EBook Gratis

APRENDIZAJE

tcl

Free unaffiliated eBook created from
Stack Overflow contributors.

#tcl

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: tcl

It is an unofficial and free tcl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official tcl.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con tcl

Observaciones

Tcl es un lenguaje multiplataforma con soporte completo de Unicode.

Flexibilidad: redefine o mejore los comandos existentes o escriba nuevos comandos.

Programación dirigida por eventos: E / S controlada por eventos y seguimiento de variables.

Interfaz de biblioteca: Es muy fácil integrar las bibliotecas C existentes en Tcl y proporcionar una interfaz Tcl a la biblioteca C. Estos "apéndices" de la interfaz no están vinculados a ninguna versión particular de Tcl y continuarán funcionando incluso después de actualizar Tcl.

Interfaz de Tcl: Tcl proporciona una API completa para que use el intérprete de Tcl desde su programa C / Python / Ruby / Java / R.

Versiones

Versión	Notas	Fecha de lanzamiento
8.6.6	Versión actual del parche.	2016-07-27
8.6.5		2016-02-29
8.6.4		2015-03-12
8.6.3		2014-11-12
8.6.2		2014-08-27
8.6.1		2013-09-20
8.6.0	Serie de versión actual recomendada para nuevo código. Sistema de objetos introducido y motor de ejecución no recursivo.	2013-09-20
8.5.19	Versión actual de LTS	2016-02-12
8.5.18		2015-03-06
8.5.17		2014-10-25
8.5.16		2014-08-25
8.5.15		2013-09-18

Versión	Notas	Fecha de lanzamiento
8.5.14		2013-04-03
8.5.13		2012-11-12
8.5.12		2012-07-27
8.5.11		2011-11-04
8.5.10		2011-06-24
8.5.9		2010-09-08
8.5.8		2009-11-16
8.5.7		2009-04-15
8.5.6		2008-12-23
8.5.5		2008-10-15
8.5.4		2008-08-15
8.5.3		2008-06-30
8.5.2		2008-03-28
8.5.1		2008-02-05
8.5.0	Versión soportada más antigua actual. Introducción de sintaxis de expansión, diccionarios y comandos de conjunto.	2007-12-20
8.4.20	Lanzamiento final de la serie 8.4. *No habrá más lanzamientos de 8.4.*	2013-06-01
8.4.19		2008-04-18
8.4.18		2008-02-08
8.4.17		2008-01-04
8.4.16		2007-09-21
8.4.15		2007-05-25
8.4.14		2006-10-19
8.4.13		2006-04-19
8.4.12		2005-12-03

Versión	Notas	Fecha de lanzamiento
8.4.11		2005-06-28
8.4.10		2005-06-04
8.4.9		2004-12-07
8.4.8		2004-11-22
8.4.7		2004-07-25
8.4.6		2004-03-01
8.4.5		2003-11-24
8.4.4		2003-07-22
8.4.3		2003-05-19
8.4.2		2003-03-03
8.4.1		2002-10-22
8.4.0	Primer lanzamiento por Tcl Core Team. Muchas mejoras de rendimiento. Soporte mejorado de 64 bits.	2002-09-18
8.3.5		2002-10-18
8.3.4		2001-10-19
8.3.3		2001-04-06
8.3.2		2000-08-09
8.3.1		2000-04-26
8.3.0	Mejoras de rendimiento.	2000-02-10
8.2	Liberación de estabilización	1999-08-18
8.1	Se introdujo el soporte Unicode.	1999-04-30
8.0	Se introdujo el motor de compilación bytecode	1997-08-16

Examples

Instalación

Instalación de **Tcl 8.6.4** en **Windows** :

1. La forma más fácil de obtener Tcl en una máquina con Windows es instalar la distribución **ActiveTcl** desde ActiveState.

2. Vaya a www.activestate.com y siga los enlaces para descargar la Free Community Edition de ActiveTcl para Windows (elija la versión de 32/64 bits de manera apropiada).

3. Ejecute el instalador, lo que resultará en una instalación nueva de ActiveTcl generalmente en el directorio **C: \ Tcl** .

4. Abra un indicador de comandos para probar la instalación, escriba "tclsh" que debería abrir una consola tcl interactiva. Ingrese "info patchlevel" para verificar la versión de tcl que se instaló y debería mostrar una salida del formulario "8.6.x" según la edición de ActiveTcl que se haya descargado.

- También es posible que desee agregar "C: \ Tcl \ bin" o su equivalente a la variable **PATH de** su entorno.

```
C:\>tclsh
% info patchlevel
8.6.4
```

El programa Hello, world en Tcl (y Tk)

El siguiente código puede ingresarse en un shell Tcl (`tclsh`), o en un archivo de script y ejecutarse a través de un shell Tcl:

```
puts "Hello, world!"
```

Da el argumento de la cadena `Hello, world!` a la orden `puts` . El comando `puts` escribe su argumento a la salida estándar (su terminal en modo interactivo) y agrega una nueva línea después.

En un shell habilitado para Tk, esta variación se puede usar:

```
pack [button .b -text "Hello, world!" -command exit]
```

Crea un botón gráfico con el texto `Hello, world!` y lo agrega a la ventana de la aplicación. Cuando se presiona, la aplicación sale.

