Core-Style Arguments for Script Commands

Cyan Ogilvie
Ruby Lane, Inc.
cyan@rubylane.com

November 17, 2016

Abstract

Many core Tcl/Tk commands use named, optional arguments:

```
glob -nocomplain -type f -tails -directory $spooldir *@*
lsort -index 0 -stride 2 -dictionary $search_counts
entry .pw -show * -textvariable pw -width 15
```

But no support for this pattern is provided by the argument parsing of [proc]-defined commands, leading to horrors like:

```
searchdb "" $ss 0 notice "" "" $style $db $maxresults "" "" "" 1 "" \
    0 progresults "" "" "" "" 0 "" "" $userid $newtestrate 0 "" "" $website
```

`parse_args` [1] is a C extension using a custom Tcl_ObjType to provide core-like argument parsing at speeds comparable to `proc` argument handling, in a terse and self-documenting way.

1 Script-Defined Commands are Second Class Citizens

Positional arguments start to hurt readability once the arity exceeds 2 or 3, sometimes more if the set and order of the arguments is naturally obvious (`foreach v1 $list1 v2 $list2 $script`), less where it isn't (is it `lsearch $list $pattern` or `lsearch $pattern $list`?). For this reason the great majority of core Tcl commands that exceed 3 arguments employ a system of optional, named arguments:[2]

binary, chan, clock, exec, fconfigure, fcopy, file, glob, interp, load, lsearch, lsort, namespace, package, puts, read, regexp, regsub, return, socket, source, string, subst, switch, unload,

[1] https://github.com/RubyLane/parse_args

[2] I've considered the arguments that name an ensemble command part of the command, not the arguments for this survey

unset, zlib, pack, clipboard, place, event, wm, focus, font, winfo, grab, selection, send, grid, tk, bell and all Tk widget constructors and instance commands.

The exceptions are control structures (`dict {filter,for,map}`, `for`, `foreach`, `if`, `lmap`, `try`) for which the order is naturally clear; and the two that take 4 args: `trace remove` and `lreplace`.

Clearly it's Tclish to use named arguments, but the language provides no support for this pattern to script-defined commands created via proc, apply, method and coroutine. Of these, the first three provide support for positional arguments and default values, and the last (coroutine) provides no argument parsing at all, deferring to the script to interpret and verify its arguments.

Of course Tcl is expressive enough that script-defined commands can mimic all of the conventions established by the core commands, but implementing this in Tcl script has some major disadvantages:

- It's slow - about 50 times slower than native argument handling.
- It clutters procedure implementations with code that is orthogonal to their core mission.
- It obscures procedure signatures.
- Stack traces are less clear when argument requirements are not met.

These mean that core-style argument conventions are very rarely employed by script-defined commands, leaving them as second class citizens in their own language.

2 Conventions Established by the Core

Surveying the core commands that use named parameters reveals the following patterns:

- "`-foo`": a boolean toggle "foo" is enabled, its absence means that the toggle is disabled (e.g. `-nocase`, `-all`).
- "`-foo bar`": an argument named "foo" is assigned the value "bar". In some cases, not specifying the argument means that it takes a default value (e.g. `regexp -start`), in other cases that triggers behaviour different to all possible values (e.g. `lsort -command`).
- "`-foo`" / "`-bar`" / "`-baz`": a set of boolean-style arguments that are mutually exclusive and select a value for a single logical argument (e.g. `lsort -ascii / -integer / -dictionary`).
- "`--`": signals the end of the named options, further arguments are interpreted as positional parameters even if they would have matched a named argument (not universal).

- When contradicting arguments are given, later arguments override earlier ones: `lsort -increasing -ascii -dictionary -decreasing $list` uses dictionary comparison, decreasing order (possibly not universal).

parse_args is an extension that implements these patterns, prioritizing performance (so that hot code can use it guilt free); clarity (so that function signatures are obvious without requiring additional documentation); and terseness (lowering the cognitive burden to write and understand code using it).

