NAME

Locale::Maketext::TPJ13 -- article about software localization

SYNOPSIS

```
# This an article, not a module.
```

DESCRIPTION

The following article by Sean M. Burke and Jordan Lachler first appeared in *The Perl Journal* #13 and is copyright 1999 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

Localization and Perl: gettext breaks, Maketext fixes

by Sean M. Burke and Jordan Lachler

This article points out cases where gettext (a common system for localizing software interfaces — i.e., making them work in the user's language of choice) fails because of basic differences between human languages. This article then describes Maketext, a new system capable of correctly treating these differences.

A Localization Horror Story: It Could Happen To You

> ''There are a number of languages spoken by human beings in this world.''
>
> -- Harald Tveit Alvestrand, in RFC 1766, ''Tags for the Identification of Languages''

Imagine that your task for the day is to localize a piece of software — and luckily for you, the only output the program emits is two messages, like this:

```
I scanned 12 directories.

Your query matched 10 files in 4 directories.
```

So how hard could that be? You look at the code that produces the first item, and it reads:

```
printf("I scanned %g directories.",
$directory_count);
```

You think about that, and realize that it doesn't even work right for English, as it can produce this output:

```
I scanned 1 directories.
```

So you rewrite it to read:

```
printf("I scanned %g %s.",
$directory_count,
$directory_count == 1 ?
"directory" : "directories",
);
```

...which does the Right Thing. (In case you don't recall, ''%g'' is for locale-specific number interpolation, and ''%s'' is for string interpolation.)

But you still have to localize it for all the languages you're producing this software for, so you pull Locale::gettext off of CPAN so you can access the `gettext` C functions you've heard are standard for localization tasks.

And you write:

```
printf(gettext("I scanned %g %s."),
$dir_scan_count,
$dir_scan_count == 1 ?
gettext("directory") : gettext("directories"),
);
```

But you then read in the gettext manual (Drepper, Miller, and Pinard 1995) that this is not a good idea, since how a single word like ''directory'' or ''directories'' is translated may depend on

context — and this is true, since in a case language like German or Russian, you'd may need these words with a different case ending in the first instance (where the word is the object of a verb) than in the second instance, which you haven't even gotten to yet (where the word is the object of a preposition, "in %g directories") — assuming these keep the same syntax when translated into those languages.

So, on the advice of the gettext manual, you rewrite:

```
printf( $dir_scan_count == 1 ?
gettext("I scanned %g directory.") :
gettext("I scanned %g directories."),
$dir_scan_count );
```

So, you email your various translators (the boss decides that the languages du jour are Chinese, Arabic, Russian, and Italian, so you have one translator for each), asking for translations for "I scanned %g directory." and "I scanned %g directories.". When they reply, you'll put that in the lexicons for gettext to use when it localizes your software, so that when the user is running under the "zh" (Chinese) locale, gettext("I scanned %g directory.") will return the appropriate Chinese text, with a "%g" in there where printf can then interpolate $dir_scan.

Your Chinese translator emails right back — he says both of these phrases translate to the same thing in Chinese, because, in linguistic jargon, Chinese "doesn't have number as a grammatical category" — whereas English does. That is, English has grammatical rules that refer to "number", i.e., whether something is grammatically singular or plural; and one of these rules is the one that forces nouns to take a plural suffix (generally "s") when in a plural context, as they are when they follow a number other than "one" (including, oddly enough, "zero"). Chinese has no such rules, and so has just the one phrase where English has two. But, no problem, you can have this one Chinese phrase appear as the translation for the two English phrases in the "zh" gettext lexicon for your program.

Emboldened by this, you dive into the second phrase that your software needs to output: "Your query matched 10 files in 4 directories.". You notice that if you want to treat phrases as indivisible, as the gettext manual wisely advises, you need four cases now, instead of two, to cover the permutations of singular and plural on the two items, $dir_count and $file_count. So you try this:

```
printf( $file_count == 1 ?
( $directory_count == 1 ?
gettext("Your query matched %g file in %g directory.") :
gettext("Your query matched %g file in %g directories.") ) :
( $directory_count == 1 ?
gettext("Your query matched %g files in %g directory.") :
gettext("Your query matched %g files in %g directories.") ),
$file_count, $directory_count,
);
```

(The case of "1 file in 2 [or more] directories" could, I suppose, occur in the case of symlinking or something of the sort.)

