The following paper was originally published in the
Proceedings of the Fifth Annual Tcl/Tk Workshop
Boston, Massachusetts, July 1997

Simple Multilingual Support for Tcl

Henry Spencer
SP Systems

Simple Multilingual Support for Tcl

Henry Spencer

SP Systems
henry@zoo.toronto.edu

ABSTRACT

The first, and often largest, step toward internationalization of an interactive application is translation of its output messages, prompts, etc. into the desired language. This is typically done with a ''message catalog'': the program supplies a key of some kind, which is looked up in the catalog to yield the appropriate translation. While this basic approach is impossible to avoid (given the infeasibility of automated translation), the engineering details matter a great deal.

Programmers can cope with almost infinitely messy interfaces if they must (as witness X programming before Tk), but convincing them to use a new interface voluntarily requires ease of use. Ideally, doing it right should be easier than doing it wrong. If there are concrete benefits from doing it right, doing it right can be *slightly* harder than doing it wrong. However, message-catalog facilities are almost always quite a bit harder to use than *puts*. Hence applications typically use message catalogs only when it is explicitly demanded by the requirements, and retrofitting an application to use a message catalog is painful.

A *carefully designed* message-catalog facility can be almost as easy to use as *puts*, and retrofitting it into a program can be relatively easy. The lookup key should be the message in the original language. Simple provisions for nested subtranslations, untranslated substrings, disambiguating tags, and either explicit (call a procedure) or implicit (done as part of output) translation are essential. Doing all this for Tcl takes care but is practical.

Introduction

Building applications that are portable across computers is easy. Building applications that are portable across national, linguistic, and cultural boundaries is not easy. Many issues need to be addressed: character sets, screen layout, language, formatting conventions, etc. Parameterizing an application to automatically adapt itself to all of these things is an unsolved problem, but much can (and should) be done.

The first, most obvious, and arguably worst problem is language. Most applications have words evident everywhere in the user interface, even when the interface is graphical. Even applications which strive for a purely pictorial interface—not as easy as it sounds, not as culturally neutral as one might think, and often not a good idea—typically have words hidden in places like error messages. Other areas of internationalization can cause misunderstandings and problems, but if users simply can't understand what the application is saying, they can't even get that far.

Automatic translation by software is not yet feasible; human languages are too complex and there is too much context required. Translation of an application's messages, prompts, etc. from one language to another is necessarily done by humans. The most obvious technique, simply modifying the program by editing in a new version of the messages, is unappealing because it creates a new version of the program which must then be maintained separately. What is wanted is a level of abstraction which permits the same program to use different sets of messages depending on the user.

The obvious way to do this, found in many existing applications and even some standards, is a ''message catalog'': the program supplies a key of some kind, which is looked up in the catalog to yield the appropriate text. (Whether the catalog contains all versions of the text, with an identifier for the language used as a secondary key, or whether a change of language implies switching to a new catalog, is an issue of terminology rather than basic concept.)

While this basic approach is impossible to avoid (given the infeasibility of automated translation), the engineering details matter a great deal. This became clear when a customer (Cancom Business Networks) asked about the possibility of a French-language version of an existing interactive user interface, which (like so many) had been written in English without any consideration of support for other languages. Some preliminary experiments suggested that careful design of the internal interfaces would make a very large difference in the amount of pain involved.

Upon reflection, the problem generalizes, because a facility which is easy to retrofit is also usually easy to use in the first place. The reverse is less often true, because structural decisions are easier to get right the first time than to fix later, but there is still a correlation: facilities which don't have structural implications are much easier to retrofit and are also somewhat easier to use.

The overall desire was for a facility which would be both easy to retrofit into the existing application and easy to use for future applications. We dubbed the resulting software package ‘‘Transit’’.

Interface Design

Programmers can cope with almost infinitely messy interfaces if they must (as witness X programming before Tk), but simple and easy-to-use interfaces are popular for more reasons than just because programmers are lazy. Programming well is hard enough without adding unnecessary difficulties. As a result, there is great temptation to class almost any difficulty as unnecessary unless there is convincing evidence otherwise.

If you want people to do things right, doing it right should be easier than doing it wrong. If there are concrete benefits from doing it right, doing it right can be *slightly* harder than doing it wrong: most good programmers are willing to take a *little* bit of extra effort for well-defined benefits. But a facility which is painful to use will be used only when it is compulsory.

