CHAPTER 10

Quoting Issues and Eval

This chapter describes explicit calls to the interpreter with the `eval` command. An extra round of substitutions is performed that results in some useful effects. The chapter describes the quoting problems with `eval` and the ways to avoid them. The `uplevel` command evaluates commands in a different scope. The `subst` command does substitutions but no command invocation.

This chapter is from *Practical Programming in Tcl and Tk*, 3rd Ed.

http://www.beedub.com/book/

Dynamic evaluation makes Tcl flexible and powerful, but it can be tricky to use properly. The basic idea is that you create a string and then use the `eval` command to interpret that string as a command or a series of commands. Creating program code on the fly is easy with an interpreted language like Tcl, and very hard, if not impossible, with a statically compiled language like C++ or Java. There are several ways that dynamic code evaluation is used in Tcl:

- In some cases, a simple procedure isn't quite good enough, and you need to glue together a command from a few different pieces and then execute the result using `eval`. This often occurs with *wrappers*, which provide a thin layer of functionality over existing commands.
- *Callbacks* are script fragments that are saved and evaluated later in response to some event. Examples include the commands associated with Tk buttons, `fileevent` I/O handlers, and `after` timer handlers. Callbacks are a flexible way to link different parts of an application together.
- You can add new control structures to Tcl using the `uplevel` command. For example, you can write a function that applies a command to each line in a file or each node in a tree.
- You can have a mixture of code and data, and just process the code part with the `subst` command. For example, this is useful in HTML templates described in Chapter 18. There are also some powerful combinations of `subst` and `regsub` described in Chapter 11.

Constructing Code with the `list` Command

It can be tricky to assemble a command so that it is evaluated properly by `eval`. The same difficulties apply to commands like `after`, `uplevel`, and the Tk `send` command, all of which have similar properties to `eval`, except that the command evaluation occurs later or in a different context. Constructing commands dynamically is a source of many problems. The worst part is that you can write code that works sometimes but not others, which can be very confusing.

Use `list` when constructing commands.

The root of the quoting problems is the internal use of `concat` by `eval` and similar commands to concatenate their arguments into one command string. The `concat` can lose some important list structure so that arguments are not passed through as you expect. The general strategy to avoid these problems is to use `list` and `lappend` to explicitly form the command callback as a single, well-structured list.

The `eval` Command

The `eval` command results in another call to the Tcl interpreter. If you construct a command dynamically, you must use `eval` to interpret it. For example, suppose we want to construct the following command now but execute it later:

```
puts stdout "Hello, World!"
```

In this case, it is sufficient to do the following:

```
set cmd {puts stdout "Hello, World!"}
=> puts stdout "Hello, World!"
# sometime later...
eval $cmd
=> Hello, World!
```

In this case, the value of `cmd` is passed to Tcl. All the standard grouping and substitution are done again on the value, which is a `puts` command.

However, suppose that part of the command is stored in a variable, but that variable will not be defined at the time `eval` is used. We can artificially create this situation like this:

```
set string "Hello, World!"
set cmd {puts stdout $string}
=> puts stdout $string
unset string
eval $cmd
=> can't read "string": no such variable
```

In this case, the command contains `$string`. When this is processed by `eval`, the interpreter looks for the current value of `string`, which is undefined. This example is contrived, but the same problem occurs if `string` is a local variable, and `cmd` will be evaluated later in the global scope.

A common mistake is to use double quotes to group the command. That will

let $string be substituted now. However, this works only if string has a simple value, but it fails if the value of string contains spaces or other Tcl special characters:

```
set cmd "puts stdout $string"
=> puts stdout Hello, World!
eval $cmd
=> bad argument "World!": should be "nonewline"
```

The problem is that we have lost some important structure. The identity of $string as a single argument gets lost in the second round of parsing by eval. The solution to this problem is to construct the command using list, as shown in the following example:

Example 10–1 Using list to construct commands.

```
set string "Hello, World!"
set cmd [list puts stdout $string]
=> puts stdout {Hello, World!}
unset string
eval $cmd
=> Hello, World!
```

The trick is that list has formed a list containing three elements: puts, stdout, and the value of string. The substitution of $string occurs before list is called, and list takes care of grouping that value for us. In contrast, using double quotes is equivalent to:

```
set cmd [concat puts stdout $string]
```

Double quotes lose list structure.

