Expect and TCL mini reference manual

Expect is an extension of the TCL language. This means that you can do everything with expect that you can do with TCL.

1 TCL expressions

First of all you need to know a bit on how the TCL language works and how you can write expressions.

1.1 evaluate and return

If you want to evaluate something and return the value then use the [expression] syntax. This is evaluated before everything else.

```
set a [ 2 + 1 ]
```

This will evaluate 2 + 1 and returns the value 3 which then a is set to.

1.2 Quoting

There are two ways to quote strings. The fist way is to use "string" and the second way is to use {string}. The first will evaluate expressions and the later will not.

```
set a "This number [i]"
```

This will set a to "This number 0" assuming i is 0.

```
Example 2: set a {This number [i]}
```

This will set a to "This number [i]"

1.3 Variable dereferencing

During expression evaluation a '$' followed by a name is interpreted as the value that the variable with that name, before the rest is evaluated.

1.4 Lists

Lists are simply strings separated with whitespaces.

```
set list_of_two_elements "a b"
set list2_of_two_elements {a b}
```

In many situations it is necessary to be able to determine the difference between the list containing "a" and "b" and the list containing the single element "a b". This is done by proceeding double quotes with a backslash before the double quote. Here is a (ugly) list of one element:

```
"\"a b\""
```

A much nicer looking thing is to use braces:

```
{{a b}}
```

This is very similar to other languages and makes it very easy to construct arbitrarily complex lists:

```
{a b {c d} {e {f g {xyz}}} k}
```

1.5 Arrays

Tcl provide a second way to multiple strings together called arrays. This is similar to hash lists in perl. Each string in an array is called an element and has an associated name. The name is given in parentheses following the array name.

```
set uid(0) "root"
set uid(1) "daemon"
```

It can then be referenced by $uid($number)

```
set number 1
set a $uid[$number]
```

It is also possible to have strings (even with whitespace) as the reference name. You need to quote it to make it one argument to set though. This is not specific to arrays.

```
set "uid(foo bar)" recog
```

or

```
set uid({foo bar}) recog
```

Multidimentional arrays can be simulated this way as any string can be used, including the ',' caracter. Arrays can be passed to procedures but not as easy as scalar variables. The upvar command has to be used.

array size	`array size name` Return the number of elements in the array with the name name.
array names	Return the names that refer to elements in the array with the name name.

1.6 Continuation lines

By default commands do not continue after end of line. To allow this a backslash must be presented at the end of the line.

Open braces continueacross lines automatically but **note** that this syntax is incorrect:

```
if { a }
{
 expression
}
```

The reason for this is that you must either add a backslash after the if statement or move the '{' caracter to the same line.

Double quotes also continues at end of line but with the newline replaced with a single whitespace. To really give a newline a backslash must be presented.

```
set oneline "hello
  world"
set twoline "hello\
  world"
```

Oneline will be set to "hello< space >world" and twoline will be set to "hello< newline >< space >< space >world".

1.7 Unknown expressions

The unknown command or procedure is called when another command is executed which is not known to the tcl interpreter. This gives you the opportunity to recover in an intelligent way.

```
proc unknown {args} {
  expr $args
}
```

By default the unknown command tries to find procedure definitions in a library. A library is simply a file that contain procedure definitions. Libraries can explictly be read using source command.

2 Expect

The expect suite actually consist just of a few commands: expect, send, spawn and interact and is an extension to TCL.

Before the expect command can be used a program must be spawned, see expect commands below.

```
expect \
  {match1} {action1} \
  {match2} {action2} \
  {match3} {action3} \
  {match4} {action4}
```

It is possible that no such output is encountered within the time period defined by the timeout (variable name "timeout" in seconds, default to 2). In this case expect stops waiting and continues with the next command in the script.

