Eine Einführung in Tcl

von Dominik Wagenführ

Es gibt zahlreiche Skriptsprachen auf dem „Markt“: Bash, Perl, Python, PHP und viele mehr. Etwas unbekannter, aber in manchen Fällen ganz nützlich ist Tcl (Tool Command Language) [1] von John K. Ousterhout. Dieser Artikel soll anhand eines kleinen Beispiels in die Sprache einführen und deren Besonderheiten aufzeigen.

Ein Hinweis vorab: Dieser Artikel soll keine komplette Anleitung für Tcl sein. Es werden nur die Befehle und Kontrollstrukturen erklärt, die in dem Beispiel Verwendung finden. Tcl kann viel mehr . . .

Einführung

Die Idee für Tcl entstand Anfang der 1980er Jahre an der University of California in Berkeley. John K. Ousterhout benötigte eine Sprache für sich und seine Studenten, die über eine C-Bibliothek leicht erweiterbar war, um sie in verschiedenen Situationen leicht anpassen zu können. Das Ergebnis war 1988 die erste Version von Tcl.

Tcl ist eine Interpretersprache, das heißt man schreibt Skriptcode, der zur Laufzeit interpretiert und umgesetzt wird. Inzwischen gibt es aber auch verschiedene Tcl-Compiler [2], die den Quellcode schützen können, möchte man diesen nicht veröffentlichen. Der Geschwindigkeitszuwachs ist dabei aber meist gering bzw. auch stark von der jeweiligen Anwendung abhängig.

Neben Unicode-Support steht Tcl auch auf den meisten Plattformen und Systemen zur Verfügung. Es ist also in der Regel kein Problem, ein Tcl-Skript auf verschiedenen Betriebssystemen laufen zu lassen, ohne Anpassungen vornehmen zu müssen.

Ein weiterer Vorteil ist die gute Erweiterbarkeit. Die bekannteste Erweiterung ist wahrscheinlich Tk, ein GUI-Toolkit, mit dem man leicht grafische Oberflächen für bzw. in Tcl schreiben kann. Auf Tk wird in diesem Artikel aber nicht weiter eingegangen.

Installation und die Shell

Alle Linux-Distributionen sollten in ihren Paketquellen ein Paket mit Namen **tcl** (ggf. mit Versionsnummer) anbieten. In den neueren Distributionen ist zur Zeit **tcl8.5** aktuell.

Das Paket installiert man einfach über die Paketverwaltung. Danach steht die Tcl-Shell zur Verfügung:

```
$ tclsh
```

In der Shell kann man interaktiv Befehle, Konstrukte und eigene Funktionen testen.

Achtung: Die gewohnten Cursortasten zur Bewegung auf einer Zeile funktionieren in der tclsh (per Standard) nicht. Man kann nur Befehle eintippen, mit der Rücklösch-Taste („Backspace“, ←) entfernen und mit Enter abschließen. Sollte man aus Versehen eine Sondertaste gedrückt haben, darf man nicht Enter drücken, ansonsten bleibt die Shell „stehen“ und man kann sie nur mit Strg + D verlassen.

Mit **`exit`** oder Strg + D verlässt man die Shell wieder.

Hinweis: Der Prompt der Tcl-Shell beginnt mit einem Prozentzeichen **%**. Im Folgenden bedeuten also Befehlseingaben mit einem Prozentzeichen am Anfang eine Eingabe in der tclsh und nicht in der „gewöhnlichen“ Shell.

Wer Tcl-Skript schreiben will, muss entsprechend die Shebang-Zeile

```
#!/usr/bin/tclsh
```

als erste Zeile in der Skriptdatei einfügen, damit der passende Interpreter vom System beim Ausführen genutzt wird (siehe dazu auch „Shebang – All der Kram“, freiesMagazin 11/2009 [3]).

Syntax

Tcl wirkt wie eine Mischung aus vielen anderen Sprachen, sodass jeder Programmierer sicherlich etwas finden wird, was ihm bekannt vorkommt oder ggf. dafür sorgt, dass er etwas falsch macht, weil er es so ähnlich kennt.

Befehlstrenner

Befehle werden normalerweise durch Zeilenumbrüche getrennt, können aber auch in einer Zeile

stehen, müssen dann aber durch ein Semikolon getrennt werden:

```
% puts Hallo
Hallo
% puts Welt
Welt
% puts Hallo; puts Welt
Hallo
Welt
```

Am Ende einer Zeile ist aber kein Semikolon notwendig (wie man das vielleicht von C/C++ kennt) – es schadet aber auch nicht.

