Kleine Einführung in Tcl

Holger@Jakobs.com

19. April 2004

Inhaltsverzeichnis

1 Allgemeines

1.1 Warum Tcl/Tk?

Tcl steht für Tool Command Language, Tk für Tool Kit. Beide wurden und werden von John Ousterhout entwickelt, der auch Bücher über die Sprache geschrieben hat. Es handelt sich um eine interpretierte Scriptsprache, mit der sich viele Aufgaben sehr elegant und leicht lösen lassen. Nicht nur das hat sie gemein mit Perl und Python, den anderen beiden beliebten Scriptsprachen. Alle drei Sprachen haben weitere wichtige Eigenschaften:

- kostenlos im Quellcode verfügbar
- auf vielen Plattformen lauffähig (viele Unix-Versionen, Windows 95/98/NT/2000/XP, Mac OS 8, 9, X)
- erweiterbar, gutes Modul-/Package-Konzept
- leistungsfähige reguläre Ausdrücke
- Schnittstelle zu anderen Programmiersprachen, z. B. C/C++

Allerdings verfügt nur Tcl/Tk über das GUI-Toolkit Tk, das ja auch schon im Namen vorhanden ist. Genau dies fehlt den beiden anderen Sprachen, weshalb bereits Erweiterungen für diese geschrieben wurden, um Tk benutzen zu können. Da Tcl/Tk von C aus benutzt werden kann, ist dies auch relativ leicht gelungen. Trotzdem ist die Verknüpfung der eigentlichen Sprache mit dem GUI-Toolkit nur bei Tcl/Tk wirklich nahtlos, weil sie denselben Konzepten unterliegen. Aus genau diesem Grund hat sich der Autor dafür entschieden, sich mit Tcl/Tk zu befassen und nicht mit Perl oder Python. Für Perl und Python spricht die stärkere Objektorientierung, gegen Perl aber die sehr kryptische Syntax. Perl ist nicht orthogonal, d. h. für jedes Problem gibt es eine Reihe von Lösungen, weshalb „dasselbe“ Programm von verschiedenen Autoren sich noch stärker unterscheiden kann als bei anderen Sprachen.

Tcl ist von den Scriptsprachen die universellste, weil es sie nicht nur auf vielen Plattformen, sondern auch in diversen Arten gibt: normale Scriptsprache, compilierte Scriptsprache, in Webseite eingebettete Sprache (wie PHP), in C eingebettete Sprache (eingebetteter Interpreter), im Webbrowser (als Tclet, wie Java-Applet), als CGI-Scriptsprache. Für viele Zwecke muss man also nur eine Sprache mit einer Syntax erlernen.

1.2 Quellen

Tcl/Tk ist kostenlos, aber wie kommt man dran? Einerseits ist bei quasi allen Linux-Distributionen Tcl/Tk dabei und braucht bloß installiert zu werden, andererseits kann man an die neuesten Versionen (binär oder Quellcode) auch für andere Plattformen über das Internet kommen[1] Es gibt auch ein deutsches Portal: `http://www.self-tcl.de/`.

Viele Erweiterungen zu Tcl/Tk gibt es übrigens auch kostenlos, beispielsweise TkTable zur Darstellung von Tabellen wie bei einer Tabellenkalkulation und Tix, eine Sammlung von vielen weiteren Widgets. Nicht alle Erweiterungen sind auf allen Plattformen lauffähig, denn oft sind sie nur ein Nebenprodukt eines Entwicklungsauftrages, bei dem die Zeit für eine vom Autoren oder Kunden nicht benötigte Portierung fehlt.

Solange die Erweiterungen in Tcl selbst geschrieben sind, sind sie ja von vornherein nicht nur portabel, sondern direkt verwendbar, aber viele haben zusätzlich noch einiges an C- oder C++-Code, der portiert oder zumindest compiliert werden muss.

Die o. g. Adresse bei `activestate` zeigt übrigens auf die Variante „TclPro“, die auch einen Compiler, einen Debugger, einen Dialog-Editor und anderes enthält. All dies ist mittlerweile ebenfalls kostenlos erhältlich.

1.3 Aufruf

Tcl wird vom Interpreter `tclsh` (Tcl-Shell) interpretiert. Genau wie bei jeder anderen Shell muss der Interpreter aufgerufen und ihm das zu interpretierende Script übergeben werden. Das kann man durch konkreten Aufruf von `tclsh prog1.tcl` tun, wenn in `prog1.tcl` ein Tcl-Script steht. Man kann den Interpreter auch ohne Argument aufrufen und ein Script mit dem Kommando `source` starten:

```
$ tclsh
% source prog1.tcl
  ... Ausgabe von prog1 ...
% exit
$
```

$ stellt hier den Prompter der Unix-Shell dar, % ist der Standard-Prompter der Tcl-Shell.

Grundsätzlich kann man für kleine Experimente die `tclsh` oder die `wish` (Windowing Shell) auch interaktiv verwenden, allerdings sind sie nicht besonders komfortabel. Besser ist es, mit seinem Lieblingseditor (oder irgendeinem anderen) ein Script zu schreiben und dies auszuführen. Lediglich wer die `wish` auf Macintosh nutzt, freut sich über die lange ersehnte Kommandozeile! Dazu kommt dann noch die prima Sprache, mit der man

[1] `http://tcl.tk/`
`http://freshmeat.net/projects/activetcl/`
`http://sourceforge.net/projects/tcl`
`http://sourceforge.net/projects/tktoolkit`
`http://www.activestate.com/Products/TclPro`

endlich normal arbeiten kann, ohne sich dumm und dämlich zu klicken. Im Gegensatz zu AppleScript ist das Wissen um die Sprache auch auf anderen Plattformen nutzbar.

Die eleganteste Möglichkeit des Aufrufs auf einem Unix-System ist allerdings implizit, d. h. durch die auch bei anderen Interpretern verwendete Hash-Bang-Notation, die aus einem Script direkt ein ausführbares Kommando macht, das sogar vom `exec`-System-Call verwendet werden kann. Hierzu schreibt man den absoluten Pfad des Interpreters hinter die Zeichenkombination `#!`, die direkt am Anfang der Datei stehen muss – wirklich als die ersten beiden Bytes! Dann macht man die Datei ausführbar (siehe Kommando `chmod` in der Unix-Dokumentation, online durch das Kommando `man chmod` aufrufbar) und fertig ist das neue Kommando. Möchte man Tk verwenden, d. h. GUI-Scripts schreiben, so schreibt man statt des absoluten Pfads zur `tclsh` den Pfad zur `wish` in das Script. Bei Windows ist die Dateinamenerweiterung `.tcl` üblicherweise mit der Windowing Shell `wish` verbunden, so dass man dort nicht die Möglichkeit hat, die `tclsh` aufzurufen. Gegebenenfalls kann man die Assoziationen derart ändern, dass `.tcl`-Dateien mit der `tclsh` und `.tk`-Dateien mit der `wish` assoziiert werden. Interaktiv kann man übrigens beide benutzen, die `wish` hat sogar von vornherein einen guten History-Mechanismus.

Eine unter UNIX beliebte Alternative ist folgende: Man gibt als Interpreter zunächst die Bourne-Shell an und führt dann ein `exec` auf `tclsh` mit den Parametern `"$0"` und `"$@"` aus. Dies hat den Vorteil, dass man den absoluten Pfad zum Tcl-Interpreter nicht kennen muss, weil die Bourne-Shell den üblichen Suchpfad (`$PATH`) absucht. Um das `exec`-Kommando vor Tcl zu verstecken, schreibt man einen Kommentar davor, den man mit einem Backslash beendet. Für Tcl ist die folgende Zeile dann auch ein Kommentar, für die Bourne-Shell dagegen nicht.

So sieht der Anfang des Scripts dann aus:

```
#!/bin/sh
# starte Tcl-Interpreter \
exec tclsh "$0" "$@"
```

Wichtig ist, dass hinter dem \ kein weiteres Zeichen und zwischen dieser Zeile und dem `exec tclsh`... keine weitere Zeile steht – auch keine Leerzeile!

1.4 Referenz

Wenn man weiß, welche Tcl-Kommandos man verwenden muss, kann man Details schnell auf den mit dem Tcl/Tk-Paket installierten Man-Pages nachschauen. Üblicherweise befinden sie sich im Abschnitt `n`, d. h. man ruft die Manpage zum Tcl-Kommando `open` mit `man n open` oder `man -s n open` auf. Ob der Zusatz `-s` (für section) angegeben werden muss oder nicht, hängt von der Unix-Variante ab. Unter Windows gibt es eine Hilfedatei namens `tcl80.hlp` (für die Version 8.0). Um unter X einen ähnlichen Komfort in der Bedienung zu haben, kann man das Programm TkMan[2] verwenden oder die HTML-Variante der Tcl-Dokumentation benutzen, die in unserem BES-Portal[3] vorhanden ist.