Message-catalog facilities are almost always painful to use. In C, the usual approach is to make the programmer call a function which returns a string to be used as the format for *printf*. Often the messages must be numbered, because the key used to locate the message is numeric. The possibility that items to be substituted into the string by *printf* might not be in the same order in all languages results in clumsy circumlocutions, notably the XPG3 [1] position specifiers found in Tcl's *format* command (which indicate, by number, which of *printf*'s arguments should be converted by each *printf* conversion specifier).

The obvious conversion of this approach into Tcl isn't any happier. In Tcl, one would at least use a string as the key, but other complications intervene.

Tcl does have a *printf* equivalent, *format*, but it is not frequently used for simple output: Tcl substitutions are a simpler and easier way to get things into strings. Precisely because string manipulation is rather easier in Tcl, Tcl output strings tend to be constructed in more complex ways, often by assembling output a piece at a time rather than relying on a single invocation of *format* to do all the work. Forcing Tcl output back into the C mold would be a significant backward step in ease of use. (Running into this problem in a retrofitting experiment was what triggered deeper thought about the problem.)

Moreover, precisely because Tcl is much more string-oriented, the basic structure of Tcl output is often more complicated. For example, an error message reporting a system error during a file access would typically include both the file name and an indication of the nature of the error. In Tcl, the latter arrives in the application as a string, and would itself need translation (given that both circumstances and good judgement forbid meddling with the Tcl core). This means that building a typical output string can involve multiple calls to a translation facility. Things get even worse when dealing with input abstractions, which can build complex messages out of prompts, current values, defaults, etc.

After considerable thought, Transit evolved as a rather un-C-like facility which is both easier to use and more versatile than the typical C message catalog. We think the result is almost as easy to use as *puts*. Some details are still evolving.

Limitations

Certain issues were ruled to be beyond the scope of Transit, to keep it manageable. We believe that solutions for most of these belong elsewhere anyway.

We decided that we would not get involved in the complications of character sets. Our definition of a ‘‘character’’ is whatever Tcl gives us, and a ‘‘string’’ is just a sequence of those. Transit maps strings to other strings. It ignores questions like how many bits are in a character, and whether it takes one or several of them to put a single symbol on the screen. We assume that the message-catalog writer has adequate tools for composing the catalog, and that if Transit pumps the specified strings out, the results will be as desired. Any special arrangements needed to render the characters of the translated message on the user's screen are handled by other means. (We suspect we will probably end up providing for language-specific setup and teardown strings, at the very least, but the issue hasn't come up yet.)

We assume that no serious changes in the technique of interaction are needed. In particular, Transit ignores the issues of character sets whose text runs right-to-left or vertically. Much of this comes under the heading of character sets, discussed above, but it's possible that some amendments to interactive techniques might also be in order. We simply don't know enough about this to deal with it.

Transit makes no particular attempt to address details of formatting of structured values, e.g. whether the decimal point is written as a period or a comma. We think this is best dealt with by the facilities that are doing the formatting, e.g. Tcl's *format* primitive, and we haven't had enough need for it so far to plunge into it ourselves.

Finally, at the moment we're restricting Transit's environment to Tcl, because that's what our current applications are using. (Variable user environments and slow communications links dictated a text-only lowest-common-denominator interface for this work.) Generalizing this to include Tk is certainly interesting, and we've tried to keep an eye on the possibility, but what we have to talk about now was done for Tcl.

The Programming Interface

Our first decision was that the main programming interface to Transit would be named *puts*. With some trepidation, we decided that Transit would rename the real *puts* and define its own procedure by that name, functioning as a wrapper around the real one. While this did incur some worries, it also eliminated a lot of fiddly little code changes in the retrofitting process. The wrapper does translation of any text sent to *stdin*, *stdout*, or *stderr*, and leaves everything else alone.[1] This avoids interference with file and network I/O that happens to use *puts*.

The second decision, fairly obvious both from the nature of Tcl and from the choice of *puts* as the interface, was that the key used to look up a message in the Transit catalog would simply be the message itself, in the original language of the program's author. Again, this makes retrofitting tremendously easier, but it also has less obvious benefits. It eliminates the need to define a whole new key space, and also the error-prone maintenance of the relationship between that key space and the program. It provides a ‘‘message of last resort’’ for use when the key lookup fails, instead of having to say ‘‘the program wanted to tell you something but I don't know what it was’’; even a message in the wrong language is often better than a mysterious number or no information at all. And last but not least, it makes the program easier to read, to write, and to test.