Variablen

Es gibt in Tcl nur eine Art von Typ: Strings (Zeichenketten). Natürlich wird auch mit Zahlen gerechnet, intern werden diese aber als String gespeichert.

Das heißt, Tcl könnte man trivialerweise als typsicher bezeichnen, da es nur einen Typ gibt. Natürlich zählt das nicht, denn wenn man zwei „Strings“ addiert und der eine repräsentiert keine Zahl, fällt dies erst bei der Ausführung auf, nicht während der Interpretation.

Bevor man eine Variable verwenden kann, muss ihr mit **set** ein Wert zugewiesen werden:

```
% set a "Hallo Welt!"
Hallo Welt!
```

Mittels des Dollarzeichens **$** kann man auf Variablen zugreifen (Variablenersetzung/-substitution):

```
% puts $a
Hallo Welt!
```

Verwendet man Variablen, die nicht definiert/deklariert wurden, erhält man eine Fehlermeldung:

```
% puts $b
can't read "b": no such variable
```

Mittels der **info**-Funktion kann man überprüfen, ob eine Variable existiert:

```
% info exists a
1
% info exists b
0
```

Für Variablennamen gibt es fast keine Einschränkungen. Selbst so etwas ist möglich:

```
% set 1 0
0
% expr 42*$1
0
```

Aber so etwas Böses macht hoffentlich niemand.

Kommentare

Kommentare können über die Raute **#** in einem Skript eingefügt werden. Bei Kommentaren hinter einem Befehl (in der selben Zeile), ist es wichtig, den Befehl mit einem Semikolon abzuschließen. Es hat sich eingebürgert, das Semikolon direkt vor das Kommentarzeichen zu setzen:

```
# Ein Kommentar
set a 42      ;# noch ein Kommentar
```

Arithmetik

Rechnen bzw. mathematische Ausdrücke auswerten kann man mit **expr**:

```
% expr 14*3
42
```

Natürlich kann man auch in Verbindung mit Variablen rechnen:

```
% set a 42
42
% expr 14*3-$a
0
```

Man kann das Ergebnis einer Rechnung auch einer neuen Variablen zuweisen:

```
% set b [expr 14*3-$a]
0
% puts $b
0
```

Allgemein werten eckige Klammern **[...]** den Ausdruck darin aus und „ersetzen“ die Klammern durch das Ergebnis/den Rückgabewert der Auswertung (Befehlsersetzung/-substitution).

Auch eine Selbstzuweisung ist möglich, um zum Beispiel einen Zähler zu erhöhen:

```
% set a [expr $a+1]
43
```

Hierfür existiert der vordefinierte Befehl **incr**:

```
% incr a
44
```

```
% incr a -2
42
```

Wichtig ist, dass das erste Argument, also die Variable, die erhöht („inkrementiert“) werden soll, ohne Auswertezeichen **$** angegeben wird. Als zweiten, optionalen Parameter gibt man an, wie weit man die Variable erhöhen (bei negativer Angabe erniedrigen) will. Der Standardwert ist 1.

Strings

Da alles in Tcl ein String ist, gibt es zahlreiche Funktionen zur String-Manipulation.

Ohne speziellen Befehl kommt das Zusammenfügen von Strings aus:

```
% set a Hallo
Hallo
% set b Welt
Welt
% set c "$a $b"
Hallo Welt
```

Neben den Anführungszeichen sind auch geschweifte Klammern **{...}** in Tcl wichtig (siehe auch weiter unten). So findet innerhalb dieser Klammern keine Variablen- oder Befehlsersetzung statt:

```
% set a Hallo
Hallo
% set b Welt
Welt
% set c {$a $b}
$a $b
```

Wozu ist das gut? Nun, man kann so neue Befehle in Variablen stecken und diese dann mit **eval** auswerten:

```
% set c {puts "$a $b"}
puts "$a $b"
% eval $c
Hallo Welt
```

Natürlich ist dies auch mit Anführungszeichen möglich, wenn man die Auswertung durch das Escapezeichen **** (Backslash) unterdrückt:

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

Hinweis: Die Interpretation der inneren Anführungszeichen muss innerhalb der äußeren Anführungszeichen auch durch einen Backslash unterdrückt werden, damit diese in der Variablen **c** normal erscheinen.

