Uni*de*code!

Sean M. Burke

> I'm holding out but not getting an answer.
> I want to do right by you.
> But I'm finding out that cheating gets it faster.
>
> — Jimmy Eat World, "Get It Faster"

Back in the days, the Latin-1 character set was the de facto standard for the Internet. It was just Latin letters plus a few accents, and that was enough for most of the Western European languages, but it left a whole lot of other languages out in the cold to try to make do with mutually incompatible encodings. Now, we have Unicode, a single character set that can encode all the world's languages, whether they're written in accented Latin letters, like Icelandic or Vietnamese or Navajo, or in some quite different script, like Greek, Armenian, Chinese, Cherokee, or Hindi, to name a few.

In an ideal world, all computers, applications, operating systems, and protocols would have all the fonts and support for turning Unicode data into correctly formatted text on the screen. But the real world is stubbornly less than ideal, and for a very long time we will still have to deal with some systems that can't reliably show much more than US-ASCII. To cope with that fact, I wrote a Perl module called Text::Unidecode, which takes Unicode text using any writing system, and tries to convey it using just US-ASCII. This article is about how complicated that task can be, how it should have taken me years — and how I actually managed it in just a few days.

Basics of Different Scripts

To explain why I made Text::Unidecode work the way it does, I need to describe some basic principles of world writing systems. Dealing with all the writing systems in Unicode has made me appreciate that while they are all superficially quite different, they are mostly just variations on a few basic themes.

Most writing systems basically work on this plan:

• Figure out the sounds in what you're trying to express.

This sometimes involves some arbitrary decisions — such as what to do when a word's pronunciation can change freely (like the fact that "heat" has a "t" sound its own, but put an unstressed vowel after it as in "heat up a bagel" and it sounds like "heed"). Most writing systems, however, end up settling on these points without much bother and often without any conscious thought.

• Possibly toss out distinctions that aren't crucial.

Here, written English tosses out the distinction between stressed and unstressed syllables (so you can't see the difference between a cold *com*-press, and having to com-*press* data). Written Latin tossed out all distinctions between stressed and unstressed vowels, and between short and long. And Hebrew just usually tosses out its vowels altogether. And if you're Cherokee, you toss out all the information about what tones you have in a word, and also toss out some sound distinctions, like between /k/'s and /g/'s.

• Group sounds according to some consistent scheme.

This might mean just breaking between the words, or it might consider where the syllables stop and start, or it might mean both these things.

• Write that out.

This might mean going from a sound to a symbol one at a time and without appeal to context. Or it might involve some context, as with Spanish, where you have to figure in Spanish, "well, I want a

/k/ sound, and I'd normally write that *c*, but it's going before an /i/ sound here, so I need to write it as *qu* instead" (as in "Quito", the city). Or it might mean figuring (as with Japanese kana and Cherokee syllabics) "this whole syllable is /ki/" and looking up the way to write that syllable, in the chart of all possibly whole syllables.

• Have the computer encode it.

Since we're talking about computer text, we also have an additional step: when you "write out" what you mean, by moving your fingers over the keyboard, the computer eventually saves it to disk in some encoding. Surprisingly, the same character on the screen, could be represented in fundamentally different ways, in different encodings.

A good example of this process, in a non-Western writing system and its encoding, is Divehi, the language spoken (and written) in the Maldives, an archipelago southwest of India. Divehi is written right-to-left, with vowels written as marks over or under the consonants. The word "divehi" itself (the Divehi word for "Divehi"!) illustrates how this works.

• Figure out the sounds of what we're writing.

The word we've expressed is represented phonetically as /divehi/.

• Possibly toss out distinctions that aren't crucial.

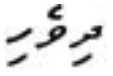
Divehi writing doesn't distinguish stressed syllables from unstressed syllables. But it does happen to distinguish 10 different vowels.

• Group the sounds together.

We note word breaks, since we'll represent those as whitespace later. But more importantly, we group the word's sounds into syllables. /divehi/ becomes three syllables, /di/ /ve/ /hi/.

• Write that out.

