Tcl was created by John Ousterhout as a reusable scripting language for chip design tools he was creating. In 1997 he was awarded the ACM Software System Award for Tcl. Tcl can be used both as an embeddable scripting language and as a general programming language. It can also be used as a portable C library, even in cases where no scripting capability is needed, as it provides data structures such as dynamic strings, lists, and hash tables. The C library also provides portable functionality for loading dynamic libraries, string formatting and code conversion, filesystem operations, network operations, and more. Various features of Tcl stand out:

- Convenient cross-platform networking API
- Fully virtualized filesystem
- Stackable I/O channels
- Asynchronous to the core
- Full coroutines
- A threading model recognized as robust and easy to use

If Lisp is a list processor, then Tcl is a string processor. All values are strings. A list is a string format. A procedure definition is a string format. To achieve performance, Tcl internally caches structured representations of these values. The list commands, for example, operate on the internal cached representation, and Tcl takes care of updating the string representation if it is ever actually needed in the script. The copy-on-write design of Tcl allows script authors can pass around large data values without actually incurring additional memory overhead. Procedures are automatically byte-compiled unless they use the more dynamic commands such as “uplevel”, “upvar”, and “trace”.

Tcl is a pleasure to program in. It will appeal to hacker types who find Lisp, Forth, or Smalltalk interesting, as well as to engineers and scientists who just want to get down to business with a tool that bends to their will. Its discipline of exposing all programmatic functionality as commands, including things like loops and mathematical operations that are usually baked into the syntax of other languages, allows it to fade into the background of whatever domain-specific functionality a project needs. It’s syntax, which is even lighter that that of Lisp, just gets out of the way.

```
#! /bin/env tclsh

################################################################################
## 1. Guidelines
################################################################################

# Tcl is not Bash or C!  This needs to be said because standard shell quoting
# habits almost work in Tcl and it is common for people to pick up Tcl and try
# to get by with syntax they know from another language.  It works at first,
# but soon leads to frustration with more complex scripts.

# Braces are just a quoting mechanism, not a code block constructor or a list
# constructor.  Tcl doesn't have either of those things.  Braces are used,
# though, to escape special characters in procedure bodies and in strings that
# are formatted as lists.


################################################################################
## 2. Syntax
################################################################################
```

```
# Every line is a command.  The first word is the name of the command, and
# subsequent words are arguments to the command. Words are delimited by
# whitespace.  Since every word is a string, in the simple case no special
# markup such as quotes, braces, or backslash, is necessary.  Even when quotes
# are used, they are not a string constructor, but just another escaping
# character.

set greeting1 Sal
set greeting2 ut
set greeting3 ations


#semicolon also delimits commands
set greeting1 Sal; set greeting2 ut; set greeting3 ations


# Dollar sign introduces variable substitution
set greeting $greeting1$greeting2$greeting3


# Bracket introduces command substitution.  The result of the command is
# substituted in place of the bracketed script.  When the "set" command is
# given only the name of a variable, it returns the value of that variable.
set greeting $greeting1$greeting2[set greeting3]


# Command substitution should really be called script substitution, because an
# entire script, not just a command, can be placed between the brackets. The
# "incr" command increments the value of a variable and returns its value.
set greeting $greeting[
    incr i
    incr i
    incr i
]


# backslash suppresses the special meaning of characters
set amount \$16.42


# backslash adds special meaning to certain characters
puts lots\nof\n\n\n\n\n\nnewlines


# A word enclosed in braces is not subject to any special interpretation or
# substitutions, except that a backslash before a brace is not counted when
# looking for the closing brace
set somevar {
    This is a literal $ sign, and this \} escaped
    brace remains uninterpreted
}


# In a word enclosed in double quotes, whitespace characters lose their special
# meaning
set name Neo
set greeting "Hello, $name"


#variable names can be any string
set {first name} New


# The brace form of variable substitution handles more complex variable names
set greeting "Hello, ${first name}"


# The "set" command can always be used instead of variable substitution
set greeting "Hello, [set {first name}]"


# To promote the words within a word to individual words of the current
# command, use the expansion operator, "{*}".

set {*}{name Neo}

# is equivalent to
set name Neo


# An array is a special variable that is a container for other variables.
set person(name) Neo
set person(gender) male
set greeting "Hello, $person(name)"


# A namespace holds commands and variables
namespace eval people {
    namespace eval person1 {
        variable name Neo
    }
}


#The full name of a variable includes its enclosing namespace(s), delimited by two colons:
set greeting "Hello $people::person1::name"



################################################################################
## 3. A Few Notes
################################################################################

# All other functionality is implemented via commands.  From this point on,
# there is no new syntax.  Everything else there is to learn about Tcl is about
# the behaviour of individual commands, and what meaning they assign to their
# arguments.
# To end up with an interpreter that can do nothing, delete the global
# namespace.  It's not very useful to do such a thing, but it illustrates the
# nature of Tcl.
namespace delete ::


# Because of name resolution behaviour, it's safer to use the "variable" command to
# declare or to assign a value to a namespace. If a variable called "name" already
# exists in the global namespace, using "set" here will assign a value to the global variable
# instead of creating a new variable in the local namespace.
namespace eval people {