You can use '*' to match any characters. You can also match using [] syntax and similar.

```
expect "\[a-f0-9]"  ; # strongly preferred as \n and similar is not taken as litteral charactesr \ and n.
expect {a-f0-9}     ; # can also be used
```

A note on \ characters. The pattern matcher translate \x to x but this is done once before the pattern matcher. So you need to type \\n to match n. This is especially important for * and ? characters.

```
expect "*" ;# match anything (*)
expect "\n" ;# match linefeed
expect "\*" ;# match anything
expect "\\*" ;# match * character
expect "\\\*" ;# match * character
expect "\\\\*" ;# match \ followed by anything
expect "\\\\\*" ;# match \ followed by anything
expect "\\\\\\*" ;# match \ followed * character
```

The [character is special to both tcl and the pattern matcher so it is especially messy.

```
proc xy {} { return "n*w" }
expect "[xy]" ;# match n followed by anything followed by w
expect "\[xy]" ;# match x or y
expect "\\[xy]" ;# match n followed by anything followed by w
expect "\\\[xy]" ;# match [xy]
expect "\\\\[xy]" ;# match \ followed by n followed ...
expect "\\\\\[xy]" ;# match sequence of \ and then x or y.
```

The expect command can also take flags. The default flag is -gl for glob pattern matching. It can also take the flag -re for regular expression matching.

```
expect {
  -re "a*" { action_when_a_followed_by_any_a }
  "b*" { action_when_b_followed_by_anything }
  -gl "-re" { action_when_real_dash_r_e }
  eof { action_on_end_of_file }
  timeout { action_on_command_timeout }
  default { action_on_command_timeout_or_eof }
  -re "(abc)*" { action_on_any_number_of_a_b_c }
  -re "-?(0|\[1-9]\[0-9]*)?\\.?\[0-9]*" { action_on_float_or_integer }
}
```

You can also pass the pattern as a variable.

```
set v2pats "pat2 act2"
set v3pats "pat3 act3"
expect pat1 act1 $v2pats $v3pats
```

Observe that the following expect string is wrong as $v2pats and $v3pats is considered as two different arguments. It will try to match $v2pats as a pattern and $v3pats as an action. Instead build up a string and use the -brace flag. You can not use double quotes for that though, unless you use eval and then the -brace flag is not necessary. You also need to protect the pattern and action inside a [list x] or an extra brace if they contain whitespaces.

2.1 Parentheses for feedback

In the previous section, parantheses were used to group subpatterns together. They can also play another role. When a regular expression successfully matches a string each part of the string that matches a paranthensized subpattern is stored in "expect_out(1,string)" the second in "expect_out(2,string)" etc up to "expect_out(9,string)". The entire pattern matched is stored in "expect_out(0,string)". Everthing before the pattern and the pattern matched is stored in "expect_out(buffer)". The last two assignments work in glob pattern too.

```
"a*((ab)*|b)"
```

How is this passed to expect_out? It is quite simple. The entire string withing the first parenthesis is stored in 1 (which means up to end of string). The second fount parenthesis is stored in 2. Look for the left parenthesis to determine where the data is put.

2.2 Flags

-re

-gl

-nocase Ignore case. Do not use -nocase with uppercase characters in the pattern as it will never match.

-notransfer Prevent expect from removing matching characters from the internal buffer. The characters can be matched repeatedly as long as this option is associated with the pattern. It can be abbreviated with "-n" when expect is running interactively.

-brace Expect normally see the first argument as a pattern to match, but sometimes you want to give a list of patterns and actions to expect. You can then use -brace to expand the list before it is handled by expect.

```
expect -brace { pat1 act1 pat2 act2 }
```

This can be useful when building up lists of pattern to match with corresponding actions to do.

-i n Use an alternative spawn_id. This applies to all patterns after the -i flag.

```
expect {
 -i $ftp
 "ftp>" "exit"
 eof "exit"
}
```

You can also expect values from multiple spawned processes at the same time. At most one action can be executed just as expect normally do.

```
expect {
 -i $ftp "ftp> " { ftp_action }
 -i $shell $prompt { shell_action }
}
```

There is an input buffer associated with each spawn id. So any output from ftp is kept separate from thta of the shell in the example above. When output appear on terminal it will be mixed, unless you expect one at a time or set `log_user 0` and then wrote output explictly using `send_user $expect_out(buffer)`. The process that did not match will keep its buffer until matched by next expect command.