[2] `ftp.cs.Berkeley.EDU in the /ucb/people/phelps/tcltk`
[3] http://www.bg.bib.de/portale/bes/Online-Doku/TclManual/

Ein bisschen problematisch ist es, wenn man zwar weiß, was man programmieren möchte, aber nicht, welches Kommando für diesen Zweck adäquat wäre. Genau um hier zu helfen, dient diese kleine Einführung, die in die Lage versetzen soll, ohne viel Sucherei die wichtigsten Sachen schnell zu überblicken und bald in den Online-Hilfen das Nötigste zu finden.

Um in Unix-Man-Pages zu blättern, bietet sich das Programm TkMan an, nicht nur für die Tcl/Tk-Manpages.

1.5 Entwicklungsumgebung

Für die Entwicklung von Tcl-Programmen bietet sich die Entwicklungsumgebung ASED[4] an, die selbst in Tcl/Tk geschrieben ist und vor allem sehr beim korrekten Einrücken des Codes hilft. Im b.i.b. ist sie bereits installiert.

Man kann aber natürlich auch jeden anderen Editor verwenden. Der im b.i.b. installierte SciTE rückt aber bei Tcl-Programmen völlig chaotisch ein und aus und ist daher weniger zu empfehlen.

Dem Tcl-Interpreter ist es übrigens völlig egal, ob die Zeilen mit Carriage Return und Newline oder nur mit einem der beiden Zeichen beendet werden. Allerdings ist die Unix-Shell da pingelig und verlangt, dass die Zeilen nur mit Newline getrennt werden, damit der Aufruf mittels `#!/bin/sh` funktioniert. Also hier bitte aufpassen!

2 Grundkonzept

Bei Tcl (natürlich auch bei Tk, aber das lassen wir ab jetzt der Kürze wegen weg) sind alle Kommandos Listen und umfassen genau eine Zeile, d. h. werden durch einen Zeilenumbruch begrenzt. Wenn man den Zeilenumbruch durch einen Backslash maskiert oder noch eine geschweifte Klammer offen ist, können Kommandos auch über mehrere Zeilen gehen. Das erste Element der Liste ist immer ein Kommando, alles andere sind Argumente für dieses Kommando. Gelegentlich – insbesondere bei Tk – treten Argumente paarweise auf, d. h. ein Argument fängt mit einem Minuszeichen an, das nächste dagegen nicht, denn es gibt einen Wert an. Sie bilden gemeinsam quasi ein einziges Argument, wobei der erste Teil einen Namen, der zweite Teil einen Wert darstellt.

Alle Kommandos sind Listen – auch die Zuweisung bildet hier (im Gegensatz zu der Unix-Shell[5]) keine Ausnahme. Sie geschieht nämlich nicht mit einem Gleichheitszeichen, sondern mit dem `set`-Kommando:

```
set variable wert
```

Der genannten Variablen wird der aufgeführte Wert zugewiesen. Es darf sich bei dem Wert auch nicht ohne weiteres um einen Ausdruck oder ein Kommando handeln, dessen Ergebnis zugewiesen werden soll. Dafür ist die Syntax zu schlicht. Um Ausdrücke zu berechnen, gibt es konsequenterweise wieder ein Kommando, nämlich `expr`. Um jetzt das Ergebnis eines Ausdrucks zuzuweisen, schreibt man z. B. folgendes:

[4] `http://www.mms-forum.de/ased`

[5] Mit Unix-Shell ist hier immer die Bourne-Shell `sh` gemeint bzw. dazu kompatible wie Korn- (`ksh`), POSIX- (meist `sh`) oder Bourne-Again-Shell (`bash`).

```
set ergebnis [expr 4 + 7]
```

Die eckigen Klammern geben an, dass es sich um ein Tcl-Kommando handelt, das zunächst ausgewertet werden muss. Gibt es geschachtelte eckige Klammernpaare, so werden sie von innen nach außen aufgelöst. Hier wird also das Kommando `expr 4 + 7` ausgewertet, d. h. der Ausdruck berechnet. Das Ergebnis dieses Ausdrucks, also `11`, wird an die Stelle des Kommandos gesetzt, so dass sich ein neues Kommando ergibt:

```
set ergebnis 11
```

Nun wird die Zuweisung ausgeführt und die Zeile ist komplett abgearbeitet. Nach dem hier dargestellten Strickmuster sind *sämtliche* Kommandos von Tcl aufgebaut – ohne Ausnahme. Diese Einfachheit und Eindeutigkeit ist eine der großen Stärken von Tcl. Es gibt keine kryptischen Operator-Orgien wie in Perl, dafür aber manchmal tief geschachelte (eckige und/oder geschweifte) Klammern.

3 Variablen

3.1 Skalare

Wie man einer Variablen einen Wert zuweist, haben wir schon gesehen. Wie verwendet man jetzt eine Variable? Man nennt ihren Namen mit einem führenden $-Zeichen, was an die Unix-Shell erinnert. Hier ein Beispiel:

```
set x 4
set y 7
set ergebnis [expr $x + $y]
```

Hier werden die beiden Variablen `x` und `y` erzeugt und mit Werten versehen, die dann im nächsten Kommando als Werte verwendet werden. Verschiedene Variablentypen gibt es in Tcl nicht – jede Variable kann Zahlen oder Zeichenketten enthalten, was sich auch im Laufe der Zeit ändern kann. Jeder numerische Wert kann auch logisch betrachtet werden, d. h. jeder von 0 verschiedene Wert ist logisch wahr. So erübrigt sich auch ein spezieller Datentyp für boolesche Werte. Allerdings gibt es Listen und Arrays, die anders sind als die skalaren Variablen. Am Namen erkennt man das im Gegensatz zu Perl aber nicht.

3.2 Listen

Listen sind nichts anderes als nach bestimmtem Muster formatierte Zeichenketten und können auch als solche verwendet werden. Das bedeutet, dass sie vom Tcl-Interpreter nicht streng unterschieden werden, so dass ein Wechsel zwischen Listeninhalt und Skalarinhalt bei einer Variablen problemlos ist. Listen sind durch Blanks getrennte Wörter. Gehören mehrere Wörter zu einem Listenelement, so werden sie mit geschweiften Klammern zusammengehalten. Diese Liste hat drei Elemente:

```
Hans {Anna Maria} Uta
```

Bei der Auswertung der Liste, z. B. dem Herauslösen der einzelnen Elemente, fallen die geschweiften Klammern automatisch weg. Durch geschweifte Klammern kann man auch Listen in Listen darstellen, d. h. Listenelemente können selbst wieder Listen sein.

3.3 Arrays

Arrays dagegen stellen schon einen anderen Typ dar, wenn sie auch nicht deklariert werden müssen. Bei der ersten Zuweisung wird festgelegt, ob eine Variable ein Skalar oder ein Array ist; das kann man anschließend auch nicht mehr ändern, es sei denn, man löscht die Variable mit `unset` und legt sie neu an. Ein Array wird durch Zuweisung auf eines seiner Elemente erzeugt:

```
set anzahl(hans) 7
```

Der Name des Arrays ist `anzahl`, aber der Index `hans` ist nicht der Name einer numerischen Variablen, sondern ein Zeichenkettenliteral. Man achte auch darauf, dass der Index in runden Klammern steht, nicht etwa in eckigen, denn die dienen ja der Kommandoverschachtelung – wie oben bereits gezeigt. Den Inhalt der Variablen kann man anzeigen lassen mit:

```
puts $anzahl(hans)
```

oder, wenn man den Namen `hans` in die Variable `name` hineinschreibt:

```
set name hans
puts $anzahl($name)
```

Das Kommando `puts` steht für `put string` und schreibt ein Argument als String unformatiert auf die Standardausgabe. Wenn man zwei Argumente übergibt, bezeichnet das erste die geöffnete Datei, auf die geschrieben wird, und das zweite gibt die zu schreibende Zeichenkette an.

Die Indexierung von Arrays ist nicht numerisch, sondern kann mit beliebigen Zeichenketten erfolgen – natürlich bei Bedarf auch mit ganzen Zahlen. Diese Art von Arrays nennt man assoziativ, weil sie eine Assoziation zwischen einem Bezeichner und einem Inhalt herstellen. Das ist oft sehr viel praktischer als numerisch indexierte Arrays, weshalb dieses Konzept auch in awk (da wurden sie erfunden) und Perl (hier heißen sie hashes) und JavaScript verwendet wird.

3.4 Zuweisung

Beim Kommando `set name hans` fällt auf, dass man dem Kommando `set` die Variable `name` ohne ein führendes Dollarzeichen übergibt. Ein Kommando braucht immer dann den *Namen* und nicht den *Wert* einer Variablen, wenn das Kommando an der Variablen etwas ändert, beispielsweise den Wert setzt oder an einen String etwas anhängt. Grund dafür ist, dass es in Tcl keinen „call by reference“ gibt, sondern lediglich einen „call by value“. Über den Namen greift die Prozedur dann indirekt auf den Inhalt zu.

Beim Herauslösen eines Teilstrings dagegen genügt ein Wert – ob er in einer Variablen oder einem Literal steht, ist gleichgültig. Manchmal ist es der Syntax eines Kommandos nicht direkt entnehmbar, ob man nun ein Dollarzeichen vor den Variablennamen schreiben soll oder nicht, d. h. ob man den Namen oder den Inhalt der Variablen an das Kommando übergibt. Eine kurze Überlegung, welchen Zweck das Kommando hat, d. h. ob es den Inhalt der Variablen ggf. verändert oder nicht, bringt meistens Klärung. Notfalls kann man es auch ausprobieren.

