ABORT_PAGE - Neowebscript

Aborts execution immediately

Synopsis:

- **abort_page**

Description

Aborts *NeoWebScript* execution immediately. Does not emit any of the rest of the webpage to the requester. Remaining code in the script being executed will still be executed, so *abort_page* is typically immediately followed by *return*.

ACCESS_DATA_FILE - Neowebscript

Opens the specified datafile for reading and writing, creating if not there

Synopsis:

- **access_data_file** *datafile*

Description

access_data_file opens the specified datafile for reading and writing, creating it if its not there. Datafile name can consist of upper and lowercase letters and numbers only. No slashes, no periods, etc. The file can then be operated on in the normal manner by gets, puts, seek, tell, flush, close, etc.

APPEND - Safe TCL

Append to variable

Synopsis:

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

Description

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.

Keywords:

append, variable

ARRAY - Safe TCL

Manipulate array variables

Synopsis:

- **arrray***option arrayName ?arg arg ...?*

Description

This command performs one of several operations on the variable given by *arrayName*. Unless otherwise specified for individual commands below, *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 exists *arrayName*

Returns 1 if *arrayName* is an array variable, 0 if there is no variable by that name or if it is a scalar variable.

array get *arrayName* ?*pattern*?
Returns a list containing pairs of elements. The first element in each pair is the name of an element in *arrayName* and the second element of each pair is the value of the array element. The order of the pairs is undefined. If *pattern* is not specified, then all of the elements of the array are included in the result. If *pattern* is specified, then only those elements whose names match *pattern* (using the glob-style matching rules of **string match**) are included. If *arrayName* isn´t the name of an array variable, or if the array contains no elements, then an empty list is returned.

array names *arrayName* ?*pattern*?
Returns a list containing the names of all of the elements in the array that match *pattern* (using the glob-style matching rules of **string match**). If *pattern* is omitted then the command returns all of the element names in the array. If there are no (matching) elements in the array, or if *arrayName* isn´t the name of an array variable, 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 set *arrayName list*
Sets the values of one or more elements in *arrayName*. *list* must have a form like that returned by **array get**, consisting of an even number of elements. Each odd-numbered element in *list* is treated as an element name within *arrayName*, and the following element in *list* is used as a new value for that array element.

array size *arrayName*
Returns a decimal string giving the number of elements in the array. If *arrayName* isn´t the name of an array then 0 is returned.

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.

See Also:

bgerror

Keywords:

array, element names, search

BACKLINK - Neowebscript

Emit a hyperlink pointing to the referer URL

Synopsis:

- **backlink** *[linktext] [nolinktext]*

Description

Emit a hyperlink pointing to the referer URL, with *linktext* emitted as HTML text within the hyperlink. Emits *nolinktext* if there was no referer.

See Also:

bgerror

Keywords:

array, element names, search

BREAK - Safe TCL

Abort looping command

Synopsis:

- **break**

Description

This command is typically invoked inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL_BREAK code, which causes a break exception to occur. The exception causes the current script to be aborted out to the innermost containing loop command, which then aborts its execution and returns normally. Break exceptions are also handled in a few other situations, such as the **catch** command, Tk event bindings, and the outermost scripts of procedure bodies.

See Also:

bgerror

Keywords:

abort, break, loop

BSEARCH - Extended TCL

Searches an sorted file for a match

Synopsis:

- **bsearch** fileId key ?retvar? ?compare_proc?

Description

Search an opened file fileId containing lines of text sorted into ascending order for a match. Key contains the string to match. If retvar is specified, then the line from the file is returned in retvar, and the command returns 1 if key was found, and 0 if it wasn´t. If retvar is not specified or is a null name, then the command returns the line that was found, or an empty string if key wasn´t found.

By default, the key is matched against the first white-space separated field in each line. The field is treated as an ASCII string. If compare_proc is specified, then it defines the name of a Tcl procedure to evaluate against each line read from the sorted file during the execution of the bsearch command. Compare_proc takes two arguments, the key and a line extracted from the file. The compare routine should return a number less than zero if the key is less than the line, zero if the key matches the line, or greater than zero if the key is greater than the line. The file must be sorted in ascending order according to the same criteria compare_proc uses to compare the key with the line, or erroneous results will occur.

See Also:

bgerror

Keywords:

abort, break, loop

CASE - Safe TCL

Evaluate one of several scripts, depending on a given value

Synopsis:

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

Description

Note: the **case** *command is obsolete and is supported only for backward compatibility. At some point in the future it may be removed entirely. You should use the* **switch** *command instead.*

The **case** command matches *string* against each of the *patList* arguments in order. Each *patList* argument is a list of one or more patterns. If any of these patterns matches *string* then **case** evaluates the following *body* argument by passing it recursively to the Tcl interpreter and returns 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 for the *patList* and *body* arguments. 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.

See Also:

bgerror

Keywords:

case, match, regular expression

CATCH - Safe TCL

Evaluate script and trap exceptional returns

Synopsis:

- **catch** *script ?varName?*

Description

The **catch** command may be used to prevent errors from aborting command interpretation. **Catch** calls the Tcl interpreter recursively to execute *script*, and always returns a TCL_OK code, regardless of any errors that might occur while executing *script*. The return value from **catch** is a decimal string giving the code returned by the Tcl interpreter after executing *script*. This will be **0** (TCL_OK) if there were no errors in *script*; 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 variable to the string returned from *script* (either a result or an error message).

Note that **catch** catches all exceptions, including those generated by **break** and **continue** as well as errors.

See Also:

bgerror

Keywords:

catch, error

CCOLLATE - Extended TCL

Compares two strings

Synopsis:

- **ccollate** ?-local? string1 string2

Description

This command compares two strings. If returns -1 if string1 is less than string2, 0 if they are equal amd 1 if string1 is greater than string2.

If -local is specified, the strings are compared according to the collation environment of the cur- rent locale.

See Also:

bgerror

Keywords:

catch, error

CEQUAL - Extended TCL

Compares two strings for equality

Synopsis:

- **cequal** string1 string2

Description

This command compares two strings for equality. It returns 1 if string1 and string2 are the identical and 0 if they are not. This command is a short-cut for string compare and avoids the problems with string expressions being treated unintentionally as numbers.

See Also:

bgerror

Keywords:

catch, error

CINDEX - Extended TCL

Returns character indexed by expression indexExpr

Synopsis:

- **cindex** string indexExpr

Description

Returns the character indexed by the expression indexExpr (zero based) from string.

If the expression indexExpr starts with the string end, then end is replaced with the index of the last character in the string. If the expression starts with len, then len is replaced with the length of the string.

See Also:

bgerror

Keywords:

catch, error

CLENGTH - Extended TCL

Returns length of string (number of characters)

Synopsis:

- **clength** string

Description

Returns the length of string in characters. This command is a shortcut for:

string length string

See Also:

bgerror

Keywords:

catch, error

CLOCK - Safe TCL

Obtain and manipulate time

Synopsis:

- **clock** *option?arg arg...?*

Description

This command performs one of several operations that may obtain or manipulate strings or values that represent some notion of time. The *option* argument determines what action is carried out by the command. The legal *options* (which may be abbreviated) are:

clock clicks
Return a high-resolution time value as a systemdependent integer value. The unit of the value is system-dependent but should be the highest resolution clock available on the system such as a CPU cycle counter. This value should only be used for the relative measurement of elapsed time.

clock format *clockValue* ?**-format** *string*? ?**-gmt** *boolean*? Converts an integer time value, typically returned by **clock seconds**, **clock scan**, or the **atime**, **mtime**, or **ctime** options of the **file** command, to humanreadable form. If the **-format** argument is present the next argument is a string that describes how the date and time are to be formatted. Field descriptors consist of a **%** followed by a field descriptor character. All other characters are copied into the result. Valid field descriptors are:

%%
Insert a %.

%a
Abbreviated weekday name. (Mon, Tue, etc.)

%A
Full weekday name. (Monday, Tuesday, etc.)

%b
Abbreviated month name. (Jan, Feb, etc.)

%B
Full month name.

%d
Day of month (01 - 31).

%D
Date as %m/%d/%y.

%e
Day of month (1 - 31), no leading zeros.

%h

Abbreviated month name.

%H
Hour (00 - 23).

%I
Hour (00 - 12).

%j
Day number of year (001 - 366).

%m
Month number (01 - 12).

%M
Minute (00 - 59).

%n
Insert a newline.

%p
AM or PM.

%r
Time as %I:%M:%S %p.

%R
Time as %H:%M.

%S
Seconds (00 - 59).

%t
Insert a tab.

%T
Time as %H:%M:%S.

%U
Week number of year (01 - 52), Sunday is the first day of the week.

%w
Weekday number (Sunday = 0).

%W
Week number of year (01 - 52), Monday is the first day of the week.

%x

Local specific date format.

%X
Local specific time format.

%y
Year within century (00 - 99).

%Y
Year as ccyy (e.g. 1990)

%Z
Time zone name.

If the **-format** argument is not specified, the format string "**%a %b %d %H:%M:%S %Z %Y**" is used. If the **-gmt** argument is present the next argument must be a boolean which if true specifies that the time will be formated as Greenwich Mean Time. If false then the local timezone will be used as defined by the operating environment.

clock scan *dateString* ?**-base** *clockVal*? ?**-gmt** *boolean*? Convert *dateString* to an integer clock value (see **clock seconds**). This command can parse and convert virtually any standard date and/or time string, which can include standard time zone mnemonics. If only a time is specified, the current date is assumed. If the string does not contain a time zone mnemonic, the local time zone is assumed, unless the **-gmt** argument is true, in which case the clock value is calculated assuming that the specified time is relative to Greenwich Mean Time.

If the **-base** flag is specified, the next argument should contain an integer clock value. Only the date in this value is used, not the time. This is useful for determining the time on a specific day or doing other date-relative conversions.

The *dateString* consists of zero or more specifications of the following form:

time A time of day, which is of the form: *hh*?*:mm*?*:ss*?? ?*meridian*? ?*zone*? or *hhmm* ?*meridian*? ?*zone*?. If no meridian is specified, *hh* is interpreted on a 24-hour clock.

date
A specific month and day with optional year. The acceptable formats are *mm/dd*?*/yy*?, *monthname dd* ?, *yy*?, *dd monthname* ?*yy*? and *day, dd monthname yy*. The default year is the current year. If the year is less then 100, then 1900 is added to it.

relative time
A specification relative to the current time. The format is *number unit* acceptable units are *year, fortnight, month, week, day, hour, minute* (or *min*), and *second* (or *sec*). The unit can be specified as a singular or plural, as in *3 weeks*. These modifiers may also be specified: *tomorrow, yesterday, today, now, last, this, next, ago.*

The actual date is calculated according to the following steps. First, any absolute date and/or time is processed and converted. Using that time as the base, day-of-week specifications are added. Next,

relative specifications are used. If a date or day is specified, and no absolute or relative time is given, midnight is used. Finally, a correction is applied so that the correct hour of the day is produced after allowing for daylight savings time dif ferences.

clock seconds
Return the current date and time as a systemdependent integer value. The unit of the value is seconds, allowing it to be used for relative time calculations. The value is usually defined as total elapsed time from an "epoch". You shouldn´t assume the value of the epoch.

See Also:

bgerror

Keywords:

clock, date, time

CLOSE - Safe TCL

Close an open channel

Synopsis:

- **close** *channelID*

Description

Closes the channel given by *channelId*. *ChannelId* must be a channel identifier such as the return value from a previous **open** or **socket** command. All buffered output is flushed to the channel´s output device, any buffered input is discarded, the underlying file or device is closed, and *channelId* becomes unavailable for use.

If the channel is blocking, the command does not return until all output is flushed. If the channel is nonblocking and there is unflushed output, the channel remains open and the command returns immediately; output will be flushed in the background and the channel will be closed when all the flushing is complete.

If *channelId* is a blocking channel for a command pipeline then **close** waits for the child processes to complete.

