Tcl Primer

The Tcl programming language started from a need for a general purpose scripting language, to enable users to extend the functionality of their applications with custom routines. The name reflects this: it stands for "Tool command language." The Tcl interpreter presented here, is a reduced version of standard Tcl.[1] It is based on ParTcl,[2] by Serge Zaitsev, but with various modifications. Both Serge Zaitsev's original implementation and my modified version are available on GitHub. See the end of this document for links.

This chapter starts with a tour over the concepts and the "idiom" of the language. After that, separate sections on specific elements of the language give more details on these elements.

Syntax

The overall syntax of Tcl resembles that of Unix shell scripts. Instructions are lists of words that are separated by spaces. Strings between double quotes are also considered a "word" in the Tcl syntax —or more accurately, it is the other way around: words are strings, even if not between quotes. In interpreting it, the first word in a list is the *command*, with the other words as its arguments.

Double quotes are needed when a word contains a space or any other special characters. Tcl also offers another means of grouping words or "quoting" words: curly braces. Strings enveloped by curly braces can be nested, creating strings inside other strings —or lists of strings. A difference between the two forms of quoting is that in a double-quoted string, variables are substituted by their contents, but this does not happen in a brace-enveloped group.

```
set age 54
puts "I am $age years old"
puts {I am $age years old}
```

[1] The language has grown since it humble beginnings, and it is now increasingly used to create applications and utilities —rather than serving as an auxiliary embedded component of said applications.

[2] Tcl is commonly pronounced as "tickle"; the name ParTcl is a pun on this convention.

In the above example, if it were run, the first `puts` command prints "I am 54 years old," but the second prints the argument verbatim.

The above snippet has three instructions. The way Tcl goes through each, is in two stages. First, it collects the words for an instruction into a list, and then it evaluates (or interprets) that list, before proceeding with the next instruction. These phases are called *parsing* and *execution* respectively. Tcl moves from parsing to execution when it sees either a line end (*newline*) or a semicolon (";"). So when putting several commands on a single line, a semicolon is needed to separate them. There is an exception for newlines and semicolons inside quoted strings and brace-enveloped groups: these are not considered execution points.

Another key concept of Tcl is substitution. As the snippet above shows, a variable name prefixed with a "$" is replaced by its contents (except inside curly braces). Moreover, text between square brackets is replaced by the result of interpreting that text as a command.

```
set age 54
puts "It's [expr 65 - $age] more years until retirement"
```

The section "`expr 65 - $age`" is first extracted and interpreted. That is, the command `expr` is executed with the argument "`65 - $age`." The result of this simple calculation is then inserted at that position. Here it is a numeric result, but the same principle applies to commands that return strings.

There is no *assignment* operator in Tcl; the `expr` command only evaluates expressions and returns the result, and the `set` command sets a variable. The correct way to change the value of a variable, is like in the following:

```
set a 5
set a [expr $a * 2]
```

Control structures follow the syntax of commands. The following snippet swaps variables "a" and "b" if the former is greater than the latter:

```
if {$a > $b} {
    set tmp $a
    set a $b
    set b $tmp
}
```

The `if` command is a built-in command, and implemented such that it always evaluates the first argument as an expression. It is therefore not necessary to write it as "`if [expr {$a > $b}]`" (though this is still allowed). Also note in the sequence of `set` commands, that you should only use the "$" prefix when referring to the value of the variable, not when referring to the variable name.

The body of the `if` statement is a brace-enveloped group. The Tcl interpreter passes the entire content to the `if` command, as-is. The `if` command then decides (based on the condition in its first argument) whether to evaluate it, or whether to ignore it.

The rule for when to move from parsing phase to execution, is important for the coding style, notably the placement of braces. As written above, a newline or a semicolon mark an execution point, unless these appear inside a string or a group. The first line of the `if` snippet ends with a "`{`." Therefore, a brace-enveloped group has started and the newline that follows the "`{`" is *not* an execution point. Instead, Tcl reads up to the closing brace, and only then executes the `if` command.

