Grundlagen der Tool Command Language

Holger Jakobs – holger@jakobs.com

2017-05-15

Inhaltsverzeichnis

1 Der Start mit Tcl

1.1 Warum Tcl/Tk?

Tcl steht für Tool Command Language, Tk für Tool Kit. Beide wurden von John Ousterhout entwickelt, der auch Bücher über die Sprache geschrieben hat. Die Pflege hat das Tcl Core Team übernommen. 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 sowie anderen beliebten Scriptsprachen. Wichtige Eigenschaften:

- kostenlos im Quellcode verfügbar
- auf vielen Plattformen lauffähig (Unix/Linux/BSD, Windows, Mac)
- erweiterbar, gutes Modul-/Package-Konzept
- leistungsfähige reguläre Ausdrücke
- Schnittstelle zu anderen Programmiersprachen, z. B. C/C++
- quasi beliebig lange Zeichenketten, keine Fehler durch Überlaufen
- keine Zeiger und keine Zeigerarithmetik

Als Besonderheit verfügt Tcl/Tk über das GUI-Toolkit Tk, das ja auch schon im Namen vorhanden ist. Genau dies fehlt anderen anderen Sprachen, weshalb bereits Erweiterungen für diese geschrieben wurden (TkInter, Perl/Tk, PyTk), 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. Für Perl und Python haben sich viele Entwickler entschieden, gegen Perl spricht 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. Python hat an Beliebtheit zugenommen, aber dass die Länge von Whitespaces eine Bedeutung trägt und die Inkompatibilität von Versionen vor 3 mit denen ab 3 sprechen noch immer stark dagegen.

Tcl ist von den Scriptsprachen die universellste, weil es sie nicht nur auf vielen Plattformen, sondern auch in diversen Arten gibt: 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. Mit der AndroWish[1] gibt es auch eine Implementation für das Betriebssystem Android auf Smartphones und Tablets.

[1] http://androwish.org

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 über den Paketmanager installiert zu werden (sofern das nicht bereits bei der Installation des Systems erfolgt ist), andererseits kann man an die neuesten Versionen (binär oder Quellcode) auch für andere Plattformen über das Internet kommen[2]. Bei Mac OS X ist es bereits vorinstalliert.

Viele Erweiterungen zu Tcl/Tk gibt es übrigens auch kostenlos, beispielsweise Tablelist[3] zur interaktiven Verwendung von Tabellen in GUI-Anwendungen, aber auch welche, die nichts mit Tk zu tun haben, beispielsweise für die Kommunikation per UDP[4] (TCP/IP ist im Kern bereits eingebaut). 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. Oft gibt es eine Version, die Maschinencode verwendet, sofern dieser vorhanden ist, und andernfalls auf eine reine Tcl-Implementation zurückgreift. Letztere ist dann weniger performant, aber funktionsidentisch und garantiert verfügbar.

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:

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

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

Grundsätzlich kann man für kleine Experimente die `tclsh` oder die `wish` (Windowing Shell) auch interaktiv verwenden, allerdings sind sie nicht besonders komfortabel. Für interaktive Experimente bietet sich daher die Tk-Konsole an: `tkcon`. Oft ist es besser, mit seinem Lieblingseditor ein Script zu schreiben und dies auszuführen.

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 ei-

[2] http://tcl.tk/
[3] http://nemethi.de
[4] http://sf.net/projects/tcludp/

nem Script direkt ein ausführbares Kommando macht, das unmittelbar vom `exec`-System-Call verwendet werden kann. Hierzu schreibt man den absoluten Pfad[5] 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 benutzen kann man übrigens beide.

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.

Die erste Zeile der Tcl-Datei darf nicht mit Windows-Zeilenenden (CR + LF)[6] versehen sein, sondern nur mit Unix-Zeilenenden (LF).

So sieht der Anfang des Scripts dann aus:

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

puts "Hello, World!"
```

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

Eine Alternative hierzu, die auch von vielen empfohlen wird, ist die folgende.

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

puts "Hello, World!"
```

Natürlich kann in beiden Fällen statt der `tclsh` auch die `wish` oder sogar ein `tclkit`[7] verwendet werden.

Das Kommando `puts` steht für `put string` und schreibt ein Argument als String unformatiert auf die Standardausgabe. Wenn man zwei Argumente übergibt, bezeichnet das

[5] Den bekommt man mittels `type -p tclsh` bzw. `type -p wish` heraus, sofern die Programme im Suchpfad `PATH` vorhanden sind.

[6] CR = carriage return, LF= line feed, siehe eine beliebige ASCII-Tabelle.

[7] http://wiki.tcl.tk/52 oder http://equi4.com/tclkit/

erste die geöffnete Datei, auf die geschrieben wird, und das zweite gibt die zu schreibende Zeichenkette an. Im obigen Beispiel wird durch die Anführungsstriche die Zeichenkette zusammengehalten, so dass `puts` nur ein Argument sieht.

Wenn ein Script als Kommandozeilenscript – also mit `tclsh` oder `tclkit` – gestartet wurde, kann man trotzdem mittels `package require Tk` das GUI-Toolkit nachladen. Auf diese Weise kann man Anwendungen schreiben, die nur bei Bedarf ein GUI öffnen und bei einer reinen Textverbindung – beispielsweise über SSH ohne X-Window – im Textmodus verbleiben. Das funktioniert allerdings nur auf unixoiden Systemen, weil bei Windows prinzipielle Unterschiede zwischen Konsolen- und GUI-Anwendungen bestehen, die zur Laufzeit nicht überbrückt werden können.

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` oder `3tcl`, d. h. man ruft die Manpage zum Tcl-Kommando `open` mit `man -s n open` oder `man -s 3tcl 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 `tcl8`*x*`.hlp` (für die Version 8.x). Alternativ dazu kann man auch im Web nachschauen unter http://www.tcl.tk/man/tcl/.

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 dieses Dokument als Grundlage, die in die Lage versetzen soll, ohne viel Sucherei die wichtigsten Sachen schnell zu überblicken und bald in den Online-Hilfen alles Nötige zu finden.

Das Wiki unter `http://wiki.tcl.tk` ist auch eine gute Quelle.

1.5 Entwicklungsumgebung

Für die Entwicklung von Tcl-Programmen bietet sich die Entwicklungsumgebung ASED[8] an, die selbst in Tcl/Tk geschrieben ist und vor allem sehr beim korrekten Einrücken des Codes hilft. Allerdings wurde ASED seit Ende 2004 nicht mehr weiter entwickelt. Wer bereits mit einer anderen Entwicklungsumgebung vertraut ist, die eine Sprachunterstützung für Tcl anbietet, kann auch diese nutzen. Notfalls tut es jeder beliebige Texteditor, denn alle guten Editoren haben auch einen Tcl-Modus mit Syntax-Highlighting. Für Eclipse[9] gibt es eine Erweiterung für dynamische Sprachen (DLTK), das auch die Entwicklung mit Tcl und Tk unterstützt. Die kleine IDE Geany [10] hat ebenfalls einen Modus für Tcl und Tk.

Wer eine kommerzielle Entwicklungsumgebung verwenden möchte, kann sich Komodo[11]

[8] `http://www.tcl-home.de/ased/asedmain.htm`
[9] http://eclipse.org
[10] http://geany.org
[11] http://komodoide.com/

anschauen, das von ActiveState[12] angeboten wird, die auch Tcl/Tk-Distributionen für diverse Plattformen anbieten[13].

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. Zeichenketten im Quellcode müssen in der Systemcodierung hinterlegt werden, sofern beim Start des Interpreters kein anderes Encoding angegeben wird[14]. Andernfalls könnten Zeichen jenseits des ASCII-Codes falsch dargestellt werden, beispielsweise Zeichen mit Akzent, Umlaute, Zeichen aus nicht-lateinischen Alphabeten.

Tcl „weiß“, ob es auf der Konsole oder auf der GUI etwas ausgibt, und verwendet daher unter Windows auf der Konsole (`cmd.exe`) in einem deutsch lokalisierten Windows die Codepage 850, auf der GUI dagegen die Codepage 1252, aber man ist jeweils auf die in der Codepage enthaltenen Zeichen beschränkt. Da die meisten Linux-Distributionen durchgehend UTF-8 verwenden, gibt es die Problematik nicht, und es stehen immer sämtliche Unicode-Zeichen zur Verfügung.

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. Zeichenketten im Quellcode müssen in der Systemcodierung hinterlegt werden, sofern beim Start des Interpreters kein anderes Encoding angegeben wird. Anderfalls könnten Zeichen jenseits des ASCII-Codes falsch dargestellt werden, beispielsweise Zeichen mit Akzent, Umlaute, Zeichen aus nicht-lateinischen Alphabeten.

2 Grundkonzept und Syntax

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 oder ein Anführungszeichen 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. Gelegentlich gibt es auch Optionen, die nur aus einem Wort mit einem Minuszeichen am Anfang bestehen, aber keinen Wert haben. Ein Beispiel hierfür wäre die Option `-nonewline` des Kommandos `puts`, die die Ausgabe des Newline-Zeichens am Ende der Ausgabe unterdrückt.

Da alle Kommandos Listen sind, bildet auch die Zuweisung hier (im Gegensatz zu der Unix-Shell[15]) keine Ausnahme. Sie geschieht nämlich nicht mit einem Gleichheitszeichen,

[12] http://www.activestate.com/

[13] http://www.activestate.com/activetcl-pro-studio

[14] Dies geschieht durch die Option `-encoding` *`encoding-name`* vor dem Script-Dateinamen.