If the channel is shared between interpreters, then **close** makes *channelId* unavailable in the invoking interpreter but has no other effect until all of the sharing interpreters have closed the channel. When the last interpreter in which the channel is registered invokes **close**, the cleanup actions described above occur. See the **interp** command for a description of channel sharing.

Channels are automatically closed when an interpreter is destroyed and when the process exits. Channels are switched to blocking mode, to ensure that all output is correctly flushed before the process exits.

The command returns an empty string, and may generate an error if an error occurs while flushing output.

See Also:

bgerror

Keywords:

blocking, channel, close, nonblocking

CMDTRACE - Neowebscript

Synopsis:

- **cmdtrace** *args*

Description

The Extended Tcl command **cmdtrace** will work if supervisor mode is enabled for the directory.

See Also:

bgerror

Keywords:

blocking, channel, close, nonblocking

CONCAT - Safe TCL

Join lists together

Synopsis:

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

Description

This command treats each argument as a list and concatenates them into a single list. It also eliminates leading and trailing spaces in the *arg´*s and adds a single separator space between *arg´*s. 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.

If no *arg*s are supplied, the result is an empty string.

See Also:

bgerror

Keywords:

concatenate, join, lists

CONTINUE - Safe TCL

Skip to the next iteration of a loop

Synopsis:

- **continue**

Description

This command is typically invoked inside the body of a looping command such as **for** or **foreach** or **while**. It returns a TCL_CONTINUE code, which causes a continue exception to occur. The exception causes the current script to be aborted out to the the innermost containing loop command, which then continues with the next iteration of the loop. Catch exceptions are also handled in a few other situations, such as the **catch** command and the outermost scripts of procedure bodies.

See Also:

bgerror

Keywords:

`continue, iteration, loop`

`CRANGE - Extended TCL`

Returns a range of characters from string

Synopsis:

- **crange** string firstExpr lastExpr

Description

Returns a range of characters from string starting at the character indexed by the expression first- Expr (zero-based) until the character indexed by the expression lastExpr.

If the expression firstExpr or lastExpr starts with the string end, then end is replaced with the index of the last character in the string. If the expression starts with len, then len is replaced with the length of the string.

See Also:

`bgerror`

Keywords:

`continue, iteration, loop`

`CSUBSTR - Extended TCL`

Returns a range of characters from string for lengthExpr characters

Synopsis:

- **crange** string firstExpr lengthExpr

Description

Returns a range of characters from string starting at the character indexed by the expression first- Expr (zero-based) for lengthExpr characters.

If the expression firstExpr or lengthExpr starts with the string end, then end is replaced with the index of the last character in the string. If the expression starts with len, then len is replaced with the length of the string.

See Also:

bgerror

Keywords:

continue, iteration, loop

CTOKEN - Extended TCL

Parse a token out of a character string

Synopsis:

- **ctoken** strvar separators

Description

Parse a token out of a character string. The string to parse is contained in the variable named strvar. The string separators contains all of the valid separator characters for tokens in the string. All leading separators are skipped and the first token is returned. The variable strvar will be modified to contain the remainder of the string following the token.

See Also:

bgerror

Keywords:

continue, iteration, loop

CTYPE - Extended TCL

Determines if all characters in string are of the specified class

Synopsis:

- **`ctype`** ?-failindex var? class string

Description

ctype determines whether all characters in string are of the specified class. It returns 1 if they are all of class, and 0 if they are not, or if the string is empty. This command also provides another method (besides format and scan) of con- verting between an ASCII character and its numeric value. The following ctype commands are available:

ctype ?-failindex var? alnum string Tests that all characters are alphabetic or numeric characters as defined by the character set.

ctype ?-failindex var? alpha string Tests that all characters are alphabetic char- acters as defined by the character set.

ctype ?-failindex var? ascii string Tests that all characters are an ASCII charac- ter (a non-negative number less than 0200).

ctype char number Converts the numeric value, string, to an ASCII character. Number must be in the range 0 through 255.

ctype ?-failindex var? cntrl string Tests that all characters are "control char- acters" as defined by the character set.

ctype ?-failindex var? digit string Tests that all characters are valid decimal digits, i.e. 0 through 9.

ctype ?-failindex var? graph string Tests that all characters within are any char- acter for which ctype print is true, except for space characters.

ctype ?-failindex var? lower string Tests that all characters are lowercase let- ters as defined by the character set.

ctype ord character Convert a character into its decimal numeric value. The first character of the string is converted.

ctype ?-failindex var? space string Tests that all characters are either a space, horizontal-tab, carriage return, newline, ver- tical-tab, or form-feed.

ctype ?-failindex var? print string Tests that all characters are a space or any character for which ctype alnum or ctype punct is true or other "printing character" as defined by the character set.

ctype ?-failindex var? punct string Tests that all characters are made up of any of the characters other

than the ones for which alnum, cntrl, or space is true.

ctype ?-failindex var? upper string Tests that all characters are uppercase let- ters as defined by the character set.

ctype ?-failindex var? xdigit string Tests that all characters are valid hexadeci- mal digits, that is 0 through 9, a through f or A through F.

If -failindex is specified, then the index into string of the first character that did not match the class is returned in var.

See Also:

bgerror

Keywords:

continue, iteration, loop

DBDELKEY - Neowebscript

Delete in database entry corresponding to key

Synopsis:

- **dbdelkey** *database key*

Description

Delete in database *database* the entry corresponding to *key*. If the entry isn´t there or the datbase doesn´t exist, returns **0**, otherwise **1** (or an error).

See Also:

bgerror

Keywords:

continue, iteration, loop

DBEXISTS - Neowebscript

Determines if database exists

Synopsis:

- **dbexists** *database*

Description

Returns **1** if *database* exists, **0** otherwise.

See Also:

bgerror

Keywords:

continue, iteration, loop

DBFETCH - Neowebscript

Fetch from database into the array specified

Synopsis:

- **dbfetch** *database key arrayName -singleVar*

Description

Fetch from database *database* into the array specified by *arrayName* the key-value pairs for the specified *key*, previously stored by *dbstore* (This was formerly called *props_to_array*.)

If -singleVar is specified, then arrayName will be treated as the name of a variable to store the results of the lookup. This should be used with *dbstore*, when -singleVar is used.

See Also:

bgerror

Keywords:

continue, iteration, loop

DBFILES - Neowebscript

Returns a list of databases

Synopsis:

- **dbfiles**

Description

Returns a list of the databases in the webpage owner´s database directory.

See Also:

bgerror

Keywords:

continue, iteration, loop

DBKEYS - Neowebscript

Return a list of all of the keys in the specified database matching pattern

Synopsis:

- **dbkeys** *database pattern*

Description

Return a list of all of the keys in the database *database* matching pattern. Pattern is in the style of a Unix wildcard. In other words, if you were looking for a key starting with **horace**, the pattern would be **horace***. To match **horace** anywhere in the string, ***horace***.

See Also:

bgerror

Keywords:

continue, iteration, loop

DBSTORE - Neowebscript

Store the contents of the specified array into a database

Synopsis:

- **dbstore** *database key arrayName ?-singleVar?*

Description

Store the contents of the array specified by *arrayName* into the database *database*, indexed by *key*, as key-value pairs. (This was formerly called *array_to_props*.)

If -singleVar is specified, then arrayName will be treated as a single value, and that value will be stored under the key given.

See Also:

bgerror

Keywords:

continue, iteration, loop

DELETE_DATA_FILE - Neowebscript

Deletes a datafile

Synopsis:

- **delete_data_file** *datafile*

Description

delete_data_file deletes a datafile. It is not an error to delete a datafile that is not there.

See Also:

bgerror

Keywords:

continue, iteration, loop

DIRECTORY_LISTING - Neowebscript

Displays disk usage for the current directory

Synopsis:

- **directory_listing**

Description

Displays disk usage for the current directory. Also shows K-bytes used from the current directory on down.

See Also:

bgerror

Keywords:

continue, iteration, loop

DUMP_ENVIRONMENT - Neowebscript

Write all of server´s environment vars and values

Synopsis:

- **dump_environment**

Description

Write all of the server´s environment vars and values into webpage.

See Also:

bgerror

Keywords:

continue, iteration, loop

EMIT_STANDARD_
MAGELLAN_SEARCH_LINK - NeoWebScript

Emits a Magellan link.

Synopsis:

- **emit_standard_magellan_search_link**

Description

Emit a Magellan link with at-link searching.

See Also:

bgerror

Keywords:

continue, iteration, loop

EMIT_STANDARD_
YAHOO_LINK - NeoWebScript

Emits a standard Yahoo link.

Synopsis:

- **emit_standard_yahoo_link**

Description

Emit a link back to Yahoo in the manner that they ask.

See Also:

bgerror

Keywords:

continue, iteration, loop

EOF - Safe TCL

Check for end of file condition on channel

Synopsis:

- **eof** *channelID*

Description

Returns 1 if an end of file condition occurred during the most recent input operation on *channelId* (such as **gets**), 0 otherwise.

See Also:

bgerror

Keywords:

channel, end of file

ERROR - Safe TCL

Generate an error

Synopsis:

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

Description

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 **tclvars** manual page 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.

See Also:

bgerror

Keywords:

error, errorCode, errorInfo

ESTIMATE_HITS_PER_HOUR - Neowebscript

Return an estimate of the number of hits to be served on the next hour

Synopsis:

- **estimate_hits_per_hour**

Description

Return a numeric estimate of the number of hits that will be served in the next hour. Works by seeking back approximately 1000 hits into the log file and sees how long ago it was, to extrapolate hits-per-hour. If there are fewer than 1000 hits in the access log file, it will return **0**.

See Also:

bgerror

Keywords:

error, errorCode, errorInfo

EVAL - Safe TCL

Evaluate a Tcl script

Synopsis:

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

Description

Eval takes one or more arguments, which together comprise a Tcl script containing one or more commands. **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).

See Also:

bgerror

Keywords:

concatenate, evaluate, script

EXPR - Safe TCL

Evaluate an expression

Synopsis:

- **expr** *arg ?arg arg ...?*

Description

Concatenates *arg´*s (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value. 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
expr 8.2 + 6
evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons.

OPERANDS

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 floatingpoint number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSIcompliant 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 floatingpoint.

[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.

[6]
As a mathematical function whose arguments have any of the above forms for operands, such as "**sin($x)**". See below for a list of defined functions.

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 command on the left side of each of the lines below will produce the value on the right side of the line:

expr 3.1 + $a
6.1

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

OPERATORS

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

-
**+ ~! ** Unary minus, unary plus, 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. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.

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

<< >> Left and right shift. Valid for integer operands only. A right shift always propogates the sign bit.

< > <= >= 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.

^

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 leftto-right within the same precedence level. For example, the command
expr 4*2 < 7
returns 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 the command
expr {$v ? [a] : [b]}
only one of **[a]** or **[b]** will actually be evaluated, depending on the value of **$v**. Note, however, that this is only true if the entire expression is enclosed in braces; otherwise the Tcl parser will evaluate both **[a]** and **[b]** before invoking the **expr** command.

MATH FUNCTIONS

Tcl supports the following mathematical functions in expressions:

acos cos hypot sinh asin cosh log sqrt atan exp log10 tan atan2 floor pow tanh ceil fmod sin

Each of these functions invokes the math library function of the same name; see the manual entries for the library functions for details on what they do. Tcl also implements the following functions for conversion between integers and floating-point numbers:

abs(arg)
Returns the absolute value of *arg*. *Arg* may be either integer or floating-point, and the result is returned in the same form.

double(arg)
If *arg* is a floating value, returns *arg*, otherwise converts *arg* to floating and returns the converted value.

int(arg)

If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by truncation and returns the converted value.

round(arg)
If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by rounding and returns the converted value.

In addition to these predefined functions, applications may define additional functions using **Tcl_CreateMathFunc**().

TYPES, OVERFLOW, AND PRECISION

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, integer overflow and underflow are generally not detected reliably for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable.

