NAME

Locale::Maketext - framework for localization

SYNOPSIS

```
package MyProgram;
use strict;
use MyProgram::L10N;
# ...which inherits from Locale::Maketext
my $lh = MyProgram::L10N->get_handle() || die "What language?";
...
# And then any messages your program emits, like:
warn $lh->maketext( "Can't open file [_1]: [_2]\n", $f, $! );
...
```

DESCRIPTION

It is a common feature of applications (whether run directly, or via the Web) for them to be ''localized'' — i.e., for them to a present an English interface to an English-speaker, a German interface to a German-speaker, and so on for all languages it's programmed with. Locale::Maketext is a framework for software localization; it provides you with the tools for organizing and accessing the bits of text and text-processing code that you need for producing localized applications.

In order to make sense of Maketext and how all its components fit together, you should probably go read Locale::Maketext::TPJ13, and *then* read the following documentation.

You may also want to read over the source for `File::Findgrep` and its constituent modules — they are a complete (if small) example application that uses Maketext.

QUICK OVERVIEW

The basic design of Locale::Maketext is object-oriented, and Locale::Maketext is an abstract base class, from which you derive a ''project class''. The project class (with a name like ''TkBocciBall::Localize'' which you then use in your module) is in turn the base class for all the ''language classes'' for your project (with names ''TkBocciBall::Localize::it'' ''TkBocciBall::Localize::en'' ''TkBocciBall::Localize::fr'' etc.).

A language class is a class containing a lexicon of phrases as class data, and possibly also some methods that are of use in interpreting phrases in the lexicon, or otherwise dealing with text in that language.

An object belonging to a language class is called a ''language handle''; it's typically a flyweight object.

The normal course of action is to call:

```
use TkBocciBall::Localize; # the localization project class
$lh = TkBocciBall::Localize->get_handle();
# Depending on the user's locale, etc., this will
# make a language handle from among the classes available,
# and any defaults that you declare.
die "Couldn't make a language handle??" unless $lh;
```

From then on, you use the `maketext` function to access entries in whatever lexicon(s) belong to the language handle you got. So, this:

```
print $lh->maketext("You won!"), "\n";
```

...emits the right text for this language. If the object in `$lh` belongs to class ''TkBocciBall::Localize::fr'' and `%TkBocciBall::Localize::fr::Lexicon` contains `("You won!" => "Tu as gagné!")`, then the above code happily tells the user ''Tu as gagné!''.

METHODS

Locale::Maketext offers a variety of methods, which fall into three categories:

- Methods to do with constructing language handles.
- `maketext` and other methods to do with accessing `%Lexicon` data for a given language handle.
- Methods that you may find it handy to use, from routines of yours that you put in `%Lexicon` entries.

These are covered in the following section.

Construction Methods

These are to do with constructing a language handle:

- `$lh` = YourProjClass->get_handle(...langtags...) || die "lg-handle?";

  This tries loading classes based on the language-tags you give (like `("en-US", "sk", "kon", "es-MX", "ja", "i-klingon")`, and for the first class that succeeds, returns YourProjClass::*language*->*new()*.

  If it runs thru the entire given list of language-tags, and finds no classes for those exact terms, it then tries "superordinate" language classes. So if no "en-US" class (i.e., YourProjClass::en_us) was found, nor classes for anything else in that list, we then try its superordinate, "en" (i.e., YourProjClass::en), and so on thru the other language-tags in the given list: "es". (The other language-tags in our example list: happen to have no superordinates.)

  If none of those language-tags leads to loadable classes, we then try classes derived from YourProjClass->*fallback_languages()* and then if nothing comes of that, we use classes named by YourProjClass->*fallback_language_classes()*. Then in the (probably quite unlikely) event that that fails, we just return undef.

- `$lh` = YourProjClass->get_handle() || die "lg-handle?";

  When `get_handle` is called with an empty parameter list, magic happens:

  If `get_handle` senses that it's running in program that was invoked as a CGI, then it tries to get language-tags out of the environment variable "HTTP_ACCEPT_LANGUAGE", and it pretends that those were the languages passed as parameters to `get_handle`.

  Otherwise (i.e., if not a CGI), this tries various OS-specific ways to get the language-tags for the current locale/language, and then pretends that those were the value(s) passed to `get_handle`.

  Currently this OS-specific stuff consists of looking in the environment variables "LANG" and "LANGUAGE"; and on MSWin machines (where those variables are typically unused), this also tries using the module Win32::Locale to get a language-tag for whatever language/locale is currently selected in the "Regional Settings" (or "International"?) Control Panel. I welcome further suggestions for making this do the Right Thing under other operating systems that support localization.

  If you're using localization in an application that keeps a configuration file, you might consider something like this in your project class:

```
sub get_handle_via_config {
my $class = $_[0];
my $chosen_language = $Config_settings{'language'};
my $lh;
if($chosen_language) {
$lh = $class->get_handle($chosen_language)
|| die "No language handle for \"$chosen_language\""
. " or the like";
} else {
# Config file missing, maybe?
$lh = $class->get_handle()
|| die "Can't get a language handle";
}
return $lh;
}
```

- `$lh` = YourProjClass::langname->*new()*;

  This constructs a language handle. You usually **don't** call this directly, but instead let `get_handle` find a language class to `use` and to then call ->new on.

