NAME

I18N::LangTags - functions for dealing with RFC3066-style language tags

SYNOPSIS

```
use I18N::LangTags();
```

...or specify whichever of those functions you want to import, like so:

```
use I18N::LangTags qw(implicate_supers similarity_language_tag);
```

All the exportable functions are listed below — you're free to import only some, or none at all. By default, none are imported. If you say:

```
use I18N::LangTags qw(:ALL)
```

...then all are exported. (This saves you from having to use something less obvious like `use I18N::LangTags qw(/./)`.)

If you don't import any of these functions, assume a `&I18N::LangTags::` in front of all the function names in the following examples.

DESCRIPTION

Language tags are a formalism, described in RFC 3066 (obsoleting 1766), for declaring what language form (language and possibly dialect) a given chunk of information is in.

This library provides functions for common tasks involving language tags as they are needed in a variety of protocols and applications.

Please see the "See Also" references for a thorough explanation of how to correctly use language tags.

- the function is_language_tag($lang1)

  Returns true iff `$lang1` is a formally valid language tag.

  ```
  is_language_tag("fr") is TRUE
  is_language_tag("x-jicarilla") is FALSE
  (Subtags can be 8 chars long at most -- 'jicarilla' is 9)

  is_language_tag("sgn-US") is TRUE
  (That's American Sign Language)

  is_language_tag("i-Klikitat") is TRUE
  (True without regard to the fact noone has actually
  registered Klikitat -- it's a formally valid tag)

  is_language_tag("fr-patois") is TRUE
  (Formally valid -- altho descriptively weak!)

  is_language_tag("Spanish") is FALSE
  is_language_tag("french-patois") is FALSE
  (No good -- first subtag has to match
  /([xXiI]|[a-zA-Z]{2,3})$/ -- see RFC3066)

  is_language_tag("x-borg-prot2532") is TRUE
  (Yes, subtags can contain digits, as of RFC3066)
  ```

- the function extract_language_tags($whatever)

  Returns a list of whatever looks like formally valid language tags in `$whatever`. Not very smart, so don't get too creative with what you want to feed it.

```
extract_language_tags("fr, fr-ca, i-mingo")
returns: ('fr', 'fr-ca', 'i-mingo')

extract_language_tags("It's like this: I'm in fr -- French!")
returns: ('It', 'in', 'fr')
(So don't just feed it any old thing.)
```

The output is untainted. If you don't know what tainting is, don't worry about it.

- the function same_language_tag($lang1, `$lang2`)

  Returns true iff `$lang1` and `$lang2` are acceptable variant tags representing the same language-form.

```
same_language_tag('x-kadara', 'i-kadara') is TRUE
(The x/i- alternation doesn't matter)
same_language_tag('X-KADARA', 'i-kadara') is TRUE
(...and neither does case)
same_language_tag('en', 'en-US') is FALSE
(all-English is not the SAME as US English)
same_language_tag('x-kadara', 'x-kadar') is FALSE
(these are totally unrelated tags)
same_language_tag('no-bok', 'nb') is TRUE
(no-bok is a legacy tag for nb (Norwegian Bokmal))
```

  `same_language_tag` works by just seeing whether `encode_language_tag($lang1)` is the same as `encode_language_tag($lang2)`.

  (Yes, I know this function is named a bit oddly. Call it historic reasons.)

- the function similarity_language_tag($lang1, `$lang2`)

  Returns an integer representing the degree of similarity between tags `$lang1` and `$lang2` (the order of which does not matter), where similarity is the number of common elements on the left, without regard to case and to x/i- alternation.

```
similarity_language_tag('fr', 'fr-ca') is 1
(one element in common)
similarity_language_tag('fr-ca', 'fr-FR') is 1
(one element in common)

similarity_language_tag('fr-CA-joual',
'fr-CA-PEI') is 2
similarity_language_tag('fr-CA-joual', 'fr-CA') is 2
(two elements in common)

similarity_language_tag('x-kadara', 'i-kadara') is 1
(x/i- doesn't matter)

similarity_language_tag('en', 'x-kadar') is 0
similarity_language_tag('x-kadara', 'x-kadar') is 0
(unrelated tags -- no similarity)

similarity_language_tag('i-cree-syllabic',
'i-cherokee-syllabic') is 0
(no B<leftmost> elements in common!)
```

- the function is_dialect_of($lang1, `$lang2`)

  Returns true iff language tag `$lang1` represents a subform of language tag `$lang2`.