Sicherlich will man auch wissen, wie lang so eine Zeichenkette ist, vor allem, um zu entscheiden, ob diese vielleicht leer ist. Da hilft die Überprüfung mit **string length**:

```
% string length $c
12
```

Listen

Listen sind auch nur wieder Zeichenketten, deren Einträge durch Leerzeichen getrennt sind.

```
% set d "A B C"
A B C
```

Die Größe einer Liste bestimmt man mit **llength**:

```
% llength $d
3
```

Hinzufügen kann man Elemente entweder über **set** oder die spezielle Funktion **lappend**:

```
% set d "$d D"
A B C D
% lappend d E
A B C D E
% llength $d
5
```

Bei **lappend** ist es wichtig, dass man die Variable angibt, nicht die Auswertung.

Auf die einzelnen Elemente kann man mit **lindex** zugreifen, der erste Index ist 0:

```
% lindex $d 0
A
% lindex $d 4
D
% lindex $d 42
```

Wie man sieht, geben Zugriffe auf nicht existierende Elemente keinen Fehler aus, sondern haben einfach nur eine leere Rückgabe, was zumindest eine extra Fehlerbehandlung spart.

Für das zu schreibende Skript benötigt man noch eine etwas speziellere Listenfunktion, um eine Zeichenkette aufzuspalten:

```
% set c "Hallo Welt"
Hallo Welt
% split $c
Hallo Welt
```

Etwas langweilig, nicht? Interessanter wird es, wenn man das optionale Trennungszeichen mit angibt (Standard ist das Leerzeichen):

```
% split $c W
{Hallo } elt
```

In dem Beispiel wurde also der String „Hallo Welt“ am „W“ getrennt. Auch wenn das Ergebnis von Tcl so ausgegeben wurde, gehören die geschweiften Klammern dabei **nicht** zum Ergebnis. Sie sollen nur zeigen, dass der erste Ausdruck ein Leerzeichen am Ende hat. Wie überprüft man das?

```
% lindex [split $c W] 0
Hallo
% string length [lindex [split $c W↷
] 0]
6
```

„Hallo “ besteht also wirklich aus sechs Zeichen.

Arrays

Arrays sind weit mächtiger als Listen. In vielen Programmiersprachen klappt die Indizierung von Arrays nur mittels ganzzahliger Indizes.

```
% set A(0) Hallo
Hallo
% set A(1) Welt
Welt
% puts "$A(0) $A(1)"
Hallo Welt
```

Soweit wirkt das noch normal. In Tcl kann man aber (fast) alles als Index nehmen:

```
% set A(Welt) Hallo
Hallo
% set A(Hallo) Welt
Welt
% puts "$A(Hallo) $A(Welt)"
Welt Hallo
```

Noch spannender wird es, wenn man Variablenersetzung dazu nimmt:

```
% set a Hallo
Hallo
% set b Welt
Welt
% puts "$A($a) $A($b)"
Welt Hallo
```

Auch mehrdimensionale Arrays sind möglich, auch wenn diese in dem Beispielskript keine Anwendung finden. Hierfür nutzt man als Index einfach eine kommaseparierte Liste von Indizes. Genau genommen ist dies aber nur ein Trick, da zum Beispiel

```
% set A(1,3) "Index 1,3"
Index 1,3
```

nicht ein zweidimensionales Array darstellt, sondern einfach nur den Index **1,3** für das Array **A** zuweist.

Wenn man wissen will, welche Indizes es zu einem Array gibt, kann man diese über **array names** bestimmen:

```
% array names A
Welt Hallo 0 1 1,3
```

Hinweis: Es ist nicht möglich, eine Variable und ein Array mit gleichem Namen zu haben. Eine doppelte Zuweisung wird mit einen Fehler abgelehnt.

Wie man oben sieht, sind Sonderzeichen in Variablen möglich. Man kann sogar nur Sonderzeichen (oder auch gar kein Zeichen) als Index benutzen, aber so etwas nutzt hoffentlich niemand ernsthaft:

```
% unset A
% set A(\\) 1
1
% set A(") 2
2
% set A(=) 3
3
% set A() 4
4
% array names A
{} \\ = {"}
```

unset „löscht“ im Übrigen eine gesetzte Variable (egal, ob Array oder normalen String) wieder.