The upshot is that we are thus not free to choose brace placement, as we are in C, Javascript, and many other programming languages. Also, white space between words and/or grouped blocks is often significant: there must be a space (or TAB) after a command name, for example; writing `if{$a < $b}` is incorrect (no space between `if` and the opening brace).

```
proc factorial x {
    if {$x == 1} { return 1 }
    return [expr {$x * [factorial [expr $x-1]]}]
}

factorial 4
```

The `proc` command adds a user procedure to the list of commands. It takes three arguments: the name for the new command, the parameter list, and the body for the user procedure. The parameter list is a series of names between curly braces (but if there is only one name in the parameter list, the curly braces may be omitted).

Variables must be set before being used, and variables that are inside a procedure are local to that procedure. When a procedure must keep global state, it must explicitly declare a variable as global.

```
proc random {} {
    # middle-square method to generate 4-digit numbers
    global random_seed
    set random_seed [expr {($random_seed ** 2 / 100 + 1234) % 10000}]
}

puts [random]
puts [random]
puts [random]
```

The `global` command marks variables that come behind it as global. The variable may already exist at global scope, but otherwise the `global` command creates it with an initially empty value.

A comment starts with a "`#`" and runs up to the end of the line, and must be placed on a line of its own, or following a semicolon.

Also note that the `random` procedure lacks a `return` command. If an explicit `return` command is lacking, the return value of a procedure is the result of the last command. Therefore, for simple cases like these, no `return` command is needed.

A final remark on the general syntax of Tcl is, that Tcl is case-sensitive. The built-in commands are all in lower case, and it would be an error to use `SET` instead of `set`. For your own user procedures and variables, you are free to choose the name, but you have to stick to it throughout the script.

Flow Control Structures

Tcl has various built-in control structures. The `if` was already briefly introduced, as a command that takes a condition and a brace-enveloped group of commands. It can, however, take a variable number of arguments. The complete syntax is:

```
if { condition } then {
    body
} elseif { condition } then {
    body
} else {
    body
}
```

A condition is *true* in Tcl if it evaluates to a non-zero value, when interpreted as a numerical expression.

The literal words `then`, `elseif` and `else` are all optional. You may insert them for clarity, or omit them for brevity. In practice, the `then` is traditionally omitted, but the `elseif` and `else` are put in. There may be any number of `elseif` blocks.

The `switch` command selects a body to execute, based on pattern matching. There are two syntaxes for the command; below is the most common one:

```
switch value {
    pattern {
        body
    }
    pattern {
        body
    }
    default {
        body
    }
}
```

The patterns can use wildcard characters "*," "?" and ranges between square brackets. The value is matched to each of the patterns, and on the first match, the relevant body is executed. All other bodies are skipped. If none of the patterns match, the `default` body executes. The `default` pseudo-pattern is optional; if it is present, it must be the last.

If a "-" follows the pattern (instead of a list of commands in curly braces), that body is a "fall-through" to the next body. This allows you to have a single instruction body for several patterns. The body (in curly braces) is set in the last of the patterns, and the patterns above it have a "-" behind the pattern.

The basic command for loops is (`while` loop):

```
while { condition } {
    body
}
```

The loop keeps running the commands in its body as long as the condition is true. There are, however, a few other instructions that break out of loops. The `break` command causes a jump out of the innermost enclosing loop, and proceeds running at the command below the loop. All commands inside the loop body that follow the `break` are skipped. The `continue` command is similar to `break` (in that it skips all remaining commands in the loop body), but it jumps back to the loop condition. If the condition is still true, the loop will then continue.

The `break` command is also similar to `return`. In a way, `return` breaks out of procedures, quite like how `break` breaks out of loops. A final command that breaks out of the entire script is `exit` —it aborts running. Like `return`, `exit` may specify a return code.

