

Tcl/Tk — Eine Einführung

Andreas Bück

Andreas.Bueck@student.uni-magdeburg.de

überarbeitet und erweitert
unvollständige Vorabversion

5. Februar 2004

Inhaltsverzeichnis

1 Die Skriptsprache Tcl	4
1.1 Ein erstes Beispiel	4
1.2 Variablen	5
1.3 Mathematik in Tcl	7
1.4 Einlesen von Werten	11
1.5 Kontrollstrukturen	12
1.6 Dateibehandlung	16
1.7 Zeichenketten	19
1.8 Prozeduren	22
1.9 Listen	26
1.10 Prozesse	31
1.11 Verschiedenes	32
2 Das Tk-Toolkit	34
2.1 Einleitung	34
2.2 Grundlegende Widgets	35
2.2.1 Klick mich! – Das <i>button</i> -Widget	35
2.2.2 Fensterteilung – Das <i>frame</i> -Widget	37
2.2.3 Her mit den Eingaben! – Das <i>entry</i> -Widget	38
2.2.4 Das Kind braucht einen Namen! – Das <i>label</i> -Widget	39
2.2.5 Das <i>message</i> -Widget	41
2.2.6 Das <i>listbox</i> -Widget	41
2.3 Die Layoutmanager	43
2.3.1 Der Layoutmanager <i>pack</i>	43
2.3.2 Der Layoutmanager <i>grid</i>	46
2.3.3 Widgets platzieren mit <i>place</i>	47
2.4 Das Key-Binding	47

Einleitung

Dieses Tutorial soll eine Einführung in die Programmierung mit der Scriptsprache Tcl und dem GUI-Kit Tk sein. Es wird hier weniger Wert auf trockene Theorie gelegt, sondern mehr auf einfache verständliche Beispiele, an denen das Prinzip dieser Sprache erläutert werden soll.

Tcl ist eine Programmiersprache, die weit verbreitet ist, aber selten in den Vordergrund tritt, wie beispielsweise PERL oder Python. In der Verbindung mit Tk, einer Erweiterung für die grafische Programmierung ist sie jedem Linux-Nutzer bestimmt schon einmal begegnet: Die Programme *tkinfo*, *tkman*, *tkdraw* sind zu großen Teilen in Tcl/Tk programmiert worden. Auch Teile des Spieles „*Pingus - A Quest For Hering*“ sind in Tcl geschrieben worden. Ein weiteres Beispiel ist die grafische Oberfläche beim Aufruf *make xconfig* zur Erstellung eines eigenen Linux-Kernels.

Wenn jetzt der Eindruck entstanden sein sollte, das Tcl/Tk nur was für Linux sei, dem ist nicht so. Tcl/Tk läuft unter diversen Unix-Versionen, Linux, Windows und MacOS. Das Gute daran ist, daß ein einmal geschriebenes Programm, so fern es keine systemspezifischen Befehle benutzt, auf all diesen Softwareplattformen ohne Veränderungen läuft. Das gilt sowohl bei der Benutzung von reinen Tcl-Programmen als auch bei der Benutzung von Tk.

Bei Tcl/Tk handelt es sich um eine interpretierte Sprache, daß heißt das Programm liegt in Textform vor und wird erst bei der Abarbeitung in den ausführbaren Binärcode umgewandelt. Dadurch sind sie etwas langsamer als kompilierte Programme, was aber in unseren Tagen bei Rechnern jenseits der 3-Gigahertz-Grenze minimalst ausfällt. Bei kommerziellen Tcl/Tk-Programmen stellt sich dann aber das Problem, daß dort der Quellcode/Sourcecode für jeden einsehbar/veränderbar ist. Dies kann man aber mit dem Programm *freewrap* von DENNIS R. LABELLE umgehen, das den Quelltext in verschlüsselter Form in einer allein lauffähigen Datei speichert.

Ein Anlaufpunkt rund um Tcl/Tk ist die folgende Web-Seite, die Informationen und auch die Tcl/Tk-Interpreter für die verschiedenen Systeme zu Verfügung stellt:

`www.tcl.tk`

Voraussetzungen

Das Erlernen einer neuen Programmiersprache setzt natürlich einige Kleinigkeiten voraus. In den meisten Fällen ist es, daß man diesen oder jenen Compiler, diese oder jene Megahertz-Zahl hat und und und... Meist ist das Ganze auch noch mit Kosten verbunden. Doch mit Tcl haben Sie Glück: der Interpreter für diese Sprache ist frei erhältlich für nahezu jedes Betriebssystem. Linux-Benutzer finden ihren Interpreter auf den Distributions-CDROMs. Benutzer anderer Betriebssysteme werden unter folgender Adresse fündig: *www.tcl.tk*. Desweiteren ist Tcl auch auf Maschinen mit wenigen 100 MHz sehr wohl lauffähig und außer einem beliebigen Texteditor wird nur noch die Bereitschaft gefordert diese

Sprache zu erlernen.