Bedingungen

Um ein vollständiges Skript zu schreiben, muss man natürlich oft Entscheidungen treffen und je nach Gegebenheit anderen Code ausführen:

```
% unset a
% if { [info exists a] } { string ↷
length $a } else { set a "Hallo ↷
Welt" }
Hallo Welt
% if { [info exists a] } { string ↷
length $a } else { set a "Hallo ↷
Welt" }
10
```

Das ist natürlich unübersichtlich. In Skript-Form sähe das so aus (das kann man aber auch so in die Tcl-Shell eingeben):

```
if { [info exists a] } {
    string length $a
} else {
    set a "Hallo Welt"
}
```

Ganz wichtig ist die Platzierung der Klammern in diesem Fall. Wer C/C++ programmiert, ist versucht, so etwas zu schreiben (beispielhaft ohne else-Zweig):

```
if { [info exists a] }
{
    string length $a
}
```

Das Ergebnis beim Ausführen wäre dann die Meldung:

```
wrong # args: no script following "↷
 [info exists a] " argument
```

Eine Besonderheit von Tcl kann man hieran auch sehen:

```
% unset a
% if { [info exists a] } { kram } ↷
else { set a "Hallo Welt" }
Hallo Welt
```

kram existiert natürlich nicht. Dennoch wird die **if**-Abfrage fehlerfrei ausgeführt – zumindest beim ersten Mal. Beim zweiten Mal erscheint:

```
% if { [info exists a] } { kram } ↷
else { set a "Hallo Welt" }
invalid command name "kram"
```

Der Grund dafür ist, dass Tcl blockweise auswertet. Das bedeutet auch, dass, wenn ein inhaltlicher Fehler in einem Codestück steckt, das nie ausgeführt wird, man dies nie bemerken wird. Solange die Syntax im ganzen Code stimmt, ist der Tcl-Interpreter glücklich.

Neben **if** und **else** kann man auch gleich eine oder mehrere weitere Abfragen mit **elseif** einbringen:

```
if { ![info exists a] } {
    set a "Hallo Welt"
} elseif { [string length $a] == 10↷
 } {
    puts $a
    set a "HW"
} else {
    puts $a
}
```

Schleifen

Neben Bedingungen sind Schleifen wohl auch die am häufigsten genutzten Kontrollstrukturen in Programmen. Es gibt in Tcl die **for**-, die **foreach**- und die **while**-Schleife.

Mit der **for**-Schleife iteriert man über eine feste Anzahl von Schritten:

```
% for { set ii 0 } { $ii < 11 } { ↷
incr ii } { puts [expr $ii*$ii] }
```

Dies gibt die Quadratzahlen von 0 bis 10 aus.

Dabei gibt es vier Argumente für den **for**-Aufruf: Das erste setzt („initialisiert“) die Laufvariable, das zweite Argument enthält den Vergleich, wie lange die Schleife ausgeführt wird, das dritte erhöht die Laufvariable und das letzte Argument enthält den Code, der ausgeführt wird, solange die Fortsetzen-Bedingung noch erfüllt ist.

Natürlich kann man auch über andere Variablen iterieren:

```
% for { set a "" } { [string length↷
 $a] < 10 } { set a 1$a } { }
% puts $a
1111111111
```

Bei obiger Schleife ist es im Übrigen egal, ob das **set a 1$a** innerhalb des Blocks zur Erhöhung oder zur Ausführung des Codes steht. Das Resultat ist das gleiche.

Die **foreach**-Schleife benutzt man in der Regel, wenn man über Listenelemente iterieren will:

```
% set A(Hallo) Welt
Welt
% set A(Welt) Hallo
Hallo
% set A(0) 1
1
% set A(1) 0
0
% set indices [array names A]
0 Welt Hallo 1
% foreach index $indices { puts "A(↷
$index) = $A($index)" }
A(0) = 1
A(Welt) = Hallo
A(Hallo) = Welt
A(1) = 0
```

Die **foreach**-Argumente sind also der Laufindex **index**, die Liste, über die gelaufen wird (**$indices**) und danach der Code, der bei jedem Schritt ausgeführt wird.