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, **expr 5 / 4**
returns 1, while
expr 5 / 4.0
expr 5 / ([string length "abcd"] + 0.0) both return 1.25. Floating-point values are always returned with a "." or an "e" so that they will not look like integer values. For example, **expr 20.0/5.0**
returns "4.0", not "4". The global variable **tcl_precision** determines the the number of significant digits that are retained when floating values are converted to strings (except that trailing zeroes are omitted). If **tcl_precision** is unset then 6 digits of precision are used. To retain all of the significant bits of an IEEE floating-point number set **tcl_precision** to 17; if a value is converted to string with 17 digits of precision and then converted back to binary for some later calculation, the resulting binary value is guaranteed to be identical to the original one.

STRING OPERATIONS

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 commands
expr {"0x03" > "2"}
expr {"0y" < "0x12"}
both return 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". Because of Tcl´s tendency to treat values as numbers whenever possible, it isn´t generally a good idea to use operators like == when you really want string comparison and the values of the operands could be arbi

trary;
it´s better in these cases to use the **string compare** command instead.

See Also:

bgerror

Keywords:

arithmetic, boolean, compare, expression

FCONFIGURE - Safe TCL

Set and get options on a channel

Synopsis:

- **fconfigure** *channelId*
- **fconfigure** *channelId name*
- **fconfigure** *channelId name value ?name value ...?*

Description

The **fconfigure** command sets and retrieves options for channels. *ChannelId* identifies the channel for which to set or query an option. If no *name* or *value* arguments are supplied, the command returns a list containing alternating option names and values for the channel. If *name* is supplied but no *value* then the command returns the current value of the given option. If one or more pairs of *name* and *value* are supplied, the command sets each of the named options to the corresponding *value*; in this case the return value is an empty string.

The options described below are supported for all channels. In addition, each channel type may add options that only it supports.

-blocking *boolean*
The **-blocking** option determines whether I/O operations on the channel can cause the process to block indefinitely. The value of the option must be a proper boolean value. Channels are normally in blocking mode; if a channel is placed into nonblocking mode it will affect the operation of the **gets**, **read**, **puts**, **flush**, and **close** commands; see the documentation for those commands for details. Setting **stdin**, **stdout**, or **stderr** into nonblocking mode is disallowed because it could interfere with the operation of the parent process. For nonblocking mode to work correctly, the application must be using the Tcl event loop (e.g. by calling **Tcl_DoOneEvent** or invoking the

vwait command).

-buffering *newValue*

If *newValue* is **full** then the I/O system will buffer output until its internal buffer is full or until the **flush** command is invoked. If *newValue* is **line**, then the I/O system will automatically flush output for the channel whenever a newline character is output. If *newValue* is **none**, the I/O system will flush automatically after every output operation. The default is for **-buffering** to be set to **full** except for channels that connect to terminal-like devices; for these channels the initial setting is **line**.

-buffersize *newSize*

Newvalue must be an integer; its value is used to set the size of buffers, in bytes, subsequently allocated for this channel to store input or output. *Newvalue* must be between ten and one million, allowing buffers of ten to one million bytes in size.

-translation *mode*

In Tcl scripts the end of a line is always represented using a single newline character (\n). However, in actual files and devices the end of a line may be represented differently on different platforms, or even for different devices on the same platform. For example, under UNIX newlines are used in files, whereas carriage-return-linefeed sequences are normally used in network connections. The Tcl I/O system automatically translates between the newlines used internally and whatever end-ofline characters are appropriate for the underlying files and devices. The **-translation** option selects the end-of-line character(s) to use on the file or device for a particular channel. Note that if you want no translations at all applied to input and output on a specific channel, you should set it to **lf** mode - see below. The following values are currently supported:

cr

The end of a line in the underlying file or device is represented by a single carriage return character. During input commands such as **read** Tcl translates carriage returns to newline characters, and during output commands such as **puts** Tcl translates newline characters to carriage returns. This mode is typically used on Macintosh platforms.

crlf The end of a line in the underlying file or device is represented by a carriage return character followed by a linefeed character. During input commands such as **read** Tcl

translates

carriage-return-linefeed sequences to newlines, and during output commands such as **puts** Tcl translates newline characters to carriage-return-linefeed sequences. This mode is typically used on Windows platforms and for network connections.

lf

The end of a line in the underlying file or device is represented by a single newline (linefeed) character. In this mode no translations occur during either intput or output. This mode is typically used on UNIX platforms.

binary This is a synonym for *lf* mode.

platform

nel according to the platform on which the application is executing; for sockets on all platforms Tcl chooses **crlf**, for all Unix flavors, it chooses **lf**, for the Macintosh platform it chooses **cr** and for the various flavors of Windows it chooses **crlf**.

auto In this mode Tcl sets the translation mode based on the first valid end-of-line sequence it sees during input on the channel. If output is generated for the channel before an end-of-line sequence has appeared on input, then Tcl uses **platform** mode for the channel. The default setting for **-translation** is **auto**.

See Also:

`close, flush, gets, puts, read`

Keywords:

`blocking, buffering, carriage return, end of line, flushing, linemode, newline, nonblocking, platform, translation`

`FILEMV - Neowebscript`

Renames the file type

Synopsis:

- **filemv** *type oldname newname*

Description

Renames the file type *type* from *oldname* to *newname*. Valid *type*s are **db** and **data**.

See Also:

`close, flush, gets, puts, read`

Keywords:

`blocking, buffering, carriage return, end of line, flushing, linemode, newline, nonblocking, platform, translation`

`FILERM - Neowebscript`

Removes the data or db file of type

Synopsis:

- **filerm** *type name*

Description

Removes the data or db file of the given *type*. Valid *type*s are **db** and **data**.

See Also:

close, flush, gets, puts, read

Keywords:

blocking, buffering, carriage return, end of line, flushing, linemode, newline, nonblocking, platform, translation

---

FLOCK - Extended TCL

Places a lock on all or part of fole specified by fileId

Synopsis:

- **flock** options fileId ?start? ?length? ?origin?

Description

This command places a lock on all or part of the file specified by fileId. The lock is either advi- sory or mandatory, depending on the mode bits of the file. The lock is placed beginning at relative byte offset start for length bytes. If start or length is omitted or empty, zero is assumed. If length is zero, then the lock always extents to end of file, even if the file grows. If origin is "start", then the offset is relative to the begin- ning of the file. If it is "current", it is rela- tive to the current access position in the file. If it is "end", then it is relative to the end-of- file (a negative is before the EOF, positive is after). If origin is omitted, start is assumed.

The following options are recognized:

-read - Place a read lock on the file. Multiple processes may be accessing the file with read- locks.

-write - Place a write lock on the file. Only one process may be accessing a file if there is a write lock.

-nowait - If specified, then the process will not block if the lock can not be obtained. With this option, the command returns 1 if the lock is obtained and 0 if it is not.

See your system´s fcntl system call documentation for full details of the behavior of file locking. If locking is being done on ranges of a file, it is best to use unbuffered file access (see the fcntl command).

See Also:

`close, flush, gets, puts, read`

Keywords:

`blocking, buffering, carriage return, end of line, flushing, linemode, newline, nonblocking, platform, translation`

FLUSH - Safe TCL

Flush buffered output for a channel

Synopsis:

- **flush** *channelId*

Description

Flushes any output that has been buffered for *channelId*. *ChannelId* must be a channel identifier such as returned by a previous **open** or **socket** command, and it must have been opened for writing. If the channel is in blocking mode the command does not return until all the buffered output has been flushed to the channel. If the channel is in nonblocking mode, the command may return before all buffered output has been flushed; the remainder will be flushed in the background as fast as the underlying file or device is able to absorb it.

See Also:

`open, socket`

Keywords:

`blocking, buffer, channel, flush, nonblocking, output`

For loop

Synopsis:

- **for** *start test next body*

Description

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.

See Also:

open, socket

Keywords:

for, iteration, looping

Iterate over all elements in one or more lists

Synopsis:

- **foreach** *varname list body*
- **foreach** *varlist1 list1 varlist2 list2...? body*

Description

The **foreach** command implements a loop where the loop variable(s) take on values from one or more lists. In the simplest case there is one loop variable, *varname*, and one list, *list*, that is a list of values to

assign to *varname*. The *body* argument is a Tcl script. For each element of *list* (in order from first to last), **foreach** assigns the contents of the element to *varname* as if the **lindex** command had been used to extract the element, then calls the Tcl interpreter to execute *body*.

In the general case there can be more than one value list (e.g., *list1* and *list2*), and each value list can be associated with a list of loop variables (e.g., *varlist1* and *varlist2*). During each iteration of the loop the variables of each *varlist* are assigned consecutive values from the corresponding *list*. Values in each *list* are used in order from first to last, and each value is used exactly once. The total number of loop iterations is large enough to use up all the values from all the value lists. If a value list does not contain enough elements for each of its loop variables in each iteration, empty values are used for the missing elements.

The **break** and **continue** statements may be invoked inside *body*, with the same effect as in the **for** command. **Foreach** returns an empty string.

EXAMPLES

The following loop uses i and j as loop variables to iterate over pairs of elements of a single list.

```
set x {}
foreach {i j} {a b c d e f} { lappend x $j $i
}
# The value of x is "b a d c f e" # There are 3 iterations of the loop.
```

The next loop uses i and j to iterate over two lists in parallel.

```
set x {}
foreach i {a b c} j {d e f g} { lappend x $i $j
}
# The value of x is "a d b e c f {} g" # There are 4 iterations of the loop.
```

The two forms are combined in the following example.

```
set x {}
foreach i {a b c} {j k} {d e f g} { lappend x $i $j $k
}
# The value of x is "a d e b f g c {} {}" # There are 3 iterations of the loop.
```

See Also:

open, socket

Keywords:

foreach, iteration, list, looping

Format a string in the style of sprintf

Synopsis:

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

Description

This command generates a formatted string in the same way as the ANSI C **sprintf** procedure (it uses **sprintf** in its implementation). *FormatString* indicates how to format the result, using **%** conversion specifiers as in **sprintf**, and the additional arguments, if any, provide values to be substituted into the result. The return value from **format** is the formatted string.

DETAILS ON FORMATTING

The command operates by scanning *formatString* from left to right. Each character from the format string is appended to the result string unless it is a percent sign. If the character is a **%** then it is not copied to the result string. Instead, the characters following the **%** character are treated as a conversion specifier. The conversion specifier controls the conversion of the next successive *arg* to a particular format and the result is appended to the result string in place of the conversion specifier. If there are multiple conversion specifiers in the format string, then each one controls the conversion of one additional *arg*. The **format** command must be given enough *arg*s to meet the needs of all of the conversion specifiers in *formatString*.

Each conversion specifier may contain up to six different parts: an XPG3 position specifier, a set of flags, a minimum field width, a precision, a length modifier, and a conversion character. Any of these fields may be omitted except for the conversion character. The fields that are present must appear in the order given above. The paragraphs below discuss each of these fields in turn.

If the **%** is followed by a decimal number and a **$**, as in "**%2$d**", then the value to convert is not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where 1 corresponds to the first *arg*. If the conversion specifier requires multiple arguments because of ***** characters in the specifier then successive arguments are used, starting with the argument given by the number. This follows the XPG3 conventions for positional specifiers. If there are any positional specifiers in *formatString* then all of the specifiers must be positional.

The second portion of a conversion specifier may contain any of the following flag characters, in any order:

-

Specifies that the converted argument should be left-justified in its field (numbers are normally right-justified with leading spaces if needed).

+

Specifies that a number should always be printed with a sign, even if positive.

space

Specifies that a space should be added to the beginning of the number if the first character isn´t a sign.

0

Specifies that the number should be padded on the left with zeroes instead of spaces.