Es wird aber davon ausgegangen, daß Sie mindestens die 5. Klasse einer beliebigen Schulform erreicht haben und wissen, was Variablen beziehungsweise Konstanten sind. Falls nicht empfehle ich eine Auffrischung mit Hilfe eines Mathematikbuches ihrer Wahl. Trotzdem ein kleiner Rat von mir: Verlieren Sie die Mathematik nicht aus den Augen. Denn bei der Programmierung erzielen sich meist die ersten Erfolge durch die Übertragung mathematischer Formeln (oder auch physikalischer Formeln) in Programme. Auch setze ich einfach mal voraus, daß Sie wissen, wie man eine Datei öffnet, speichert, kopiert und löscht.

Falls Sie zufällig Erfahrungen in der C-Programmierung haben sollten (was aber nicht erforderlich ist), so nehmen Sie sich jetzt 5 Sekunden Zeit und freuen sich, denn dadurch haben Sie es leichter diese Sprache zu erlernen, da sie sich an C orientiert. Für alle anderen wird etwas schwieriger, aber nicht unerschaffbar.

Doch jetzt ist's genug mit Palaveri, jetzt wollen wir endlich etwas tun.

1 Die Skriptsprache Tcl

1.1 Ein erstes Beispiel

Nun wollen wir ein einfaches Beispiel in die Tat umsetzen und dabei gleich etwas über Tcl lernen. Wer schon mal mit Programmiersprachen zu tun hatte, kennt es bestimmt, das berühmte „Hello World!“-Programm. Und was gibt es Besseres als an diese Tradition anzuknüpfen und es in Tcl umzusetzen...

puts, tclsh, #

Hier also nun der Quellcode:

```
# Code Anfang

puts "Hello World!"

# Code Ende
```

Speichern Sie diesen Quellcode in einer Datei und rufen Sie den Tcl-Interpreter auf:

```
tclsh Dateiname
```

und bestaunen Sie das Ergebnis, das zwar unerwartet, aber deswegen nicht weniger schön Ihnen vom wenig augenschonenden 50 Hz-Monitor entgegenflimmert. (Unter Umständen ist der Aufruf des Interpreters auf Ihrem System anders, z.B. `tclsh82` oder `tclsh74` also verbunden mit der jeweiligen Tcl-Versionsnummer.)

Ein einfaches „Hello World!“ mit nur einer Zeile Code, das ist doch schon was, oder? Und hier sehen Sie auch unseren ersten Befehl:

```
puts Argument
```

Dieser Befehl gibt alles, was in Anführungsstrichen steht (erstmal) auf dem Bildschirm aus. Spielen Sie ruhig ein wenig mit diesem Befehl. Es gibt zwar nicht viele verschiedene Verwendungsmöglichkeiten, aber trotzdem... mit diesem Befehl ist es möglich Kontakt mit dem Benutzer auf zu nehmen, denn woher soll sie/er wissen, was sie/er machen soll, wenn ihn das Programm nicht informiert?

Wie Sie sehen, brauchen Sie (erstmal) keine besonderen Regeln bei der Erstellung einer Datei beachten. Ein weiteres Merkmal von Tcl haben Sie auch schon gesehen: Kommentare werden mit einem Doppelkreuz (oder Raute, oder einfach #) eingeleitet. Alles was danach folgt, wird vom Interpreter ignoriert. Hier können Sie Ihren Code ausführlich dokumentieren ohne das dadurch der Programmablauf gestört wird.

Im nächsten Abschnitt werden wir uns mit Variablen beschäftigen, da die Ausgabe von Texten zwar reizvoll, aber auf die Dauer etwas ermüdend ist.

1.2 Variablen

*set, \$, global,
unset*

Variablen sind in Tcl alle vom Typ String, daß heißt der Interpreter nimmt erstmal jede Variable und ihren Wert als Zeichenkette auf. Auf den ersten Blick ist dies sehr vorteilhaft, da man sich nicht um Wertebereiche kümmern muss, auf den zweiten Blick aber muss bei rein arithmetischen Anwendungen darauf geachtet werden, denn hier müssen die Zeichenketten erst in Zahlen umgewandelt werden, damit diese dann auch addiert, subtrahiert, etc. werden können.

Nun ein paar kleine Beispiele, wie man Variablen vereinbart und ihnen Werte zuweist. In der Regel tut man dies gleichzeitig.

```
# Code Anfang

set autor "Andreas Bueck"
set pi 3.1415926
set zahl 5

# Code Ende
```

Im ersten Beispiel wird die Variable *autor* mit dem Wert *Andreas Bück* belegt, im zweiten die Variable *pi* mit der Zahl Pi und das dritte dürfte dann selbsterklärend sein.

Bei der Delaration von Variablen ist bei der Namensgebung folgendes zu beachten: Jede Variable muss mit einem Buchstaben beginnen und sollte keine Umlaute (ä, ö, ü) oder das „ß“ beinhalten. Unterstriche („_“) sind im Variablennamen erlaubt, Leerzeichen dagegen nicht.

Um nun zum Beispiel den Wert einer Variablen auszugeben, bedient man sich des Kommandos *puts* gefolgt von dem Variablennamen. Allerdings ist hierbei zu beachten, daß wenn man den Wert einer Variablen ausgeben möchte dem Namen ein „\$“ voranzustellen ist. Aus *pi* wird dann also *\$pi*.

```
# Code Anfang
```

```
set pi 3.1415926
puts $pi
```

```
# Code Ende
```

Immer wenn man auf den Wert einer Variablen zugreifen möchte, sei es nun um ihn auszugeben, oder um ihn in Berechnungen zu verwenden so ist dem Namen immer ein `$` voranzustellen AUßER sie geben der Variablen mit dem Befehl `set` einen neuen Wert. Dann folgt wie im Beispiel ersichtlich kein `$`.

