "北京\n"
  # Chinese characters for Beijing (U+5317 U+4EB0)
);

# That prints: Bei Jing
```

DESCRIPTION

It often happens that you have non-Roman text data in Unicode, but you can't display it-- usually because you're trying to show it to a user via an application that doesn't support Unicode, or because the fonts you need aren't accessible. You could represent the Unicode characters as "???????" or "\15BA\15A0\1610...", but that's nearly useless to the user who actually wants to read what the text says.

What Text::Unidecode provides is a function, `unidecode(...)` that takes Unicode data and tries to represent it in US-ASCII characters (i.e., the universally displayable characters between 0x00 and 0x7F). The representation is almost always an attempt at *transliteration*-- i.e., conveying, in Roman letters, the pronunciation expressed by the text in some other writing system. (See the example in the synopsis.)

NOTE:

To make sure your perldoc/Pod viewing setup for viewing this page is working: The six-letter word "résumé" should look like "resume" with an "/" accent on each "e".

For further tests, and help if that doesn't work, see below, "A POD ENCODING TEST".

DESIGN PHILOSOPHY

Unidecode's ability to transliterate from a given language is limited by two factors:

- The amount and quality of data in the written form of the original language

  So if you have Hebrew data that has no vowel points in it, then Unidecode cannot guess what vowels should appear in a pronunciation. S f y hv n vwls n th npt, y wn't gt ny vwls n th tpt. (This is a specific application of the general principle of "Garbage In, Garbage Out".)

- Basic limitations in the Unidecode design

  Writing a real and clever transliteration algorithm for any single language usually requires a lot of time, and at least a passable knowledge of the language involved. But Unicode text can convey more languages than I could possibly learn (much less create a transliterator for) in the entire rest of my lifetime. So I put a cap on how intelligent Unidecode could be, by insisting that it support only context-*in*sensitive transliteration. That means missing the finer details of any given writing system, while still hopefully being useful.

Unidecode, in other words, is quick and dirty. Sometimes the output is not so dirty at all: Russian and Greek seem to work passably; and while Thaana (Divehi, AKA Maldivian) is a definitely non-Western writing system, setting up a mapping from it to Roman letters seems to work pretty well. But sometimes the output is *very dirty:* Unidecode does quite badly on Japanese and Thai.

If you want a smarter transliteration for a particular language than Unidecode provides, then you should look for (or write) a transliteration algorithm specific to that language, and apply it instead of (or at least before) applying Unidecode.

In other words, Unidecode's approach is broad (knowing about dozens of writing systems), but shallow (not being meticulous about any of them).

FUNCTIONS

Text::Unidecode provides one function, `unidecode(...)`, which is exported by default. It can be used in a variety of calling contexts:

`$out = unidecode( $in );` # scalar context

This returns a copy of $in, transliterated.

`$out = unidecode( @in );` # scalar context

This is the same as `$out = unidecode(join "", @in);`

`@out = unidecode( @in );` # list context

This returns a list consisting of copies of @in, each transliterated. This is the same as `@out = map scalar(unidecode($_)), @in;`

`unidecode( @items );` # void context

`unidecode( @bar, $foo, @baz );` # void context

Each item on input is replaced with its transliteration. This is the same as `for(@bar, $foo, @baz) { $_ = unidecode($_) }`

You should make a minimum of assumptions about the output of `unidecode(...)`. For example, if you assume an all-alphabetic (Unicode) string passed to `unidecode(...)` will return an all-alphabetic string, you're wrong-- some alphabetic Unicode characters are transliterated as strings containing punctuation (e.g., the Armenian letter "Թ" (U+0539), currently transliterates as "T`" (capital-T then a backtick).

However, these are the assumptions you *can* make:

- Each character 0x0000 - 0x007F transliterates as itself. That is, `unidecode(...)` is 7-bit pure.
- The output of `unidecode(...)` always consists entirely of US-ASCII characters-- i.e., characters 0x0000 - 0x007F.
- All Unicode characters translate to a sequence of (any number of) characters that are newline ("\n") or in the range 0x0020-0x007E. That is, no Unicode character translates to "\x01", for example. (Although if you have a "\x01" on input, you'll get a "\x01" in output.)
- Yes, some transliterations produce a "\n" but it's just a few, and only with good reason. Note that the value of newline ("\n") varies from platform to platform-- see perlport.
- Some Unicode characters may transliterate to nothing (i.e., empty string).
- Very many Unicode characters transliterate to multi-character sequences. E.g., Unihan character U+5317, "北", transliterates as the four-character string "Bei ".
- Within these constraints, *I may change* the transliteration of characters in future versions. For example, if someone convinces me that that the Armenian letter "Թ", currently transliterated as "T`", would be better transliterated as "D", I *may* well make that change.
