Lenguaje TCL

Santiago Pavón

Que es TCL

Lenguaje de comandos integrable en programas C.

Dos partes para aprender TCL:

1. Sintaxis y reglas de sustitucion.

   Mecanismo muy simple pero lioso al principio.

2. Comandos predefinidos.

   Aprenderlos individualmente cuando se necesiten.

   Estructuras de control son comandos, no sintaxis.

Scripts y Comandos

Script TCL:

- Secuencia de comandos.
- Comandos separados por ; o *newline*.

Comando TCL:

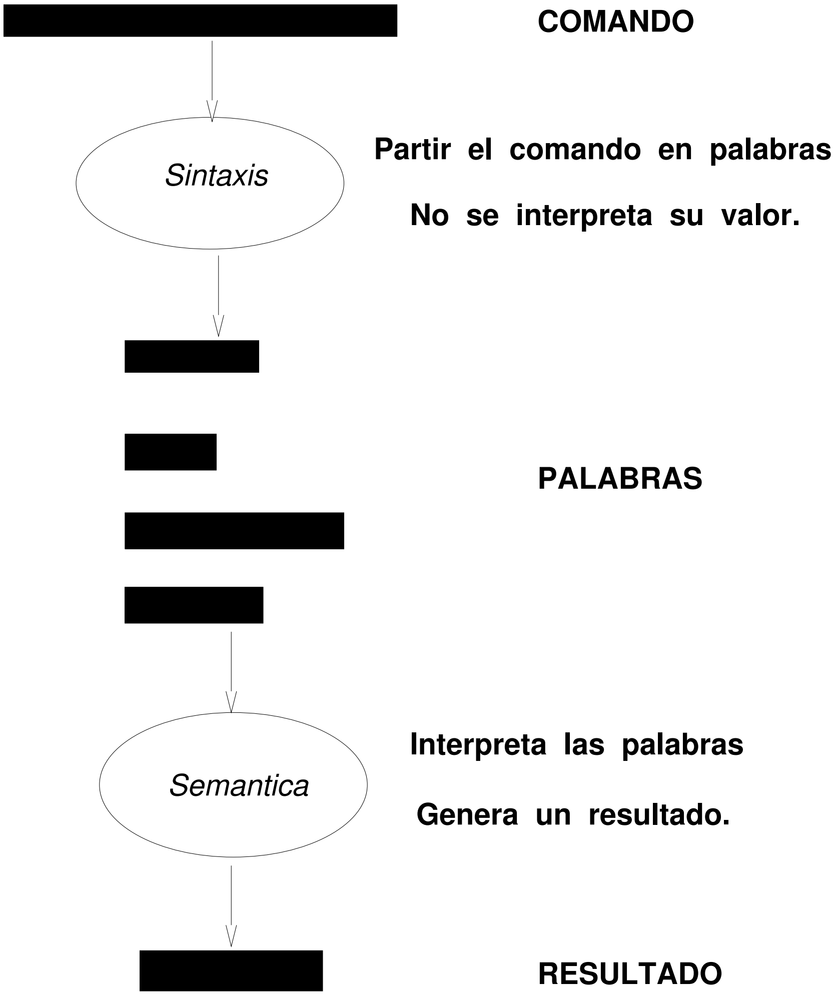
- Una o mas palabras separadas por blancos.
- La primera palabra es el nombre del comando.
  El resto son los argumentos.

Ejemplo:

```
set a 22 ; set b 33
```

```
set a 22
set b 33
```

Ejecucion de un Comando



Argumentos

El parser no asigna significado a los argumentos.

```
set x 4 ; set y x+10
```

y vale `"x+10"`

El significado de los argumentos lo proporciona cada comando.

Diferentes comandos asignan diferente significado.

```
set a 22
expr 24/3.2
eval "set a 122"
button .b -text hola -fg red
string length Abracadabra
```

Sustitucion de Variables

- Sintaxis: `$nombrevariable`
- Nombres variables: `a-z,A-Z,0-9,_`
- Puede ocurrir en cualquier sitio dentro de una palabra.

