Tcl/Tk Advanced Tutorial

(*Tcl*)

Stephen Uhler

Brent Welch

Sun Microsystems Laboratories

`http://www.sunlabs.com/research/tcl`

The Tcl Parser

- Grouping before substitution
- Lists and command structure
- Why use eval?
- `eval` concatenates its arguments
- $args and eval

Grouping Before Substitution

- Substituted values don't affect the parse

```
one two three four          ;# 4 words
{one two} {three four}      ;# 2 words

set x "\$x y \[exit]"       ;# 3 words
puts $x                     ;# 2 words

set ab"d ef"                ;# 3 words
set file [pwd]/${ab”d}      ;# 3 words
=> /usr/brent/ef”
```

Weird Values are OK

```
set pat \[a-z]\$
regexp $pat [gets stdin]

proc Pat {chars} {
  return \[$chars\]
}
regexp [Pat a-z] [gets stdin]
```

Tcl Data Structures

- Strings
  - » `format  scan`
  - » `regexp  regsub`
  - » `split  join  append`
- Lists
  - » `[info commands l*]`
- Arrays
  - » `array`

Tcl Lists

- All strings are not valid Tcl lists
  - » Same syntax rules as Tcl commands
- Do not treat arbitrary input as lists
  - » Use `regexp, scan,` or `split`
- Get help building lists
  - » `list`: builds list out of arguments
  - » `lappend`: adds to the end of a list

Why Use `list` ?

- Capture command in valid list structure
  - » Safe `button`, `after`, `send` commands

```
set string "Hello, World!"
button .foo -command [list puts $string]
after 500 [list puts $string]
send logger [list Log $string]
```

  - » Compare with doing it yourself

```
button .foo -command "puts \"$string\""
```

When **Not** to use Lists

- Lists can be slow to access
  - » String representation is reparsed each time
- Use arrays, which have constant cost
  - » If order matters, use two arrays

```
incr i
set array($key) $value
set order($i) $key
```

Arrays

- Use to simulate other data structures
- Complex indices ok, variables can help

```
set tree(left,$x) $y
```

- Copy subsets with array set/get

```
array set sub [array get whole a,*]
```

- Iteration - just use `foreach`

```
foreach x [array names whole a,*] { }
```

Why Use `eval`?

- `eval` causes a second round of parsing
  - » Save commands and execute them later: hooks and callbacks
- `eval` concatenates its arguments
  - » Splices multiple lists into one command
  - » `eval` often works well with `$args`

Hooks and Callbacks

```
proc HtmlIterate {hook} {
  global html
  foreach htag $html(tags) {
     eval $hook {$htag}
  }
}
HtmlIterate [list puts $output]

=> puts $output {<a href="...">}
```

2

1

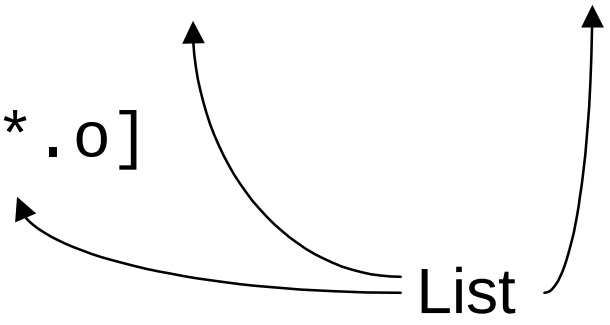
Number of parses

`eval` uses `concat` to join lists

```
set hook [list puts $output]
concat $hook {$htag}
=> puts $output $htag

set points “0 0 10 10 10 20 20 20 30 15”
eval {$canvas create polygon} $points

eval exec rm [glob *.o]
```



`eval` and `$args`

- `args` is a list of extra arguments
  - » The concat by eval passes $args through

```
proc MyButton {name t cmd args} {
  eval {button $name -text $t} \
   {-command $cmd} $args
  pack $name -side left
}
```

MyButton Examples

```
MyButton .foo "Foo" {puts foo}
=> button .foo -text "Foo" \
     -command {puts foo}

MyButton .foo "Foo" {puts foo} -fg blue
=> button .foo -text "Foo" \
     -command {puts foo} -fg blue
```

Substitution without `eval`

- `subst` just does substitutions

```
subst puts $file $string
=> puts file4 Hello, World
```

- `subst` does not honor curly braces

```
subst puts $file {$string}
=> puts file4 {Hello, World}
```