[15] Mit Unix-Shell ist hier immer die Bourne-Shell `sh` gemeint bzw. dazu kompatible wie Korn- (`ksh`),

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, denn *Tcl selbst kann nicht rechnen*. Um Ausdrücke zu berechnen, gibt es konsequenterweise wieder ein Kommando, nämlich **`expr`**. Um jetzt das Ergebnis eines Ausdrucks zuzuweisen, schreibt man z. B. folgendes:

```
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
```

Oft wird empfohlen, den Ausdruck hinter **`expr`** in geschweifte Klammern[16] zu setzen: **`expr {4 + 7}`**, denn das verhindert einerseits die doppelte Auswertung von evtl. hinter **`expr`** vorhandenen Variablen und auch die Umwandlung von Zahlen in Zeichenketten und zurück. Genau dies würde den Programmablauf ungefähr um den Faktor 3 verlangsamen. Ggf. kann die doppelte Auswertung sogar zu Sicherheitsproblem führen.

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 Operatorfolgen wie in Perl, dafür aber manchmal tief geschachelte (eckige und/oder geschweifte) Klammern.

Es folgen ein paar kurze Hinweise auf Sonderzeichen. Später folgen ausführlichere Erläuterungen und mehr Beispiele dazu.

2.1 Anführungsstriche

Paarweise Anführungsstriche (`"`) fassen Zeichenfolgen mit Leerzeichen zusammen. Interpretationen finden innerhalb aber durchaus statt, also werden Variablen (mit vorangestelltem Dollarzeichen) aufgelöst und Unterkommandos (in eckigen Klammern) ausgeführt. Auch Escape-Sequenzen mit Backslash werden durch die jeweiligen Zeichen ersetzt, also **`\n`** durch ein Newline-Zeichen, **`\t`** durch ein Tabulatorzeichen und **`\u`*`XXXX`*** durch das Zeichen des angegebenen Unicode-Codepoints usw.

Sehr praktisch ist es, wenn man Variablen in Texte einbauen möchte, denn man braucht sie bloß zu nennen und nicht umständlich mit dem Rest des Textes zu verketten.

POSIX- (meist **`sh`**) oder Bourne-Again-Shell (**`bash`**).

[16] siehe Abschnitt 2.2 auf der nächsten Seite

```
set name Fritz
puts "Hallo $name,\nwie geht's?"
```

2.2 geschweifte Klammern

Paare geschweifter Klammern fassen wie Anführungsstriche Zeichenfolgen mit Leerzeichen zusammen, aber es finden keinerlei Interpretationen statt, sondern alles bleibt völlig unverändert. Einzige Ausnahme ist, dass ein mittels Backslash gequotetes Newline, gefolgt von beliebigem Whitespace durch ein einzelnes Leerzeichen ersetzt wird. Oft enthalten geschweifte Klammern viele weitere Kommandos, beispielsweise als der zweite Parameter eines `if`-Kommandos, oder auch Ausdrücke wie der erste Parameter des `if`-Kommandos.

2.3 eckige Klammern

Der Inhalt von Paaren eckiger Klammern wird als Kommando ausgeführt und durch sein Ergebnis ersetzt. In der Unix-Shell kennt man das von Backticks bzw. der Schreibweise `$(...)`. Diese Klammern werden wie üblich von innen nach außen aufgelöst. Von diesem Syntaxkonstrukt wird sehr häufig Gebrauch gemacht, auch gerne geschachtelt. Es entspricht im wesentlichen der Kommandoverkettung von Java und C++.

2.4 Zeichen ohne Sonderbedeutung in Tcl

In fast allen Sprachen haben das Hochkomma „`'`“ und das Gleichheitszeichen „=“ Sonderbedeutung, in Tcl dagegen nicht. Das Hochkomma begrenzt *keine* Zeichenketten und das Gleichheitszeichen dient *nicht* zur Zuweisung. Zwei Gleichheitszeichen hintereinander werden an entsprechenden Stellen zum Vergleichen von Werten verwendet.

3 Variablen und Datenstrukturen

Um es ganz klar vorweg zu sagen: Bei Tcl gibt es keine verschiedenen, vorher festgelegten Datentypen für Variablen. Jede Variable kann jederzeit jeden beliebigen Datentypen annehmen, als da wären:

- Boolesche Werte – Jeder Zahlenwert kann als logischer Wert interpretiert werden. Alle von 0 verschiedenen Werte sind logisch `true`, nur 0 ist `false`.
- Ganzzahlige Werte – Jede Ziffernfolge, ggf. mit Vorzeichen, kann als Ganzzahl interpretiert werden. Hierbei ist der Länge der Ziffernfolge keine praktische Grenze gesetzt, wobei Zahlen, die in 64 bits passen, als `int` angesehen werden, größere als `bignum`.
- Gebrochenzahlige Werte – Alle Zahlen, die keine reine Ziffernfolgen sind, sondern einen Dezimalpunkt oder eine Exponentialschreibweise enthalten, können als `double`-Werte interpretiert werden. Intern wird die IEEE754-Darstellung verwendet.

- Listen – Listen, siehe Abschnitt 3.3 auf Seite 13, sind geordnete, ggf. geschachtelte Strukturen von Elementen. Viele Zeichenketten können als Listen interpretiert werden.
- Dictionaries – Dictionaries, siehe Abschnitt 3.5 auf Seite 21, sind geordnete, ggf. geschachtelte Strukturen aus Schlüssel-Wert-Paaren. Viele Zeichenketten können als Dictionaries interpretiert werden.
- Zeichenketten – Alles kann immer auch als Zeichenkette interpretiert werden.

Alle diese (Um-)Interpretationen werden bei Bedarf durchgeführt. Wird beispielsweise eine Variable auf `3.6e10` gesetzt, so ist das zunächst eine Zeichenkette und wird auch genau so wieder ausgegeben. Addiert man 0 oder 0.0 hinzu, so muss sie als eine `double`-Zahl interpretiert werden, so dass das Ergebnis nicht mehr als `3.6e10`, sondern als `36000000000.0` ausgegeben wird.

Um (nur für Debugging-Zwecke!) beobachten zu können, in welcher Form eine Variable derzeit im Interpreter vorliegt, kann man dies mit `::tcl::unsupported::representation` *`wert`* erfragen.

Wenn eine Variable in einem numerischen Kontext verwendet wird, schaut der Interpreter, ob es eine aktuelle numerische Darstellung gibt. Wenn nein, wird die Zeichenkette entsprechend geparst und als `int`-, `bignum`- oder `double`-Wert abgelegt, so dass diese Interpretation bei nachfolgenden Berechnungen nicht erneut durchgeführt werden muss.

Ähnliches gilt, wenn eine Variable in einem Listen- oder Dictionary-Kontext verwendet wird. Ein ständiges Hin- und Her-Um-Interpretieren – das sogenannte *shimmering* – muss aus Performancegründen unbedingt vermieden werden.

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 für Skalare 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 Arrays, die anders sind als die skalaren Variablen – am Namen erkennt man das jedoch nicht, sondern kann bei Bedarf zur Laufzeit abfragen `array exists` *`variablenname`*, was `1` liefert (also `true`), sofern

eine Array-Variable des genannten Namens existiert. Darüber hinaus gibt es noch Listen und Dictionaries, die aber – im Gegensatz zu Arrays – auch als Zeichenketten interpretiert werden können.

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. In den Handbuchseiten ist immer angegeben, ob ein Wert (dann steht dort z. B. „`string1`“) oder der Name einer Variablen (dann steht dort `varname`) verlangt wird. Eine kurze Überlegung, welchen Zweck das Kommando hat, d. h. ob es den Inhalt der Variablen ggf. verändert oder nicht, bringt ebenfalls Klärung. Notfalls kann man es auch ausprobieren.

3.2 Zeichenkettenoperationen

Die Möglichkeiten von Tcl zur Zeichenkettenverarbeitung sind sehr reichhaltig. Da alle Variablen als Zeichenketten betrachtet werden können und sogar Listen aller Art und Dictionaries in Zeichenketten gespeichert werden, muss dies auch so sein. Die meisten Kommandos verbergen sich als Unterkommandos von `string`, aber beispielsweise ist `append` ein eigenständiges Kommando. Das Anhängen an Zeichenketten mit `append` ist schnell und ressourcenschonend, weil die Zeichenkette meist nicht umkopiert werden muss. Erstes Argument ist der *Name* der zu verändernden Zeichenketten*variablen*, 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. Tatsächlich hält Tcl immer Speicher am Ende von Zeichenketten frei, so dass das Anhängen nicht jedes Mal neuen Speicher allokieren und umkopieren muss. Bei häufigen oder größeren Operationen dagegen schon.

```
set name1 Hans
append name1 ", geb. 1955-12-12"
set name2 Franz
set name2 "$name2, geb. 1999-01-01"
```

Das Kommando `string` hat viele Unterkommandos. Einige davon werden in Tabelle 1 kurz vorgestellt. Über `string` hinaus haben noch folgende Kommandos mit Zeichenketten zu tun: `split` spaltet eine Zeichenkette in eine Liste, `regexp` und `regsub` finden bzw. ersetzen reguläre Ausdrücke[17] in Zeichenketten. In diesen ist ein großer Teil der Leistungsfähigkeit von Tcl zu sehen.

[17] siehe https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck und die `man`-Page zu `re_syntax`.

