EDOC202

EUROGAM PROJECT

NSF DATA ACQUISITION SYSTEM

Tool Command Language

Edition 1.0

November 1992

NSF Software Systems Group

UK Science and Engineering Research Council

Daresbury Laboratory

Acknowlegdements

This document is John Ousterhout's (University of California, Berkeley) "Overview of Tool Command Language Facilities" man page for his embeddable command language, "TCL." The TCL (pronounced "tickle") software forms the basis of the "eg-ssession" EuroGam user interface program.

NAME

Tcl − overview of tool command language facilities

INTRODUCTION

Tcl stands for ''tool command language'' and is pronounced ''tickle.'' It is actually two things: a language and a library. First, Tcl is a simple textual language, intended primarily for issuing commands to interactive programs such as text editors, debuggers, illustrators, and shells. It has a simple syntax and is also programmable, so Tcl users can write command procedures to provide more powerful commands than those in the built-in set.

Second, Tcl is a library package that can be embedded in application programs. The Tcl library consists of a parser for the Tcl language, routines to implement the Tcl built-in commands, and procedures that allow each application to extend Tcl with additional commands specific to that application. The application program generates Tcl commands and passes them to the Tcl parser for execution. Commands may be generated by reading characters from an input source, or by associating command strings with elements of the application's user interface, such as menu entries, buttons, or keystrokes. When the Tcl library receives commands it parses them into component fields and executes built-in commands directly. For commands implemented by the application, Tcl calls back to the application to execute the commands. In many cases commands will invoke recursive invocations of the Tcl interpreter by passing in additional strings to execute (procedures, looping commands, and conditional commands all work in this way).

An application program gains three advantages by using Tcl for its command language. First, Tcl provides a standard syntax: once users know Tcl, they will be able to issue commands easily to any Tcl-based application. Second, Tcl provides programmability. All a Tcl application needs to do is to implement a few application-specific low-level commands. Tcl provides many utility commands plus a general programming interface for building up complex command procedures. By using Tcl, applications need not reimplement these features. Third, Tcl can be used as a common language for communicating between applications. Inter-application communication is not built into the Tcl core described here, but various add-on libraries, such as the Tk toolkit, allow applications to issue commands to each other. This makes it possible for applications to work together in much more powerful ways than was previously possible.

This manual page focuses primarily on the Tcl language. It describes the language syntax and the built-in commands that will be available in any application based on Tcl. The individual library procedures are described in more detail in separate manual pages, one per procedure.

INTERPRETERS

The central data structure in Tcl is an interpreter (C type ''Tcl_Interp''). An interpreter consists of a set of command bindings, a set of variable values, and a few other miscellaneous pieces of state. Each Tcl command is interpreted in the context of a particular interpreter. Some Tcl-based applications will maintain multiple interpreters simultaneously, each associated with a different widget or portion of the application. Interpreters are relatively lightweight structures. They can be created and deleted quickly, so application programmers should feel free to use multiple interpreters if that simplifies the application. Eventually Tcl will provide a mechanism for sending Tcl commands and results back and forth between interpreters, even if the interpreters are managed by different processes.

DATA TYPES

Tcl supports only one type of data: strings. All commands, all arguments to commands, all command results, and all variable values are strings. Where commands require numeric arguments or return numeric results, the arguments and results are passed as strings. Many commands expect their string arguments to have certain formats, but this interpretation is up to the individual commands. For example, arguments often contain Tcl command strings, which may get executed as part of the commands. The easiest way to understand the Tcl interpreter is to remember that everything is just an operation on a string. In many cases

Tcl constructs will look similar to more structured constructs from other languages. However, the Tcl constructs are not structured at all; they are just strings of characters, and this gives them a different behavior than the structures they may look like.

Although the exact interpretation of a Tcl string depends on who is doing the interpretation, there are three common forms that strings take: commands, expressions, and lists. The major sections below discuss these three forms in more detail.

BASIC COMMAND SYNTAX

The Tcl language has syntactic similarities to both the Unix shells and Lisp. However, the interpretation of commands is different in Tcl than in either of those other two systems. A Tcl command string consists of one or more commands separated by newline characters or semi-colons. Each command consists of a collection of fields separated by white space (spaces or tabs). The first field must be the name of a command, and the additional fields, if any, are arguments that will be passed to that command. For example, the command

set a 22

has three fields: the first, **set**, is the name of a Tcl command, and the last two, **a** and **22**, will be passed as arguments to the **set** command. The command name may refer either to a built-in Tcl command, an application-specific command bound in with the library procedure **Tcl_CreateCommand**, or a command procedure defined with the **proc** built-in command. Arguments are passed literally as text strings. Individual commands may interpret those strings in any fashion they wish. The **set** command, for example, will treat its first argument as the name of a variable and its second argument as a string value to assign to that variable. For other commands arguments may be interpreted as integers, lists, file names, or Tcl commands.

Command names should normally be typed completely (e.g. no abbreviations). However, if the Tcl interpreter cannot locate a command it invokes a special command named **unknown** which attempts to find or create the command. For example, at many sites **unknown** will search through library directories for the desired command and create it as a Tcl procedure if it is found. The **unknown** command often provides automatic completion of abbreviated commands, but usually only for commands that were typed interactively. It's probably a bad idea to use abbreviations in command scripts and other forms that will be re-used over time: changes to the command set may cause abbreviations to become ambiguous, resulting in scripts that no longer work.

COMMENTS

If the first non-blank character in a command is **#**, then everything from the **#** up through the next newline character is treated as a comment and ignored. When comments are embedded inside nested commands (e.g. fields enclosed in braces) they must have properly-matched braces (this is necessary because when Tcl parses the top-level command it doesn't yet know that the nested field will be used as a command so it cannot process the nested comment character as a comment).

GROUPING ARGUMENTS WITH DOUBLE-QUOTES

Normally each argument field ends at the next white space, but double-quotes may be used to create arguments with embedded space. If an argument field begins with a double-quote, then the argument isn't terminated by white space (including newlines) or a semi-colon (see below for information on semi-colons); instead it ends at the next double-quote character. The double-quotes are not included in the resulting argument. For example, the command

set a "This is a single argument"

will pass two arguments to **set**: **a** and **This is a single argument**. Within double-quotes, command substitutions, variable substitutions, and backslash substitutions still occur, as described below. If the first character of a command field is not a quote, then quotes receive no special interpretation in the parsing of that field.

GROUPING ARGUMENTS WITH BRACES

Curly braces may also be used for grouping arguments. They are similar to quotes except for two differences. First, they nest; this makes them easier to use for complicated arguments like nested Tcl command strings. Second, the substitutions described below for commands, variables, and backslashes do *not* occur in arguments enclosed in braces, so braces can be used to prevent substitutions where they are undesirable. If an argument field begins with a left brace, then the argument ends at the matching right brace. Tcl will strip off the outermost layer of braces and pass the information between the braces to the command without any further modification. For example, in the command

```
set a {xyz a {b c d}}
```

the **set** command will receive two arguments: **a** and **xyz a {b c d}**.

When braces or quotes are in effect, the matching brace or quote need not be on the same line as the starting quote or brace; in this case the newline will be included in the argument field along with any other characters up to the matching brace or quote. For example, the **eval** command takes one argument, which is a command string; **eval** invokes the Tcl interpreter to execute the command string. The command

```
eval {
	set a 22
	set b 33
}
```

will assign the value **22** to **a** and **33** to **b**.

If the first character of a command field is not a left brace, then neither left nor right braces in the field will be treated specially (except as part of variable substitution; see below).

COMMAND SUBSTITUTION WITH BRACKETS

If an open bracket occurs in a field of a command, then command substitution occurs (except for fields enclosed in braces). All of the text up to the matching close bracket is treated as a Tcl command and executed immediately. Then the result of that command is substituted for the bracketed text. For example, consider the command

```
set a [set b]
```

When the **set** command has only a single argument, it is the name of a variable and **set** returns the contents of that variable. In this case, if variable **b** has the value **foo**, then the command above is equivalent to the command

```
set a foo
```

Brackets can be used in more complex ways. For example, if the variable **b** has the value **foo** and the variable **c** has the value **gorp**, then the command

```
set a xyz[set b].[set c]
```

is equivalent to the command

```
set a xyzfoo.gorp
```

A bracketed command may contain multiple commands separated by newlines or semi-colons in the usual fashion. In this case the value of the last command is used for substitution. For example, the command

set a x[set b 22
expr $b+2]x

is equivalent to the command

set a x24x

If a field is enclosed in braces then the brackets and the characters between them are not interpreted specially; they are passed through to the argument verbatim.

VARIABLE SUBSTITUTION WITH $

The dollar sign (**$**) may be used as a special shorthand form for substituting variable values. If **$** appears in an argument that isn't enclosed in braces then variable substitution will occur. The characters after the **$**, up to the first character that isn't a number, letter, or underscore, are taken as a variable name and the string value of that variable is substituted for the name. For example, if variable **foo** has the value **test**, then the command

set a $foo.c

is equivalent to the command

set a test.c

There are two special forms for variable substitution. If the next character after the name of the variable is an open parenthesis, then the variable is assumed to be an array name, and all of the characters between the open parenthesis and the next close parenthesis are taken as an index into the array. Command substitutions and variable substitutions are performed on the information between the parentheses before it is used as an index. For example, if the variable **x** is an array with one element named **first** and value **87** and another element named **14** and value **more**, then the command

set a xyz$x(first)zyx

is equivalent to the command

set a xyz87zyx

If the variable **index** has the value **14**, then the command

set a xyz$x($index)zyx

is equivalent to the command

set a xyzmorezyx

For more information on arrays, see VARIABLES AND ARRAYS below.