We write right-to-left, writing each consonant on the baseline, and each vowel as a symbol above it or below it:



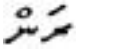
```
      e
H     V     D
i           i
```

Note that the sound /e/ is represented by a mark that looks like a little "c", and which goes over the consonant that starts that syllable. This works fine for syllables that are just a consonant and a vowel, but what about consonants and vowels elsewhere? The word for "that" is just /e/, and how is it written? Divehi wisely uses a placeholder here: a letter that stands for the lack of a preceding consonant.

```
e
X
```

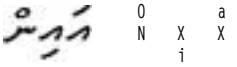
(Here we represent the placeholder consonant in our ad hoc romanization as *X*.)

When a consonant doesn't begin a syllable (as with the /n/ in /ran/, the word for "gold"), it's given a placeholder vowel, which appropriately looks like a little zero:



```
0     a
N     R
```

You can even get words where every syllable has a placeholder of some sort, as in /ain/ (meaning "a school of fish"): it's broken into /a/, /i/, and an extra /n/; the first two get a placeholder consonants, and the last bit gets a placeholder vowel:



Now, there are 10 vowels in the language plus the placeholder *0* for null vowel; and there are 23 consonants plus the placeholder *X* for null consonant. So in the above system, there's a maximum of 11 * 24, or 264 possible written syllables. We could make an encoding based on 264 codepoints (i.e., slots in the character set), where we encode each whole syllable at once. This way makes sense because that's how they're drawn on the screen, so when you want to draw the *di* in the word *divehi*, the *di* is encoded as just a single codepoint, and you fetch the font character for that.

An alternative is to save each element (as *d* is, and as *i* is) as a character on its own, in a character *code* of its own. This is useful in that it reflects how you type, a letter at a time; and if you want to change an element, you shouldn't have to re-key the whole syllable, but should be able to just hit delete and change one element.

It so happens that Unicode's representation of Divehi (in the character codes 0x0780 to 0x07B0) is the latter: *divehi* is represented not as a character *di*, a character *ve*, and a character *hi*; but as six characters:

```
0x078B = Divehi letter "d"
0x07A8 = Divehi letter "i"
0x0788 = Divehi letter "v"
0x07AC = Divehi letter "e"
0x0780 = Divehi letter "h"
0x07A8 = Divehi letter "i"
```

Don't let the issue of writing direction here confuse you: a file consisting just of the word "divehi" would start with the byte sequence for "d" and end with the byte sequence for "i". The fact that Divehi is written "backwards" is just a matter for display on the screen; Unicode doesn't make an issue of this, and encodes things "in logical order", as it's called. (The reader is invited to consider whether all alternatives are illogical orders.)

So if someone emails you in Divehi saying simply "ran!" ("gold!" — maybe it's a grizzled prospector staking claim there in the Indian Ocean), that would be encoded as five characters:

```
0x0783 = Divehi letter "r"
0x07A6 = Divehi letter "a"
0x0782 = Divehi letter "n"
0x07B0 = Divehi letter null vowel
0x0021 = Exclamation mark
```

In an ideal world, you'd get that email, and when it made its way to your mail program, it would look like this:

But unless your mail program (and its OS) knows how to deal with Divehi — which includes having the fonts, knowing how to compose the vowels over/under the consonants, as well as going right to left, then you're more likely to see this:

????!

If that's all we can see, we're left wondering what on Earth is meant by email consisting of an inscrutable four-character word and an exclamation point. Or, maybe the programmer of the mail program was clever, and he has his program show undisplayable characters using their character codes:

[0783][07A6][0782][07B0]!

While this doesn't exactly lose any information, it doesn't really blaze with significance either, unless we have a Unicode character chart on hand. I do have a Unicode character chart — but since it's the size and weight of a large *Encyclopedia Britannica* volume, it's a bit hard to imagine keeping it "on hand" wherever you go.