The problem here is that concat does not preserve list structure. The main lesson is that you should use list to construct commands if they contain variable values or command results that must be substituted now. If you use double quotes, the values are substituted but you lose proper command structure. If you use curly braces, then values are not substituted until later, which may not be in the right context.

Commands That Concatenate Their Arguments

The uplevel, after and send commands concatenate their arguments into a command and execute it later in a different context. The uplevel command is described on page 130, after is described on page 218, and send is described on page 560. Whenever I discover such a command, I put it on my danger list and make sure I explicitly form a single command argument with list instead of letting the command concat items for me. Get in the habit now:

```
after 100 [list doCmd $param1 $param2]
send $interp [list doCmd $param1 $param2];# Safe!
```

The danger here is that concat and list can result in the same thing, so

you can be led down the rosy garden path only to get errors later when values change. The two previous examples always work. The next two work only if `param1` and `param2` have values that are single list elements:

```
after 100 doCmd $param1 $param2
send $interp doCmd $param1 $param2;# Unsafe!
```

If you use other Tcl extensions that provide `eval`-like functionality, carefully check their documentation to see whether they contain commands that `concat` their arguments into a command. For example, Tcl-DP, which provides a network version of `send`, `dp_send`, also uses `concat`.

Commands That Use Callbacks

The general strategy of passing out a command or script to call later is a flexible way to assemble different parts of an application, and it is widely used by Tcl commands. Examples include commands that are called when users click on Tk buttons, commands that are called when I/O channels have data ready, or commands that are called when clients connect to network servers. It is also easy to write your own procedures or C extensions that accept scripts and call them later in response to some event.

These other callback situations may not appear to have the "`concat` problem" because they take a single script argument. However, as soon as you use double quotes to group that argument, you have created the `concat` problem all over again. So, all the caveats about using `list` to construct these commands still apply.

Command Prefix Callbacks

There is a variation on command callbacks called a *command prefix*. In this case, the command is given additional arguments when it is invoked. In other words, you provide only part of the command, the command prefix, and the module that invokes the callback adds additional arguments before using `eval` to invoke the command.

For example, when you create a network server, you supply a procedure that is called when a client makes a connection. That procedure is called with three additional arguments that indicate the client's socket, IP address, and port number. This is described in more detail on page 227. The tricky thing is that you can define your callback procedure to take four (or more) arguments. In this case you specify some of the parameters when you define the callback, and then the socket subsystem specifies the remaining arguments when it makes the callback. The following command creates the server side of a socket:

```
set virtualhost www.beedub.com
socket -server [list Accept $virtualhost] 8080
```

However, you define the `Accept` procedure like this:

```
proc Accept {myname sock ipaddr port} { ... }
```

The `myname` parameter is set when you construct the command prefix. The

remaining parameters are set when the callback is invoked. The use of list in this example is not strictly necessary because "we know" that virtualhost will always be a single list element. However, using list is just a good habit when forming callbacks, so I always write the code this way.

There are many other examples of callback arguments that are really command prefixes. Some of these include the scrolling callbacks between Tk scrollbars and their widgets, the command aliases used with Safe Tcl, the sorting functions in lsort, and the completion callback used with fcopy. Example 13–6 on page 181 shows how to use eval to make callbacks from Tcl procedures.

Constructing Procedures Dynamically

The previous examples have all focused on creating single commands by using list operations. Suppose you want to create a whole procedure dynamically. Unfortunately, this can be particularly awkward because a procedure body is not a simple list. Instead, it is a sequence of commands that are each lists, but they are separated by newlines or semicolons. In turn, some of those commands may be loops and if commands that have their own command bodies. To further compound the problem, you typically have two kinds of variables in the procedure body: some that are to be used as values when constructing the body, and some that are to be used later when executing the procedure. The result can be very messy.

The main trick to this problem is to use either format or regsub to process a template for your dynamically generated procedure. If you use format, then you can put %s into your templates where you want to insert values. You may find the positional notation of the format string (e.g., %1$s and %2$s) useful if you need to repeat a value in several places within your procedure body. The following example is a procedure that generates a new version of other procedures. The new version includes code that counts the number of times the procedure was called and measures the time it takes to run:

Example 10–2 Generating procedures dynamically with a template.

```
proc TraceGen {procName} {
    rename $procName $procName-orig
    set arglist {}
    foreach arg [info args $procName-orig] {
        append arglist "\$$arg "
    }
    proc $procName [info args $procName-orig] [format {
        global _trace_count _trace_msec
        incr _trace_count(%1$s)
        incr _trace_msec(%1$s) [lindex [time {
            set result [%1$s-orig %2$s]
        } 1] 0]
        return $result
    } $procName $arglist]
}
```