It occurs to you that this is not the prettiest code you've ever written, but this seems the way to go. You mail off to the translators asking for translations for these four cases. The Chinese guy replies with the one phrase that these all translate to in Chinese, and that phrase has two "%g"s in it, as it should — but there's a problem. He translates it word-for-word back: "In %g directories contains %g files match your query." The %g slots are in an order reverse to what they are in English. You wonder how you'll get gettext to handle that.

But you put it aside for the moment, and optimistically hope that the other translators won't have this problem, and that their languages will be better behaved — i.e., that they will be just like English.

But the Arabic translator is the next to write back. First off, your code for ''I scanned `%g` directory.'' or ''I scanned `%g` directories.'' assumes there's only singular or plural. But, to use linguistic jargon again, Arabic has grammatical number, like English (but unlike Chinese), but it's a three-term category: singular, dual, and plural. In other words, the way you say ''directory'' depends on whether there's one directory, or *two* of them, or *more than two*of them. Y our test of `($directory == 1)`no longer does the job. And it means that where English's grammatical category of number necessitates only the two permutations of the first sentence based on ''directory [singular]'' and ''directories [plural]'', Arabic has three — and, worse, in the second sentence (''Your query matched `%g` file in `%g` directory.''), where English has four, Arabic has nine. You sense an unwelcome, exponential trend taking shape.

Your Italian translator emails you back and says that ''I searched 0 directories'' (a possible English output of your program) is stilted, and if you think that's fine English, that's your problem, but that *just will not do* in the language of Dante. He insists that where `$directory_count` is 0, your program should produce the Italian text for I *didn't* scan *any* directories.''. And ditto for ''I didn't match any files in any directories'', although he says the last part about ''in any directories should probably just be left off.

You wonder how you'll get gettext to handle this; to accommodate the ways Arabic, Chinese, and Italian deal with numbers in just these few very simple phrases, you need to write code that will ask gettext for different queries depending on whether the numerical values in question are 1, 2, more than 2, or in some cases 0, and you still haven't figured out the problem with the different word order in Chinese.

Then your Russian translator calls on the phone, to *personally* tell you the bad news about how really unpleasant your life is about to become:

Russian, like German or Latin, is an inflectional language; that is, nouns and adjectives have to take endings that depend on their case (i.e., nominative, accusative, genitive, etc...) — which is roughly a matter of what role they have in syntax of the sentence — as well as on the grammatical gender (i.e., masculine, feminine, neuter) and number (i.e., singular or plural) of the noun, as well as on the declension class of the noun. But unlike with most other inflected languages, putting a number-phrase (like ''ten'' or ''forty-three'', or their Arabic numeral equivalents) in front of noun in Russian can change the case and number that noun is, and therefore the endings you have to put on it.

He elaborates: In ''I scanned `%g` directories'', you'd *expect* ''directories'' to be in the accusative case (since it is the direct object in the sentence) and the plural number, except where `$directory_count` is 1, then you'd expect the singular, of course. Just like Latin or German. *But!* Where `$directory_count` % 10 is 1 (''%'' for modulo, remember), assuming `$directory` count is an integer, and except where `$directory_count` % 100 is 11, ''directories'' is forced to become grammatically singular, which means it gets the ending for the accusative singular... You begin to visualize the code it'd take to test for the problem so far, *and still work for Chinese and Arabic and Italian*, and how many gettext items that'd take, but he keeps going... But where `$directory_count` % 10 is 2, 3, or 4 (except where `$directory_count` % 100 is 12, 13, or 14), the word for ''directories'' is forced to be genitive singular — which means another ending... The room begins to spin around you, slowly at first... But with *all other* integer values, since ''directory'' is an inanimate noun, when preceded by a number and in the nominative or accusative cases (as it is here, just your luck!), it does stay plural, but it is forced into the genitive case — yet another ending... And you never hear him get to the part about how you're going to run into similar (but maybe subtly different) problems with other Slavic languages like Polish, because the floor comes up to meet you, and you fade into unconsciousness.