As it is common for a loop to have a fixed number of iterations, there is a special construct for it:

```
for { setup } { condition } { post } {
    body
}
```

The instruction in "setup" is only evaluated once, before entering the loop. The condition has the same function as in the `while` loop: the `body` is only evaluated (i.e. executed) when the condition is true. After the body runs, the `for` command first evaluates the "post" word, before proceeding to the condition —to evaluate whether the loop must run another iteration. A typical use case is:

```
set total 0
for {set count 1} {$count <= 10} {incr count} {
    set total [expr $total + $count]
}
```

This loop runs ten times: the count variable starts at 1 and is incremented by 1 after every iteration.

The `break` and `continue` instructions can be used for the same purpose in a `for` loop as in the `while` loop, with the caveat that `continue` jumps to the "post" argument of the loop, rather than directly to the condition.

The last control structure is `foreach`, which loops over all items in a list. On every iteration, the variable in the first argument of the list is set to the consecutive item from the list.

```
set words [list the quick brown fox]
foreach w $words {
    puts $w
}
```

Numbers

Although the basic type in Tcl is a string, when arithmetic needs to be performed, these words are interpreted as numbers. Tcl supports three number bases:

- Decimal: a series of digits (between `0` and `9`), *not* starting with a `0`.
- Octal: a series of digits between `0` and `7`, prefixed with a `0`.
- Hexadecimal: a series of digits between `0`...`9` and between `A`...`F`, prefixed with `0x`. Hexadecimal numbers are *not* case-sensitive, so you may use `a`...`f` instead of upper case letters.

Lists and Strings

Lists were mentioned a few times, like how instructions are a list of words —the first word is the command and the successive words are its arguments. A list is not an explicit data structure in Tcl. Rather, lists are strings that are formatted in a particular way. More concretely, a list contains words that are separated by a space. A "word" in Tcl is a sequence of letters and/or digits, or a group of words enclosed in curly braces (or on occasion, enclosed by double quotes or square brackets).

There is, in essence, no difference between a string and a list. However, the distinction is made because Tcl offers a separate set of commands for list manipulation and for string manipulation.

Variables and Arrays

Simple variables have already been used in the snippets presented so far. A variable has a name and a value. Tcl attributes no type to the value; it can contain text or a number —or a list. Tcl imposes few restrictions on the variable name; a name like "`has-completed?`" would be invalid in most programming languages, but is perfectly valid in Tcl. Even spaces are permitted if you wrap the name in curly braces, like in "`{top level}`." When using the value of such a variable, put the "`$`" before the opening brace: "`${top level}`."

Yet, such special variable names are not recommended when the variables might also be used in expressions of the `expr` command. The infix expression evaluator has its own syntax, and variables with characters that conflict with operators, may confuse the evaluation.

When the variable name is prefixed with a "`$`," it is substituted by its value. From this follows that a variable must exist before it can be used.

Variables are created automatically when you set them, either with `set` or with another command that sets a variable. Variables set inside a use procedure are created as local variables; these cease to exist once the procedure ends. To access a global variable from within a user procedure, the variable needs to be declared inside the procedure with the `global` command. One or more variable names follow the `global` keyword. The `global` command creates a reference to each of the specified variables, but if the variable does not yet exist at the global level, it first creates it (with empty content). Once the global variable exists, it is not re-created or re-initialized by the `global` command.

A variable name can have an index appended. The index is a positive number between parentheses. The lowest valid index is zero. This allows a variable to have multiple values, each at a unique index.

```
for {set i 0} {$i < 10} {incr i} {
    set series($i) [expr 2 * $i]
}
```

Only single-dimension arrays are supported. In the full Tcl language, indices may be any text,[3] and their implementation is actually that of an associative *map*. In ParTcl, however, arrays need to be indexed with numbers.

Non-text data is often easier to process as an array. The `array` command enables conversion of text and binary data to an array of values. See the section Binary data (on page 12) for details.