Suppose that we have a trivial procedure foo:

```
proc foo {x y} {
     return [expr $x * $y]
}
```

If you run TraceGen on it and look at the results, you see this:

```
TraceGen foo
info body foo
=>
     global _trace_count _trace_msec
     incr _trace_count(foo)
     incr _trace_msec(foo) [lindex [time {
        set result [foo-orig $x $y]
     } 1] 0]
     return $result
```

Exploiting the concat inside eval

The previous section warns about the danger of concatenation when forming commands. However, there are times when concatenation is done for good reason. This section illustrates cases where the concat done by eval is useful in assembling a command by concatenating multiple lists into one list. A concat is done internally by eval when it gets more than one argument:

```
eval list1 list2 list3 ...
```

The effect of concat is to join all the lists into one list; a new level of list structure is *not* added. This is useful if the lists are fragments of a command. It is common to use this form of eval with the args construct in procedures. Use the args parameter to pass optional arguments through to another command. Invoke the other command with eval, and the values in $args get concatenated onto the command properly. The special args parameter is illustrated in Example 7–2 on page 82.

Using eval in a Wrapper Procedure.

Here, we illustrate the use of eval and $args with a simple Tk example. In Tk, the button command creates a button in the user interface. The button command can take many arguments, and commonly you simply specify the text of the button and the Tcl command that is executed when the user clicks on the button:

```
button .foo -text Foo -command foo
```

After a button is created, it is made visible by packing it into the display. The pack command can also take many arguments to control screen placement. Here, we just specify a side and let the packer take care of the rest of the details:

```
pack .foo -side left
```

Even though there are only two Tcl commands to create a user interface button, we will write a procedure that replaces the two commands with one. Our first version might be:

```
proc PackedButton {name txt cmd} {
    button $name -text $txt -command $cmd
    pack $name -side left
}
```

This is not a very flexible procedure. The main problem is that it hides the full power of the Tk button command, which can really take about 20 widget configuration options, such as -background, -cursor, -relief, and more. They are listed on page 391. For example, you can easily make a red button like this:

```
button .foo -text Foo -command foo -background red
```

A better version of PackedButton uses args to pass through extra configuration options to the button command. The args parameter is a list of all the extra arguments passed to the Tcl procedure. My first attempt to use $args looked like this, but it was not correct:

```
proc PackedButton {name txt cmd args} {
    button $name -text $txt -command $cmd $args
    pack $name -side left
}
PackedButton .foo "Hello, World!" {exit} -background red
=> unknown option "-background red"
```

The problem is that $args is a list value, and button gets the whole list as a single argument. Instead, button needs to get the elements of $args as individual arguments.

Use eval with $args

In this case, you can use eval because it concatenates its arguments to form a single list before evaluation. The single list is, by definition, the same as a single Tcl command, so the button command parses correctly. Here we give eval two lists, which it joins into one command:

```
eval {button $name -text $txt -command $cmd} $args
```

The use of the braces in this command is discussed in more detail below. We also generalize our procedure to take some options to the pack command. This argument, pack, must be a list of packing options. The final version of PackedButton is shown in Example 10–3:

Example 10–3 Using eval with $args.

```
# PackedButton creates and packs a button.
proc PackedButton {path txt cmd {pack {-side right}} args} {
    eval {button $path -text $txt -command $cmd} $args
    eval {pack $path} $pack
}
```

In PackedButton, both pack and args are list-valued parameters that are used as parts of a command. The internal concat done by eval is perfect for this situation. The simplest call to PackedButton is:

```
PackedButton .new "New" { New }
```

The quotes and curly braces are redundant in this case but are retained to convey some type information. The quotes imply a string label, and the braces imply a command. The pack argument takes on its default value, and the args variable is an empty list. The two commands executed by PackedButton are:

```
button .new -text New -command New
pack .new -side right
```

PackedButton creates a horizontal stack of buttons by default. The packing can be controlled with a packing specification:

```
PackedButton .save "Save" { Save $file } {-side left}
```

The two commands executed by PackedButton are:

```
button .new -text Save -command { Save $file }
pack .new -side left
```