As an example, a script implementation of the `glob` command might start like this:

```
proc glob args {
    parse_args $args {
        -directory  {}
        -join       {-boolean}
        -nocomplain {-boolean}
        -path       {}
        -tails      {-boolean}
        -types      {-default {}}
        args        {-name patterns}
    }

    if {$join} {
        set patterns    [list [file join {*}$patterns]]
    }

    if {[llength $patterns] == 0 && $nocomplain} return

    foreach pattern $patterns {
        if {[info exists directory]} {
            ...
        }
    }
    ...
}
```

And `regexp`:

```
proc regexp args {
    parse_args $args {
        -about      {-boolean}
        -expanded   {-boolean}
        -indices    {-boolean}
```

```
        -line       {-boolean}
        -linestop   {-boolean}
        -lineanchor {-boolean}
        -nocase     {-boolean}
        -all        {-boolean}
        -inline     {-boolean}
        -start      {-default 0}
        exp         {-required}
        string      {-required}
        matchvar    {}
        args        {-name submatchvars}
    }
    ...
}
```

3 Performance

Two design goals are in conflict when it comes to designing the signature format: intuitive definitions and high performance. Choosing a syntax that is easy for programmers to write and understand leaves more work for the code that interprets that syntax.

To reconcile these, `parse_args` saves the parsing configuration that it builds from the signature definition as the internal representation of a custom Tcl_ObjType (the string representation is just the signature definition as supplied). In this way the expensive work of interpreting the signature definition is only done once, when it is first used. This also neatly hooks memory management into the natural lifecycle of the definition, and ensures that Tcl_Objs aren't shared across Tcl_Interp instances or threads.

Part of the parsing configuration saved in the internal representation are string tables used to look up option names using Tcl_GetIndexFromObj. Since Tcl_GetIndexFromObj shimmers the input Tcl_Obj to a specialized type that saves the index found, subsequent lookups are very fast. A similar approach is used to efficiently validate enum-style options whose value must belong to a defined set (such as the -state option of most Tk widgets).

Tcl_Objs for the default values and enum choices are stored in the internal representation, leaving very little allocation of Tcl_Objs at parse time - almost all work is copying pointers and incrementing reference counts.

Performance is good enough that it is nearly on par with native positional argument support (times are in microseconds):

tcl parsing	24.540
native	0.535
parse_args	0.838

```
# Strange function signature is to allow the benchmarking machinery to
# pass the same args to both procs
proc native {t_a title c_a category  w_a wiki {r_a rating} {rating 1.0}} {
    list $title $category $wiki $rating
}

proc using_parse_args args {
    parse_args $args {
        -title      {-required}
        -category   {-default {}}
        -wiki       {-required}
        -rating     {-default 1.0 -validate {string is double -strict}}
    }
    list $title $category $wiki $rating
}
```

4 Beyond Parsing `proc` Arguments

Rather than provide a custom shim over `proc` that replaces the argument list parameter with a richer signature description [3] I opted to expose the argument parsing facility as a separate command. This allows it to serve in more contexts than just `proc` commands: TclOO constructors and methods; lambdas; command line parsing; configuration file handling; etc.

One particularly useful case is handling coroutine resume arguments, since no support is provided by the core for handling these beyond just supplying the list of arguments it was called with. A bit of boilerplate allows `parse_args` to be neatly slotted into place to handle these arguments:

```
coroutine foo apply [list {} {
    set res     {}
    set options {-code 0 -level 0}
    while 1 {
        catch {
            parse_args [yieldto return -options $options $res] {
```

[3] This is the approach taken by nsf::proc, part of the Next Scripting Framework

```
                -foo    {-default xyzzy}
                -count  {-required}
            }

            ... generate next value
        } res options
    }
}]
```

5 Future Work

- It would be helpful to expose a C API.
- Support positional parameters interspersed with (or preceding) named parameters.
- Some proper documentation is probably a good idea.

Appendix A: Parse Signature Format

The signature format argument to `parse_args` is a dictionary whose keys define the valid parameters and whose values define the properties of that parameter. If the parameter name begins with a "-" character it is treated as a named parameter, otherwise it is a positional parameter that must appear after all named parameters.

