# expander

## SYNOPSIS

package require expander 1.0

## DESCRIPTION

The Tcl "subst" command is often used to support a kind of template processing. Given a string with embedded variables or function calls, "subst" will interpolate the variable and function values, returning the new string:

% set greeting "Howdy"  
Howdy  
% proc place {} {return "World"}  
% subst {$greeting, [place]!}  
Howdy, World!  
%

By defining a suitable set of Tcl commands, "subst" can be used to implement a markup language similar to HTML.

The "subst" command is efficient, but it has three drawbacks for this kind of template processing:

* There's no way to identify and process the plain text between two embedded Tcl commands; that makes it difficult to handle plain text in a context-sensitive way.
* Embedded commands are necessarily bracketed by "[" and "]"; it's convenient to be able to choose different brackets in special cases. Someone producing web pages that include a large quantity of Tcl code examples might easily prefer to use "<<" and ">>" as the embedded code delimiters instead.
* There's no easy way to handle incremental input, as one might wish to do when reading data from a socket.

At present, expander solves the first two problems; eventually it will solve the third problem as well.

To begin, create an expander object:

% package require textutil::expander  
1.0  
% ::textutil::expander myexp  
::myexp  
%

The created "::myexp" object can be used to expand text strings containing embedded Tcl commands. By default, embedded commands are delimited by square brackets. Note that expander doesn't attempt to interpolate variables, since variables can be referenced by embedded commands:

% set greeting "Howdy"  
Howdy  
% proc place {} {return "World"}  
% ::myexp expand {[set greeting], [place]!}  
Howdy, World!  
%

### Embedding Macros

An expander macro is simply a Tcl script embedded within a text string. Expander evaluates the script in the global context, and replaces it with its result string. For example,

% set greetings {Howdy Hi "What's up"}  
Howdy Hi "What's up"  
% ::myexp expand {There are many ways to say "Hello, World!":  
[set result {}  
foreach greeting $greetings {  
 append result "$greeting, World!\n"  
}  
set result]  
And that's just a small sample!}  
There are many ways to say "Hello, World!":  
Howdy, World!  
Hi, World!  
What's up, World!  
  
And that's just a small sample!  
%

### Writing Macro Commands

More typically, "macro commands" are used to create a markup language. A macro command is just a Tcl command that returns an output string. For example, expand can be used to implement a generic document markup language that can be retargeted to HTML or any other output format:

% proc bold {} {return "<b>"}  
% proc /bold {} {return "</b>"}  
% ::myexp expand {Some of this text is in [bold]boldface[/bold]}  
Some of this text is in <b>boldface</b>  
%

The above definition of "bold" and "/bold" returns HTML, but such commands can be as complicated as needed; they could, for example, decide what to return based on the desired output format.

### Changing the Expansion Brackets

By default, embedded macros are enclosed in square brackets, "[" and "]". If square brackets need to be included in the output, the input can contain the [lb](#4i7ojhp) and [rb](#2xcytpi) commands. Alternatively, or if square brackets are objectionable for some other reason, the macro expansion brackets can be changed to any pair of non-empty strings.

The [setbrackets](#3whwml4) command changes the brackets permanently. For example, you can write pseudo-html by change them to "<" and ">":

% ::myexp setbrackets < >  
% ::myexp expand {<bold>This is boldface</bold>}  
<b>This is boldface</b>

Alternatively, you can change the expansion brackets temporarily by passing the desired brackets to the [expand](#1y810tw) command:

% ::myexp setbrackets "\[" "\]"  
% ::myexp expand {<bold>This is boldface</bold>} {< >}  
<b>This is boldface</b>  
%

### Customized Macro Expansion

By default, macros are evaluated using the Tcl "uplevel #0" command, so that the embedded code executes in the global context. The application can provide a different evaluation command using [evalcmd](#3j2qqm3); this allows the application to use a safe interpreter, for example, or even to evaluated something other than Tcl code. There is one caveat: to be recognized as valid, a macro must return 1 when passed to Tcl's "info complete" command.

For example, the following code "evaluates" each macro by returning the macro text itself.

proc identity {macro} {return $macro}  
::myexp evalcmd identity

### Using the Context Stack

Often it's desirable to define a pair of macros which operate in some way on the plain text between them. Consider a set of macros for adding footnotes to a web page: one could have implement something like this:

Dr. Pangloss, however, thinks that this is the best of all  
 possible worlds.[footnote "See Candide, by Voltaire"]

The footnote macro would, presumably, assign a number to this footnote and save the text to be formatted later on. However, this solution is ugly if the footnote text is long or should contain additional markup. Consider the following instead:

Dr. Pangloss, however, thinks that this is the best of all  
 possible worlds.[footnote]See [bookTitle "Candide"], by  
 [authorsName "Voltaire"], for more information.[/footnote]

Here the footnote text is contained between footnote and /footnote macros, continues onto a second line, and contains several macros of its own. This is both clearer and more flexible; however, with the features presented so far there's no easy way to do it. That's the purpose of the context stack.