Kommando	Bedeutung
`string compare`	Vergleichen von Zeichenketten auf größer, kleiner, gleich
`string equal`	Vergleichen von Zeitenketten auf Gleichheit
`string first`	Suchen in einer Zeichenkette
`string index`	einzelnes Zeichen aus einer Zeichenkette ermitteln
`string length`	Länge einer Zeichenkette ermitteln
`string match`	nach einfachen Mustern in Zeichenketten suchen
`string range`	Bereich aus einer Zeichenkette ermitteln
`string replace`	Bereich einer Zeichenkette ersetzen
`string trim`	Zeichen (meist Leerzeichen) links und rechts entfernen

Tabelle 1: Einige Unterkommandos des Kommandos `string`.

3.2.1 Anwendungsbeispiele für `split`

Wenn man eine Textdatei komplett eingelesen hat, steht der gesamte Dateiinhalt in einer einzigen Zeichenkette, die – im Gegensatz zu Zeichenketten in C – auch '\0'-Zeichen enthalten kann, denn Tcl-Zeichenketten sind auch für binäre Strings geeignet. 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]
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]"
```

Wenn das hier zu umständlich oder nicht sprechend erscheint, könnte man statt des wiederholten `lindex` auch `lassign` verwenden:

```
set zeile {34534435:44:Kugelschreiber:250}
lassign [split $zeile :] anr bestand bezeichnung preis
```

```
puts "Artikelnummer ..: $anr"
puts "Lagerbestand ...: $bestand"
puts "Bezeichnung ....: $bezeichnung"
puts "Preis in Cent ..: $preis"
```

3.2.2 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 /etc/passwd]
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]
set inhalt [read $f]
close $f
foreach zeile [split $inhalt \n]  {
  if [regexp {:(M[ae][iy]er); ?([^:]):} $zeile - fname vname] {
    puts "Vorname: $vname, Familienname: $fname"
  } ;# if
} ;# foreach
```

Der gefundene Gesamtausdruck ist hier uninteressant, weshalb das „-“ angegeben wurde. Wichtig sind für uns jetzt Familienname und Vorname, die als geklammerte Unterausdrücke im regulären Ausdruck stehen und den Variablen `famname` und `vorname` zugewiesen werden.

3.2.3 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]
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
```

Allerdings lässt sich diese Änderung auch einfacher herstellen, indem man eine Datei einliest und direkt wieder ausgibt, wobei für die Ausgabedatei eingestellt wird, dass CR + LF als Zeilenendekennung verwendet werden sollen. Wie es sich diesbezüglich bei der Eingabedatei verhält, ist bei dieser Lösung sogar gleichgültig.

```
set fIn [open datei.unix r]
set fOut [open datei.dos w]
chan configure $fOut -translation crlf
puts [read -nonewline $fIn]
close $fOut
close $fIn
```

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

```
# Einlesen
set f [open alt.html]
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.

3.3 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, aber dennoch rechenaufwendig ist. Listen sind durch Blanks getrennte Wörter. Gehören mehrere Wörter zu einem Listenelement, so werden sie mit geschweiften Klammern zusammengehalten.

3.3.1 Listenerzeugung

Diese Zeichenkette – als Liste interpretiert – 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. Auch das geschachtelte Aufrufen des `list`-Kommandos ist möglich und sinnvoll.

In Tcl sind zunächst einmal *alle Kommandos* Listen, aber auch in der Datenverwaltung lassen sich Listen hervorragend einsetzen. Das Kommando für die direkte Erzeugung von Listen ist `list`, für die Verarbeitung gibt es weitere, siehe Tabelle 2 auf der nächsten Seite. `list` gibt alle seine Argumente zurück – zusammengefasst zu einer Liste. 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 *möglichst vermeiden*, diese selbst mit Zeichenkettenoperationen zusammenzubauen.

```
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.

3.3.2 Bearbeiten von Listen

Die einfachste Bearbeitung einer Liste ist ihre Erweiterung. Hängen wir an die Liste noch einmal die Liste `$l1` an und geben noch einmal aus:

Kommando	Bedeutung
`list`	Erzeugung von Listen aus einzelnen Elementen
`lappend`	Anhängen von neuen Elementen an eine Liste, besonders performant!
`lassign`	Zuweisen von Listenelementen an Variablen
`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
`lset`	Setzen eines Werts in einer Liste bzw. einer Unterliste
`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 3.2.1 auf Seite 10

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

Tabelle 2: Listenkommandos

```
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 Nummerierung mit `0` beginnt. Auf das letzte Element kann man auch mit `end` zugreifen (und mit `end-1` auf das vorletzte), 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.
```

3.3.3 Geschachtelte Listen

Bei geschachtelten Listen kann man durch die Angabe einer Liste aus Indexen direkt tief in die Struktur hineinschauen.

```
set schachtelListe [list 12 [list 4 [list 17 44 66] 55] 99 33 [list 3 4 6]]
puts $schachtelListe
12 {4 {17 44 66} 55} 99 33 {3 4 6}
puts [lindex $schachtelListe 1 1 2]
66
```

Das `lindex`-Kommando geht in das Element der äußersten Liste mit dem Index 1, also `4 {17 44 66} 55`, dann wiederum in das Element mit dem Index 1, also `17 44 66`, und holt davon das Element mit dem Index 2, also `66`.

Mittels `lset` kann man auch Elemente tief in der geschachtelten Struktur nach demselben Strickmuster ändern.

```
lset schachtelListe 1 1 2 333
puts $schachtelListe
12 {4 {17 44 333} 55} 99 33 {3 4 6}
puts [lindex $schachtelListe 1 1 2]
333
```

3.3.4 Zugriff auf Listenteile

Möchte man nicht nur einzelne Elemente einer Liste haben, sondern Bereiche 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.

3.3.5 Einfügen in eine Liste

Mit `linsert` kann man Elemente in eine bestehende Liste einfügen, wobei eine neue Liste generiert wird, die man wiederum einer Variablen zuweisen kann – ggf. auch derselben. 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}
```

3.3.6 Sortieren von Listen

Üblicherweise sortiert `lsort` nach den Unicode-Codepoints. 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.

3.3.7 Durchsuchen von Listen

Mit `lsearch` kann man feststellen, ob und wo 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`.

Geht es nur darum festzustellen, ob ein Element in einer Liste (exakt) enthalten ist oder nicht, kann man das ab Version 8.5 auch ohne `lsearch` lösen. Für die Prüfung „ist enthalten" wird `in` verwendet, für das Gegenteil gibt es `ni` (für „not in").

```
if {"Meier" in $namen} {
  puts "Meier kommt vor"
}
```

3.3.8 Löschen und Ersetzen von Listenelementen

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 Nummerierung der Elemente mit `0` beginnt, d. h. „`Anna`“ hat nach wie vor den Index `0` (siehe `for`-Schleifenbeispiel auf Seite 15).

Oft liest man Zeilen aus Textdateien, die mehrere Felder enthalten – beispielsweise bei den Unix-Konfigurationsdateien wie `/etc/group`.

```
set f [open /etc/group]
foreach zeile [split [read -nonewline $f] \n] {
  lassign [split $zeile :] gname - gid secmembers
  puts "Die Gruppe $gname hat die Id $gid und\
        [llength [split $secmembers ,]] sekundäre Mitglieder."
}
close $f
```

Zunächst wird die Datei geöffnet, dann unter Entfernung des letzten „Newline“ gelesen und in Zeilen gesplittet. Diese Liste wird vom `foreach` abgearbeitet. Jede Zeile wird anhand der Doppelpunkte gesplittet und die Felder den Variablen `gname`, `gid` und `secmembers` zugewiesen. Das zweite Feld ist nicht von Interesse und wird daher der Dummyvariablen `-` zugewiesen. Im Ausgabekommando `puts` wird die Variable `secmembers` an den Kommas gesplittet und die Länge der sich ergebenden Liste mittels `llength` ermittelt und neben den Variablen `gname` und `gid` in die auszugebende Zeichenkette eingefügt.

Da auch das Kommando `lsearch` eine neue Liste zurückgibt, kann man durch einen *negativen Suchausdruck* bestimmte Elemente aus einer Liste „entfernen“ – eigentlich alle nicht passenden suchen und die Liste damit wieder überschreiben. Bei `lsearch` mann man per Option angeben, wie gesucht werden soll: `-exact`, `-glob` (wie bei `string match`) oder `-regexp` (reguläre Ausdrücke). Aus einer Liste kann man nun alle entfernen, die ein „v“ enthalten:

```
set l3 [lsearch -all -inline -not -glob $l3 {*v*}]
puts $l3
Anna Marc Bill
```

3.4 Arrays

Arrays stellen im Gegensatz zu Listen und Dictionaries schon einen völlig anderen Typ dar, in den eine existierende skalare Variable nicht umgewandelt werden kann, 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)
```

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 so ähnlich auch in JavaScript (hier sind es Objekte) verwendet wird. Da der Index eine Zeichenkette ist, kann natürlich auch jede Liste und jedes Dictionary verwendet werden. In Java würde man eine `HashMap` verwenden, in C++ eine `unordered_map`, wobei man bei diesen Sprachen die Datentypen für Index und Wert angeben muss, was bei Tcl weder nötig noch möglich ist.

Für die Bearbeitung von Arrays gibt es das Kommando `array`, welches in Tabelle 3 auf der nächsten Seite vorgestellt wird.
Sehr häufig arbeitet man alle Elemente eines Arrays ab, wozu man sich die Indexe (Schlüssel) als Liste geben lässt und diese dann durchgeht:

```
set auto(Franz) Mercedes
set auto(Ulla) Maserati Porsche
set auto(Anna) Volkswagen
set auto(Heinrich) Peugeot

foreach x [array names auto] {
  puts "$x fährt $auto($x)"
}

Franz fährt Mercedes
Heinrich fährt Peugeot
Anna fährt Volkswagen
```

Kommando	Bedeutung
`parray`	Anzeigen der Inhalte eines Arrays auf der Standardausgabe zwecks Fehlersuche
`array`	Ein komplexes Kommando ähnlich `string` (siehe Tabelle 1 auf Seite 10) und `dict` (siehe Tabelle 5 auf Seite 23 und `file` (siehe Tabelle 7.2.4 auf Seite 50), d. h. man muss immer ein Unterkommando auswählen. Hier eine Liste der am häufigsten verwendeten Unterkommandos:
	`set` Setzen der Array-Elemente entsprechend der übergebenen Liste aus Indexen und Werten
	`size` Ermitteln der Anzahl Elemente in einem Array
	`exists` prüft, ob die übergebene Zeichenkette der Name eines Arrays ist
	`get` liefert die Array-Elemente als eine Liste aus Index-Wert-Paaren, die Umkehrung von `set`
	`names` liefert eine Liste mit Namen aller Array-Indexe, sinnvoll mit einer `foreach`-Schleife zu verwenden
	`unset` entfernt Array-Elemente, die auf ein Muster passen

Tabelle 3: Einige Array-Kommandos

Die Zuweisungen am Anfang können auch durch ein einziges Kommando ersetzt werden: `array set auto [list Franz Mercedes Heinrich Peugeot Anna Volkswagen]`.

Das Kommando `array names` gibt die Indexeinträge nicht in einer bestimmten Reihenfolge wieder, ggf. ist die gelieferte Liste zu sortieren.

3.5 Dictionaries

3.5.1 Grundidee von Dictionaries

Dictionaries und das zugehörige Kommando `dict` wurden im September 2007 mit Tcl 8.5 eingeführt. Sie haben dieselbe Grundidee wie Listen, nämlich dass speziell formatierte Zeichenketten als Datenstruktur interpretiert werden und einen schnellen Zugriff auf die Elemente erlauben. Es wird derselbe Tokenizer verwendet, der auch Quellcode zerlegt. Die Listenelemente bilden in Dictionaries allerdings Paare aus Name und Wert (key-value pairs), so dass ihre Anzahl zwingend eine gerade Zahl sein muss. Auf diese Weise werden die positiven Eigenschaften von Zeichenketten und Arrays vereint.

Eine Ähnlichkeit besteht mit JSON (JavaScript Object Notation[18]), allerdings ohne die strenge Unterscheidung in Objekte (in JavaScript in geschweiften Klammern gefasst, in Tcl dagegen Dictionaries) und Arrays (in JavaScript in eckigen Klammern gefasst, in Tcl dagegen Listen). Auch die zusätzlichen Anführungsstriche und Doppelpunkte fehlen bei Tcl-Dictionaries, siehe Tabelle 4. Die Information, was Listen mit einer geraden Anzahl

[18]JSON ist in konkurrierenden Standards „normiert“, in RFC 7159 und ECMA-404

JSON

```
 {
  "Herausgeber": "Xema",
  "Nummer": "1234-5678-9012-3456",
  "Deckung": 2e+6,
  "Waehrung": "EURO",
  "Inhaber":   {
    "Name": "Mustermann",
    "Vorname": "Max Theodor",
    "maennlich": true,
    "Hobbys": [ "Reiten", "Golfen" ],
    "Alter": 42,
    "Kinder": []
  }
}
```

Tcl Dictionary

```
 {
  Herausgeber Xema
  Nummer 1234-5678-9012-3456
  Deckung 2e+6
  Waehrung EURO
  Inhaber {
    Name Mustermann
    Vorname {Max Theodor}
    maennlich true
    Hobbys {Reiten Golfen}
    Alter 42
    Kinder {}
  }
}
```

Tabelle 4: Vergleich von JSON mit Tcl-Dictionaries

von Elementen sind (beispielsweise 2 Kindernamen) und was Dictionaries mit einem Key-Value-Paar sind, ist im Tcl-Dictionary nicht hinterlegt.

Ein ganz einfaches Dictionary besteht aus einer Liste mit einer geraden Anzahl von Elementen. Als Liste interpretiert gibt es 2N Werte, als Dictionary interpretiert Name-Wert-Paare. Aber natürlich kann das Dictionary auch als Zeichenkette angesehen und damit als Parameter übergeben oder in eine Datei geschrieben werden.

```
set chemelem {Au Gold Fe Eisen Na Natrium Ag Silber O Sauerstoff}
puts [dict get $chemelem Fe]
puts [lindex $chemelem 3]
puts [string range $chemelem 11 15]
```

Das gibt dreimal die Zeichenkette „Eisen“ aus, denn der Wert, der zum Namen „Fe“ gehört, lautet so. Ebenso ist das Element mit dem Index 3 die Zeichenkette „Eisen“ (zur Erinnerung: Indexe werden ab 0 gezählt). Zum dritten stellen die Zeichen 11 bis 15 (ebenfalls ab 0 gezählt) die Zeichenkette „Eisen“ dar.

Aber Dictionaries können mehr. Man kann sich alle Schlüssel anzeigen lassen.

```
puts [dict keys $chemelem]
```

Das zeigt „`Au Fe Na Ag O`“ an. Um jetzt alle Name-Wert-Paare zu erhalten, schreibt man wie folgt. Die Ausgabe steht hier direkt unter dem Code.

```
foreach el [dict keys $chemelem] {
```

```
    set wert [dict get $chemelem $el]
    puts "Das Zeichen $el steht für $wert."
}
Das Zeichen Au steht für Gold.
Das Zeichen Fe steht für Eisen.
Das Zeichen Na steht für Natrium.
Das Zeichen Ag steht für Silber.
Das Zeichen O steht für Sauerstoff.
```

3.5.2 Bearbeiten von Dictionaries

Um Dictionaries kontrolliert zu erzeugen und zu bearbeiten sowie Daten zu extrahieren, gibt es das Kommando `dict`, welches – ganz ähnlich wie `string` und `file` – eine ganze Reihe von Unterkommandos (subcommands) hat. Einen Auszug davon zeigt die Tabelle 5 auf der nächsten Seite.

Für den Zugriff auf geschachtelte Dictionaries muss lediglich der Pfad zum gewünschten Element angegeben werden. Ein Dictionary kann also auch ähnlich wie ein XML-Dokument betrachtet werden, bei dem der Zugriff mittels XPath stattfindet.
Beispiel:

```
dict set chem Alkalimetalle Li Name Lithium
dict set chem Alkalimetalle Li Zustand fest
dict set chem Alkalimetalle Li Ordnungszahl 3
dict set chem Alkalimetalle Li Gewicht 6.941

dict set chem Erdalkalimetalle Be Name Beryllium
dict set chem Erdalkalimetalle Be Zustand fest
dict set chem Erdalkalimetalle Be Ordnungszahl 4
dict set chem Erdalkalimetalle Be Gewicht 9.0122

dict set chem Alkalimetalle Na Name Natrium
dict set chem Alkalimetalle Na Zustand fest
dict set chem Alkalimetalle Na Ordnungszahl 11
dict set chem Alkalimetalle Na Gewicht 22.99

dict set chem Erdalkalimetalle Mg Name Magnesium
dict set chem Erdalkalimetalle Mg Zustand fest
dict set chem Erdalkalimetalle Mg Ordnungszahl 12
dict set chem Erdalkalimetalle Mg Gewicht 24.305

dict set chem Alkalimetalle K Name Kalium
dict set chem Alkalimetalle K Zustand fest
dict set chem Alkalimetalle K Ordnungszahl 19
```

Kommando	Bedeutung
`dict create`	neues Dictionary erzeugen
`dict append`	Zeichenkette an einen Wert im Dictionary anhängen
`dict lappend`	ein neues Name-Wert-Paar ans Dictionary anhängen
`dict merge`	zwei Dictionaries zusammenfügen, wobei doppelte Schlüssel entfernt werden
`dict replace`	Name-Wert-Paare basierend auf Schlüssel ersetzen
`dict filter`	Dictionary filtern
`dict size`	Anzahl Name-Wert-Paare ermitteln
`dict exists`	Namen auf Existenz prüfen
`dict for`	über Dictionary iterieren
`dict get`	Wert in Dictionary finden und liefern
`dict set`	Wert in Dictionary setzen
`dict values`	auf ein Muster passende Werte im Dictionary liefern

Tabelle 5: Einige Unterkommandos des Kommandos `dict`.