Get the order right! It doesn't work the other way around!

```
is_dialect_of('en-US', 'en') is TRUE
(American English IS a dialect of all-English)

is_dialect_of('fr-CA-joual', 'fr-CA') is TRUE
is_dialect_of('fr-CA-joual', 'fr') is TRUE
(Joual is a dialect of (a dialect of) French)

is_dialect_of('en', 'en-US') is FALSE
(all-English is a NOT dialect of American English)

is_dialect_of('fr', 'en-CA') is FALSE

is_dialect_of('en', 'en' ) is TRUE
is_dialect_of('en-US', 'en-US') is TRUE
(B<Note:> these are degenerate cases)

is_dialect_of('i-mingo-tom', 'x-Mingo') is TRUE
(the x/i thing doesn't matter, nor does case)

is_dialect_of('nn', 'no') is TRUE
(because 'nn' (New Norse) is aliased to 'no-nyn',
as a special legacy case, and 'no-nyn' is a
subform of 'no' (Norwegian))
```

- the function super_languages($lang1)

  Returns a list of language tags that are superordinate tags to `$lang1` — it gets this by removing subtags from the end of `$lang1` until nothing (or just ''i'' or ''x'') is left.

  ```
  super_languages("fr-CA-joual") is ("fr-CA", "fr")

  super_languages("en-AU") is ("en")

  super_languages("en") is empty-list, ()

  super_languages("i-cherokee") is empty-list, ()
  ...not ("i"), which would be illegal as well as pointless.
  ```

  If `$lang1` is not a valid language tag, returns empty-list in a list context, undef in a scalar context.

  A notable and rather unavoidable problem with this method: ''x-mingo-tom'' has an ''x'' because the whole tag isn't an IANA-registered tag — but super_languages('x-mingo-tom') is ('x-mingo') — which isn't really right, since 'i-mingo' is registered. But this module has no way of knowing that. (But note that same_language_tag('x-mingo', 'i-mingo') is TRUE.)

  More importantly, you assume *at your peril* that superordinates of `$lang1` are mutually intelligible with `$lang1`. Consider this carefully.

- the function locale2language_tag($locale_identifier)

  This takes a locale name (like ''en'', ''en_US'', or ''en_US.ISO8859-1'') and maps it to a language tag. If it's not mappable (as with, notably, ''C'' and ''POSIX''), this returns empty-list in a list context, or undef in a scalar context.

  ```
  locale2language_tag("en") is "en"

  locale2language_tag("en_US") is "en-US"
  ```

```
locale2language_tag("en_US.ISO8859-1") is "en-US"

locale2language_tag("C") is undef or ()

locale2language_tag("POSIX") is undef or ()

locale2language_tag("POSIX") is undef or ()
```

I'm not totally sure that locale names map satisfactorily to language tags. Think REAL hard about how you use this. YOU HAVE BEEN WARNED.

The output is untainted. If you don't know what tainting is, don't worry about it.

- the function encode_language_tag($lang1)

  This function, if given a language tag, returns an encoding of it such that:

  * tags representing different languages never get the same encoding.

  * tags representing the same language always get the same encoding.

  * an encoding of a formally valid language tag always is a string value that is defined, has length, and is true if considered as a boolean.

  Note that the encoding itself is **not** a formally valid language tag. Note also that you cannot, currently, go from an encoding back to a language tag that it's an encoding of.