4 Ablaufsteuerung

Ein Tcl-Script wird für gewöhnlich von oben nach unten abgearbeitet – wie bei den meisten Programmiersprachen. Auch gibt es keine implizite Schleife wie in awk. Die üblichen Ablaufsteuerungen sind vorhanden: Verzweigung und Wiederholung in diversen Ausprägungen. Ein allgemeiner Hinweis für alle Kommandos, die viele geschweifte Klammern enthalten: Ein geschweiftes Klammernpaar stellt ein Listenelement dar, das von seinen benachbarten Listenelementen durch ein Blank getrennt werden muss. Daher muss zwischen einer schließenden und einer folgenden öffnenden Klammer unbedingt ein Leerzeichen stehen. Sonst ist die Zuordnung der Listenelemente falsch, was zu semantischen und Syntaxfehlern führt.

4.1 Einfach-Verzweigung

Beim `if`-Kommando wird ein Ausdruck genauso wie bei `expr` ausgewertet und sein Ergebnis als boolescher Wert interpretiert. Die Tatsache, dass alle Kommandos Listen sind, beeinflusst die Syntax allerdings etwas. Hier die Elemente der Liste, die ein `if`-Kommando darstellt:

- das Kommandowort `if`
- Bedingung, die geprüft wird
- wahlweise das Wort `then`
- Zweig, der bei erfüllter Bedingung ausgeführt wird
- wahlweise das Wort `else`
- wahlweise der Zweig, der bei nicht erfüllter Bedingung ausgeführt wird

Die als wahlweise gekennzeichneten Listenelemente können auch weggelassen werden. Die Wörter `then` und `else` sind lediglich schmückendes Beiwerk ohne jede Funktion. Damit die Zweige jeweils nur ein einziges Listenelement darstellen, müssen sie durch geschweifte Klammern zusammengehalten werden. Das sieht dann beispielsweise so aus:

```
if {$x > 4} {set x 0} {incr x}
```

Das kann man auch noch ausführlicher und strukturierter schreiben:

```
if {$x > 4} then {
  set x 0
} else {
  incr x
}
```

Wichtiger Hinweis: Wer es sich nicht schon bei C angewöhnt hat, die öffnende geschweifte Klammer hinter das `if`-Statement zu schreiben, wird sich hier umgewöhnen müssen!

Sonst funktioniert das Kommando nämlich nicht, sondern es fehlen die Elemente hinter der Bedingung.

Hier wird `$x` einmal als Wert verwendet, nämlich in der Bedingung. Daher das führende Dollarzeichen. In beiden Zweigen wird jeweils der Variablen*name* verwendet, denn die Kommandos verändern beide den Inhalt der Variablen `x`; daher ist ihnen mit dem Wert `$x` nicht gedient. Man kann das ein bisschen vergleichen mit der Übergabe von Zeigern auf Variablen an Funktionen in C, damit die Funktion den Wert der Variablen verändern darf. Allerdings benötigt man in C bei der Zuweisung keinen Zeiger; in Tcl dagegen muss man bei `set` den Namen ohne Dollarzeichen nehmen. Zeiger als solche gibt es in Tcl übrigens nicht.

4.2 Mehrfach-Verzweigung

Die aus C bekannte Mehrfach-Verzweigung mittels `switch` gibt es in Tcl auch. Hierbei kann ein Wert (im Gegensatz zu C) nicht nur auf exakte Übereinstimmung geprüft werden, sondern auch auf Ähnlichkeit mittels `glob`-Muster (d. h. so wie die Shell Dateinamenmuster generiert) oder regulärer Ausdrücke prüfen. Die Vergleichsverfahren werden auf den Manual-Pages `string` (Kommando `string match`) und `re_syntax` erläutert.

Die Syntax des `switch`-Kommandos kann man in diesem Beispiel erkennen:

```
switch -regexp "$var" {
  ^a.+b$  -
  b       {puts "Möglichkeit 1"}
  a+x     {puts "Möglichkeit 2"}
  default {puts "default-Zweig"}
}
```

Es wird der Inhalt der Variablen `$var` mit den regulären Ausdrücken verglichen, denn es wurde die Option `-regexp` verwendet. Neben dem regulären Ausdruck `^a.*b$` steht kein Codeblock, sondern nur ein Minuszeichen. Dies bedeutet, dass der Codeblock hinter dem folgenden regulären Ausdruck auch hier verwendet wird. Das Minuszeichen kann man auch als „oder“ lesen, wenn man es so formatiert:

```
switch -regexp "$var" {
  ^a.+b$ - b   {puts "Möglichkeit 1"}
  a+x          {puts "Möglichkeit 2"}
  default      {puts "default-Zweig"}
}
```

Als Optionen vor dem zu prüfenden Ausdruck sind möglich: `-exact` zur Prüfung auf exakte Übereinstimmung, `-glob` zum Vergleich wie Dateinamenmuster bei der Shell und `-regexp` zu Vergleich mit regulären Ausdrücken. Ohne Option wird exakt verglichen.

Sobald einer der Ausdrücke passt, wird der zugehörige Zweig ausgeführt und anschließend nach dem `switch` fortgesetzt. Ein `break` wie bei C gibt es also nicht.

4.3 Wiederholung

Es gibt in Tcl neben der `while`-Schleife auch die `for`-Schleife und eine besondere, die `foreach`-Schleife. Die `while`- und die `for`-Schleife sind in ihrer Bedeutung den gleichnamigen anderer Sprachen sehr ähnlich, die `foreach`-Schleife entspricht der `for`-Schleife der Unix-Shell (und der `foreach`-Schleife der unsäglichen C-Shell).

Die `while`-Schleife hat wie die `if`-Verzweigung eine Bedingung und dahinter ein Listenelement mit einem Anweisungsblock, aber natürlich gibt es keinen zweiten Anweisungsblock. Auch kann man hier keine schmückenden Zusatzworte verwenden, d. h. kein `do` oder ähnliches einfügen.

```
set y 3
while {$y < 10} {puts $y; incr y}
```

Das ist wieder so eine Kurzform, die ausgeschrieben wie folgt aussieht:

```
set y 3
while {$y < 10} {
  puts $y
  incr y
}
```

Wichtiger Hinweis: Wer es sich nicht schon bei C/C++ angewöhnt hat, die öffnende geschweifte Klammer hinter das `while`-Statement zu schreiben, wird sich hier umgewöhnen müssen! Sonst funktioniert die Schleife nämlich nicht, sondern es fehlt das dritte Element der Liste – der Anweisungsblock.

Die `for`-Schleife ist ähnlich wie in C keine reine Zählschleife, sondern es gibt ein Initialisierungskommando, ein Abbruchkriterium, eine Anweisung, die am Ende der Schleife ausgeführt wird, und einen Schleifenrumpf. In genau dieser Reihenfolge sind das die Argumente zum `for`-Kommando.

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

Das ist wieder so eine Kurzform, die ausgeschrieben wie folgt aussieht:

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

Das zweite Listenelement wird als Kommando ausgeführt, das dritte als logischer Ausdruck ausgewertet, das vierte und das fünfte wieder als Kommando ausgeführt. Auch hier ist es wichtig, dass die öffnende Klammer noch auf der Zeile mit dem `for`-Kommando steht, damit die Liste komplett ist.

Die dritte Schleifenform ist die `foreach`-Schleife, die für die Abarbeitung von Listen geschaffen wurde. Die einfachste Form ist

```
foreach i {rot gelb grün blau} {puts $i}
```

Die im ersten Argument genannte Variable `i` nimmt nacheinander alle Werte aus der Liste im zweiten Argument an. Jedes Mal wird die im dritten Argument angegebene Liste als Script abgearbeitet. Eine Besonderheit ist die paarweise (oder allgemeiner ausgedrückt tupelweise) Abarbeitung von Listen. Enthält eine Liste paarweise Werte, so kann man zwei Schleifenvariablen verwenden:

```
foreach {name gehalt} {hans 5000 uschi 5500 werner 4000} {
  puts "$name bekommt $gehalt"
}

hans bekommt 5000
uschi bekommt 5500
werner bekommt 4000

foreach {name grundg kinder} {hans 5000 3 uschi 5500 1 werner 4000 0} {
  puts "$name bekommt [expr $grundg + $kinder * 250]"
}
hans bekommt 5750.
uschi bekommt 5750.
werner bekommt 4000.
```

4.4 Ausnahmebehandlung

In Tcl Scripts können – wie in allen anderen Programmiersprachen auch – gelegentlich Fehler auftreten. Unbehandelte Fehler führen zum Abbruch des Scripts mit einer entsprechenden System-Fehlermeldung. Möchte man den Abbruch verhindern oder selbst steuern, wie die Meldung aussieht, verwendet man die Ausnahmebehandlung (exception handling) von Tcl. Dies ist ähnlich wie in C++ und Java sehr elegant. Verwendung findet es bei beispielsweise bei der Dateiverarbeitung, siehe Abschnitt 7.2.1 auf Seite 16.