Un shell habilitado para Tk se inicia como: `wish` O usando `tclsh` junto con la siguiente declaración:

```
package require Tk
```

Características de Tcl

- Portabilidad multiplataforma
    - Se ejecuta en Windows, Mac OS X, Linux y prácticamente en todas las variantes de Unix.
- Programación dirigida por eventos
    - Desencadenar eventos basados en variables de lectura / escritura / desarmado.
    - Desencadena eventos cuando se ingresa un comando o se deja.
    - Desencadena eventos cuando un canal de E / S (archivo o red) se vuelve legible / grabable.
    - Crea tus propios eventos.
    - Activar un comando basado en un temporizador.
- Programación orientada a objetos
    - Mixins.
    - Superclases y subclases.
- Gramática sencilla
- Soporte completo de Unicode
    - Simplemente funciona. No se necesitan comandos especiales para manejar cadenas de Unicode.
    - Convierte desde y hacia diferentes sistemas de codificación con facilidad.
- Flexible
    - Crear nuevas estructuras de control y comandos.
    - Acceder a las variables en el contexto del procedimiento de llamada.
    - Ejecutar código en el contexto del procedimiento de llamada.
- Potentes capacidades de introspección.
    - Muchos depuradores de Tcl se han escrito en Tcl.
- Interfaz de la biblioteca
    - Integre las bibliotecas C existentes y proporcione una interfaz Tcl a la biblioteca.
    - Los "talones" de la biblioteca no están vinculados a ninguna versión particular de Tcl y seguirán funcionando después de una actualización de Tcl.
- API completa
    - Incruste un intérprete de Tcl en su idioma favorito.
    - Python, Ruby, R, Java y otros incluyen una API Tcl.
- Biblioteca Bigint incrustada.
    - No se necesitan acciones especiales para manejar números muy grandes.
- Intérpretes seguros
    - Crea espacios limitados en los que se pueda ejecutar el código de usuario.
    - Habilitar y deshabilitar comandos específicos para el intérprete.
- Expresiones regulares
    - Un potente y rápido motor de expresión regular escrito por Henry Spencer (creador de expresiones regulares).

Instalación de paquetes a través de la taza de té

Hoy en día, muchos idiomas son compatibles con el servidor de **archivos** para instalar sus paquetes en su máquina local. TCL también tiene el mismo servidor de archivo que llamamos como Teacup

```
teacup version
teacup search <packageName>
```

Ejemplo

```
teacup install Expect
```

Lea Empezando con tcl en línea: https://riptutorial.com/es/tcl/topic/3029/empezando-con-tcl

Capítulo 2: Argumentos de procedimiento

Observaciones

Referencias: proc
Expansión del argumento (sección 5)

Examples

Un procedimiento que no acepta argumentos.

```
proc myproc {} {
    puts "hi"
}
myproc
# => hi
```

Una lista de argumentos vacía (el segundo argumento después del nombre del procedimiento, "myproc") significa que el procedimiento no aceptará argumentos.

Un procedimiento que acepta dos argumentos.

```
proc myproc {alpha beta} {
    ...
    set foo $alpha
    set beta $bar     ;# note: possibly useless invocation
}

myproc 12 34          ;# alpha will be 12, beta will be 34
```

Si la lista de argumentos consta de palabras, esos serán los nombres de las variables locales en el procedimiento y sus valores iniciales serán iguales a los valores de los argumentos en la línea de comando. Los argumentos se pasan por valor y lo que suceda con los valores de las variables dentro del procedimiento no influirá en el estado de los datos fuera del procedimiento.

Un procedimiento que acepta un número variable de argumentos.

```
### Definition
proc myproc {alpha {beta {}} {gamma green}} {
    puts [list $alpha $beta $gamma]
}
```

```
### Use
myproc A
# => A {} green
myproc A B
# => A B green
myproc A B C
```

```
# => A B C
```

Este procedimiento acepta uno, dos o tres argumentos: aquellos parámetros cuyos nombres son el primer elemento en una lista de dos elementos son opcionales. Las variables de parámetros (`alpha` , `beta` , `gamma`) obtienen tantos valores de argumento como están disponibles, asignados de izquierda a derecha. Las variables de parámetros que no obtienen ningún valor de argumento obtienen sus valores del segundo elemento de la lista de la que formaban parte.

Tenga en cuenta que los argumentos opcionales deben aparecer al final de la lista de argumentos. Si el argumento $_{N-1}$ es opcional, el argumento $_{N}$ también debe ser opcional. Si en un caso, donde el usuario tiene el argumento $_{N}$ pero no el argumento $_{N-1}$, el valor predeterminado del argumento $_{N-1}$ debe mencionarse explícitamente antes del argumento $_{N}$, mientras se llama al procedimiento.

```
myproc A B C D
# (ERROR) wrong # args: should be "myproc alpha ?beta? ?gamma?"
```

El procedimiento no acepta más de tres argumentos: tenga en cuenta que se crea automáticamente un mensaje de error útil que describe la sintaxis del argumento.

Un procedimiento que acepta cualquier número de argumentos.

```
proc myproc args { ... }
proc myproc {args} { ... } ;# equivalent
```

Si el nombre de parámetro especial `args` es el último elemento en la lista de argumentos, recibe una lista de todos los argumentos en ese punto en la línea de comando. Si no hay ninguno, la lista está vacía.

Puede haber argumentos, incluidos los opcionales, antes de `args` :

```
proc myproc {alpha {beta {}} args} { ... }
```

Este procedimiento aceptará uno o más argumentos. Los primeros dos, si están presentes, serán consumidos por `alpha` y `beta` : la lista del resto de los argumentos se asignará a `args` .

Un procedimiento que acepta un nombre / referencia a una variable.

```
proc myproc {varName alpha beta} {
    upvar 1 $varName var
    set var [expr {$var * $alpha + $beta}]
}
set foo 1
myproc foo 10 5
puts $foo
# => 15
```

En este caso particular, al procedimiento se le da el nombre de una variable en el alcance actual. Dentro de un procedimiento Tcl, tales variables no son visibles automáticamente, pero el

comando `upvar` puede crear un alias para una variable de otro nivel de pila: 1 significa el nivel de pila de la persona que llama, # 0 significa el nivel global, etc. En este caso, el nivel 1 de la pila y el nombre `foo` (de la variable de parámetro `varName`) permiten que `upvar` encuentre esa variable y cree un alias llamado `var` . Cada operación de lectura o escritura en `var` también pasa a `foo` en el nivel de pila de la persona que llama.

