A Short Introduction to Tcl

Remington Furman

January 2, 2025

Outline

About Tcl

Basics

Datatypes

Examples

Q&A

History

- Created by John Ousterhout while at UC, Berkeley
  - Made to script electronic design automation tools, like the Magic VLSI design program
  - Released Tcl in 1988, and Tk in 1991
  - Declined to work at Netscape in 1994
    - So we have Javascript in browsers instead
- Now managed by a "Tcl Core Team" of about a dozen members
- Popular in the ’90s and declined in popularity in the last 20 years

What is it?

- Tool Command Language
- An interpreted scripting language
- Feels like a cross between Bash and Lisp, but easier than both

What is it?

- Extensible
  - Designed to work with external libraries and programs
  - Can be extended with Tcl code or libraries written in C (or other languages with C bindings)
- Embeddable
  - Can be built into other programs to add a scripting interface

Features

- Very well documented
  - Web, man pages, books, wiki
- Simple, minimal syntax
  - More simple than Bash, Python, Perl[1], etc.
  - This is one of the things I like most about Tcl
  - Visually uncluttered
  - Easier to edit, both mecanically and cognitively

Features

- "Stringly typed"
  - "Everything is a string" or can be represented by a string
- String interpolation everywhere
- Homoiconic, if you're into that kind of thing
  - Code is text, and text is code
  - Data is text, and text is data
  - More Lispy than Lisp in that way
- Unicode support

Features

- Regular expressions
- Numerics
  - Big integers (arbitrary precision, no overflow)
  - Doubles
- Tk GUI library
- Multiple OOP libraries to choose from, if you want

Features

- Turing complete
  - I’ll prove it now

What does it look like?

```
#!/usr/bin/env tclsh9.0
# This is an implementation of BF.

# BF Program
set prog [split [lindex $argv 0] {}]
set lp [llength $prog]
set ip 0

# BF Data
array set data {}
array default set data 0
set dp 0
```

What does it look like?

```
proc mb {dir} {
    # Match a bracket forwards if dir=1,
    # backwards if dir=-1.
    global prog ip
    set opp [expr -1 * $dir]
    set count 1
    while {$count} {
        switch [lindex $prog [incr ip $dir]] {
            \[ {incr count $dir}
            \] {incr count $opp}
        }
    }
    if {$dir == 1} {incr count $dir}
}
```

What does it look like?

```
# BF interpreter
while {$ip < $lp} {
    switch [lindex $prog $ip] {
        > {incr dp}
        < {incr dp -1}
        + {incr data($dp)}
        - {incr data($dp) -1}
        . {puts -nonewline
                [format "%c" $data($dp)]}
        , {scan [read stdin 1]
                "%c" data($dp)}
        \[ { if { !$data($dp) } { mb  1 } }
        \] { if {  $data($dp) } { mb -1 } }
    }
    incr ip
}
```

Turing complete

```
./bf9.tcl "++++++++[>++++\
          [>++>+++>+++>+<<<<-]\
          >+>+>->>+[<]<-]>>.\
          >---.+++++++..+++.>>.\
          <-.<.+++.------.\
          --------.>>+.>++."
Hello World!
```

- BF is Turing complete
- I showed a working implementation of BF in Tcl
- This proves that Tcl is Turing complete as well
- Therefore, Tcl is capable of serious work

What's missing?

- Pointers/references, oddly
  - It took me a while to notice this
  - Simply store variable names or array element names in strings to make references
- A package manager for easily installing packages
  - Use your OS package manager instead
  - It is possible to export a Tcl interpreter and script as a standalone program though
- A community large enough to achieve global domination
  - Looking at you, Python

Latest versions

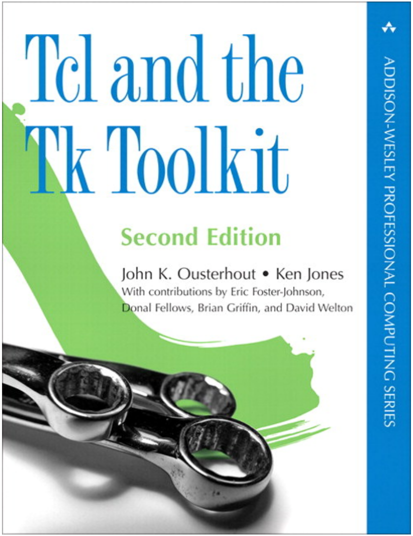
- Tcl 8.6 has been the most recent version since 2012
  - Tcl 8.6.15 released on 2024-09-13
  - Still recommended for now
- Tcl 9.0 just released on 2024-09-24
  - Will take a while to roll out packages for distributions and external Tcl packages/extensions to be updated
  - I had to compile Tcl 9.0 myself to try it
  - Not that hard (`configure`, `make`, `make install`)