#

Requests an alternate output form. For **o** and **O** conversions it guarantees that the first digit is always **0**. For **x** or **X** conversions, **0x** or **0X** (respectively) will be added to the beginning of the result unless it is zero. For all floatingpoint conversions (**e**, **E**, **f**, **g**, and **G**) it guarantees that the result always has a decimal point. For **g** and **G** conversions it specifies that trailing zeroes should not be removed.

The third portion of a conversion specifier is a number giving a minimum field width for this conversion. It is typically used to make columns line up in tabular printouts. If the converted argument contains fewer characters than the minimum field width then it will be padded so that it is as wide as the minimum field width. Padding normally occurs by adding extra spaces on the left of the converted argument, but the **0** and **-** flags may be used to specify padding with zeroes on the left or with spaces on the right, respectively. If the minimum field width is specified as ***** rather than a number, then the next argument to the **format** command determines the minimum field width; it must be a numeric string.

The fourth portion of a conversion specifier is a precision, which consists of a period followed by a number. The number is used in different ways for different conversions. For **e**, **E**, and **f** conversions it specifies the number of digits to appear to the right of the decimal point. For **g** and **G** conversions it specifies the total number of digits to appear, including those on both sides of the decimal point (however, trailing zeroes after the decimal point will still be omitted unless the **#** flag has been specified). For integer conversions, it specifies a minimum number of digits to print (leading zeroes will be added if necessary). For **s** conversions it specifies the maximum number of characters to be printed; if the string is longer than this then the trailing characters will be dropped. If the precision is specified with ***** rather than a number then the next argument to the **format** command determines the precision; it must be a numeric string.

The fifth part of a conversion specifier is a length modifier, which must be **h** or **l**. If it is **h** it specifies that the numeric value should be truncated to a 16-bit value before converting. This option is rarely useful. The **l** modifier is ignored.

The last thing in a conversion specifier is an alphabetic character that determines what kind of conversion to perform. The following conversion characters are currently supported:

d

Convert integer to signed decimal string.

u

Convert integer to unsigned decimal string.

i
Convert integer to signed decimal string; the integer may either be in decimal, in octal (with a leading **0**) or in hexadecimal (with a leading **0x**).

o
Convert integer to unsigned octal string.

x or **X**
Convert integer to unsigned hexadecimal string, using digits "0123456789abcdef" for **x** and "0123456789ABCDEF" for **X**).

c
Convert integer to the 8-bit character it represents.

s
No conversion; just insert string.

f
Convert floating-point number to signed decimal string of the form *xx.yyy*, where the number of *y´*s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output.

e or **e**
Convert floating-point number to scientific notation in the form *x.yyye+-zz*, where the number of *y´*s is determined by the precision (default: 6). If the precision is 0 then no decimal point is output. If the **E** form is used then **E** is printed instead of **e**.

g or **G**
If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for **%e** or **%E**. Otherwise convert as for **%f**. Trailing zeroes and a trailing decimal point are omitted.

%
No conversion: just insert **%**.

For the numerical conversions the argument being converted must be an integer or floating-point string; format converts the argument to binary and then converts it back to a string according to the conversion specifier.

DIFFERENCES FROM ANSI SPRINTF

The behavior of the format command is the same as the ANSI C **sprintf** procedure except for the following differences:

[1]
%p and **%n** specifiers are not currently supported.

[2]
For **%c** conversions the argument must be a decimal string, which will then be converted to the corresponding character value.

[3]
The **l** modifier is ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the **l** modifier were present (i.e. type **double** is used for the internal representation). If the **h** modifier is specified then integer values are truncated to **short** before conversion.

See Also:

`open, socket`

Keywords:

`conversion specifier, format, sprintf, string, substitution`

`FSTAT - Extended TCL`

Obtains status info about an open file

Synopsis:

- **fstat** fileId ?item?| ?stat arrayvar?

Description

Obtain status information about an open file.

The following keys are used to identify data items:

- atime - The time of last access.
- ctime - The time of last file status change
- dev - The device containing a directory for the file. This value uniquely identifies the file sys- tem that contains the file.
- gid - The group ID of the file´s group.
- ino - The inode number. This field uniquely identifies the file in a given file system.

- mode - The mode of the file (see the mknod system call).
- mtime - Time when the data in the file was last modified.
- nlink - The number of links to the file.
- size - The file size in bytes.
- tty - If the file is associated with a terminal, then 1 otherwise 0.
- type - The type of the file in symbolic form, which is one of the following values: file, direc- tory, characterSpecial, blockSpecial, fifo, link, or socket.
- uid - The user ID of the file´s owner.

If one of these keys is specified as item, then that data item is returned.

If stat arrayvar is specified, then the information is returned in the array arrrayvar. Each of the above keys indexes an element of the array contain- ing the data.

If only fileId is specified, the command returns the data as a keyed list.

The following values may be returned only if explicitly asked for, it will not be returned with the array or keyed list forms:

- remotehost - If fileId is a TCP/IP socket connection, then a list is returned with the first element being the remote host IP address. If the remote host name can be found, it is returned as the second element of the list. The remote host IP port number is returned as the this element.
- localhost - If fileId is a TCP/IP socket connec- tion, then a list is returned with the first ele- ment being the local host IP address. If the local host name can be found, it is returned as the sec- ond element of the list. The local host IP port number is returned as the this element.

See Also:

`open, socket`

Keywords:

`conversion specifier, format, sprintf, string, substitution`

FUNLOCK - Extended TCL

Remove a locked from a file that was placed with flock

Synopsis:

- **flock** fileId ?start? ?length? ?origin?

Description

Remove a locked from a file that was previously placed with the flock command. The arguments are the same as for the flock command, see that command for more details.

See Also:

open, socket

Keywords:

conversion specifier, format, sprintf, string, substitution

GET_AVG_NC - NeoWebScript

Returns an average daily count.

Synopsis:

- **get_avg_nc** *counter_name [browser]*

Description

Returns the average daily counts assigned to the counter. The average is calculated by dividing the real counts by the number of days since the counter was registered. (NOTE: initial counts are not used to calculate the average.) By default the total value for all browsers is returned. Caller may request a specific browser (all, mozilla, mosaic, lynx, mie, other).

See Also:

open, socket

Keywords:

conversion specifier, format, sprintf, string, substitution

Returns the initial counter value.

Synopsis:

- **get_init_nc** *counter_name [browser]*

Description

Returns the initial counts assigned to the counter when it was registered. By default the total value for all browsers is returned. Caller may request a specific browser (all, mozilla, mosaic, lynx, mie, other).

See Also:

open, socket

Keywords:

conversion specifier, format, sprintf, string, substitution

Returns the current counter value.

Synopsis:

- **get_nc** *counter_name [browser] [val_type]*

Description

Returns the current counter value. By default the overall lifetime count is returned. Caller may request a specific browser (all, mozilla, mosaic, lynx, mie, other). Caller may request lifetime of today´s count only using val_type.

See Also:

open, socket

Keywords:

conversion specifier, format, sprintf, string, substitution

GET_REAL_NC - NeoWebScript

Returns an actual daily count.

Synopsis:

- **get_real_nc** *counter_name [browser]*

Description

Returns the actual counts assigned to the counter (i.e. count-init_count). By default the total value for all browsers is returned. Caller may request a specific browser (all, mozilla, mosaic, lynx, mie, other).

See Also:

open, socket

Keywords:

conversion specifier, format, sprintf, string, substitution

GETS - Safe TCL

Read a line from a channel

Synopsis:

- **gets** *channelID ? varName?*

Description

This command reads the next line from *channelId*, returns everything in the line up to (but not including) the endof-line character(s), and discards the end-of-line character(s). If *varName* is omitted the line is returned as the result of the command. 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 returned.

If end of file occurs while scanning for an end of line, the command returns whatever input is available up to the end of file. If *channelId* is in nonblocking mode and there is not a full line of input available,

the command returns an empty string and does not consume any input. If *varName* is specified and an empty string is returned in *varName* because of end-of-file or because of insufficient data in nonblocking mode, then the return count is -1. Note that if *varName* is not specified then the end-of-file and no-full-line-available cases can produce the same results as if there were an input line consisting only of the end-of-line character(s). The **eof** and **fblocked** commands can be used to distinguish these three cases.

See Also:

`eof, fblocked`

Keywords:

`blocking, channel, end of file, end of line, line, nonblocking, read`

GLOBAL - Safe TCL

Access global variables

Synopsis:

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

Description

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 refer to the global variable by the same name.

See Also:

`eof, fblocked`

Keywords:

`global, procedure, variable`

GM_TIMESTR_822 - NeoWebScript

Returns standard unix time.

Synopsis:

- **gm_timestr_822** *time*

Description

Returns an RFC822 string representing of time which is in standard Unix time, ie. seconds from the start of the epoch, ie. Jan 1, 1970. Currently used only to generate the Expires header for server-cached images.

See Also:

eof, fblocked

Keywords:

global, procedure, variable

HISTORY - Safe TCL

Manipulate the history list

Synopsis:

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

Description

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**?
Adds 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*?
Replaces 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-executes 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*?
Retrieves 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*?
Retrieves 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:

C*ommand*
R*esult*

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.

HISTORY REVISION

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.

C*ommand_Typed* C*ommand_Recorded*

history redo
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.

See Also:

eof, fblocked

Keywords:

event, history, record, revision

HTML - Neowebscript

Write string into the webpage

Synopsis:

- **html** *string [tag]*

Description

write *string* into the webpage being sent to the user. If *tag* is specified, it is emitted in its on-state in front, and its off state behind of, the string.

See Also:

eof, fblocked

Keywords:

event, history, record, revision

ID - Extended TCL

Provides a means of getting, setting, and converting user, group, and process ids

Synopsis:

- **id** options

Description

This command provides a means of getting, setting and converting user, group and process ids. The id command has the following options:

id user ?name?

id userid ?uid? Set the real and effective user ID to name or uid, if the name (or uid) is valid and permis- sions allow it. If the name (or uid) is not specified, the current name (or uid) is returned.

id convert userid uid

id convert user name Convert a user ID number to a user name, or vice versa.

id group ?name?

id groupid ?gid? Set the real and effective group ID to name or gid, if the name (or gid) is valid and permis- sions allow it. If the group name (or gid) is not specified, the current group name (or gid) is returned.

id groups

id groupids Return the current group access list of the process. The option groups returns group names and groupids returns id numbers.

id convert groupid gid

id convert group name Convert a group ID number to a group name, or vice versa.

id effective user

id effective userid Return the effective user name, or effective user ID number, respectively.

id effective group

id effective groupid Return the effective group name, or effective group ID number, respectively. id effective groupids Return all of the groupids the user is a mem- ber of.

id host Return the hostname of the system the program is running on.

id process Return the process ID of the current process.

id process parent Return the process ID of the parent of the current process.

id process group Return the process group ID of the current process.

id process group set Set the process group ID of the current pro- cess to its process ID.

id host Returns the standard host name of the machine the process is executing on.

See Also:

eof, fblocked

Keywords:

event, history, record, revision

IF - Safe TCL

Execute scripts conditionally

Synopsis:

- **if** *expr1* ?**then**? *body1* **elseif** *expr2* ?**then**? *body2* **elseif** ... ?**else**? ?*bodyN*?

Description

The *if* command evaluates *expr1* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then **body2** is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

See Also:

eof, fblocked

Keywords:

boolean, conditional, else, false, if, true

IMAGE_CREATE - NeoWebScript

Returns a file handle which writes to the cache for the given name

Synopsis:

- **image_create** *name ?expire-seconds?*

Description

NOTE: Image generation is experimental in the current release of NeoWebScript.

Returns a file handle which writes to the cache for the given name. Name must be a file name relative to the current directory. The user should close the file when finished with it using the \"close\" command. The old image, if any, is already destroyed when this call returns.

Expire-seconds defines the number of seconds the image may be delivered from the server cache before it becomes invalid.

For further info, see the Gd command set

See Also:

`eof, fblocked`

Keywords:

`boolean, conditional, else, false, if, true`

IMAGE_EXPIRE - NeoWebScript

Allows server caching of current image

Synopsis:

- **image_expire** *seconds*

Description

NOTE: Image generation is experimental in the current release of NeoWebScript.