Expressions

The `expr` command evaluates arithmetic expressions, like addition and multiplication. There are also relational and logical operators, and operators for bit twiddling. ParTcl does all arithmetic in integers; it does not support floating point. The relational operators ("==", "`!=`", "<" etc.) can compare strings (case-sensitive), but for matching with wild-cards or case-insensitive comparison, you need to use the `string` command instead.

There is no *assignment* operator; in Tcl you need to use the `set` command to assign a value to a variable. A few code snippets on preceding pages have already illustrated this —see for example the body of the `for` loop in the snippet on page 6.

The operator table (and precedence levels) of ParTcl are below:

`- + ! ~ ()`	unary operators: negate, unary plus (a no-operation operator), logic not, binary invert, and sub-expressions between parentheses
`**`	exponentiation
`* / %`	multiply, divide, remainder after division
`+ -`	addition, subtraction
`<< >>`	binary shift left & shift right
`< <= > >=`	smaller than, smaller than or equal, greater than, greater than or equal
`== !=`	equal, not equal
`&`	binary and

[3] Using this property, multi-dimensional arrays are simulated by a convention of joining indices with a "." separator.

^	binary exclusive or
|	binary or
&&	logic and
||	logic or
? :	conditional selection (ternary operator)

ParTcl (like Tcl) uses "floored" integer division. For positive numerators and denominators, floored division gives the same results as the (more common) truncated division: "14 / 3" is truncated to 4 (and with remainder 2). The difference is with negative results: "-14 / 3" with *floored* division gives -5 with remainder 1. Floored division is defined such that the remainder is always a positive value.

User procedures

The `proc` command takes three parameters. The first is the name for the user procedure. It is followed by a parameter list, which is a Tcl list of parameter names. These parameters are local variables inside the procedure. When creating a procedure without any parameters, you must explicitly declare an empty parameter list with `{}`. See also the examples on page 3.

The final parameter of the `proc` command it the *body*, which is a list of commands that will be evaluated when the user procedure is called.

```
proc name { parameters } {
    body
}
```

The user procedure itself results in a value. This can be explicitly given with the `return` command, which sets the "outcome" for the procedure and exits it. Alternatively (and quite common in Tcl code) is to use a `set` command just before the end of the procedure's body, since the result of the procedure is the value of the last command that ran. If a variable already has the correct value, you can skip the second argument of the `set` command, as the snippet below illustrates.

```
# Greatest Common Divisor, by means of Euclid's algorithm
proc gcd {p q} {
    while {$q != 0} {
        set r [expr {$p % $q}]
        set p $q
        set q $r
    }
    set p
}
```

On the last line of the body, there is no need to say "`set p $p`" —when called with a single argument, `set` returns the value of the variable *without* changing it.

Note, though that the "last command that ran" may not be the last command in the body. If, in the above snippet, the final "`set p`" were omitted, the last statement that ran, would be the evaluation of "`$q != 0`" in the `while` loop (via an implicit `expr` command) —and for the case that this expression evaluates to zero.

Parameters in the parameter list may specify a default value. This allows you to have optional arguments, when calling the user procedure. All parameters with a default value must be at the end of the parameter list; that is, when a parameter has a default value, any parameters that follow it must also specify a default value. To set a default value, enclose the parameter name and its default value in curly braces. In the example below, the user procedure `pow` takes either one or two arguments, and if no argument is passed for `exp`, this parameter is set to 2.

```
proc pow {base {exp 2}} {
    expr $base ** $exp
}
```

The special parameter name "`args`" collects all arguments beyond the fixed arguments. Thus, the user procedure accepts a variable argument list.

```
proc sum args {
    set total 0
    foreach v $args {
      incr total $v
    }
    set total
}
```

The `args` parameter must appear last in the parameter list —if used at all. It may be the only parameter, as in the above example. A variable argument list may be used in combination with parameters that have default values. The `args` parameter itself may not specify a default value.