Ejemplo:

```
set b 66             66
set a b              b
set a $b             66
set a $b+$b+$b       66+66+66
set a $b.3           66.3
set a $b4            no existe b4
set a ${b}4          664
```

Sustitucion de Comandos

- Sintaxis: `[script]`
- Sustituir por el resultado de ejecutar el script.
- Puede ocurrir en cualquier sitio dentro de una palabra.

Ejemplo:

```
set b 8                         8
set a [expr $b+2]               10
set a "b-3 es [expr $b-3]"      b-3 es 5
```

Controlando Estructura de la Palabras

Palabras separadas por blancos o ; excepto:

- Dobles comillas previenen separacion.

```
set a "Buenos dias; adios"
```

- Llaves previenen sepaciones y sustituciones.

```
set a {Laves {} anidadas}
```

- Backslash escapa caracteres especiales

```
set a palabra\ con\ \$\ y\ espacios
```

Las sustituciones no alteran la estructura de las palabras:

```
set a "dos palabras"
set b $a
```

Variables

- Se crean con el comando `set`.

```
set a 4
```

- Se destruyen con el comando `unset`.

```
unset a
```

- Existen arrays: `nombre(indice)`.

```
set b(uno) cosa
set b(dos,cosa) hola
set c $b(dos,$b(uno))
unset b(uno)
unset b
```

Expresiones

- Enteros y reales son tipos C *int* y *double*.
- Mas funciones para strings que en C.
- Soporta sustitucion de variables y comandos en las expressiones.
- Comando `expr` usado para evaluar expresiones.

Ejemplo:

```
set   b 5                5
expr  ($b*4)-3           17
expr  $b<=2              0
expr  {$b * [fac 4]}     120
set   a Banco            Banco
expr  {$a < "Arbol"}     0
```

Listas

- Cero o mas elementos separados por blancos.

```
uno dos tres
```

- Llaves y backslashes para agrupar:

```
a b {c d e} f
una\ palabra dos tres
```

- Comandos sobre listas

```
concat    linsert   lreplace
foreach   list      lsearch
lappend   llength   lsort
lindex    lrange
```

- Ejemplo:

```
lindex {a b {c d e} f} 2
```

$\rightarrow$ `c d e`

- Parser: identico a los comandos, cada elemento de una lista es una palabra.

Estructuras de Control

Apariencia similar a C.

Son comandos que toman como argumentos scripts.

Comando if:

if *cond1* [then] *cuerpo1*
[elseif *cond2* [then] *cuerpo2*
[elseif ...]]
[else] *cuerpon*

```
if {$a != "hola"} then {
   puts $a
} elseif {$b == "adios"} then {
   puts 1000
} else {
   set a $b
}
```

Estructuras de Control (cont.)

Comando while :

while *cond cuerpo*

```
while {![eof $fd]} {
   puts [gets $fd]
}

while {$a < 10} {
   incr a ; puts $a
}

while "$a < 10" {
   incr a ; puts $a
}
```

Comando for :

for *inicio cond reinicio cuerpo*

```
for {set i 1} {$i < 10} {incr i} {
   puts $i
}
```

Estructuras de Control (cont.)

Comando foreach :

foreach *variable lista cuerpo*

```
set lista {unos dos tres}
foreach i $lista {
   puts $i
}

foreach i [exec ls] {
   puts "$i [file readable $i]"
}
```

Estructuras de Control (cont.)

Comando switch :

switch [*opciones*] *string* {
patron cuerpo
[*patron cuerpo* ...]
}
opciones: -exact -glob -regexp –

```
switch -exact $a {
      uno { puts 1111111 }
      dos { puts 2222222 }
  default { puts 0000000 }
}
```

```
switch -regexp $a {
   ^a.*b$  -
   b       { puts 1111111 }
   a*      { puts 2222222 }
   default { puts 3333333 }
}
```

Estructuras de Control (cont.)

`break`

`continue`

`source` *fichero*

`eval` *arg [arg arg ...]*

Procedures

- Comando `proc` para definir procedures.

  `proc nombre argumentos cuerpo`

  proc sub1 x {expr $x-1}

- Procedures se comportan como comando predefinidos.

  `sub1 3` devuelve `2`

- Valores por defecto de los argumentos.