Turns on server-side caching for the current image being generated, and sets an expiration time on the cache to *seconds*. (0 is a valid value for seconds, and will cause it to always be expired.) Cached files are served out obeying the \"If-modified-since\" header, allowing for client-side caching as well.

For further info, see the Gd command set

See Also:

`eof, fblocked`

Keywords:

boolean, conditional, else, false, if, true

IMAGE_OPEN - NeoWebScript

Returns a file handle for image name

Synopsis:

- **image_open** *name*

Description

NOTE: Image generation is experimental in the current release of NeoWebScript.

Return a file handle for image name. Name refers to a file in the current directory, including the extension (usually .gd). Makes it possible to access previously generated images. If name is empty, then a handle to the previsouly cached image for the current URI is returned. The user should close the file when finished with it using the \"close\" command. The implementation takes special care not to destroy the old copy of the current image until the current image has been generated.

For further info, see the Gd command set

See Also:

eof, fblocked

Keywords:

boolean, conditional, else, false, if, true

IMPORT_KEYVALUE_PAIRS - NeoWebScript

Loads key-value pairs from a string into an array.

Synopsis:

- **import_keyvalue_pairs** *arrayName string*

Description

load the key-value pairs from string into array arrayName, as -element string.

Example: import_keyvalue_pairs foo "-action paint -customer fred"

Will set element action of array foo to paint and element customer to fred.

The normal usage would be to get optional key-value pairs as arguments to a proc, as in:

```
proc sell_item {customer item args} {
import_keyvalue_pairs options $args
.
.
}
```

This will pull an arbitrary number of optional key-value pairs, read into the args list when the proc began execution (because of the special meaning Tcl attaches to this keyword in argument lists), and store the key-value pairs into the options array.

See Also:

`eof, fblocked`

Keywords:

`boolean, conditional, else, false, if, true`

INCLUDE_FILE - NeoWebScript

Copies the contents of a local file onto the page.

Synopsis:

- **include_file** *fileName*

Description

Copy the contents of the specified file into the webpage being generated. File must be in the same directory or somewhere beneath the current directory. If the filename matches the MIME type for a server side include, the server side include is still performed. If it also contains embedded NeoWebScript, the NeoWebScript is executed. If the owner of the current page and the page being included are the same, the code is executed within the same safe interpreter that´s handling the current page. If the file is owned by someone else, it gets its own interpreter for its neoscript.

See Also:

eof, fblocked

Keywords:

boolean, conditional, else, false, if, true

INCLUDE_VIRTUAL - NeoWebScript

Copies contents of virtual files.

Synopsis:

- **include_virtual** *virtualPath*

Description

Copy the contents of the file named by the specified virtual path (rooted from the htdocs directory) into the webpage currently being generated.

Matching MIME types and processing embedded requests proceeds as for *include_file* above.

See Also:

eof, fblocked

Keywords:

boolean, conditional, else, false, if, true

INCR - Safe TCL

Increment the value of the variable

Synopsis:

- **incr** *varName ?increment?*

Description

Increments the value stored in the variable whose name is *varName*. The value of the variable must be an integer. 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.

See Also:

`eof, fblocked`

Keywords:

`add, increment, variable, value`

`INCR_NC - NeoWebScript`

Increments a neo-web counter.

Synopsis:

- **incr_nc** *counter_name [show] [browser] [val_type]*

Description

Increment the indicated named counter. This increments the overall lifetime total, the browser´s lifetime total and the daily browser total. Any value other than "show" in the show argument causes no value to be returned. If "show" is specified, the indicated value is returned by the incr_nc call. The default return value is the overall lifetime total. The caller may request only a specific browser (all, mozilla, mosaic, lynx, mie, other) in the browser argument. The caller may request only today´s count value by specifying the val_type argument (lifetime, today).

On the first incr_nc call of a new day, a new daily-count record is created automatically.

See Also:

`eof, fblocked`

Keywords:

`add, increment, variable, value`

INCR_PAGE_COUNTER - Neowebscript

Increment a counter for this webpage and return count

Synopsis:

- **incr_page_counter**

Description

Increment a counter for this webpage and return the new count. (Automatically creates and zeros the counter if it doesn´t already exist.)

See Also:

eof, fblocked

Keywords:

add, increment, variable, value

INFO - Safe TCL

Return information about the state of the Tcl interpreter

Synopsis:

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

Description

This command provides information about various internals of 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 builtin 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 complete *command*
Returns 1 if *command* is a complete Tcl command in the sense of having no unclosed quotes, braces, brackets or array element names, If the command doesn´t appear to be complete then 0 is returned. This command is typically used in line-oriented input environments to allow users to type in commands that span multiple lines; if the command isn´t complete, the script can delay evaluating it until additional lines have been typed to complete the command.

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 hostname
Returns the name of the computer on which this invocation is being executed.

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 toplevel. 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. This is actually the value of the **tcl_library** variable and may be changed by setting **tcl_library**. See the **tclvars** manual

entry for more information.

info loaded
?*interp*?
Returns a list describing all of the packages that have been loaded into *interp* with the **load** command. Each list element is a sub-list with two elements consisting of the name of the file from which the package was loaded and the name of the package. For statically-loaded packages the file name will be an empty string. *Interp* defaults to the current interpreter.

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
nameofexecutable
Returns the full path name of the binary file from which the application was invoked. If Tcl was unable to identify the file, then an empty string is returned.

info
patchlevel
Returns the value of the global variable **tcl_patchLevel**; see the **tclvars** manual entry for more information.

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
sharedlibextension
Returns the extension used on this platform for the names of files containing shared libraries (for example, **.so** under Solaris). If shared libraries aren´t supported on this platform then an empty string is returned.

info
tclversion
Returns the value of the global variable **tcl_version**; see the **tclvars** manual entry for more information.

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**.

See Also:

`eof, fblocked`

Keywords:

`command, information, interpreter, level, procedure, variable`

INFOX - Extended TCL

Return info about Extended Tcl or the current application

Synopsis:

- **infox** option

Description

Return information about Extended Tcl, or the cur- rent application. The following infox command options are available:

version Return the version number of Extended Tcl. The version number for Extended Tcl is gener- ated by combining the base version of the standard Tcl code with a letter indicating the version of Extended Tcl being used. This is the documentation for version 7.4a.

patchlevel Return the patchlevel for Extended Tcl.

have_fchown Return 1 if the fchown system call is avail- able. This supports the -fileid option on the chown and chgrp commands.

have_fchmod Return 1 if the fchmod system call is avail- able. This supports the -fileid option on the chmod command.

have_flock Return 1 if the flock command defined, 0 if it is not available.

have_fsync Return 1 if the fsync system call is available and the sync command will sync individual files. 0 if it is not available and the sync command will always sync all file buffers.

have_ftruncate Return 1 if the ftruncate or chsize system call is available. If it is, the ftruncate command -fileid option maybe used.

have_msgcats Return 1 if XPG message catalogs are avail- able, 0 if they are not. The catgets is designed to continue to function without mes- sage catalogs, always returning the default string.

have_posix_signals Return 1 if Posix signals are available (block and unblock options available for the signal command). 0 is returned if Posix signals are not available. have_truncate Return 1 if the truncate system call is avail- able. If it is, the ftruncate command may truncate by file path.

have_waitpid Return 1 if the waitpid system call is avail- able and the wait command has full functional- ity. 0 if the wait command has limited func- tionality.

appname Return the symbolic application name of the current application linked with the Extended Tcl library. The C variable tclAppName must be set by the application to return an appli- cation specific value for this variable.

applongname Return a natural language name for the current application. The C variable tclLongAppName must be set by the application to return an application specific value for this variable.

appversion Return the version number for the current application. The C variable tclAppVersion must be set by the application to return an application-specific value for this variable.

apppatchlevel Return the patchlevel for the current applica- tion. The C variable tclAppPatchlevel must be set by the application to return an applica- tion-specific value for this variable.

See Also:

`eof, fblocked`

Keywords:

`command, information, interpreter, level, procedure, variable`

INTERP - Safe TCL

Create and manipulate Tcl interpreters

Synopsis:

- **interp** *option ?arg arg ...?*

Description

This command makes it possible to create one or more new Tcl interpreters that co-exist with the creating interpreter in the same application. The creating interpreter is called the *master* and the new

interpreter is called a *slave*. A master can create any number of slaves, and each slave can itself create additional slaves for which it is master, resulting in a hierarchy of interpreters.

Each interpreter is independent from the others: it has its own name space for commands, procedures, and global variables. A master interpreter may create connections between its slaves and itself using a mechanism called an *alias*. An *alias* is a command in a slave interpreter which, when invoked, causes a command to be invoked in its master interpreter or in another slave interpreter. The only other connections between interpreters are through environment variables (the **env** variable), which are normally shared among all interpreters in the application. Note that the name space for files (such as the names returned by the **open** command) is no longer shared between interpreters. Explicit commands are provided to share files and to transfer references to open files from one interpreter to another.

The **interp** command also provides support for *safe* interpreters. A safe interpreter is a slave whose functions have been greatly restricted, so that it is safe to execute untrusted scripts without fear of them damaging other interpreters or the application´s environment. For example, all IO channel creation commands and subprocess creation commands are removed from safe interpreters. See SAFE INTERPRETERS below for more information on what features are present in a safe interpreter. The alias mechanism can be used for protected communication (analogous to a kernel call) between a slave interpreter and its master.

A qualified interpreter name is a proper Tcl lists containing a subset of its ancestors in the interpreter hierarchy, terminated by the string naming the interpreter in its immediate master. Interpreter names are relative to the interpreter in which they are used. For example, if *a* is a slave of the current interpreter and it has a slave *a1*, which in turn has a slave *a11*, the qualified name of *a11* in *a* is the list *{a1 a11}*.

The **interp** command, described below, accepts qualified interpreter names as arguments; the interpreter in which the command is being evaluated can always be referred to as *{}* (the empty list or string). Note that it is impossible to refer to a master (ancestor) interpreter by name in a slave interpreter except through aliases. Also, there is no global name by which one can refer to the first interpreter created in an application. Both restrictions are motivated by safety concerns.