Die Syntax ist `catch` *`Kommando Variable`*. Dabei wird das *`Kommando`* ganz normal ausgeführt und sein Ergebnis der *`Variable`*n zugewiesen. Man könnte also auch schreiben: `set` *`Variable`* `[`*`Kommando`*`]`. Im Fehlerfall, d. h. wenn das Kommando schiefgeht, wird statt seines Ergebnisses die Fehlermeldung in die Variable geschrieben. Beispiel:

```
set x 73
set y keineZahl
set z [expr $x * $y]
puts $z
```

Das gibt einen Syntaxfehler – also Programmabbruch –, weil `$y` keine Zahl ist. Zur Ausgabe von `z` kommt es gar nicht mehr. Also fangen wir den Fehler ab mit:

```
set x 73
set y keineZahl
catch {expr $x * $y} z
puts $z
```

Jetzt enthält die Variable z statt des nicht zu ermittelnden Rechenergebnisses die Fehlermeldung. Bei der Ausgabe wird kontrolliert die Meldung ausgegeben; das Programm bricht nicht ab.

Damit man nicht nur über den Inhalt der Ergebnisvariablen feststellen kann, ob ein Fehler aufgetreten ist, stellt das catch-Kommando auch einen logischen Wert dar, der für eine Verzweigung verwendet werden kann. Anwendungsbeispiel:

```
set x 73
set y keineZahl
if [catch {expr $x * $y} z] {
  puts "Es ist ein Fehler aufgetreten\n$z"
} else {
  puts "Das Berechnungsergebnis lautet $z"
}
```

Das ist folgendermaßen zu verstehen: Falls ein Fehler auftaucht, dann wird die Fehlermeldung ausgegeben. Andernfalls läuft das Programm normal weiter. Bei schweren Fehlern, bei denen man das Programm nicht fortsetzen kann, ist der else-Zweig entbehrlich:

```
set x 73
set y keineZahl
if [catch {expr $x * $y} z] {
  puts "Schwerer Fehler: $z"
  exit 1
}
puts "Das Berechnungsergebnis lautet $z"
```

In Prozeduren (siehe Abschnitt 10 auf Seite 26) kann statt exit zum Programmabbruch ggf. auch return zur Rückkehr in die rufende Prozedur verwendet werden.

5 Ausgabeformatierung

Die Ausgabeformatierung wurde zum größten Teil von C übernommen, vom berühmten printf-statement. Allerdings heißt es hier format und formatiert den String lediglich, ohne ihn aber auszugeben. Daher wird zusätzlich immer noch ein puts-Kommando benötigt:

```
set name Hans
set personalnr 1233
puts [format "%-10s /%5d" $name $personalnr]
Hans      / 1233
```

Hier wird das in eckigen Klammern eingeschachtelte format-Kommando zuerst ausgeführt. Es formatiert die beiden Werte $name und $personalnr entsprechend den Vorgaben als linksbündige, 10 Zeichen lange Zeichenkette, gefolgt von einem Leerzeichen, einem Slash und einer fünfstelligen, rechtsbündig angeordneten Ganzzahl. Da der Wert

nur 4 Stellen umfasst, wird ein Leerzeichen vorweg ausgegeben. Der formatierte String ist der Rückgabewert des `format`-Kommandos, der wiederum als einziges Argument dem `puts`-Kommando übergeben wird, das ihn auf die Standardausgabe schreibt.

6 Kommentare

In Tcl beginnen Kommentare – wie in vielen Scriptsprachen – mit einem Lattenzaun: #. Sie gehen jeweils bis zum Ende der Zeile. Zeilenübergreifende Kommentare gibt es nicht, jede Kommentarzeile muss wieder mit einem # beginnen. Im Gegensatz zu anderen Scriptsprachen wird # allerdings nur zu Beginn eines Kommandos als Kommentarzeichen akzeptiert. Man darf also nicht einfach hinter einem Kommando ein # setzen und hoffen, dass der Rest der Zeile ignoriert wird. Kommentare benötigen eine eigene Zeile, oder sie werden vom vorigen Kommando zusätzlich durch ein Semikolon ; abgetrennt. Mit einem Semikolon kann man auch mehrere Kommandos in einer Zeile trennen. Obiges Beispiel zum `if`-Kommando kann man also so kommentieren:

```
if {$x > 4} then { ;# Bedingung
  set x 0
} else { ;\# sonst
  incr x
} ;# if
```

Vorsicht! Tcl ignoriert Kommentarzeilen leider nicht vollständig. Insbesondere Klammern in Kommentaren können zu großen Problemen führen, wenn sie nicht paarweise verwendet werden. Das ist zwar ziemlich blödsinnig, weil ein Kommentar eigentlich *keine* Wirkung haben sollte, aber der Interpreter ist halt so gestrickt, dass der Lattenzaun nur ein einfaches Kommando darstellt, die Klammern aber dennoch dessen Argumente auch über Zeilengrenzen hinweg zusammenfassen. Dieses Verhalten wird oft in Newsgroups diskutiert, ist aber bis auf weiteres so hinzunehmen. Man kann also folgendes nicht als Kommentarzeile schreiben, weil die öffnende geschweifte Klammer nicht geschlossen wird:

```
# while a != 0 { ;# erzeuge FEHLER!
```

7 Ein- und Ausgabe

Natürlich kann Tcl nicht nur mit direkt zugewiesenen Werten arbeiten, sondern auch Werte von der Standardeingabe einlesen oder mit Dateien arbeiten. Da gibt es keinerlei Einschränkungen – auch die Kommunikation über TCP/IP-Sockets ist möglich, so dass man verteilte Anwendungen schreiben kann, die völlig plattformunabhängig sind, sogar einschließlich der GUI-Programmierung (mit Tk).

7.1 Standard-Ein- und Ausgabe

Mit `puts` gibt man zeilenweise aus, mit `gets` liest man zeilenweise ein. Möchte man Daten auf einen anderen Ausgabekanal als die Standardausgabe schicken, so gibt man ihn als erstes Argument an. Das zweite Argument enthält dann den auszugebenden Wert.

```
puts stderr "So geht es nicht!"
```

gibt die Meldung auf dem Standardfehlerkanal aus. In auszugebende Zeichenketten kann man auch Variablen einbauen, solange die Zeichenkette durch Anführungszeichen begrenzt wird und nicht durch geschweifte Klammern.

```
set x 7
puts "x hat den Wert $x"
puts {x hat den Wert $x}
```

würde folgendes ausgeben:

```
x hat den Wert 7
x hat den Wert $x
```

denn die geschweiften Klammern verhindern die Interpretation des Wertes `$x`. Ihre Funktion entspricht also den Hochkommas bei der Unix-Shell, während die Funktion der Anführungszeichen bei der Unix-Shell und Tcl gleich ist. Auch kann man bei Anführungszeichen Kommandos in eckigen Klammern in die Zeichenkette einbauen:

```
puts "x hat den Wert [expr $x + 1]"
puts {x hat den Wert [expr $x + 1]}
```

würde folgende Ausgabe erzeugen:

```
x hat den Wert 8
x hat den Wert [expr $x +1]
```

Die geschweiften Klammern verhindern wieder die Auswertung, so dass die Zeichenkette völlig unverändert ausgegeben wird.

Dem Kommando `gets` muss *immer* der Name eines Eingabekanals (offene Datei) mitgegeben werden, beispielsweise `stdin` für die Standardeingabe. Gibt man keine Variable an, so wird die eingelesene Zeile als Wert zurückgegeben.

```
set zeile [gets stdin]
```

und

```
gets stdin zeile
```

sind weitgehend identisch, allerdings gibt auch die zweite Variante einen Wert zurück, nämlich die Anzahl Zeichen, die eingelesen werden konnte. Daher kann man sie noch erweitern:

```
set laenge [gets stdin zeile]
```

wodurch man leicht prüfen kann, ob überhaupt Daten eingelesen werden konnten, denn sind keine Daten verfügbar, wird -1 zurückgegeben. Andernfalls bekommt die Variable `laenge` die Anzahl eingelesener Zeichen zugewiesen.