- Use backslash to quote `$`, `\` and `[`

Quirks in `expr`

- `expr` does its own substitutions

```
expr {$x + [llength $y]}
```

- `expr` converts strings to numbers first

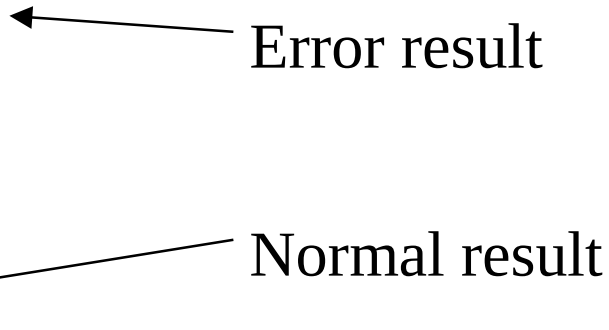
```
expr {"0xa" == "10"}
```

- May convert back to string, but uses %d

```
expr {"0xa" > "0xbx"}
=> "10" > "0xbx" => true!
```

Trapping Errors

```
if [catch {open $file w} out] {
  puts stderr $out
} else {
  puts $out ...
  close $out
}
```



A longer `catch` Phrase

```
if [catch {
  cmd1 ; cmd2 ; cmd3 ...
} result] {
  global errorInfo
  # print stack trace, including $result
  puts stderr "*Trace*\n$errorInfo"
} else {
  # $result has normal result
}
```

Extending `catch`

- Alternate return codes

  `return -code` *`value`*

  `value:` `return break continue` *`integer`*

- Multiway catch phrases

```
switch [catch {command} result] {
  3: { # if return -code break }
  4: { # if return -code continue }
  5: { # if return -code 5 }
}
```

Using Upvar

- Passing arrays to procedures
- Static procedure variables
- Local aliases

Passing Arrays to Procedures

- Call-by-name

```
proc fruit {objectName} {
  upvar $objectName obj
  return $obj(fruit)
}
set object7(fruit) orange
fruit object7
=> orange
```

Upvar: Static Procedure Variables

- Share state among package procedures
- Keep state “hidden” from package user

```
proc my_proc {handle args} {
  upvar #0 my$handle local
  set local(something) ...
}
```

Upvar: Local Aliases

- Use to avoid awkward indirect references

➪ `set result $$var` (**Wrong!**)

⌘ `set result [set $var]` (**Awkward**)

⌘ `upvar 0 $var alias`
`  set result $alias`

Tcl is a Dynamic Language

- Define code at runtime
  - » Map data into code that processes it
    `parse_html` converts HTML to a Tcl program
  - » Rewrite procedure bodies for tracing
  - » Compute “hardwired” search procedures

Tcl Regular Expressions

- Regular expressions - a review
- Constructing regular expressions
- Using **`regexp`**
- Using **`regsub`**

Regexp - Special Characters

- `*` zero or more
- `+` one or more
- `?` zero or one
- `.` any character
- `^` beginning of string
- `$` end of string
- `|` alternation
- `()` grouping and sub-variable matching
- `[]` Character classes
  - » `-` range indicator
  - » `^` negation operator
- `\` turns off special meaning

Constructing Regular Expressions - the problem

- The regular expression parser and Tcl both use the same “special” characters
- The special characters have different meanings depending on the context
- Keeping track of who interprets which special character when can be confusing

Using Regexp and Regsub

- Regexp
  - » Searching for patterns
  - » Parsing structured text with *submatchvar*
- Regsub
  - » String substitution
  - » counting matches
  - » Converting structured text to Tcl programs

Constructing Regular Expressions - Strategies

- {}
  - » Use when no Tcl substitutions are required
- " "
  - » Use sparingly for simple expressions
- format
  - » Use when few Tcl substitutions are needed
- append
  - » Use to build complex expressions

Constructing Expressions

Eliminating redundant white space

- **`regsub -all` *`expression`* `$text { } new`**
  - `{[\t\n\r ]+}` **Wrong!**
  - `"[\t\n\r ]+"` **Wrong!**
  - `"\[\t\n\r ]+"`
  - `[format {[%s]+} "\t\n\r "]`
  - `append exp {[} "\t\n\r " {]}`

Regexp Example

- Parsing URLS

```
set pat ^(\[^:]+):              ;# Protocol
append pat  //(\[^:/]+)         ;# Server
append pat (:(\[0-9]+))?        ;# Port
append pat (/.*\$)              ;# Path
regexp $pat $url match \
  proto server x port path
```