When the expect command match something it record the spawn_id to expect_out(spawn_id) so you can know which process that it matched against.

You can also set the spawn_id definition to a list of spawn_ids.

```
expect {
  -i "$host1 $host2 $host3" $prompt {
    an_action $expect_out(spawn_id)
  }
  eof exit
  -i $host1 another_pattern {host1_action}
}
```

This example will do an_action if $prompt is matched agains any of the $hostnr, exit if end of file is matched against same list and do host1_action if matchina another_pattern from host1.

2.3 Special patterns

Note that these patcherns will only match if none of the '-gl', '-re' or '-ex' flags has been used.

eof	Match when end of file is reached.
timeout	Match when the timeout value has been reached.
default	Match when end of file or timeout is reached.
full_buffer	Match when no other pattern match and expect would otherwise throw away part of the input to make room fore more. When this matches, all of the unmatched input is moved to expect_out(buffer).
null	By default null characters are removed from the buffer before expect matching is done. This can be disabled using the remove_nulls command below. If it is disabled null characters will be matched using this special keyword. Null characters can not be matched inside a pattern as expect first look for null characters and then do pattern matching. If a null character is found the characters before it is skilled. These caracters can be found in expect_out(buffer). Since the null character is used internally to represent the end of a string unanchored patterns cannot be matched past a null. This is not a problem since null pattern can always be listed last.

2.4 Expect variables

expect_out	
spawn_id	
user_spawn_id	Predefined variable that refer to standard input and standard output.
error_spawn_id	Predefined variable that refer to standard error.
tty_spawn_id	Predefined variable that refer to controlling termina (i.e. /dev/tty).
any_spawn_id	A special global variable predefined to any spawn_id mentioned earlier in the current expect command.

2.5 Expect commands

The expect commands treat scope a bit differently from other commands. If they are used in procedures they will first look up the local scope. If one of the expect variables (like timeout etc) is not defined there it look up the variable in the global scope. This is not the normal behaviour so you should note that.

When writing a variable only one scope is used. The local scope will be used unless a global directive is used.

This is important to know.

expect_user	Same as expect but expect data from standard input. Same as 'expect -i $tty_spawn_id'.
expect_tty	Same as expect_user but read from terminal device even if standard input is redirected. Same as 'expect -i $tty_spawn_id'.
exp_continue	In some cases you may want to continue the expect matching even if a match has occured. You can then use the command exp_continue to re_execute the expect command but the buffered data up to the current match discarded. When exp_continue action is expecuted the timeout variable is reread. To avoid resetting the timer the flag -continue_timer can be used as a flag to exp_continue.
expect_before	Sometimes it is useful to declare patterns that are used automatically by expect commands. It takes the same arguments as expect do and prepend this list to the normal expect list. The effect of this command for a specific spawn_id remain until next expect_before change something for the specific spawn_id. Running expect after this will apply all expect_before patterns for all specified spawn_id:s to the expect list. Observe! When an expect_before command is used without any explicit spawn_id, the pattern is associated with the current spawn_id rather than the spawn_id used in the expect statement.
expect_after	See expect_before but with the difference that the list is appended to the expect list instead of prepended.
log_file	The default is to log everything to stdout. This can be redirected to file using log_file. If you use the flag -noappend the log file will be truncated before data is written to it. The log command can also be used with the -a flag so it log everything even output that is normally surpressed.
log_user	Log to stdout can also be disabled using log_user 0 and enabled again using log_user 1.
send	Send characters to a spawned command. **Notice** that each line sent is terminated with the '\r' character. This is what a return charater pressed so that is what Expect has to send. Output sent to the user (or stdout) is indeed terminated with the '\n' character. Similarly a program that normally is used by a terminal you will see a '\r\n' sequence. Send can also take the argument '-i n' where n is a spawn_id.
send_user	Same as send but directed to stdout just as puts but with two important differences. The first is that it do not add a newline and the second is that it is also logged to the log file (if specified). If log_user has been used to disable output send_user still send such output. This means that you normally want to use send_user instead of puts in expect scripts. Same as 'send -i $user_spawn_id'.
send_error	Works in a similar to send_user but output is directed to standard error instead of standard output. Same as 'send -i $error_spawn_id'.
send_tty	Works in a similar way to send_user but can not be redirected. Expect is one of the few commands that have the power to redirect communications directly to and/or from /dev/tty. Same as 'send -i $tty_spawn_id'.
spawn	Spawn a command. `spawn telnet localhost` You can now use send and expect on the spawned command. Status if the command was successful or not is stored in the variable $?.
match_max	When a program produce lot of output this can cause a problem because the computer memory is limited. Therefore expect guarantee that the last 2000 characters will be matched. This is the same as a 25 row, 80 column screen. In order to change this guaranteed value you can set it using match_max command. Internally Expect allocates double the value match_max in order to have good performance, so double the value can be matched. To specify the default for all current spawned programs, use the -d flag. This do not change the size for the current spawned process. With no arguments the command return the current value (the default value if -d flag is used).
remove_nulls	Removal of null characters can be enabled (parameter = 1) or disabled (parameter = 0) with this command. The flag -d will set the default just as match_max and is handled similarly.