- `$lh`->*init()*;

  This is called by ->new to initialize newly-constructed language handles. If you define an init method in your class, remember that it's usually considered a good idea to call `$lh`->SUPER::init in it (presumably at the beginning), so that all classes get a chance to initialize a new object however they see fit.

- YourProjClass->*fallback_languages()*

  `get_handle` appends the return value of this to the end of whatever list of languages you pass `get_handle`. Unless you override this method, your project class will inherit Locale::Maketext's `fallback_languages`, which currently returns `('i-default', 'en', 'en-US')`. (''i-default'' is defined in RFC 2277).

  This method (by having it return the name of a language-tag that has an existing language class) can be used for making sure that `get_handle` will always manage to construct a language handle (assuming your language classes are in an appropriate `@INC` directory). Or you can use the next method:

- YourProjClass->*fallback_language_classes()*

  `get_handle` appends the return value of this to the end of the list of classes it will try using. Unless you override this method, your project class will inherit Locale::Maketext's `fallback_language_classes`, which currently returns an empty list, `()`. By setting this to some value (namely, the name of a loadable language class), you can be sure that `get_handle` will always manage to construct a language handle.

The ''maketext'' Method

This is the most important method in Locale::Maketext:

```
$text = $lh->maketext(I<key>, ...parameters for this phrase...);
```

This looks in the `%Lexicon` of the language handle `$lh` and all its superclasses, looking for an entry whose key is the string *key*. Assuming such an entry is found, various things then happen, depending on the value found:

If the value is a scalarref, the scalar is dereferenced and returned (and any parameters are ignored).

If the value is a coderef, we return &$value($lh, ...parameters...).

If the value is a string that *doesn't* look like it's in Bracket Notation, we return it (after replacing

it with a scalarref, in its `%Lexicon`).

If the value *does* look like it's in Bracket Notation, then we compile it into a sub, replace the string in the `%Lexicon` with the new coderef, and then we return &$new_sub($lh, ...parameters...).

Bracket Notation is discussed in a later section. Note that trying to compile a string into Bracket Notation can throw an exception if the string is not syntactically valid (say, by not balancing brackets right.)

Also, calling &$coderef($lh, ...parameters...) can throw any sort of exception (if, say, code in that sub tries to divide by zero). But a very common exception occurs when you have Bracket Notation text that says to call a method ''foo'', but there is no such method. (E.g., You have [qua**tn**,_1,ball]. will throw an exception on trying to call `$lh->qua`**`tn`**($_[1],'ball') — you presumably meant ''quant''.) `maketext` catches these exceptions, but only to make the error message more readable, at which point it rethrows the exception.

An exception *may* be thrown if *key* is not found in any of `$lh`'s `%Lexicon` hashes. What happens if a key is not found, is discussed in a later section, ''Controlling Lookup Failure''.

Note that you might find it useful in some cases to override the `maketext` method with an ''after method'', if you want to translate encodings, or even scripts:

```
package YrProj::zh_cn; # Chinese with PRC-style glyphs
use base ('YrProj::zh_tw'); # Taiwan-style
sub maketext {
my $self = shift(@_);
my $value = $self->maketext(@_);
return Chineeze::taiwan2mainland($value);
}
```

Or you may want to override it with something that traps any exceptions, if that's critical to your program:

```
sub maketext {
my($lh, @stuff) = @_;
my $out;
eval { $out = $lh->SUPER::maketext(@stuff) };
return $out unless $@;
...otherwise deal with the exception...
}
```

Other than those two situations, I don't imagine that it's useful to override the `maketext` method. (If you run into a situation where it is useful, I'd be interested in hearing about it.)

`$lh`->fail_with *or* `$lh`->fail_with(*PARAM*)
`$lh`->failure_handler_auto
: These two methods are discussed in the section ''Controlling Lookup Failure''.

Utility Methods

These are methods that you may find it handy to use, generally from `%Lexicon` routines of yours (whether expressed as Bracket Notation or not).

`$language`->quant($number, `$singular`)
`$language`->quant($number, `$singular`, `$plural`)
`$language`->quant($number, `$singular`, `$plural`, `$negative`)
: This is generally meant to be called from inside Bracket Notation (which is discussed later), as in

```
"Your search matched [quant,_1,document]!"
```

It's for *quantifying* a noun (i.e., saying how much of it there is, while giving the correct form of it). The behavior of this method is handy for English and a few other Western European

languages, and you should override it for languages where it's not suitable. You can feel free to read the source, but the current implementation is basically as this pseudocode describes:

```
if $number is 0 and there's a $negative,
return $negative;
elsif $number is 1,
return "1 $singular";
elsif there's a $plural,
return "$number $plural";
else
return "$number " . $singular . "s";
#
# ...except that we actually call numf to
# stringify $number before returning it.
```

So for English (with Bracket Notation) `"...[quant,_1,file]..."` is fine (for 0 it returns "0 files", for 1 it returns "1 file", and for more it returns "2 files", etc.)