Natürlich kann man auch über die Liste ohne Zwischenspeicherung in eine Variable iterieren:

```
% foreach index [array names A] { ↷
puts "A($index) = $A($index)" }
```

Die **while**-Schleife ist ähnlich zu einer **for**-Schleife. Die Parameter geben die Fortsetzbedingung und danach den auszuführenden Code an.

```
% set ii 0
0
% while { $ii < 11 } { puts [expr ↷
$ii*$ii]; incr ii }
```

Reguläre Ausdrücke

In Strings lässt sich auch leicht nach regulären Ausdrücken [4] suchen bzw. diese gleich ersetzen:

```
% set c "Hallo Welt"
Hallo Welt
% regexp {l{2}.* W.*t$} $c
1
```

Der Ausdruck sucht also in der Variablen **c** zwei aufeinanderfolgende **l**, danach irgendwelche Zeichen, gefolgt von einem **W** und ein abschließendes **t**.

```
% set d "Ballons fliegen in die ↷
Welt"
Ballons fliegen in die Welt
% regexp {l{2}.* W.*t$} $d
1
% set d "Ballons fliegen nicht"
Ballons fliegen nicht
% regexp {l{2}.* W.*t$} $d
0
```

Es ist im Übrigen immer sinnvoll, reguläre Ausdrücke in geschweifte Klammern zu setzen, damit nicht aus Versehen Befehle oder Variablen ersetzt werden (außer man bezweckt genau das).

Über **regsub** kann man die gefundenen Muster dann ersetzen:

```
% regsub {l{2}.* W.*t$} $c "i" d
1
% puts $d
Hai
```

Hierbei wird der gefundene Ausdruck durch **i** ersetzt und das Ergebnis in die Variable **d** geschrieben.

Reguläre Ausdrücke sind natürlich noch wesentlich mächtiger und beide Tcl-Kommandos **regexp** und **regsub** haben weitere Optionen und Parameter, um das Ergebnis zu verfeinern.

Eigene Funktionen

Wenn man ein Skript schreibt, gibt es immer wieder Aufgaben, die sich wiederholen. Es wäre unsinnig, den Code für die Aufgabe jedes Mal zu kopieren. Daher schreibt man sich dafür eine Funktion, die man im Skript aufruft und die dann den Code ausführt. Dabei können natürlich auch Parameter übergeben und ein Rückgabewert zurückgegeben werden.

```
% proc potenz { basis { exponent 2 ↷
} } {
    set result 1
    for { set ii 0 } { $ii < ↷
    $exponent } { incr ii } {
        set result [expr $result*↷
        $basis]
    }
    return $result
}
```

Diese Funktion berechnet Potenzen. Nach dem Schlüsselwort **proc** folgt zuerst der Namen der neuen Funktion **potenz**.

Danach folgt in Klammern eine Liste von Argumenten, die übergeben werden können. In dem Fall sind es zwei: **basis** und **exponent**.

exponent hat dabei die Besonderheit, dass er durch die geschweiften Klammern einen Standardwert **2** zugeordnet bekommt. Man kann **potenz** also mit einem oder mit zwei Parametern aufrufen.

Das Ergebnis wird über **return** zurückgeliefert:

```
% potenz 5
25
% potenz 2 3
8
% potenz 3 0
1
% potenz -2 3
-8
```

Kommandozeileparameter

Da das Beispielskript unten mit einer Datei als Argument aufgerufen werden soll, muss man im Skript selbst irgendwie an die Argumente kommen. Hierfür dienen die beiden vordefinierten Variablen **argv** und **argc**:

```
#!/usr/bin/tclsh
puts "Anzahl der Parameter: $argc"
puts "Parameterliste:       $argv"
```

Listing 1: args.tcl

argc beschreibt also die Anzahl der übergebenen Parameter und über **argv** kann man auf diese zugreifen:

```
$ ./args.tcl Hallo Welt
Anzahl der Parameter: 2
Parameterliste:       Hallo Welt
```

Beispielaufgabe

Die Befehle und Kontrollstrukturen von Tcl lernt man leichter, wenn man sich diese anhand eines echten Beispiels anschaut, anstatt nur kleinere Codeschnipsel vorliegen zu haben. Aus diesem Grund soll hier ein „echtes“ Beispiel behandelt werden.

Ein Schüler zeichnet seinen täglichen Tagesablauf in digitaler Form auf. Das Programm, das die Daten aufzeichnet, speichert diese in folgender Form in einer Textdatei **tasks.txt** ab:

```
newTask \
  "1000.01 Schule/Hausaufgaben" \
  "08.02.2010 07:45:00" \
  "08.02.2010 12:55:00"
```

Die erste Zeile dient als Trenner zwischen den verschiedenen Tätigkeiten. Die zweite Zeile enthält eine eindeutige Identifizierungsnummer samt einer Klartextbeschreibung der Tätigkeit. Die dritte Zeile gibt den Anfang der Tätigkeit an und die letzte Zeile das Ende der Tätigkeit.