The remaining arguments, if any, are passed through to the button command. This lets the caller fine-tune some of the button attributes:

```
PackedButton .quit Quit { Exit } {-side left -padx 5} \
    -background red
```

The two commands executed by PackedButton are:

```
button .quit -text Quit -command { Exit } -background red
pack .quit -side left -padx 5
```

You can see a difference between the pack and args argument in the call to PackedButton. You need to group the packing options explicitly into a single argument. The args parameter is automatically made into a list of all remaining arguments. In fact, if you group the extra button parameters, it will be a mistake:

```
PackedButton .quit Quit { Exit } {-side left -padx 5} \
    {-background red}
=> unknown option "-background red"
```

Correct Quoting with eval

What about the peculiar placement of braces in PackedButton?

```
eval {button $path -text $txt -command $cmd} $args
```

By using braces, we control the number of times different parts of the command are seen by the Tcl evaluator. Without any braces, everything goes through two rounds of substitution. The braces prevent one of those rounds. In the above command, only $args is substituted twice. Before eval is called, the $args is replaced with its list value. Then, eval is invoked, and it concatenates its two list arguments into one list, which is now a properly formed command. The second round of substitutions done by eval replaces the txt and cmd values.

Do not use double quotes with eval.

You may be tempted to use double quotes instead of curly braces in your uses of eval. *Don't give in!* Using double quotes is, mostly likely, wrong. Suppose the first eval command is written like this:

```
eval "button $path -text $txt -command $cmd $args"
```

Incidentally, the previous is equivalent to:

```
eval button $path -text $txt -command $cmd $args
```

These versions happen to work with the following call because txt and cmd have one-word values with no special characters in them:

```
PackedButton .quit Quit { Exit }
```

The button command that is ultimately evaluated is:

```
button .quit -text Quit -command { Exit }
```

In the next call, an error is raised:

```
PackedButton .save "Save As" [list Save $file]
=> unknown option "As"
```

This is because the button command is this:

```
button .save -text Save As -command Save /a/b/c
```

But it should look like this instead:

```
button .save -text {Save As} -command {Save /a/b/c}
```

The problem is that the structure of the button command is now wrong. The value of txt and cmd are substituted first, before eval is even called, and then the whole command is parsed again. The worst part is that sometimes using double quotes works, and sometimes it fails. The success of using double quotes depends on the value of the parameters. When those values contain spaces or special characters, the command gets parsed incorrectly.

Braces: the one true way to group arguments to eval.

To repeat, the safe construct is:

```
eval {button $path -text $txt -command $cmd} $args
```

The following variations are also correct. The first uses list to do quoting automatically, and the others use backslashes or braces to prevent the extra round of substitutions:

```
eval [list button $path -text $txt -command $cmd] $args
eval button \$path -text \$txt -command \$cmd $args
eval button {$path} -text {$txt} -command {$cmd} $args
```

Finally, here is one more *incorrect* approach that tries to quote by hand:

```
eval "button {$path} -text {$txt} -command {$cmd} $args"
```

The problem above is that quoting is not always done with curly braces. If a value contains an unmatched curly brace, Tcl would have used backslashes to quote it, and the above command would raise an error:

```
set blob "foo\{bar space"
=> foo{bar space
eval "puts {$blob}"
=> missing close brace
eval puts {$blob}
=> foo{bar space
```

The uplevel Command

The uplevel command is similar to eval, except that it evaluates a command in a different scope than the current procedure. It is useful for defining new control structures entirely in Tcl. The syntax for uplevel is:

```
uplevel ?level? command ?list1 list2 ...?
```

As with upvar, the *level* parameter is optional and defaults to 1, which means to execute the command in the scope of the calling procedure. The other common use of level is #0, which means to evaluate the command in the global scope. You can count up farther than one (e.g., 2 or 3), or count down from the global level (e.g., #1 or #2), but these cases rarely make sense.