The above cautionary tale relates how an attempt at localization can lead from programmer consternation, to program obfuscation, to a need for sedation. But careful evaluation shows that your choice of tools merely needed further consideration.

The Linguistic View

> ''It is more complicated than you think.''
>
> -- The Eighth Networking Truth, from RFC 1925

The field of Linguistics has expended a great deal of effort over the past century trying to find grammatical patterns which hold across languages; it's been a constant process of people making generalizations that should apply to all languages, only to find out that, all too often, these generalizations fail — sometimes failing for just a few languages, sometimes whole classes of languages, and sometimes nearly every language in the world except English. Broad statistical trends are evident in what the ''average language'' is like as far as what its rules can look like, must look like, and cannot look like. But the ''average language'' is just as unreal a concept as the ''average person'' — it runs up against the fact no language (or person) is, in fact, average. The wisdom of past experience leads us to believe that any given language can do whatever it wants, in any order, with appeal to any kind of grammatical categories wants — case, number, tense, real or metaphoric characteristics of the things that words refer to, arbitrary or predictable classifications of words based on what endings or prefixes they can take, degree or means of certainty about the truth of statements expressed, and so on, ad infinitum.

Mercifully, most localization tasks are a matter of finding ways to translate whole phrases, generally sentences, where the context is relatively set, and where the only variation in content is *usually* in a number being expressed — as in the example sentences above. Translating specific, fully-formed sentences is, in practice, fairly foolproof — which is good, because that's what's in the phrasebooks that so many tourists rely on. Now, a given phrase (whether in a phrasebook or in a gettext lexicon) in one language *might* have a greater or lesser applicability than that phrase's translation into another language — for example, strictly speaking, in Arabic, the ''your'' in ''Your query matched...'' would take a different form depending on whether the user is male or female; so the Arabic translation ''your[feminine] query'' is applicable in fewer cases than the corresponding English phrase, which doesn't distinguish the user's gender. (In practice, it's not feasible to have a program know the user's gender, so the masculine ''you'' in Arabic is usually used, by default.)

But in general, such surprises are rare when entire sentences are being translated, especially when the functional context is restricted to that of a computer interacting with a user either to convey a fact or to prompt for a piece of information. So, for purposes of localization, translation by phrase (generally by sentence) is both the simplest and the least problematic.

Breaking gettext

> ''It Has To Work.''
>
> -- First Networking Truth, RFC 1925

Consider that sentences in a tourist phrasebook are of two types: ones like ''How do I get to the marketplace?'' that don't have any blanks to fill in, and ones like ''How much do these ___ cost?'', where there's one or more blanks to fill in (and these are usually linked to a list of words that you can put in that blank: ''fish'', ''potatoes'', ''tomatoes'', etc.) The ones with no blanks are no problem, but the fill-in-the-blank ones may not be really straightforward. If it's a Swahili phrasebook, for example, the authors probably didn't bother to tell you the complicated ways that the verb ''cost'' changes its inflectional prefix depending on the noun you're putting in the blank. The trader in the marketplace will still understand what you're saying if you say ''how much do these potatoes cost?'' with the wrong inflectional prefix on ''cost''. After all, *you* can't speak proper Swahili, *you're* just a tourist. But while tourists can be stupid, computers are supposed to be smart; the computer should be able to fill in the blank, and still have the results be grammatical.

In other words, a phrasebook entry takes some values as parameters (the things that you fill in the blank or blanks), and provides a value based on these parameters, where the way you get that final value from the given values can, properly speaking, involve an arbitrarily complex series of operations. (In the case of Chinese, it'd be not at all complex, at least in cases like the examples

at the beginning of this article; whereas in the case of Russian it'd be a rather complex series of operations. And in some languages, the complexity could be spread around differently: while the act of putting a number-expression in front of a noun phrase might not be complex by itself, it may change how you have to, for example, inflect a verb elsewhere in the sentence. This is what in syntax is called ''long-distance dependencies''.)