parity	Normally expect respect parity bits. Such bits can be stripped with this command and works siliarly to remove_nulls and match_max.
interact	Let the user interact with the spawned command.
exp_internal	This command is useful for debugging expect matches. Set to 1 to enable debugging and set to 0 to disable. This command can take the flag -f followed by a filename where the internal logging will be put.

```
exp_internal 0          # no diagnostics
exp_internal 1          # diagnostics to standard error
exp_internal -f file 0  # std out and diagnostics to file
exp_internal -f file 1  # std out and diagnostics to file and standard error
```

2.6 Manage multiple commands

Each spawned command has a spawn_id associated to it. By setting the variable spawn_id you can switch back and forth between different spawned processes. Both the expect and send commands honor this variable. Other commands that honor this are: interact, close, wait, match_max, parity, remove_nulls and some more.

Even exp_continue honor spawn_id so the process can be switched in the middle of an expect statement.

```
foreach $spawn_id $spaws_ids {
  close
  wait
}
```

Close and wait for all spawned commands in the list.

2.7 Terminal modes

The terminal mode can be changed using the stty command similarly as the UNIX stty command do. This is useful with the expect_user command is used.

stty raw	Raw mode enabled. Keystrokes are not interpreted in any way by the terminal driver. Once the terminal is in raw mode, patterns without \n match without the user pressing return key.
stty cooked	Opposit of stty raw.
stty -cooked	Opposit of stty -raw.
stty -raw	Set back to line-oriented mode.
stty -echo	Disable echoing. Useful for example when user is about to enter a password.
stty echo	Enable echoing.
stty rows n	Set the number of rows to n. If no argument the current number of rows is returned.
stty columns n	Set the number of columns to n. If no argument the current number of colums is returned.
stty < /dev/ttyXX	Set the named terminal.
stty cmd < /dev/ttyXX	Any stty command can be applied to other terminals. The terminal settings can also queried this way.

If stty has no arguments it will return system specific output from the UNIX stty command. The output may vary from system to system. The parameters are portable though.

3 TCL commands

set Set a variable to value or a list of values and return the values. If no argument is given then the value of the variable is returned.

```
% set b 1
1
% set b
1
% set xc 1
1
% set b x
x
% set d [set [set b]c]  ; set d to the value of xc
1
```

Replacing the set notation with $ can fail.

```
% set a(1) foo
foo
% set a2 a
a
% puts [set [set a2](1)]
foo
% puts [set $a2(1)]
can't read "a2(1)": variable isn't array
% puts $[set a2](1)
$a(1)
% puts [$[set a2](1)]
invalid command name "$a(1)"
```

You can also set a variable name that contain whitespaces if you quote it properly.

```
% set "a b" 1
1
```

unset Unset a variable, array element or entire array.

```
unset a
unset array(elf)
unset array
```

expr You can valuate expressions using the expr command.

```
expr $count < 0
expr {$count < 0}
```