Wie in vielen Programmiersprachen gibt es auch in Tcl lokale und globale Variablen. Doch was hat es damit auf sich? Folgendes: Lokale Variablen sind nur in den Bereichen sicht- und veränderbar in denen sie definiert sind. Globale Variablen sind überall sicht- und editierbar. Das macht jetzt noch wenig Sinn und verwirrt vielleicht, aber zu einem späteren Zeitpunkt (wenn wir Prozeduren behandeln) wird der Unterschied deutlicher. In Tcl sind standardmäßig alle Variablen lokal, möchte man eine globale Variable schaffen, so tut man dies mit dem Befehl `global` gefolgt vom Variablennamen:

```
global Variablennamen
```

Die globalen Variablen (hier dargestellt durch *Variablennamen*) werden außerhalb aller anderen Prozeduren vereinbart. In einer Prozedur wird durch oben genannten Aufruf die Variable in der Prozedur mit der globalen verbunden und keine neue lokale Variable geschaffen. Alle Änderungen wirken sich dann direkt auf diese Variable aus.

Ein weiterer wichtiger Befehl in Verbindung mit Variablen ist der Befehl `unset`. Er erwartet als Parameter einen Variablennamen. Er entfernt dann diese Variable aus dem Programm, daß heißt sie existiert dann im folgenden Programmablauf nicht mehr. Probieren Sie doch einmal folgenden Code aus:

```
# Code Anfang
```

```
set name "Andreas"
set zahl 3.4
```

```
puts $name
puts $zahl
```

```
unset name
```

```
puts $name
```

```
# Code Ende
```

Wie sie sehen erhalten Sie bei der Abarbeitung des Skripts folgenden Fehler:

```
can't read "name": no such variable
```

Dies ist der Beweis dafür, daß die Variable *name* entfernt wurde und nicht mehr auf sie zugegriffen werden kann. Der Aufruf von *unset* ist folgender:

```
unset Variablenamen
```

Auch hier ist zu beachten, daß kein *\$* vor dem Variablennamen steht, denn das Dollar-Zeichen besitzt die Bedeutung „Wert von“ und dies macht dann natürlich auch keinen Sinn.

Spielen sie nun ein wenig mit diesen neuen Möglichkeiten, testen Sie sie bevor wir uns im nächsten Punkt dann der Mathematik in Tcl widmen.

1.3 Mathematik in Tcl

expr, incr

Mathematik ist natürlich auch mit Tcl möglich. Sie beschränkt sich nicht auf einfache Additionen und Subtraktionen sondern man kann mit Tcl auch Sinus, Kosinus, Exponential- und Logarithmusfunktionen berechnen. Also los geht's. Als erstes wollen wir natürlich mit etwas Einfachem beginnen, doch vorher müssen wir noch einmal auf die Variablen eingehen. Wie im vorherigen Abschnitt beschrieben, sind alle Variablen in Tcl erstmal Strings. Doch wie soll man dem Interpreter jetzt klar machen, daß es sich um Zahlen handelt? Dies ist ganz einfach. Man benutzt die Anweisung *expr*. Sie macht dem Interpreter klar, daß ein mathematischer Ausdruck folgt und das die folgenden Variablen Zahlen-Werte beinhalten und keine Zeichenketten. Um das Ganze zu verdeutlichen ein kleines Beispiel (eine einfache Addition):

```
# Code Anfang

set a 5.0
set b 3.0

set c [expr $a + $b]

puts $c

# Code Ende
```

Die Zeile, die hier von Interesse ist, ist *set c [expr a+b]*. Sie ist im Grunde nichts Neues. Wir erinnern uns: Der *set*-Befehl erwartete einen Variablennamen und einen Wert. Die eckigen Klammern veranlassen den Tcl-Interpreter, daß dieser Ausdruck zuerst ausgewertet und berechnet wird und das Ergebnis dann als zweiter Parameter an den *set*-Befehl übergeben wird. Probieren Sie es ruhig einmal ohne eckige Klammern, Sie werden eine Fehlermeldung erhalten. Berechnungen werden also in der Regel wie folgt formuliert:

```
set Ergebnis [expr $A Op.-Zeichen $B Op.-Zeichen ...]
```

Wobei Op.-Zeichen für ein beliebiges (Tcl bekanntes) Operationszeichen steht. Die bekanntesten sind hierbei natürlich *+*, *-* für Addition und Subtraktion, ***, */* für Multiplikation und Division. Nun ein weiteres Beispiel um auf eine Eigenheit von Tcl hinzuweisen:

```

# Code Anfang

set a 3
set b 4

set c [expr $a / $b ]
set d [expr $b / $a ]

puts $c
puts $d

# Code Ende

```

Lassen sie dieses Skript einmal durchlaufen und ... Überraschung! Die Ergebnisse dürften sehr wohl von dem abweichen, was Sie erwarteten. Statt 0.75 und 1.3333 erscheinen nämlich 0 und 1 als Ergebnisse. Woran das liegt? Ganz einfach: bei der Übergabe von ganzen Zahlen ohne Nachkommastellen in eine Berechnung, geht Tcl davon aus, daß es sich um Integer-Werte handelt und rundet bei Division und Multiplikation auf die nächste ganze Zahl. Um dies zu umgehen und die erwarteten Werte zu erhalten ist es also notwendig, die Nachkommastellen mit anzugeben. Ersetzen Sie im obigen Script einfach 3 durch 3.0 und 4 durch 4.0 und lassen Sie das Skript noch einmal durchlaufen. Und siehe da, die Ergebnisse stimmen!