But for "directory", you'd want `"[quant,_1,directory,directories]"` so that our elementary `quant` method doesn't think that the plural of "directory" is "directorys". And you might find that the output may sound better if you specify a negative form, as in:

```
"[quant,_1,file,files,No files] matched your query.\n"
```

Remember to keep in mind verb agreement (or adjectives too, in other languages), as in:

```
"[quant,_1,document] were matched.\n"
```

Because if _1 is one, you get 1 document **were** matched. An acceptable hack here is to do something like this:

```
"[quant,_1,document was, documents were] matched.\n"
```

`$language`->numf($number)

This returns the given number formatted nicely according to this language's conventions. Maketext's default method is mostly to just take the normal string form of the number (applying sprintf "%G" for only very large numbers), and then to add commas as necessary. (Except that we apply `tr/,././` if `$language->{'numf_comma'}` is true; that's a bit of a hack that's useful for languages that express two million as "2.000.000" and not as "2,000,000").

If you want anything fancier, consider overriding this with something that uses Number::Format, or does something else entirely.

Note that numf is called by quant for stringifying all quantifying numbers.

`$language`->numerate($number, `$singular`, `$plural`, `$negative`)

This returns the given noun form which is appropriate for the quantity `$number` according to this language's conventions. `numerate` is used internally by `quant` to quantify nouns. Use it directly — usually from bracket notation — to avoid `quant`'s implicit call to `numf` and output of a numeric quantity.

`$language`->sprintf($format, `@items`)

This is just a wrapper around Perl's normal `sprintf` function. It's provided so that you can use "sprintf" in Bracket Notation:

```
"Couldn't access datanode [sprintf,%10x=[%s],_1,_2]!\n"
```

returning...

```
Couldn't access datanode Stuff=[thangamabob]!
```

`$language->`*language_tag()*
Currently this just takes the last bit of `ref($language)`, turns underscores to dashes, and returns it. So if `$language` is an object of class Hee::HOO::Haw::en_us, `$language->`*language_tag()* returns ''en-us''. (Yes, the usual representation for that language tag is ''en-US'', but case is *never* considered meaningful in language-tag comparison.)

You may override this as you like; Maketext doesn't use it for anything.

`$language->`*encoding()*
Currently this isn't used for anything, but it's provided (with default value of `(ref($language) && $language->{'encoding'})) or "iso-8859-1"`) as a sort of suggestion that it may be useful/necessary to associate encodings with your language handles (whether on a per-class or even per-handle basis.)

Language Handle Attributes and Internals
A language handle is a flyweight object — i.e., it doesn't (necessarily) carry any data of interest, other than just being a member of whatever class it belongs to.

A language handle is implemented as a blessed hash. Subclasses of yours can store whatever data you want in the hash. Currently the only hash entry used by any crucial Maketext method is ''fail'', so feel free to use anything else as you like.

Remember: Don't be afraid to read the Maketext source if there's any point on which this documentation is unclear. This documen tation is vastly longer than the module source itself.

LANGUAGE CLASS HIERARCHIES

These are Locale::Maketext's assumptions about the class hierarchy formed by all your language classes:

- You must have a project base class, which you load, and which you then use as the first argument in the call to YourProjClass->get_handle(...). It should derive (whether directly or indirectly) from Locale::Maketext. It**doesn't matter** ho w you name this class, although assuming this is the localization component of your Super Mega Program, good names for your project class might be SuperMegaProgram::Localization, SuperMegaProgram::L10N, SuperMegaProgram::I18N, SuperMegaProgram::International, or even SuperMegaProgram::Languages or SuperMegaProgram::Messages.
- Language classes are what YourProjClass->get_handle will try to load. It will look for them by taking each language-tag (**skipping** it if it doesn't look like a language-tag or locale-tag!), turning it to all lowercase, turning dashes to underscores, and appending it to YourProjClass . ''::''. So this:

  ```
  $lh = YourProjClass->get_handle(
  'en-US', 'fr', 'kon', 'i-klingon', 'i-klingon-romanized'
  );
  ```

  will try loading the classes YourProjClass::en_us (note lowercase!), YourProjClass::fr, YourProjClass::kon, YourProjClass::i_klingon and YourProjClass::i_klingon_romanized. (And it'll stop at the first one that actually loads.)
- I assume that each language class derives (directly or indirectly) from your project class, and also defines its `@ISA`, its `%Lexicon`, or both. But I anticipate no dire consequences if these assumptions do not hold.
- Language classes may derive from other language classes (although they should have use *Thatclassname*'' or ''use base qw(*...classes...*)). They may derive from the project class. They may derive from some other class altogether. Or via multiple inheritance, it may derive from any mixture of these.
- I foresee no problems with having multiple inheritance in your hierarchy of language classes. (As usual, however, Perl will complain bitterly if you have a cycle in the hierarchy: i.e., if any

class is its own ancestor.)

ENTRIES IN EACH LEXICON

A typical `%Lexicon` entry is meant to signify a phrase, taking some number (0 or more) of parameters. An entry is meant to be accessed by via a string *key* in `$lh->`maketext(*key*, ...parameters...), which should return a string that is generally meant for be used for "output" to the user — regardless of whether this actually means printing to STDOUT, writing to a file, or putting into a GUI widget.

While the key must be a string value (since that's a basic restriction that Perl places on hash keys), the value in the lexicon can currently be of several types: a defined scalar, scalarref, or coderef. The use of these is explained above, in the section 'The "maketext" Method', and Bracket Notation for strings is discussed in the next section.

While you can use arbitrary unique IDs for lexicon keys (like "_min_larger_max_error"), it is often useful for if an entry's key is itself a valid value, like this example error message:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
```

Compare this code that uses an arbitrary ID...