Procedure parameters and variables set inside a user procedures are local to the frame of the procedure. A user procedure can access variables in frames lower down in the call chain with the `upvar` command. This command is mostly used to implement "pass-by-reference" arguments, where a procedure can modify a variable that is passed as a parameter.

```
proc decr {name {count 1}} {
    upvar $name var
    incr var [expr {-$count}]
}
```

The `decr` procedure is declared to take a name and (optionally) a count. It then uses the `upvar` command to create a local variable ("`var`") as a reference to a variable with the given name and which is one level lower than the frame for the `decr` command itself. Any operation on `var` in fact reads or writes the variable that it references.

```
set counter 10
decr counter
```

On the call to `decr`, the name "`counter`" is passed in. Thus, inside the `decr` procedure, `$name` equals to "`counter`," and `upvar` binds this name to local variable `var`. However, when changing `var`, it is actually `counter` that is modified.

This example illustrates the most common use of `upvar`, but it is more flexible. The `upvar` command may refer to a variable two (or more) levels up, and it may refer to an absolute frame level. To do this, the level may be specified as the first parameter after the command:

```
upvar level name variable...
```

When the level is a number, it is a *relative* from the current level; when it starts with a "`#`", the value behind it is taken as the *absolute* level. The usual case is `#0`, meaning the global level.

The last point is that pairs of reference names and local variable names may be declared on a single `upvar` command. This allows you to create multiple references at once (on the same level). The reference variables must exist on the targeted level —unlike the `global` command, the `upvar` command does not *create* variables at a lower level.

Comments

The "`#`" character starts a comment, which runs up to the end of the line. However, a comment may only appear at an "execution point," which is either after a newline or after a semicolon. In practice, it means that you can place a comment on a line of its own, or alternatively —if you want to add a trailing comment behind a command, place a semicolon in front of the `#`.

Exception handling

A run-time exceptions occurs when attempting to perform an operation that cannot proceed, such as opening a file that does not exist, or using a variable that was never set. With Tcl, you would normally be able to avoid such errors —for example, by first checking the existence of a file before opening it, with the "`file exists`" command. However, it is often more convenient to catch the exception and handle it when it occurs.

```
if [catch {set fd [open $filename]} errmsg] {
    puts "Error: $errmsg"
} else {
    puts [read $fd]
}
```

The `catch` command evaluates its first argument, which is the command "`set fd [open $filename]`". This in turn evaluates the nested command "`[open $filename]`" first. If variable `filename` does not hold a name of a file that can be opened, the `open` command throws an exception. The exception cascades through the `set` command, and would eventually abort the script —but it the `catch` command stores the error message in the `errmsg` variable (the second argument to `catch`) and clears the error.

The result of the `catch` command indicates whether an exception occurred. It returns one of the following values:

- `0` Normal return.
- `1` Error or exception occurred, `$errorInfo` holds the message.
- `2` Abort due to a `return` or `exit` statement.
- `3` Abort due to a `break` statement.
- `4` Abort due to a `continue` statement.

In practice, `catch` will not return values 2, 3 or 4 for well-written code, because these cases have already been handled by procedure and loop commands. However, if you use `break` outside a loop, `catch` may indeed... well, *catch* that.

In your own code, you may throw an exception with the `error` command. This command takes a message as an argument, which is the message that `catch` will subsequently store in the variable.

Binary data

Tcl offers two ways to extract values from binary data: the `array` and `binary` commands. If you have a chunk of binary data in a variable called "`blob`," you can convert it to a byte array with:

```
set blob 12345
array slice data $blob
```

The "data" variable will have as many entries as the length of the contents of "blob." Each entry in data has the value of the respective byte in blob. In this example, data has five entries, from data(0) to data(4); where data(0) is set to 49, data(1) to 50, and so forth up to 53 for data(4). In other words, the first character of blob is "1," which has ASCII code 49, and thus 49 is stored in the first array element.