Ein weiterer mathematischer Befehl ist *incr*. Er erwartet einen Variablennamen und erhöht den Wert der Variablen um 1. Es ist jedoch darauf zu achten, daß es sich um eine ganzzahlige Variable (also ohne Nachkommastellen) wie zum Beispiel die Zahl 7 handelt. Dazu ein Beispiel:

```

# Code Anfang

set a 5
puts $a

incr a
puts $a

# Code Ende

```

Sie erhalten die Ausgaben 5 und 6. Als nächstes wollen wir etwas höhere Mathematik mit Tcl umsetzen. Sicherlich kennen Sie den *Satz des Pythagoras*, der in etwa aussagt, daß die Summe der Quadrate der Flächen über den Katheten gleich der Fläche des Quadrates über der Hypothenuse in einem rechtwinkligen Dreieck ist. Oder einfach $h^2 = k_1^2 + k_2^2$ mit k_1 und k_2 als Katheten des Dreiecks.

Diese Formel wollen wir jetzt in ein Tcl-Skript umsetzen. Dabei werden viele Dinge auftauchen, die wir bisher schon behandelt haben, aber auch Unbekanntes, das dann anschließend erklärt wird.

```

# Code Anfang

```

```

set a 3.0
set b 4.0

set c [expr sqrt(pow($a,2) + pow($b,2))]

puts $c

# Code Ende

```

Lassen Sie dieses Skript abarbeiten und Sie erhalten das (richtige) Ergebnis 5.0. Sehen wir uns noch einmal die berechnende Zeile noch einmal an:

```
set c [expr sqrt(pow($a,2) + pow($b,2))]
```

Was geschieht hier? Was klar sein sollte, ist daß der Variablen *c* der Wert des mathematischen Ausdrucks in der eckigen Klammer zugewiesen wird. Aber dann? Es ist eigentlich ganz einfach: der berechnende Ausdruck ist $\sqrt{\text{pow}(\$a,2) + \text{pow}(\$b,2)}$, was wiederum nichts anderes ist, als Wurzel aus *a* zum Quadrat plus *b* zum Quadrat. Die Funktion $\text{pow}(x, y)$ liefert als Ergebnis *x* in der *y*-ten Potenz, also x^y . In diesem Fall ist $y = 2$, daher wird also x^2 berechnet. Die Funktion $\text{sqrt}(z)$ bildet dann die Quadratwurzel von der Zahl *z*. In unserem Fall also von der Summe aus a^2 und b^2 . Es gibt viele mathematische Funktionen, die in Tcl bereits vordefiniert sind. Nun eine kleine Aufzählung der wichtigsten:

Name	Funktion
$\text{abs}(x)$	bildet den Betrag von <i>x</i> ($ x $)
$\text{cos}(x)$	Cosinus von <i>x</i> (<i>x</i> im Bogenmaß)
$\text{int}(x)$	schneidet die Nachkommastellen von <i>x</i> ab
$\text{log}(x)$	natürlicher Logarithmus von <i>x</i>
$\text{log } 10(x)$	dekadischer Logarithmus von <i>x</i>
$\text{pow}(x,y)$	berechnet x^y
$\text{sin}(x)$	Sinus von <i>x</i> (<i>x</i> im Bogenmaß)
$\text{sqrt}(x)$	Quadratwurzel von <i>x</i> ($x \geq 0$)
$\text{tan}(x)$	Tangens von <i>x</i> (<i>x</i> im Bogenmaß)

Desweiteren gibt es noch diverse Umkehrfunktionen von Winkelfunktionen. Zum Beispiel $\text{acos}(x)$ (Arcuskosinus), $\text{asin}(x)$ (Arcussinus), $\text{atan}(x)$ (Arcustangens). Zu beachten bei den Winkelfunktionen ist, daß die Werte immer im Bogenmaß sein müssen. Für all diejenigen die nicht wissen, wie das Bogenmaß errechnet wird, hier die Formel:

$$\text{Bogenmaß} = \frac{\text{Gradzahl} * \pi}{180}$$

Damit dürfte auch den Winkel-Spielereien nichts mehr im Wege stehen. Zum Abschluß noch ein kleines Beispiel zu den Winkelfunktionen:

```

# Code Anfang

set pi 3.1415926
set grad 30.0

```

```

set bogen [expr ($grad*$pi)/180.0]
set sinus [expr sin($bogen)]

puts $sinus

# Code Ende