The second special form for variables occurs when the dollar sign is followed by an open curly brace. In this case the variable name consists of all the characters up to the next curly brace. Array references are not possible in this form: the name between braces is assumed to refer to a scalar variable. For example, if variable **foo** has the value **test**, then the command

set a abc${foo}bar

is equivalent to the command

set a abctestbar

Variable substitution does not occur in arguments that are enclosed in braces: the dollar sign and variable name are passed through to the argument verbatim.

The dollar sign abbreviation is simply a shorthand form. **$a** is completely equivalent to **[set a]**; it is provided as a convenience to reduce typing.

SEPARATING COMMANDS WITH SEMI-COLONS

Normally, each command occupies one line (the command is terminated by a newline character). However, semi-colon (“;”) is treated as a command separator character; multiple commands may be placed on one line by separating them with a semi-colon. Semi-colons are not treated as command separators if they appear within curly braces or double-quotes.

BACKSLASH SUBSTITUTION

Backslashes may be used to insert non-printing characters into command fields and also to insert special characters like braces and brackets into fields without them being interpreted specially as described above. The backslash sequences understood by the Tcl interpreter are listed below. In each case, the backslash sequence is replaced by the given character:

\b Backspace (0x8).

\f Form feed (0xc).

\n Newline (0xa).

\r Carriage-return (0xd).

\t Tab (0x9).

\v Vertical tab (0xb).

\{ Left brace (“{”).

\} Right brace (“}”).

\[Open bracket (“[”).

\] Close bracket (“]”).

\$ Dollar sign (“$”).

\<space> Space (“ ”): doesn’t terminate argument.

\; Semi-colon: doesn’t terminate command.

\" Double-quote.

\<newline> Nothing: this joins two lines together into a single line. This backslash feature is unique in that it will be applied even when the sequence occurs within braces.

**** Backslash (“\”).

ddd The digits *ddd* (one, two, or three of them) give the octal value of the character. Null characters may not be embedded in command fields; if *ddd* is zero then the backslash sequence is ignored (i.e. it maps to an empty string).

For example, in the command

set a \{x\[\ yz\141

the second argument to **set** will be ``**{x[yza**''.

If a backslash is followed by something other than one of the options described above, then the backslash is transmitted to the argument field without any special processing, and the Tcl scanner continues normal processing with the next character. For example, in the command

set *a \\\{foo

The first argument to **set** will be ***a** and the second argument will be **\{foo**.

If an argument is enclosed in braces, then backslash sequences inside the argument are parsed but no substitution occurs (except for backslash-newline): the backslash sequence is passed through to the argument as is, without making any special interpretation of the characters in the backslash sequence. In particular, backslashed braces are not counted in locating the matching right brace that terminates the argument. For example, in the command

set a {\{abc}

the second argument to **set** will be **\{abc**.

This backslash mechanism is not sufficient to generate absolutely any argument structure; it only covers the most common cases. To produce particularly complicated arguments it is probably easiest to use the **format** command along with command substitution.

COMMAND SUMMARY

[1] A command is just a string.

[2] Within a string commands are separated by newlines or semi-colons (unless the newline or semi-colon is within braces or brackets or is backslashed).

[3] A command consists of fields. The first field is the name of the command. The other fields are strings that are passed to that command as arguments.

[4] Fields are normally separated by white space.

[5] Double-quotes allow white space and semi-colons to appear within a single argument. Command substitution, variable substitution, and backslash substitution still occur inside quotes.

[6] Braces defer interpretation of special characters. If a field begins with a left brace, then it consists of everything between the left brace and the matching right brace. The braces themselves are not included in the argument. No further processing is done on the information between the braces except that backslash-newline sequences are eliminated.

[7] If a field doesn't begin with a brace then backslash, variable, and command substitution are done on the field. Only a single level of processing is done: the results of one substitution are not scanned again for further substitutions or any other special treatment. Substitution can occur on any field of a command, including the command name as well as the arguments.

[8] If the first non-blank character of a command is a **#**, everything from the **#** up through the next newline is treated as a comment and ignored.

EXPRESSIONS

The second major interpretation applied to strings in Tcl is as expressions. Several commands, such as **expr**, **for**, and **if**, treat one or more of their arguments as expressions and call the Tcl expression processors (**Tcl_ExprLong**, **Tcl_ExprBoolean**, etc.) to evaluate them. The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

8.2 + 6

evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified, and in that Tcl expressions support non-numeric operands and string comparisons.

A Tcl expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression processor. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is **0**), or in hexadecimal (if the first two characters of the operand are **0x**). If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the ‘‘f’’, ‘‘F’’, ‘‘l’’, and ‘‘L’’ suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

[1] As an numeric value, either integer or floating-point.

[2] As a Tcl variable, using standard **$** notation. The variable’s value will be used as the operand.

[3] As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand

[4] As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.

[5] As a Tcl command enclosed in brackets. The command will be executed and its result will be used as the operand.

Where substitutions occur above (e.g. inside quoted strings), they are performed by the expression processor. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents.

For some examples of simple expressions, suppose the variable **a** has the value 3 and the variable **b** has the value 6. Then the expression on the left side of each of the lines below will evaluate to the value on the right side of the line:

```
3.1 + $a                    6.1
2 + "$a.$b"                 5.6
4*[llength "6 2"]           8
{word one} < "word $a"      0
```

The valid operators are listed below, grouped in decreasing order of precedence:

– ˜ ! Unary minus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.

*** / %** Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers.

+ – Add and subtract. Valid for any numeric operands.

<< >> Left and right shift. Valid for integer operands only.

< > <= >= Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

== != Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

& Bit-wise AND. Valid for integer operands only.

ˆ Bit-wise exclusive OR. Valid for integer operands only.

| Bit-wise OR. Valid for integer operands only.

&& Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for numeric operands only (integers or floating-point).

|| Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for numeric operands only (integers or floating-point).

x?*y*:*z* If-then-else, as in C. If *x* evaluates to non-zero, then the result is the value of *y*. Otherwise the result is the value of *z*. The *x* operand must have a numeric value.

See the C manual for more details on the results produced by each operator. All of the binary operators group left-to-right within the same precedence level. For example, the expression

4*2 < 7

evaluates to 0.

The **&&**, **||**, and **?:** operators have ‘‘lazy evaluation’’, just as in C, which means that operands are not evaluated if they are not needed to determine the outcome. For example, in

$v ? [a] : [b]

only one of **[a]** or **[b]** will actually be evaluated, depending on the value of **$v**.

All internal computations involving integers are done with the C type *long*, and all internal computations involving floating-point are done with the C type *double*. When converting a string to floating-point, exponent overflow is detected and results in a Tcl error. For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, overflow and underflow are generally not detected reliably for intermediate results.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

5 / 4

yields the result 1, while

5 / 4.0
5 / ([string length "abcd"] + 0.0)

both yield the result 1.25.

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C *sprintf* format specifier **%d** for integers and **%g** for floating-point values. For example, the expressions

"0x03" > "2"
"0y" < "0x12"

both evaluate to 1. The first comparison is done using integer comparison, and the second is done using string comparison after the second operand is converted to the string ‘‘18’’.

In general it is safest to enclose an expression in braces when entering it in a command: otherwise, if the expression contains any white space then the Tcl interpreter will split it among several arguments. For example, the command

expr $a + $b

results in three arguments being passed to **expr**: **$a**, **+**, and **$b**. In addition, if the expression isn't in braces then the Tcl interpreter will perform variable and command substitution immediately (it will happen in the command parser rather than in the expression parser). In many cases the expression is being passed to a command that will evaluate the expression later (or even many times if, for example, the expression is to be used to decide when to exit a loop). Usually the desired goal is to re-do the variable or command substitutions each time the expression is evaluated, rather than once and for all at the beginning. For example, the command

for {set i 1} $i<=10 {incr i} {...} *** WRONG ***

is probably intended to iterate over all values of **i** from 1 to 10. After each iteration of the body of the loop, **for** will pass its second argument to the expression evaluator to see whether or not to continue processing. Unfortunately, in this case the value of **i** in the second argument will be substituted once and for all when the **for** command is parsed. If **i** was 0 before the **for** command was invoked then **for**'s second argument will be **0<=10** which will always evaluate to 1, even though **i**'s value eventually becomes greater than 10. In the above case the loop will never terminate. Instead, the expression should be placed in braces:

for {set i 1} {$i<=10} {incr i} {...} *** RIGHT ***

This causes the substitution of **i**'s value to be delayed; it will be re-done each time the expression is evaluated, which is the desired result.