The following settings are valid in the parameter properties:

- `-default default_value`
  If the parameter is not supplied it takes the value *default_value*.
- `-required`
  Flags the parameter as being required – if no value was supplied an error is thrown. If neither `-required` nor `-default` are specified and no value is supplied by the caller, the corresponding output variable is not set. The script can then use `info exists param_name` to distinguish this case from any possible value that could be passed by the caller.
- `-validate function`
  The command prefix *function* has the supplied value appended and the resulting command is executed. If the result is an error or a boolean false value then the value is rejected and an error is thrown.
- `-name output_name`
  Normally the parameter key supplies the name for the output parameter (sans the

leading "-" for named parameters). If `-name` is specified then *output_name* is used instead.

- `-boolean`
  Flags the parameter as being a boolean toggle. If the parameter is supplied then the output parameter will contain a boolean true value, otherwise false.

- `-args`
  Ordinary named parameters consume the following argument as the value to assign to the output parameter. `-args` specifies how many arguments to consume instead (must be greater than 1).

- `-multi`
  Flags this parameter as one of the choices for a mode selection type parameter – should be used together with `-name`. All `-multi` parameters that share the same `-name` are treated as flags that supply the value stored in the output parameter (sans the leading "-"). If conflicting parameters are supplied the last one sets the value. Specifying `-required` on any of the linked `-multi` parameters means that at least one of the choices must be set by the caller. Specifying `-default` on any of the linked parameters establishes the default value if none is supplied.

- `-enum valid_values`
  Enforces that the supplied value exactly matches one of the elements in the list *valid_values*.

- `-# comment`
  Ignores *comment*, allowing comments to be inserted into the signature definition.

Appendix B: Examples That Mimic Core Commands

These examples show what the argument parsing for selected core Tcl commands would look like if implemented in a script using parse_args:

```
proc lsort args {
    parse_args $args {
        -ascii      {-name compare_as -multi -default ascii}
        -dictionary {-name compare_as -multi}
        -integer    {-name compare_as -multi}
        -real       {-name compare_as -multi}

        -command    {}

        -increasing {-name order -multi -default increasing}
```

```
        -decreasing {-name order -multi}

        -indices    {-boolean}
        -index      {}
        -stride     {-default 1}
        -nocase     {-boolean}
        -unique     {-boolean}
        list        {-required}
    }

    if {![info exists command]} {
        switch -- $compare_as {
            ascii {
                set command {string compare}
                if {$nocase} {
                    lappend command -nocase
                }
            }
            dictionary {
                ...
            }
            integer - real {
                set command tcl::mathop::-
            }
        }
    }
}


proc lsearch args {
    parse_args $args {
        -exact      {-name matchtype -multi}
        -glob       {-name matchtype -multi -default glob}
        -regexp     {-name matchtype -multi}

        -sorted     {-boolean}
        -all        {-boolean}
        -inline     {-boolean}
        -not        {-boolean}
        -start      {-default 0}
```

```
        -ascii      {-name compare_as -multi -default ascii}
        -dictionary {-name compare_as -multi}
        -integer    {-name compare_as -multi}
        -real       {-name compare_as -multi}

        -nocase     {-boolean}

        -decreasing {-name order -multi}
        -increasing {-name order -multi -default increasing}
        -bisect     {-boolean}

        -index      {}
        -subindices {-boolean}
    }

    if {$sorted && $matchtype in {glob regexp}} {
        error "-sorted is mutually exclusive with -glob and -regexp"
    }
}


proc entry {widget args} {
    parse_args $args {
        -disabledbackground {-default {}}
        -disabledforeground {-default {}}
        -invalidcommand     {-default {}}
        -readonlybackground {-default {}}
        -show               {}
        -state              {-default normal -enum {
            normal disabled readonly
        }}
        -validate           {-default none -enum {
            none focus focusin focusout key all
        }}
        -validatecommand    {-default {}}
        -width              {-default 0}
        -textvariable       {}
    }
}
```