  Note also that you **must** consider the encoded value as atomic; i.e., you should not consider it as anything but an opaque, unanalysable string value. (The internals of the encoding method may change in future versions, as the language tagging standard changes over time.)

  `encode_language_tag` returns undef if given anything other than a formally valid language tag.

  The reason `encode_language_tag` exists is because different language tags may represent the same language; this is normally treatable with `same_language_tag`, but consider this situation:

  You have a data file that expresses greetings in different languages. Its format is ''[language tag]=[how to say 'Hello']'', like:

  ```
  en-US=Hiho
  fr=Bonjour
  i-mingo=Hau'
  ```

  And suppose you write a program that reads that file and then runs as a daemon, answering client requests that specify a language tag and then expect the string that says how to greet in that language. So an interaction looks like:

  ```
  greeting-client asks: fr
  greeting-server answers: Bonjour
  ```

  So far so good. But suppose the way you're implementing this is:

```
my %greetings;
die unless open(IN, "<in.dat");
while(<IN>) {
chomp;
next unless /([=]+)=(.+)/s;
my($lang, $expr) = ($1, $2);
$greetings{$lang} = $expr;
}
close(IN);
```

at which point `%greetings` has the contents:

```
"en-US" => "Hiho"
"fr" => "Bonjour"
"i-mingo" => "Hau'"
```

And suppose then that you answer client requests for language `$wanted` by just looking up `$greetings`{$wanted}.

If the client asks for ''fr'', that will look up successfully in `%greetings`, to the value ''Bonjour''. And if the client asks for ''i-mingo'', that will look up successfully in `%greetings`, to the value ''Hau'''.

But if the client asks for ''i-Mingo'' or ''x-mingo'', or ''Fr'', then the lookup in `%greetings` fails. That's the Wrong Thing.

You could instead do lookups on `$wanted` with:

```
use I18N::LangTags qw(same_language_tag);
my $response = '';
foreach my $l2 (keys %greetings) {
if(same_language_tag($wanted, $l2)) {
$response = $greetings{$l2};
last;
}
}
```

But that's rather inefficient. A better way to do it is to start your program with:

```
use I18N::LangTags qw(encode_language_tag);
my %greetings;
die unless open(IN, "<in.dat");
while(<IN>) {
chomp;
next unless /([=]+)=(.+)/s;
my($lang, $expr) = ($1, $2);
$greetings{
encode_language_tag($lang)
} = $expr;
}
close(IN);
```

and then just answer client requests for language `$wanted` by just looking up

```
$greetings{encode_language_tag($wanted)}
```

And that does the Right Thing.

- the function alternate_language_tags($lang1)

  This function, if given a language tag, returns all language tags that are alternate forms of this language tag. (I.e., tags which refer to the same language.) This is meant to handle

legacy tags caused by the minor changes in language tag standards over the years; and the x-/i- alternation is also dealt with.

Note that this function does *not* try to equate new (and never-used, and unusable) ISO639-2 three-letter tags to old (and still in use) ISO639-1 two-letter equivalents — like ‘‘ara’’ -> ‘‘ar’’ — because ‘‘ara’’ has*never* been in use as an In ternet language tag, and RFC 3066 stipulates that it never should be, since a shorter tag (‘‘ar’’) exists.

Examples:

```
alternate_language_tags('no-bok') is ('nb')
alternate_language_tags('nb') is ('no-bok')
alternate_language_tags('he') is ('iw')
alternate_language_tags('iw') is ('he')
alternate_language_tags('i-hakka') is ('zh-hakka', 'x-hakka')
alternate_language_tags('zh-hakka') is ('i-hakka', 'x-hakka')
alternate_language_tags('en') is ()
alternate_language_tags('x-mingo-tom') is ('i-mingo-tom')
alternate_language_tags('x-klikitat') is ('i-klikitat')
alternate_language_tags('i-klikitat') is ('x-klikitat')
```

This function returns empty-list if given anything other than a formally valid language tag.