  `proc decr {x {y 1}} {expr $x-$y}`

- Numero variable de argumentos (*args*).

  `proc cosa {a b args} { ... }`

- Retorno: `return [opciones] [valor]`

Ambitos

Existen dos ámbitos:

- global
- local a cada procedimiento (stack llamadas).

Tcl versión 8 incluye namespaces.
Itcl incluye namespaces y clases.

Comandos:

- **`global var1 [var2 ...]`**
  Acceso a las variables del ámbito global.

```
% set x 1
  1
% set y 2
  2
% proc Pinta {} {
    global x
    set x 3
    set y 4
    puts "$x $y"
  }
% Pinta
  3 4
% puts "$x $y"
  3 2
```

Ambitos (cont)

Comandos:

- `upvar [nivel] var1 v1 [var2 v2 ...]`

  Une la variable `var1` del nivel especificado con la variable local `v1`.

  Niveles retrocediendo en el stack: 1 2 3 ...
  Niveles desde el ámbito global: #0 #1 #2 ...
  Nivel por defecto es 1.

  Ejemplo:

  `upvar x y`
  La variable `x` del ámbito anterior se asocia a `y` en el ámbito local.

  Ejemplo: Paso de argumentos por referencia.

```
% proc P x {
      upvar $x y
      incr y
  }
% set z 10
  10
% P z
  11
% puts $z
  11
```

Ambitos (cont)

Comandos:

- `uplevel [nivel] arg [arg arg ...]`

  Concatena los argumentos (igual que `concat`) y ejecuta el resultado en el ámbito especificado por `nivel`.

  Niveles retrocediendo en el stack: 1 2 3 ...
  Niveles desde el ámbito global: #0 #1 #2 ...
  Nivel por defecto es 1.

  `uplevel` no es igual que `eval`.
  Ejemplo:

```
% set saludo "buenos dias"
  buenos dias
% uplevel #0 puts $saludo
  bad file identifier "buenos"
% uplevel #0 puts [list $saludo]
  buenos dias
% uplevel #0 puts \"buenos dias{\"{}}
  buenos dias
```

  Ambito actual: `info level`

Ejecución en background

Comandos:

- `after` *ms*
  Dormirse los milisegundos especificados.
- `after` *ms arg arg arg ...*
  Ejecuta el comando especificado tras *ms*.
  El comando es el (`concat`) de los *arg*.
  Si hay error se llama a `bgerror`.
  Devuelve un *id* de la tarea.
- `after idle` *arg arg arg ...*
  Similar al anterior pero ejecuta el comando en el estado idle.
- `after cancel` *id*
  Cancela la tarea asociada a *id* si existe.
- `after cancel` *arg arg arg ...*
  Cancela la tarea asociada al comando especificado si existe.
  El comando es el (`concat`) de los *arg*.
- `after info [`*id*`]`
  Devuelve la lista de todos los *id* pendientes o el comando y ms o idle asociado al *id* especificado.

Ejemplos: after

- Proceso periodico

```
proc PintaHora {} {
   puts [exec date "+%H:%M:%S"]
   after 1000 PintaHora
}

PintaHora
```

- Espera de un suceso

```
proc BorraFichero nombre {
   if {[file exist $nombre]} {
      file delete $nombre
   } else {
      after 5000 BorraFichero $nombre
   }
}
```

Errores

- Los errores normalmente abortan el comando en curso y la aplicacion presenta un mensaje de error:

```
set n 0
foreach i {1 2 3 4 5} {
   set n [expr {$n + i*i}]
}
```

  devuelve:

```
syntax error in expression "$n + i*i"
```

- La variable global *errorInfo* proporciona la traza de llamadas:

```
set errorInfo
```

  devuelve:

```
syntax error in expression "$n + i*i"
   while executing
"expr {$n + i*i}"
   invoked from within
"set n [expr {$n + i+i}]..."
   ("foreach" body line 2)
   ...
```

Excepciones

- Generacion:

  `return -code` *code* `[-errorInfo` *info*`]`
  `[-errorCode` *code*`] valor`

  `error` *msg* `[`*info*`] [`*code*`]`

  `continue`

  `break`

- Captura: `catch` *comando variable*

  `catch {expr {2 +}} msg`
  devuelve: `1`

  `set msg`
  devuelve: `syntax error in expres.  "2 +"`

- La variable global *errorCode* mantiene informacion (formato maquina) sobre los errores (ej: UNIX errno).