```
die $lh->maketext( "_min_larger_max_error", $min, $max )
if $min > $max;
```

...to this code that uses a key-as-value:

```
die $lh->maketext(
"Minimum ([_1]) is larger than maximum ([_2])!\n",
$min, $max
) if $min > $max;
```

The second is, in short, more readable. In particular, it's obvious that the number of parameters you're feeding to that phrase (two) is the number of parameters that it *wants* to be fed. (Since you see _1 and a _2 being used in the key there.)

Also, once a project is otherwise complete and you start to localize it, you can scrape together all the various keys you use, and pass it to a translator; and then the translator's work will go faster if what he's presented is this:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
=> "", # fill in something here, Jacques!
```

rather than this more cryptic mess:

```
"_min_larger_max_error"
=> "", # fill in something here, Jacques
```

I think that keys as lexicon values makes the completed lexicon entries more readable:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
=> "Le minimum ([_1]) est plus grand que le maximum ([_2])!\n",
```

Also, having valid values as keys becomes very useful if you set up an _AUTO lexicon. _AUTO lexicons are discussed in a later section.

I almost always use keys that are themselves valid lexicon values. One notable exception is when the value is quite long. For example, to get the screenful of data that a command-line program might return when given an unknown switch, I often just use a brief, self-explanatory key such as "_USAGE_MESSAGE". At that point I then go and immediately to define that lexicon entry in the ProjectClass::L10N::en lexicon (since English is always my "project language"):

```
'_USAGE_MESSAGE' => <<'EOSTUFF',
...long long message...
EOSTUFF
```

and then I can use it as:

```
getopt('oDI', \%opts) or die $lh->maketext('_USAGE_MESSAGE');
```

Incidentally, note that each class's `%Lexicon` inherits-and-extends the lexicons in its superclasses. This is not because these are special hashes *per se*, but because you access them via the `maketext` method, which looks for entries across all the `%Lexicon` hashes in a language class *and* all its ancestor classes. (This is because the idea of ''class data'' isn't directly implemented in Perl, but is instead left to individual class-systems to implement as they see fit..)

Note that you may have things stored in a lexicon besides just phrases for output: for example, if your program takes input from the keyboard, asking a ''(Y/N)'' question, you probably need to know what the equivalent of ''Y[es]/N[o]'' is in whatever language. You probably also need to know what the equivalents of the answers ''y'' and ''n'' are. You can store that information in the lexicon (say, under the keys ''~answer_y'' and ''~answer_n'', and the long forms as ''~answer_yes'' and ''~answer_no'', where ''~'' is just an ad-hoc character meant to indicate to programmers/translators that these are not phrases for output).

Or instead of storing this in the language class's lexicon, you can (and, in some cases, really should) represent the same bit of knowledge as code in a method in the language class. (That leaves a tidy distinction between the lexicon as the things we know how to *say*, and the rest of the things in the lexicon class as things that we know how to *do*.) Consider this example of a processor for responses to French ''oui/non'' questions:

```
sub y_or_n {
return undef unless defined $_[1] and length $_[1];
my $answer = lc $_[1]; # smash case
return 1 if $answer eq 'o' or $answer eq 'oui';
return 0 if $answer eq 'n' or $answer eq 'non';
return undef;
}
```

...which you'd then call in a construct like this:

```
my $response;
until(defined $response) {
print $lh->maketext("Open the pod bay door (y/n)? ");
$response = $lh->y_or_n( get_input_from_keyboard_somehow() );
}
if($response) { $pod_bay_door->open() }
else { $pod_bay_door->leave_closed() }
```

Other data worth storing in a lexicon might be things like filenames for language-targetted resources:

```
...
"_main_splash_png"
=> "/styles/en_us/main_splash.png",
"_main_splash_imagemap"
=> "/styles/en_us/main_splash.incl",
"_general_graphics_path"
=> "/styles/en_us/",
"_alert_sound"
=> "/styles/en_us/hey_there.wav",
"_forward_icon"
=> "left_arrow.png",
"_backward_icon"
=> "right_arrow.png",
# In some other languages, left equals
# BACKwards, and right is FOREwards.
...
```

You might want to do the same thing for expressing key bindings or the like (since hardwiring "q" as the binding for the function that quits a screen/menu/program is useful only if your language happens to associate "q" with "quit"!)

BRACKET NOTATION

Bracket Notation is a crucial feature of Locale::Maketext. I mean Bracket Notation to provide a replacement for the use of sprintf formatting. Everything you do with Bracket Notation could be done with a sub block, but bracket notation is meant to be much more concise.

Bracket Notation is a like a miniature "template" system (in the sense of Text::Template, not in the sense of C++ templates), where normal text is passed thru basically as is, but text in special regions is specially interpreted. In Bracket Notation, you use square brackets ("[...]"), not curly braces ("{...}") to note sections that are specially interpreted.

For example, here all the areas that are taken literally are underlined with a "^", and all the in-bracket special regions are underlined with an X:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
 XX  XX
```

When that string is compiled from bracket notation into a real Perl sub, it's basically turned into:

```
sub {
my $lh = $_[0];
my @params = @_;
return join '',
"Minimum (",
...some code here...
") is larger than maximum (",
...some code here...
")!\n",
}
# to be called by $lh->maketext(KEY, params...)
```

In other words, text outside bracket groups is turned into string literals. Text in brackets is rather more complex, and currently follows these rules:

- Bracket groups that are empty, or which consist only of whitespace, are ignored. (Examples: "[]", "[ ]", or a [and a] with returns and/or tabs and/or spaces between them.