The **interp** command is used to create, delete, and manipulate slave interpreters. It can have any of several forms, depending on the *option* argument:

interp alias *srcPath srcCmd*
Returns a Tcl list whose elements are the *targetCmd* and *arg*s associated with the alias named *srcCmd* (all of these are the values specified when the alias was created; it is possible that the actual source command in the slave is different from *srcCmd* if it was renamed).

interp alias *srcPath srcCmd* **{}**
Deletes the alias for *srcCmd* in the slave interpreter identified by *srcPath*. *srcCmd* refers to the name under which the alias was created; if the source command has been renamed, the renamed command will be deleted.

interp alias *srcPath srcCmd targetPath targetCmd* ?*arg arg* ...?
This command creates an alias between one slave and another (see the **alias** slave command below for creating aliases between a slave and its master). In this command, either of the slave interpreters may be anywhere in the hierarchy of interpreters under the interpreter invoking the command. *SrcPath* and

srcCmd identify the source of the alias. *SrcPath* is a Tcl list whose elements select a particular interpreter. For example, "**a b**" identifies an interpreter **b**, which is a slave of interpreter **a**, which is a slave of the invoking interpreter. An empty list specifies the interpreter invoking the command. *srcCmd* gives the name of a new command, which will be created in the source interpreter. *TargetPath* and *targetCmd* specify a target interpreter and command, and the *arg* arguments, if any, specify additional arguments to *targetCmd* which are prepended to any arguments specified in the invocation of *srcCmd*. *TargetCmd* may be undefined at the time of this call, or it may already exist; it is not created by this command. The alias arranges for the given target command to be invoked in the target interpreter whenever the given source command is invoked in the source interpreter. See ALIAS INVOCATION below for more details.

interp aliases ?*path*?
This command returns a Tcl list of the names of all the source commands for aliases defined in the interpreter identified by *path*.

interp create ?**-safe**? ?**--**? ?*path*?
Creates a slave interpreter identified by *path* and a new command, called a *slave command*. The name of the slave command is the last component of *path*. The new slave interpreter and the slave command are created in the interpreter identified by the path obtained by removing the last component from *path*. For example, if *path is "a* **b c**" then a new slave interpreter and slave command named "**c**" are created in the interpreter identified by the path "**a b**". The slave command may be used to manipulate the new interpreter as described below. If *path* is omitted, Tcl creates a unique name of the form **interpx**, where *x* is an integer, and uses it for the interpreter and the slave command. If the **-safe** switch is specified (or if the master interpreter is a safe interpreter), the new slave interpreter will be created as a safe interpreter with limited functionality; otherwise the slave will include the full set of Tcl built-in commands and variables. The **--** switch can be used to mark the end of switches; it may be needed if *path* is an unusual value such as **-safe**. The result of the command is the name of the new interpreter. The name of a slave interpreter must be unique among all the slaves for its master; an error occurs if a slave interpreter by the given name already exists in this master.

interp delete ?*path ...?*
Deletes zero or more interpreters given by the optional *path* arguments, and for each interpreter, it also deletes its slaves. The command also deletes the slave command for each interpreter deleted. For each *path* argument, if no interpreter by that name exists, the command raises an error.

interp eval *path arg* ?*arg* ...?
This command concatenates all of the *arg* arguments in the same fashion as the **concat** command, then evaluates the resulting string as a Tcl script in the slave interpreter identified by *path*. The result of this evaluation (including error information such as the **errorInfo** and **errorCode** variables, if an error occurs) is returned to the invoking interpreter.

interp exists *path*

Returns
> **1** if a slave interpreter by the specified *path* exists in this master, **0** otherwise. If *path* is omitted, the invoking interpreter is used.

interp issafe ?*path*?

Returns **1** if the interpreter identified by the specified *path* is safe, **0** otherwise.

interp share *srcPath channelId destPath* Causes the IO channel identified by *channelId* to become shared between the interpreter identified by *srcPath* and the interpreter identified by *destPath*. Both interpreters have the same permissions on the IO channel. Both interpreters must close it to close the underlying IO channel; IO channels accessible in an interpreter are automatically closed when an interpreter is destroyed.

interp slaves ?*path*?
Returns a Tcl list of the names of all the slave interpreters associated with the interpreter identified by *path*. If *path* is omitted, the invoking interpreter is used.

interp target *path alias*
Returns a Tcl list describing the target interpreter for an alias. The alias is specified with an interpreter path and source command name, just as in **interp alias** above. The name of the target interpreter is returned as an interpreter path, relative to the invoking interpreter. If the target interpreter for the alias is the invoking interpreter then an empty list is returned. If the target interpreter for the alias is not the invoking interpreter or one of its descendants then an error is generated. The target command does not have to be defined at the time of this invocation.

interp transfer *srcPath channelId destPath* Causes the IO channel identified by *channelId* to become available in the interpreter identified by *destPath* and unavailable in the interpreter identified by *srcPath*.

SLAVE COMMAND

For each slave interpreter created with the **interp** command, a new Tcl command is created in the master interpreter with the same name as the new interpreter. This command may be used to invoke various operations on the interpreter. It has the following general form: *slave command* ?*arg arg* ...?
Slave is the name of the interpreter, and *command* and the *arg*s determine the exact behavior of the command. The valid forms of this command are:

slave **aliases**
Returns a Tcl list whose elements are the names of all the aliases in *slave*. The names returned are the *srcCmd* values used when the aliases were created (which may not be the same as the current names of the commands, if they have been renamed).

slave **alias** *srcCmd*
Returns a Tcl list whose elements are the *targetCmd* and *arg*s associated with the alias named *srcCmd* (all of these are the values specified when the alias was created; it is possible that the actual source command in the slave is different from *srcCmd* if it was renamed).

slave **alias** *srcCmd* **{}**
Deletes the alias for *srcCmd* in the slave interpreter. *srcCmd* refers to the name under which the alias was created; if the source command has been renamed, the renamed command will be deleted.

slave **alias** *srcCmd targetCmd* ?*arg* ..?
Creates an alias such that whenever *srcCmd* is invoked in *slave*, *targetCmd* is invoked in the master. The

arg arguments will be passed to *targetCmd* as additional arguments, prepended before any arguments passed in the invocation of *srcCmd*. See ALIAS INVOCATION below for details.

slave **eval** *arg* ?*arg* ..?
This command concatenates all of the *arg* arguments in the same fashion as the **concat** command, then evaluates the resulting string as a Tcl script in *slave*. The result of this evaluation (including error information such as the **errorInfo** and **errorCode** variables, if an error occurs) is returned to the invoking interpreter.

slave **issafe**
Returns **1** if the slave interpreter is safe, **0** otherwise.

ALIAS INVOCATION

The alias mechanism has been carefully designed so that it can be used safely when an untrusted script is executing in a safe slave and the target of the alias is a trusted master. The most important thing in guaranteeing safety is to ensure that information passed from the slave to the master is never evaluated or substituted in the master; if this were to occur, it would enable an evil script in the slave to invoke arbitrary functions in the master, which would compromise security.

When the source for an alias is invoked in the slave interpreter, the usual Tcl substitutions are performed when parsing that command. These substitutions are carried out in the source interpreter just as they would be for any other command invoked in that interpreter. The command procedure for the source command takes its arguments and merges them with the *targetCmd* and *arg*s for the alias to create a new array of arguments. If the words of *srcCmd* were "*srcCmd arg1 arg2 ... argN*", the new set of words will be "*targetCmd arg arg ... arg arg1 arg2 ... argN*", where *targetCmd* and *arg*s are the values supplied when the alias was created. *TargetCmd* is then used to locate a command procedure in the target interpreter, and that command procedure is invoked with the new set of arguments. An error occurs if there is no command named *targetCmd* in the target interpreter. No additional substitutions are performed on the words: the target command procedure is invoked directly, without going through the normal Tcl evaluation mechanism. Substitutions are thus performed on each word exactly once: *targetCmd* and *args* were substituted when parsing the command that created the alias, and *arg1 - argN* are substituted when the alias´s source command is parsed in the source interpreter.

When writing the *targetCmd*s for aliases in safe interpreters, it is very important that the arguments to that command never be evaluated or substituted, since this would provide an escape mechanism whereby the slave interpreter could execute arbitrary code in the master. This in turn would compromise the security of the system.

SAFE INTERPRETERS

A safe interpreter is one with restricted functionality, so that is safe to execute an arbitrary script from your worst enemy without fear of that script damaging the enclosing application or the rest of your computing environment. In order to make an interpreter safe, certain commands and variables are removed from the interpreter. For example, commands to create files on disk are removed, and the **exec** command is removed, since it could be used to cause damage through subprocesses. Limited access to these facilities can be provided, by creating aliases to the master interpreter which check their arguments

carefully and provide restricted access to a safe subset of facilities. For example, file creation might be allowed in a particular subdirectory and subprocess invocation might be allowed for a carefully selected and fixed set of programs.

A safe interpreter is created by specifying the **-safe** switch to the **interp create** command. Furthermore, any slave created by a safe interpreter will also be safe.

A safe interpreter is created with exactly the following set of built-in commands:

append array break case catch clock close concat continue eof error eval

expr
fblocked flush for foreach format gets global history if incr info interp join lappend lindex

list
llength lrange lreplace pid proc puts read regexp regsub rename return scan set seek split string switch tell trace

All commands not on this list are removed from the interpreter by the **interp create** command. Of course, the missing commands can be recreated later as Tcl procedures or aliases.

In addition, the **env** variable is not present in a safe interpreter, so it cannot share environment variables with other interpreters. The **env** variable poses a security risk, because users can store sensitive information in an environment variable. For example, the PGP manual recommends storing the PGP private key protection password in the environment variable *PGPPASS*. Making this variable available to untrusted code executing in a safe interpreter would incur a security risk.

If extensions are loaded into a safe interpreter, they may also restrict their own functionality to eliminate unsafe commands. The management of extensions for safety will be explained in the manual entries for the **package** and **load** Tcl commands.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`alias, master interpreter, safe interpreter, slave interpreter`

JOIN - Safe TCL

Create a string by joining together list elements

Synopsis:

- **join** *list ?joinString?*

Description

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.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, join, list, separator`

`KEYLDEL - Extended TCL`

Delete field specified by key from the keyed list in listvar

Synopsis:

- **keyldel** listvar key

Description

Delete the field specified by key from the keyed list in the variable listvar. This removes both the key and the value from the keyed list.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, join, list, separator`

`KEYLGET - Extended TCL`

Return value associated with key from the keyed list in listvar

Synopsis:

- **keylget** listvar ?key? ?retvar | {}?

Description

Return the value associated with key from the keyed list in the variable listvar. If retvar is not specified, then the value will be returned as the result of the command. In this case, if key is not found in the list, an error will result.

If retvar is specified and key is in the list, then the value is returned in the variable retvar and the command returns 1 if the key was present within the list. If key isn´t in the list, the command will return 0, and retvar will be left unchanged. If {} is specified for retvar, the value is not returned, allowing the Tcl programmer to determine if a key is present in a keyed list without setting a variable as a side-effect.

If key is omitted, then a list of all the keys in the keyed list is returned.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, join, list, separator

KEYLKEYS - Extended TCL

Return the list of the keys in the keyed list in listvar

Synopsis:

- **keylkeys** listvar ?key?

Description

Return the a list of the keyes in the keyed list in the variable listvar. If keys is specified, then it is the name of a key field who´s subfield keys are to be retrieve.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, join, list, separator`

`KEYLSET - Extended TCL`

Set the value associated with key in the keyed list contained in the variable listvar

Synopsis:

- **keylset** listvar key value ?key2 value2...?

Description

Set the value associated with key, in the keyed list contained in the variable listvar, to value. If listvar does not exists, it is created. If key is not currently in the list, it will be added. If it already exists, value replaces the existing value. Multiple keywords and values may be speci- fied, if desired.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, join, list, separator`

`LAPPEND - Safe TCL`

Append list elements onto a variable

Synopsis:

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

Description

This command treats the variable given by *varName* as a list and appends 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.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`append, element, list, variable`

`LASSIGN - Extended TCL`

Assign successive elements of a list to specified variables

Synopsis:

- **lassign** list var ?var...?

Description

Assign successive elements of a list to specified variables. If there are more variable names than fields, the remaining variables are set to the empty string. If there are more elements than variables, a list of the unassigned elements is returned.

For example,

lassign {dave 100 200 {Dave Foo}} name uid gid longName

Assigns name to "dave", uid to "100", gid to "200", and longName to "Dave Foo".

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`append, element, list, variable`

LEMPTY - Extended TCL

Determines whether a list is empty or not.

Synopsis:

- **lempty** *list*

Description

Determine if the specified list is empty. If empty, 1 is returned, otherwise, 0 is returned. This command is an alternative to comparing a list to an empty string.

See Also:

load, package, Tcl_CreateSlave

Keywords:

append, element, list, variable

LINDEX - Safe TCL

Retrieve an element from a list

Synopsis:

- **lindex** *list index*

Description

This command 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. If *index* has the value **end**, it refers to the last element in the list.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, index, list`

`LINSERT - Safe TCL`

Insert elements into a list

Synopsis:

- **linsert** *list index element* 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* has the value **end**, or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

Description

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* has the value **end**, or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, insert, list`

`LIST - Safe TCL`

Create a list

Synopsis:

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

Description

This command returns a list comprised of all the *arg*s, or an empty string if no *arg*s are specified. 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}**

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list`

LIST_DATA_FILES - Neowebscript

Lists datafiles

Synopsis:

- **list_data_files**

Description

Returns a list of all of the user´s datafiles.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list

LLENGTH - Safe TCL

Count the number of elements in a list

Synopsis:

- **llength** *list*

Description

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

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list, length

LMATCH - Extended TCL

Searches a list for matching patterns.