```

Nun wollen wir uns einem letzten Thema im Abschnitt Mathematik widmen, den Vektoren. Diese kann man leicht als Feld (*array*) auffassen. Vielleicht ist Ihnen ja die grundlegende Vektor-Rechnung ein Begriff. Man notiert dort Vektoren in der folgenden Form:

$$\vec{v} = \begin{pmatrix} 1 \\ 0 \\ 4 \end{pmatrix}$$

Nun stellen wir uns einmal vor, wir wollten diesen Vektor in einem Tcl-Programm darstellen. Wir könnten jetzt 3 Variablen vx, vy und vz vereinbaren und ihnen die Werte 1, 0 und 4 zuweisen. Das mag bei einem Vektor mit nur 3 Elementen gehen, was aber wenn er 9, 12 oder 25 Elemente besitzt? Hier kann man sehr leicht die Übersicht verlieren und somit gab es einen Grund zur Schaffung und Benutzung von Feldern. Ein Array besteht aus einer Menge von Elementen (hier sind es 3, nämlich 1, 0 und 4) die unter einem Namen (hier v) gesammelt werden. Um die einzelnen Elemente unterscheiden zu können, bekommt jedes seinen eigenen Namen. Das folgende Beispiel erzeugt das Array v, daß den Vektor in Tcl repräsentiert.

```

# Code Anfang

set v(x) 1.0
set v(y) 0.0
set v(z) 4.0

# Code Ende

```

Wir besitzen nun das Array v mit den 3 Einträgen x, y und z. Um den Begriff des Arrays noch zu verdeutlichen: Stellen Sie sich vor, Sie haben einen Weihnachtskalender. Die Türen sind alle durchnummeriert von 1 bis 24. Nun können Sie zum Beispiel den Schokoladenvorrat im Kalender erfassen. Das geht (unübersichtlich) mit 24 Variablen oder einem Feld. Sehen sie sich dazu folgendes Beispiel an. (Angenommen es ist der 5.12 und Sie öffnen nicht vorzeitig ein Türchen.)

```

# Code Anfang

set kalender(1) 0
set kalender(2) 0
set kalender(3) 0
set kalender(4) 0
set kalender(5) 0
set kalender(6) 1

# Code Ende

```

Damit haben Sie jetzt also im Feld kalender erfasst, daß hinter den Türchen 1 bis 5 keine Schokolade mehr vorhanden ist, wohl aber hinter Türchen 6. Mehrdimensionale Felder lassen sich nur über einen kleinen Umweg realisieren. Ein mehrdimensionales Feld hat folgenden Aufbau:

```
set Feldname(Dimension1,Dimension2,...) Wert
```

Es wird empfohlen nach den Kommas keine Leerzeichen zu lassen, da es dadurch mitunter zu Schwierigkeiten kommen kann.

Dies soll genug der Mathematik sein, aber sie wird uns in den Beispielen noch weiter beschäftigen. Als nächstes geht es mit dem Einlesen von Zahlen und Zeichen weiter.

1.4 Einlesen von Werten

gets, stdin

Bisher hatten all unsere Beispiele nur feste Variablen-Werte. Der Nutzer konnte also bei Berechnungen zum Beispiel keine eigenen Werte eingeben, ohne den Quelltext verändern zu müssen. Dies wollen wir jetzt ändern. Dieser Abschnitt zeigt wie man Zahlen oder Zeichenketten einliest um sie dann weiter zu verwenden.

Das Einlesen von Zahlen und Werten gestalten sich relativ einfach. Man benutzt dazu den Tcl-Befehl *gets* gefolgt von der Quelle, von der gelesen werden soll. Das ist meistens die Tastatur und die trägt in Tcl den Namen *stdin* (für Standard Input). Um das Eingelesene in Variablen zu speichern gibt es zwei Möglichkeiten. Die erste ist die schon bekannte Zuweisung mit Hilfe von *set*:

```
set c [gets stdin]
```

Dies speichert das Eingegebene in die Variable *c*. Die andere Möglichkeit ist direkt über den Befehl *gets*:

```
gets stdin Variablenname
```

Der Befehl *gets* liest solange von der angegebenen Quelle, bis ein Enter-Zeichen eingelesen wird. In unserem Beispiel also bis Enter gedrückt wird. Nach dem Druck von Enter gibt *gets* - aufgerufen mit der zweiten Möglichkeit - die Länge des eingelesenen Wertes zurück. Dabei ist zu beachten, daß *gets* das abschließende Enter-Zeichen stets mitliest und somit auch mitzählt. Damit ist die Länge der Eingabe um eins kleiner als die angezeigte Länge.

Im nun folgenden Beispiel wollen wir den *gets*-Befehl nutzen:

```
# Code Anfang
```

```
puts "Berechnung der Flaeche eines Rechtecks"
puts ""
puts "Fl = a * b"
puts "Geben Sie bitte nacheinander folgen Werte fuer a und b ein "
puts ""
```

```

puts "a = "
gets stdin a
puts "b = "
set b [gets stdin]
set Fl [expr $a*$b]
puts ""
puts -nonewline "Die Flaeche betraegt: "
puts $Fl

# Code Ende

```

Das Aussehen der Ausgaben ist nicht so toll, aber in seiner Funktionsweise ist es uneingeschränkt: Bei der Eingabe von zwei Zahlen berechnet es uns die Fläche des Rechteckes.

An dieser Stelle noch ein Wort zum Befehl *puts*. Er kann bis zu 3 Parameter haben. Der erste ist der Parameter *-nonewline*. Dieser verhindert einen Zeilenvorschub nach Ausgabe des Textes oder der Zahl. Er muss nicht benutzt werden. Der zweite Parameter nennt sich *ChannelId* und gibt an, wohin *puts* ausgeben soll. Standardmäßig ist der Bildschirm eingestellt, aber es besteht die Möglichkeit diese Ausgabe umzulenken.