  Otherwise, each group is taken to be a comma-separated group of items, and each item is interpreted as follows:

- An item that is *_digits*" or "*_-digits* is interpreted as `$_`[*value*]. I.e., "_1" becomes with `$_`[1], and "_-3" is interpreted as `$_`[-3] (in which case `@_` should have at least three elements in it). Note that `$_`[0] is the language handle, and is typically not named directly.
- An item "_*" is interpreted to mean "all of `@_` except `$_`[0]". I.e.,`@_[1..$#_]`. Note that this is an empty list in the case of calls like `$lh->`maketext(*key*) where there are no parameters (except `$_`[0], the language handle).
- Otherwise, each item is interpreted as a string literal.

The group as a whole is interpreted as follows:

- If the first item in a bracket group looks like a method name, then that group is interpreted like this:

  ```
  $lh->that_method_name(
  ...rest of items in this group...
  ),
  ```

- If the first item in a bracket group is "*", it's taken as shorthand for the so commonly called "quant" method. Similarly, if the first item in a bracket group is "#", it's taken to be

shorthand for "numf".

- If the first item in a bracket group is the empty-string, or "_*" or _*digits*" or "_-*digits*, then that group is interpreted as just the interpolation of all its items:

```
join('',
...rest of items in this group...
),
```

  Examples: "[_1]" and "[,_1]", which are synonymous; and `[,ID-(,_4,-,_2,)]`, which compiles as `join "", "ID-(", $_[4], "-", $_[2], ")"`.

- Otherwise this bracket group is invalid. For example, in the group "[!@#,whatever]", the first item `"!@#"` is neither the empty-string, _*number*", "_-*number*", "_*, nor a valid method name; and so Locale::Maketext will throw an exception of you try compiling an expression containing this bracket group.

Note, incidentally, that items in each group are comma-separated, not `/\s*,\s*/`-separated. That is, you might expect that this bracket group:

```
"Hoohah [foo, _1 , bar ,baz]!"
```

would compile to this:

```
sub {
my $lh = $_[0];
return join '',
"Hoohah ",
$lh->foo( $_[1], "bar", "baz"),
"!",
}
```

But it actually compiles as this:

```
sub {
my $lh = $_[0];
return join '',
"Hoohah ",
$lh->foo(" _1 ", " bar ", "baz"), # note the <space> in " bar "
"!",
}
```

In the notation discussed so far, the characters "[" and "]" are given special meaning, for opening and closing bracket groups, and "," has a special meaning inside bracket groups, where it separates items in the group. This begs the question of how you'd express a literal "[" or "]" in a Bracket Notation string, and how you'd express a literal comma inside a bracket group. For this purpose I've adopted "~" (tilde) as an escape character: "~[" means a literal '[' character anywhere in Bracket Notation (i.e., regardless of whether you're in a bracket group or not), and ditto for "~]" meaning a literal ']', and "~," meaning a literal comma. (Altho "," means a literal comma outside of bracket groups — it's only inside bracket groups that commas are special.)

And on the off chance you need a literal tilde in a bracket expression, you get it with "~~".

Currently, an unescaped "~" before a character other than a bracket or a comma is taken to mean just a "~" and that character. I.e., "~X" means the same as "~~X" — i.e., one literal tilde, and then one literal "X". However, by using "~X", you are assuming that no future version of Maketext will use "~X" as a magic escape sequence. In practice this is not a great problem, since first off you can just write "~~X" and not worry about it; second off, I doubt I'll add lots of new magic characters to bracket notation; and third off, you aren't likely to want literal "~" characters in your messages anyway, since it's not a character with wide use in natural language text.

Brackets must be balanced — every openbracket must have one matching closebracket, and vice versa. So these are all **invalid**:

```
"I ate [quant,_1,rhubarb pie."
"I ate [quant,_1,rhubarb pie[."
"I ate quant,_1,rhubarb pie]."
"I ate quant,_1,rhubarb pie[."
```

Currently, bracket groups do not nest. That is, you **cannot** say:

```
"Foo [bar,baz,[quux,quuux]]\n";
```

If you need a notation that's that powerful, use normal Perl:

```
%Lexicon = (
...
"some_key" => sub {
my $lh = $_[0];
join '',
"Foo ",
$lh->bar('baz', $lh->quux('quuux')),
"\n",
},
...
);
```

Or write the "bar" method so you don't need to pass it the output from calling quux.

I do not anticipate that you will need (or particularly want) to nest bracket groups, but you are welcome to email me with convincing (real-life) arguments to the contrary.

AUTO LEXICONS

If maketext goes to look in an individual `%Lexicon` for an entry for *key* (where *key* does not start with an underscore), and sees none, **but does see** an entry of "_AUTO" => *some_true_value*, then we actually define `$Lexicon{`*key*`}` = *key* right then and there, and then use that value as if it had been there all along. This happens before we even look in any superclass `%Lexicons`!

(This is meant to be somewhat like the AUTOLOAD mechanism in Perl's function call system — or, looked at another way, like the AutoLoader module.)