7.2 Datei-Ein- und -Ausgabe

Natürlich kann man auch beliebig Dateien anlegen, öffnen, lesen, schreiben, schließen, löschen, umbenennen usw. Das muss allerdings explizit geschehen. Tcl enthält auch Mechanismen zum Abfangen von Fehlern, wenn beispielsweise eine Datei, die zum Lesen geöffnet werden soll, nicht existiert. Verwendet man diese Mechanismen nicht, wird das Script mit einer Fehlermeldung abgebrochen. Eine Datei öffnet man mit dem `open`-Kommando, dem man den Namen der zu öffnenden Datei und einen Modifier für die Art des Zugriffs übergibt (Lesezugriff ist default). Directories werden in Dateinamen bei Tcl am besten plattformunabhängig mit Slashes getrennt, nicht mit dem plattformspezifischen Backslash von Windows und auch nicht mit dem Doppelpunkt von Macintosh. Näheres zur Konstruktion von Dateinamen auf der Manpage von `filename`. Laufwerksangaben gibt es bei Unix natürlich nicht. Die Zugriffsmodifier werden in Unix-Manier angegeben, aber wahlweise in der Form der stdio-Bibliothek (high-level calls, ANSI-C) oder der io-Bibliothek (low-level calls, POSIX). Der Rückgabewert des `open`-Kommandos ist der Filehandle für die geöffnete Datei, der bei den folgenden Lese- und Schreiboperationen und beim Schließen wieder benötigt wird. Hier einige Beispiele:

Tcl-Kommando	**Auswirkung**
`set f [open "beispiel.txt" r]`	öffnet die Datei zum Lesen, muss existieren
`set f [open "beispiel.txt"]`	identisch, denn Lesen ist default, muss existieren
`set f [open "beispiel.txt" r+]`	öffnet die Datei zum Lesen und Schreiben, muss existieren
`set f [open "beispiel.txt" w]`	öffnet die Datei zum (Über-) Schreiben
`set f [open "beispiel.txt" w+]`	öffnet die Datei zum Lesen und Schreiben, wird ggf. überschrieben
`set f [open "beispiel.txt" a]`	öffnet die Datei zum Anhängen
`set f [open "beispiel.txt" a+]`	öffnet die Datei zum Lesen und Schreiben, Position am Ende

Nun einige Beispiele mit POSIX-Modifiern:

Tcl-Kommando	**Auswirkung**
`set f [open "beispiel.txt" RDONLY]`	öffnet die Datei zum Lesen
`set f [open "beispiel.txt" WRONLY]`	öffnet die Datei zum Schreiben
`set f [open "beispiel.txt" RDWR]`	öffnet die Datei zum Lesen und Schreiben

In allen diesen POSIX-Fällen muss die Datei bereits existieren. Um sie ggf. zu erzeugen, gibt man zusätzlich den Modifier `CREAT` an. Damit die beiden Modifier dann gemeinsam

eine Liste bilden, muss man sie mit geschweiften Klammern zu einer solchen zusammenfügen:

```
set f [open "beispiel.txt" {RDWR CREAT}]
```

Es gibt noch weitere Modifier, die man zusätzlich angeben kann:

Modifier	Auswirkung
APPEND	vor jedem Schreibzugriff wird ans Dateiende positioniert.
EXCL	nur gemeinsam mit CREAT zu verwenden. Datei darf nicht bereits existieren.
TRUNC	Dateiinhalt wird gelöscht (abgeschnitten).

Die genaue Bedeutung der Modifier sind im POSIX-Standard definiert und auch in der Dokumentation zum open-System-Call eines jeden POSIX-konformen Unix nachzulesen. Bei Verwendung auf anderen Betriebssystemen kann die Funktionalität eventuell eingeschränkt sein, wenn sie nicht komplett unterstützt wird. Schließlich bedient sich Tcl des Betriebssystems.

7.2.1 Fehler abfangen beim Öffnen von Dateien

Möchte man, dass das Script nicht abbricht, wenn es einen Fehler beim Öffnen der Datei gibt – z. B. wenn sie nicht existiert oder bei Verwendung von CREAT und EXCL schon existiert, so muss man das catch-Kommando verwenden, siehe auch Abschnitt 4.4 auf Seite 11.

```
catch {open "label.tcl" RDONLY} f
```

Das erste Argument von catch ist ein Stück Tcl-Code, der ausgeführt wird. Sein Ergebnis wird der Variablen zugewiesen, die im zweiten Argument steht, hier f. Falls das Kommando schiefgeht, so bekommt f nicht das Ergebnis der Operation zugewiesen, sondern die (englische) Fehlermeldung. Die Prüfung, ob es geklappt hat oder nicht, geschieht am einfachsten mit einem if-Kommando, denn catch liefert einen Wahrheitswert zurück: TRUE, wenn es einen Fehler abgefangen hat, FALSE, wenn kein Fehler aufgetreten ist.

```
if [catch {open "beispiel.txt" RDONLY} f] {
  puts stderr $f
} else {
  gets $f zeile
  puts $zeile
  close $f
}
```

Üblicherweise steht hinter if ein logischer Ausdruck in geschweiften Klammern. Hier aber steht ein Kommando in eckigen Klammern. Da die eckigen Klammern, die ein

Sub-Kommando umschließen, bereits ihren Inhalt zu einem Argument zusammenfassen, sind die geschweiften Klammern entbehrlich. Man benötigt sie aber, wenn man die Bedingung mit einem Ausrufezeichen negieren will. So könnte man das Kommando also umstellen:

```
if {![catch {open "beispiel.txt" RDONLY} f]} {
  gets $f zeile
  puts $zeile
  close $f
} else {
  puts $f
}
```

Übersichtlicher ist diese Variante hier allerdings nicht, weshalb die erste vorzuziehen ist.

7.2.2 Dateiinhalt komplett verarbeiten

Möchte man eine Datei komplett lesen und ausgeben, so kann man das auf zwei verschiedene Arten machen. Entweder liest man Zeile um Zeile, bis man das Dateiende erreicht hat, oder aber man liest die gesamte Datei ein und gibt sie auf einen Schlag wieder aus. Die erste Variante ist speicherplatzsparender, die zweite schneller, setzt aber voraus, dass entsprechend viel Hauptspeicher verfügbar ist.

```
if [catch {open "beispiel.txt" RDONLY} f] {
  puts $f
} else {
  while {1} {
    gets $f zeile
    if [eof $f] break
    puts $zeile
  }
  close $f
}
```

Die Schleife ist eine bauchgesteuerte, was auch notwendig ist. Das Dateiende wird erst dann erkannt, wenn man versucht, über das Dateiende hinaus zu lesen. Ist der Leseversuch fehlgeschlagen, so darf keine Ausgabe mehr erfolgen, sondern die Schleife muss sofort verlassen werden, was mit dem `break`-Kommando geschieht. Diese Funktionalität ist außer mit der Schleife, die das Abbruchkriterium in der Mitte trägt, nur durch wiederholte Kommandos (Vor- und Nachlesen) oder doppelte Abfragen möglich, was beides aus Gründen der Strukturierung abzulehnen ist.

```
if [catch  {open "beispiel.txt" RDONLY} f] {
  puts $f
} else {
  set alles [read -nonewline $f]
  puts $alles
```

```
  close $f
}
```

Diese Variante liest mit einem `read`-Kommando den gesamten Dateiinhalt in die Variable `alles` ein, wozu entsprechend viel Hauptspeicher belegt wird, ggf. also auch Mega- und Gigabytes. Die Option `-nonewline` beim `read`-Kommando bewirkt, dass das letzte Newline-Zeichen am Ende der Datei nicht in die Variable `alles` eingetragen wird. Da das `puts`-Kommando sowieso ein Newline am Ende anfügt, würden sonst evtl. zwei Newlines ausgegeben. So wie es hier programmiert ist, würde ein fehlendes Newline am Dateiende ergänzt, ein vorhandenes nicht verändert. Wenn man eine Datei nicht mehr benötigt, sollte man sie umgehend mit dem Kommando `close` schließen – wie in obigen Beispielen gezeigt. Andernfalls wird sie am Ende des Programms automatisch geschlossen.

7.2.3 Datei-Direktzugriff

Alle Plattendateien sind automatisch Direktzugriffsdateien. Da alle von Tcl unterstützten Betriebssysteme Dateien als (mindestens einen) Strom von Bytes verwalten und nicht zwischen Direktzugriffsdateien und sequentiellen Dateien unterscheiden, gibt es hier keine Probleme. Diese träten dann auf, wenn man versuchte, Tcl auf MPE, MVS, OS/400 oder ähnliche Betriebssysteme zu portieren; dann wären Tcl-Programme nicht mehr voll übertragbar. Um den Schreib-Lese-Zeiger einer Datei zu verändern, verwendet man das Kommando `seek`. Argumente sind der Filehandle, die numerische Position und eine Angabe, ob die Position relativ zum Anfang (`start`), zur aktuellen Position (`current`) oder zum Ende (`end`) gemeint ist. Default ist Anfang. Um den Schreib-Lese-Zeiger um 50 Bytes zurückzupositionieren, schreibt man also

```
seek $f -50 current
```

Um die Position aufs Dateiende zu stellen, mit folgenden Schreiboperationen also an die Datei anzuhängen, schreibt man

```
seek $f 0 end
```

Dies waren die Grundlagen im Zusammenhang mit Datei-Ein- und Ausgabe; natürlich gibt es weitere Kommandos in diesem Zusammenhang, siehe vor allem Kommando `file` in der Dokumentation. Da geht es dann um die Existenzprüfung von Dateien, die Ermittlung, ob es sich bei einer Datei um eine gewöhnliche Datei oder ein Verzeichnis handelt, um das Kopieren, Löschen und Umbenennen von Dateien usw.

7.2.4 Temporäre Dateien

Braucht man – ausnahmsweise – einmal temporäre Dateien für Zwischenergebnisse, so sind diese erstens im Verzeichnis für temporäre Dateien und zweitens mit einem eindeutigen Dateinamen zu erstellen. Schließlich kann man nicht sicher sein, dass das Tcl-Script zu jeder Zeit nur ein einziges Mal läuft.