LISTS

The third major way that strings are interpreted in Tcl is as lists. A list is just a string with a list-like structure consisting of fields separated by white space. For example, the string

Al Sue Anne John

is a list with four elements or fields. Lists have the same basic structure as command strings, except that a newline character in a list is treated as a field separator just like space or tab. Conventions for braces and quotes and backslashes are the same for lists as for commands. For example, the string

a b\ c {d e {f g h}}

is a list with three elements: **a**, **b c**, and **d e {f g h}**. Whenever an element is extracted from a list, the same rules about braces and quotes and backslashes are applied as for commands. Thus in the example above when the third element is extracted from the list, the result is

d e {f g h}

(when the field was extracted, all that happened was to strip off the outermost layer of braces). Command substitution and variable substitution are never made on a list (at least, not by the list-processing commands; the list can always be passed to the Tcl interpreter for evaluation).

The Tcl commands **concat**, **foreach**, **lappend**, **lindex**, **linsert**, **list**, **llength**, **lrange**, **lreplace**, **lsearch**, and **lsort** allow you to build lists, extract elements from them, search them, and perform other list-related functions.

REGULAR EXPRESSIONS

Tcl provides two commands that support string matching using **egrep**-style regular expressions: **regexp** and **regsub**. Regular expressions are implemented using Henry Spencer's package, and the description of regular expressions below is copied verbatim from his manual entry.

A regular expression is zero or more *branches*, separated by "|". It matches anything that matches one of the branches.

A branch is zero or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by "*", "+", or "?". An atom followed by "*" matches a sequence of 0 or more matches of the atom. An atom followed by "+" matches a sequence of 1 or more matches of the atom. An atom followed by "?" matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a *range* (see below), "." (matching any single character), "^" (matching the null string at the beginning of the input string), "$" (matching the null string at the end of the input string), a "\" followed by a single character (matching that character), or a single character with no other significance (matching that character).

A *range* is a sequence of characters enclosed in "[]". It normally matches any single character from the sequence. If the sequence begins with "^", it matches any single character *not* from the rest of the sequence. If two characters in the sequence are separated by "−", this is shorthand for the full list of ASCII characters between them (e.g. "[0-9]" matches any decimal digit). To include a literal "]" in the sequence, make it the first character (following a possible "^"). To include a literal "−", make it the first or last character.

If a regular expression could match two different parts of a string, it will match the one which begins earliest. If both begin in the same place but match different lengths, or match the same length in different ways, life gets messier, as follows.

In general, the possibilities in a list of branches are considered in left-to-right order, the possibilities for "*", "+", and "?" are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered leftmost-first. The match that will be chosen is the one that uses the earliest possibility in the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice. And so forth.

For example, "(ab|a)b*c" could match "abc" in one of two ways. The first choice is between "ab" and "a"; since "ab" is earlier, and does lead to a successful overall match, it is chosen. Since the "b" is already spoken for, the "b*" must match its last possibility—the empty string—since it must respect the earlier choice.

In the particular case where no "|"s are present and there is only one "*", "+", or "?", the net effect is that the longest possible match will be chosen. So "ab*", presented with "xabbbby", will match "abbbb". Note that if "ab*" is tried against "xabyabbbz", it will match "ab" just after "x", due to the begins-earliest rule. (In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.)

COMMAND RESULTS

Each command produces two results: a code and a string. The code indicates whether the command completed successfully or not, and the string gives additional information. The valid codes are defined in tcl.h, and are:

TCL_OK This is the normal return code, and indicates that the command completed successfully. The string gives the command's return value.

TCL_ERROR Indicates that an error occurred; the string gives a message describing the error. In addition, the global variable **errorInfo** will contain human-readable information describing which commands and procedures were being executed when the error occurred, and the global variable **errorCode** will contain machine-readable details about the error, if they are available. See the section BUILT-IN VARIABLES below for more information.

TCL_RETURN Indicates that the **return** command has been invoked, and that the current procedure (or top-level command or **source** command) should return immediately. The string gives the return value for the procedure or command.

TCL_BREAK Indicates that the **break** command has been invoked, so the innermost loop should abort immediately. The string should always be empty.

TCL_CONTINUE Indicates that the **continue** command has been invoked, so the innermost loop should go on to the next iteration. The string should always be empty.

Tcl programmers do not normally need to think about return codes, since TCL_OK is almost always returned. If anything else is returned by a command, then the Tcl interpreter immediately stops processing commands and returns to its caller. If there are several nested invocations of the Tcl interpreter in progress, then each nested command will usually return the error to its caller, until eventually the error is reported to the top-level application code. The application will then display the error message for the user.

In a few cases, some commands will handle certain ‘‘error’’ conditions themselves and not return them upwards. For example, the **for** command checks for the TCL_BREAK code; if it occurs, then **for** stops executing the body of the loop and returns TCL_OK to its caller. The **for** command also handles TCL_CONTINUE codes and the procedure interpreter handles TCL_RETURN codes. The **catch** command allows Tcl programs to catch errors and handle them without aborting command interpretation any further.

PROCEDURES

Tcl allows you to extend the command interface by defining procedures. A Tcl procedure can be invoked just like any other Tcl command (it has a name and it receives one or more arguments). The only difference is that its body isn't a piece of C code linked into the program; it is a string containing one or more other Tcl commands. See the **proc** command for information on how to define procedures and what happens when they are invoked.

VARIABLES – SCALARS AND ARRAYS

Tcl allows the definition of variables and the use of their values either through **$**-style variable substitution, the **set** command, or a few other mechanisms. Variables need not be declared: a new variable will automatically be created each time a new variable name is used.

Tcl supports two types of variables: scalars and arrays. A scalar variable has a single value, whereas an array variable can have any number of elements, each with a name (called its ‘‘index’’) and a value. Array indexes may be arbitrary strings; they need not be numeric. Parentheses are used refer to array elements in Tcl commands. For example, the command

set x(first) 44

will modify the element of **x** whose index is **first** so that its new value is **44**. Two-dimensional arrays can be simulated in Tcl by using indexes that contain multiple concatenated values. For example, the commands

set a(2,3) 1
set a(3,6) 2

set the elements of **a** whose indexes are **2,3** and **3,6**.

In general, array elements may be used anywhere in Tcl that scalar variables may be used. If an array is defined with a particular name, then there may not be a scalar variable with the same name. Similarly, if there is a scalar variable with a particular name then it is not possible to make array references to the variable. To convert a scalar variable to an array or vice versa, remove the existing variable with the **unset** command.

The **array** command provides several features for dealing with arrays, such as querying the names of all the elements of the array and searching through the array one element at a time.

Variables may be either global or local. If a variable name is used when a procedure isn't being executed, then it automatically refers to a global variable. Variable names used within a procedure normally refer to local variables associated with that invocation of the procedure. Local variables are deleted whenever a procedure exits. The **global** command may be used to request that a name refer to a global variable for the duration of the current procedure (this is somewhat analogous to **extern** in C).

BUILT-IN COMMANDS

The Tcl library provides the following built-in commands, which will be available in any application using Tcl. In addition to these built-in commands, there may be additional commands defined by each application, plus commands defined as Tcl procedures. In the command syntax descriptions below, words in boldface are literals that you type verbatim to Tcl. Words in italics are meta-symbols; they serve as names for any of a range of values that you can type. Optional arguments or groups of arguments are indicated by enclosing them in question-marks. Ellipses ("...") indicate that any number of additional arguments or groups of arguments may appear, in the same format as the preceding argument(s).

append *varName value* ?*value value ...*?

Append all of the *value* arguments to the current value of variable *varName*. If *varName* doesn't exist, it is given a value equal to the concatenation of all the *value* arguments. This command provides an efficient way to build up long variables incrementally. For example, "**append a $b**" is much more efficient than "**set a ab**" if **$a** is long.

array *option arrayName* ?*arg arg ...*?

This command performs one of several operations on the variable given by *arrayName*. *ArrayName* must be the name of an existing array variable. The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

array anymore *arrayName searchId*

Returns 1 if there are any more elements left to be processed in an array search, 0 if all elements have already been returned. *SearchId* indicates which search on *arrayName* to check, and must have been the return value from a previous invocation of **array startsearch**. This option is particularly useful if an array has an element with an empty name, since the return value from **array nextelement** won't indicate whether the search has been completed.

array donesearch *arrayName searchId*

This command terminates an array search and destroys all the state associated with that search. *SearchId* indicates which search on *arrayName* to destroy, and must have been the return value from a previous invocation of **array startsearch**. Returns an empty string.

array names *arrayName*

Returns a list containing the names of all of the elements in the array. If there are no elements in the array then an empty string is returned.

array nextelement *arrayName searchId*

Returns the name of the next element in *arrayName*, or an empty string if all elements of *arrayName* have already been returned in this search. The *searchId* argument identifies the search, and must have been the return value of an **array startsearch** command. Warning: if elements are added to or deleted from the array, then all searches are automatically terminated just as if **array donesearch** had been invoked; this will cause **array nextelement** operations to fail for those searches.

array size *arrayName*

Returns a decimal string giving the number of elements in the array.

array startsearch *arrayName*

This command initializes an element-by-element search through the array given by *arrayName*, such that invocations of the **array nextelement** command will return the names of the individual elements in the array. When the search has been completed, the **array donesearch** command should be invoked. The return value is a search identifier that must be used in **array nextelement** and **array donesearch** commands; it allows multiple searches to be underway simultaneously for the same array.

break This command may be invoked only inside the body of a loop command such as **for** or **foreach** or **while**. It returns a TCL_BREAK code to signal the innermost containing loop command to return immediately.

case *string* ?**in**? *patList body* ?*patList body* ...?

case *string* ?**in**? {*patList body* ?*patList body* ...?}

Match *string* against each of the *patList* arguments in order. If one matches, then evaluate the following *body* argument by passing it recursively to the Tcl interpreter, and return the result of that evaluation. Each *patList* argument consists of a single pattern or list of patterns. Each pattern may contain any of the wild-cards described under **string match**. If a *patList* argument is **default**, the corresponding body will be evaluated if no *patList* matches *string*. If no *patList* argument matches *string* and no default is given, then the **case** command returns an empty string.