I can picture all sorts of circumstances where you just do not want lookup to be able to fail (since failing normally means that maketext throws a `die`, although see the next section for greater control over that). But here's one circumstance where _AUTO lexicons are meant to be *especially* useful:

As you're writing an application, you decide as you go what messages you need to emit. Normally you'd go to write this:

```
if(-e $filename) {
go_process_file($filename)
} else {
print qq{Couldn't find file "$filename"!\n};
}
```

but since you anticipate localizing this, you write:

```
use ThisProject::I18N;
my $lh = ThisProject::I18N->get_handle();
# For the moment, assume that things are set up so
# that we load class ThisProject::I18N::en
# and that that's the class that $lh belongs to.
...
if(-e $filename) {
go_process_file($filename)
} else {
print $lh->maketext(
qq{Couldn't find file "[_1]"!\n}, $filename
);
}
```

Now, right after you've just written the above lines, you'd normally have to go open the file ThisProject/I18N/en.pm, and immediately add an entry:

```
"Couldn't find file \"[_1]\"!\n"
=> "Couldn't find file \"[_1]\"!\n",
```

But I consider that somewhat of a distraction from the work of getting the main code working — to say nothing of the fact that I often have to play with the program a few times before I can decide exactly what wording I want in the messages (which in this case would require me to go changing three lines of code: the call to maketext with that key, and then the two lines in ThisProject/I18N/en.pm).

However, if you set "_AUTO => 1" in the `%Lexicon` in, ThisProject/I18N/en.pm (assuming that English (en) is the language that all your programmers will be using for this project's internal message keys), then you don't ever have to go adding lines like this

```
"Couldn't find file \"[_1]\"!\n"
=> "Couldn't find file \"[_1]\"!\n",
```

to ThisProject/I18N/en.pm, because if _AUTO is true there, then just looking for an entry with the key "Couldn't find file "[_1]"!n" in that lexicon will cause it to be added, with that value!

Note that the reason that keys that start with "_" are immune to _AUTO isn't anything generally magical about the underscore character — I just wanted a way to have most lexicon keys be autoable, except for possibly a few, and I arbitrarily decided to use a leading underscore as a signal to distinguish those few.

READONLY LEXICONS

If your lexicon is a tied hash the simple act of caching the compiled value can be fatal.

For example a GDBM_File GDBM_READER tied hash will die with something like:

```
gdbm store returned -1, errno 2, key "..." at ...
```

All you need to do is turn on caching outside of the lexicon hash itself like so:

```
sub init {
my ($lh) = @_;
...
$lh->{'use_external_lex_cache'} = 1;
...
}
```

And then instead of storing the compiled value in the lexicon hash it will store it in `$lh->{'_external_lex_cache'}`

CONTROLLING LOOKUP FAILURE

If you call `$lh->`maketext(*key*, ...parameters...), and there's no entry *key* in `$lh`'s class's `%Lexicon`, nor in the superclass `%Lexicon` hash, *and* if we can't auto-make *key* (because either it

starts with a "_", or because none of its lexicons have `_AUTO => 1,`), then we have failed to find a normal way to maketext *key*. What then happens in these failure conditions, depends on the `$lh` object's "fail" attribute.

If the language handle has no "fail" attribute, maketext will simply throw an exception (i.e., it calls `die`, mentioning the *key* whose lookup failed, and naming the line number where the calling `$lh->`maketext(*key*,...) was.

If the language handle has a "fail" attribute whose value is a coderef, then `$lh->`maketext(*key*,...params...) gives up and calls:

```
return $that_subref->($lh, $key, @params);
```

Otherwise, the "fail" attribute's value should be a string denoting a method name, so that `$lh->`maketext(*key*,...params...) can give up with:

```
return $lh->$that_method_name($phrase, @params);
```

The "fail" attribute can be accessed with the `fail_with` method:

```
# Set to a coderef:
$lh->fail_with( \&failure_handler );

# Set to a method name:
$lh->fail_with( 'failure_method' );

# Set to nothing (i.e., so failure throws a plain exception)
$lh->fail_with( undef );

# Get the current value
$handler = $lh->fail_with();
```

Now, as to what you may want to do with these handlers: Maybe you'd want to log what key failed for what class, and then die. Maybe you don't like `die` and instead you want to send the error message to STDOUT (or wherever) and then merely `exit()`.

Or maybe you don't want to `die` at all! Maybe you could use a handler like this:

```
# Make all lookups fall back onto an English value,
# but only after we log it for later fingerpointing.
my $lh_backup = ThisProject->get_handle('en');
open(LEX_FAIL_LOG, ">>wherever/lex.log") || die "GNAARGH $!";
sub lex_fail {
my($failing_lh, $key, $params) = @_;
print LEX_FAIL_LOG scalar(localtime), "\t",
ref($failing_lh), "\t", $key, "\n";
return $lh_backup->maketext($key,@params);
}
```

Some users have expressed that they think this whole mechanism of having a "fail" attribute at all, seems a rather pointless complication. But I want Locale::Maketext to be usable for software projects of *any* scale and type; and different software projects have different ideas of what the right thing is to do in failure conditions. I could simply say that failure always throws an exception, and that if you want to be careful, you'll just have to wrap every call to `$lh->`maketext in an eval { }. However, I want programmers to reserve the right (via the "fail" attribute) to treat lookup failure as something other than an exception of the same level of severity as a config file being unreadable, or some essential resource being inaccessible.