This talk of parameters and arbitrary complexity is just another way to say that an entry in a phrasebook is what in a programming language would be called a ''function''. Just so you don't miss it, this is the crux of this article: *A phrase is a function; a phrasebook is a bunch of functions.*

The reason that using gettext runs into walls (as in the above second-person horror story) is that you're trying to use a string (or worse, a choice among a bunch of strings) to do what you really need a function for — which is futile. Preforming (s)printf interpolation on the strings which you get back from gettext does allow you to do *some* common things passably well... sometimes... sort of; but, to paraphrase what some people say about `csh` script programming, ''it fools you into thinking you can use it for real things, but you can't, and you don't discover this until you've already spent too much time trying, and by then it's too late.''

Replacing gettext

So, what needs to replace gettext is a system that supports lexicons of functions instead of lexicons of strings. An entry in a lexicon from such a system should *not* look like this:

```
"J'ai trouv\xE9 %g fichiers dans %g r\xE9pertoires"
```

[xE9 is e-acute in Latin-1. Some pod renderers would scream if I used the actual character here. — SB]

but instead like this, bearing in mind that this is just a first stab:

```
sub I_found_X1_files_in_X2_directories {
my( $files, $dirs ) = @_[0,1];
$files = sprintf("%g %s", $files,
$files == 1 ? 'fichier' : 'fichiers');
$dirs = sprintf("%g %s", $dirs,
$dirs == 1 ? "r\xE9pertoire" : "r\xE9pertoires");
return "J'ai trouv\xE9 $files dans $dirs.";
}
```

Now, there's no particularly obvious way to store anything but strings in a gettext lexicon; so it looks like we just have to start over and make something better, from scratch. I call my shot at a gettext-replacement system ''Maketext'', or, in CPAN terms, Locale::Maketext.

When designing Maketext, I chose to plan its main features in terms of ''buzzword compliance''. And here are the buzzwords:

Buzzwords: Abstraction and Encapsulation

The complexity of the language you're trying to output a phrase in is entirely abstracted inside (and encapsulated within) the Maketext module for that interface. When you call:

```
print $lang->maketext("You have [quant,_1,piece] of new mail.",
scalar(@messages));
```

you don't know (and in fact can't easily find out) whether this will involve lots of figuring, as in Russian (if `$lang` is a handle to the Russian module), or relatively little, as in Chinese. That kind of abstraction and encapsulation may encourage other pleasant buzzwords like modularization and stratification, depending on what design decisions you make.

Buzzword: Isomorphism

''Isomorphism'' means ''having the same structure or form''; in discussions of program design, the word takes on the special, specific meaning that your implementation of a solution to a problem *has the same structure* as, say, an informal verbal description of the solution, or maybe of the

problem itself. Isomorphism is, all things considered, a good thing — it's what problem-solving (and solution-implementing) should look like.

What's wrong the with gettext-using code like this...

```
printf( $file_count == 1 ?
( $directory_count == 1 ?
"Your query matched %g file in %g directory." :
"Your query matched %g file in %g directories." ) :
( $directory_count == 1 ?
"Your query matched %g files in %g directory." :
"Your query matched %g files in %g directories." ),
$file_count, $directory_count,
);
```

is first off that it's not well abstracted — these ways of testing for grammatical number (as in the expressions like `foo == 1 ? singular_form : plural_form`) should be abstracted to each language module, since how you get grammatical number is language-specific.

But second off, it's not isomorphic — the ''solution'' (i.e., the phrasebook entries) for Chinese maps from these four English phrases to the one Chinese phrase that fits for all of them. In other words, the informal solution would be ''The way to say what you want in Chinese is with the one phrase 'For your question, in Y directories you would find X files'" — and so the implemented solution should be, isomorphically, just a straightforward way to spit out that one phrase, with numerals properly interpolated. It shouldn't have to map from the complexity of other languages to the simplicity of this one.

Buzzword: Inheritance

There's a great deal of reuse possible for sharing of phrases between modules for related dialects, or for sharing of auxiliary functions between related languages. (By ''auxiliary functions'', I mean functions that don't produce phrase-text, but which, say, return an answer to ''does this number require a plural noun after it?''. Such auxiliary functions would be used in the internal logic of functions that actually do produce phrase-text.)