Synopsis:

- **lmatch** *?mode? list pattern*

Description

Search the elements of list, returning a list of all elements matching pattern. If none match, an empty list is returned.

The mode argument indicates how the elements of the list are to be matched against pattern and it must have one of the following values:

-exact The list element must contain exactly the same string as pattern.

-glob Pattern is a glob-style pattern which is matched against each list element using the same rules as the string match command.

-regexp Pattern is treated as a regular expression and matched against each list element using the same rules as the regexp command.

If mode is omitted then it defaults to -glob.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, length`

`LOAD_COOKIES - Neowebscript`

Loads the cookie into global array

Synopsis:

- **load_cookies** *[cookieArrayName]*

Description

Load the HTTP persistent data cookies into the global array **cookies** or the array named by *cookieArrayName*.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, length`

Loads and evaluates code.

Synopsis:

- **load_file** *fileName*

Description

Load and evaluate the *NeoWebScript* code contained in *fileName*. This is to support libraries of procs and shared code.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list, length

LOAD_MIME_RESPONSE - Neowebscript

In multipart/form-data response, master interpreter creates a temporary directory to receive MIME data

Synopsis:

- **load_mime_response** *[responseVarName] [uploadedDataVar]*

Description

In the event of a multipart/form-data response, the master interpreter creates a temporary directory to receive and decode the MIME data. The posted-to file must use **load_mime_response** to access the MIME data. *responseVarName* defaults to "response", and *uploadedDataVar* defaults to "filesUploaded". *responseVarName* is as in **load_response**.

uploadedDataVar is the name of an array, indexed by field name, in which to place information about the files uploaded. The information stored in the array is a list as follows:

- the temporary name of the file on disk

- the name of the requested file (in the browser) (OS dependent, and may or may not be the full pathname).
- content-length
- content-type

The return value of **load_mime_response** is a list of field names (or else an empty string) which may be used as arguments to **open_mime_file**.

Note that an additional program, **webunpack** must be compiled to support file uploads. Webunpack is derived munpack, (C) Copyright 1993,1994 by Carnegie Mellon University, All Rights Reserved, ftp.andrew.cmu.edu:pub/mpack/

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, length`

`LOAD_RESPONSE - Neowebscript`

Load the CGI response into the global array

Synopsis:

- **load_response** *[responseArrayName] [multiList]*

Description

Load the CGI response into the global array **response** or the array named by *responseArrayName*

If the same *name* is repeated in the post or query, **response**(*__name*) is set, and **response**(*name*) is converted to a list. Use `[info exists response(__name)]` to check if the array member is a straight value or a list.

If multiple response is expected (as in <*select multiple*>), use *multiList* to force the array members of the response array to always be a list.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

element, list, length

LOAD_VIRTUAL - NeoWebScript

Loads and evaluates virtual code.

Synopsis:

- **load_virtual** *virtualPath*

Description

Load and evaluate the *NeoWebScript* code specified by the virtual path *virtualPath*. This also supports libraries of procs and shared code.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list, length

LOG_MESSAGE - NeoWebScript

Logs a message to a database.

Synopsis:

- **log_message** *loggingDatabase message*

Description

The *Neoscript* programmer can log messages to a number of self-defined logging databases. When *log_message* is executed, the message is logged along with the server time (in integer-seconds format) and the remote hostname (or IP number, if hostname either was unavailable due to DNS problems or if Apache was compiled with MINIMAL_DNS selected) fetching the page.

If you need something fancier you can always roll your own with *access_data_file*.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, length`

`LOOP - Extended TCL`

A superior looping procedure.

Synopsis:

- **loop** *var first limit ?increment? body*

Description

Loop is a looping command, similar in behavior to the Tcl for statement, except that the loop state- ment achieves substantially higher performance and is easier to code when the beginning and ending values of a loop are known, and the loop variable is to be incremented by a known, fixed amount every time through the loop.

The var argument is the name of a Tcl variable that will contain the loop index. The loop index is set to the value specified by first. The Tcl interpreter is invoked upon body zero or more times, where var is incremented by increment every time through the loop, or by one if increment is not specified. Increment can be negative in which case the loop will count downwards.

When var reaches limit, the loop terminates without a subsequent execution of body. For instance, if the original loop parameters would cause loop to terminate, say first was one, limit was zero and increment was not specified or was non-negative, body is not executed at all and loop returns.

The first, limit and increment are integer expres- sions. They are only evaulated once at the begin- ning of the loop.

If a continue command is invoked within body then any remaining commands in the current execution of body are skipped, as in the for command. If a break command is invoked within body then the loop command will return immediately. Loop returns an empty string.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, length`

`LRANGE - Safe TCL`

Return one or more adjacent elements from a list

Synopsis:

- **lrange** *list first last*

Description

List must be a valid Tcl list. This command will return a new list consisting of elements *first* through *last*, inclusive. *First* or *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]*"

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, range, sublist`

`LREPLACE - Safe TCL`

Replace elements in a list with new elements

Synopsis:

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

Description

Lreplace 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 (0 refers to the first element). 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. If *last* is less than *first* then no elements are deleted; the new elements are simply inserted before *first*. *First* or *last* may be **end** (or any abbreviation of it) to refer to the last element of the list. 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.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, replace`

`LSEARCH - Safe TCL`

See if a list contains a particular element

Synopsis:

- **lsearch** *?mode? list pattern*

Description

This command searches 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**. The *mode* argument indicates how the elements of the list are to be matched against *pattern* and it must have one of the following values:

-exact
The list element must contain exactly the same string as *pattern*.

-glob
Pattern is a glob-style pattern which is matched against each list element using the same rules as the **string match** command.

-regexp
Pattern is treated as a regular expression and matched against each list element using the same rules as the **regexp** command.

If *mode* is omitted then it defaults to **-glob**.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`list, match, pattern, regular expression, search, string`

`LSORT - Safe TCL`

Sort the elements of a list

Synopsis:

- **lsort** *?switches? list*

Description

This command sorts the elements of *list*, returning a new list in sorted order. By default ASCII sorting is used with the result returned in increasing order. However, any of the following switches may be specified before *list* to control the sorting process (unique abbreviations are accepted):

-ascii
Use string comparison with ASCII collation order. This is the default.

-integer
Convert list elements to integers and use integer comparison.

-real
Convert list elements to floatingpoint values and use floating comparison.

-command *command*
Use *command* as a comparison command. To compare two elements, evaluate a Tcl script consisting of *command* with the two elements appended as additional arguments. The script should return an integer less than, equal to, or greater than zero if the first element is to be considered less than, equal to, or greater than the second, respectively.

-increasing
Sort the list in increasing order ("smallest" items first). This is the default.

-decreasing
Sort the list in decreasing order ("largest" items first).

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`LVARCAT - Extended TCL`

Treats each string argument as a list and concatenates them into a single list.

Synopsis:

- **lvarcat** *var string ?string...?*

Description

This command treats each string argument as a list and concatenates them to the end of the contents of var, forming a single list. The list is stored back into var and also returned as the result. If var does not exist, it is created.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`LVARPOP - Extended TCL`

Deletes an expression from a list.

Synopsis:

- **lvarpop** *var ?indexExpr? ?string?*

Description

The lvarpop command pops (deletes) the element indexed by the expression indexExpr from the list contained in the variable var. If index is omit- ted, then 0 is assumed. If string, is specified, then the deleted element is replaced by string. The replaced or deleted element is returned. Thus "lvarpop argv 0" returns the first element of argv, setting argv to contain the remainder of the string.

If the expression indexExpr starts with the string end, then end is replaced with the index of the last element in the list. If the expression starts with len, then len is replaced with the length of the list.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`LVARPUSH - Extended TCL`

Inserts a string into a list.

Synopsis:

- **lvarpush** *var string ?indexExpr?*

Description

The lvarpush command pushes (inserts) string as an element in the list contained in the variable var. The element is inserted before position indexExpr in the list. If index is omitted, then 0 is assumed. If var does not exists, it is created.

If the expression indexExpr starts with the string end, then end is replaced with the index of the last element in the list. If the expression starts with len, then len is replaced with the length of the list. Note the a value of end means insert the string before the last element.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list, order, sort

MAKE_NC_REGISTRY - NeoWebScript

Increments a neo-web counter.

Synopsis:

- **make_nc_registry** *[max_counters] [comments]*

Description

Executed one time only, to create the page owner´s Named Counter Registry file (NCRegistry). Defaults to 20 counters in the registry. Maximum of 200 counters allowed.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list, order, sort

MAX - Extended TCL

Returns the higher value argument.

Synopsis:

- **max** *num1 ?..numN?*
- *expr* **max***(num1,num2)*

Description

Returns the argument that has the highest numeric value. Each argument may be any integer or floating point value.

This functionality is also available as a math function max in the Tcl expr command.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`MD5 - NeoWebScript`

Returns an md5 digest of string.

Synopsis:

- **md5** *string*

Description

Returns an md5 digest of string.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`MIN - Extended TCL`

Returns the lesser of two values.

Synopsis:

- **min** *num1 ?..numN?*
- *expr* **min**(*num1, num2*)

Description

Returns the argument that has the lowest numeric value. Each argument may be any integer or float- ing point value.

This functionality is also available as a math function min in the Tcl expr command.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`NEO_CLOCK_TO_RFC850_GMT - NeoWebScript`

Returns the RFC-850 time in GMT.

Synopsis:

- **neo_clock_to_rfc850_gmt** *clock*

Description

Returns the passed integer-since-1970 time formatted according to RFC-850, with the additional proviso that it be specified in GMT timezone only. (Because that´s the way Netscape wanted it.)

For example, `neo_clock_to_rfc850_gmt 84236782` should produce "Fri, 01-Sep-72 18:06:22 GMT".

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`NEO_MAKE_COOKIE - Neowebscript`

Return HTML to create a cookie

Synopsis:

- **neo_make_cookie** *cookieName cookieValue [-days expireInDays] [-hours expireInHours] [-minutes expireInMinures] [-path urlPathCookieAppliesTo] [-domainCookieAppliesTo] [-secure 1|0]*

Description

Return HTML to create a cookie which, if emitted to a browser that is cookie-capable, will cause the cookie to be included subject to defined restrictions.

Example: `neo_make_cookie email karl@neosoft.com -days 30 -path /myApp`

*Create a cookie named **email** containing **karl@neosoft.com** that will be included in all HTTP requests from the browser we´re responding to, for a period of 30 days, when the requests are underneath/**myApp** on this server, and the browser is cookie-enabled.*

For more information, see Persistent Client State HTTP Cookies at Netscape, and also Which Browsers Support Cookies at Digital.

This only works with Netscape if it´s emitted before any other HTML. It uses <meta http-equiv> to set the cookie, and won´t work with browsers that don´t support this.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

NEOTRACK - NeoWebScript

Records a visitor´s path through a site.

Synopsis:

- **neotrack** [timeout] [filename]

Description

neotrack records information that may be used to determine a visitor´s path through a site. NeoTrack uses IP number to distinguish between users. timeout is an optional specification of the number of minutes that the visitor can remain on the current page before we will assume that he has left the site. Default is 30 minutes. A dash, "-", may be used in this position if you wish to supply a filename argument but still wish to use the default timeout. filename - is an optional specification of the name of the file in which the visitor´s path information is to be recorded. Default is NeoTrack (.db is implied and should not be specified). The neotrack command returns 1 if the page is served to a new visitor. Otherwise, 0 is returned.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

`NEOTRACK_REPORT - NeoWebScript`

Reports a visitor´s path through a site.