    namespace eval person1 {
        variable name Neo
    }
}


# The full name of a variable can always be used, if desired.
set people::person1::name Neo



################################################################################
## 4. Commands
################################################################################

# Math can be done with the "expr" command.
set a 3
set b 4
set c [expr {$a + $b}]

# Since "expr" performs variable substitution on its own, brace the expression
# to prevent Tcl from performing variable substitution first.  See
# "http://wiki.tcl.tk/Brace%20your%20#%20expr-essions" for details.


# The "expr" command understands variable and command substitution
set c [expr {$a + [set b]}]


# The "expr" command provides a set of mathematical functions
set c [expr {pow($a,$b)}]


# Mathematical operators are available as commands in the ::tcl::mathop
# namespace
::tcl::mathop::+ 5 3

# Commands can be imported from other namespaces
namespace import ::tcl::mathop::+
set result [+ 5 3]
```

```
# New commands can be created via the "proc" command.
proc greet name {
    return "Hello, $name!"
}

#multiple parameters can be specified
proc greet {greeting name} {
    return "$greeting, $name!"
}


# As noted earlier, braces do not construct a code block.  Every value, even
# the third argument of the "proc" command, is a string.  The previous command
# rewritten to not use braces at all:
proc greet greeting\ name return\ \"Hello,\ \$name!



# When the last parameter is the literal value, "args", it collects all extra
# arguments when the command is invoked
proc fold {cmd args} {
    set res 0
    foreach arg $args {
        set res [$cmd $res $arg]
    }
}
fold ::tcl::mathop::* 5 3 3 ;# ->  45


# Conditional execution is implemented as a command
if {3 > 4} {
    puts {This will never happen}
} elseif {4 > 4} {
    puts {This will also never happen}
} else {
    puts {This will always happen}
}


# Loops are implemented as commands.  The first, second, and third
# arguments of the "for" command are treated as mathematical expressions
for {set i 0} {$i < 10} {incr i} {
    set res [expr {$res + $i}]
}


# The first argument of the "while" command is also treated as a mathematical
# expression
set i 0
while {$i < 10} {
    incr i 2
}
```

```tcl
# A list is a specially-formatted string.  In the simple case, whitespace is sufficient to delimit valu
set amounts 10\ 33\ 18
set amount [lindex $amounts 1]


# Braces and backslash can be used to format more complex values in a list.  A
# list looks exactly like a script, except that the newline character and the
# semicolon character lose their special meanings.  This feature makes Tcl
# homoiconic.  There are three items in the following list.
set values {

    one\ two

    {three four}

    five\{six

}


# Since a list is a string, string operations could be performed on it, at the
# risk of corrupting the formatting of the list.
set values {one two three four}
set values [string map {two \{} $values] ;# $values is no-longer a \
    properly-formatted listwell-formed list


# The sure-fire way to get a properly-formmated list is to use "list" commands
set values [list one \{ three four]
lappend values { } ;# add a single space as an item in the list


# Use "eval" to evaluate a value as a script
eval {
    set name Neo
    set greeting "Hello, $name"
}


# A list can always be passed to "eval" as a script composed of a single
# command.
eval {set name Neo}
eval [list set greeting "Hello, $name"]


# Therefore, when using "eval", use [list] to build up a desired command
set command {set name}
lappend command {Archibald Sorbisol}
eval $command


# A common mistake is not to use list functions when building up a command
set command {set name}
append command { Archibald Sorbisol}
```

```
eval $command ;# There is an error here, because there are too many arguments \
    to "set" in {set name Archibald Sorbisol}


# This mistake can easily occur with the "subst" command.
set replacement {Archibald Sorbisol}
set command {set name $replacement}
set command [subst $command]
eval $command ;# The same error as before: too many arguments to "set" in \
    {set name Archibald Sorbisol}


# The proper way is to format the substituted value using use the "list"
# command.
set replacement [list {Archibald Sorbisol}]
set command {set name $replacement}
set command [subst $command]
eval $command


# It is extremely common to see the "list" command being used to properly
# format values that are substituted into Tcl script templates.  There are
# several examples of this, below.


# The "apply" command evaluates a string as a command.
set cmd {{greeting name} {
    return "$greeting, $name!"
}}
apply $cmd Whaddup Neo


# The "uplevel" command evaluates a script in some enclosing scope.
proc greet {} {
    uplevel {puts "$greeting, $name"}
}

proc set_double {varname value} {
    if {[string is double $value]} {
        uplevel [list variable $varname $value]
    } else {
        error [list {not a double} $value]
    }
}


# The "upvar" command links a variable in the current scope to a variable in
# some enclosing scope
proc set_double {varname value} {
    if {[string is double $value]} {
        upvar 1 $varname var
        set var $value
    } else {
        error [list {not a double} $value]
    }
}


#get rid of the built-in "while" command.
rename ::while {}


# Define a new while command with the "proc" command.  More sophisticated error
# handling is left as an exercise.
proc while {condition script} {
    if {[uplevel 1 [list expr $condition]]} {
        uplevel 1 $script
        tailcall [namespace which while] $condition $script
    }
}


# The "coroutine" command creates a separate call stack, along with a command
# to enter that call stack. The "yield" command suspends execution in that
# stack.
proc countdown {} {
    #send something back to the initial "coroutine" command
    yield

    set count 3
    while {$count > 1} {
        yield [incr count -1]
    }
    return 0
}
coroutine countdown1 countdown
coroutine countdown2 countdown
puts [countdown 1] ;# -> 2
puts [countdown 2] ;# -> 2
puts [countdown 1] ;# -> 1
puts [countdown 1] ;# -> 0
puts [coundown 1] ;# -> invalid command name "countdown1"
puts [countdown 2] ;# -> 1
```

Reference

Official Tcl Documentation

Tcl Wiki

Tcl Subreddit