Documentation

- Web reference
  - `https://www.tcl-lang.org/man/tcl/TclCmd/contents.htm`
  - Same content as man pages, but easier to browse
  - Interactive search at:
    - `https://www.magicsplat.com/tcl-docs/`
- Man pages in `3tcl` section (depending on your OS)
- Tclwiki
  - `https://wiki.tcl-lang.org/`
  - Lots of discussion and code examples here

Tcl and the Tk Toolkit, Second Edition

- Covers Tcl/Tk 8.5, still useful for 8.6 and 9.0


Tcl and the Tk Toolkit
Second Edition
John K. Ousterhout • Ken Jones
With contributions by Eric Foster-Johnson, Donal Fellows, Brian Griffin, and David Welton
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Where It’s used

- gitk
  - First GUI for Git, written with Tcl/Tk
- tkinter
  - Python bindings for Tk GUI library
- FlightAware uses Tcl for ADS-B data collection on Raspberry Pi
  - `https://github.com/orgs/flightaware/repositories?q=lang%3Atcl&type=all`
- EDA tools
  - Xilinx Vivado, Vitis, and XSCT for FPGA development
    - I use these at work

TkDocs Tutorial

- A useful tutorial on Tk GUIs
- Provides side-by-sie code samples in Tcl, Python, Perl, and Ruby
  - Sort of like Rosetta code
- `https://tkdocs.com/tutorial/`

Syntax

- A little confusing at first, feels a bit hacky
- Once you learn the ruls and get used to them they form a powerful system
- Don't think in terms of a traditional lexer+parser structure. It's more like a shell
- The Dodekalogue: twelve rules that specify the parsing behavior
  - `https://www.tcl-lang.org/man/tcl/TclCmd/Tcl.htm`

Basics

- Every line is a "command" made of space separated words
- The first word is a procedure name
- The rest of the "words" (if any) are arguments to the procedure
- Tcl performs variable substitution (`$`) and command substitution (`[]`) in each word before executing the command

Types of Quotes

- Double quotes (`" "`)
  - Allows a single "word" to contain whitespace
  - Performs variable substitution (`$`) and command substitution (`[ ]`)
- Square brackets (`[ ]`)
  - Immediate execution of the text inside the braces as a command
  - Expands to the string returned by the command
  - Strangely doesn't allow newlines. Escape newlines with a backslash (`\`), if needed.
- Curly braces (`{ }`)
  - Used for deferred execution
  - Disables string substitution, command substitution, newline separators

Variables (`set`)

- Sets a variable (when given two arguments)

```
set var 1
```

- Reads a variable (when given one argument)

```
% set var
1
```

- Variable substitution

```
% puts "var is: $var"
var is: 1
```

Output

- `puts` prints a string, with trailing newline:

```
puts "Hello world!"
```

- Use `-nonewline` to suppress newline:

```
puts -nonewline "Starting long thing..."
# Do something for a while...
puts " done!"
```

- `format` is a lot like C `printf()`, with more options:

```
% format 0x%x 100
0x64
% format 0b%b 100
0b1100100
```

Comments (#)

- Comments, like everything else in Tcl, are a command

```
# This is a comment.
```

- Warning! Commands have to start on a new line, or after a semicolon, including comments

```
puts "$var" ; # Print var.
```

Control Flow Commands

- All control flow is done with commands
- There is no special syntax or pasing for conditionals

Conditionals (if, else, etc.)

```
if {condition_expression} {
    true_cmds
} else {
    false_cmds
}
```

- if command expects all arguments on one line
- Must use the "One True Brace Style" with curly braces to span multiple lines

Loops (`for`)

- Like a `for` loop in C:

```
for {set i 0} {i < 0} {incr i} {
    puts $i
}
```

- Form: `for init test update body`
- The second argument is an "expression string" evaluated with `expr`
  - More on `expr` later

Loops (`foreach`)

- Loops through every element in a list:

```
foreach {element} $list {
    puts $element
}
```