Auch hier wieder die Aufforderung ein wenig mit der Ein- und Ausgabe zu spielen. Wie wäre es mit einem Programmlein, daß den Namen und Alter einer Person einliest und dann wieder ausgibt? Zugegeben nicht unbedingt sinnvoll aber zum Üben ideal...

Im nächsten Abschnitt werden wir uns mit den grundlegenden Kontrollstrukturen beschäftigen, die Tcl bereitstellt.

1.5 Kontrollstrukturen

*if, for, while,
foreach*

Kontrollstrukturen sind Elemente, die den Programmablauf in Abhängigkeit von bestimmten Werten beeinflussen. Dazu ein kleines Beispiel: Nehmen wir einmal an Sie wollten ein Programm entwickeln, daß der Einfachheit halber einfach nur die Wurzel aus einer eingegebenen Zahl zieht. Wie Sie wissen ist die Wurzel aus negativen Zahlen im Bereich der reellen Zahlen nicht definiert. Was also passiert, wenn eine negative Zahl eingegeben wird? Das Programm stürzt ab. Damit so etwas nicht passiert gibt es Kontrollstrukturen. Und genau diese wollen wir in diesem Abschnitt näher berachten.

Eine der wichtigsten Kontrollstrukturen ist die *if-Abfrage*. Sie überprüft eine ihr übergebene Bedingung und führt von deren Wahrheitsgehalt abhängig bestimmte Anweisungen durch.

Zunächst einmal den genauen Aufbau dieser Kontrollstruktur:

```

if { Bedingung } {
    Befehle wenn Bedingung wahr
} else {

```

```

        Befehle wenn Bedingung falsch
    }

```

Der else-Block kann entfallen, wenn keine Befehle ausgeführt werden sollen, wenn sich die Bedingung als falsch erweist. Die Bedingungen werden mit Hilfe von Variablen und den mathematischen Relationszeichen (>, <, >=, <=, == (für Gleichheit), != (für Ungleichheit)) formuliert. Bei dieser und bei allen folgenden Kontrollstrukturen ist darauf zu achten, daß die geschweifte Klammer des Körpers in der selben Zeile steht wie die zu untersuchende Bedingung. Diese Schreibweise führt zu einer Fehlermeldung:

```

    if { Bedingung }
    {
        Anweisungen
    }

```

Nun der Quellcode des eingangs beschriebenen Programms:

```

# Code Anfang

puts "Bitte geben Sie eine positive Zahl (>= 0) ein:"
set c [gets stdin]

if { $c < 0 } {
    puts "Sie sollten doch eine positive Zahl eingeben!"
} else {
    set erg [expr sqrt($c)]
    puts -nonewline "Die Wurzel betraegt: "
    puts $erg
}

```

Code Ende

Dieses Programm akzeptiert als gültige Eingaben nur Zahlen, die entweder 0 oder größer sind. Ansonsten wird der Text „*Sie sollten doch eine positive Zahl eingeben!*“ ausgegeben.

Es ist bei dieser Kontrollstruktur möglich bestimmte Bedingungen logisch zu einer zu verknüpfen. Ein Beispiel: Sie wollen den Bereich für das Programm weiter einschränken, so daß zum Beispiel nur Zahlen zwischen 5 und 25 gültige Eingaben sind. Sie müssten also zweimal die Eingabe abfragen. Dies ginge ohne weiteres mit verschachtelten *if-Abfragen*. Zum Beispiel:

```

if { $c >= 5 }
{
    if { $c <= 25 }
    {
        Wurzelziehen
    } else {

```

```

        Fehlermeldung
    } else {
        Fehlermeldung
    }
}

```

Oder aber Sie verknüpfen beide Bedingungen logisch. Dabei stehen folgende logische Operatoren zur Verfügung: `&&` (UND), `||` (ODER) und `!` (NICHT). Damit verkürzt sich die *if-Abfrage* und bleibt übersichtlicher.

```

    if { ($c >= 5) && ($c <= 25) } {
        Wurzelziehen
    } else {
        Fehlermeldung
    }
}

```

Wie man sieht müssen bei der logischen Verknüpfung, die einzelnen Bedingungen in runde Klammern gefasst werden. Nun wollen wir diese Struktur in unser Programm übertragen:

```

# Code Anfang

puts "Bitte geben Sie eine positive Zahl (>= 0) ein:"
set c [gets stdin]

if { ($c >= 5) && ($c <= 25) } {
    set erg [expr sqrt($c)]
    puts -nonewline "Die Wurzel betraegt: "
    puts $erg
} else {
    puts -nonewline $c
    puts " liegt nicht im Bereich."
}

# Code Ende

```

Beim Ausführen des Programmes werden Sie erkennen, daß das Programm nur Werte zwischen 5 und 25 akzeptiert. Es lassen sich auch Zeichenketten mit einander vergleichen, wie das folgende Beispiel zeigt:

```

# Code Anfang

puts "Bitte geben Sie ihren Namen ein: "
set name [gets stdin]

if { $name != "Andreas Bueck" } {
    puts "Sie sind nicht der Autor."
} else {
    puts "Sie sind der Autor dieses Tutorials."
}

# Code Ende