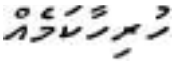
If we're using a system that can't handle all of Unicode, that maybe can't be trusted with anything but US-ASCII, it'd be nice if instead of "????!" or "[0783][07A6][0782][07B0]!", we could just see the Divehi word expressed in Latin letters, as plain old "ran!". That's what I wanted Text::Unidecode to do. And if only all the world's scripts were as simple as Divehi seems so far, then writing Text::Unidecode would have been barely a few day's work. But it turns out that not even Divehi is really as simple as that.

When the Going Gets Messy, The Messy Turn Pro

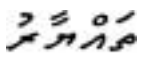
Writing a program to "parse" Divehi characters and spit out Roman letters (what we in the biz call "transliteration") seems merrily simple so far. But it gets stranger.

Previously, I said that there are 264 possible written syllables in Divehi, 11 * 24, 11 for the 10 vowels plus the null vowel, and 24 for 23 consonants plus the null consonant. The astute reader might have realized that this includes the possibility of a syllable consisting of a null consonant and a null vowel. And, a syllable consisting entirely of a placeholder consonant for a placeholder vowel seems the sort of thing that could never actually happen. Sometimes, however, things that don't actually occur are allowed to exist in code tables so that there aren't any gaps in the lookup grid that says "this consonant plus this vowel makes this displayable pair". But the shocking truth is that the Divehi written syllable consisting of null consonant plus null vowel *actually does occur* — in fact it has *two* meanings. This is where things get messy.

The first meaning of this null syllable is to express a sound that has no letter of its own: the glottal stop sound. English has this sound between the two vowels in the interjection "uh-oh!"; but in Divehi it occurs in normal words, like "bo'", which means "frog", or "hurihâkame'" which means "everything" — although it may seem ironic that the word for everything ends in a double nothing:



The second meaning of the null syllable, is to make the following consonant last longer. Long consonants ("geminates" in linguistic jargon) are pretty rare in English; the closest we come is the double-t sound in "cat tail". But they're common in many languages (Italian, Finnish), and come up plenty in Divehi, in words like "bappa" ("father") or "ta-yyâ-ru" ("ready"):



Now, a program that reads the Unicode encoding of this (*taX0y,ru*) should presumably turn it into something like "ta-yyâ-ru", doubling the following consonant (here, a "y"). And where the *X0* syllable occurs at word-end, it should be replaced by some good symbol for the glottal stop sound. The apostrophe will do for that, since it's not otherwise in use in Divehi script.

Another way to express this idea, is that *X0* and a consonant should turn into two of that consonant (*X0y* to *yy); other X0*s should turn into an apostrophe; and otherwise *X* and *0* just delete. This is a snap for regexps:

```
s/X0(\w)/$1$1/g;
s/X0/'/g;
s/(\w)/$Divehi2roman{$1}/g;
```

...except that we can go use the real Unicode characters:

```
# \x{0787}\x{07b0} is "XO", the null syllable
# \p{InThaana} is \w for just Divehi characters
#   ("Thaana" is the official name of the Divehi script)
# See perldoc perlre for more about \p{...}

s/ \x{0787}       # the null consonant
   \x{07b0}       # the null vowel
  (\p{InThaana})  # and some letter
 /$1$1/gx;

s/\x{0787}\x{07b0}/'/g;

s/(\p{InThaana})/$Divehi2roman{$1}/g;
```

Then we just make sure we've filled out `%Divehi2roman` with things like:

```
...
"\x{0786}" => "k",  # Divehi "k" => Roman "r"
"\x{0787}" => "",   # Divehi null consonant => nullstring
"\x{0788}" => "v",  # Divehi "v" => Roman "v"
"\x{0789}" => "m",  # Divehi "m" => Roman "m"
...
"\x{07A6}" => "a",  # Divehi short a => Roman "a"
"\x{07A7}" => "A",  # Divehi long a => Roman "A"
"\x{07A8}" => "i",  # Divehi short i => Roman "i"
"\x{07A9}" => "I",  # Divehi short I => Roman "I"
...
"\x{07b0}" => "",   # Divehi no-vowel => nullstring
```

This constitutes a full working transliterator program, built from three regexps and one hash[1], which does a fine job of turning Unicode text in Divehi into US-ASCII. The fact that the Divehi, in proper script, would have been written right to left, with vowels superimposed on the preceding consonants, doesn't show up in the Unicode representation, so our program doesn't need to deal with it.