```
dict set chem Alkalimetalle K Gewicht 39.098

dict set chem Erdalkalimetalle Ca Name Calcium
dict set chem Erdalkalimetalle Ca Zustand fest
dict set chem Erdalkalimetalle Ca Ordnungszahl 20
dict set chem Erdalkalimetalle Ca Gewicht 40.078
```

Diese vielen Zuweisungen sehen etwas unpraktisch aus. Hier kann das Kommando `dict with` helfen, das einen verkürzten Zugriff auf Teile eines Dictionarys erlaubt, indem es die Schlüssel an der angegebenen Stelle als gewöhnliche Variablen in den aktuellen Scope einblendet. Aber Vorsicht: Erstens kann es Kollisionen mit den Namen anderer Variablen geben (evtl. sogar mit dem Namen der Dictionary-Variablen selbst), und zweitens werden nur bereits bestehene Schlüssel eingeblendet, so dass das Erzeugen neuer Schlüssel nicht möglich ist. Ein Abschnitt aus obigen Zuweisungen könnte so gefasst werden, wobei die erste Zuweisung auf `Mg` dazu dient, diesen Schlüssel anzulegen, so dass er eingeblendet wird. Die Variable `Mg` wird angelegt und als Dictionary verwendet. Am Ende des `with`-Blocks werden die Werte ins Dictionary zurück transferiert:

```
dict set chem Erdalkalimetalle Mg {}
dict with chem Erdalkalimetalle {
  dict set Mg Name Magnesium
  dict set Mg Zustand fest
  dict set Mg Ordnungszahl 12
  dict set Mg Gewicht 24.305
}
```

Die vielen Zuweisungen – ob verkürzt oder nicht – werden zu folgendem Dictionary kombiniert:
`Alkalimetalle {Li {Name Lithium Zustand fest Ordnungszahl 3 Gewicht 6.941} Na {Name Natrium Zustand fest Ordnungszahl 11 Gewicht 22.99} K {Name Kalium Zustand fest Ordnungszahl 19 Gewicht 39.098}} Erdalkalimetalle {Be {Name Beryllium Zustand fest Ordnungszahl 4 Gewicht 9.0122} Mg {Name Magnesium Zustand fest Ordnungszahl 12 Gewicht 24.305} Ca {Name Calcium Zustand fest Ordnungszahl 20 Gewicht 40.078}}`

An die einzelnen Angaben kann man mittel `dict get` leicht herankommen:

```
puts [dict get $chem Erdalkalimetalle Ca Gewicht]
```

erzeugt als Ausgabe das Atomgewicht von Calcium, also `40.078`. Die Schachtelung der Strukturen kann beliebig tief gehen, auch können in jedem Element beliebige Namen und beliebige Werte verwendet werden.

Möchte man dieses Dictionary nun anders darstellen, so dass nicht mehr nur zwei Namen (die Gruppen) auf der obersten Ebene stehen, sondern jedes Element einzeln, wobei die Gruppenzugehörigkeit als weitere Eigenschaft beim einzelnen Element steht, so kann man es wie folgt transformieren:

```
set chem2 ""
foreach gruppe [dict keys $chem] {
  set elemente [dict get $chem $gruppe]
  foreach el [dict keys $elemente] {
    dict set elemente $el Gruppe $gruppe
  }
  set chem2 [dict merge $chem2 $elemente]
}
```

Und schon steht in der Variablen `chem2` folgendes: `Li {Name Lithium Zustand fest Ordnungszahl 3 Gewicht 6.941 Gruppe Alkalimetalle} Na {Name Natrium Zustand fest Ordnungszahl 11 Gewicht 22.99 Gruppe Alkalimetalle} K {Name Kalium Zustand fest Ordnungszahl 19 Gewicht 39.098 Gruppe Alkalimetalle} Be {Name Beryllium Zustand fest Ordnungszahl 4 Gewicht 9.0122 Gruppe Erdalkalimetalle} Mg {Name Magnesium Zustand fest Ordnungszahl 12 Gewicht 24.305 Gruppe Erdalkalimetalle} Ca {Name Calcium Zustand fest Ordnungszahl 20 Gewicht 40.078 Gruppe Erdalkalimetalle}`

Zur Ausgabe des Atomgewichts von Calcium genügt nun

```
puts [dict get $chem2 Ca Gewicht]
```

3.6 Zeit und Datum

Zeit- und Datumsarithmetik sind nicht einfach, aber Tcl gibt hier gute Hilfestellung. Statt einen neuen Datentyp für Zeit und Datum einzuführen, wird schlicht eine ganze Zahl mit der Anzahl Sekunden seit 1970-01-01T00:00 UTC verwendet. Das zeigt schon, dass Zeitstempel zeitzonenunabhängig gespeichert werden. Durch die Verwendung der Sekundenanzahl ist eine einfache Addition und Subtraktion jederzeit möglich und korrekt.

Nach außen jedoch möchte man natürlich die üblichen Darstellungen zeigen und auch einlesen können. Das ist möglich mit dem Kommando `clock`, siehe Tabelle 6.

Kommando	**Bedeutung**
`clock add`	Addieren von Stunden, Tagen, Wochen zu einem Zeitpunkt
`clock format`	Formatierung eines Zeitstempelwerts mithilfe der C-Funktion `strftime()`
`clock seconds`	liefert Zeit in Sekunden (Standardwert)
`clock scan`	versucht, Zeichenkette als Datum-/Zeit-Wert zu interpretieren
`clock add`	Addition von Zeiteinheiten zu einem Sekundenwert unter Berücksichtigung von Sommer-/Winterzeit und Schaltsekunden

Tabelle 6: Zeit- und Datum-Kommandos

Beispielanwendung für `clock`:

```
set jetzt [clock seconds]
puts "Jetzt ist es hier: [clock format $jetzt]"
puts "Jetzt ist es hier: [clock format $jetzt -format "%Y-%m-%d %T"]"
puts "Jetzt ist es in New York: [clock format $jetzt \
         -format "%Y-%m-%d %T"  -timezone :America/New_York]"
```

Die erste Ausgabezeile enthält das Datum in traditioneller US-amerikanischer Schreibweise, aber mit lokaler Zeitzone, die zweite Zeile dasselbe in ISO8601-, also DIN-Schreibweise, während die dritte Zeile zwar das ISO-Format verwendet, sich aber auf die Zeitzone von New York bezieht.

Auch beim `clock scan`, also beim Einlesen, kann man ein Format angeben. Ohne dieses werden nur traditionelle amerikanische Schreibweisen und ISO8601/DIN richtig interpretiert.

Für den Fall, dass höhere Auflösungen als Sekunden benötigt werden, gibt es auch `clock milliseconds` und `clock microseconds`. Die Variante `clock clicks` hat die höchstmögliche Auflösung, die allerdings systemabhängig ist.

3.7 Variablenname in eine Variablen

Gelegentlich kommt man in die Situation, dass man den Namen der zu verwendenden Variablen wiederum in einer Variablen hat. Das bedeutet, dass zunächst der Inhalt der Variablen ausgewertet werden muss, um an den Namen der eigentlichen Variablen zu kommen. In Tcl gibt es für diesen indirekten Zugriff (der in dynamischen Sprachen gängig, in voll compilierten Sprachen unmöglich ist) zwei Möglichkeiten.

```
set m name
set $m "Franz Meier"
puts $m           => zeigt 'name' an
puts [subst $$m]  => zeigt 'Franz Meier' an
puts [set $m]     => zeigt 'Franz Meier' an
```

Wenn möglich, sollte man aber eher auf Arrays oder Dictionarys zurückgreifen, wenn man auf Werte anhand einer Zeichenkette zugreifen möchte.

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 mehrere 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 ohne jeden Einfluss auf das Kommando 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. Diese dürfen lediglich dann entfallen, wenn ein Zweig aus nur einem einzigen Wort besteht – beispielsweise `return` oder `break`. Der Normallfall sieht beispielsweise so aus:

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

Das kann man auch noch ausführlicher und strukturierter schreiben, wobei in der Praxis das Wort `then` sehr selten, `else` dagegen häufig angegeben wird:

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

Wichtiger Hinweis: Wer es sich nicht schon bei anderen Sprachen angewöhnt hat, die öffnende geschweifte Klammer hinter das `if`-Statement zu schreiben (sogenannte „cuddled“ Schreibweise), wird sich für Tcl umgewöhnen müssen! Sonst funktioniert das Kommando nämlich nicht, sondern es fehlen die Listenelemente hinter der Bedingung, weil das Zeilenende die Liste und damit das gesamte Kommando beendet.

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. Diese Übergabe kann man „call by name“ nennen.

4.2 Mehrfach-Verzweigung

Die aus C, Java und anderen Sprachen bekannte Mehrfach-Verzweigung mittels `switch` gibt es in Tcl auch. Hierbei kann ein Wert (im Gegensatz zu C und zu Java) nicht nur auf exakte Übereinstimmung von abzählbaren Werten (oder Strings bei Java seit Version 1.5) geprüft werden, sondern auch auf Ähnlichkeit von Zeichenketten 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 bei `string match` 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. Das häufig vergessene `break` wie bei C oder Java gibt es beim `switch`-Statement von Tcl 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.

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
}
```

Die for-Schleife ist ähnlich wie in C und Java 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. Natürlich kann man statt eines einzelnen Kommandos beim initialen und beim finalen Kommando auch beliebig lange Kommandofolgen eintragen. Indem diese Teile alle einzelne Argumente sind, wird vermieden, hierfür eine Erweiterung der Syntax vornehmen zu müssen.