```

Nur wenn Sie meinen Namen eingeben, erhalten Sie die Meldung, daß Sie (aber eigentlich ich, weil Sie dann ich, ich aber nicht Sie bin) der Autor sind. Dies sollte zur if-Abfrage genügen. Kommen wir zur nächsten. Es handelt sich hierbei um die while-Schleife. Sie hat folgenden Aufbau:

```
while { Bedingung } {
    Anweisungen, die ausgeführt werden
    so lange Bedingung wahr ist
}
```

Dies Schleife läuft solange bis die formulierte Bedingung nicht mehr wahr ist. Auch hierzu ein kleines praktisches Beispiel: Sie möchten ein Programm schreiben, daß solange Zahlen einliest und addiert, bis die Zahl 0 eingegeben wird. Dann sieht das Programm folgendermaßen aus:

```
# Code Anfang

set zahl 1.0
set erg 0.0

while { $zahl != 0 } {
    set erg [expr $erg + $zahl]
    set zahl [gets stdin]
}

set erg [expr $erg - 1]
puts -nonewline "Gesamtergebnis: "
puts $erg

# Code Ende
```

Jetzt werden Sie sich sicher fragen warum wir der Variablen den Wert 1.0 übergeben. Das hat folgenden Grund: Standardmäßig werden alle neuen Variablen in Tcl mit dem Wert Null vorbelegt. Damit wird allerdings die Bedingung in der *while-Schleife* falsch und sie wird somit nicht ausgeführt. also setzen wir die Variable auf 1.0 und können dann in die Schleife einsteigen. In der Variablen erg wird das Gesamtergebnis gespeichert, das beträgt nach dem Eintritt in die Schleife und der Abarbeitung der ersten Anweisung dann durch den Inhalt von zahl schon 1. Deshalb müssen wir dann am Ende diese 1 wieder abziehen um das korrekte Ergebnis zu erhalten.

Auch in der *while-Schleife* können Bedingungen logisch verknüpft werden und in einer *while-Schleife* können auch weitere *while-Schleifen* und *if-Abfragen* vorkommen.

Die nächsten beiden Schleifen-Arten sind einander ähnlich, da sie beide nur eine bestimmte Anzahl oft durchlaufen werden. Man spricht hier von sogenannten Zählschleifen. Die beiden Schleifen um die es geht sind die for- und die foreach-Schleife. Die *foreach-Schleife* wird an dieser Stelle nur erwähnt und später im Abschnitt *Listen* behandelt. Die *for-Schleife* ist also eine Zählschleife. Sie hat folgenden Aufbau:

```

    for { Startwert } { Abbruchwert } { Veraenderung } {
        auszufuehrende Anweisungen
    }

```

Auch hierzu ein Beispiel: Sie möchten die Zahlen von 0 bis 1E+06 (1 Million) ausgeben. Dafür gibt es zwei Möglichkeiten: Die erste ist, das Sie sich von all Ihren Verwandten verabschieden, einen Liefervertrag mit Ihrem Lebensmittelhändler abschließen und dann unbestimmte Zeit damit verbringen mit Hilfe von *puts* die Zahlen auszugeben. (Man verzeihe mir den Anflug von Humor.) Die andere Möglichkeit ist, es mit einer for-Schleife zu realisieren. Dazu hier der Quelltext:

Code Anfang

```

for { set i 0 } { $i <= 1000000 } { incr i } {
    puts $i
}

```

Code Ende

Dieses Programm leistet das Gewünschte. Wie man sieht wird in der ersten Anweisung die Variable *i* auf Null gesetzt. In der zweiten Anweisung wird der Abbruchwert festgelegt. Hier soll der Abbruch erfolgen wenn *i* größer als 1.000.000 ist. Mit der dritten Anweisung wird festgelegt, wie sich die Variable *i* von Durchlauf zu Durchlauf ändern soll. Hier soll der Wert jeweils um eins erhöht werden. Dann erst schließen sich die Anweisungen an, die die bestimmte Anzahl mal ausgeführt werden sollen.

Experimentieren Sie ruhig mit diesen Kontrollstrukturen. Der Tcl-Interpreter macht Sie auf jeden Fehler aufmerksam. Entwickeln Sie zum Beispiel ein Programm, das die beliebige (ganzzahlige) Potenz einer Zahl durch fortlaufende Multiplikation berechnet. Einen Lösungsvorschlag finden Sie in *mul.tcl*.