La sintaxis {*}

A veces, lo que tiene es una lista, pero el comando que desea pasar a los elementos de la lista exige que se obtenga cada elemento como un argumento separado. Por ejemplo: el comando `winfo children` devuelve una lista de ventanas, pero el comando `destroy` solo tomará una secuencia de argumentos de nombre de ventana.

```
set alpha [winfo children .]
# => .a .b .c
destroy $alpha
# (no response, no windows are destroyed)
```

La solución es usar la sintaxis `{*}` :

```
destroy {*}[winfo children .]
```

o

```
destroy {*}$alpha
```

Lo que hace la sintaxis `{*}` es tomar el siguiente valor (¡sin espacios en blanco entre ellos!) Y empalmar los elementos de ese valor en la línea de comandos como si fueran argumentos individuales.

Si el siguiente valor es una lista vacía, nada se empalma en:

```
puts [list a b {*}{} c d]
# => a b c d
```

Si hay uno o más elementos, se insertan:

```
puts [list a b {*}{1 2 3} c d]
# => a b 1 2 3 c d
```

Lea Argumentos de procedimiento en línea: https://riptutorial.com/es/tcl/topic/3365/argumentos-de-procedimiento

Capítulo 3: Construcciones de lenguaje Tcl

Sintaxis

- # Este es un comentario válido
- # Este es un {comentario} válido

Examples

Colocando comentarios

Los comentarios en Tcl son mejor considerados como otro comando.
Un comentario consiste en un `#` seguido de cualquier número de caracteres hasta la nueva línea.
Puede aparecer un comentario donde se pueda colocar un comando.

```
# this is a valid comment
proc hello { } {
  # the next comment needs the ; before it to indicate a new command is
  # being started.
  puts "hello world" ; # this is valid
  puts "dlrow olleh" # this is not a valid comment

  # the comment below appears in the middle of a string.
  # is is not valid.
  set hw {
      hello ; # this is not a valid comment
      world
      }

  gets stdin inputfromuser
  switch inputfromuser {
     # this is not a valid comment.
     # switch expects a word to be here.
     go {
       # this is valid.  The switch on 'go' contains a list of commands
       hello
     }
     stop {
       exit
     }
  }
}
```

Tirantes en los comentarios.

Debido a la forma en que funciona el analizador de lenguaje Tcl, las llaves en el código deben coincidir correctamente. Esto incluye las llaves en los comentarios.

```
proc hw {} {
   # this { code will fail
   puts {hello world}
```

```
}
```

Falta un corchete faltante: se lanzará *un corsé desbalanceado posible en un* error de *comentario* .

```
proc hw {} {
  # this { comment } has matching braces.
  puts {hello world}
}
```

Esto funcionará ya que las llaves se emparejan correctamente.

Citando

En el lenguaje Tcl en muchos casos, no se necesita una cita especial.

Estas son cadenas válidas:

```
abc123
4.56e10
my^variable-for.my%use
```

El lenguaje Tcl divide las palabras en espacios en blanco, por lo que se deben citar los literales o cadenas con espacios en blanco. Hay dos formas de citar cadenas. Con llaves y entre comillas.

```
{hello world}
"hello world"
```

Al citar con llaves, no se realizan sustituciones. Las llaves incrustadas pueden escaparse con una barra invertida, pero tenga en cuenta que la barra diagonal inversa es parte de la cadena.

```
% puts {\{ \}}
\{ \}
% puts [string length {\{ \}}]
5
% puts {hello [world]}
hello [world]
% set alpha abc123
abc123
% puts {$alpha}
$alpha
```

Al citar con comillas dobles, se procesan las sustituciones de comando, barra invertida y variables.

```
% puts "hello [world]"
invalid command name "world"
% proc world {} { return my-world }
% puts "hello [world]"
hello my-world
% puts "hello\tworld"
hello   world
% set alpha abc123
```

```
abc123
% puts "$alpha"
abc123
% puts "\{ \}"
{ }
```

Lea Construcciones de lenguaje Tcl en línea:
https://riptutorial.com/es/tcl/topic/4470/construcciones-de-lenguaje-tcl

Capítulo 4: Estructuras de Control

Sintaxis

- si expr1? entonces? body1 elseif expr2? entonces? body2 ...? else? ? bodyN?
- para comenzar a probar el siguiente cuerpo
- mientras prueba el cuerpo
- foreach varlist1 list1? varlist2 list2 ...? cuerpo

Observaciones

Documentación: break , for , foreach , if , switch , uplevel , while

Examples

Añadiendo una nueva estructura de control a Tcl

En Tcl, una estructura de control es básicamente otro comando. Esta es una posible implementación de un `do ... while` / `do ... until` estructura de control.

```
proc do {body keyword expression} {
    uplevel 1 $body
    switch $keyword {
        while {uplevel 1 [list while $expression $body]}
        until {uplevel 1 [list while !($expression) $body]}
        default {
            return -code error "unknown keyword \"$keyword\": must be until or while"
        }
    }
}
```

Común para ambos tipos de `do` loops es que el script denominado `body` siempre se ejecutará al menos una vez, por lo que lo hacemos de inmediato. El `uplevel 1 $body` invocación de `uplevel 1 $body` significa "ejecutar script en el nivel de pila de la persona que llama". De esta manera, todas las variables utilizadas por el script serán visibles, y los resultados producidos se mantendrán en el nivel de la persona que llama. Luego, el script selecciona, en función del parámetro de `keyword` , si iterar mientras una condición es verdadera o hasta que sea falsa, que es lo mismo que iterar mientras que la negación lógica de la condición es verdadera. Si se da una palabra clave inesperada, se produce un mensaje de error.

si / mientras / para

si *expr1* ? entonces? *body1* elseif *expr2* ? entonces? *body2* ...? else? *? bodyN?*

exprN es una expresión que se evalúa como un valor booleano. *bodyN* es una lista de comandos.

```
set i 5
if {$i < 10} {
  puts {hello world}
} elseif {$i < 70} {
  puts {enjoy world}
} else {
  puts {goodbye world}
}
```

para *comenzar a probar el siguiente cuerpo*

Inicio , *siguiente* y *cuerpo* son listas de comandos. *test* es una expresión que se evalúa como valores booleanos.