All macro expansion takes place in a particular context. Here, the footnote macro pushes a new context onto the context stack. Then, all expanded text gets placed in that new context. /footnote retrieves it by popping the context. Here's a skeleton implementation of these two macros:

proc footnote {} {  
 ::myexp cpush footnote  
 }  
  
 proc /footnote {} {  
 set footnoteText [::myexp cpop footnote]  
  
 # Save the footnote text, and return an appropriate footnote  
 # number and link.  
 }

The [cpush](#1ksv4uv) command pushes a new context onto the stack; the argument is the context's name. It can be any string, but would typically be the name of the macro itself. Then, [cpop](#35nkun2) verifies that the current context has the expected name, pops it off of the stack, and returns the accumulated text.

Expand provides several other tools related to the context stack. Suppose the first macro in a context pair takes arguments or computes values which the second macro in the pair needs. After calling [cpush](#1ksv4uv), the first macro can define one or more context variables; the second macro can retrieve their values any time before calling [cpop](#35nkun2). For example, suppose the document must specify the footnote number explicitly:

proc footnote {footnoteNumber} {  
 ::myexp cpush footnote  
 ::myexp csave num $footnoteNumber  
 # Return an appropriate link  
 }  
  
 proc /footnote {} {  
 set footnoteNumber [::myexp cget num]  
 set footnoteText [::myexp cpop footnote]  
  
 # Save the footnote text and its footnoteNumber for future  
 # output.  
 }

At times, it might be desirable to define macros that are valid only within a particular context pair; such macros should verify that they are only called within the correct context using either [cis](#26in1rg) or [cname](#lnxbz9).

## TCL COMMANDS

The package defines the following Tcl commands:

expander *name* This command creates a new expander object; name is the name of the object, and becomes a new command. By default, if the name isn't fully qualified, i.e., if it doesn't completely specify the namespace in which to create the new command, the command is created in the caller's current namespace.

## EXPANDER OBJECT COMMANDS

Every expander object will accept the following subcommands:

cappend *text* Appends a string to the output in the current context. This command should rarely be used by macros or application code.

cget *varname* Retrieves the value of variable *varname*, defined in the current context.

cis *cname* Determines whether or not the name of the current context is *cname*.

cname Returns the name of the current context.

cpop *cname* Pops a context from the context stack, returning all accumulated output in that context. The context must be named *cname*, or an error results.

cpush *cname* Pushes a context named *cname* onto the context stack. The context must be popped by [cpop](#35nkun2) before expansion ends or an error results.

cset *varname* *value* Sets variable *varname* to *value* in the current context.

cvar *varname* Retrieves the internal variable name of context variable *varname*; this allows the variable to be passed to commands like **lappend**.

errmode ?*newErrmode*? Sets the macro expansion error mode to one of "nothing", "macro", "error", or "fail"; the default value is "fail". The value determines what the expander does if an error is detected during expansion of a macro.

If the error mode is "fail", the error propagates normally and can be caught or ignored by the application.

If the error mode is "error", the macro expands into a detailed error message, and expansion continues.

If the error mode is "macro", the macro expands to itself; that is, it is passed along to the output unchanged.

If the error mode is "nothing", the macro expands to the empty string, and is effectively ignored.

evalcmd ?*newEvalCmd*? Returns the current evaluation command, which defaults to "uplevel #0". If specified, *newEvalCmd* will be saved for future use and then returned; it must be a Tcl command expecting one additional argument: the macro to evaluate.

expand *inputString* ?*brackets*? Expands the input string, replacing embedded macros with their expanded values, and returns the expanded string.

If *brackets* is given, it must be a list of two strings; the items will be used as the left and right macro expansion bracket sequences for this expansion only.

lb ?*newbracket*? Returns the current value of the right macro expansion bracket; this is for use as or within a macro, when the bracket needs to be included in the output text. If *newbracket* is specified, it becomes the new bracket, and is returned.

rb ?*newbracket*? Returns the current value of the right macro expansion bracket; this is for use as or within a macro, when the bracket needs to be included in the output text. If *newbracket* is specified, it becomes the new bracket, and is returned.

reset Resets all expander settings to their initial values. Unusual results are likely if this command is called from within a call to [expand](#1y810tw).

setbrackets *lbrack* *rbrack* Sets the left and right macro expansion brackets. This command is for use as or within a macro, or to permanently change the bracket definitions. By default, the brackets are "[" and "]", but any non-empty string can be used; for example, "<" and ">" or "(\*" and "\*)" or even "Hello," and "World!".

textcmd ?*newTextCmd*?] Returns the current command for processing polain text, which defaults to the empty string, meaning *identity*. If specified, *newTextCmd* will be saved for future use and then returned; it must be a Tcl command expecting one additional argument: the text to process. The expander object will this command for all plain text it encounters, giving the user of the object the ability to process all plain text in some standard way before writing it to the output. The object expects that the command returns the processed plain text.

**Note** that the combination of *textcmd plaintext* is run through the *evalcmd* for the actual evaluation. In other words, the *textcmd* is treated as a special macro implicitly surrounding all plain text in the template.

## HISTORY

expander was written by William H. Duquette; it is a repackaging of the central algorithm of the [expand](http://www.wjduquette.com/expand) macro processing tool.

Copyright © 2001, by William H. Duquette. All rights reserved.