Aufgabe ist es nun, ein Tcl-Skript zu schreiben, welches diese Aufgabendatei auswertet, dabei die Zeiten der jeweiligen gleichen Tätigkeiten addiert und eine Zusammenfassung ausgibt. Wieso die Datei ein so „seltsames“ Format hat, wird sich später zeigen.

Beispielskript

Es wird nun das Tcl-Skript **check_tasks.tcl** analysiert, wobei nur einzelne Funktionen oder Codeteile angegeben werden sollen.

Möchte man das Skript bzw. die Methoden interaktiv in der Tcl-Shell testen, kann man das Skript über

```
% source check_tasks.tcl
```

einbinden. Danach ist es möglich, alle Skript-Methoden in der tclsh direkt aufzurufen.

Skriptaufruf ausgeben

Sollte der Benutzer das Skript falsch aufrufen, ist es immer sinnvoll, wenn man ihm zeigt, wie der Aufruf normalerweise lauten sollte. Dies erledigt die Funktion **printUsage**, die als Argument den Skriptaufruf erhält:

```
proc printUsage { call } {
  puts "Usage: $call TASK_FILE"
}
```

Zeit und Datum überprüfen

Damit das Skript richtig arbeiten kann und die Zeitwerte korrekt extrahiert, sollte man vorher überprüfen, ob die übergebenen Daten überhaupt korrekt sein können, also ein bestimmtes Muster erfüllen.

Mit Hilfe von regulären Ausdrücken wird so in den Funktionen **checkTimeFormat** und **checkDateFormat** überprüft, ob die übergebene Zeichenkette die Form **HH:MM:SS** bzw. **DD.MM.YYYY** hat:

```
proc checkTimeFormat { timeStr } {
  return [regexp ↷
  {^[0-9]{2}:[0-9]{2}:[0-9]{2}$} ↷
  $timeStr]
```

```
}

proc checkDateFormat { timeStr } {
  return [regexp ↷
  {^[0-9]{2}.[0-9]{2}.[0-9]{4}$} ↷
  $timeStr]
}
```

Führende Null entfernen und hinzufügen
Für die spätere Addition und Subtraktion der Zeiten muss man ggf. die führende Null in einer Zeit entfernen, damit man ordentlich damit rechnen kann. Hierfür dient die Methode **delLeadingZero**:

```
proc delLeadingZero { line } {
  regsub "^0" $line "" newline
  return $newline
}
```

Nach der Addition oder Subtraktion der Zahl ohne führende 0 kann man aus gestalterischen Gründen die 0 wieder hinzufügen, wofür die Funktion **addLeadingZero** dient:

```
proc addLeadingZero { hours minutes↷
 seconds } {
  if { $hours >= 0 && $minutes >= 0↷
   && $seconds >= 0 } {
    if { $hours < 10 } {
      set hours "0$hours"
    }
    if { $minutes < 10 } {
      set minutes "0$minutes"
    }
    if { $seconds < 10 } {
      set seconds "0$seconds"
    }
    set realTime "$hours:$minutes:↷
    $seconds"
  } else {
    error "Error: Time is negative:↷
     $hours $minutes $seconds"
  }
  return $realTime
}
```

Neu ist hier der Aufruf von **error**. Effektiv ist das wie ein **puts**, gibt also das Argument auf der Konsole aus. Zusätzlich bricht ein Skript an dieser Stelle aber komplett ab und gibt auch aus, an welcher Stelle im Skript das Problem auftrat.

Diese Funktion sollte natürlich nur bei schwerwiegenden Fehlern benutzt werden, die nicht abgefangen werden können oder sollen, verdeutlicht in dem Beispiel aber zumindest die Benutzung.

Zeiten addieren und subtrahieren
Für die spätere Ausgabe der Zeiten muss die Differenz zwischen End- und Startzeit gebildet und das Ergebnis zu den bisherigen Zeiten addiert werden.