In the case of sharing phrases, consider that you have an interface already localized for American English (probably by having been written with that as the native locale, but that's incidental). Localizing it for UK English should, in practical terms, be just a matter of running it past a British person with the instructions to indicate what few phrases would benefit from a change in spelling or possibly minor rewording. In that case, you should be able to put in the UK English localization module *only* those phrases that are UK-specific, and for all the rest, *inherit* from the American English module. (And I expect this same situation would apply with Brazilian and Continental Portugese, possibly with some *very* closely related languages like Czech and Slovak, and possibly with the slightly different ''versions'' of written Mandarin Chinese, as I hear exist in Taiwan and mainland China.)

As to sharing of auxiliary functions, consider the problem of Russian numbers from the beginning of this article; obviously, you'd want to write only once the hairy code that, given a numeric value, would return some specification of which case and number a given quantified noun should use. But suppose that you discover, while localizing an interface for, say, Ukranian (a Slavic language related to Russian, spoken by several million people, many of whom would be relieved to find that your Web site's or software's interface is available in their language), that the rules in Ukranian are the same as in Russian for quantification, and probably for many other grammatical functions. While there may well be no phrases in common between Russian and Ukranian, you could still choose to have the Ukranian module inherit from the Russian module, just for the sake of inheriting all the various grammatical methods. Or, probably better organizationally, you could move those functions to a module called `_E_Slavic` or something, which Russian and Ukrainian could inherit useful functions from, but which would (presumably) provide no lexicon.

Buzzword: Concision

Okay, concision isn't a buzzword. But it should be, so I decree that as a new buzzword, ''concision'' means that simple common things should be expressible in very few lines (or maybe even just a few characters) of code — call it a special case of ''making simple things easy and hard things possible'', and see also the role it played in the MIDI::Simple language, discussed elsewhere in this issue [TPJ#13].

Consider our first stab at an entry in our ''phrasebook of functions'':

```
sub I_found_X1_files_in_X2_directories {
my( $files, $dirs ) = @_[0,1];
$files = sprintf("%g %s", $files,
$files == 1 ? 'fichier' : 'fichiers');
$dirs = sprintf("%g %s", $dirs,
$dirs == 1 ? "r\xE9pertoire" : "r\xE9pertoires");
return "J'ai trouv\xE9 $files dans $dirs.";
}
```

You may sense that a lexicon (to use a non-committal catch-all term for a collection of things you know how to say, regardless of whether they're phrases or words) consisting of functions *expressed* as above would make for rather long-winded and repetitive code — even if you wisely rewrote this to have quantification (as we call adding a number expression to a noun phrase) be a function called like:

```
sub I_found_X1_files_in_X2_directories {
my( $files, $dirs ) = @_[0,1];
$files = quant($files, "fichier");
$dirs = quant($dirs, "r\xE9pertoire");
return "J'ai trouv\xE9 $files dans $dirs.";
}
```

And you may also sense that you do not want to bother your translators with having to write Perl code — you'd much rather that they spend their *very costly time*on just translation. And this is to say nothing of the near impossibility of finding a commercial translator who would know even simple Perl.

In a first-hack implementation of Maketext, each language-module's lexicon looked like this:

```
%Lexicon = (
"I found %g files in %g directories"
=> sub {
my( $files, $dirs ) = @_[0,1];
$files = quant($files, "fichier");
$dirs = quant($dirs, "r\xE9pertoire");
return "J'ai trouv\xE9 $files dans $dirs.";
},
... and so on with other phrase => sub mappings ...
);
```

but I immediately went looking for some more concise way to basically denote the same phrase-function — a way that would also serve to concisely denote *most* phrase-functions in the lexicon for *most* languages. After much time and even some actual thought, I decided on this system:

* Where a value in a `%Lexicon` hash is a contentful string instead of an anonymous sub (or, conceivably, a coderef), it would be interpreted as a sort of shorthand expression of what the sub does. When accessed for the first time in a session, it is parsed, turned into Perl code, and then eval'd into an anonymous sub; then that sub replaces the original string in that lexicon. (That way, the work of parsing and evaling the shorthand form for a given phrase is done no more than once per session.)

* Calls to `maketext` (as Maketext's main function is called) happen thru a ''language session handle'', notionally very much like an IO handle, in that you open one at the start of the session, and use it for ''sending signals'' to an object in order to have it return the text you want.