Errores de comandos background

Los comandos en background son los asociados a manejadores de eventos.
Ejemplo: comandos ejecutados con `after`.

Al producirse el error se invoca `bgerror mensaje`.

`bgerror` no es un comando predefinido. Si el usuario no lo ha creado se escribe el mensaje de error por `stderr`.

`bgerror` se ejecutara en un estado idle, restaurando previamente los valores de `errorInfo` y `errorCode`.

Varios errores pueden acumularse antes de ejecutar `bgerror`.
Se invoca una vez por cada error producido, excepto que `bgerror` ejecute `break`, ignorando los errores pendientes.

Strings

- `format` *spec valor valor valor ...*
  ```
  format {%-6.2f %#x} 123 444
  123.00 0x1bc
  ```
- `scan` *valor formato var var var ...*
  ```
  scan abcABC {%[a-z]} result
  scan 65 {%c} result
  scan a123b {a%db} result
  ```

Strings (cont.)

- `string` *operacion string-valor argumentos*
  - `string compare $a $b`
  - `string first $a $b`
  - `string index $a 5`
  - `string last / /usr/local/dos/babia`
    `14`
  - `string length cacacola`
    `8`
  - `string match *.exe $a`
  - `string range abcd 1 end`
    `bcd`
  - `string tolower aBcDeF`
    `abcdef`
  - `string toupper aBcDeF`
    `ABCDEF`
  - `string trim $a`
  - `string trimleft $a abc`
  - `string trimright $a`
  - `string wordend $a 4`
  - `string wordstart $a 4`

Expresiones Regulares

`.`	cualquier caracter
`*`	0 o mas
`+`	1 o mas
`?`	0 o 1
`()`	agrupa subpatron
`|`	alternativa

`[]`
`[a-z]`
`[^ ]`
	conjunto de caracteres
	rango de caracteres
	complementario

`^`	principio string
`$`	final de string

Varios casamientos:

1. Casar con la primera ocurrencia
2. Casar con lo mayor posible

- `regexp` *<flags> patron valor [v v1 v2 ...]*
  v_i asociada a subpatron i-esimo (entre ()).
  `regexp {([^:]*):} $env(DISPLAY) match host`
- `regsub` *<flags> patron valor subspec var*
  `regsub {([^\.]*)\.c} f.c {cc -c & -o \1.o} c`
  → `cc -c f.c -o f.o`
  `\1 .. \9` referencian subpatrones.
  `&` referencia todo el string que casa.

Ficheros

Comandos:

- `cd` [*directorio*]
- `close` *stream*
- `eof` *stream*
- `flush` *stream*
- `gets` [*stream*] [*variable*]
- `glob` [*flags*] *patron* [*patron*] ...
- `open` *file_o_pipe* [*acceso*] [*permisos*]
- `puts` [-nonewline] [*stream*] *cadena*
- `pwd`
- `read` [-nonewline] [*stream*] *num_bytes*
- `seek` *stream* *offset* [*origen*]
- `tell` *stream*

Ficheros (cont.)

comando `file`:

- `file atime` *nombre*
- `file dirname` *nombre*
- `file executable` *nombre*
- `file exists` *nombre*
- `file extension` *nombre*
- `file isdirectory` *nombre*
- `file isfile` *nombre*
- `file lstat` *nombre var*
- `file mtime` *nombre*
- `file owned` *nombre*
- `file readable` *nombre*
- `file readlink` *nombre*
- `file size` *nombre*
- `file rootname` *nombre*
- `file stat` *nombre var*
- `file tail` *nombre*
- `file type` *nombre*
- `file writable` *nombre*

Manejadores de eventos para Canales

Sintaxis:

fileevent *canal* **readable** [*script*]
fileevent *canal* **writable** [*script*]

Canales creados con `open` o `socket`.