Die Zeiten liegen im Format **HH:MM:SS** vor (siehe oben) und werden nach der Überprüfung per **split** an den Doppelpunkten getrennt:

```
set timeList1 [split $timeStr1 :]
set timeList2 [split $timeStr2 :]
```

Danach werden jeweils die beiden Stunden, Minuten und Sekunden subtrahiert (bzw. addiert mit einem Plus anstelle des Minus):

```
set hours   [expr [delLeadingZero [↷
lindex $timeList2 0]]-[↷
delLeadingZero [lindex $timeList1 ↷
0]]]

set minutes [expr [delLeadingZero [↷
lindex $timeList2 1]]-[↷
delLeadingZero [lindex $timeList1 ↷
1]]]

set seconds [expr [delLeadingZero [↷
lindex $timeList2 2]]-[↷
delLeadingZero [lindex $timeList1 ↷
2]]]
```

Damit man mit den Werten rechnen kann, muss wie oben bereits erwähnt ggf. die führende Null mithilfe von **delLeadingZero** entfernt werden.

Nachdem man auf Überträge geachtet hat (damit beispielsweise keine 72 Minuten irgendwo auftauchen) fügt man die führende Null wieder hinzu und erzeugt ein neue Zeit im gewohnten Format **HH:MM:SS**:

```
set realTime [addLeadingZero $hours↷
 $minutes $seconds]
```

Datum vergleichen und Zeiten berechnen
Eine Einschränkung hat das Skript: Es arbeitet nur korrekt, wenn Start und Ende einer Aufgabe am gleichen Tag sind. Ansonsten gibt es bei der Differenzbildung negative Stundenwerte. Daher überprüft die Funktion **checkSameDate**, ob die beiden Daten identisch sind.

Ist das der Fall, wird in **checkDateAndTime** die Zeit berechnet, die die Aufgabe gedauert hat:

```
proc checkDateAndTime { ↷
timeDateStr1 timeDateStr2 } {
  set realTime ""
  if { [checkSameDate [lindex ↷
  $timeDateStr1 0] [lindex ↷
  $timeDateStr2 0]] } {
    set realTime [diffTime [lindex ↷
    $timeDateStr1 1] [lindex ↷
    $timeDateStr2 1]]
  }
  return $realTime
}
```

Aufgabe prüfen

Bevor die Zeiten für jede Aufgabe gespeichert werden, sollte man überprüfen, ob das Format korrekt ist. Dies geht in der Methode **checkTask** wie zuvor über einen regulären Ausdruck:

```
if { [regexp {^[0-9]{4}.[0-9]{2}$} ↷
[lindex $task 0] ] } {
  # ...
}
```

Die Überprüfung ist aber noch zu unsicher, deshalb soll auch überprüft werden, ob die jeweilige Nummer zur Aufgabe passt. Dazu legt man eine globale Aufgabeliste an:

```
global tasklist
if { [info exists tasklist(↷
$taskNumber)] } {
  if { $tasklist($taskNumber) != [↷
  lindex $task 1] } {
    error "Error: ..."
    set taskNumber ""
  }
} else {
  set tasklist($taskNumber) [lindex↷
   $task 1]
}
```

Wenn es zu einer Aufgabennummer **taskNumber** noch keinen Eintrag im Array **tasklist** gibt, dann fügt man die aktuelle Bezeichnung (im **else**-Zweig) als Referenz hinzu. Sollte aber bereits ein Eintrag existieren, überprüft man diesen und gibt bei Nichtübereinstimmung einen Fehler aus.

Neue Aufgabe auswerten

Jetzt kommt der Knackpunkt des ganzen Skripts: die Methode **newTask**. Das sollte jetzt jedem bekannt vorkommen, denn dies kommt auch als Trennwort in der Aufgabedatei vor. Der Grund ist einfach, dass sich die Datei in diesem Format ganz einfach aus Tcl heraus ausführen lässt und dabei jedesmal die Funktion **newTask** mit drei Parametern für Aufgabe, Start- und Endzeit aufruft. Es reicht also aus, eine Funktion **newTask** zu schreiben, die die Auswertung der drei Parameter vornimmt. Auf die Art spart man sich das zeilenweise Auslesen der Datei.

Als erstes wird die Aufgabennummer extrahiert

```
set taskNumber [checkTask $task]
```

und danach die Differenz von End- und Startzeit gebildet:

```
set diff [checkDateAndTime ↷
$starttime $endtime]
```

Wie bereits bei der globalen Aufgabenliste **tasklist** (siehe oben) gibt es eine globale Zeitliste **timelist**. Anhand der Aufgabennummer **taskNumber** wird überprüft, ob es bereits eine Zeit zu dieser Aufgabe gab. Falls ja, wird die vorher berechnete Differenzzeit addiert. Im anderen Fall wird die Zeit als Startwert gesetzt:

```
global timelist

if { [info exists timelist(↷
$taskNumber)] } {
  set timelist($taskNumber) [↷
  addTime $timelist($taskNumber) ↷
  $diff]
} else {
  set timelist($taskNumber) $diff
}
```