Two syntaxes are provided. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line case commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *patList* arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

Below are some examples of **case** commands:

```
case abc in {a b} {format 1} default {format 2} a* {format 3}
```

will return 3,

```
case a in {
    {a b} {format 1}
    default {format 2}
    a* {format 3}
}
```

will return 1, and

```
case xyz {
    {a b}
        {format 1}
    default
        {format 2}
    a*
        {format 3}
}
```

will return 2.

catch *command* ?*varName*?

The **catch** command may be used to prevent errors from aborting command interpretation. **Catch** calls the Tcl interpreter recursively to execute *command*, and always returns a TCL_OK code, regardless of any errors that might occur while executing *command*. The return value from **catch** is a decimal string giving the code returned by the Tcl interpreter after executing *command*. This will be **0** (TCL_OK) if there were no errors in *command*; otherwise it will have a non-zero value corresponding to one of the exceptional return codes (see tcl.h for the definitions of code values). If the *varName* argument is given, then it gives the name of a variable; **catch** will set the value of the variable to the string returned from *command* (either a result or an error message).

cd ?*dirName*?

Change the current working directory to *dirName*, or to the home directory (as specified in the HOME environment variable) if *dirName* is not given. If *dirName* starts with a tilde, then tilde-expansion is done as described for **Tcl_TildeSubst**. Returns an empty string. This command can potentially be disruptive to an application, so it may be removed in some applications.

close *fileId*

Closes the file given by *fileId*. *FileId* must be the return value from a previous invocation of the **open** command; after this command, it should not be used anymore. If *fileId* refers to a command pipeline instead of a file, then **close** waits for the children to complete. The normal result of this command is an empty string, but errors are returned if there are problems in closing the file or waiting for children to complete.

concat *arg* ?*arg ...*?

This command treats each argument as a list and concatenates them into a single list. It permits any number of arguments. For example, the command

concat a b {c d e} {f {g h}}

will return

a b c d e f {g h}

as its result.

continue

This command may be invoked only inside the body of a loop command such as **for** or **foreach** or **while**. It returns a TCL_CONTINUE code to signal the innermost containing loop command to skip the remainder of the loop's body but continue with the next iteration of the loop.

eof *fileId*

Returns 1 if an end-of-file condition has occurred on *fileId*, 0 otherwise. *FileId* must have been the return value from a previous call to **open**, or it may be **stdin**, **stdout**, or **stderr** to refer to one of the standard I/O channels.

error *message* ?*info*? ?*code*?

Returns a TCL_ERROR code, which causes command interpretation to be unwound. *Message* is a string that is returned to the application to indicate what went wrong.

If the *info* argument is provided and is non-empty, it is used to initialize the global variable **errorInfo**. **errorInfo** is used to accumulate a stack trace of what was in progress when an error occurred; as nested commands unwind, the Tcl interpreter adds information to **errorInfo**. If the *info* argument is present, it is used to initialize **errorInfo** and the first increment of unwind information will not be added by the Tcl interpreter. In other words, the command containing the **error** command will not appear in **errorInfo**; in its place will be *info*. This feature is most useful in conjunction with the **catch** command: if a caught error cannot be handled successfully, *info* can be used to return a stack trace reflecting the original point of occurrence of the error:

catch {...} errMsg
set savedInfo $errorInfo
...
error $errMsg $savedInfo

If the *code* argument is present, then its value is stored in the **errorCode** global variable. This variable is intended to hold a machine-readable description of the error in cases where such information is available; see the section BUILT-IN VARIABLES below for information on the proper format for the variable. If the *code* argument is not present, then **errorCode** is automatically reset to ''NONE'' by the Tcl interpreter as part of processing the error generated by the command.

eval *arg* ?*arg* ...?

Eval takes one or more arguments, which together comprise a Tcl command (or collection of Tcl commands separated by newlines in the usual way). **Eval** concatenates all its arguments in the same fashion as the **concat** command, passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it).

exec *arg* ?*arg* ...?

This command treats its arguments as the specification of one or more UNIX commands to execute as subprocesses. The commands take the form of a standard shell pipeline; ''|'' arguments separate commands in the pipeline and cause standard output of the preceding command to be piped into standard input of the next command.

Under normal conditions the result of the **exec** command consists of the standard output produced by the last command in the pipeline. If any of the commands in the pipeline exit abnormally or are killed or suspended, then **exec** will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the **errorCode** variable will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file, then **exec** will return an error, and the error message will include the pipeline's output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is deleted from the result or error message for consistency with normal Tcl return values.

If an *arg* has the value ''>'' then the following argument is taken as the name of a file and the standard output of the last command in the pipeline is redirected to the file. In this situation **exec** will normally return an empty string.

If an *arg* has the value ''<'' then the following argument is taken as the name of a file to use for standard input to the first command in the pipeline. If an argument has the value ''<<'' then the following argument is taken as an immediate value to be passed to the first command as standard input. If there is no ''<'' or ''<<'' argument then the standard input for the first command in the pipeline is taken from the application's current standard input.

If the last *arg* is ''&'' then the command will be executed in background. In this case the standard output from the last command in the pipeline will go to the application's standard output unless redirected in the command, and error output from all the commands in the pipeline will go to the application's standard error file.

Each *arg* becomes one word for a command, except for ''|'', ''<'', ''<<'', ''>'', and ''&'' arguments, and the arguments that follow ''<'', ''<<'', and ''>''. The first word in each command is taken as the command name; tilde-substitution is performed on it, and the directories in the PATH environment variable are searched for an executable by the given name. No ''glob'' expansion or other shell-like substitutions are performed on the arguments to commands.

exit ?returnCode?
Terminate the process, returning *returnCode* to the parent as the exit status. If *returnCode* isn't specified then it defaults to 0.

expr *arg*
Calls the expression processor to evaluate *arg*, and returns the result as a string. See the section EXPRESSIONS above.

file *option name* ?*arg arg ...*?
Operate on a file or a file name. *Name* is the name of a file; if it starts with a tilde, then tilde substitution is done before executing the command (see the manual entry for **Tcl_TildeSubst** for details). *Option* indicates what to do with the file name. Any unique abbreviation for *option* is acceptable. The valid options are:

file atime *name*
Return a decimal string giving the time at which file *name* was last accessed. The time is measured in the standard UNIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its access time cannot be queried then an error is generated.

file dirname *name*
Return all of the characters in *name* up to but not including the last slash character. If there are no slashes in *name* then return ".". If the last slash in *name* is its first character, then return "/".

file executable *name*
Return **1** if file *name* is executable by the current user, **0** otherwise.

file exists *name*
Return **1** if file *name* exists and the current user has search privileges for the directories leading to it, **0** otherwise.

file extension *name*
Return all of the characters in *name* after and including the last dot in *name*. If there is no dot in *name* then return the empty string.

file isdirectory *name*
Return **1** if file *name* is a directory, **0** otherwise.

file isfile *name*
Return **1** if file *name* is a regular file, **0** otherwise.

file lstat *name varName*
Same as **stat** option (see below) except uses the *lstat* kernel call instead of *stat*. This means that if *name* refers to a symbolic link the information returned in *varName* is for the link rather than the file it refers to. On systems that don't support symbolic links this option behaves exactly the same as the **stat** option.

file mtime *name*
Return a decimal string giving the time at which file *name* was last modified. The time is measured in the standard UNIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its modified time cannot be queried then an error is generated.

file owned *name*
Return **1** if file *name* is owned by the current user, **0** otherwise.

file readable *name*
Return **1** if file *name* is readable by the current user, **0** otherwise.

file readlink *name*

Returns the value of the symbolic link given by *name* (i.e. the name of the file it points to). If *name* isn't a symbolic link or its value cannot be read, then an error is returned. On systems that don't support symbolic links this option is undefined.

file rootname *name*

Return all of the characters in *name* up to but not including the last "." character in the name. If *name* doesn't contain a dot, then return *name*.

file size *name*

Return a decimal string giving the size of file *name* in bytes. If the file doesn't exist or its size cannot be queried then an error is generated.

file stat *name varName*

Invoke the **stat** kernel call on *name*, and use the variable given by *varName* to hold information returned from the kernel call. *VarName* is treated as an array variable, and the following elements of that variable are set: **atime**, **ctime**, **dev**, **gid**, **ino**, **mode**, **mtime**, **nlink**, **size**, **type**, **uid**. Each element except **type** is a decimal string with the value of the corresponding field from the **stat** return structure; see the manual entry for **stat** for details on the meanings of the values. The **type** element gives the type of the file in the same form returned by the command **file type**. This command returns an empty string.

file tail *name*

Return all of the characters in *name* after the last slash. If *name* contains no slashes then return *name*.

file type *name*

Returns a string giving the type of file *name*, which will be one of **file**, **directory**, **characterSpecial**, **blockSpecial**, **fifo**, **link**, or **socket**.

file writable *name*

Return **1** if file *name* is writable by the current user, **0** otherwise.