Crear manejadores de eventos que realicen las operaciones de lectura y escritura cuando los canales tengan datos que leer o pueda escribirse en ellos, evitando bloqueos en la aplicación.

Como máximo un manejadore para leer y uno para escribir.

Si no se especifica *script* devuelve el *script* en uso.

Para cambiar el script reinvocar `fileevent`.

El *script* se ejecuta en el ámbito global.

Los errores se procesan por `bgerror` y se destuye el manejador.

Manejadores de eventos (cont)

Evento de lectura se dispara si:

- datos disponibles
- excepto última operación sea `gets` y no haya una línea completa en el buffer de entrada.
- error en fichero o dispositivo.

Evento de escritura se dispara si:

- puede aceptarse un byte sin bloquear.
- error en fichero o dispositivo.

Conviene trabajar en modo no bloqueante.

Puede bloquearse si se lee mas de lo disponible, o se escribe mas de lo aceptable.

```
fconfigure canal -blocking 0
```

Sockets

Protocolo disponible TCP.

- CLIENTES:

  `socket [opciones] host port`

  Opciones:

  - `-myaddr addr`
  - `-myport port`
  - `-async`

- SERVIDORES:

  `socket -server comando [opciones] port`

  Opciones:

  - `-myaddr addr`

  Por cada conexión aceptada se invoca:
  `comando channel ipcliente portcliente`

- Opciones de configuración específicas:

  `fconfigure` *channel* `-sockname`
  `fconfigure` *channel* `-peername`
  Devuelven: `addr host port`

Ejemplo Sockets

```
proc MakeServer {port} {
   set server [socket -server AcceptConnection $port]
   fconfigure $server -buffering none -blocking 0
   set port [lindex [fconfigure $server -sockname] 2]
}

proc MakeClient {host port} {
   set client [socket $host $port]
   fileevent  $client readable "ReadSock $client"
   fconfigure $client -buffering none -blocking 0
   return $client
}

proc AcceptConnection {sock ip port} {
   puts "Aceptada conexion desde $ip. ($sock)"
   fconfigure $sock -buffering none -blocking 0
   fileevent  $sock readable "ReadSock $sock"
}

proc ReadSock {sock} {
   if {[eof $sock]} {
      close $sock
      return
   }
   puts "Recibido: [gets $sock]"
}
```

Procesos

Comandos:

- `exec` *comando*
  `| |& < > >> >& 2>> 2> & etc.`
  `exec sort < .login | wc -l 2> /dev/null`
  `auto_noexec`
- `exit` [*status*]
- `open | command`
- `pid`

Trazado de variables

Asociar la ejecucion de un script con la lectura, escritura o desctruccion de una variable.

Comandos:

- `trace variable` *var op comando*

  var es variable, array o elemento del array.
  op es `r`, `w` o `u`.
  A *comando* se añaden tres argumentos, el nombre de la variable o array, el índice del array si es el caso, y *op*.
  El comando se ejecuta en el mismo ámbito que disparó la traza.
  El comando puede modificar el valor de la variable (ej: read-only).
  Si el comando genera un error se propaga a la operación que disparó la traza.
  Dentro del comando asociado a las trazas de lectura o escritura, se desabilitan temporalmente las trazas de lectura y escritura, pero no las trazas de `unset`.
  Pueden definirse varias trazas sobre la misma variable. Se ejecutan en el orden inverso de creación.

Trazado de variables (cont)

- `trace vdelete` *var op comando*
  Elimina la traza especificada.
- `trace info` *var*
  Devuelve una lista de pares <*op,comando*>.

EJEMPLO:

```
% set a(12) 23
  23
% proc MultiploDe5 {name indice op} {
     upvar $name x
     set x($indice) [expr $x/5*5]
  }
% proc Fin {args} {
     puts sacabo
  }
% trace variable a w MultiploDe5
% trace variable a u Fin
% set a(15) 53
  50
% set a(12) 23
  20
% unset a(15)
  sacabo
% unset a
  sacabo
```

Mas comandos

- array
- auto_mkindex
- info
- rename
- time
- trace
- unknown
- history
- fconfigure
- fblocked
- library
- load
- package
- subst
- vwait
- clock
- interp
- etc.