The above two are equal. Unquoted expressions is not allowed inside expressions to avoid ambigous statements unless they are numbers. So the following is not allowed:

```
expr {$name == Don}
```

Avoid expr for these implicit string operations as expr tries to interpret strings as numbers. Only if they are not numbers they are treated as strings. The "string compare" operation is better for string comparision.

```
% expr {2E5000=="1E5000"}
floating-point value too large to represent
```

exec UNIX commands can be executed using the exec command. The arguments generally follow the /bin/sh-like conventions as open including >, <, |, 'litterl and' and variations on them. Use whitespace before and after redirection symbols.

```
% exec date
Tue Dec 28 16:06:22 MET 2004
% puts "The date is [exec date]"
The date is Tue Dec 28 16:14:14 MET 2004
```

The output is returned so it can be used in variables. Tcl assumes that UNIX commands exit with value 0 if successful. Use catch to test whether a program succeeds or not. Many programs return 0 even if unsuccessful though.

Tilde substituion is performed on on the command but not on the arguments, and no globbing is done at all, so you need to use the glob directive for that.

```
exec rm -f [glob *.o]
```

Beyond the /bin/sh conventions, exec supports special redirection to already open files using a '@' caracter directly after a redirection symbol.

```
set handle [ open "/tmp/foo" "w" ]
exec date >@ $handle
```

Exec also have some other exoteric features.

system Similar to the exec command but do not redirect standard out or standard error.

source Procedures and variables can be stored in other files and read using the source command. As the file is read, the commands are executed.

```
source ~/def.tcl
```

if

```
if {expression} { expression}
elseif {expression} { expression }
else {expression} { expression }
```

switch The switch command is similar to the if command but can take arguments and compare string values to a set of patterns. The first pattern matched will be used. Since switch support several flags '–' must be used as the final flag. If no patterns is matched the default target is used.

The arguments available are:

- -glob turn shell pattern matching on. '?' match any character and '*' match any string.
- -regexp turn regexp pattern matching on.

```
switch -glob -- $count \
  1 { puts "Match one" } \
  2 { puts "Match two" } \
  str1 { puts "String one matched" } \
  a*b  { puts "Start with a and end with b using shell matching"
  default { puts "No match" }
```

while

```
while expression { expression }
```

for

`for start evalexpression next { expression }` The for statement will take four arguments: a start expression, an evaluation expression, a next value expression and finally the expression to loop.

```
for { set a 0 } { a < 2 } { incr a } { puts a }
```

break Break current while or for loop.

continue Continue to next step in the while or for loop.

proc

It is possible to create functions or procedures with the "proc" and "return" commands. The return command can be omitted if it is not needed.

```
proc name { list of variables } { expression }
```

The proc command takes three arguments as specified above. When the name is defined it can then be used as a command:

```
name 0 1 2
```

Variables are local to the procedure unless "global" or "upvar" command is useed, see below.

If the last argument is named args all the remaining arguments are stored in this list. This way a procedure can have variable number of arguments.

global

```
global variablename variablename2 ...
```

Indicate that the variable with name variablename is in the global scope.

upvar

```
upvar $name1 a $name2 b ...
```

Indicate the that procedure can change data in the callers scope for argument name1 using the name a and name2 using name b, etc. Use with care!

```
upvar $name a
set a 1
```

uplevel Similar to the upvar command but used to evaluate commands in the scope of the calling procedure. Use with care!

```
uplevel incr x
```

Increments the variable x in the calling procedure.

return Return a string and return from the procedure.

exit Exit script and return a value.

incr Increment a variable with one (default) or with the amount that the second variable specifies.

trace Evaluate a procedure and trace variable access.

catch

```
catch { expression } optional_result
```

Run a procedure. If no abnormal behavoir was encountered it return 0 and if something got really bad it return 1 (true). The variable "optional_result" will be assigned to the value that the procedure return if everything goes ok. When run unsuccessfully it record the error messave in the variable "optional_result" and the full error information in the variable "errorInfo".

error You can cause errors with the command error that can then be catched with the catch command.

eval You can also evaluate strings in different ways.

```
% set output puts
puts
% $output a
a
```

If you want to evaluate an entire string you must use the eval command as it will treat the entire string as a command otherwise.

```
% set a "puts \"a b c\""
puts "a b c"
% eval $a
a b c
```

sleep This command make expect/tcl sleep for a number of seconds. Any non negative floating number value is acceptable. It may sleep slightly longer than requested.

```
sleep 2.5
```

Sleeps for 2 and a half seconds.