Counting with Regsub

- Example - count words in $text

```
   set word "0-9a-zA-Z"
1  append exp {[} ^$word {]*[} $word {]+}
2  set exp [format {[^%1$s]*[%1$s]+} $word]
3  set exp "\[^$word\]*\[$word\]+"
   set count [regsub -all $exp $text {} x]
```

Converting Text to Tcl:

Tcl *un-cgi* script

- Convert a=b&c=d... to {a} {b} {c} {d} ...

```
proc cgiDecode {data} {
  foreach i [split $data "&="] {
    lappend result [cgiMap $i]
  }
  return $result
}
```

Converting Text to Tcl:

Tcl *un-cgi* script

- Convert "+" to " "
- Convert %xx to equivalent character

```
proc cgiMap {data} {
  regsub -all {\+} $data " " data
  regsub -all {[[$\\]} $data {\\&} data
  regsub -all {%([0-9a-zA-Z][0-9a-zA-Z])}\
     $data {[format %c 0x\1]} data
  return [subst $data]
}
```

Un-cgi Script in Action

- Sample Input text

```
age=%3e+25&name=Stephen+A%2e+Uhler
```

- Split into a list

```
{age} {%3e+25} {name} {Stephen+A%2e+Uhler}
```

- Convert "+" to " "

```
{age} {%3e 25} {name} {Stephen A%2e Uhler}
```

Un-cgi Script in Action

- Convert %xx to format command

```
{age} {[format %c 0x3e] 25} {name}
   {Stephen A[format %c 0x2e] Uhler}
```

- Use "subst" for the final result

```
{age} {> 25} {name} {Stephen A. Uhler}
```

Complete Tcl HTML Parser

```
proc htmlParse {html cmd start} {
  regsub -all \{ $html {\&ob;} html
  regsub -all \} $html {\&cb;} html
  regsub -all \\ $html {\&bsl;} html
  set ws " \t\r\n"
  append exp {<(/?)([^} $ws {>]+)[}
  append exp $ws {]*([^>]*)>
  set sub "\}\n$cmd {\\2} {\\1} {\\3} \{"
  regsub -all $exp $html $sub html
  eval "$cmd $start {} {} \{ $html \}"
}
```

HTML Parser at Work

- Sample HTML document

```
<title>Tcl/Tk Project At
Sun Microsystems Laboratories&copy;
  {really!}</title>
<img src="images/tcltk.gif">
<br>
<h1>The Tcl/Tk Project</h1>
```

HTML Parser at Work

- Hide Tcl special characters

```
<title>Tcl/Tk Project At
Sun Microsystems Laboratories&copy;
  &ob;really!&cb;</title>
<img src="images/tcltk.gif">
<br>
<h1>The Tcl/Tk Project</h1>
```

HTML Parser at Work

- After Command Substitutions

```
}
command {title} {} {} {Tcl/Tk Project At
Sun Microsystems Laboratories&copy;
  &ob;really!&cb;}
command {title} {/} {} {
}
command {img} {} {src="images/tcltk.gif"}
  {
}
command {br} {} {} {
}
command {h1} {} {} {The Tcl/Tk Project}
command {h1} {/} {} {
```

HTML Parser at Work

- Command sent to `eval`

```
command {start} {} {} {}
command {title} {} {} {Tcl/Tk Project At
Sun Microsystems Laboratories&copy;
  &ob;really!&cb;}
command {title} {/} {} {
}
command {img} {} {src="images/tcltk.gif"} {
}
command {br} {} {} {
}
command {h1} {} {} {The Tcl/Tk Project}
command {h1} {/} {} {
}
```

Interprocess Communication

- With `exec` *`command`*
  - » Blocks until *`command`* completes
  - » Returns standard output of *`command`*
  - » Use only for short lived processes
- With `exec` *`command &`*
  - » Returns immediately with process id (pid)
  - » Poll for completion using “exec kill -0 $pid”
  - » Use “send” to receive status

Polling for Completion

```
proc watch {command callback {int 1000} {pid 0}} {
  if {$pid == 0} {
      set pid [eval exec $command &]
  } elseif {[catch {exec csh -c "kill -0 $pid"}]} {
      eval $callback
      return
  }
  after $int \
      [list watch $command $callback $int $pid]
  return $pid
}
```

Using Pipelines

- Starting the *command*

  `set fd [open "|`*`command`*`" w+]`

- gets/puts handshaking
  - » *command* **must** flush each line

```
puts $fd $stuff
flush $fd
gets $fd result
```