The **file** commands that return 0/1 results are often used in conditional or looping commands, for example:

```
if {![file exists foo]} then {error {bad file name}} else {...}
```

flush *fileId*

Flushes any output that has been buffered for *fileId*. *FileId* must have been the return value from a previous call to **open**, or it may be **stdout** or **stderr** to access one of the standard I/O streams; it must refer to a file that was opened for writing. This command returns an empty string.

for *start test next body*

For is a looping command, similar in structure to the C **for** statement. The *start*, *next*, and *body* arguments must be Tcl command strings, and *test* is an expression string. The **for** command first invokes the Tcl interpreter to execute *start*. Then it repeatedly evaluates *test* as an expression; if the result is non-zero it invokes the Tcl interpreter on *body*, then invokes the Tcl interpreter on *next*, then repeats the loop. The command terminates when *test* evaluates to 0. If a **continue** command is invoked within *body* then any remaining commands in the current execution of *body* are skipped; processing continues by invoking the Tcl interpreter on *next*, then evaluating *test*, and so on. If a **break** command is invoked within *body* or *next*, then the **for** command will return immediately. The operation of **break** and **continue** are similar to the corresponding statements in C. **For** returns an empty string.

foreach *varname list body*

In this command, *varname* is the name of a variable, *list* is a list of values to assign to *varname*, and *body* is a collection of Tcl commands. For each field in *list* (in order from left to right),

foreach assigns the contents of the field to *varname* (as if the **lindex** command had been used to extract the field), then calls the Tcl interpreter to execute *body*. The **break** and **continue** statements may be invoked inside *body*, with the same effect as in the **for** command. **Foreach** returns an empty string.

format *formatString* ?*arg arg ...*?

This command generates a formatted string in the same way as the C **sprintf** procedure (it uses **sprintf** in its implementation). *FormatString* indicates how to format the result, using **%** fields as in **sprintf**, and the additional arguments, if any, provide values to be substituted into the result. All of the **sprintf** options are valid; see the **sprintf** man page for details. Each *arg* must match the expected type from the **%** field in *formatString*; the **format** command converts each argument to the correct type (floating, integer, etc.) before passing it to **sprintf** for formatting. The only unusual conversion is for **%c**; in this case the argument must be a decimal string, which will then be converted to the corresponding ASCII character value. **Format** does backslash substitution on its *formatString* argument, so backslash sequences in *formatString* will be handled correctly even if the argument is in braces. The return value from **format** is the formatted string.

gets *fileId* ?*varName*?

Reads the next line from the file given by *fileId* and discards the terminating newline character. If *varName* is specified, then the line is placed in the variable by that name and the return value is a count of the number of characters read (not including the newline). If the end of the file is reached before reading any characters then −1 is returned and *varName* is set to an empty string. If *varName* is not specified then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so **eof** may have to be used to determine what really happened. If the last character in the file is not a newline character, then **gets** behaves as if there were an additional newline character at the end of the file. *FileId* must be **stdin** or the return value from a previous call to **open**; it must refer to a file that was opened for reading.

glob ?**−nocomplain**? *filename* ?*filename ...*?

This command performs filename globbing, using csh rules. The returned value from **glob** is the list of expanded filenames. If **−nocomplain** is specified as the first argument then an empty list may be returned; otherwise an error is returned if the expanded list is empty. The **−nocomplain** argument must be provided exactly: an abbreviation will not be accepted.

global *varname* ?*varname ...*?

This command is ignored unless a Tcl procedure is being interpreted. If so, then it declares the given *varname*'s to be global variables rather than local ones. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varname*s will be bound to a global variable instead of a local one.

history ?*option*? ?*arg arg ...*?

Note: this command may not be available in all Tcl-based applications. Typically, only those that receive command input in a typescript form will support history. The **history** command performs one of several operations related to recently-executed commands recorded in a history list. Each of these recorded commands is referred to as an ''event''. When specifying an event to the **history** command, the following forms may be used:

[1] A number: if positive, it refers to the event with that number (all events are numbered starting at 1). If the number is negative, it selects an event relative to the current event (**−1** refers to the previous event, **−2** to the one before that, and so on).

[2] A string: selects the most recent event that matches the string. An event is considered to match the string either if the string is the same as the first characters of the event, or if the string matches the event in the sense of the **string match** command.

The **history** command can take any of the following forms:

history Same as **history info**, described below.

history add *command* ?**exec**?
Add the *command* argument to the history list as a new event. If **exec** is specified (or abbreviated) then the command is also executed and its result is returned. If **exec** isn't specified then an empty string is returned as result.

history change *newValue* ?*event*?
Replace the value recorded for an event with *newValue*. *Event* specifies the event to replace, and defaults to the *current* event (not event **−1**). This command is intended for use in commands that implement new forms of history substitution and wish to replace the current event (which invokes the substitution) with the command created through substitution. The return value is an empty string.

history event ?*event*?
Returns the value of the event given by *event*. *Event* defaults to **−1**. This command causes history revision to occur: see below for details.

history info ?*count*?
Returns a formatted string (intended for humans to read) giving the event number and contents for each of the events in the history list except the current event. If *count* is specified then only the most recent *count* events are returned.

history keep *count*
This command may be used to change the size of the history list to *count* events. Initially, 20 events are retained in the history list. This command returns an empty string.

history nextid
Returns the number of the next event to be recorded in the history list. It is useful for things like printing the event number in command-line prompts.

history redo ?*event*?
Re-execute the command indicated by *event* and return its result. *Event* defaults to **−1**. This command results in history revision: see below for details.

history substitute *old new* ?*event*?
Retrieve the command given by *event* (**−1** by default), replace any occurrences of *old* by *new* in the command (only simple character equality is supported; no wild cards), execute the resulting command, and return the result of that execution. This command results in history revision: see below for details.

history words *selector* ?*event*?
Retrieve from the command given by *event* (**−1** by default) the words given by *selector*, and return those words in a string separated by spaces. The **selector** argument has three forms. If it is a single number then it selects the word given by that number (**0** for the command name, **1** for its first argument, and so on). If it consists of two numbers separated by a dash, then it selects all the arguments between those two. Otherwise **selector** is treated as a pattern; all words matching that pattern (in the sense of **string match**) are returned. In the numeric forms **$** may be used to select the last word of a command. For example, suppose the most recent command in the history list is

format {%s is %d years old} Alice [expr $ageInMonths/12]

Below are some history commands and the results they would produce:

<u>Command</u> <u>Result</u>

history words $	**[expr $ageInMonths/12]**
history words 1-2	**{%s is %d years old} Alice**
history words *a*o*	**{%s is %d years old} [expr $ageInMonths/12]**

History words results in history revision: see below for details.

The history options **event**, **redo**, **substitute**, and **words** result in ''history revision''. When one of these options is invoked then the current event is modified to eliminate the history command and replace it with the result of the history command. For example, suppose that the most recent command in the history list is

set a [expr $b+2]

and suppose that the next command invoked is one of the ones on the left side of the table below. The command actually recorded in the history event will be the corresponding one on the right side of the table.

Command Typed	Command Recorded
history	**set a [expr $b+2]**
history s a b	**set b [expr $b+2]**
set c [history w 2]	**set c [expr $b+2]**

History revision is needed because event specifiers like **−1** are only valid at a particular time: once more events have been added to the history list a different event specifier would be needed. History revision occurs even when **history** is invoked indirectly from the current event (e.g. a user types a command that invokes a Tcl procedure that invokes **history**): the top-level command whose execution eventually resulted in a **history** command is replaced. If you wish to invoke commands like **history words** without history revision, you can use **history event** to save the current history event and then use **history change** to restore it later.

if *test* ?**then**? *trueBody* ?**else**? ?*falseBody*?

The *if* command evaluates *test* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must be numeric; if it is non-zero then *trueBody* is called by passing it to the Tcl interpreter. Otherwise *falseBody* is executed by passing it to the Tcl interpreter. The **then** and **else** arguments are optional ''noise words'' to make the command easier to read. *FalseBody* is also optional; if it isn't specified then the command does nothing if *test* evaluates to zero. The return value from **if** is the value of the last command executed in *trueBody* or *falseBody*, or the empty string if *test* evaluates to zero and *falseBody* isn't specified.

incr *varName* ?*increment*?

Increment the value stored in the variable whose name is *varName*. The value of the variable must be integral. If *increment* is supplied then its value (which must be an integer) is added to the value of variable *varName*; otherwise 1 is added to *varName*. The new value is stored as a decimal string in variable *varName* and also returned as result.

info *option* ?*arg arg ...*?