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

Auch dies kann man über mehrere Zeilen verteilen, was man auf jeden Fall tun sollte, wenn der Schleifenrumpf mehrere Kommandos enthält.

```
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 (oder tripelweise) Werte (Name und Gehalt bzw. Name, Grundgehalt und Kinderanzahl), so kann man zwei bzw. drei 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 38.

4.4.1 Fehler abfangen mit `catch`

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 8 auf Seite 41) kann statt `exit` zum Programmabbruch ggf. auch `return` zur Rückkehr in die rufende Prozedur verwendet werden.

4.4.2 Fehler abfangen mit `try`

Neben dem oben beschriebenen `catch`, welches einen Block ausführt und ggf. auftretende Fehler abfängt, gibt es auch das `try`-Kommando, welches eine feinere Abstufung ermöglicht und auch die Möglichkeit gibt, eine `finally`-Block in jedem Fall auszuführen – auch wenn eine Schleife oder Prozedur vorzeitig verlassen wird.

Eine Syntaxvariante ist wie folgt:

```
try {
  ...
} on code1 variableList1 {
  ...
} on code2 variableList2 {
  ...
} finally {
  ...
}
```

Als `code`s wird einer der Werte `ok`, `error`, `return`, `break` oder `continue` angegeben. `variableList` ist eine Liste von Variablennamen, wobei es meistens nur eine einzige ist, die mit einer lesbaren Meldung versehen wird.

Es werden hier also schon mehrere verschiedene Arten des Ergebnisses unterschieden und entsprechend darauf reagiert. Noch feiner kann die Einteilung geschehen, indem man die zweite Syntaxvariante verwendet:

```
try {
  ...
} trap muster1 variableList1 {
  ...
} trap muster2 variableList2 {
  ...
} finally {
  ...
}
```

Wenn hierbei ein Fehler auftritt und das Präfix des `errorcode` im Status-Dictionary auf `muster` passt, wird der entsprechende Zweig betreten. Hierbei wird oft auf die Fehlercodes des Systems zurückgegriffen, die sehr von Unix inspiriert und teilweise im POSIX-Standard[19] festgelegt sind. Diese Fehlercodes sind in der Datei `errno.h` abgelegt[20].

Beispielsweise kann man jetzt fein differenzieren, welche Art von Fehler aufgetreten ist, d. h. ob es eine Datei nicht zum Schreiben geöffnet werden kann, weil das Elternverzeichnis nicht existiert, bereits ein Verzeichnis gleichen Namens vorhanden ist oder die Rechte für die Erzeugung (fehlende Schreibrechte am Elternverzeichnis) oder für das Öffnen (fehlende Schreibrechte an der bereits bestehenden Datei) nicht vorhanden sind.

```
try {
  set f [open /home/meier/2016/briefe/anfrage.txt w]
  puts $f "neuer Text in der Datei"
} trap {POSIX EISDIR} err  {
  puts stderr "gleichnamiges Verzeichnis vorhanden:\n\t$err"
} trap {POSIX ENOENT} err {
  puts stderr "Elternverzeichnis fehlt:\n\t$err"
} trap {POSIX EACCES} err {
  puts stderr "Zugriff verweigert:\n\t$err"
} finally {
  catch {close $f}
}
```

Je nach Situation bekommt man eine dieser Meldungen (die im Detail je nach Betriebssystem abweichen können):

```
Elternverzeichnis fehlt:
  couldn't open "/tmp/briefe/anfrage.txt": no such file or directory
```

[19] https://de.wikipedia.org/wiki/Portable_Operating_System_Interface
[20] auch online abrufbar: http://man7.org/linux/man-pages/man3/errno.3.html

```
gleichnamiges Verzeichnis vorhanden:
  couldn't open "/tmp/briefe/anfrage.txt": illegal operation on a directory
Zugriff verweigert:
  couldn't open "/tmp/briefe/anfrage.txt": permission denied
```

Andere Fehlerklassen (neben `POSIX`) sind folgende: `ARITH` (z. B. beim Versuch, durch 0 zu dividieren), `CHILDKILLED` (wenn ein Kindprozess durch ein Signal beendet wurde), `CHILDSTATUS` (wenn ein Kindprozess einen von 0 verschiedenen Exit-Status geliefert hat), `CHILDSUSP` (wenn ein Kindprozess suspendiert wurde), `NONE` (wenn es keine weitere Information zum Fehler gibt), `POSIX` (wenn der Fehler bei einem Aufruf einer Betriebssystemfunktion aufgetreten ist), or `TCL` (wenn es ein Tcl-internes Problem gibt).

Weil bei `trap` immer nur Fehler bestimmter Fehlerklassen abgefangen werden, ist es durchaus sinnvoll, nach einem oder mehreren `trap`-Zweigen noch einen `on error`-Zweig dahinter zu schreiben, um alle weiteren Fehler mit diesem zu fangen. Dies ist vergleichbar mit einem `catch (Exception e)` in Java, welches auch unspezifisch alle Ausnahmen fängt.

4.4.3 Ausnahmen melden mit `return`

Auch in Tcl-Code kann man Fehler erzeugen, beispielsweise wenn eine Prozedur mit falschen Parameter aufgerufen wurde. Natürlich kann man sich Mechanismen ausdenken, die kunstvoll in Rückgabewerten Fehlersituationen beschreiben. Es ist auch völlig in Ordnung, wenn bei einer Suche ohne Ergebnis ein Wert -1 zurückgegeben wird, sofern die Suche an sich durchgeführt werden konnte. Geht die Durchführung der Suche an sich jedoch schief, dann sollte das anders gemeldet werden. Zu diesem Zweck kann man beim Verlassen einer Prozedur nicht nur einen Wert, sondern auch einen Code zurückgeben: `return -code error`.

Wurde die Prozedur in einem `try`- (oder auch `catch`-)Block aufgerufen, so wird der Fehler dort abgefangen und bearbeitet.

Gibt man `return -code return` zurück, wirkt der Prozeduraufruf in der rufenden Prozedur wie ein `return`-Statement, so dass auch diese Prozedur verlassen wird.

Durch `return -code break` und `return -code continue` wird die Schleife in der rufenden Prozedur beeinflusst wie durch ein `break` bzw. `continue`. Die Prozedur darf hierfür nur in einer Schleife aufgerufen worden sein, sonst gibt es einen Fehler `invoked "break" outside of a loop`.

4.4.4 Ausnahmen erzeugen mit `throw`

Alternativ zur Zusatzangabe von `-code` bei `return` kann man auch das Kommando `throw` verwenden, welches zwei Parameter bekommt: Der erste Parameter enthält eine maschinenlesbare Information zur Ausnahme, die zum ersten Parameter hinter `trap` passen muss. Der zweite Parameter enthält eine menschenlesbare textuelle Beschreibung der Ausnahme.

Einen Versuch, den Durchschnittwert einer leeren Liste von Zahlen zu berechnen, führt zu einer Division durch 0, die man so mitteilen kann:

```
throw {ARITH DIVZERO} {attempt to divide by zero}
```

5 Ausgabeformatierung

Die Ausgabeformatierung wurde zum größten Teil von C übernommen, vom berühmten `sprintf`-statement – ähnlich wie Java das mit `String.format` ebenfalls getan hat. Hier heißt es schlicht `format` und formatiert den String, ohne ihn auszugeben. Daher wird für eine Ausgabe zusätzlich 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.

Auch Datum-/Zeit-Werte kann man gut formatieren, und zwar mit dem Unterkommando `format` des `clock`-Kommandos.

Folgende Platzhalter sind gängig: `%i` oder `%d` für vorzeichenbehaftete Dezimalausgabe von Ganzzahlen, `%u` für vorzeichenlose Ausgabe von Ganzzahlen, `%o` für Oktalausgabe, `%x` für Hexadezimalausgabe, `b` für Binärausgabe, `%c` für die Ausgabe des Unicodezeichens mit dem Codepoint der Zahl im Wert, `%f` für Fixpunktdarstellung, `%e` für Exponentialdarstellung, `%g` für Fixpunkt- oder Exponentialdarstellung (in Abhängigkeit vom Wert) und `%s` für Zeichenkettenausgabe. Die Breite und ggf. die Nachkommastellenanzahl kann angegeben werden. Ist die Breite negativ, geschieht eine linkbündige, sondst eine rechtsbündige Ausgabe. Bei Zeichenketten wird also vorwiegend eine „negative“ Breite angegeben. Um ein Prozentzeichen ausgzugeben, schreibt man zwei Prozentzeichen hintereinander in den Formatstring.

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
```

Es werden gelegentlich Ersatzschreibweisen für mehrzeilige Kommentare diskutiert, aber wirkliche Kommentare sind es nicht, wenn man wie folgt schreibt:

```
if {0} {
   Dies soll ein Kommentartext sein
   { geschweifte Klammern machen Probleme
}
```

Tatsächlich ist der „Kommentar“ bei der schließenden Klammer nicht zu Ende, denn die zusätzlich geöffnete Klammer vor dem Wort „geschweifte“ hat eine Wirkung. Der Interpreter parst nämlich den Inhalt des Anweisungsblocks, auch wenn die Bedingung niemals erfüllt werden kann, so dass eine Übersetzung in Bytecode nicht erfolgt. Ein Kommentar, dessen Inhalt völlig egal ist, ist es also nicht.

Tatsächlich ist es beim „#“-Kommando ähnlich, denn auch hier führen unpaarige geschweifte Klammern zu Schwierigkeiten.

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, bei denen es übrigens gleichgültig ist, ob der Dateiname in Anführungsstrichen steht oder nicht, da der Name keine Leerzeichen enthält:

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. Da einfache Dateioperationen (zumindest bei unixoiden Systemen) atomar sind, werden auf diese Weise erzeugte Dateien als „Lock"-Dateien verwendet, um gegenseitigen Ausschluss zu realisieren.
`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 Systems 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 kann man das `catch`- oder das `try`-Kommando verwenden, siehe auch Abschnitte 4.4.1 auf Seite 30 und 4.4.2 auf Seite 31.

```
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.

```
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.