As we consider doing the same for the dozens of other scripts in Unicode, we face the unpleasant news that Divehi, for all this strangeness with null syllables, is uncommonly *straightforward* as writing systems go. What took three regexps for Divehi (after a bit of research), could take a dozen for Hindi (which is partly like Divehi, but partly not). As for Thai, the transliterator would have to guess at syllable and word boundaries, since Thai doesn't normally mark them (yesitallrunstogether!). However, you need to know where they are in order to know which way to transliterate some characters.

And it gets worse. My Library of Congress *ALA-LC Romanization Tables* reference for Arabic goes on for pages and pages. It notes, for example, that one character (Unicode 0x0629, called "teh marbuta", which looks oddly like a "ö" is to be transliterated as "h" when it's on nouns that are indefinite or preceded by a definite article, or as "t" when it is on construct state nouns, or as "tan" when it's an adverb suffix. I have not the faintest idea what the "construct state" is or how to identify it, or how to tell an indefinite noun or an adverb from any other kind of word in Arabic. I am rather sure, however, that it *cannot be done with a mere regular expression*, and that is not something I say lightly!

In short, it was looking as if producing a system that could take Unicode text and spit out US-ASCII romanization, was going to involve phenomenal amounts of work. Some scripts are simpler than Divehi, but many are much more complicated. The situation I was facing is exactly the sort of thing that programmers have in mind when they talk about the "eighty-twenty rule".

The Eighty-Twenty Rule

With writing systems and computer encodings of them, things are pretty straightforward most of the time, but still manage to get

a bit messy some of the time, and very messy every now and then. Most of the problem can be treated with a cheap hack or two, but to deal with the rest of the problem, you have to write code that is longer and introduces whole new levels of complexity into your program. In other words, to pick some favorite arbitrary numbers, you can deal with 80% of the problem by writing only about 20% of the code (or expending 20% of the time or effort) that it would take to treat the whole problem.

A non-linguistic example of this is parsing addresses out of "From:" lines in email headers. This is a notoriously and pointlessly hard task to do "by the book" (where "the book" is RFC 2822). However, a quick look at my mail spool file shows that you get 57% of the addresses parsed correctly if you just look for lines matching the pattern `From: Their Name <user@host>`, with a fairly constrained idea of what can be in *user* or *host*. There can be no spaces, no parens, no backslashes, no quotes, no `\@`'s or anything else that actually *is* RFC2822-legal. Add quotes to that pattern, as in `From: "Their Name" <user@host>`, and you get another 25%. Another 11% is gained by `From: user@host`, and it's sharply diminishing returns from there. `From: user@host (Their Name)` is another few percent, and after that it's off into hard-to-parse and mercifully rare things like `From: Pete(A wonderful \) chap) <pete(his account)@silly.test(his host)>`.

So, if you can make do without total coverage, your time is probably better spent with just a simple regexp to match the most common formats. Although generally, your time is *best* spent skimming CPAN and the Perl FAQ first, to see whether someone's done all the work for you already. For problems where that method doesn't turn up anything, you must decide whether you can get away with a quick hack that does most of the work. Doing something by the book is almost always the right thing to do for keeping the code maintainable, extensible, and debuggable. But if doing the right thing means the job will take impossibly long, then it may be time to look into doing the *wrong* thing.

This is exactly the situation I faced when writing Text::Unidecode; I figured out that many scripts (like Cherokee, Amharic, Greek, Coptic, Cyrillic, Armenian, Georgian, Yi, and Korean hangul, to name a few) were almost no problem at all, and could be translated a character at a time by looking in a hash that said that thus-and-such Cyrillic character is to be represented by "b", thus-and-such Cherokee glyph is to be represented everywhere by "la", and so on. Some scripts, like Divehi and the dozen Indic scripts (Hindi, Bengali, Telugu, Burmese, etc.), could *probably* be tackled by a few regexps, with some recourse to advanced regexp features such as lookahead assertions. But, it gets tricky — as the Indic scripts get odder, I probably would be able to learn enough of the language to make sense of its writing system in just a few days. But it would take a serious investment of time, for each of these languages, to learn the language well enough to know whether my transliteration algorithm was doing something wrong.