El comando **break** se romperá fuera de onda. El comando de **continuar** pasará a la siguiente iteración del bucle.

El uso común es:

```
for {set i 0} {$i < 5} {incr i} {
  puts "$i: hello world"
}
```

Como *inicio* y las *siguientes* son listas de comandos, cualquier comando puede estar presente.

```
for {set i 0; set j 5} {$i < 5} {incr i; incr j -1} {
  puts "i:$i j:$j"
}
```

mientras *prueba el cuerpo*

La *prueba* es cualquier expresión que se evalúa como un valor booleano. Mientras que la *prueba* es verdadera, el *cuerpo* es ejecutado.

```
set x 0
while {$x < 5} {
  puts "hello world"
  incr x
}
```

El comando **break** se romperá fuera de onda. El comando de **continuar** pasará a la siguiente iteración del bucle.

```
set lineCount 0
while {[gets stdin line] >= 0} {
  puts "[incr lineCount]: $line"
  if { $line eq "exit" } {
    break
  }
}
```

Lista de iteración: Foreach

foreach *varlist1 list1 ? lista_var2 lista2 ...? cuerpo*

Foreach es una poderosa estructura de control que permite recorrer una lista o varias listas.

```
set alpha [list a b c d e f]
foreach {key} $alpha {
   puts "key: $key"
}
```

Se pueden especificar múltiples nombres de variables.

```
set alphaindexes [list a 1 b 2 c 3 d 4 e 5 f 6]
foreach {key num} $alphaindexes {
   puts "key:$key num:$num"
}
```

Se pueden iterar varias listas al mismo tiempo.

```
set alpha [list a b c d e f]
set indexes [list 1 2 3 4 5 6]
foreach {key} $alpha {idx} $indexes {
   puts "key: $key idx:$idx"
}
```

Lea Estructuras de Control en línea: https://riptutorial.com/es/tcl/topic/4723/estructuras-de-control

Capítulo 5: Expresiones

Observaciones

Otro beneficio del uso de cadenas de expresión apuntaladas es que el compilador de bytes generalmente puede generar código más eficiente (5 - 10 veces más rápido) a partir de ellos.

Examples

Los problemas con expresiones sin apuntalar.

Es una buena práctica proporcionar argumentos de cadena de expresión como cadenas apuntaladas. El encabezado "Sustitución doble" describe razones importantes detrás de la misma.

El comando `expr` evalúa una cadena de expresión basada en el operador para calcular un valor. Esta cadena se construye a partir de los argumentos en la invocación.

```
expr 1 + 2    ; # three arguments
expr "1 + 2"  ; # one argument
expr {1 + 2}  ; # one argument
```

Estas tres invocaciones son equivalentes y la cadena de expresión es la misma.

Los comandos `if` , `for` , y `while` usan el mismo código de evaluador para sus argumentos de condición:

```
if {$x > 0} ...
for ... {$x > 0} ... ...
while {$x > 0} ...
```

La principal diferencia es que la cadena de expresión de condición siempre debe ser un único argumento.

Como con todos los argumentos en una invocación de comando en Tcl, el contenido puede o no estar sujeto a sustitución, dependiendo de cómo se citan / escapan:

```
set a 1
set b 2
expr $a + $b   ; # expression string is {1 + 2}
expr "$a + $b" ; # expression string is {1 + 2}
expr \$a + \$b ; # expression string is {$a + $b}
expr {$a + $b} ; # expression string is {$a + $b}
```

Hay una diferencia en el tercer y cuarto caso, ya que las barras diagonales inversas impiden la sustitución. El resultado sigue siendo el mismo, ya que el evaluador dentro de `expr` puede realizar la sustitución de la variable Tcl y transformar la cadena a `{1 + 2}` .

```
set a 1
set b "+ 2"
expr $a $b   ; # expression string is {1 + 2}
expr "$a $b" ; # expression string is {1 + 2}
expr {$a $b} ; # expression string is {$a $b}: FAIL!
```

Aquí nos metemos en problemas con el argumento preparado: cuando el evaluador en `expr` realiza sustituciones, la cadena de expresión ya se ha analizado en operadores y operandos, por lo que el evaluador ve una cadena que consta de dos operandos sin operador entre ellos. (El mensaje de error es " `missing operator at _@_ in expression "$a _@_$b"` ".)

En este caso, la sustitución de variables antes de llamar a `expr` evitó un error. Refuerzo del argumento impidió la sustitución de variables hasta la evaluación de la expresión, lo que provocó un error.

Pueden ocurrir situaciones como esta, más típicamente cuando una expresión para evaluar se pasa como una variable o un parámetro. En esos casos, no hay otra opción que dejar el argumento sin unir para permitir que el evaluador de argumentos "descomprima" la cadena de expresión para la entrega a `expr` .

Sin embargo, en la mayoría de los otros casos, el refuerzo de la expresión no hace daño y de hecho puede evitar muchos problemas. Algunos ejemplos de esto:

Doble sustitucion

```
set a {[exec make computer go boom]}
expr $a      ; # expression string is {[exec make computer go boom]}
expr {$a}    ; # expression string is {$a}
```

La forma no marcada realizará la sustitución del comando, que es un comando que destruye la computadora de alguna manera (o encripta o formatea el disco duro, o lo que sea). La forma reforzada realizará una sustitución de variable y luego intentará (y fallará) hacer que algo de la cadena "[exec make computer go boom]". Se evitó el desastre.