These two decisions, together, meant that a lot of the code didn't have to change at all. Disregarding issues of initialization, the Transit equivalent of

```
puts "hello, world"
```

is

```
puts "hello, world"
```

As mentioned earlier, there is a problem with messages which are built up out of multiple parts, by substituting substrings into a master string. Some of the parts themselves need to be translated (e.g., error messages) while some shouldn't be (e.g., filenames). Moreover, the translation of the master string needs to be done *before* substitutions are made, because the contents of the substitutions are unpredictable, so only the pre-substitution master string can appear in the message catalog.

The obvious approach—explicitly invoking translation for each translated substring, and then substituting them into a translated master string—is obviously a nuisance, but worse, it doesn't work very well. Because the order of substitutions may change with translation, the substitution really has to be done with a call to *format*, not with Tcl substitutions. This would require substantial rewriting of retrofitted code and would be a considerable nuisance in new code.

Some thought about the problem led to a crucial observation: the translation of the master string doesn't *have* to be done before the substitution, if the substitution is made reversible. The solution that evolved was to mark substrings within the overall string, and parse the markings at translation time. This typically makes it possible to retrofit translation by just inserting suitable markings, which requires no structural changes to the code. Two flavors of substring brackets are needed, one for translated substrings (henceforth usually just ‘‘substrings’’) and one for untranslated substrings (‘‘literals’’).

Much of our existing code uses balanced single quotes to delimit things like filenames in error messages. This made it easy to decide how to bracket literals: using balanced single quotes (` ´) for literals greatly reduced rewriting. So this:

[1] It would be desirable to provide for user selection of translated file descriptors, eventually, but so far the need has not arisen. We included *stdin* out of general paranoia.

```
puts "cannot find file `$name´"
```

results in a message-catalog lookup of "`cannot find file `□´`" (where the □ is used to show the substitution point) and then substitution of the untranslated `$name` into the translation. This does not deal with all cases, but we'll come back to that in a moment.

Translated substrings do need new brackets, especially since *these* brackets must vanish before final output. They have to be reasonably easy to type, because in either retrofitting or new code, there are going to be quite a few of them. We settled on double angle brackets (<< >>) as a distinctive and easy-to-type set of brackets which seldom appear in strings. So this:

```
puts "*** oops: <<$complaint>>"
```

results in `$complaint` being translated and then substituted into the translation of "`*** oops: □`".

So far, this seems to address only two out of four possibilities. We have substring brackets which vanish before final output, and literal brackets which don't. There is obviously a requirement for literal brackets which vanish, for interpolation of numbers etc. And it's conceivable that there will be a need for a substring which happens to be surrounded by single quotes.

To avoid intruding further on the vocabulary of strings, we decided to use combinations of our existing brackets. To force translation of a quoted substring, we put substring brackets inside literal brackets (`<< >>´); the substring brackets vanish but the literal brackets don't. To substitute without translation but with no quotes in the output, we use the opposite combination (<<` ´>>), which vanishes completely from the output. (The translate-quoted combination is reasonably intuitive; the unquoted-literal one is less so, but seemed the simplest choice.)

Translated substrings are scanned recursively in case they have their own bracketed substrings. Literals are *not* scanned recursively. The latter seemed right in itself, but on inspection it also gave us a bonus: the vanishing literal brackets can be used to quote most of the other brackets, if sometimes a bit clumsily (e.g., <<`<<´>> puts << into the output). This also yields a way of putting single quotes around a literal string which might contain closing single quotes used as apostrophes (<<``isn´t´´>> puts `isn´t´ into the output), a problem which otherwise resists simple solution.

Initially, we decided that inability to find a string in the message catalog would simply result in the string being used untranslated, with no error generated. Signalling an error in this case would help debugging, but it would also require that the message catalog contain translations for some pretty vacuous strings, which serve only as containers for translated strings. For example, an error-printing routine contains

```
puts "*** <<$complaint>>"
```

which would require a translation for "`*** □`" if a string with no translation caused an error. This was a valid point, but after some experience, we reversed the decision, with two flourishes.