Das Verzeichnis für temporäre Dateien ist betriebssystemabhängig, bei Unix ist es `/tmp`, bei Windows oft (aber nicht immer) `c:/temp`. Um einen eindeutigen Dateinamen zu erzeugen, baut man die aktuelle Prozessnummer ein, beispielsweise so: `set tempname "/tmp/beispiel_[pid].tmp"`

Bitte auch Dateien in den speziellen Verzeichnissen für temporäre Dateien ordnungsgemäß schließen und beim Programmende löschen.

7.2.5 Datei- und Directory-Operationen

Es gibt weitere Datei- und Directory-Operationen, die gelegentlich sehr hilfreich sind. Hier eine kleine Übersicht:

Kommando	Bedeutung
`glob`	Erzeugen einer Dateinamenliste aus einem Verzeichnis, ähnlich wie die Bourne-Shell das tut.
`file`	Eine komplexes Kommando ähnlich `string` (siehe Abschnitt 9 auf Seite 23), d. h. man muss immer eine Unterfunktion auswählen. Hier eine Liste der am häufigsten verwendeten Unterfunktionen:
	`atime` Ermittlung der letzten Zugriffszeit (access time)
	`attributes` Ermittlung der Datei-Attribute (plattformabhängig)
	`copy` Kopieren von Dateien. Also bitte nicht selbst öffnen, lesen und schreiben!
	`delete` Löschen von Dateien
	`dirname` Entfernen des letzten Slashs und der Zeichen dahinter, so dass nur der Directory-Pfad übrigbleibt
	`executable` Ermitteln, ob eine Datei ausführbar ist (wie `test -x` in der Shell)
	`exists` Prüfen, ob ein Dateiname existiert
	`extension` Extrahieren der Dateinamenserweiterung
	`join` Zusammensetzen mehrerer Namensbestandteile zu einem Pfad; arbeitet plattformunabhängig (Gegenteil: `split`)
	`mkdir` Erzeugen eines Verzeichnisses
	`mtime` Ermittlung der letzten Dateiänderung (modification time)
	`rename` Umbenennen einer Datei (wie `mv` in der Shell)
	`size` Ermittlung der Dateigröße
	`split` Zerlegen eines Pfades in Bestandteile (Gegenteil: `join`)
	`tail` Ermittlung des Dateinamens ohne Pfad

Tabelle 1: Datei- und Directory-Kommandos

8 Listenoperationen

In Tcl sind zunächst einmal *alle Kommandos* Listen, aber auch in der Datenverwaltung lassen sich Listen hervorragend einsetzen. In früheren Tcl-Versionen waren Listen recht langsam, wenn sie wuchsen, aber die Zeiten sind vorbei. Ab Tcl 8.0 sind Listen genauso schnell wie Arrays – unabhängig von ihrer Größe. Listen sind eigentlich nichts anderes als speziell formatierte Zeichenketten. Genauer gesagt, alle Listenelemente, die nicht genau ein Wort darstellen, sind in geschweifte Klammern gepackt. Ebenso wenn Listenelemente wiederum Listen sind, stehen sie auch in geschweiften Klammern. Daraus ergibt sich, dass jedes Listenelement, das aus mehreren Wörtern besteht, tatsächlich wiederum als Liste betrachtet werden kann, aber das macht ja nichts. Das Kommando für die Erzeugung von Listen ist `list`, für die Verarbeitung gibt es u. a. `lappend`, `lindex`, `linsert`, `llength`, `lrange`, `lreplace`, `lsearch`, `lsort`, `concat` und `split`, siehe auch Tabelle 2. `list` gibt alle seine Argumente, zusammengefasst zu einer Liste, zurück. Dabei kümmert es sich automatisch um das richtige Setzen von geschweiften Klammern und ggf. auch Backslashes, denn Fehler dabei führen oft zu falschen Ergebnissen oder unerwartetem Verhalten des Programms. Bei der Konstruktion von Listen sollte man also *unbedingt* `list` verwenden und *keinesfalls* versuchen, diese selbst mit Zeichenkettenoperationen zusammenzubauen.

Kommando	**Bedeutung**
`list`	Erzeugung von Listen aus einzelnen Elementen
`lappend`	Anhängen von neuen Elementen an eine Liste, besonders performant!
`lindex`	Herauslösen eines einzelnen Elements aus einer Liste
`linsert`	Einfügen eines oder mehrerer Elemente in eine Liste
`llength`	Ermitteln der Anzahl Elemente einer Liste
`lrange`	Herauslösen einer Teilliste, die aus einem oder mehreren Elementen bestehen kann
`lreplace`	Ersetzen eines oder mehrerer Elemente durch eine Reihe neuer Elemente
`lsearch`	Durchsuchen einer Liste nach Elementen, die auf ein glob-Muster oder einen regulären Ausdruck passen
`lsort`	Sortieren einer Liste
`concat`	Zusammenfügen von mehreren Listen zu einer neuen Liste
`split`	Zerlegt eine Zeichenkette in eine Liste anhand von Trennzeichen, die in der Zeichenkette enthalten sind, siehe Anwendungsbeispiel in Abschnitt 9.1 auf Seite 24

Alle Kommandos außer `lappend` geben eine neue Liste als Ergebnis zurück.

Tabelle 2: Listenkommandos

```
set l1 [list rot grün]
set l2 [list $l1 gelb blau]
```

Welches Ergebnis bringt jetzt wohl eine Ausgabe der Liste?

```
puts $l2
{rot grün} gelb blau
```

Die Liste `$l1` ist das erste Element der neuen Liste, also müssen die beiden Teile von `$l1`, `rot` und `grün`, irgendwie zusammengehalten werden. Darum kümmert sich `list` automatisch. Hängen wir an die Liste noch einmal die Liste `$l1` an und geben noch einmal aus:

```
lappend l2 $l1
puts $l2
{rot grün} gelb blau {rot grün}
```

Die Liste hat nun 4 Elemente, die wir alle der Reihe nach ausgeben wollen, wenn auch etwas umständlich mit einer Zählschleife statt einer `foreach`-Schleife. Das Kommando `lindex` löst ein einzelnes Listenelement heraus, wobei die Numerierung mit `0` beginnt. Auf das letzte Element kann man auch mit `end` zugreifen, statt den numerischen Index des letzten Elements anzugeben.

```
for {set i 0} {$i < [llength $l2]} {incr i} {
  puts [lindex $l2 $i]
}
rot grün
gelb
blau
rot grün
```

Das einzelne Listenelement hat hier keine geschweiften Klammern mehr – sie wurden von der Listenoperation, hier `lindex`, automatisch wieder entfernt. Man muss sich also nicht darum kümmern, wenn man ein Listenelement herauslöst. Schauen wir uns die einzelnen Listenelemente noch einmal als Listen an und lassen uns ihre Länge zeigen:

```
for {set i 0} {$i < [llength $l2]} {incr i} {
  set element [lindex $l2 $i]
  puts "Die Liste $element hat [llength $element] Elemente."
}
Die Liste rot grün hat 2 Elemente
Die Liste gelb hat 1 Elemente
Die Liste blau hat 1 Elemente
Die Liste rot grün hat 2 Elemente
```

Möchte man nicht nur einzelne Elemente einer Liste haben, sondern Teile einer Liste, so verwendet man `lrange`, dem neben der Liste selbst noch zwei Argumente übergeben werden:

```
puts [lrange $l2 1 end]
gelb blau {rot grün}
```

Wie bei `lindex` kann auch hier `end` anstatt des numerischen Index des letzten Elements verwendet werden. Auf diese Weise kann man sehr leicht wie hier eine Liste ohne ihr erstes Element erhalten. Mit `linsert` kann man Elemente in eine bestehende Liste einfügen, wobei eine neue Liste generiert wird, die man wiederum einer Variablen zuweisen kann. Als Argumente gibt man eine Liste, einen numerischen Index (oder `end`) und beliebig viele weitere Argumente an. Letztere werden vor das Element mit dem genannten Index eingefügt.

```
set l3 [linsert $l2 2 neu1 neu2 neu3 $l1]
puts $l3
{rot grün} gelb neu1 neu2 neu3 {rot grün} blau {rot grün}
```

Der neuen Variablen `l3` wird die aus der bestehenden Liste `$l2`, in die vor dem Element mit dem Index `2` die Elemente `neu1 neu2 neu3` und die bestehende Liste `$l1` eingefügt wurden, konstruierte Liste zugewiesen. Das Element `blau` hat den Index `2`, also wird vor ihm eingefügt. Die Einfügung besteht aus vier Elementen, von denen das letzte wieder um eine Liste ist. So ergibt sich die neue Liste. Die Variable `l2` wird bei der Operation nicht verändert – das ist auch gar nicht möglich, weil nicht der Variablenname (`l2`), sondern nur der Inhalt (`$l2`) übergeben wurde. Nun können wir eine Liste auch sortieren lassen. Auch hierbei wird eine neue Liste erzeugt, nicht die bestehende verändert:

```
puts [lsort $l3]
blau gelb neu1 neu2 neu3 {rot grün} {rot grün} {rot grün}
puts $l3
{rot grün} gelb neu1 neu2 neu3 {rot grün} blau {rot grün}
```

Üblicherweise sortiert `lsort` nach dem Code der Maschine. Möchte man numerisch oder ohne Berücksichtigung von Groß- und Kleinschreibung sortieren, so muss man entsprechende Optionen verwenden. Man kann sogar selbst eine Funktion angeben, die den Vergleich bewerkstelligt; allerdings ist das meist deutlich langsamer. Andererseits ist die korrekte Verarbeitung von Umlauten nach den deutschen Orthographieregeln durch die Standardoptionen nicht abgedeckt. Das wäre auch zuviel verlangt, weil diese Regeln sprachabhängig sind und sich nicht allgemeingültig auf einen Zeichensatz beziehen.