- Unfortunately, there are many characters that Unidecode doesn't know a transliteration for. This is generally because the character has been added since I last revised the Unidecode data tables. I'm *always* catching up!

DESIGN GOALS AND CONSTRAINTS

Text::Unidecode is meant to be a transliterator of last resort, to be used once you've decided that you can't just display the Unicode data as is, *and once you've decided you don't have a more clever, language-specific transliterator available,* or once you've *already applied* smarter algorithms or mappings that you prefer and you now just want Unidecode to do cleanup.

Unidecode transliterates context-insensitively-- that is, a given character is replaced with the same US-ASCII (7-bit ASCII) character or characters, no matter what the surrounding characters are.

The main reason I'm making Text::Unidecode work with only context-insensitive substitution is that it's fast, dumb, and straightforward enough to be feasible. It doesn't tax my (quite limited) knowledge of world languages. It doesn't require me writing a hundred lines of code to get the Thai syllabification right (and never knowing whether I've gotten it wrong, because I don't know Thai), or spending a year trying to get Text::Unidecode to use the ChaSen algorithm for Japanese, or trying to write heuristics for telling the difference between Japanese, Chinese, or Korean, so it knows how to transliterate any given Uni-Han glyph. And moreover, context-insensitive substitution is still mostly useful, but still clearly couldn't be mistaken for authoritative.

Text::Unidecode is an example of the 80/20 rule in action-- you get 80% of the usefulness using just 20% of a "real" solution.

A "real" approach to transliteration for any given language can involve such increasingly tricky contextual factors as these:

The previous / preceding character(s)

What a given symbol "X" means, could depend on whether it's followed by a consonant, or by vowel, or by some diacritic character.

Syllables

A character "X" at end of a syllable could mean something different from when it's at the start-- which is especially problematic when the language involved doesn't explicitly mark where one syllable stops and the next starts.

Parts of speech

What "X" sounds like at the end of a word, depends on whether that word is a noun, or a verb, or what.

Meaning

By semantic context, you can tell that this ideogram "X" means "shoe" (pronounced one way) and not "time" (pronounced another), and that's how you know to transliterate it one way instead of the other.

Origin of the word

"X" means one thing in loanwords and/or placenames (and derivatives thereof), and another in native words.

"It's just that way"

"X" normally makes the /X/ sound, except for this list of seventy exceptions (and words based on them, sometimes indirectly). Or: you never can tell which of the three ways to pronounce "X" this word actually uses; you just have to know which it is, so keep a dictionary on hand!

Language

The character "X" is actually used in several different languages, and you have to figure out which you're looking at before you can determine how to transliterate it.

Out of a desire to avoid being mired in *any* of these kinds of contextual factors, I chose to exclude *all of them* and just stick with context-insensitive replacement.

A POD ENCODING TEST

- "Brontë" is six characters that should look like "Bronte", but with double-dots on the "e" character.
- "Résumé" is six characters that should look like "Resume", but with /-shaped accents on the "e" characters.
- "læti" should be *four* letters long-- the second letter should not be two letters "ae", but should be a single letter that looks like an "a" entirely fused with an "e".
- "χρονος" is six Greek characters that should look kind of like: xpovoc
- "КАК ВАС ЗОВУТ" is three short Russian words that should look a lot like: KAK BAC 3OBYT
- "സവ" is two Malayalam characters that should look like: sw
- "丫二十一" is four Chinese characters that should look like: `Y=+-`
- "Ｈｅｌｌｏ" is five characters that should look like: Hello

If all of those come out right, your Pod viewing setup is working fine-- welcome to the 2010s! If those are full of garbage characters, consider viewing this page as HTML at https://metacpan.org/pod/Text::Unidecode or http://search.cpan.org/perldoc?Text::Unidecode

If things look mostly okay, but the Malayalam and/or the Chinese are just question-marks or empty boxes, it's probably just that your computer lacks the fonts for those.

TODO

Lots:

* Rebuild the Unihan database. (Talk about hitting a moving target!)

* Add tone-numbers for Mandarin hanzi? Namely: In Unihan, when tone marks are present (like in "kMandarin: dào", should I continue to transliterate as just "Dao", or should I put in the tone number: "Dao4"? It would be pretty jarring to have digits appear where previously there was just alphabetic stuff-- But tone numbers make Chinese more readable. (I have a clever idea about doing this, for Unidecode v2 or v3.)

* Start dealing with characters over U+FFFF. Cuneiform! Emojis! Whatever!

* Fill in all the little characters that have crept into the Misc Symbols Etc blocks.

* More things that need tending to are detailed in the TODO.txt file, included in this distribution. Normal installs probably don't leave the TODO.txt lying around, but if nothing else, you can see it at http://search.cpan.org/search?dist=Text::Unidecode

MOTTO

The Text::Unidecode motto is:

```
It's better than nothing!
```

...in *both* meanings: 1) seeing the output of `unidecode(...)` is better than just having all font-unavailable Unicode characters replaced with "?"'s, or rendered as gibberish; and 2) it's the worst, i.e., there's nothing that Text::Unidecode's algorithm is better than. All sensible transliteration algorithms (like for German, see below) are going to be smarter than Unidecode's.

WHEN YOU DON'T LIKE WHAT UNIDECODE DOES

I will repeat the above, because some people miss it:

Text::Unidecode is meant to be a transliterator of *last resort,* to be used once you've decided that you can't just display the Unicode data as is, *and once you've decided you don't have a more clever, language-specific transliterator available*-- or once you've *already applied* a smarter algorithm and now just want Unidecode to do cleanup.

In other words, when you don't like what Unidecode does, *do it yourself.* Really, that's what the above says. Here's how you would do this for German, for example:

In German, there's the typographical convention that an umlaut (the double-dots on: ä ö ü) can be written as an "-e", like with "Schön" becoming "Schoen". But Unidecode doesn't do that-- I have Unidecode simply drop the umlaut accent and give back "Schon".

(I chose this not because I'm a big meanie, but because *generally* changing "ü" to "ue" is disastrous for all text that's *not in German*. Finnish "Hyvää päivää" would turn into "Hyvaeae paeivaeae". And I discourage you from being *yet another* German who emails me, trying to impel me to consider a typographical nicety of German to be more important than *all other languages*.)

If you know that the text you're handling is probably in German, and you want to apply the "umlaut becomes -e" rule, here's how to do it for yourself (and then use Unidecode as *the fallback* afterwards):

```
use utf8;  # <-- probably necessary.

our( %German_Characters ) = qw(
 Ä AE   ä ae
 Ö OE   ö oe
 Ü UE   ü ue
 ß ss
);

use Text::Unidecode qw(unidecode);

sub german_to_ascii {
  my($german_text) = @_;

  $german_text =~
    s/([ÄäÖöÜüß])/$German_Characters{$1}/g;

  # And now, as a *fallthrough*:
  $german_text = unidecode( $german_text );
  return $german_text;
}
```

To pick another example, here's something that's not about a specific language, but simply having a preference that may or may not agree with Unidecode's (i.e., mine). Consider the "¥" symbol. Unidecode changes that to "Y=". If you want "¥" as "YEN", then...

```
use Text::Unidecode qw(unidecode);

sub my_favorite_unidecode {
  my($text) = @_;

  $text =~ s/¥/YEN/g;

  # ...and anything else you like, such as:
  $text =~ s/€/Euro/g;

  # And then, as a fallback,...
  $text = unidecode($text);

  return $text;
}
```

Then if you do:

```
print my_favorite_unidecode("You just won ¥250,000 and €40,000!!!");
```

...you'll get:

```
You just won YEN250,000 and Euro40,000!!!
```

...just as you like it.

(By the way, the reason *I* don't have Unidecode just turn "¥" into "YEN" is that the same symbol also stands for yuan, the Chinese currency. A "Y=" is nicely, *safely* neutral as to whether we're talking about yen or yuan-- Japan, or China.)

Another example: for hanzi/kanji/hanja, I have designed Unidecode to transliterate according to the value that that character has in Mandarin (otherwise Cantonese,...). Some users have complained that applying Unidecode to Japanese produces gibberish.

To make a long story short: transliterating from Japanese is *difficult* and it requires a *lot* of context-sensitivity. If you have text that you're fairly sure is in Japanese, you're going to have to use a Japanese-specific algorithm to transliterate Japanese into ASCII. (And then you can call Unidecode on the output from that-- it is useful for, for example, turning ｆｕｌｌｗｉｄｔｈ characters into their normal (ASCII) forms.

(Note, as of August 2016: I have titanic but tentative plans for making the value of Unihan characters be something you could set parameters for at runtime, in changing the order of "Mandarin else Cantonese else..." in the value retrieval. Currently that preference list is hardwired on my end, at module-build time. Other options I'm considering allowing for: whether the Mandarin and Cantonese values should have the tone numbers on them; whether every Unihan value should have a terminal space; and maybe other clever stuff I haven't thought of yet.)

CAVEATS

If you get really implausible nonsense out of `unidecode(...)`, make sure that the input data really is a utf8 string. See perlunicode and perlunitut.

Unidecode will work disastrously bad on Japanese. That's because Japanese is very very hard. To extend the Unidecode motto, Unidecode is better than nothing, and with Japanese, *just barely!*

On pure Mandarin, Unidecode will frequently give odd values-- that's because a single hanzi can have several readings, and Unidecode only knows what the Unihan database says is the most common one.

THANKS

Thanks to (in only the sloppiest of sorta-chronological order): Jordan Lachler, Harald Tveit Alvestrand, Melissa Axelrod, Abhijit Menon-Sen, Mark-Jason Dominus, Joe Johnston, Conrad Heiney, fileformat.info, Philip Newton, 唐鳳, Tomaž Šolc, Mike Doherty, JT Smith and the MadMongers, Arden Ogg, Craig Copris, David Cusimano, Brendan Byrd, Hex Martin, and *many* other pals who have helped with the ideas or values for Unidecode's transliterations, or whose help has been in the secret F5 tornado that constitutes the internals of Unidecode's implementation.

And thank you to the many people who have encouraged me to plug away at this project. A decade went by before I had any idea that more than about 4 or 5 people were using or getting any value out of Unidecode. I am told that actually my figure was missing some zeroes on the end!

PORTS

Some wonderful people have ported Unidecode to other languages!

- Python: https://pypi.python.org/pypi/Unidecode
- PHP: https://github.com/silverstripe-labs/silverstripe-unidecode
- Ruby: http://www.rubydoc.info/gems/unidecode/1.0.0/frames
- JavaScript: https://www.npmjs.org/package/unidecode
- Java: https://github.com/xuender/unidecode

I can't vouch for the details of each port, but these are clever people, so I'm sure they did a fine job.

SEE ALSO

An article I wrote for *The Perl Journal* about Unidecode: http://interglacial.com/tpj/22/ (**READ IT!**)

Jukka Korpela's http://www.cs.tut.fi/~jkorpela/fui.html8 which is brilliantly useful, and its code is brilliant (so, view source!). I was *kinda* thinking about maybe doing something *sort of* like that for the v2.x versions of Unicode-- but now he's got me convinced that I should go right ahead.

Tom Christiansen's *Perl Unicode Cookbook*, http://www.perl.com/pub/2012/04/perlunicook-standard-preamble.html

Unicode Consortium: http://www.unicode.org/

Searchable Unihan database: http://www.unicode.org/cgi-bin/GetUnihanData.pl

Geoffrey Sampson. 1990. *Writing Systems: A Linguistic Introduction.* ISBN: 0804717567

Randall K. Barry (editor). 1997. *ALA-LC Romanization Tables: Transliteration Schemes for Non-Roman Scripts.* ISBN: 0844409405 [ALA is the American Library Association; LC is the Library of Congress.]

Rupert Snell. 2000. *Beginner's Hindi Script (Teach Yourself Books).* ISBN: 0658009109

LICENSE

Copyright (c) 2001, 2014, 2015, 2016 Sean M. Burke.

Unidecode is distributed under the Perl Artistic License (perlartistic), namely:

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

DISCLAIMER

Much of Text::Unidecode's internal data is based on data from The Unicode Consortium, with which I am unaffiliated. A good deal of the internal data comes from suggestions that have been contributed by people other than myself.

The views and conclusions contained in my software and documentation are my own-- they should not be interpreted as representing official policies, either expressed or implied, of The Unicode Consortium; nor should they be interpreted as necessarily the views or conclusions of people who have contributed to this project.

Moreover, I discourage you from inferring that choices that I've made in Unidecode reflect political or linguistic prejudices on my part. Just because Unidecode doesn't do great on your language, or just because it might seem to do better on some another language, please don't think I'm out to get you!

AUTHOR

Your pal, Sean M. Burke `sburke@cpan.org`

O HAI!

If you're using Unidecode for anything interesting, be cool and email me, I'm always curious what people use this for. (The answers so far have surprised me!)