One possibly useful value for the "fail" attribute is the method name "failure_handler_auto". This is a method defined in the class Locale::Maketext itself. You set it with:

```
$lh->fail_with('failure_handler_auto');
```

Then when you call `$lh->`maketext(*key*, ...parameters...) and there's no *key* in any of those lexicons, maketext gives up with

```
return $lh->failure_handler_auto($key, @params);
```

But failure_handler_auto, instead of dying or anything, compiles `$key`, caching it in

```
$lh->{'failure_lex'}{$key} = $compiled
```

and then calls the compiled value, and returns that. (I.e., if `$key` looks like bracket notation, `$compiled` is a sub, and we return &{$compiled}(@params); but if `$key` is just a plain string, we just return that.)

The effect of using ''failure_auto_handler'' is like an AUTO lexicon, except that it 1) compiles `$key` even if it starts with ''_'', and 2) you have a record in the new hashref `$lh->`{'failure_lex'} of all the keys that have failed for this object. This should avoid your program dying — as long as your keys aren't actually invalid as bracket code, and as long as they don't try calling methods that don't exist.

''failure_auto_handler'' may not be exactly what you want, but I hope it at least shows you that maketext failure can be mitigated in any number of very flexible ways. If you can formalize exactly what you want, you should be able to express that as a failure handler. You can even make it default for every object of a given class, by setting it in that class's init:

```
sub init {
my $lh = $_[0]; # a newborn handle
$lh->SUPER::init();
$lh->fail_with('my_clever_failure_handler');
return;
}
sub my_clever_failure_handler {
...you clever things here...
}
```

HOW TO USE MAKETEXT

Here is a brief checklist on how to use Maketext to localize applications:

- Decide what system you'll use for lexicon keys. If you insist, you can use opaque IDs (if you're nostalgic for `catgets`), but I have better suggestions in the section ''Entries in Each Lexicon'', above. Assuming you opt for meaningful keys that double as values (like ''Minimum ([_1]) is larger than maximum ([_2])!n''), you'll have to settle on what language those should be in. For the sake of argument, I'll call this English, specifically American English, ''en-US''.
- Create a class for your localization project. This is the name of the class that you'll use in the idiom:

  ```
  use Projname::L10N;
  my $lh = Projname::L10N->get_handle(...) || die "Language?";
  ```

  Assuming you call your class Projname::L10N, create a class consisting minimally of:

  ```
  package Projname::L10N;
  use base qw(Locale::Maketext);
  ...any methods you might want all your languages to share...

  # And, assuming you want the base class to be an _AUTO lexicon,
  # as is discussed a few sections up:

  1;
  ```

- Create a class for the language your internal keys are in. Name the class after the language-tag for that language, in lowercase, with dashes changed to underscores. Assuming your project's first language is US English, you should call this Projname::L10N::en_us. It should consist minimally of:

```
package Projname::L10N::en_us;
use base qw(Projname::L10N);
%Lexicon = (
'_AUTO' => 1,
);
1;
```

  (For the rest of this section, I'll assume that this ''first language class'' of Projname::L10N::en_us has _AUTO lexicon.)

- Go and write your program. Everywhere in your program where you would say:

```
print "Foobar $thing stuff\n";
```

  instead do it thru maketext, using no variable interpolation in the key:

```
print $lh->maketext("Foobar [_1] stuff\n", $thing);
```

  If you get tired of constantly saying `print $lh->maketext`, consider making a functional wrapper for it, like so:

```
use Projname::L10N;
use vars qw($lh);
$lh = Projname::L10N->get_handle(...) || die "Language?";
sub pmt (@) { print( $lh->maketext(@_)) }
# "pmt" is short for "Print MakeText"
$Carp::Verbose = 1;
# so if maketext fails, we see made the call to pmt
```

  Besides whole phrases meant for output, anything language-dependent should be put into the class Projname::L10N::en_us, whether as methods, or as lexicon entries — this is discussed in the section ''Entries in Each Lexicon'', above.

- Once the program is otherwise done, and once its localization for the first language works right (via the data and methods in Projname::L10N::en_us), you can get together the data for translation. If your first language lexicon isn't an _AUTO lexicon, then you already have all the messages explicitly in the lexicon (or else you'd be getting exceptions thrown when you call `$lh->`maketext to get messages that aren't in there). But if you were (advisedly) lazy and are using an _AUTO lexicon, then you've got to make a list of all the phrases that you've so far been letting _AUTO generate for you. There are very many ways to assemble such a list. The most straightforward is to simply grep the source for every occurrence of ''maketext'' (or calls to wrappers around it, like the above `pmt` function), and to log the following phrase.

- You may at this point want to consider whether your base class (Projname::L10N), from which all lexicons inherit from (Projname::L10N::en, Projname::L10N::es, etc.), should be an _AUTO lexicon. It may be true that in theory, all needed messages will be in each language class; but in the presumably unlikely or ''impossible'' case of lookup failure, you should consider whether your program should throw an exception, emit text in English (or whatever your project's first language is), or some more complex solution as described in the section ''Controlling Lookup Failure'', above.

- Submit all messages/phrases/etc. to translators.

  (You may, in fact, want to start with localizing to *one* other language at first, if you're not sure that you've properly abstracted the language-dependent parts of your code.)