- the function `@langs` = panic_languages(@accept_languages)

  This function takes a list of 0 or more language tags that constitute a given user's Accept-Language list, and returns a list of tags for *other* (non-super) languages that are probably acceptable to the user, to be used *if all else fails*.

  For example, if a user accepts only 'ca' (Catalan) and 'es' (Spanish), and the documents/interfaces you have available are just in German, Italian, and Chinese, then the user will most likely want the Italian one (and not the Chinese or German one!), instead of getting nothing. So `panic_languages('ca', 'es')` returns a list containing 'it' (Italian).

  English ('en') is *always* in the return list, but whether it's at the very end or not depends on the input languages. This function works by consulting an internal table that stipulates what common languages are ‘‘close’’ to each other.

  A useful construct you might consider using is:

```
@fallbacks = super_languages(@accept_languages);
push @fallbacks, panic_languages(
@accept_languages, @fallbacks,
);
```

- the function implicate_supers(...languages...)

  This takes a list of strings (which are presumed to be language-tags; strings that aren't, are ignored); and after each one, this function inserts super-ordinate forms that don't already appear in the list. The original list, plus these insertions, is returned.

  In other words, it takes this:

```
pt-br de-DE en-US fr pt-br-janeiro
```

  and returns this:

```
pt-br pt de-DE de en-US en fr pt-br-janeiro
```

  This function is most useful in the idiom

```
implicate_supers( I18N::LangTags::Detect::detect() );
```

  (See I18N::LangTags::Detect.)

- the function implicate_supers_strictly(...languages...)

  This works like `implicate_supers` except that the implicated forms are added to the end of the return list.

  In other words, implicate_supers_strictly takes a list of strings (which are presumed to be language-tags; strings that aren't, are ignored) and after the whole given list, it inserts the super-ordinate forms of all given tags, minus any tags that already appear in the input list.

  In other words, it takes this:

  ```
  pt-br de-DE en-US fr pt-br-janeiro
  ```

  and returns this:

  ```
  pt-br de-DE en-US fr pt-br-janeiro pt de en
  ```

  The reason this function has "_strictly" in its name is that when you're processing an Accept-Language list according to the RFCs, if you interpret the RFCs quite strictly, then you would use implicate_supers_strictly, but for normal use (i.e., common-sense use, as far as I'm concerned) you'd use implicate_supers.

ABOUT LOWERCASING

I've considered making all the above functions that output language tags return all those tags strictly in lowercase. Having all your language tags in lowercase does make some things easier. But you might as well just lowercase as you like, or call `encode_language_tag($lang1)` where appropriate.

ABOUT UNICODE PLAINTEXT LANGUAGE TAGS

In some future version of I18N::LangTags, I plan to include support for RFC2482-style language tags — which are basically just normal language tags with their ASCII characters shifted into Plane 14.

SEE ALSO

* I18N::LangTags::List

* RFC 3066, http://www.ietf.org/rfc/rfc3066.txt, "Tags for the Identification of Languages". (Obsoletes RFC 1766)

* RFC 2277, http://www.ietf.org/rfc/rfc2277.txt, "IETF Policy on Character Sets and Languages".

* RFC 2231, http://www.ietf.org/rfc/rfc2231.txt, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations".

* RFC 2482, http://www.ietf.org/rfc/rfc2482.txt, "Language Tagging in Unicode Plain Text".

* Locale::Codes, in http://www.perl.com/CPAN/modules/by-module/Locale/

* ISO 639-2, "Codes for the representation of names of languages", including two-letter and three-letter codes, http://www.loc.gov/standards/iso639-2/php/code_list.php

* The IANA list of registered languages (hopefully up-to-date), http://www.iana.org/assignments/language-tags

COPYRIGHT

AUTHOR

Sean M. Burke `sburke@cpan.org`