When you specify the *command* argument, you must be aware of any substitutions that might be performed by the Tcl interpreter before uplevel is called. If you are entering the command directly, protect it with curly braces so that substitutions occur in the other scope. The following affects the variable x in the caller's scope:

```
uplevel {set x [expr $x + 1]}
```

However, the following will use the value of x in the current scope to define the value of x in the calling scope, which is probably not what was intended:

```
uplevel "set x [expr $x + 1]"
```

If you are constructing the command dynamically, again use list. This fragment is used later in Example 10–4:

```
uplevel [list foreach $args $valueList {break}]
```

It is common to have the command in a variable. This is the case when the command has been passed into your new control flow procedure as an argument. In this case, you should evaluate the command one level up. Put the level in explicitly to avoid cases where $cmd looks like a number!

```
uplevel 1 $cmd
```

Another common scenario is reading commands from users as part of an application. In this case, you should evaluate the command at the global scope. Example 16–2 on page 220 illustrates this use of uplevel:

```
uplevel #0 $cmd
```

If you are assembling a command from a few different lists, such as the args parameter, then you can use concat to form the command:

```
uplevel [concat $cmd $args]
```

The lists in $cmd and $args are concatenated into a single list, which is a valid Tcl command. Like eval, uplevel uses concat internally if it is given extra arguments, so you can leave out the explicit use of concat. The following commands are equivalent:

```
uplevel [concat $cmd $args]
uplevel "$cmd $args"
uplevel $cmd $args
```

Example 10–4 shows list assignment using the foreach trick described on Page 75. List assignment is useful if a command returns several values in a list.

The lassign procedure assigns the list elements to several variables. The lassign procedure hides the foreach trick, but it must use the uplevel command so that the loop variables get assigned in the correct scope. The list command is used to construct the foreach command that is executed in the caller's scope. This is necessary so that $variables and $values get substituted before the command is evaluated in the other scope.

Example 10–4 lassign: list assignment with foreach.

```
# Assign a set of variables from a list of values.
# If there are more values than variables, they are returned.
# If there are fewer values than variables,
# the variables get the empty string.

proc lassign {valueList args} {
    if {[llength $args] == 0} {
        error "wrong # args: lassign list varname ?varname..?"
    }
    if {[llength $valueList] == 0} {
        # Ensure one trip through the foreach loop
        set valueList [list {}]
    }
    uplevel 1 [list foreach $args $valueList {break}]
    return [lrange $valueList [llength $args] end]
}
```

Example 10–5 illustrates a new control structure with the File_Process procedure that applies a callback to each line in a file. The call to uplevel allows the callback to be concatenated with the line to form the command. The list command is used to quote any special characters in line, so it appears as a single argument to the command.

Example 10–5 The File_Process procedure applies a command to each line of a file.

```
proc File_Process {file callback} {
    set in [open $file]
    while {[gets $file line] >= 0} {
        uplevel 1 $callback [list $line]
    }
    close $in
}
```

What is the difference between these two commands?

```
uplevel 1 [list $callback $line]
uplevel 1 $callback [list $line]
```

The first form limits callback to be the name of the command, while the second form allows callback to be a command prefix. Once again, what is the bug with this version?

```
uplevel 1 $callback $line
```

The arbitrary value of `$line` is concatenated to the `callback` command, and it is likely to be a malformed command when executed.

The subst Command

The `subst` command is useful when you have a mixture of Tcl commands, Tcl variable references, and plain old data. The `subst` command looks through the data for square brackets, dollar signs, and backslashes, and it does substitutions on those. It leaves the rest of the data alone:

```
set a "foo bar"
subst {a=$a date=[exec date]}
=> a=foo bar date=Thu Dec 15 10:13:48 PST 1994
```

The `subst` command does not honor the quoting effect of curly braces. It does substitutions regardless of braces:

```
subst {a=$a date={[exec date]}}
=> a=foo bar date={Thu Dec 15 10:15:31 PST 1994}
```

You can use backslashes to prevent variable and command substitution.

```
subst {a=\$a date=\[exec date]}
=> a=$a date=[exec date]
```

You can use other backslash substitutions like `\uXXXX` to get Unicode characters, `\n` to get newlines, or `\`-newline to hide newlines.

The `subst` command takes flags that limit the substitutions it will perform. The flags are `-nobackslashes`, `-nocommands`, or `-novariables`. You can specify one or more of these flags before the string that needs to be substituted:

```
subst -novariables {a=$a date=[exec date]}
=> a=$a date=Thu Dec 15 10:15:31 PST 1994
```

String Processing with subst

The `subst` command can be used with the `regsub` command to do efficient, two-step string processing. In the first step, `regsub` is used to rewrite an input string into data with embedded Tcl commands. In the second step, `subst` or `eval` replaces the Tcl commands with their result. By artfully mapping the data into Tcl commands, you can dynamically construct a Tcl script that processes the data. The processing is efficient because the Tcl parser and the regular expression processor have been highly tuned. Chapter 11 has several examples that use this technique.