Mit `lsearch` kann man feststellen, ob eine Liste ein bestimmtes Element enthält. Dabei kann exakt gesucht werden oder auch nach regulären Ausdrücken. Falls ein Element gefunden wird, liefert lsearch den Index zurück, andernfalls `-1`. Das erste Argument von `lsearch` ist der wahlfreie Suchmodus `exact`, `glob` (default) oder `regexp`, die weiteren Argumente sind die zu durchsuchende Liste und der zu suchende Ausdruck. Beim Suchmodus `glob` werden dieselben Regeln angewendet wie beim Kommando `string match`, bei `regexp` dagegen die Regeln des Kommandos `regexp`.

```
puts [lsearch {eins zwei drei vier} {*r*}]
2
```

Es wird der Index des ersten Listenelementes ausgegeben, das ein "r" enthält. `lsearch` verwendet beim Überprüfen auf Übereinstimmung (Matching) also die Regeln, die bei

der Shell für Dateinamen gelten. Wahlweise kann man mittels Optionen auch einstellen, dass die Zeichenkette exakt passen muss oder dass richtige reguläre Ausdrücke verwendet werden:

```
puts [lsearch -exact {eins zwei drei vier} "dr"]
-1
```

Es gibt kein Listenelement, das exakt „dr“ lautet, also wird -1 ausgegeben.

```
puts [lsearch -regexp {eins zwei drei vier} "^v.*e"]
3
```

Der Index des ersten Listenelements, das am Anfang ein „v“ hat und irgendwo dahinter ein „e“, ist das mit dem Index 3.

Es gibt kein direktes Kommando zum Löschen einzelner Listenelemente, sondern eines zum Ersetzen: `lreplace`. Hier gibt man die zu verändernden Liste, die Indexe des ersten und des letzten zu ersetzenden Elementes und anschließend die statt dessen einzusetzenden Elemente an. Rückgabe ist die veränderte Liste Beispiel:

```
set l1 [list Anna Berta Fritz Hans Xaver]
set l2 [lreplace $l1 2 3 Kevin Marc Bill]
puts $l2
Anna Berta Kevin Marc Bill Xaver
```

`lreplace` ersetzt in der übergebenen Liste die deutschen Namen durch englische; die so entstandene Liste wird `l2` zugewiesen. Gibt man keine neu einzusetzenden Elemente an, wird der Bereich durch nichts ersetzt, d. h. gelöscht:

```
set l3 [lreplace $l2 1 1]
puts $l3
Anna Kevin Marc Bill Xaver
```

Bitte darauf achten, dass die Numerierung der Elemente mit `0` beginnt, d. h. „Anna“ hat nach wie vor den Index `0` (siehe `for`-Schleifenbeispiel in Abschnitt 8 auf Seite 21).

9 Zeichenkettenoperationen

Die Möglichkeiten von Tcl zur Zeichenkettenverarbeitung sind sehr reichhaltig. Da alle Variablen als Zeichenketten betrachtet werden können und sogar Listen aller Art in Zeichenketten gespeichert werden, muss dies auch so sein. Die meisten Kommandos verbergen sich als Unterkommandos von `string`, aber beispielsweise `append` ist ein eigenständiges Kommando. Das Anhängen an Zeichenketten mit `append` ist schnell und ressourcenschonend, weil die Zeichenkette nicht umkopiert werden muss. Erstes Argument ist der Name der zu verändernden Zeichenkettenvariablen, alle weiteren Argumente sind die anzuhängenden Zeichenketten. Gerade bei großen Zeichenketten ist das schneller als die Zuweisung von zwei verketteten Zeichenketten, was hier im Beispiel `Franz` gezeigt wird.

```
set name1 Hans
append name1 ", geb. am 12.12.1955"
set name2 Franz
set name2 "$name2, geb am 1.1.1955"
```

Weitere Information über das umfangreiche `string`-Kommando mit all seinen Optionen findet man in der Online-Dokumentation zu Tcl. Darüber hinaus haben noch folgende Kommandos mit Zeichenketten zu tun: `split` spaltet eine Zeichenkette in eine Liste, `regexp` und `regsub` finden bzw. ersetzen reguläre Ausdrücke in Zeichenketten. In diesen ist ein großer Teil der Leistungsfähigkeit von Tcl zu sehen.

9.1 Anwendungsbeispiele für `split`

Wenn man eine Textdatei komplett eingelesen hat, steht der gesamte Dateiinhalt in einer einzigen Zeichenkette. Meist möchte man diese dann Zeile für Zeile verarbeiten. Dazu zerlegt man sie mit Hilfe von `split` in eine Liste aus Zeilen:

```
set f [open "beispiel.txt" r]
set inhalt [read $f]
close $f
foreach zeile [split $inhalt \n]  {
  puts $zeile ;# Verarbeitung Zeile für Zeile
}
```

Auch das Zerlegen einer Zeile, die aus mehreren durch Doppelpunkt (oder ähnlichen Trennzeichen) getrennten Feldern besteht, ist ein häufiger Einsatz von `split`:

```
set zeile {34534435:44:Kugelschreiber:250}
set liste [split $zeile :]
puts "Artikelnummer ..: [lindex $liste 0]"
puts "Lagerbestand ...: [lindex $liste 1]"
puts "Bezeichnung ....: [lindex $liste 2]"
puts "Preis in Cent ..: [lindex $liste 3]"
```

9.2 Anwendungsbeispiel für `regexp`

Beim Suchen von regulären Ausdrücken in Zeichenketten verwendet man `regexp`. Als Wert liefert es 0 oder 1 – je nachdem, ob der Ausdruck in der Zeichenkette gefunden wurde oder nicht. Darüber hinaus kann man sich auch die gefundene Teilzeichenkette liefern lassen. Suchen wir beispielsweise alle „Meiers“ (in allen Schreibweisen) aus der passwd-Datei, so programmieren wir:

```
set f [open "passwd" r]
set inhalt [read $f]
close $f
foreach zeile [split $inhalt \n]  {
  if [regexp {M[ae][iy]er[^:]} $zeile name] {
    puts "gefundener Name: $name"
```

```
  } ;# if
} ;# foreach
```

Die regulären Ausdrücke können in runden Klammern angegebene Unter-Ausdrücke enthalten, die man sich ebenfalls liefern lassen kann. Möchte man von den obigen „Meiers“ die Vor- und Zunamen haben, die mit Semikolon abgetrennt sind, so programmiert man:

```
set f [open "passwd" r]
set inhalt [read $f]
close $f
foreach zeile [split $inhalt \n]  {
  if [regexp {:(M[ae][iy]er); ?([^:]):} $zeile dummy famname vorname] {
    puts "Vorname: $vorname, Familienname: $famname"
  } ;# if
} ;# foreach
```

Der gefundene Gesamtausdruck ist hier uninteressant, weshalb er einer Variablen namens `dummy` zugewiesen wird. Wichtig sind für uns jetzt Familienname und Vorname, die als geklammerte Unterausdrücke im regulären Ausdruck stehen und den Variablen `famname` und `vorname` zugewiesen werden.

9.3 Anwendungsbeispiele für `regsub`

Im Gegensatz zu `regexp` verändert `regsub` die Zeichenkette und schreibt sie dann in eine neue Variable (oder auch dieselbe, wenn sie als Ziel angegeben wird). Im allgemeinen wird nur eine Ersetzung durchgeführt. Möchte man denselben Ausdruck in der ganzen Zeichenkette ersetzen, so muss man die Option `-all` verwenden.

Als Beispiel sollen einer Datei Zeilenrücklaufzeichen am Ende jeder Zeile eingefügt werden, d. h. sie soll vom Unix-Format ins DOS-Format geändert werden.

```
# Einlesen
set f [open "datei.unix" r]
set inhalt [read $f]
close $f
# Ändern
regsub -all \n $inhalt \r\n inhalt
# Ausgeben
set f [open "datei.dos" w]
puts -nonewline $f $inhalt
close $f
```

Jetzt sollen alle Textstellen einer HTML-Datei, die mit dem Tag <b> formatiert sind, auf das Markup geändert werden – einschließlich der Ende-Tags natürlich.

```
# Einlesen
set f [open "alt.html" r]
set inhalt [read $f]
```

```
close $f
# Ändern
regsub -all {<b>(.*?)</b>} $inhalt {<em>\1</em>} inhalt
# Ausgeben
set f [open "neu.html" w]
puts -nonewline $f $inhalt
close $f
```

Hierbei wird ein in Klammern angegebener Unterausdruck (subexpression) innerhalb des regulären Ausdrucks verwendet. Der erste geklammerte Unterausdruck kann im Ersatztext mit `\1` wieder eingesetzt werden (der zweite hieße `\2` usw.). Um diese Zeichenkette herum werden nun die neuen emphasize-Tags gesetzt.