Bucles sin fin

```
set i 10
while "$i > 0" {puts [incr i -1]}
```

Este problema afecta tanto `for` como `for while` . Si bien parece que este bucle haría una cuenta regresiva a 0 y saldría, el argumento de condición a while en realidad siempre es `10>0` porque así se evaluó que era el argumento cuando se activó el comando `while` . Cuando el argumento está preparado, se pasa al comando `while` como `$i>0` , y la variable se sustituye una vez por cada iteración. Use esto en su lugar:

```
while {$i > 0} {puts [incr i -1]}
```

Evaluación total

```
set a 1
if "$a == 0 && [incr a]" {puts abc}
```

¿Cuál es el valor de `a` después de ejecutar este código? Dado que el operador `&&` solo evalúa el operando derecho si el operando izquierdo es verdadero, el valor aún debería ser 1. Pero en realidad, es 2. Esto se debe a que el evaluador de argumentos ya realizó todas las sustituciones de variables y comandos en el momento en que la cadena de expresión es evaluado Use esto en su lugar:

```
if {$a == 0 && [incr a]} {puts abc}
```

Varios operadores (las conectivas lógicas `||` y `&&` , y el operador condicional `?:` :) Se definen para *no* evaluar todos sus operandos, pero solo pueden funcionar según lo diseñado si la cadena de expresión está preparada.

Multiplicando una variable por 17

```
set myVariable [expr { $myVariable * 17 }]
```

Esto muestra cómo puedes usar una expresión simple para actualizar una variable. El comando `expr` no actualiza la variable por ti; necesitas tomar su resultado y escribirlo en la variable con `set` .

Tenga en cuenta que las nuevas líneas no son importantes en el pequeño lenguaje que entiende `expr` , y agregarlas puede hacer que las expresiones más largas sean mucho más fáciles de leer.

```
set myVariable [expr {
    $myVariable * 17
}]
```

Esto hace *exactamente* lo mismo sin embargo.

Llamando un comando Tcl desde una expresión

A veces necesitas llamar a un comando Tcl desde tu expresión. Por ejemplo, suponiendo que necesitas la longitud de una cadena. Para hacer eso, simplemente usa una `[...]` secuencia en la expresión:

```
set halfTheStringLength [expr { [string length $theString] / 2 }]
```

Puede llamar a cualquier comando Tcl de esta manera, pero si se encuentra a sí mismo llamando a `expr` , ***¡deténgase!*** y piensa si realmente necesitas esa llamada extra. Por lo *general,* puedes hacerlo bien poniendo la expresión interna entre paréntesis.

Error de palabra desnuda no válido

En Tcl en sí, no es necesario citar una cadena que consiste en una sola palabra. En el lenguaje de las cadenas de expresión que `expr` evalúa, todos los operandos deben tener un tipo

identificable.

Los operandos numéricos se escriben sin ninguna decoración:

```
expr {455682 / 1.96e4}
```

Así son las constantes booleanas:

```
expr {true && !false}
```

Se reconoce la sintaxis de sustitución de variables de Tcl: el operando se establecerá en el valor de la variable:

```
expr {2 * $alpha}
```

Lo mismo ocurre con la sustitución de comandos:

```
expr {[llength $alpha] > 0}
```

Los operandos también pueden ser llamadas a funciones matemáticas, con una lista de operandos separados por comas entre paréntesis:

```
expr {sin($alpha)}
```

Un operando puede ser una cadena entre comillas dobles o cruzada. Una cadena entre comillas dobles estará sujeta a sustitución como en una línea de comando.

```
expr {"abc" < {def}}
```

Si un operando no es uno de los anteriores, es ilegal. Como no hay ninguna pista que muestre qué tipo de palabra es, `expr` señala un error de palabra desnuda.

Lea Expresiones en línea: https://riptutorial.com/es/tcl/topic/3052/expresiones

Capítulo 6: Expresiones regulares

Sintaxis

- ¿Regexp? ¿Switches? ¿Cadena exp? matchVar? ? subMatchVar subMatchVar ...?
- ¿Regsub? ¿Switches? exp string subSpec? varName?

Observaciones

Este tema no pretende discutir las expresiones regulares en sí. Hay muchos recursos en Internet que explican expresiones regulares y herramientas para ayudar a construir expresiones regulares.

Este tema tratará de cubrir los conmutadores comunes y los métodos de uso de expresiones regulares en Tcl y algunas de las diferencias entre Tcl y otros motores de expresiones regulares.

Las expresiones regulares son generalmente lentas. La primera pregunta que debe hacer es "¿Realmente necesito una expresión regular?". Solo iguala lo que quieras. Si no necesita los otros datos, no los combine.

A los efectos de estos ejemplos de expresiones regulares, el interruptor expandido se utilizará para poder comentar y explicar la expresión regular.

Examples

Pareo

El comando `regexp` se usa para hacer coincidir una expresión regular con una cadena.

```
# This is a very simplistic e-mail matcher.
# e-mail addresses are extremely complicated to match properly.
# there is no guarantee that this regex will properly match e-mail addresses.
set mydata "send mail to john.doe.the.23rd@no.such.domain.com please"
regexp -expanded {
    \y           # word boundary
    [^@\s]+      # characters that are not an @ or a space character
    @            # a single @ sign
    [\w.-]+      # normal characters and dots and dash
    \.           # a dot character
    \w+          # normal characters.
    \y           # word boundary
    } $mydata emailaddr
puts $emailaddr
john.doe.the.23rd@no.such.domain.com
```

El comando `regexp` devolverá un valor 1 (verdadero) si se realizó una coincidencia o 0 (falso) si no lo hizo.

```
set mydata "hello wrld, this is Tcl"
```

```
# faster would be to use: [string match *world* $mydata]
if { [regexp {world} $mydata] } {
   puts "spelling correct"
} else {
   puts "typographical error"
}
```

Para hacer coincidir todas las expresiones en algunos datos, use el interruptor -all y el interruptor -inline para devolver los datos. Tenga en cuenta que el valor predeterminado es tratar las líneas nuevas como cualquier otro dato.

```
# simplistic english ordinal word matcher.
set mydata {
    This is the first line.
    This is the second line.
    This is the third line.
    This is the fourth line.
    }
set mymatches [regexp -all -inline -expanded {
    \y                  # word boundary
    \w+                 # standard characters
    (?:st|nd|rd|th)     # ending in st, nd, rd or th
                        # The ?: operator is used here as we don't
                        # want to return the match specified inside
                        # the grouping () operator.
    \y                  # word boundary
    } $mydata]
puts $mymatches
first second third fourth
# if the ?: operator was not used, the data returned would be:
first st second nd third rd fourth th
```