So, this:

```
$lang->maketext("You have [quant,_1,piece] of new mail.",
scalar(@messages));
```

basically means this: look in the lexicon for `$lang` (which may inherit from any number of other lexicons), and find the function that we happen to associate with the string ''You have [quant,_1,piece] of new mail'' (which is, and should be, a functioning ''shorthand'' for this function in the native locale — English in this case). If you find such a function, call it with `$lang` as its first parameter (as if it were a method), and then a copy of scalar(@messages) as its second, and then return that value. If that function was found, but was in string shorthand instead of being a fully specified function, parse it and make it into a function before calling it the first time.

* The shorthand uses code in brackets to indicate method calls that should be performed. A full explanation is not in order here, but a few examples will suffice:

```
"You have [quant,_1,piece] of new mail."
```

The above code is shorthand for, and will be interpreted as, this:

```
sub {
my $handle = $_[0];
my(@params) = @_;
return join '',
"You have ",
$handle->quant($params[1], 'piece'),
"of new mail.";
}
```

where ''quant'' is the name of a method you're using to quantify the noun ''piece'' with the number `$params`[0].

A string with no brackety calls, like this:

```
"Your search expression was malformed."
```

is somewhat of a degenerate case, and just gets turned into:

```
sub { return "Your search expression was malformed." }
```

However, not everything you can write in Perl code can be written in the above shorthand system — not by a long shot. For example, consider the Italian translator from the beginning of this article, who wanted the Italian for ''I didn't find any files'' as a special case, instead of ''I found 0 files''. That couldn't be specified (at least not easily or simply) in our shorthand system, and it would have to be written out in full, like this:

```
sub { # pretend the English strings are in Italian
my($handle, $files, $dirs) = @_[0,1,2];
return "I didn't find any files" unless $files;
return join '',
"I found ",
$handle->quant($files, 'file'),
" in ",
$handle->quant($dirs, 'directory'),
".";
}
```

Next to a lexicon full of shorthand code, that sort of sticks out like a sore thumb — but this*is* a

special case, after all; and at least it's possible, if not as concise as usual.

As to how you'd implement the Russian example from the beginning of the article, well, There's More Than One Way To Do It, but it could be something like this (using English words for Russian, just so you know what's going on):

```
"I [quant,_1,directory,accusative] scanned."
```

This shifts the burden of complexity off to the quant method. That method's parameters are: the numeric value it's going to use to quantify something; the Russian word it's going to quantify; and the parameter ''accusative'', which you're using to mean that this sentence's syntax wants a noun in the accusative case there, although that quantification method may have to overrule, for grammatical reasons you may recall from the beginning of this article.

Now, the Russian quant method here is responsible not only for implementing the strange logic necessary for figuring out how Russian number-phrases impose case and number on their noun-phrases, but also for inflecting the Russian word for ''directory''. How that inflection is to be carried out is no small issue, and among the solutions I've seen, some (like variations on a simple lookup in a hash where all possible forms are provided for all necessary words) are straightforward but *can* become cumbersome when you need to inflect more than a few dozen words; and other solutions (like using algorithms to model the inflections, storing only root forms and irregularities) *can* involve more overhead than is justifiable for all but the largest lexicons.

Mercifully, this design decision becomes crucial only in the hairiest of inflected languages, of which Russian is by no means the *worst* case scenario, but is worse than most. Most languages have simpler inflection systems; for example, in English or Swahili, there are generally no more than two possible inflected forms for a given noun (''error/errors''; ''kosa/makosa''), and the rules for producing these forms are fairly simple — or at least, simple rules can be formulated that work for most words, and you can then treat the exceptions as just ''irregular'', at least relative to your ad hoc rules. A simpler inflection system (simpler rules, fewer forms) means that design decisions are less crucial to maintaining sanity, whereas the same decisions could incur overhead-versus-scalability problems in languages like Russian. It may *also* be likely that code (possibly in Perl, as with Lingua::EN::Inflect, for English nouns) has already been written for the language in question, whether simple or complex.