The array slice command can also chop up the blob in chunks of 16, 24, 32 or 64 bits. The "word size" (1 to 8) can be optionally appended at the end of the command. When the word size is not 1, the default is that the data is sliced in Little-Endian order (low byte first). Optionally, the argument "be" can be appended behind the word size, for Big-Endian byte order.

When the binary data has a mix of fields with different sizes, the binary command is suitable. The command takes a "format" argument that allows you to specify the type, size and count of each subsequent field. There are two sub-commands: format is to pack Tcl values into a binary blob, and scan is to unpack a binary blob into Tcl variables.

The snippet below illustrates the binary scan command, to interpret a blob as a series of bytes, and set variable data to a list of individual byte values. After the command, data is set to {49 50 51 52 53}.

```
set blob 12345
binary scan $blob cu* data
```

In this example, array slice and binary scan perform essentially the same function —the only difference is that binary scan creates a list, rather than an array. However, the format string ("cu*" in the above example) offers a lot of flexibility. For example, the "Read Input Registers" frame from the MODBUS-RTU protocol can be decoded with the following format pattern:

```
# suppose blob contains the byte sequence 01 04 00 00 00 02 71 cb
binary scan $blob cucSSsu address function register number crc
```

The device address, function code, register start address, register count and CRC that are extraced from the blob, are all stored in separate variables. The format string has a field for each of the five variables that follow. The first letter gives the size and byte order of the field.

- c 8-bit integer
- s 16-bit integer
- i 32-bit integer
- w 64-bit integer

When this letter is upper case, the field is set in Big-Endian; otherwise it is set in Little-Endian. The upper case "C" is undefined, because byte-order is irrelevant for a single-byte field.

The letter "u" may follow the leader letter, to indicate that the integer has an unsigned value (the default is signed). After that, a number may follow for the count of these integers to store in the respective variable. If there is a "*" instead of a number, it means that it runs up to the end of the data in the binary blob.

Referring tp the MODBUS-RTU example, `address` is an 8-bit unsigned integer ("cu"), whereas `function` is an 8-bit *signed* integer ("c"); `register` and `number` are both 16-bit signed integers in Big-Endian ("S"); and `crc` is a 16-bit unsigned integer in Little-Endian ("su").

Built-in commands

Several of the built-in commands have subcommands —for example, the `file` and `string` commands. In the following table, these subcommands are listed separately.

A few other command accept switches. Switches are optional parameters that change the operation of the command. An example is the `puts` command. It normally ends the argument that it prints with a newline; however, when the switch `-nonewline` is added to the command, `puts` prints the argument without newline.

Switches must be placed after the command, but before the first normal argument. If a command takes a subcommand, the switches must be placed after the subcommand. If a switch appears at an incorrect position, or if a switch is not recognized as valid, it is taken to be a normal argument.

append *var word*	Append contents to a variable (concatenate strings).
array size *var*	Return the number of elements in the array.
array length *var*	Same as `array size`.
array slice *var word*	Slice the word into bytes or multi-byte fields, and store the values (when interpreted as binary data) into array elements. See page 12.
array split *var word*	Split the string on a separator and store the elements in an array. If no *sep*arator is given, the string is split on whitespace.
array split *var word sep*	