Aufgaben und Zeiten ausgeben

Am Ende des Skripts sollen natürlich alle Aufgaben mitsamt der addierten Zeiten ausgegeben werden:

```
proc printTaskTimes { } {
  global timelist
  global tasklist
  if { [lsort [array names timelist↷
  ]] == [lsort [array names ↷
  tasklist]] } {
    set taskNumbers [lsort [array ↷
    names timelist]]
```

```
    foreach taskNumber $taskNumbers↷
     {
      puts "$taskNumber: $timelist(↷
      $taskNumber) $tasklist(↷
      $taskNumber)"
    }
  }
}
```

Zur Ausgabe iteriert man einfach über die Indizes der Aufgaben- bzw. Zeitliste. Damit das funktioniert, sollte man vorher überprüfen, ob die Indizes beider Arrays identisch sind. Damit ein Vergleich aber überhaupt möglich ist, sollte man die beiden Listen mit den Indizes vorher mit **lsort** sortieren.

Start des Skriptes

Am Ende (in der Datei bzw. am Anfang beim Abarbeiten) des Skriptes überprüft man dann die Anzahl der, an das Skript übergebenen, Argumente. Nur falls dieser Wert 1 ist, wird mittels

```
source [lindex $argv 0]
```

die übergebene Datei eingelesen und ruft damit implizit für jede neue Aufgabe die Methode **newTask** auf. Danach werden die Zeiten und Aufgaben ausgegeben (siehe oben).

Die (bisher noch nicht erwähnte) globale Variable **argv0** bei

```
printUsage $argv0
```

gibt dabei den Skriptaufruf an, also in diesem Beispiel **./check_tasks.tcl**.

Wenn man das Skript ausführbar gemacht hat, kann man es starten:

```
$ ./check_tasks.tcl tasks.txt
```

Die Ausgabe (anhand der Beispieldatei **tasks.txt**) sollte dann wie folgt aussehen:

```
1000.01: 29:05:00 Schule/Hausaufg.
1000.02: 05:54:42 Aushilfsjob
2000.01: 03:54:58 Kino
2000.02: 06:13:17 TV
2000.03: 02:58:54 PC
3000.01: 03:07:12 Karatetraining
4000.01: 02:56:37 Klavierunterricht
```

Zuerst kommen die Aufgabennummern, dann die addierten Zeiten und zum Schluss die Aufgabenbezeichnungen im Klartext.

Fazit

Natürlich hätte man die Aufgabedatei **tasks.txt** mit jeder anderen Skriptsprache auch auslesen und die Zeiten ermitteln können. Aufgrund des Formats konnte man sich aber das zeilenweise Auslesen sparen und die Datei als Tcl-Skript ausführen.

Insgesamt hat das kleine Beispiel ansatzweise gezeigt, was man mit Tcl machen kann, wobei dies aber nicht den Aufgaben gerecht wird. Tcl beherrscht daneben auch Namespaces, Dateibehandlung, Prozesskommunikation, Fehlerbehandlung und vieles mehr.

Als Literatur empfiehlt sich das Standardwerk „Tcl and the Tk Toolkit“ von John K. Ousterhout. In der überarbeiteten Version von 2009 sind noch einige neue Themen und Beispiele dazu gekommen. In der Regel findet man alle Antworten auf die eigenen Tcl-Fragen in diesem Buch.

Weitere Erweiterungen zu Tcl findet man im Tcler's Wiki [5]. Damit kann Tcl dann auch mit (weiteren) Bildformaten, Sound, Multithreading, XML oder Datenbanken umgehen. Wer sich intensiver mit Tk beschäftigen will, sollte sich die Seite TkDocs [6] anschauen.

LINKS

[1] http://www.tcl.tk/

[2] http://www.tcl.tk/software/tclpro/compiler.html

[3] http://www.freiesmagazin.de/freiesMagazin-2009-11

[4] http://de.wikipedia.org/wiki/Regulärer_Ausdruck

[5] http://wiki.tcl.tk/

[6] http://tkdocs.com/

Autoreninformation

Dominik Wagenführ ist Software-Entwickler und hat dabei auch täglich mit Tcl zu tun. Auf die Art hat er diese Skriptsprache schätzen gelernt.

Diesen Artikel kommentieren