4 List manipulation

llength Return number of elements in the list.

lindex Return value at a specific position in the list, starting on 0.

lrange Return a range of values from the list starting at one position and ending on another position.

foreach Iterating thought a list from front to back is so common that a command foreach has been defined.

```
foreach element $list { expression }
```

Each item in the list is assigned to the variable element.

list To simplify the creation of lists you can use the list command.

```
% lindex [list a b "Hello world"]
Hello world
```

concat Concatinates lists.

```
% concat a {b {c d}}
a b {c d}
```

lappend

linsert First argument is a list, the second argument is a numeric index and the rest are the arguments to insert.

```
% set l {a b c d}
a b c d
% set l [linsert $l 0 new]
new a b c d
% linsert $l 1 foo bar {hello world}
new foo bar {hello world} a b c d
```

lreplace `linsert $list startpos endpos values`

Same as linsert but delete the items before inserting.

lsearch The lsearch is the opposite of lindex. It return the index where it can find the string and -1 if it can not find it.

```
% learch {a b c d e} "b"
1
```

By default it use shell-type pattern matching and if you want to use exact matching use the -exact flag.

```
% lsearch {a b c d e ?} "?"
0
% lsearch -exact {a b c d e ?} "?"
5
```

lsort To sort a list lsort can be used. By default it sport the values in increasing order but flags can be used: -integer, -real, -decreasing.

split Split a string into a list.

```
% split "/etc/passwd" "/"
{} etc passwd
```

join The reverse of split. The first argument is a list to join and the second argument is what to put between the elements.

```
% join {{} etc passwd} "/"
/etc/passwd
```

5 Manipulating strings

Format and scan commands can be used to extract and create string based on low level formats as integers.

format

The first argument is a format string. Most characters in a format string is passed litterally but the ones after a '%' charater is treated differently. The rest of the arguments are used as values for the '%' definitions in the format string.

- '-' means left justify.
- a number indicates the minimum field width.
- 'c' treat as character (caracter 64 is '@' for example if ASCII is used).
- 'd' treat as decimal.
- 's' treat as string.
- 'f' treat as float number.

```
% set x [format "%3c,%4d%-8s==%20f" 64 2 foo 17.2]
  @,   2foo     ==           17.200000
```

scan

The scan command is the opposit of format. It can also use the %*\[characters] syntax to exclude any number of caracters of specified type.

```
% scan $x "%*\[ ]%c,%4d%8s%*\[ =]%f" char dec string float
4
% set char
64
% set dec
2
% set string
foo
% set float
17.2
```

regexp

```
regexp pattern string var0 var1 var2 var3 ...
```

The regexp command patch a regexp pattern to a string and store the values from expect_out(i,string) to vari. Var0 is often ignored. The command return 1 when matched and 0 when not.

```
% regexp (.*)@(.*) $addr ignore user host
1
% set user
root
% set host
localhost
```

regsub

```
regsub pattern string newstring variable
```

The regsub command substitutes values matching pattern in string, replaces it with newstring into the named variable. Matched parenthesized patterns can be referred to inside the newstring as '\1', '\2' and so on up to '\9'. Entire string is referred to as '\0' as usual.

string match

```
string match "*.c" "main.c"
```

Return true if they match.

string first

Return index of first occurance, -1 if none.

string last

Return index of last occurance, -1 if none.

string length {foo}

Return length of string.

string index

```
string index {abcedf} pos
```

Return character at position, starting at 0.

string range

```
string range "abcdef" start end
```

Return string starting at first position and ending on last.

string tolower

Return converted string to lowercase characters.

string toupper

Return converted string to uppercase characters.

string trimleft

Removes characters from beginning of a string. The characters removed are any which appear on the second argument. If no second argument is provided, then whitespace is removed.