As I considered going toward harder scripts like Thai and Chinese, and then on to really hard (but also really important) writing systems like Japanese, Hebrew, and Arabic, it looked as though each would require at least months (and probably years) of effort. I like writing open source software, and I like learning languages, but any project that would require me to become literate in Arabic, Hebrew, and all the major languages of Asia clearly needed some scaling back. If nothing else, by the time I finished this (i.e., in the distant future), Unicode would probably finally be well supported, at which point few people would need anything like Text::Unidecode.

One way to tackle this would be to do what I could with the more straightforward writing systems (Cyrillic, etc.), then look for existing algorithms for the harder languages (e.g., using some of

the great work that's already been done toward automatic analysis of Japanese), and encourage friends and friends-of-friends to write algorithms for the languages they personally know well. (This alone would cover a good six or seven main languages of India.) But this would still leave great big gaps; I would run into a good number of languages (probably including Georgian, Syriac, Coptic, and Mongolian Old Script) where I would be unable to find anyone to advise me *and* texts in Unicode to test my transliterator against. In short, doing the right thing either alone or collaboratively would either take a long time, or still be shoddy, or both.

So, I decided that what was feasible was a (relatively) quick hack — a transliteration algorithm whose view of things was inherently *too* simple to work right, but which would still work well *enough* most of the time, and which could still be done in a reasonable amount of time. That would mean that I'd have time to cover all of Unicode. People could (and will!) still produce smart transliterators for any language that they wanted done properly, but Text::Unidecode could take up the slack.

Internals of Text::Unidecode, and Surveying the Damage

I decided that Text::Unidecode should be just a wrapper around this one operation:

```
s/([^\x00-\x7f])/$Unicode2Ascii{$1}/g;
```

This simply replaces every character above 0x007F with `$Unicode2Ascii{`*`that character`*`}`, without any regard to the context. This approach kept Text::Unidecode from being about complex rules formalized as dozens of regexps, and made it just about considering each Unicode glyph and guessing the *one* best way to transliterate it. That worked just fine for "straightforward" scripts (Greek, Cherokee, Cyrillic, etc.), but let's consider how that works for Divehi.

We figured out the null consonant *X* and the null vowel *0* are, by themselves, basically just placeholders, and the only reason they're needed is because in Divehi you can't have a vowel on its own or a consonant on its own. That would make us want to say that the `%Unicode2Ascii` entry for each should just be "", an empty string. But then the special pair *X0* (meaning glottal stop, or meaning to double the next consonant) would disappear without a trace, and *baX0pa*, *boX0*, and *taX0yIâru* would come out as "bapa", "bo", and "tayAru"; whereas we'd prefer something more like "ba**p**pa", "bo'", and "ta**y**yaru".

After much consideration, I decided that the way to do this, for context-insensitive Text::Unidecode, would be to have the null vowel *0* just delete, but have the null consonant *X* be replaced with an apostrophe. To illustrate:

Input in Divehi letters	**Unidecode output**
divehi	*divehi*
ranX	*ran*
baX0pa	*ba'pa*
boX0	*bo'*
taX0y,ru	*ta'yAru*
Xe	*'e*
XiXa	*'i'a*

My criterion here was that I tried to imagine not whether someone who knew no Divehi could make sense of a single word in this transcription, but whether someone who *did* know Divehi could make sense of *several sentences* of this. I figured that while the above system produces the completely superfluous apostrophes in "'e" and "'i'a", people will not see any clear meaning for them. They will see that if you ignore those apostrophes, the word ("e" or "ia")

makes sense in context. In cases where the apostrophe is what's left of an *X0* as in "bo'" and "ba'pa", people should be able to infer its function(s) from context. And after a line or two, it will probably dawn on the Divehi reader that the apostrophe is just a stand-in for the Divehi letter for null consonant.