Provide information about various internals to the Tcl interpreter. The legal *option*'s (which may be abbreviated) are:

info args *procname*

Returns a list containing the names of the arguments to procedure *procname*, in order. *Procname* must be the name of a Tcl command procedure.

info body *procname*

Returns the body of procedure *procname*. *Procname* must be the name of a Tcl command procedure.

info cmdcount
Returns a count of the total number of commands that have been invoked in this interpreter.

info commands ?*pattern*?
If *pattern* isn't specified, returns a list of names of all the Tcl commands, including both the built-in commands written in C and the command procedures defined using the **proc** command. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info default *procname arg varname*
Procname must be the name of a Tcl command procedure and *arg* must be the name of an argument to that procedure. If *arg* doesn't have a default value then the command returns **0**. Otherwise it returns **1** and places the default value of *arg* into variable *varname*.

info exists *varName*
Returns **1** if the variable named *varName* exists in the current context (either as a global or local variable), returns **0** otherwise.

info globals ?*pattern*?
If *pattern* isn't specified, returns a list of all the names of currently-defined global variables. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info level ?*number*?
If *number* is not specified, this command returns a number giving the stack level of the invoking procedure, or 0 if the command is invoked at top-level. If *number* is specified, then the result is a list consisting of the name and arguments for the procedure call at level *number* on the stack. If *number* is positive then it selects a particular stack level (1 refers to the top-most active procedure, 2 to the procedure it called, and so on); otherwise it gives a level relative to the current level (0 refers to the current procedure, -1 to its caller, and so on). See the **uplevel** command for more information on what stack levels mean.

info library
Returns the name of the library directory in which standard Tcl scripts are stored. If there is no such directory defined for the current installation then an error is generated. See the **library** manual entry for details of the facilities provided by the Tcl script library. Normally each application will have its own application-specific script library in addition to the Tcl script library; I suggest that each application set a global variable with a name like **$***app***Library** (where *app* is the application's name) to hold the location of that application's library directory.

info locals ?*pattern*?
If *pattern* isn't specified, returns a list of all the names of currently-defined local variables, including arguments to the current procedure, if any. Variables defined with the **global** and **upvar** commands will not be returned. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info procs ?*pattern*?
If *pattern* isn't specified, returns a list of all the names of Tcl command procedures. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

info script
If a Tcl script file is currently being evaluated (i.e. there is a call to **Tcl_EvalFile** active or there is an active invocation of the **source** command), then this command returns the

name of the innermost file being processed. Otherwise the command returns an empty string.

info tclversion

Returns the version number for this version of Tcl in the form *x.y*, where changes to *x* represent major changes with probable incompatibilities and changes to *y* represent small enhancements and bug fixes that retain backward compatibility.

info vars ?*pattern*?

If *pattern* isn't specified, returns a list of all the names of currently-visible variables, including both locals and currently-visible globals. If *pattern* is specified, only those names matching *pattern* are returned. Matching is determined using the same rules as for **string match**.

join *list* ?*joinString*?

The *list* argument must be a valid Tcl list. This command returns the string formed by joining all of the elements of *list* together with *joinString* separating each adjacent pair of elements. The *joinString* argument defaults to a space character.

lappend *varName value* ?*value value ...*?

Treat the variable given by *varName* as a list and append each of the *value* arguments to that list as a separate element, with spaces between elements. If *varName* doesn't exist, it is created as a list with elements given by the *value* arguments. **Lappend** is similar to **append** except that the *value*s are appended as list elements rather than raw text. This command provides a relatively efficient way to build up large lists. For example, ‘‘**lappend a $b**’’ is much more efficient than ‘‘**set a [concat $a [list $b]]**’’ when **$a** is long.

lindex *list index*

Treats *list* as a Tcl list and returns the *index*'th element from it (0 refers to the first element of the list). In extracting the element, *lindex* observes the same rules concerning braces and quotes and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If *index* is negative or greater than or equal to the number of elements in *value*, then an empty string is returned.

linsert *list index element* ?*element element ...*?

This command produces a new list from *list* by inserting all of the *element* arguments just before the *index*th element of *list*. Each *element* argument will become a separate element of the new list. If *index* is less than or equal to zero, then the new elements are inserted at the beginning of the list. If *index* is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

list *arg* ?*arg ...*?

This command returns a list comprised of all the *arg*s. Braces and backslashes get added as necessary, so that the **index** command may be used on the result to re-extract the original arguments, and also so that **eval** may be used to execute the resulting list, with *arg1* comprising the command's name and the other *arg*s comprising its arguments. **List** produces slightly different results than **concat**: **concat** removes one level of grouping before forming the list, while **list** works directly from the original arguments. For example, the command

list a b {c d e} {f {g h}}

will return

a b {c d e} {f {g h}}

while concat with the same arguments will return

a b c d e f {g h}

llength *list*

Treats *list* as a list and returns a decimal string giving the number of elements in it.

lrange *list first last*

List must be a valid Tcl list. This command will return a new list consisting of elements *first* through *last*, inclusive. *Last* may be **end** (or any abbreviation of it) to refer to the last element of the list. If *first* is less than zero, it is treated as if it were zero. If *last* is greater than or equal to the number of elements in the list, then it is treated as if it were **end**. If *first* is greater than *last* then an empty string is returned. Note: ‘‘**lrange** *list first first*’’ does not always produce the same result as ‘‘**lindex** *list first*’’ (although it often does for simple fields that aren’t enclosed in braces); it does, however, produce exactly the same results as ‘‘**list [lindex** *list first*]’’

lreplace *list first last* ?*element element ...*?

Returns a new list formed by replacing one or more elements of *list* with the *element* arguments. *First* gives the index in *list* of the first element to be replaced. If *first* is less than zero then it refers to the first element of *list*; the element indicated by *first* must exist in the list. *Last* gives the index in *list* of the last element to be replaced; it must be greater than or equal to *first*. *Last* may be **end** (or any abbreviation of it) to indicate that all elements between *first* and the end of the list should be replaced. The *element* arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each *element* argument will become a separate element of the list. If no *element* arguments are specified, then the elements between *first* and *last* are simply deleted.

lsearch *list pattern*

Search the elements of *list* to see if one of them matches *pattern*. If so, the command returns the index of the first matching element. If not, the command returns **−1**. Pattern matching is done in the same way as for the **string match** command.

lsort *list*

Sort the elements of *list*, returning a new list in sorted order. ASCII sorting is used, with the result in increasing order.

open *fileName* ?*access*?

Opens a file and returns an identifier that may be used in future invocations of commands like **read**, **puts**, and **close**. *FileName* gives the name of the file to open; if it starts with a tilde then tilde substitution is performed as described for **Tcl_TildeSubst**. If the first character of *fileName* is ‘‘|’’ then the remaining characters of *fileName* are treated as a command pipeline to invoke, in the same style as for **exec**. In this case, the identifier returned by **open** may be used to write to the command’s input pipe or read from its output pipe. The *access* argument indicates the way in which the file (or command pipeline) is to be accessed. It may have any of the following values:

- **r** Open the file for reading only; the file must already exist.
- **r+** Open the file for both reading and writing; the file must already exist.
- **w** Open the file for writing only. Truncate it if it exists. If it doesn’t exist, create a new file.
- **w+** Open the file for reading and writing. Truncate it if it exists. If it doesn’t exist, create a new file.
- **a** Open the file for writing only. The file must already exist, and the file is positioned so that new data is appended to the file.
- **a+** Open the file for reading and writing. The file must already exist, and the initial access position is set to the end of the file.

Access defaults to **r**. If a file is opened for both reading and writing, then **seek** must be invoked between a read and a write, or vice versa (this restriction does not apply to command pipelines opened with **open**). When *fileName* specifies a command pipeline and a write-only access is used, then standard output from the pipeline is directed to the current standard output unless overridden by the command. When *fileName* specifies a command pipeline and a read-only access is used, then standard input from the pipeline is taken from the current standard input unless overridden by the command.

proc *name args body*

The **proc** command creates a new Tcl command procedure, *name*, replacing any existing command there may have been by that name. Whenever the new command is invoked, the contents of *body* will be executed by the Tcl interpreter. *Args* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier, then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value. braces and backslashes may be used in the usual way to specify complex default values.

When *name* is invoked, a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name **args**, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to **args** are combined into a list (as if the **list** command had been used); this combined value is assigned to the local variable **args**.

When *body* is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the **global** command.

The **proc** command returns the null string. When a procedure is invoked, the procedure's return value is the value specified in a **return** command. If the procedure doesn't execute an explicit **return**, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure-as-a-whole will return that same error.

puts *fileId string* ?**nonewline**?

Writes the characters given by *string* to the file given by *fileId*. **Puts** normally outputs a newline character after *string*, but this feature may be suppressed by specifying the **nonewline** argument. Output to files is buffered internally by Tcl; the **flush** command may be used to force buffered characters to be output. *FileId* must have been the return value from a previous call to **open**, or it may be **stdout** or **stderr** to refer to one of the standard I/O channels; it must refer to a file that was opened for writing.

pwd

Returns the path name of the current working directory.

read *fileId*

read *fileId* **nonewline**

read *fileId numBytes*

In the first form, all of the remaining bytes are read from the file given by *fileId*; they are returned as the result of the command. If **nonewline** is specified as an additional argument, then the last character of the file is discarded if it is a newline. In the third form, the extra argument specifies how many bytes to read; exactly this many bytes will be read and returned, unless there are fewer than *numBytes* bytes left in the file; in this case, all the remaining bytes are returned. *FileId* must be **stdin** or the return value from a previous call to **open**; it must refer to a file that was opened for reading.

regexp ?**−indices**? ?**−nocase**? *exp string* ?*matchVar*? ?*subMatchVar subMatchVar ...*?