Synopsis:

- **neotrack_report** key[filename]

Description

neotrack_report produces a report table that details a visitors path. For each page in the path, this command reports 1) the Refering address (if available), the current pages URI, 3) the time the current page was entered, 4) the time that the visitor spent on the page (not available for the last page in a path) and 5) the timeout assigned to the current page. You may write your orw proc to produce this or a similar report. This built in proc is provided only for convenience. You must suply the key required to access the visitor that you want reported. The key is formed as time_IP where time is the time (in seconds) that the visitor entered your site and IP is the visitor´s IP number. The filename argument is an optional identification of the tracking data file from which the report is to be produced.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

element, list, order, sort

OPEN_OUTBOUND_MAIL - NeoWebScript

Opens an email message.

Synopsis:

- **open_outbound_mail** *subject [to]*

Description

This opens an email message and returns the filehandle for use with puts, etc, for the contents of the message body. The message always comes from the user name of the owner of the webpage file that´s being interpreted, and the name of the server that did the serving. If to is not specified, the recipient is also set to be the user name of the owner of the webpage file.

See Also:

load, package, Tcl_CreateSlave

Keywords:

element, list, order, sort

OPEN_POST_NEWS - NeoWebScript

Starts a news posting.

Synopsis:

- **open_post_news** *-subject* ***subject*** *-newsgroups* ***grouplist*** *-distribution* ***distribution***

Description

This starts a news posting and returns the filehandle for use with puts, etc, for the contents of the message body. The message always comes "from" the user name of the owner of the webpage file that´s being interpreted, and the name of the server that did the serving.

When done writing the body of the news article, write a single line to the filehandle consisting only of a single period, then close the file.

Note that for this to work you must have a news server within your domain named or aliased as **news**. For example, within ***neosoft.com***, *open_post_news* will try to contact the news server at news.neosoft.com.

This is an experimental capability that obviously needs work.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`element, list, order, sort`

PACKAGE - Safe TCL

Facilites for pachage loading and version control

Synopsis:

- **package forget** *package*
- **package ifneeded** *package version* ?*script*? **package names**
- **package provide** *package* ?*version*?
- **package require** ?**-exact**? *package* ?*version*? **package unknown** ?*command*?
- **package vcompare** *version1 version2*
- **package versions** *package*
- **package vsatisfies** *version1 version2*

Description

This command keeps a simple database of the packages available for use by the current interpreter and how to load them into the interpreter. It supports multiple versions of each package and arranges for the

correct version of a package to be loaded based on what is needed by the application. This command also detects and reports version clashes. Typically, only the **package require** and **package provide** commands are invoked in normal Tcl scripts; the other commands are used primarily by system scripts that maintain the package database.

The behavior of the **package** command is determined by its first argument. The following forms are permitted:

package forget *package*
Removes all information about *package* from this interpreter, including information provided by both **package ifneeded** and **package provide**.

package ifneeded *package version* ?*script*? This command typically appears only in system configuration scripts to set up the package database. It indicates that a particular version of a particular package is available if needed, and that the package can be added to the interpreter by executing *script*. The script is saved in a database for

use by subsequent **package require** commands;

> typically, *script* sets up auto-loading for the commands in the package (or calls **load** and/or **source** directly), then invokes **package provide** to indicate that the package is present. There may be information in the database for several different versions of a single package. If the database already contains information for *package* and *version*, the new *script* replaces the existing one. If the *script* argument is omitted, the current script for version *version* of package *package* is returned, or an empty string if no **package ifneeded** command has been invoked for this *package* and *version*.

package names
Returns a list of the names of all packages in the interpreter for which a version has been provided (via **package provide**) or for which a **package ifneeded** script is available. The order of elements in the list is arbitrary.

package provide *package* ?*version*?
This command is invoked to indicate that version *version* of package *package* is now present in the interpreter. It is typically invoked once as part of an **ifneeded** script, and again by the package itself when it is finally loaded. An error occurs if a different version of *package* has been provided by a previous **package provide** command. If the *version* argument is omitted, then the command returns the version number that is currently provided, or an empty string if no **package provide** command has been invoked for *package* in this interpreter.

package require ?**-exact**? *package* ?*version*? This command is typically invoked by Tcl code that wishes to use a particular version of a particular package. The arguments indicate which package is wanted, and the command ensures that a suitable version of the package is loaded into the interpreter. If the command succeeds, it returns the version number that is loaded; otherwise it generates an error. If both the **-exact** switch and the *version* argument are specified then only the given version is acceptable. If **-exact** is omitted but *version* is specified, then versions later than *version* are also acceptable as long as they have the same major version number as *version*. If both **-exact** and *version* are omitted then any version whatsoever is acceptable. If a version of *package* has already been provided (by invoking the

package provide command), then its version number must satisfy the criteria given by **-exact** and *version* and the command returns immediately. Otherwise, the command searches the database of information provided by previous **package ifneeded** commands to see if an acceptable version of the package is available. If so, the script for the highest acceptable version number is invoked; it must do whatever is necessary to load the package, including calling **package provide** for the package. If the **package ifneeded** database does not contain an acceptable version of the package and a **package unknown** command has been specified for the interpreter then that command is invoked; when it completes, Tcl checks again to see if the package is now provided or if there is a **package ifneeded** script for it. If all of these steps fail to provide an acceptable version of the package, then the command returns an error.

package unknown ?*command*?
This command supplies a "last resort" command to invoke during **package require** if no suitable version of a package can be found in the **package ifneeded** database. If the *command* argument is supplied, it contains the first part of a command; when the command is invoked during a **package require** command, Tcl appends two additional arguments giving the desired package name and version. For example, if *command* is **foo bar** and later the command **package require test 2.4** is invoked, then Tcl will execute the command **foo bar test 2.4** to load the package. If no version number is supplied to the **package require** command, then the version argument for the invoked command will be an empty string. If the **package unknown** command is invoked without a *command* argument, then the current **package unknown** script is returned, or an empty string if there is none. If *command* is specified as an empty string, then the current **package unknown** script is removed, if there is one.

package vcompare *version1 version2*
Compares the two version numbers given by *version1* and *version2*. Returns -1 if *version1* is an earlier version than *version2*, 0 if they are equal, and 1 if *version1* is later than **version2**.

package versions *package*
Returns a list of all the version numbers of *package* for which information has been provided by **package ifneeded** commands.

package vsatisfies *version1 version2*
Returns 1 if scripts written for *version2* will work unchanged with *version1* (i.e. *version1* is equal to or greater than *version2* and they both have the same major version number), 0 otherwise.

VERSION NUMBERS

Version numbers consist of one or more decimal numbers separated by dots, such as 2 or 1.162 or 3.1.13.1. The first number is called the major version number. Larger numbers correspond to later versions of a package, with leftmost numbers having greater significance. For example, version 2.1 is later than 1.3 and version 3.4.6 is later than 3.3.5. Missing fields are equivalent to zeroes: version 1.3 is the same as version 1.3.0 and 1.3.0.0, so it is earlier than 1.3.1 or 1.3.0.2. A later version number is assumed to be upwards compatible with an earlier version number as long as both versions have the same major version number. For example, Tcl scripts written for version 2.3 of a package should work unchanged under versions 2.3.2, 2.4, and 2.5.1. Changes in the major version number signify incompatible changes: if code is written to use version 2.1 of a package, it is not guaranteed to work unmodified with either version 1.7.3 or version 3.1.

PACKAGE INDICES

The recommended way to use packages in Tcl is to invoke **package require** and **package provide** commands in scripts, and use the procedure **pkg_mkIndex** to create package index files. Once you´ve done this, packages will be loaded automatically in response to **package require** commands. See the documentation for **pkg_mkIndex** for details.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`package, version`

`PARRAY - Neowebscript`

Dump global array keys and values into page

Synopsis:

- **parray** *arrayName*

Description

Dump the specified global arrays keys and values into webpage.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`package, version`

`PID - Safe TCL`

Retrieve process id(s)

Synopsis:

- **pid** *fileId?*

Description

If the *fileId* argument is given then it should normally refer to a process pipeline created with the **open** command. In this case the **pid** command will return a list whose elements are the process identifiers of all the processes in the pipeline, in order. The list will be empty if *fileId* refers to an open file that isn´t a process pipeline. If no *fileId* argument is given then **pid** returns the process identifier of the current process. All process identifiers are returned as decimal strings.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`file, pipeline, process identifier`

`PROC - Safe TCL`

Create a Tcl procedure

Synopsis:

- **proc** *name args body*

Description

The **proc** command creates a new Tcl procedure named *name*, replacing any existing command or procedure 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.

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 or the **upvar** command.

The **proc** command returns an empty 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.

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`argument, procedure`

PROFILE - Extended TCL

Collects information on the performance of a Tcl script.

Synopsis:

- **profile** *?-commands? On*
- **profile** *off arrayVar*

Description

This command is used to collect a performance pro- file of a Tcl script. It collects data at the Tcl procedure level. The number of calls to a proce- dure, and the amount of real and CPU time is col- lected. Time is also collected for the global con- text. The procedure data is collected by bucketing it based on the procedure call stack, this allows determination of how much time is spent in a par- ticular procedure in each of it´s calling contexts.

The on option enables profile data collection. If the -commands option is specifed, data on all com-

mands within a procedure is collected as well a procedures. Multiple occurrences of a command within a procedure are not distinguished, but this data may still be useful for analysis.

The off option turns off profiling and moves the data collected to the array arrayVar. The array is address by a list containing the procedure call stack. Element zero is the top of the stack, the procedure that the data is for. The data in each entry is a list consisting of the procedure call count and the real time and CPU time in millisec- onds spent in the procedure (and all procedures it called). The list is in the form {count real cpu}. A Tcl procedure profrep is supplied for reducing the data and producing a report

See Also:

`load, package, Tcl_CreateSlave`

Keywords:

`argument, procedure`

`PUTS - Safe TCL`

Write a channel

Synopsis:

- **puts** ?**-nonewline**? ?*channelId*? *string*

Description

Writes the characters given by *string* to the channel given by *channelId*. *ChannelId* must be a channel identifier such as returned from a previous invocation of **open** or **socket**. It must have been opened for output. If no *channelId* is specified then it defaults to **stdout**. **Puts** normally outputs a newline character after *string*, but this feature may be suppressed by specifying the **-nonewline** switch.

Newline characters in the output are translated by **puts** to platform-specific end-of-line sequences according to the current value of the **-translation** option for the channel (for example, on PCs newlines are normally replaced with carriage-return-linefeed sequences; on Macintoshes newlines are normally replaced with carriage-returns). See the **fconfigure** manual entry for a discussion of end-ofline translations.

Tcl buffers output internally, so characters written with **puts** may not appear immediately on the output file or device; Tcl will normally delay output until the buffer is full or the channel is closed. You can force output to appear immediately with the **flush** command.

When the output buffer fills up, the **puts** command will normally block until all the buffered data has

been accepted for output by the operating system. If *channelId* is in nonblocking mode then the **puts** command will not block even if the operating system cannot accept the data. Instead, Tcl continues to buffer the data and writes it in the background as fast as the underlying file or device can accept it. The application must use the Tcl event loop for nonblocking output to work; otherwise Tcl never finds out that the file or device is ready for more output data. It is possible for an arbitrarily large amount of data to be buffered for a channel in nonblocking mode, which could consume a large amount of memory. To avoid wasting memory, nonblocking I/O should normally be used in an event-driven fashion with the **fileevent** command (don´t invoke **puts** unless you have recently been notified via a file event that the channel is ready for more output data).

See Also:

`fileevent`

Keywords:

`channel, newline, output, write`

`REPLICATE - Extended TCL`

Creates a random integer.

Synopsis:

`random limit | seed ?seedval?`

Description

Generate a pseudorandom integer number greater than or equal to zero and less than limit. If seed is specified, then the command resets the random num- ber generator to a starting point derived from the seedval. This allows one to reproduce pseudorandom number sequences for testing purposes. If seedval is omitted, then the seed is set to a value based on current system state and the current time, pro- viding a reasonably interesting and ever-changing seed.

See Also:

`fileevent`

Keywords:

`channel, newline, output, write`