Moreover, a third possibility may even be simpler than anything discussed above: ''Just require that all possible (or at least applicable) forms be provided in the call to the given language's quant method, as in:''

```
"I found [quant,_1,file,files]."
```

That way, quant just has to chose which form it needs, without having to look up or generate anything. While possibly not optimal for Russian, this should work well for most other languages, where quantification is not as complicated an operation.

The Devil in the Details

There's plenty more to Maketext than described above — for example, there's the details of how language tags (''en-US'', ''i-pwn'', ''fi'', etc.) or locale IDs (''en_US'') interact with actual module naming (''BogoQuery/Locale/en_us.pm''), and what magic can ensue; there's the details of how to record (and possibly negotiate) what character encoding Maketext will return text in (UTF8? Latin-1? KOI8?). There's the interesting fact that Maketext is for localization, but nowhere actually has a `use locale;` anywhere in it. For the curious, there's the somewhat frightening details of how I actually implement something like data inheritance so that searches across modules' `%Lexicon` hashes can parallel how Perl implements method inheritance.

And, most importantly, there's all the practical details of how to actually go about deriving from Maketext so you can use it for your interfaces, and the various tools and conventions for starting out and maintaining individual language modules.

That is all covered in the documentation for Locale::Maketext and the modules that come with it, available in CPAN. After having read this article, which covers the why's of Maketext, the

documentation, which covers the how's of it, should be quite straightforward.

The Proof in the Pudding: Localizing Web Sites

Maketext and gettext have a notable difference: gettext is in C, accessible thru C library calls, whereas Maketext is in Perl, and really can't work without a Perl interpreter (although I suppose something like it could be written for C). Accidents of history (and not necessarily lucky ones) have made C++ the most common language for the implementation of applications like word processors, Web browsers, and even many in-house applications like custom query systems. Current conditions make it somewhat unlikely that the next one of any of these kinds of applications will be written in Perl, albeit clearly more for reasons of custom and inertia than out of consideration of what is the right tool for the job.

However, other accidents of history have made Perl a well-accepted language for design of server-side programs (generally in CGI form) for Web site interfaces. Localization of static pages in Web sites is trivial, feasible either with simple language-negotiation features in servers like Apache, or with some kind of server-side inclusions of language-appropriate text into layout templates. However, I think that the localization of Perl-based search systems (or other kinds of dynamic content) in Web sites, be they public or access-restricted, is where Maketext will see the greatest use.

I presume that it would be only the exceptional Web site that gets localized for English *and* Chinese *and* Italian *and* Arabic *and* Russian, to recall the languages from the beginning of this article — to say nothing of German, Spanish, French, Japanese, Finnish, and Hindi, to name a few languages that benefit from large numbers of programmers or Web viewers or both.

However, the ever-increasing internationalization of the Web (whether measured in terms of amount of content, of numbers of content writers or programmers, or of size of content audiences) makes it increasingly likely that the interface to the average Web-based dynamic content service will be localized for two or maybe three languages. It is my hope that Maketext will make that task as simple as possible, and will remove previous barriers to localization for languages dissimilar to English.

__END__

Sean M. Burke (sburke@cpan.org) has a Master's in linguistics from Northwestern University; he specializes in language technology. Jordan Lachler (lachler@unm.edu) is a PhD student in the Department of Linguistics at the University of New Mexico; he specializes in morphology and pedagogy of North American native languages.

References

Alvestrand, Harald Tveit. 1995. *RFC 1766: Tags for the Identification of Languages.* http://www.ietf.org/rfc/rfc1766.txt [Now see RFC 3066.]

Callon, Ross, editor. 1996. *RFC 1925: The Twelve Networking Truths.* http://www.ietf.org/rfc/rfc1925.txt

Drepper, Ulrich, Peter Miller, and François Pinard. 1995-2001. GNU `gettext`. Available in `ftp://prep.ai.mit.edu/pub/gnu/`, with extensive docs in the distribution tarball. [Since I wrote this article in 1998, I now see that the gettext docs are now trying more to come to terms with plurality. Whether useful conclusions have come from it is another question altogether. — SMB, May 2001]

Forbes, Nevill. 1964. *Russian Grammar.* Third Edition, revised by J. C. Dumbreck. Oxford University Press.