Manejo de nueva linea

```
# find real numbers at the end of a line (fake data).
set mydata {
    White 0.87 percent saturation.
    Specular reflection: 0.995
    Blue 0.56 percent saturation.
    Specular reflection: 0.421
    }
# the -line switch will enable newline matching.
# without -line, the $ would match the end of the data.
set mymatches [regexp -line -all -inline -expanded {
    \y                  # word boundary
    \d\.\d+             # a real number
    $                   # at the end of a line.
    } $mydata]
puts $mymatches
0.995 0.421
```

Unicode no requiere manejo especial.

```
% set mydata {123ÂÃÄÈ456}
123ÂÃÄÈ456
% regexp {[[:alpha:]]+} $mydata match
1
```

```
% puts $match
ÂÃÄÈ
% regexp {\w+} $mydata match
1
% puts $match
123ÂÃÄÈ456
```

Documentación: regexp re_syntax

Mezclando codificadores codificadores y no codiciosos

Si tiene una coincidencia codiciosa como el primer cuantificador, todo el RE será codicioso,

Si tiene una coincidencia no codiciosa como el primer cuantificador, el RE completo no será codicioso.

```
set mydata {
    Device widget1: port: 156 alias: input2
    Device widget2: alias: input1
    Device widget3: port: 238 alias: processor2
    Device widget4: alias: output2
    }
regexp {Device\s(\w+):\s(.*?)alias} $mydata alldata devname devdata
puts "$devname $devdata"
widget1 port: 156 alias: input2
regexp {Device\s(.*?):\s(.*?)alias} $mydata alldata devname devdata
puts "$devname $devdata"
widget1 port: 156
```

En el primer caso, el primer \ w + es codicioso, por lo que todos los cuantificadores se marcan como codiciosos y el. *? coincide más de lo esperado.

En el segundo caso, el primero. *? no es codicioso y todos los cuantificadores están marcados como no codiciosos.

Es posible que otros motores de expresión regular no tengan problemas con los cuantificadores codiciosos / no codiciosos, pero son mucho más lentos.

Henry Spencer escribió : *... El problema es que es muy, muy difícil escribir una generalización de esos estados que abarca las expresiones regulares mixto greediness - una definición adecuada, independiente de la implementación de lo que expresiones regulares mixto greediness deben coincidir - - Y les hace hacer "lo que la gente espera". He intentado. Todavía estoy intentando No hay suerte hasta ahora. ...*

Sustitución

El comando `regsub` se utiliza para la coincidencia y sustitución de expresiones regulares.

```
set mydata {The yellow dog has the blues.}
# create a new string; only the first match is replaced.
set newdata [regsub {(yellow|blue)} $mydata green]
puts $newdata
```

```
The green dog has the blues.
# replace the data in the same string; all matches are replaced
regsub -all {(yellow|blue)} $mydata red mydata
puts $mydata
The red dog has the reds.
# another way to create a new string
regsub {(yellow|blue)} $mydata red mynewdata
puts $mynewdata
The red dog has the blues.
```

Uso de referencias inversas para hacer referencia a datos coincidentes.

```
set mydata {The yellow dog has the blues.}
regsub {(yellow)} $mydata {"\1"} mydata
puts $mydata
The "yellow" dog has the blues.
```

Documentación: regsub re_syntax

Diferencias entre el motor RE de Tcl y otros motores RE.

- \ m: principio de una palabra.
- \ M: Fin de una palabra.
- \ y: límite de la palabra.
- \ Y: un punto que no es un límite de palabra.
- \ Z: coincide con el final de los datos.

Documentación: re_syntax

Coincidencia de una cadena literal con una expresión regular

A veces es necesario hacer coincidir una cadena (sub) literal con una expresión regular a pesar de la subcadena que contiene metacaracteres RE. Mientras que sí, es posible escribir código para insertar barras diagonales apropiadas para hacer que funcione (usando el `string map`) es más fácil simplemente prefijar el patrón con `***=` , lo que hace que el motor de RE trate el resto de la cadena como solo caracteres literales , deshabilitando *todos los* metacaracteres adicionales.

```
set sampleText "This is some text with \[brackets\] in it."
set searchFor {[brackets]}

if {[ regexp ***=$searchFor $sampleText ]} {
    # This message will be printed
    puts "Found it!"
}
```

Tenga en cuenta que esto también significa que no puede usar ninguno de los anclajes.

Lea Expresiones regulares en línea: https://riptutorial.com/es/tcl/topic/5205/expresiones-regulares

Capítulo 7: Los diccionarios

Observaciones

Los diccionarios en Tcl son *valores* que contienen una asignación de valores arbitrarios a otros valores arbitrarios. Fueron introducidos en Tcl 8.5, aunque hay versiones limitadas para (ahora sin soporte) Tcl 8.4. Los diccionarios son sintácticamente lo mismo que las listas con un número par de elementos; el primer par de elementos es la primera clave y el valor del diccionario, el segundo par es la segunda tupla.

Así:

```
fox "quick brown" dogs "lazy"
```

Es un diccionario válido. La misma clave puede ser varias veces, pero es exactamente como si el valor de este último estuviera en el valor anterior; estos son el mismo diccionario:

```
abcd {1 2 3} defg {2 3 4} abcd {3 4 5}
```

```
abcd {3 4 5} defg {2 3 4}
```

El espacio en blanco no es importante, al igual que con las listas.

Un concepto importante con los diccionarios es el orden de iteración; los diccionarios intentan usar el orden de inserción de claves como su orden de iteración, aunque cuando actualiza el valor de una clave que ya existe, sobrescribe el valor de esa clave. Nuevas llaves van al final.

Referencias: dict

Examples

Lista-anexando a un diccionario anidado

Si tenemos este diccionario:

```
set alpha {alice {items {}} bob {items {}} claudia {items {}} derek {items {}}}
```

Y desea agregar "fork" y "peanut" a los artículos de Alice, este código no funcionará:

```
dict lappend alpha alice items fork peanut
dict get $alpha alice
# => items {} items fork peanut
```

Debido a que sería imposible para el comando saber dónde terminan los tokens de clave y los valores que se deben agregar al inicio de la lista, el comando se limita a un token de clave.