Alternativ und mittlerweile beliebter ist die Variante mit `try`, vor allem weil es die Möglichkeit des `finally` gibt, so dass man eine Prozedur nach dem Öffnen an beliebiger Stelle verlassen und dennoch sicher sein kann, dass die Datei wieder geschlossen wird.

```
try {
  set f [open beispiel.txt RDONLY]
  gets $f zeile
  puts $zeile
  if {$zeile == "ende"} return
  gets $f zeile
  puts $zeile
} on error msg {
  puts stderr $msg
} finally {
  catch {close $f}
}
```

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.

```
try {
  set f [open beispiel.txt RDONLY]
  while {1} {
    gets $f zeile
    if [eof $f] break
    puts $zeile
  }
} on error err {
  puts stderr $err
} finally {
  catch {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, gemäß dem DRY-Prinzip[21].

[21] `http://en.wikipedia.org/wiki/Don%27t_repeat_yourself`

```
try {
  set f [open beispiel.txt RDONLY]
  set alles [read -nonewline $f]
  puts $alles
} on error err {
  puts stderr $err
} finally {
  catch {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 – im `finally`-Block ist das todsicher nicht zu vergessen, egal was passiert. Andernfalls wird sie erst am Ende des Programms automatisch geschlossen. Läuft der Block mit dem Öffnen (und vergessenen Schließen) in einer Schleife, ist nach einer bestimmten Anzahlen Durchläufen Schluss[22].

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 das 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
```

[22] Diese Anzahl wird oft in Tests nicht erreicht, so dass der Fehler unglücklicherweise erst später im Feld beim Kunden zutage tritt.

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 Datei- und Directory-Operationen

Es gibt weitere Datei- und Directory-Operationen, die gelegentlich sehr hilfreich sind. Tabelle 7.2.4 auf Seite 50 zeigt eine Übersicht.

Kommando	Bedeutung
`glob`	Erzeugen einer Dateinamenliste aus einem Verzeichnis, ähnlich wie die Bourne-Shell das tut.
`file`	Ein komplexes Kommando ähnlich `string` (siehe Tabelle 1 auf Seite 10), `array` (siehe Tabelle 3 auf Seite 20 und `dict` (siehe Tabelle 5 auf Seite 23), d. h. man muss immer ein Unterkommando auswählen. Hier eine Liste der am häufigsten verwendeten Unterkommandos:
	`atime` Ermittlung der letzten Zugriffszeit (access time)
	`attributes` Ermittlung der Datei-Attribute (plattformabhängig)
	`copy` Kopieren von Dateien.
	`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`)
	`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 Dateiänderungszeit (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 7: Einige Datei- und Directory-Kommandos

8 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 können wie gewöhnlich erzeugt werden und bleiben nach Prozedurende bestehen.

Statt Variablen mittels `global` als global zu deklarieren, kann man auch die Namensraum-Schreibweise verwenden. Der globale Namensraum hat keinen Namen, so dass man vor die Variable lediglich einen doppelten Doppelpunkt schreibt.

```
proc verdoppleX {} {
  global x
  set x [expr {$x*2}]
}
proc verdoppleY {} {
  set ::y [expr {$::y*2}]
}
```

Auch das Kommando zum 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 und nicht wertliefernden Prozeduren. Tatsächlich liefern alle Prozeduren einen Wert zurück, ggf. ist es der Wert, den das letzte in der Prozedur ausgeführte Kommando geliefert hat. Bei einer Zuweisung mittels `set` ist das der zugewiesene Wert. Gängig ist auch ein `set`, ohne dabei einen neuen Wert für die Variable anzugeben. In obigem Beispiel könnte man genauso gut das Wort `return` und die eckigen Klammern weglassen, was aber für nicht so erfahrende Tcl-Programmierer undurchsichtig erscheint, weil die Rückgabe nicht ausdrücklich erfolgt.

```
proc doppel {x} {expr {2*$x}}
```

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.

```
proc doppel {x} {
  upvar $x xx
  if [info exists xx]  {
    set xx [expr {2*$xx}]
  } else {
   return -code error "Die übergebene Variable '$x' existiert nicht."
  } ;# if
} ;# proc doppel
```

Ein Aufruf bringt es an den Tag:

```
doppel gibtesnicht
Die übergebene Variable 'gibtesnicht' existiert nicht.
    while executing
"doppel gibtesnicht"
```

9 Namensräume

Ein hervorragendes Mittel zur Strukturierung von Programmen sind Namensräume (name spaces). Sie dienen neben der Strukturierung auch dazu, Namenskollisionen zu vermeiden. Die meisten Erweiterungen von Tcl und Tk verwenden einen eigenen Namensraum.

Der „globale“ Namensraum, welches der bislang verwendete ist, trägt keinen Namen. Um von Prozeduren oder auch aus benannten Namensräumen heraus auf Variablen des globalen Namensraums zuzugreifen, stellt man dem Variablennamen `::` vor. Dies ist vergleichbar mit dem `/` bei einem absoluten Pfad im Dateisystem. Tatsächlich können Namensräume genauso geschachtelt werden wie Verzeichnisse im Dateisystem, und es gibt auch hier absolute und relative Pfade.

Ohne ausdrückliche Namensräume gibt es nur den „globalen“ Namensraum und lokale Variablen innerhalb von Prozeduren, die aber mit Verlassen der jeweiligen Prozedur „out of scope“ gehen und damit aufhören zu existieren.

Ein Namensraum ist eine Sammlung von Kommandos (Prozeduren) und Variablen, die darin eingekapselt werden. Eine Variable kann in einem Namensraum länger existieren als eine Prozedur läuft. Eine Prozedur in einem Namensraum kann über eine Namensraumvariable mitzählen, wie oft sie bereits aufgerufen wurde. Ohne Namensräume ist das nur durch „Verschmutzen“ des globalen Namensraums möglich, da es kein Pendant zu „statischen“ Variablen wie in C/C++ innerhalb von Prozeduren gibt.

9.1 Erzeugung eines Namensraums

Ein Namensraum wird mittels des Kommandos `namespace` mit dem Unterkommando `eval` erzeugt. Dahinter schreibt man einen Block, der in diesem Namensraum evaluiert werden soll. Existiert der Namensraum bislang nicht, wird er angelegt.

In Namensräumen gibt es die Kommandos `proc` zum Erzeugen von Prozeduren (genau wie im globalen Namensraum) und `variable` zum Erzeugen von Variablen.

Prozeduren werden genaus wie im „globalen“ Namensraum verwendet. Innerhalb desselben Namensraums braucht nichts berücksichtigt zu werden.

Um Prozeduren eines anderen Namensraums aufzurufen, muss man den (absoluten oder relativen) Pfad vom aktuellen Namensraum (abfragbar mit `namespace current`) zum Namensraum der Prozedur angegeben werden, beispielsweise `::namesp1::proc2` für die Angabe des Namensraums `namesp1`, der sich unmittelbar im „globalen“ Namensraum befindet (absoluter Pfad), aus dem die Prozedur `proc2` aufgerufen wird.

9.2 Variablen in Namensräumen

Mit dem Kommando `variable` kann ein Namensraumvariable erzeugt und ggf. mit einem Wert initialisiert werden. Array-Variablen müssen erst mit `variable` ohne Initialisierung angelegt und anschließend mit Werten versehen werden.

Um innerhalb von Prozeduren auf Namensraumvariablen zuzugreifen, muss in der Prozedur das entsprechende `variable`-Kommando verwendet werden, damit keine lokale Variable verwendet wird. Ob die Variable vorher auf Namensraumebene angelegt wurde oder nicht, spielt keine Rolle.

9.3 Verwalten von Namensräumen

Um festzustellen, welche Namensräume es gibt, kann man das Kommando `namespace children ?namespace? ?muster?` verwenden. Werden die beiden wahlfreien Parameter nicht angegeben, werden alle Namensräume geliefert, die Kinder des aktuellen Namensraums sind. Wird ein Namensraum als Parameter (mit absolutem Pfad, also beginnend mit `::`, oder relativ) mitgegeben, werden dessen Kinder geliefert. Hierbei kann man die Liste mittels des Musters auf bestimmte Namen einschränken, wobei das Verfahren von `string match` verwendet wird.

Namensräume können mittels `namespace delete` gelöscht werden, was in der Praxis sehr selten vorkommt.

9.4 Ex- und Importieren von Prozeduren in Namensräumen

Das Kommando `namespace export` *`prozedur ...`* legt fest, welche Kommandos (Prozeduren) aus dem aktuellen Namensraum exportiert werden, also später in andere Namensräume importiert werden können. Das Importieren hat den Vorteil, dass die Prozeduren mit einfachem Namen, also ohne Namensraumangabe, verwendet werden können.

`namespace import` *`namespace`*`::`*`prozedur ...` dient zum Importieren von Prozeduren aus anderen Namensräumen in den aktuellen. Alternativ dazu kann man auch mittels `namespace path` *`namespace-Liste`* eine Liste von Namensräume angeben, in denen nach einem einfachen (also ohne Namensraum) aufgerufenen Kommando durchsucht werden sollen – vergleichbar dem Suchmechanismus der Unix-Shell bezogen auf Verzeichnisse oder dem Java-Classpath.

9.5 Weitere Namensraum-Kommandos

In Tabelle 9.5 auf Seite 51 sind weitere Kommandos aufgeführt, die min Namensräumen zu tun haben.

Kommando	Bedeutung
`namespace current`	liefert den absoluten Pfad des aktuellen Namensraums.
`namespace children ?namespace? ?muster?`	liefert Liste alle Unter-Namensräume des aktuellen Namensraums. Wird ein Namensraum angegeben, so werden die Unter-Namensräume dessen geliefert. Die Ergebnisliste kann mittels eines Musters eingeschränkt werden.
`namespace code script`	liefert den Code in `script` so, dass er bei zukünftigen Aufrufen im aktuellen Namensraum ausgeführt wird. Ist bei Callbacks in GUI-Programmen sehr hilfreich.
`namespace export ?muster? ...`	richtet alle auf das `muster` passenden Kommandos des aktuellen Namensraums so ein, dass sie von anderen Namensräumen importiert werden können. Ohne Angabe von `muster` wird eine Liste der exportierten Kommandos geliefert.
`namespace import ?-force? ?muster? ...`	importiert die auf `muster` passenden Kommandos in den aktuellen Namensraum. Ohne Angabe von `muster` wird eine Liste der importierten Kommandos geliefert. Das `muster` muss hier ein qualifizierter Names, also der Name eines Namensraums, gefolgt von `::` und einem Prozedurnamenmuster sein. Durch die Angabe von `-force` werden bereits vorhandene Kommandos durch importierte Kommandos ersetzt.

caption Einige Unterkommandos des Kommandos `namespace`.

10 Packages

In Packages kann man wiederverwendbaren Code ablegen, der von allen oder nur bestimmten Applikationen genutzt werden kann. Auch eine Versionsverwaltung ist vorhanden, d. h. man kann eine bestimmte Version eines Packages verlangen (Zusatz `-exact` bei `package require`). Packages können in Dateien mit der Namenserweiterung `.tcl` abgelegt werden, oder in Moduldateien, welche eine Versionsangabe im Namen haben und die Namenserweiterung `.tm` tragen. In beiden werden Pakete mittels `package provide packageName ?versionsNr?` bereitgestellt. Sie unterscheiden sich nur im Erstellen und im Suchvorgang bei der Verwendung mittels textttpackage require *paketName ?versionsNr?*. Versionsnummern bestehen aus durch Punkt getrennten Dezimalzahlen, oft zwei Zahlen durch einen Punkt getrennt. Auch (höchstens) ein `a` (für Alphaversionen) und `b` (für Betaversionen) ist in der Versionsangabe anstelle eines Punktes erlaubt.

Die Manual Page hierzu erreicht man über `man package`.

Viele Packages bestehen nicht nur aus Tcl-Code, sondern auch aus in C geschriebenen Funktionen. Das gilt auch für viele Packages aus dem Core. Dieser Code liegt dann in Form von `.so`-Dateien (Linux, Mac) oder `.DLL`-Dateien (Windows) vor. Die Namenserweiterung dieser Shared Libraries kann man mittels `info sharedlibextension` erfragen. Solche Bibliotheken werden mittels `load` geladen, was vom `pkg_mkIndex` entsprechend in die Indexdatei eingetragen wird.

10.1 Packages mit Indexdateien

Indexdateien sind das ältere Konzept, werden aber auch in aktuellen Versionen von Tcl von viel verwendet. Um ein Package mit Indexdatei zu erzeugen, schreibt man in eine Datei mit der Namenserweiterung `.tcl` das Kommando `package provide` *`packageName versionsnummer`*] – ggf. mehrfach, wen in einer Datei mehrere Pakete definiert werden. Meist werden mehrere Paketdateien in einem Unterverzeichnis abgelegt. Damit der Interpreter die Pakete leichter finden kann, werden hierzu Indexdateien angelegt. Dies geschieht durch `pkg_mkIndex` *`verzeichnis`*, welches die Datei `pkgIndex.tcl` erzeugt. Die Manual Page hierzu erreicht man über `man pkg_mkIndex`.

Alle Verzeichnisse mit `pkgIndex.tcl`-Dateien werden in die globale Variable `auto_path` aufgenommen. Wird nun in einem Tcl-Script das Kommando `package require` *`packageName`* (ggf. gefolgt von einer Versionsnummer) auferufen, wird das entsprechende Modul geladen. Wie bei anderem Quellcode auch findet lediglich ein Parse-Vorgang statt, aber der Code wird erst später in Bytecode compiliert, wenn die jeweilige Prozedur aufgerufen wird. Auf diese Weise können viele Module verwendet werden, ohne dass der Startvorgang von Tcl-Scipts zu sehr verzögert wird. Ggf. kann man benötigte Pakete auch erst später laden, denn es ist nicht notwendig, alle `package require`-Kommandos am Scriptanfang einzutragen.

10.2 Packages in Moduldateien

Moduldateien, die ebenfalls das `package provide`-Kommando verwenden, haben einen festgelegten Namen: *`paketName-versionsNr`*`.tm`. Hierdurch können problemlos verschiedene Versionen nebeneinander existieren, weil es keine Namenskollision gibt. Ohne Angabe einer Versionsnummer wird von aufrufenden Scripts immer die neueste Version (höchste Nummer) verwendet. Die Versionsnummern setzen sich meistens aus zwei Zahlen mit einem Punkt dazwischen zusammen. Das Modul für eine Bildschirmverwaltung könnte beispielsweise `screen-2.3.tm` heißen, aber auch `screen-2.4a6` für die sechste Alphaausgabe der Version 2.4.

Um ein Verzeichnis zum Suchpfad für Moduldateien hinzuzufügen, wird `::tcl::tm::path add` *`modulVerzeichnis1 ...`* verwendet. Die Manual Page hierzu erreicht man über `man tm`.

10.3 Suche nach Packages

Leider sieht man nur den Moduldateien an ihrem Namen an, welche Pakete sie bereitstellen (`provide`). Daher kann es schon einmal mühsam sein herauszufinden, wo eine bestimmte Prozedur überhaupt definiert wurde. An den Code und die Parameterliste gelangt man aber immer mittels Introspection (`info`-Kommando).

Um die genaue Datei zu bestimmen, welche geladen wird, lädt man ein Package, beispielsweise mit `package require jpeg`. Dieses Kommando gibt die gefundene Version zurück, beispielsweise 0.5. Diese Versionsangabe benötigt man für das folgende Kommando, welches das Kommando `source` gefolgt vom Pfad der Datei zurückgibt: `package ifneeded jpeg 0.5`.

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`. Tatsächlich kann man sogar Datumsarithmetik betreiben, wobei die ganzen Besonderheiten der Zeitzonen, Sommer-/Winterzeit, Schaltsekunden und weiteres komplett berücksichtigt werden.

Viele Datei-Funktionen verstecken sich hinter `file`, siehe Tabelle 7.2.4 auf Seite 50. Dateien nach Namensmuster suchen kann man mit `glob`.

Nur unter Windows gibt es eine „Registry“. Auf diese kann man zugreifen und dort Werte lesen und schreiben mit dem Package `registry`. Für weitere Windows-spezifische Funktionen gibt es das Paket `twapi`[23].

[23] http://twapi.sourceforge.net/

12 Weitergehendes

Tcl kann noch viel mehr als das in diesem Grundlagendokument beschriebene. Insbesondere gibt es noch folgendes:

Internationalisierung: Mithilfe von Message-Katalogen wird die Entwicklung mehrsprachiger Anwendungen unterstützt. Durch die Verwendung von Unicode gelingt dies ohne Klimmzüge auch mit nichteuropäischen Sprachen, das zugehörige Paket heißt `msgcat`.

TclOO: Objektorientierung gibt es in Tcl schon lange, aber auf verschiedene Weise und mithilfe verschiedener Erweiterungen ([incr Tcl], snit, XOTcl). Seit Version 8.6 ist nun TclOO im Kern enthalten, welches längerfristig ziemlich sicher die anderen Varianten ablösen wird. Bei TclOO gibt es Klassen mit Einfach- und Mehrfach-Vererbung, Mixins, Konstruktoren, Destruktoren, Methoden und Filtern. Neben Klassen kann auch einzelnen Objekten zur Laufzeit eine neue Methode oder Eigenschaft hinzugefügt oder auch entfernt werden.

Socket-Programmierung: Über Sockets kann man Anwendungen über TCP/IP-Netze kommunizieren lassen. In Tcl wurden schon große Webanwendungen geschrieben, beispielsweise FlightAware.[24]

GUI: Mit dem Tk aus Tcl/Tk kann man multiplattformfähige GUI-Applikationen schreiben. Neben den Standard-Widgets gibt es mehrere erweiterte Widget-Sets, z. B. BWidgets und Ttk, welche seit Version 8.5 im Kern enthalten sind. Hinzu kommen so hilfreiche Erweiterungen wie Widget Callback Package (Wcb), Multi-Entry Widget Package (Mentry) und das Multi-Column and Tree Widget Package (Tablelist).

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 Applets, ist aber nicht an eine Browserumgebung gebunden, sondern kann überall verwendet werden.

mod_dtcl, mod_tcl, rivet, woof: 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, über SCGI wird das Web Oriented Object Framework (woof) angesprochen.

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.

[24] http://de.flightaware.com/

Datenbanken: Mit Tcl kann man auf viele Datenbanken direkt (u. a. PostgreSQL und Oracle) und auf alle anderen mit ODBC zugreifen. SQLite spielt besonders gut mit Tcl zusammen, denn es wurde zunächst als Tcl-Erweiterung implementiert und hat erst später Schnittstellen zu anderen Programmiersprachen bekommen. Der Autor von SQLite, D. Richard Hipp, ist ein begeisterter Anwender von Tcl. Seit Version 8.5 gibt es `tdbc`, eine genormte Schnittstelle unabhängig von den konkreten Datenbanksystemen. Hiermit ist ein Wechsel von einem Datenbanksystem zu einem anderen oder die Unterstützung mehrerer Systeme durch eine Anwendung nochmals deutlich erleichtert. Bei PostgreSQL kann man Tcl sogar im Server verwenden und darin Prozeduren schreiben, z. B. für Trigger.

Für Interessierte sei auf jeden Fall empfohlen, nach dem Stichwort „Tcl“ im Web zu suchen und auch bei `http://tcl.tk`. Es gibt überwältigend viel Material hierzu.