Using Pipelines - `fileevent`

```
proc watch {command callback} {
  set fd [open "|$command"]
  fileevent $fd readable \
      [list run_callback $fd $callback]
  return [pid $fd]
}

proc run_callback {fd callback args} {
  if {[gets $fd dummy] == -1} {
      catch {close $fd}
      eval $callback
  }
}
```

Tk Send Command

- Registry of Tk applications

```
winfo interps
```

- Passing your name to a subprocess

```
exec foobar [tk appname] &
```

- List warning: internal concat like `eval`

```
send $interp [list doit $arg1 $arg2]
```

Tcl Introspection

- Rewriting Procedures
- Interactive command evaluation
- Looking at the execution stack

Rewriting Tcl Procedures

```
proc rewrite {proc redo} {
  set args {}
  foreach arg [info args $proc] {
    if {[info default $proc $arg default]} {
      lappend args [list $arg $default]
    } else {
      lappend args $arg
    }
  }
  proc $proc $args [$redo [info body $proc]]
}
```

Read-Print-Eval Loop

```
proc get_command {{prompt "% "}} {
  puts -nonewline $prompt
  gets stdin line
  while {![info command complete $line] {
     puts -nonewline "? "
     append line "\n[gets stdin]"
  }
  return $line
}
```

Simple Debugger

```
proc breakpoint {show} {
   set top [expr [info level] -1]
   set current $top
   while {1} {
      set line [get_command #$current]
      switch -- $line {
        + {if $current < $top} {$show [incr current]}
        - {if $current > 0} {$show [incr current -1]}
        ? {$show $current}
        C {return}
        default {
          catch {uplevel #$current $line} result
          puts stderr $result
        }
      }
   }
}
```

Print Stack Information

```
proc show {level} {
  if {$level > 0} {
    set info [info level $level]
    set proc [lindex $info 0]
    set i 0
    puts stderr "$level: $proc"
      foreach arg [info args $proc] {
         puts stderr "\t$arg = [lindex $info [incr i]]
      }
    show [incr level -1]
  } else {
    puts stderr "Top Level"
  }
}
```

Managing Large Tcl Programs

- One module per file
- Auto load modules via `tclIndex`
- Module prefix for proc names
- Global arrays for module state
  - » Use `upvar #0` trick for instance data

Extending Tcl/Tk

- Dynamic Loading
- Six ways to extend Tcl/Tk
- "C" Calling conventions
- Tricks
  - » Bypassing the parser with Tcl_Invoke
  - » *Faking* new widget sub-commands

Dynamic Loading

- Create new functionality in C
- Generate a shared library
- Incorporate into “wish” with **`load`**
  - » `load ./libfoo.so Foo`
- No need for *custom* “wish”

Six Ways to Extend Tcl

- New Tcl commands
- New Tk widgets
- New expr math functions
- New photo image handlers
- New image types
- New canvas items

"C" Calling Conventions

- Module Initialization

```
#include <tcl.h>

static int  newCmd _ANSI_ARGS_((ClientData clientData,
          Tcl_Interp *interp, int argc, char **argv));
int
New_Init(interp)
    Tcl_Interp *interp;
{
    Tcl_CreateCommand(interp, "new", newCmd,
       (ClientData) 0,(Tcl_CmdDeleteProc *) NULL);
    return TCL_OK;
}
```

"C" Calling Conventions

- Command Procedure

```
static int
newCmd(data, interp, argc, argv)
    ClientData data;
    Tcl_Interp *interp;
    int argc;
    char **argv;
{
  /* put command behavior here */
  Tcl_SetResult(interp,"The result", TCL_STATIC);
  return(TCL_OK);
}
```

Bypassing the Parser

- Calling Command Procedures Directly

```
Tcl_CmdInfo info;
char *cmd;

Tcl_ResetResult(interp);
Tcl_GetCommandInfo(interp, cmd, &info)
return (*info.proc)(info.clientData, interp,
    argc, argv);
```

Faking sub-widget commands

- Extracting text widget data

```
TkText *
GetTextData(Tcl_Interp *interp, char *widget)
  Tcl_CmdInfo info;
  Tcl_GetCommandInfo(interp, widget, &info);
  Tcl_VarEval(interp, "winfo class ", widget,
      NULL);
  if (strcmp(interp->result, "Text") == 0) {
    return (TkText *)info.clientData;
  } else {
    return NULL;
  }
}
```