This means people may have a bit of work to do the first time they see Text::Unidecode output, but it's always work to read text in an alphabet that you're not used to reading it in. In cases where the output actually gets a bit mangled, people will have a harder time, but people are generally pretty good at figuring out what things mean from context, even if it's distorted. In any case, it's *always* better than the entire message being visible only as "???? ??? ???? ?????? ?? ???? ????" or as hex character codes.

Indic scripts are a good example of how something a bit more complex still generally survives the mayhem of Text::Unidecode's algorithm. Indic scripts are basically (in the encoding) like Divehi, except that vowels don't need a null consonant, and you don't bother writing the short "a" vowel. So, if you have a consonant that has no vowel (or no null-vowel!) after it, it must have an "inherent" (implicit) short /a/. So, for example, in the Malâyalam script, the word "malâyalam" is encoded, as *mlyâlm0*. (The final *0* null vowel, really character 0x0D4D, is there to keep the final *m* from being read as "ma".) A context-sensitive algorithm could insert "a" characters as appropriate, but my Text::Unidecode one-context-fits-all approach has to have one representation for the Indic *m* codepoint, in spite of the fact that sometimes it means "m" and sometimes it means "ma".

I observed that most of the time, Indic script *m* means just "m", not "ma", suggesting that I should transliterate it as "m". For the cases where it really did mean "ma", people would usually be able to infer that something was missing, and then be able to imagine what it was. So, if you spoke Malâyalam and you saw a word like "mlyaalm" in Roman script, you would know that "mly" wasn't a possible way to start a word in your language, and you'd infer that something was missing. Context, or even the barest knowledge of the normal written form of the language, would lead you to infer that it's an "a".

Conversely, if I had decided that Indic script *m* should be represented as "ma", *mlyâlm0* would come out as "malayaaalam". That's only slightly strange, but it's very bad for syllables with different vowels. For example, the personal name Abhijît is encoded as *aBijît0*. If we assume every consonant has no "inherent a" (so *m* is "m", not "ma"), then that comes out "abhijiit", or "aBijiit", depending on how we decide to Romanize the "special *b*" — as "B" or as "bh" (I settled on the latter, since it's more standard). But if we decided that every consonant *did* have an "inherent a" (so *m* is "ma" not "m"), *aBijît0* comes out as "abhaijaiita", which is far afield of how anyone thinks of it or pronounces it. The grand lesson here is tht if y lv lttrs ot, ppl cn stll make sense of it, but ifa yaou gao araounada inasaeratainaga laetataerasa, the result is pretty confusing. This rule bodes well for Text::Unidecode output for Arabic and Hebrew, where the normal written form of the language is missing vowels.

The Special Chinese Problem

Most of what I've written so far assumes that each Unicode symbol has pretty much one or two closely related pronunciations, which we then pick from based on context. This assumption falls apart when we get to Korean, Japanese, and Chinese — the languages that use Chinese characters (or "Han" characters, in Unicode jargon). Consider, for example, this character, Unicode 0x4E0B:

If the text is in Korean, this character will be pronounced "ha", and should probably be transliterated as such. If it appears in Chinese text, a Mandarin speaker will pronounce it "xia", and a Cantonese speaker will pronounce it "ha". If that character is in a Japanese text, it will be pronounced as "shita", "shimo", "moto", "ka", or "ge" — and which it is, depends on complex contextual factors.