First, the message catalog can specify a regular expression indicating which strings are too vacuous to require translation. Second, the application can specify a procedure to be called for non-vacuous strings which don't have translations. If the procedure signals an error (as the default one does), that error aborts the translation. If the procedure instead returns a string, then that string is used as the "translation" of the unknown one. Our major application has such a procedure, which logs the problem and then returns the original string enclosed in conspicuous delimiters.

The translation of a (sub)string is not rescanned, so another way to put awkward sequences into the output is to give them names and translate them (e.g., one might define << as the translation of `lb` in all languages, so <<lb>> would put << into the output).

This observation spurred another thought. One disadvantage of using the original message strings as search keys is the possibility of duplicate keys, particularly with short substrings. For example, single-letter abbreviations for commands (e.g. `q` "quit") need to be translated, but can easily be used for different purposes in different places in the program.[2] One can obviously avoid this by using an arbitrary unique string as the key, but this neutralizes most of the advantages of using the original-language message as the key. We decided that a *tag* of the form `#string#`, if found at the beginning of a (sub)string being translated, vanishes before output even if no translation is being done.

With the addition of tagging, it is possible to also do limited input handling using the translation facility, by translating the strings and patterns used for input recognition as well. This doesn't fully solve the problems of multilingual input, but so far it has sufficed for our relatively limited and structured interactions.[3]

[2] We do not address the more difficult problem of collisions between such abbreviations *after* translation; preparation of a translated message catalog is not always easy!

[3] There is a bit of a practical annoyance here because Tcl's *switch* statement, in its usually-preferable form (entire body enclosed in a single set of { }) doesn't do substitutions into its patterns. So far we haven't devised any elegant solution for this.

For these and other purposes, it is useful to have a slightly richer programming interface, with some further primitives that are expected to be less often used. The *translate* procedure does the full translation process on its string argument and returns the result; it takes a `-quote` option which wraps the output in <<` ´>> to prevent further translation. As a convenience for input handling, the *translating* procedure (name chosen because it contains "in") prepends `#input#` to its string argument and then looks the result up in the message catalog. The procedure *untranslatable* specifies a procedure (possibly with arguments) to be called when an untranslatable string is found.

Finally, there needs to be a way to indicate what translation is desired. While almost unlimited complexity is possible here, we've tentatively opted for a very simple solution: *translateto* activates the whole facility, and it takes an argument to indicate the language, which is used to form the filename of a message catalog. (By convention, following various precedents, language "C" means "no translation"; this is not quite the same as not invoking Transit at all, because the various disappearing substring brackets are still stripped out, as are tags.) An optional second argument indicates the directory where the file can be found; failing that, the directory name is obtained from the environment variable `TRANSIT_DIR`. In the interests of localizing performance impact, *translateto* reads the entire message catalog into memory so that lookups can be done quickly.

Message Catalog

For the message catalog itself, we opted for a very simple form with some hooks for future expansion. It's a text file, with the usual text-file conventions of # introducing a comment line, backslash at the end of a line indicating continuation, and empty lines being insignificant.

The message catalog begins with a *prelude*, terminated by a single line containing only "`---`". The prelude contains declarations, one per line, providing overall control; their syntax is that of Tcl commands. After that, the file is a sequence of translations, one per line.

The prelude was originally just a hook for future expansion, with an eye on character-set issues. Now it does have one type of declaration, `vacuous`, for specifying what strings do not need translation. For example, the declaration

```
vacuous {[^a-zA-Z_0-9]*}
```

says that any string which doesn't contain anything alphanumeric does not require translation.

Each translation is a key, followed by "`->`" (possibly surrounded by white space), followed by its translation.[4] Within the key, substring locations are marked using the translate brackets (not necessarily implying a translated substring, just a matter of not inventing yet another syntax) and arbitrary names, which can then be used in the translation. For example:

```
change <<x>> to <<y>>  ->  \
     changez <<x>> à <<y>>
```

This avoids the endless counting of parameters found in schemes (e.g., XPG3) where insertions are numbered rather than named.

Keys and translations can optionally be surrounded by balanced single quotes, to provide for including white space at beginning or end (where it is normally stripped) or having one string or the other include the sequence "`->`".

Experience

Experience with this facility has been rather limited as yet, but so far the decisions seem to be working out reasonably well. The bracket syntax is sometimes clumsy, but overall it's causing much less grief than our earlier experiments with a more C-like model in which everything had to be run through a *format* variant.