La forma correcta de adjuntar al diccionario interno es esta:

```
dict with alpha alice {
    lappend items fork peanut
}
dict get $alpha alice
# => items {fork peanut}
```

Esto funciona porque el comando `dict with` comando nos permite atravesar diccionarios anidados, tantos niveles como la cantidad de tokens clave que proporcionamos. Luego crea variables con los mismos nombres que las claves en ese nivel (solo una aquí: `items`). Las variables se inicializan al valor del elemento correspondiente en el diccionario. Si cambiamos el valor, ese valor modificado se usa para actualizar el valor del elemento del diccionario cuando finaliza el script.

(Tenga en cuenta que las variables continúan existiendo cuando el comando ha finalizado).

Uso básico de un diccionario.

Creando un diccionario:

```
set mydict [dict create a 1 b 2 c 3 d 4]
dict get $mydict b ; # returns 2
set key c
set myval [dict get $mydict $key]
puts $myval
# remove a value
dict unset mydict b
# set a new value
dict set mydict e 5
```

Las claves del diccionario se pueden anidar.

```
dict set mycars mustang color green
dict set mycars mustang horsepower 500
dict set mycars prius-c color orange
dict set mycars prius-c horsepower 99
set car [dict get $mycars mustang]
# $car is: color green horsepower 500
dict for {car cardetails} $mycars {
  puts $car
  dict for {key value} $cardetails {
    puts "  $key: $value"
  }
}
```

El comando dict get puede provocar un error.

```
set alpha {a 1 b 2 c 3}
dict get $alpha b
# => 2
dict get $alpha d
# (ERROR) key "d" not known in dictionary
```

Si se usa `dict get` para recuperar el valor de una clave faltante, se genera un error. Para evitar el error, use `dict exists` :

```
if {[dict exists $alpha $key]} {
    set result [dict get $alpha $key]
} else {
    # code to deal with missing key
}
```

Cómo manejar una clave faltante, por supuesto, depende de la situación: una forma simple es establecer el `result` en un valor "vacío" predeterminado.

Si el código nunca intenta recuperar otras claves que están en el diccionario, no `dict get` producirá un error en la `dict get` del dictado. Pero para claves arbitrarias, `dict get` es una operación que debe ser protegida. Preferiblemente, al probar con `dict exists` , aunque la captura de excepciones también funcionará.

Iterando sobre un diccionario

Puede recorrer el contenido de un diccionario con `dict for` , que es similar a `foreach` :

```
set theDict {abcd {ab cd} bcde {ef gh} cdef {ij kl}}
dict for {theKey theValue} $theDict {
    puts "$theKey -> $theValue"
}
```

Esto produce esta salida:

```
abcd -> ab cd
bcde -> ef gh
cdef -> ij kl
```

Obtendría la misma salida usando las `dict keys` para listar las teclas e iterando sobre eso:

```
foreach theKey [dict keys $theDict] {
    set theValue [dict get $theDict $theKey]
    puts "$theKey -> $theValue"
}
```

Pero `dict for` es más eficiente.

Lea Los diccionarios en línea: https://riptutorial.com/es/tcl/topic/4065/los-diccionarios

Capítulo 8: Nombres y nombres de archivo

Sintaxis

- archivo dirname *filepath*
- file tail *filepath*
- *rutaarchivos* nombre raíz de archivos
- extensión de archivo *filepath*
- archivo unirse *ruta1 ruta2 ...*
- archivo normalizar *ruta*
- archivo nativename *ruta*

Examples

Trabajar con rutas y nombres de archivo

```
% set mypath /home/tcluser/sources/tcl/myproject/test.tcl
/home/tcluser/sources/tcl/myproject/test.tcl
% set dir [file dirname $mypath]
/home/tcluser/sources/tcl/myproject
% set filename [file tail $mypath]
test.tcl
% set basefilename [file rootname $filename]
test
% set extension [file extension $filename]
.tcl
% set newpath [file join $dir .. .. otherproject]
/home/tcluser/sources/tcl/myproject/../../otherproject
% set newpath [file normalize $newpath]
/home/tcluser/source/otherproject
% set pathdisp [file nativename $newpath] ; # not on windows...
/home/tcluser/source/otherproject
% set pathdisp [file nativename C:$newpath] ; # on windows...
C:\home\tcluser\source\otherproject
% set normpath [file normalize $pathdisp]
C:/home/tcluser/source/otherproject
```

Documentación: archivo

Lea Nombres y nombres de archivo en línea: https://riptutorial.com/es/tcl/topic/5566/nombres-y-nombres-de-archivo

Capítulo 9: Variables

Sintaxis

- establecer ***varName ? valor?***
- unset? **-nocomplain** ? - - ? *varName varName varName ?*
- pone $ *varName*
- pone [set *varName*]
- variable *varName*
- *varName* global? ***varName varName*** *?*

Observaciones

- Parámetros incluidos dentro de *? ...?* tales como *? varName?* Representa argumentos opcionales a un comando Tcl.
- Documentación: global , upvar

Examples

Asignando valores a las variables.

El `set` comandos se utiliza para asignar valores en Tcl. Cuando se llama con dos argumentos de la siguiente manera,

```
% set tempVar "This is a string."
This is a string.
```

coloca el segundo argumento ("Esto es una cadena") en el espacio de memoria al que hace referencia el primer argumento (tempVar). `set` siempre devuelve el contenido de la variable nombrada en el primer argumento. En el ejemplo anterior, `set` devolvería "Esto es una cadena". sin las comillas.

- Si se especifica el *valor* , entonces el contenido de la variable *varName* se establece igual al *valor* .
- Si *varName* consta solo de caracteres alfanuméricos y no tiene paréntesis, es una variable escalar.
- Si *varName* tiene la forma *varName (índice)* , es un miembro de una matriz asociativa.