binary format *fmt arg ...*	Return a string with the binary representation of the arguments, and according to the type specifications in the *fmt* parameter.
binary scan *word fmt var ...*	Extract values from binary data in *word*, according to the type specifications in the *fmt* parameter, and store these in the variables listed at the tail of the command. See page 13.
break	Abort a loop, jumps to the first instruction below the loop.
catch *body* **catch** *body var*	Evaluate the body, catch any error; return 0 if the body evaluated normally, and 1 if an exception occurred. The error message is stored in variable *var* (if given). See page 12.
clock seconds	Return the number of seconds since the Unix Epoch (00:00:00 January 1st, 1970).
clock format *time format*	Format time and date according to the *format* string. The *time* parameter is the number of seconds since the start of the Unix Epoch.
close *file*	Close the file indicated by the file handle.
concat *word ...*	Join multiple lists into a single list.
continue	Skip the remainder of the loop body, jumps back to the start of the loop.
eof *file*	Return 1 if the file (specified by handle) is at its end.
error *msg*	Set an exception or error, which aborts execution (but which can be caught with **catch**). See page 12.
exec *word ...*	Run the parameter as an executable in the shell, with any additional words as the arguments to the program. The command returns the console output of the program.
exit **exit** *word*	End the script with an optional return code. Note that this command aborts the script, but not the application.
expr *expression*	Interpret the infix expression that follows. It supports only integer arithmetic. See page 8.
file dirname *path*	Return the directory part of the path.
file exists *path*	Return whether the path exists (0 or 1).
file extension *path*	Return the file extension of the path.
file isdirectory *path*	Return whether the path refers to a directory (0 or 1).
file isfile *path*	Return whether the path refers to a regular file (0 or 1).
file rootname *path*	Return the part of the path *without* extension.
file size *path*	Return the size of the file.
file tail *path*	Return the base name of the path (*without* directory).
flush *file*	Flush buffered data to the file.
for *setup cond post body*	Evaluate *setup*, then run *body* in a loop as long as *cond* stays true. At the end of every iteration, *post* is evaluated. See page 5.
foreach *var list body*	Run a loop over all elements in *list*. Each time that *body* is evaluated, *var* is set to the next element from *list*. See page 6.

format *string word ...*	Format a string with placeholders, similar to `sprintf` in C. Currently "%c," "%d," "%i," "%x" and "%s" are supported, plus optional padding and alignment modifiers (for example "%04x" or "%-20s").
gets *file*	Read a single line from the file; the trailing newline is stripped.
global *var ...*	Mark any variable following it as a global variable. Multiple names may follow, separated by spaces.
if *cond* **then** *body* **elseif** *cond* **then** *body* **else** *body*	Conditional execution of *body*. The keywords **then**, **elseif** and **else** are optional. See page 4.
incr *var value*	Increment a variable by *value*. If the *value* parameter is omitted, *var* is incremented by 1.
info exists *var* **info tclversion**	Return 1 if the variable exists, and 0 otherwise. Return the version of the Tcl interpreter.
join *list* **join** *list separator*	Create a string from a list, by concatenating elements, with a separator chosen by the user. If the *separator* parameter is omitted, a space is used for separation.
lappend *var word ...*	Append values to a variable (where the variable is presumed to contain a list).
lindex *list index*	Return a specified element from the list (parameter *index* must contain a valid element number, between 0 and the list length minus 1).
linsert *list index word ...*	Insert elements in a list in front of the value of *index*. The first list element has index 0.
list *word ...*	Create a list from the values that follow it.
llength *list*	Return the number of elements in a list.
lrange *list first last*	Return the elements *first* to *last* (inclusive) of *list* as a new list. Parameter *last* may be set to "end" (instead of a number) to indicate the end of the list.
lreplace *list first last ...*	Delete the elements *first* to *last* (inclusive) from *list* and insert a set of elements at that position. If there are no new elements behind parameter *last*, it deletes the elements between *first* and *last*.
lsearch *list pattern*	Find the index of the first element in the list that matches the pattern. The *pattern* argument may contain wildcard characters "*" and "?", or character sets or ranges between square brackets. However, if the switch `-exact` is set, wildcards are disabled and all characters must match.
lsort *list*	Sort the elements of a list, returning a sorted list. The default is an alphabetic sort in increasing order; switches `-integer` and `-decreasing` toggle these settings.
open *path mode*	Open a file and return a file handle (this file handle must be passed to other file commands, like **close** or **puts**).
proc *name* args body	Create a new (user-defined) command. See page 9.
puts *word*	Print the argument to the stdout.