In Text::Unidecode's table of what transliterates as what, there is no allowance for any kind of context, even contextual guesses about what language the text is in. I had on hand the "Unihan database" that says what the pronunciations are for most of the Chinese characters in Unicode, for all the languages that use each particular character — and I could pick only one pronunciation per character. Somebody had to lose. These are the things I considered:

- More people speak Chinese than Japanese.
- More people speak Japanese than Korean.
- Of people who speak Chinese, most can understand Mandarin pronunciations, as it's the national standard dialect of mainland China.
- The Korean and Japanese pronunciations are often derived from the Chinese pronunciations. It rarely if ever went the other way.
- A Japanese or Korean person is more likely to have studied Chinese (meaning modern spoken Mandarin), than a Chinese person is to have studied Japanese or Korean.
- If the Japanese, Korean, and Mandarin pronunciations are rather different, there is a small chance that a Japanese or Korean reader will be able to understand the Mandarin pronunciation, but nearly no chance that a Chinese person will be able to understand the Korean or Japanese pronunciations.
- Most importantly, if you take the Mandarin pronunciations as the standard, then context-insensitive transliteration of Chinese text works almost perfectly. Japanese needs more context sensitivity, and if you take the most common Japanese pronunciation as the standard, context-insensitive transliteration of Japanese text doesn't work very well.

In other words, if I took Mandarin as the canonical pronunciation for Chinese/Japanese/Korean Unicode characters, it'd do pretty well for a whole lot of text for a whole lot of people. If I chose Japanese as the canonical pronunciation, it'd work less well, and would be good for fewer people. While Text::Unidecode is not necessarily a majoritarian project, I do like to please the most people while frustrating the fewest.

So, that's why when you enter this:

```
use Text::Unidecode;
print unidecode(
  "\x{5317}\x{4EB0}"
   # Those are the Chinese characters for the
   # name of the capital city of mainland China.
);
```

You get this Mandarin Romanization:

```
Bei Jing
```

Instead of Japanese or Korean attempts at transliterations of those characters' pronunciations ("Kita Miyako" and "Pwuk Kyeng", respectively).

Future Developments

I don't think of Text::Unidecode as being the last word on the subject of transliteration — *not by a long shot*. It's a cheap hack whose coverage of world languages is very broad but very *very*

thin. It does pretty well with most alphabets, is so-so with Indic scripts, and its solution to Chinese/Japanese/Korean isn't good, but is the least bad. I hope people will write smart transliterators for the Indic scripts, Thai, Japanese, and whatever else they feel a need for; and I hope they'll put them in CPAN. But for whatever languages are left over, you can always fall back on Text::Unidecode.

References

`perldoc perlunicode` [part of the standard Perl distribution]

Bruce D. Cain and James W. Gair. 2000. *Dhivehi (Maldivian)*. Lincom Europa (Munich).

Randall K. Barry (editor). 1997. *ALA-LC Romanization Tables: Transliteration Schemes for Non-Roman Scripts*. Library of Congress.

Jimmy Eat World. 2001. "Get It Faster", *Bleed American* [Album]. http://www.jimmyeatworld.net

Nara Institute of Science and Technology, Computational Linguistics Laboratory. 1997. *ChaSen v1.0* [a morphological analyzer for Japanese]. http://chasen.aist-nara.ac.jp/

Peter W. Resnick [editor]. 2001. *RFC 2822: Internet Message Format*. http://www.rfc-editor.org/rfc/rfc2822.txt. [Obsoletes RFC 822.]

Geoffrey Sampson. 1990. *Writing Systems: A Linguistic Introduction.* Stanford University Press. [A must-read — SMB]

Rupert Snell. 2000. *Beginner's Hindi Script (Teach Yourself Books).* McGraw Hill NTC.

The Unicode Consortium. 2000. *The Unicode Standard 3.0*. Addison-Wesley.

The Unicode Consortium. 2001. *The Unihan Database v1.1*., Unihan.txt in http://www.unicode.org/Public/UNIDATA/.

1. Incidentally, you may have noticed that while I've been representing long a's as ",", as in "hurihâkame'", the above part of a `%Divehi2roman` would give us a capital instead, as in "hurihAkame'". Since Text::Unidecode is, apparently, for use in systems that lack full Unicode support, I decided to play it safe and assume that they don't even have Latin-1 support either — so I use only US-ASCII characters, involving no Latin-1 characters like "â". In this case, since there's no uppercase/lowercase distinction in Divehi script, nothing on input will give us an "A", so we're free to use that character for long a's.

Sean M. Burke has a Master's in linguistics from Northwestern University.

MODULES USED
Text::Unicode