Tenga en cuenta que el nombre de la variable no está restringido al alfabeto latino, puede consistir en cualquier combinación de caracteres Unicode (por ejemplo, armenio):

```
% set sոuն house
house
% puts ${sոuն}
house
```

Alcance

```
set alpha 1

proc myproc {} {
    puts $alpha
}

myproc
```

Este código no funciona porque los dos alfas están en diferentes ámbitos.

El `set alpha 1` comandos `set alpha 1` crea una variable en el ámbito global (que la convierte en una variable global).

El comando `puts $alpha` se ejecuta en un ámbito que se crea cuando se ejecuta el comando `myproc` .

Los dos ámbitos son distintos. Esto significa que cuando `puts $alpha` intenta buscar el nombre `alpha` , no encuentra ninguna de esas variables.

Podemos arreglar eso, sin embargo:

```
proc myproc {} {
    global alpha beta
    puts $alpha
}
```

En este caso, dos variables globales, `alpha` y `beta` , están vinculadas a variables de alias (con el mismo nombre) en el alcance del procedimiento. La lectura de las variables de alias recupera el valor en las variables globales y al escribirlas cambia los valores en las variables globales.

Más generalmente, el comando `upvar` crea alias a variables de cualquiera de los ámbitos anteriores. Se puede usar con el alcance global (`#0`):

```
proc myproc {} {
    upvar #0 alpha alpha beta b
    puts $alpha
}
```

Los alias pueden recibir el mismo nombre que la variable que está vinculada a (`alpha`) u otro nombre (`beta` / `b`).

Si llamamos a `myproc` desde el ámbito global, esta variante también funciona:

```
proc myproc {} {
    upvar 1 alpha alpha beta b
    puts $alpha
}
```

El número de alcance `1` significa "el alcance anterior" o "el alcance de la persona que llama".

A menos que realmente sepa lo que está haciendo, `#0` , `0` y `1` son los únicos ámbitos que tienen sentido utilizar con `upvar` . (`upvar 0` crea un alias local para una variable local, no es estrictamente una operación de alcance).

Algunos otros lenguajes definen el alcance mediante llaves y permiten que el código que se ejecuta en cada alcance vea todos los nombres en los ámbitos circundantes. En Tcl se crea un solo ámbito cuando se llama a un procedimiento, y solo sus propios nombres son visibles. Si un procedimiento llama a otro procedimiento, su alcance se apila sobre el alcance anterior, y así sucesivamente. Esto significa que, a diferencia de los lenguajes de estilo C que solo tienen un alcance global y un alcance local (con subtemas), cada alcance actúa como un ámbito cerrado (aunque no inmediatamente visible) para cualquier alcance que haya abierto. Cuando un procedimiento vuelve, su alcance se destruye.

Documentación: global , upvar

Imprimiendo el valor de una variable

Para imprimir el valor de una variable tal como,

```
set tempVar "This is a string."
```

El argumento en la declaración de put está precedido por un signo **$** , que le dice a Tcl que use el valor de la variable.

```
% set tempVar "This is a string."
This is a string.
% puts $tempVar
This is a string.
```

Invocando set con un argumento

`set` también se puede invocar con un solo argumento. Cuando se llama con un solo argumento, devuelve el contenido de ese argumento.

```
% set x 235
235
% set x
235
```

Borrando variable / s

El comando `unset` se usa para eliminar una o más variables.

```
unset ?-nocomplain? ?--? ?name name name name?
```

- Cada *nombre* es un nombre de variable especificado en cualquiera de las formas aceptables para el comando `set` .
- Si un *nombre se* refiere a un elemento de una matriz, ese elemento se elimina sin afectar el

resto de la matriz.
- Si un *nombre* consta de un nombre de matriz sin índice entre paréntesis, se elimina toda la matriz.
- Si se da **-nocomplain** como primer argumento, todos los errores posibles se eliminan de la salida del comando.
- La opción - indica el final de las opciones, y debe usarse si desea eliminar una variable con el mismo nombre que cualquiera de las opciones.

```
% set x 235
235
% set x
235
% unset x
% set x
can't read "x": no such variable
```

Variables del espacio de nombres

El comando `variable` asegura que se crea una variable de espacio de nombres dada. Hasta que se le asigne un valor, el valor de la variable no está definido:

```
namespace eval mynamespace {
    variable alpha
    set alpha 0
}
```

Se puede acceder a la variable desde fuera del espacio de nombres (desde cualquier lugar, de hecho) adjuntando el nombre del espacio de nombres:

```
set ::mynamespace::alpha
```

El acceso se puede simplificar dentro de un procedimiento usando el comando `variable` nuevamente:

```
proc ::mynamespace::myproc {} {
    variable alpha
    set alpha
}
```

Esto crea un alias local para la variable de espacio de nombres.

Para un procedimiento definido en otro espacio de nombres, el nombre de la variable debe contener el espacio de nombres en la invocación de la `variable` :

```
proc myproc {} {
    variable ::mynamespace::alpha
    set alpha
}
```

Lea Variables en línea: https://riptutorial.com/es/tcl/topic/3740/variables

Creditos

S. No	Capítulos	Contributors
1	Empezando con tcl	Brad Lanam, Community, Donal Fellows, klas2iop, Mallikarjunarao Kosuri, Peter Lewerin, stark, vasili111
2	Argumentos de procedimiento	Brad Lanam, Codename_DJ, Donal Fellows, klas2iop, Peter Lewerin
3	Construcciones de lenguaje Tcl	Brad Lanam, stark
4	Estructuras de Control	Brad Lanam, Codename_DJ, Donal Fellows, Peter Lewerin, stark
5	Expresiones	Donal Fellows, nurdglaw, Peter Lewerin, stark
6	Expresiones regulares	Brad Lanam, Donal Fellows
7	Los diccionarios	Brad Lanam, Donal Fellows, Peter Lewerin
8	Nombres y nombres de archivo	Brad Lanam
9	Variables	klas2iop, Peter Lewerin, stark