Translators may request clarification of the situation in which a particular phrase is found. For example, in English we are entirely happy saying *n* files found‘‘, regardless of whether we mean ’’I looked for files, and found *n* of them‘‘ or the rather distinct situation of ’’I looked for something else (like lines in files), and along the way I saw *n* files.‘‘ This may involve rethinking things that you thought quite clear: should ’’Edit‘‘ on a toolbar be a noun (’’editing‘‘) or a verb (’’to edit‘‘)? Is there already a conventionalized way to express that menu option, separate from the target language’s normal word for ’’to edit?

In all cases where the very common phenomenon of quantification (saying *N* files, for **any** value of N) is involved, each translator should make clear what dependencies the number causes in the sentence. In many cases, dependency is limited to words adjacent to the number, in places where you might expect them (I found the-?PLURAL *N* empty-?PLURAL directory-?PLURAL‘‘), but in some cases there are unexpected dependencies (’’I found-?PLURAL ...‘‘!) as well as long-distance dependencies ’’The *N* directory-?PLURAL could not be deleted-?PLURAL!).

Remind the translators to consider the case where N is 0: ‘‘0 files found’’ isn’t exactly natural-sounding in any language, but it may be unacceptable in many — or it may condition special kinds of agreement (similar to English ‘‘I didN’T find ANY files’’).

Remember to ask your translators about numeral formatting in their language, so that you can override the `numf` method as appropriate. Typical variables in number formatting are: what to use as a decimal point (comma? period?); what to use as a thousands separator (space? nonbreaking space? comma? period? small middot? prime? apostrophe?); and even whether the so-called ‘‘thousands separator’’ is actually for every third digit — I’ve heard reports of two hundred thousand being expressible as ‘‘2,00,000’’ for some Indian (Subcontinental) languages, besides the less surprising ‘‘200 000’’, ‘‘200.000’’, ‘‘200,000’’, and ‘‘200’000’’. Also, using a set of numeral glyphs other than the usual ASCII ‘‘0’’-‘‘9’’ might be appreciated, as via `tr/0-9/\x{0966}-\x{096F}/` for getting digits in Devanagari script (for Hindi, Konkani, others).

The basic `quant` method that Locale::Maketext provides should be good for many languages. For some languages, it might be useful to modify it (or its constituent `numerate` method) to take a plural form in the two-argument call to `quant` (as in ‘‘[quant,_1,files]’’) if it’s all-around easier to infer the singular form from the plural, than to infer the plural form from the singular.

But for other languages (as is discussed at length in Locale::Maketext::TPJ13), simple `quant`/`numf` is not enough. For the particularly problematic Slavic languages, what you may need is a method which you provide with the number, the citation form of the noun to quantify, and the case and gender that the sentence’s syntax projects onto that noun slot. The method would then be responsible for determining what grammatical number that numeral projects onto its noun phrase, and what case and gender it may override the normal case and gender with; and then it would look up the noun in a lexicon providing all needed inflected forms.

- You may also wish to discuss with the translators the question of how to relate different subforms of the same language tag, considering how this reacts with `get_handle`’s treatment of these. For example, if a user accepts interfaces in ‘‘en, fr’’, and you have interfaces available in ‘‘en-US’’ and ‘‘fr’’, what should they get? You may wish to resolve this by establishing that ‘‘en’’ and ‘‘en-US’’ are effectively synonymous, by having one class zero-derive from the other.

  For some languages this issue may never come up (Danish is rarely expressed as ‘‘da-DK’’, but instead is just ‘‘da’’). And for other languages, the whole concept of a ‘‘generic’’ form may verge on being uselessly vague, particularly for interfaces involving voice media in forms of Arabic or Chinese.

- Once you've localized your program/site/etc. for all desired languages, be sure to show the result (whether live, or via screenshots) to the translators. Once they approve, make every effort to have it then checked by at least one other speaker of that language. This holds true even when (or especially when) the translation is done by one of your own programmers. Some kinds of systems may be harder to find testers for than others, depending on the amount of domain-specific jargon and concepts involved — it's easier to find people who can tell you whether they approve of your translation for "delete this message" in an email-via-Web interface, than to find people who can give you an informed opinion on your translation for "attribute value" in an XML query tool's interface.

SEE ALSO

I recommend reading all of these:

Locale::Maketext::TPJ13 — my *The Perl Journal* article about Maketext. It explains many important concepts underlying Locale::Maketext's design, and some insight into why Maketext is better than the plain old approach of having message catalogs that are just databases of sprintf formats.

File::Findgrep is a sample application/module that uses Locale::Maketext to localize its messages. For a larger internationalized system, see also Apache::MP3.

I18N::LangTags.

Win32::Locale.

RFC 3066, *Tags for the Identification of Languages*, as at http://sunsite.dk/RFC/rfc/rfc3066.html

RFC 2277, *IETF Policy on Character Sets and Languages* is at http://sunsite.dk/RFC/rfc/rfc2277.html — much of it is just things of interest to protocol designers, but it explains some basic concepts, like the distinction between locales and language-tags.

The manual for GNU `gettext`. The gettext dist is available in `ftp://prep.ai.mit.edu/pub/gnu/` — get a recent gettext tarball and look in its "doc/" directory, there's an easily browsable HTML version in there. The gettext documentation asks lots of questions worth thinking about, even if some of their answers are sometimes wonky, particularly where they start talking about pluralization.

The Locale/Maketext.pm source. Observe that the module is much shorter than its documentation!

COPYRIGHT AND DISCLAIMER

AUTHOR

Sean M. Burke `sburke@cpan.org`