Determines whether the regular expression *exp* matches part or all of *string* and returns 1 if it does, 0 if it doesn't. See REGULAR EXPRESSIONS above for complete information on the syntax of *exp* and how it is matched against *string*.

If the **−nocase** switch is specified then upper-case characters in *string* are treated as lower case during the matching process. The **−nocase** switch must be specified before *exp* and may not be abbreviated.

If additional arguments are specified after *string* then they are treated as the names of variables to use to return information about which part(s) of *string* matched *exp*. *MatchVar* will be set to the range of *string* that matched all of *exp*. The first *subMatchVar* will contain the characters in *string* that matched the leftmost parenthesized subexpression within *exp*, the next *subMatchVar* will contain the characters that matched the next parenthesized subexpression to the right in *exp*, and so on.

Normally, *matchVar* and the *subMatchVar*s are set to hold the matching characters from **string**. However, if the **−indices** switch is specified then each variable will contain a list of two decimal strings giving the indices in *string* of the first and last characters in the matching range of characters. The **−indices** switch must be specified before the *exp* argument and may not be abbreviated.

If there are more *subMatchVar*'s than parenthesized subexpressions within *exp*, or if a particular subexpression in *exp* doesn't match the string (e.g. because it was in a portion of the expression that wasn't matched), then the corresponding *subMatchVar* will be set to "**−1 −1**" if **−indices** has been specified or to an empty string otherwise.

regsub ?**−all**? ?**−nocase**? *exp string subSpec varName*

This command matches the regular expression *exp* against *string* using the rules described in REGULAR EXPRESSIONS above. If there is no match, then the command returns 0 and does nothing else. If there is a match, then the command returns 1 and also copies *string* to the variable whose name is given by *varName*. When copying *string*, the portion of *string* that matched *exp* is replaced with *subSpec*. If *subSpec* contains a "&" or "\0", then it is replaced in the substitution with the portion of *string* that matched *exp*. If *subSpec* contains a "*n*", where *n* is a digit between 1 and 9, then it is replaced in the substitution with the portion of *string* that matched the *n*-th parenthesized subexpression of *exp*. Additional backslashes may be used in *subSpec* to prevent special interpretation of "&" or "\0" or "*n*" or backslash. The use of backslashes in *subSpec* tends to interact badly with the Tcl parser's use of backslashes, so it's generally safest to enclose *subSpec* in braces if it includes backslashes. If the **−all** argument is specified, then all ranges in *string* that match *exp* are found and substitution is performed for each of these ranges; otherwise only the first matching range is found and substituted. If **−all** is specified, then "&" and "*n*" sequences are handled for each substitution using the information from the corresponding match. If the **−nocase** argument is specified, then upper-case characters in *string* are converted to lowercase before matching against *exp*; however, substitutions specified by *subSpec* use the original unconverted form of *string*. The **−all** and **−nocase** arguments must be specified exactly: no abbreviations are permitted.

rename *oldName newName*

Rename the command that used to be called *oldName* so that it is now called *newName*. If *newName* is an empty string (e.g. {}) then *oldName* is deleted. The **rename** command returns an empty string as result.

return ?*value*?

Return immediately from the current procedure (or top-level command or **source** command), with *value* as the return value. If *value* is not specified, an empty string will be returned as result.

scan *string format varname1* ?*varname2 ...*?

This command parses fields from an input string in the same fashion as the C **sscanf** procedure. *String* gives the input to be parsed and *format* indicates how to parse it, using **%** fields as in **sscanf**. All of the **sscanf** options are valid; see the **sscanf** man page for details. Each *varname* gives the name of a variable; when a field is scanned from *string*, the result is converted back into a string and assigned to the corresponding *varname*. The only unusual conversion is for **%c**. For **%c** conversions a single character value is converted to a decimal string, which is then assigned to the corresponding *varname*; no field width may be specified for this conversion.

seek *fileId offset* ?*origin*?

Change the current access position for *fileId*. The *offset* and *origin* arguments specify the position at which the next read or write will occur for *fileId*. *Offset* must be a number (which may be negative) and *origin* must be one of the following:

start The new access position will be *offset* bytes from the start of the file.

current

The new access position will be *offset* bytes from the current access position; a negative *offset* moves the access position backwards in the file.

end The new access position will be *offset* bytes from the end of the file. A negative *offset* places the access position before the end-of-file, and a positive *offset* places the access position after the end-of-file.

The *origin* argument defaults to **start**. *FileId* must have been the return value from a previous call to **open**, or it may be **stdin**, **stdout**, or **stderr** to refer to one of the standard I/O channels. This command returns an empty string.

set *varname* ?*value*?

Returns the value of variable *varname*. If *value* is specified, then set the value of *varname* to *value*, creating a new variable if one doesn't already exist, and return its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. If no procedure is active, then *varname* refers to a global variable. If a procedure is active, then *varname* refers to a parameter or local variable of the procedure, unless the *global* command has been invoked to declare *varname* to be global.

source *fileName*

Read file *fileName* and pass the contents to the Tcl interpreter as a sequence of commands to execute in the normal fashion. The return value of **source** is the return value of the last command executed from the file. If an error occurs in executing the contents of the file, then the **source** command will return that error. If a **return** command is invoked from within the file, the remainder of the file will be skipped and the **source** command will return normally with the result from the **return** command. If *fileName* starts with a tilde, then it is tilde-substituted as described in the **Tcl_TildeSubst** manual entry.

split *string* ?*splitChars*?

Returns a list created by splitting *string* at each character that is in the *splitChars* argument. Each element of the result list will consist of the characters from *string* between instances of the characters in *splitChars*. Empty list elements will be generated if *string* contains adjacent characters in *splitChars*, or if the first or last character of *string* is in *splitChars*. If *splitChars* is an empty string then each character of *string* becomes a separate element of the result list. *SplitChars* defaults to the standard white-space characters. For example,

split "comp.unix.misc" .

returns **"comp unix misc"** and

split "Hello world" {}

returns **"H e l l o { } w o r l d"**.

string *option arg ?arg ...?*

Perform one of several string operations, depending on *option*. The legal *option*s (which may be abbreviated) are:

string compare *string1 string2*

Perform a character-by-character comparison of strings *string1* and *string2* in the same way as the C **strcmp** procedure. Return -1, 0, or 1, depending on whether *string1* is lexicographically less than, equal to, or greater than *string2*.

string first *string1 string2*

Search *string2* for a sequence of characters that exactly match the characters in *string1*. If found, return the index of the first character in the first such match within *string2*. If not found, return -1.

string index *string charIndex*

Returns the *charIndex*'th character of the *string* argument. A *charIndex* of 0 corresponds to the first character of the string. If *charIndex* is less than 0 or greater than or equal to the length of the string then an empty string is returned.

string last *string1 string2*

Search *string2* for a sequence of characters that exactly match the characters in *string1*. If found, return the index of the first character in the last such match within *string2*. If there is no match, then return −1.

string length *string*

Returns a decimal string giving the number of characters in *string*.

string match *pattern string*

See if *pattern* matches *string*; return 1 if it does, 0 if it doesn't. Matching is done in a fashion similar to that used by the C-shell. For the two strings to match, their contents must be identical except that the following special sequences may appear in *pattern*:

***** Matches any sequence of characters in *string*, including a null string.

? Matches any single character in *string*.

[*chars***]** Matches any character in the set given by *chars*. If a sequence of the form *x*−*y* appears in *chars*, then any character between *x* and *y*, inclusive, will match.

*****x* Matches the single character *x*. This provides a way of avoiding the special interpretation of the characters ***?[]** in *pattern*.

string range *string first last*

Returns a range of consecutive characters from *string*, starting with the character whose index is *first* and ending with the character whose index is *last*. An index of 0 refers to the first character of the string. *Last* may be **end** (or any abbreviation of it) to refer to the last character of the string. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were **end**. If *first* is greater than *last* then an empty string is returned.

string tolower *string*

Returns a value equal to *string* except that all upper case letters have been converted to lower case.

string toupper *string*

Returns a value equal to *string* except that all lower case letters have been converted to upper case.

string trim *string* ?*chars*?

Returns a value equal to *string* except that any leading or trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimleft *string* ?*chars*?

Returns a value equal to *string* except that any leading characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

string trimright *string* ?*chars*?

Returns a value equal to *string* except that any trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

tell *fileId*

Returns a decimal string giving the current access position in *fileId*. *FileId* must have been the return value from a previous call to **open**, or it may be **stdin**, **stdout**, or **stderr** to refer to one of the standard I/O channels.

time *command* ?*count*?

This command will call the Tcl interpreter *count* times to execute *command* (or once if *count* isn't specified). It will then return a string of the form

503 microseconds per iteration

which indicates the average amount of time required per iteration, in microseconds. Time is measured in elapsed time, not CPU time.

trace *option* ?*arg arg* ...?

Cause Tcl commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented. The legal *option*'s (which may be abbreviated) are:

trace variable *name ops command*

Arrange for *command* to be executed whenever variable *name* is accessed in one of the ways given by *ops*. *Name* may refer to a normal variable, an element of an array, or to an array as a whole (i.e. *name* may be just the name of an array, with no parenthesized index). If *name* refers to a whole array, then *command* is invoked whenever any element of the array is manipulated.