Procedures (proc)

- Tcl's term for functions is "procedures"
  - "proc" for short

```
proc example {arg1 {arg2 default}} {
    puts "arg1: $arg1"
    puts "arg2: $arg2"
}
```

- Procedures always return a string
  - Possibly empty

Tcl Command Internals

- C programs take a list of null-terminated strings:

```
int
main(int argc, char *argv[]) {
    return 0;
}
```

- Argument strings can have spaces in them, just like Tcl "words"
- This is a major inspiration for the Tcl language

Tcl Command Internals

- Early Tcl procedures took a list of strings and some interpreter state:

```c
int
Tcl_CmdProc(ClientData clientData,
            Tcl_Interp *interp,
            int argc, char *argv[]) {
    interp->result = "true";
    return TCL_OK;
}
```

Tcl Command Internals

- Tcl later switched to using `Tcl_Obj` structs for representing values:

```
int
Tcl_CmdProc(ClientData clientData,
            Tcl_Interp *interp,
            int objc, Tcl_Obj *const objv[]) {
    Tcl_SetObjResult(interp,
                     Tcl_NewBooleanObj(1));
    return TCL_OK;
}
```

- This allows optimizations for data structures like dictionaries.

expr

- The expr command provides infix syntax for math
- Infix notation very similar to C syntax and features

```
% puts [expr 0x07 | (1<<3)]
15
```

- The expr command is a neat example of how to include a mini-language in Tcl

Math Functions

- `expr` supports all the C standard library math functions and a few more:

```
abs     acos    asin    atan
atan2   bool    ceil    cos
cosh    double  entier  exp
floor   fmod    hypot   int
isqrt   log     log10   max
min     pow     rand    round
sin     sinh    sqrt    srand
tan     tanh    wide
```

Lisp-like Math Functions

- You can get Lisp-like math functions in the `::tcl::mathfunc` and `::tcl::mathop` namespaces:

```
% ::tcl::mathfunc::max 1 2 3 4
4
% ::tcl::mathop::+ 1 2 3 4
10
```

Tcl is "stringly typed"

- Everything can be represented by a string
  - This includes code!
- String representation used for both input and output
- Strings are first class objects
- Under the hood, variables can be implemented as other types to increase performance
  - Automatically converted to and from strings as necessary

"Shimmering"

- A frequent conversion between a string representation and an optimized type and back
- Avoided by calling only commands that operate on the optimized representation (`dict` commands, for example)

Lists (`list`, `lindex`, etc)

- Lists are simply represented as space separated words in a string
- Can be created with the `list` command
- And efficiently operated on with many commands starting with `l`:
    - `llength`, `linsert`, `lsort`, etc.

```
set sharps {c cis d dis e f fis g gis a ais b}
lsort -decreasing $sharps
# gis g fis f e dis d cis c b ais a
```

Dictionaries (`dict`)

- Dictionaries are simply a list of alternating key value pairs in a string
- Can be created and modified with the `dict` command and sub-commands
  - `dict create`, `dict set`, `dict keys`, etc.

```
set pitch_names {
    sharps {C C♭ D D♭ E F F♭ G G♭ A A♭ B}
    flats  {C D♯ D E♯ E F G♯ G A♯ A B♯ B}
    both   {C C♭/D♯ D D♭/E♯ E F F♭/G♯ G G♭/A♯ A
     ↪  A♭/B♯ B}
}
set interval_dict {
     W 2   H 1   T 2   S 1
    P1 0  m2 1  M2 2  m3 3   M3 4   P4 5   TT 6
    P5 7  m6 8  M6 9  m7 10  M7 11  P8 12
}
```

Arrays (`array`)

- Arrays are groups of Tcl variables with hash table semantics
- For example:

```
set array(element1) 1
set index element1
puts $array($index)
# 1
```

- Kind of wonky, and I much prefer the newer `dict` methods

Example: Hamming distance

Given two bit strings `a` and `b`, count the number of bits that differ (`popcount(a XOR b)`).

```
    01110100011001010111001101110100
XOR 01110010011001010111001101110100
------------------------------------
    00000110000000000000000000000000
```

```
proc hamming_distance {a b} {
    return [regexp -all 1 \
                [format %b \
                     [expr 0b$a ^ 0b$b]]]
}
```

```
% hamming_distance $a $b
2
```

Q&A

- Any questions?