Using matched single quotes as the literal brackets saved a lot of rewriting in this code, although that's obviously a function of how one usually composes messages. The normal substring brackets are okay; we haven't used quoted substrings yet. The unquoted-literal brackets (<<` ´>>) see a fair bit of use and are annoyingly clumsy.

A minor annoyance of the bracketing approach is that the brackets foul up string-width calculations in procedures which try to format output (e.g., showing a long list in as many columns as will fit on the screen). So far it's been practical to deal with this by having such procedures invoke *translate* explicitly, do their formatting, and then use unquoted-literal brackets to suppress further translation.

There is obviously a potential problem arising from using brackets that are not *guaranteed* to be absent in normal text. We had one nasty surprise: the first time an error message like

```
`$type´ invalid
```

was substituted into substring brackets in a statement

[4] We're considering declaring that a line ending in "->" is implicitly continued, which would appear to be helpful.

like:

```
puts "*** <<$message>>"
```

the result was:

```
puts "*** <<`$type´ invalid>>"
```

which produced major internal indigestion and a garbled complaint about mismatched brackets. We had to do two things to fix this.

First, we revised Transit's error reporting to minimize garbling. In particular, it was a serious mistake for the error message (which naturally got translated!) to attempt to report that it was looking for `´>>`! Producing a somewhat less informative message turned out to be the easiest way out of this one.

Second, after a bit of floundering we decided to strip leading and trailing white space within substring brackets before using the remaining contents as a key. This lets us write the problematic statement as:

```
puts "*** << $message >>"
```

which evades the problem of the two opening brackets merging. We were a bit concerned that the white-space stripping might cause difficulties, but in practice we've found white space showing up at the ends of translated strings only at the outermost level, e.g. in things like

```
puts "<<$prompt>>: "
```

and since the trailing space is not within any substring brackets, it is exempt from the stripping. (This required minor revisions to the parsing, which originally started out by wrapping the outermost level in brackets to eliminate treating it as a special case!)

We've seen no other problems with nested brackets, even in an application which often has them several levels deep. Some care has been needed to establish conventions for who supplies brackets and who doesn't, but it is immensely convenient to be able to *make* such choices rather than having them dictated by Transit.

There is an obvious problem with using the original messages as keys: it's difficult to systematically enumerate the key space, to be sure you've supplied all necessary translations (especially after the program changes) and that there are no duplicates which need to be tagged. At the moment, all we've come up with is thorough testing, aided by use of an *untranslatable* procedure which logs unknown strings.

For debugging the translation arrangements themselves, it is sometimes desirable to be able to bypass translation in output. This is the dark side of the convenience of having *puts* do translation. The third or fourth time we ran into this, we added a procedure *untranslated*, which invokes the system *puts* without translation, passing any arguments through.

Conclusion

Careful attention to the engineering of a message-catalog facility is important, particularly in Tcl where strings are everywhere and output is built up in complex and flexible ways. Forcing everything into a C-based model, centered on a *printf* equivalent, is awkward for new programs and requires labor-intensive structural changes to old ones. A more sophisticated approach using substring bracketing works much better.

Acknowledgements

Cancom Business Networks first brought up the question of multilingual support, and has been patient while it was sorted out. Although major parts of Transit were developed independently by the author, Cancom was the first guinea pig for it and some of the development necessarily ended up being done on their time; they have graciously agreed that it can remain freeware.

Although Transit is not part of the Shuse account-administration system [2], its first major uses have occurred in connection with Shuse developments.

Doug Berry of Cancom supported and encouraged this work. Ozan Yigit made a number of useful comments, and in particular made a suggestion which eventually turned into the *untranslatable* facility. Toby the cat kept me company during late-night work sessions when everybody else had given up and gone to bed.

Availability

Transit is copyrighted but freely redistributable. It's available for anonymous FTP on *ftp.zoo.toronto.edu* as *pub/transit.tar.Z* .

References

[1] The X/OPEN Group, *X/OPEN Portability Guide*, 3rd edition ("XPG3"), 1990.

[2] Henry Spencer, *Shuse: Multi-Host Account Administration*, in Proceedings of the Tenth Systems Administration Conference (LISA '96), Usenix Association 1996.