puts *file word*	Print the argument to the *file* (handle). The command ends the output with a newline, unless the option `-nonewline` is set.
read *file* **read** *file count*	Read the complete file as a string. Read a maximum of *count* bytes from the file and return it as a string. If the switch `-nonewline` is set, a final newline character (if any) is stripped from the returned data.
return **return** *word* **scan** *word format var ...*	Jump out of the current command ("proc"), with an optional explicit return value. Parse a string and stores extracted values into variables. This command currently supports "%c," "%d," "%i" and "%x" placeholders, plus optional "width" modifiers (for example "%2x").
seek *file position* **seek** *file position whence*	Set the read/write position of the file. The *whence* parameter can be set to `current` or to `end` to move the file position relative to these markers.
set *var word*	Assign a value to the variable, and return this value. If no *word* parameter is present, the current value is returned.
source *path*	Read the file and evaluates it as a Tcl script. It returns the value of the last command in the file, or that of a **return**.
split *word* **split** *word separator*	Create a list from a string, by splitting the string on a separator chosen by the user. If no *separator* is given, the string is split on whitespace.
string compare *word word*	Compare two strings, returns an order ranking value (0 if both strings are equal).
string equal *word word*	Test strings for equality (returns 1 if equal, 0 otherwise).
string first *word sub skip*	Finds the first occurrence of *sub* in *word*, skipping the first *skip* characters in *word*.
string index *word value*	Return the character in *word* at the given index.
string last *word sub skip*	Finds the last occurrence of *sub* in *word*, skipping the last *skip* characters from the end of *word*.
string length *word*	Return the length (in characters) of the string.
string match *pattern word*	Return 1 if the pattern matches the word, and 0 otherwise. The pattern may use "*" and "?" wildcards, plus character sets or ranges between square brackets.
string range *word first last*	Return a string that has the range of characters between *first* and *last* (inclusive). Parameter *last* may be set to "end" to indicate the end of the string.
string tolower *word*	Returns the *word* string in lower case.
string toupper *word*	Returns the *word* string in upper case.
string trim *word charset*	Removes characters in *charset* from the start and end of the string. The *charset* parameter defaults to whitespace.
string trimleft *word chars*	Like **trim**, but only trim the start of the string.
string trimright *word chars*	Like **trim**, but only trim the end of the string.
subst *word*	Perform command and variable substitution in the parameter.

switch *word* **{** *pattern body* *pattern body* **default** *body* **}**	Control flow structure, executing a block selected from matching one out of several patterns. The patterns may use "*" and "?" wildcards, plus sets or ranges between square brackets. However, the `-exact` switch disables wildcards. The **default** clause is optional (it is taken if none of the patterns match). See page 4.
tell file	return the current position of the file.
unset *var ...*	Clear variables (remove the given variables completely).
upvar *name var ...* **upvar** *level name var ...*	Create a "reference variable" to a variable at a lower scope, so that setting the local variable changes the value of the referenced variable. See page 10.
while *cond body*	Run a loop as long as the condition is true. If the condition is already false on start, the body is never evaluated.

Further Reading

This primer is brief for a reason: so much fine information on Tcl is already available in on-line tutorials and books. John Ousterhout, creator of Tcl, wrote a very readable, and comprehensive book on it: **Tcl and the Tk Toolkit**; ISBN 0-201-63337-X. The draft of this book is freely available on: http://csis.pace.edu/~benjamin/software/book1.pdf

Old books are fine, because, as stated earlier, ParTcl actually draws back to the roots of Tcl: as a light-weight extension language for applications. It is closer (in syntax and semantics) to Tcl versions before 7.0 than to the current 8.6 release.

Keep in mind that the expression parser in ParTcl is integer-only. There is no floating point arithmetic, and operators that function on lists are also unavailable. Another limitation of ParTcl is that arrays in ParTcl must be indexed with a number (equal to, or greater than zero); non-numeric array indices are not supported.

Copyright