10 Prozeduren

Sobald Programme etwas größer werden, benötigt man Prozeduren zur Strukturierung. Diese werden von Tcl auch bereitgestellt, allerdings ist die Handhabung von Variablenübergaben deutlich anders als bei anderen Sprachen. Alle Übergaben geschehen als Werte. Um eine Variable veränderbar zu übergeben, reicht man ihren Namen an die Prozedur. Die Prozedur verschafft sich dann Zugang zum Namensraum der aufrufenden Prozedur und greift darüber auf den Inhalt auch schreibend zu. Globale Variablen sind in Prozeduren nicht ohne weiteres verfügbar. In Prozeduren angelegte Variablen sind lokal und verlieren ihre Gültigkeit mit dem Ende der Prozedur. Beides kann man mit dem Zusatz `global` beeinflussen, denn eine bereits existierende Variable, die in einer Prozedur mit `global` definiert wird, steht in der Prozedur zur Verfügung. Noch nicht existierende Variablen werden auf dieselbe Weise erzeugt und bleiben auch nach Prozedurende bestehen.

Auch das Kommando zu Erzeugen einer Prozedur ist eine Liste. Das erste Element ist das Wort `proc`, das zweite der Prozedurname, das dritte eine Liste von Argumenten und das vierte ein Stück Tcl-Code, der Prozedurrumpf.

```
proc doppel {x} {return [expr 2 * $x]}
```

Das ist eine einfache Prozedur, die einen Wert verdoppelt und wieder zurückgibt. Es wird keine formale Unterscheidung gemacht zwischen wertliefernden (in Pascal Funktionen genannt) und nicht-wertliefernden Prozeduren (in Pascal Prozeduren, in C void-Funktionen genannt). Längere Prozeduren schreibt man der Übersichtlichkeit wegen meistens nach diesem Muster:

```
proc doppel {x} {
  return [expr 2 * $x]
}
```

Die Argumente können auch default-Werte erhalten, dadurch werden sie dann wahlfrei:

```
proc doppel {{x 0}} {
  return [expr 2 * $x]
}
```

Aufrufe sind so möglich:

```
puts [doppel 4]
set x 99
set x [doppel $x]
puts $x
puts [doppel]
8
198
0
```

Das sieht hier ein wenig merkwürdig aus, aber es ist so, dass alle Argumente, die default-Werte haben sollen, als Liste aus zwei Elementen dargestellt werden, nämlich Argumentname und default-Wert. Da hier nur ein Argument vorliegt, liegen die umschließenden Klammern der Argumentliste direkt an.

Möchte man eine Variable direkt verändern, also keinen Wert zurückliefern, schreibt man die Verdoppelungsprozedur so:

```
proc doppel {x} {
  upvar $x xx
  set xx [expr 2 * $xx]
}
```

Das Kommando `upvar` bewirkt, dass die Variable `xx` dem in `$x` enthaltenen Variablen*namen* gleichgesetzt wird. Die Variable `xx` wird also zu einem Alias der in `$x` genannten Variable aus dem Namensraum der rufenden Prozedur. Der Aufruf geschieht dann nicht mehr mit einem Wert, sondern mit einem Variablen`namen`. Also ist als Literal auch kein numerisches Literal mehr erlaubt, da ein Variablenname mit einem Buchstaben beginnen muss. Der verwendete Mechanismus ist „call by name“ im Gegensatz zu den bekannten „call by value“ und „call by reference“.

```
set a 14
doppel a ;# hier KEIN Dollarzeichen, weil die Variable a verändert wird!
puts $a
28

doppel 12
bad level "12"
```

Das Literal `12` ist schließlich nicht verdoppelbar, denn es repräsentiert keine Variable, sondern nur einen fixen Wert. Ein weiteres Problem tritt auf, wenn der übergebene Variablenname gar nicht existiert:

```
doppel gibtesnicht
can’t read "xx": no such variable
```

Die Variable namens `$gibtesnicht`, für die ein Alias gebildet werden soll, gibt es nicht. Allerdings kann man solche Fälle auch abfangen, indem man die Existenz der Variablen prüft. Das ist ja gerade der Vorteil von Scriptsprachen, dass dies möglich ist:

```
proc doppel {x} {
  upvar $x xx
  if [info exists xx]  {
    set xx [expr 2 * $xx]
 } else {
   puts stderr "Die übergebene Variable $x existiert nicht."
 }
}
```

Ein Aufruf bringt es an den Tag:

```
doppel gibtesnicht
Die übergebene Variable gibtesnicht existiert nicht.
```

11 Bibliotheksfunktionen

Tcl kennt so ziemlich alle Bibliotheksfunktionen, die es auch in C gibt. Beispielsweise kann man leicht auf das aktuelle Datum und auf die Zeit zugreifen und Datums-/Zeitangaben gut formatieren mit `clock`.

Viele Datei-Funktionen verstecken sich hinter `file`, siehe Tabelle 1 auf Seite 19. Dateien nach Namensmuster suchen kann man mit `glob`.

Nur unter Windows gibt es eine „Registry“. Auf diese kann mann zugreifen und dort Werte lesen und schreiben mit dem Paket `registry`.

12 Weitergehendes

Tcl kann noch viel mehr als das in dieser kleinen Einführung beschriebene. Insbesondere gibt es noch folgendes:

Packages: In Packages kann man wiederverwendbaren Code ablegen, der von allen oder nur bestimmten Applikationen genutzt werden kann. Auch eine Versionsverwaltung – das was Windows bei den DLLs fehlt – ist vorhanden, ähnlich den Versionen bei den Unix-Shared-Libraries.

Namespaces: Mit Namespaces (Namensräumen) kann man verhindern, dass es Namenskollisionen gibt, wenn man viele Packages verwendet. Jeder Namespace hat seine eigenen globalen Variablen, deren Lebensdauer die des Programms ist. Das Konzept ist ähnlich dem von C++.

Socket-Programmierung: Über Sockets kann man Anwendungen über TCP/IP-Netze kommunizieren lassen. In Tcl wurden schon ganze Webserver und Webbrowser sowie Mail-Frontends geschrieben.

GUI: Mit dem Tk aus Tcl/Tk kann man multiplattformfähige GUI-Applikationen schreiben. Auf Unix (X) und Windows gibt es einen erweiterten Satz von Widgets mit dem ebenfalls kostenlosen Zusatzpaket Tix und mit TclPro.

Safe-Tcl: Mit sicheren Tcl-Interpretern kann man auch Tcl-Scripts aus dubiosen Quellen ablaufen lassen, denn alle Kommandos, die gefährlich sein könnten, sind abgeschaltet. Dies ist ähnlich dem Sandbox-Modell von Java.

Tclets: Tclets sind ähnlich Applets in Java in einem Browser ablauffähig, benötigen allerdings das Browser-Plug-In, weshalb es eher für Intra- als für Internet-Anwendungen geeignet erscheint.

mod_dtcl bzw. mod_tcl: Tcl kann als Modul in den Apache Web Server eingebaut werden, so das Tcl-Kommandos in Webseiten eingebettet werden können, ähnlich wie bei PHP. Als CGI-Sprache kann es natürlich auch verwendet werden.

Schnittstellen: Es existieren Aufrufschnittstellen, um Routinen der Tcl-Library aus C-Programmen heraus aufzurufen, aber auch, um C-Routinen aus Tcl heraus zu benutzen, d. h. die Tcl-Sprache um eigene, in C geschriebene Kommandos zu erweitern, die man in einer Shared Library ablegt.

Datenbanken: Mit Tcl kann man auf viele Datenbanken direkt (PostgreSQL, Oracle, MySQL) und auf alle anderen mit ODBC zugreifen.

incr Tcl: Inzwischen ist sogar die erweiterte Version von Tcl, genannt „incr Tcl“ – analog zu C++ – zum Open Source geworden. Es bringt Tcl Objektorientierung und bessere Unterstützung größerer Projekte. „incr Widgets“ ist eine Sammlung von „Mega-Widgets“, „incr Tk“ ist ein Framework für die Erstellung von „Mega-Widgets“. All das steht jetzt kostenlos zur Verfügung.

Für Interessierte sei auf jeden Fall empfohlen, nach dem Stichwort „Tcl“ im Web zu suchen. Es gibt überwältigend viel Material hierzu.

URL: http://www.bg.bib.de/portale/bes/Scripting/Tcl/Einfuehrung_Tcl.pdf
$RCSfile: Einfuehrung_Tcl.tex,v $ $Date: 2004/04/19 06:54:54 $ $Revision: 1.3 $