string trimright	Same as trimleft but right side is trimmed.

append	Appending is so common that a command is specified for this.
	`% append var "abc"`
	`abc`
	`% append var "cde" "efg"`
	`abccdeefg`
	Append with two arguments is equal to:
	`set var1 "$var1$var2"`

6 Internal information

With the info command you can obtain different kind of information about internal information in tcl.

info exists	Return 1 if the variable name is defined in the current scope.
info locals, info globals, info vars	Return a list of variable in local, global or all variables respectively. You can also specify a expression on what kind of variable names you want. `% info globals mail*`

info depth	Return information about the stack. If no argument it presented the stack depth is returned.
	`info level 0`
	Return command and arguments for the current procedure.
	`info level -1`
	Return command and arguments of the calling procedure.

info script	Return full path to the script being executed.
info command	Return information about a command or procedure.
…	There are more but not documented here.

7 File operations

The open command opens a file and takes two arguments. The first is the file to open and the second is how to open the file. "r" open the file for reading, "a" append (write without truncate) and "w" truncate the file and then open the file for writing.

The open command return a file identifier that can be passed to many other file commands.

```
set handle [open "/etc/passwd" "r"] ; # open a file
close $handle                       ; # close same file
```

The open should be used with the catch command.

```
if { [catch {open "/etc/passwd" "r"} handle ] } {
  puts "Error open /etc/passwd."
}
else {
  while {[gets $handle line] != -1} {
    # do something with line
  }
}
```

You can also open commands. This is specified using the pipe character '|' in the beginning of the string.

```
open "| ls * | grep -v {foo}"
```

When you have done a close on a spawned process you have to use the wait command (without arguments) in order to get rid of zombies. This is not needed if exit is used as it wait for child processes anyway.

flush	`flush $handle` Flush out cached data.

puts	`puts $handle "Hello world"`
	or to avoid adding a extra newline
	`puts -nonewline $handle "Hello world"`

gets	Gets takes a filehandle and an optional variable name to assign with the data read. It return the number of characters read or -1 if eof is reached. `gets $handle line`
read	Read takes two arguments, a file handle and a number of characters to read.

```
set chunk [read $handle 10000]
```

The characters read may be less than specified if there are no more characters left to be read.

eof	Return 1 if end of file has been encountered (by read or gets) otherwise it will return 0.
seek	Analogous to UNIX lseek system call.
tell	Analogous to UNIX tell system call.
glob	Return a list of names matching the arguments using shell pattern matching. `glob *.c *.h` An error occurs if no file is matched, unless you use the "-nocomplain" flag. Most characters understood by the shell can be used for matching including '~', '?', '*', '[]', '{}'.
file dirname	`file dirname $path` Return the directory name for the file specified.
file extension	`file extension $path` Return file extension of the file specified. Empty string if there is no dot character.
file rootname	Return everything (full path) but the extension.
file size	Return number of bytes in file.
file exists	True if path exits.
file isdirectory	True if this is a directory.
file isfile	True if this is a file.
file executable	True if you have permission to execute file.
file owned	Return true if you own file.
file readable	True if you have permission to read from file.
file writable	True if you have permission to write to file.
file atime	Return last accessed time in seconds since January 1, 1970.
file mtime	Return last modified time in seconds since January 1, 1970.
file type	Return the file type: file, directory, characterSpecial, blockSpecial, link or socket.
file readlink	Return the name to which the file points to assuming it is a symbolic link.
file stat	If you want raw information from the inode you can use file stat. It takes to arguments, a path and name where to assign the array of information. `file stat $path arr` Elements written to arr is: atime, ctime, mtime, type, uid, gid, ino, mode, nlink, dev, size. They are all integers except type that is specified as before. If the file is a symbolic link then "file stat" return information to where the link points. If information is wanted about the symbolic link itself, then use "file lstat" instead.

8 Built in variables

pid	Process id of the tcl/expect process.
env	There is an array named env that contain all environment variables.
argv	List of arguments to the expect program. You need the -- option to expect in order to make this work correctly.

9 References

Expect reference

TCL reference