Ops indicates which operations are of interest, and consists of one or more of the following letters:

r Invoke *command* whenever the variable is read.

w Invoke *command* whenever the variable is written.

u Invoke *command* whenever the variable is unset. Variables can be unset explicitly with the **unset** command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to *command* so that the actual command is as follows:

command name1 name2 op

Name1 and *name2* give the name(s) for the variable being accessed: if the variable is a scalar then *name1* gives the variable's name and *name2* is an empty string; if the variable is an array element then *name1* gives the name of the array and name2 gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then *name1* gives the array name and *name2* is an empty string. *Op* indicates what operation is being performed on the variable, and is one of **r**, **w**, or **u** as defined above.

Command executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a Tcl procedure, then *command* will have access to the same local variables as code in the procedure. This context may be different than the context in which the trace was created. If *command* invokes a procedure (which it normally does) then the procedure will have to use **upvar** or **uplevel** if it wishes to access the traced variable. Note also that *name1* may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the **upvar** command.

For read and write traces, *command* can modify the variable to affect the result of the traced operation. If *command* modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from *command* is ignored except that if it returns an error of any sort then the traced operation is aborted with an error message saying that the access was denied (this mechanism can be used to implement read-only variables, for example). For write traces, *command* is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, *command* will have to restore the old value of the variable.

While *command* is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by *command* will occur directly, without invoking *command* (or any other traces) again.

When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset trace command creates a new trace and accesses the variable, the trace will be invoked.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the **trace vdelete** command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

This command returns an empty string.

trace vdelete *name ops command*
If there is a trace set on variable *name* with the operations and command given by *ops* and *command*, then the trace is removed, so that *command* will never again be invoked. Returns an empty string.

trace vinfo *name*
Returns a list containing one element for each trace currently set on variable *name*. Each element of the list is itself a list containing two elements, which are the *ops* and *command* associated with the trace. If *name* doesn't exist or doesn't have any traces set, then the result of the command will be an empty string.

unknown *cmdName* ?*arg arg ...*?

This command doesn't actually exist as part of Tcl, but Tcl will invoke it if it does exist. If the Tcl interpreter encounters a command name for which there is not a defined command, then Tcl checks for the existence of a command named **unknown**. If there is no such command, then the interpeter returns an error. If the **unknown** command exists, then it is invoked with arguments consisting of the fully-substituted name and arguments for the original non-existent command. The **unknown** command typically does things like searching through library directories for a command procedure with the name *cmdName*, or expanding abbreviated command names to full-length, or automatically executing unknown commands as UNIX sub-processes. In some cases (such as expanding abbreviations) **unknown** will change the original command slightly and then (re-)execute it. The result of the **unknown** command is used as the result for the original non-existent command.

unset *name* ?*name name ...*?

Remove one or more variables. Each *name* is a variable name, specified in any of the ways acceptable to the **set** command. If a *name* refers to an element of an array, then that element is removed without affecting the rest of the array. If a *name* consists of an array name with no parenthesized index, then the entire array is deleted. The **unset** command returns an empty string as result. An error occurs if any of the variables doesn't exist.

uplevel ?*level*? *command* ?*command ...*?

All of the *command* arguments are concatenated as if they had been passed to **concat**; the result is then evaluated in the variable context indicated by *level*. **Uplevel** returns the result of that evaluation. If *level* is an integer, then it gives a distance (up the procedure calling stack) to move before executing the command. If *level* consists of **#** followed by a number then the number gives an absolute level number. If *level* is omitted then it defaults to **1**. *Level* cannot be defaulted if the first *command* argument starts with a digit or **#**. For example, suppose that procedure **a** was invoked from top-level, and that it called **b**, and that **b** called **c**. Suppose that **c** invokes the **uplevel** command. If *level* is **1** or **#2** or omitted, then the command will be executed in the variable context of **b**. If *level* is **2** or **#1** then the command will be executed in the variable context of **a**. If *level* is **3** or **#0** then the command will be executed at top-level (only global variables will be visible). The **uplevel** command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose **c** invokes the command

uplevel 1 {set x 43; d}

where d is another Tcl procedure. The **set** command will modify the variable **x** in **b**'s context, and **d** will execute at level 3, as if called from **b**. If it in turn executes the command

uplevel {set x 42}

then the set command will modify the same variable **x** in **b**'s context: the procedure **c** does not appear to be on the call stack when **d** is executing. The command "**info level**" may be used to obtain the level of the current procedure. **Uplevel** makes it possible to implement new control constructs as Tcl procedures (for example, **uplevel** could be used to implement the **while** construct as a Tcl procedure).

upvar ?*level*? *otherVar myVar* ?*otherVar myVar ...*?

This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. *Level* may have any of the forms permitted for the **uplevel** command, and may be omitted if the first letter of the first *otherVar* isn't **#** or a digit (it defaults to **1**). For each *otherVar* argument, **upvar** makes the variable by that name in the procedure frame given by *level* (or at global level, if *level* is **#0**) accessible in the current procedure by the name given in the corresponding *myVar* argument. The variable named by *otherVar* need not exist at the time of the call; it will be created the first time *myVar* is referenced, just like an ordinary variable. **Upvar** may only be invoked from within procedures. Neither *otherVar* or *myVar* may refer to an element of an array. **Upvar** returns an empty string.

The **upvar** command simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as Tcl procedures. For example, consider the following procedure:

```
proc add2 name {
  upvar $name x
  set x [expr $x+2]
}
```

Add2 is invoked with an argument giving the name of a variable, and it adds two to the value of that variable. Although **add2** could have been implemented using **uplevel** instead of **upvar**, **upvar** makes it simpler for **add2** to access the variable in the caller's procedure frame.

while *test body*

The *while* command evaluates *test* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must be numeric; if it is non-zero then *body* is executed by passing it to the Tcl interpreter. Once *body* has been executed then *test* is evaluated again, and the process repeats until eventually *test* evaluates to a zero numeric value. **Continue** commands may be executed inside *body* to terminate the current iteration of the loop, and **break** commands may be executed inside *body* to cause immediate termination of the **while** command. The **while** command always returns an empty string.

BUILT-IN VARIABLES

The following global variables are created and managed automatically by the Tcl library. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users.

env

This variable is maintained by Tcl as an array whose elements are the environment variables for the process. Reading an element will return the value of the corresponding environment variable. Setting an element of the array will modify the corresponding environment variable or create a new one if it doesn't already exist. Unsetting an element of **env** will remove the corresponding environment variable. Changes to the **env** array will affect the environment passed to children by commands like **exec**. If the entire **env** array is unset then Tcl will stop monitoring **env** accesses and will not update environment variables.

errorCode

After an error has occurred, this variable will be set to hold additional information about the error in a form that is easy to process with programs. **errorCode** consists of a Tcl list with one or more elements. The first element of the list identifies a general class of errors, and determines the format of the rest of the list. The following formats for **errorCode** are used by the Tcl core; individual applications may define additional formats.

CHILDKILLED *pid sigName msg*

This format is used when a child process has been killed because of a signal. The second element of **errorCode** will be the process's identifier (in decimal). The third element will be the symbolic name of the signal that caused the process to terminate; it will be one of the names from the include file signal.h, such as **SIGPIPE**. The fourth element will be a short human-readable message describing the signal, such as "write on pipe with no readers" for **SIGPIPE**.

CHILDSTATUS *pid code*

This format is used when a child process has exited with a non-zero exit status. The second element of **errorCode** will be the process's identifier (in decimal) and the third element will be the exit code returned by the process (also in decimal).

CHILDSUSP *pid code*

This format is used when a child process has been suspended because of a signal. The second element of **errorCode** will be the process's identifier, in decimal. The third element will be the symbolic name of the signal that caused the process to suspend; this will be one of the names from the include file signal.h, such as **SIGTTIN**. The fourth element will be a short human-readable message describing the signal, such as ''background tty read'' for **SIGTTIN**.

NONE

This format is used for errors where no additional information is available for an error besides the message returned with the error. In these cases **errorCode** will consist of a list containing a single element whose contents are **NONE**.

UNIX *errName msg*

If the first element of **errorCode** is **UNIX**, then the error occurred during a UNIX kernel call. The second element of the list will contain the symbolic name of the error that occurred, such as **ENOENT**; this will be one of the values defined in the include file errno.h. The third element of the list will be a human-readable message corresponding to *errName*, such as ''no such file or directory'' for the **ENOENT** case.

To set **errorCode**, applications should use library procedures such as **Tcl_SetErrorCode** and **Tcl_UnixError**, or they may invoke the **error** command. If one of these methods hasn't been used, then the Tcl interpreter will reset the variable to **NONE** after the next error.

errorInfo

After an error has occurred, this string will contain one or more lines identifying the Tcl commands and procedures that were being executed when the most recent error occurred. Its contents take the form of a stack trace showing the various nested Tcl commands that had been invoked at the time of the error.

AUTHOR

John Ousterhout, University of California at Berkeley (ouster@sprite.berkeley.edu)

Many people have contributed to Tcl in various ways, but the following people have made unusually large contributions:

Bill Carpenter
Peter Da Silva
Mark Diekhans
Karl Lehenbauer
Mary Ann May-Pumphrey