1.6 Dateibehandlung

In jeder Programmiersprache ist es möglich Dateien zu lesen oder zu schreiben. Das geht natürlich auch mit Tcl. Doch wir wollen noch mehr. Wir werden in diesem Abschnitt auch noch erfahren, wie man Informationen über bestimmte Dateien erhält, oder wie man bestimmte Dateien in einem Verzeichnis findet.

open, close,
read, glob,
pwd, file

Doch zunächst erstmal etwas Einfaches: Wir wollen als erstes eine kleine Datei zum Lesen öffnen und ihren Inhalt auf dem Bildschirm wiedergeben. Um eine Datei zu öffnen benutzt man den Befehl *open* der einen Dateinamen und einen Zugriffsmodus erwartet. Gültige Zugriffsmodi sind in der folgenden Tabelle dargestellt:

Modus	Bedeutung
r	existierende Datei wird zum Lesen geöffnet
r+	Datei wird zum Lesen und Schreiben geöffnet
w	öffnet Datei zum Schreiben; existiert die Datei so wird ihr Inhalt gelöscht, sonst wird eine neue Datei angelegt
w+	öffnet zum Lesen und Schreiben; sonst alles wie „w“
a	öffnet zum Schreiben und setzt Schreibmarke an das Datei-Ende; existiert die Datei noch nicht wird sie angelegt
a+	öffnet zum Lesen und Schreiben; Rest wie „a“

Der Befehl *open* hat einen Rückgabewert, die *fileId*, über die wir auf die Datei zugreifen können. Diese *fileId* ist also unbedingt in einer Variablen zu speichern. Mit dem Befehl *gets* können wir dann die Datei zeilenweise auslesen, indem wir statt *stdin* die *fileId* an *gets* übergeben. Mit dem Befehl *eof* gefolgt von der *fileId* können wir überprüfen ob das Dateiende erreicht wurde. Mit *close* gefolgt von *fileId* wird der Datei-Zugriff beendet. Das wollen wir gleich an einem kleinen Beispiel zeigen:

```
# Code Anfang

set fileId [open "mul.tcl" r]

while { (eof $fileId) == 0 } {
    set zeile [gets $fileId]
    puts $zeile
}

close $fileId

# Code Ende
```

Der Wert von *eof \$fileId* bleibt solange Null bis das Dateiende erreicht wird. Dann wird der Wert auf 1 gesetzt und die Bedingugn stimmt nicht mehr. Damit haben wir einen kleinen Textreader, der Dateien auf der Standard-Ausgabe ausgibt. Bei langen Texten allerdings kommt es zu einem Zeilenvorschub, der das Betrachten dann behindert. Also modifizieren Sie das Programm so, daß es nach 22 Zeilen anhält, auf einen Druck auf Enter wartet und dann die nächsten 22 Zeilen ausgibt. Einen Lösungsvorschlag finden Sie in *reader.tcl*.

Kommen wir nun zum Schreiben in eine Datei. Dies ist ähnlich simpel. Man öffnet eine Datei zum Schreiben und fügt dann den Text mit *puts \$fileId Text* in die Datei ein. Dazu auch hier ein Beispiel:

```
# Code Anfang

set fileId [open "xxx.yyy" w]

puts $fileId "Hello World"
puts $fileId "-----"
```

```
close $fileId
```

```
# Code Ende
```

Wie Sie sehen ist die Dateibehandlung mit Tcl relativ einfach. Vielleicht noch eine Bemerkung: Mit *gets* wird die Datei nur zeilenweise ausgelesen. Möchte man die Datei zeichenweise auslesen, so tut man dies mit *read \$fileId* gefolgt von der Anzahl der zu lesenden Bytes. Möchte man alles einlesen so lässt man die Byte-Zahl weg. Hier kurz die Syntax und ihre Anwendung:

```
set inhalt200 [read $fileId 200]
set alles [read $fileId]
```

Unser nächstes Thema in der Dateibehandlung ist der *glob*-Befehl. Er findet über ein Suchmuster spezifizierte Dateien (ähnlich dem *dir*-Befehl). Ein kleines Beispiel um das ganze zu verdeutlichen. Das nächste Skript zeigt alle Dateien im aktuellen Verzeichnis an. Das aktuelle Verzeichnis können Sie übrigens mit dem Befehl *pwd* in Erfahrung bringen.

```
# Code Anfang
```

```
set verzeichnis [ pwd ]
```

```
glob $verzeichnis/*.*
```

```
# Code Ende
```

Bei der Dateibehandlung ist darauf zu achten, daß das Verzeichnistrennzeichen bei Tcl der *Slash* (/) und nicht wie unter DOS der Backslash. Nun haben die Möglichkeit uns jede Datei in jedem Verzeichnis anzeigen zu lassen.

Als nächstes wollen wir einige Informationen über unsere Dateien einholen. Dies geschieht mit dem Befehl *file* gefolgt von einem Parameter und dem Dateinamen der Datei, die wir untersuchen wollen. Es folgt nun eine Liste der wichtigsten Parameter des Befehls *file*:

delete	löscht die übergebene Datei, mit <i>-force</i> können nicht-leere Verzeichnisse gelöscht werden
dirname	trennt den Verzeichnisnamen vom Dateinamen
exist	gibt 1 zurück wenn die Datei existiert, 0 wenn nicht
extension	liefert die Dateiendung ab dem letzten Punkt
isdirectory	liefert 1 wenn es sich um ein Verzeichnis handelt
size	liefert die Größe in Bytes
type	gibt die Art der Datei an: <i>characterSpecial</i> , <i>blockSpecial</i> , <i>socket</i> , <i>fifo</i> , <i>link</i> , <i>directory</i> oder <i>file</i>
writable	liefert 1 wenn die Datei beschreibbar ist

Hier ist zu beachten, daß unter Betriebssystemen die über eine Rechteverwaltung verfügen (z. B. Unix-Derivate) einige Parameter (z. B. *delete*) auf bestimmte Dateien nicht angewendet werden können, da die nötigen Benutzerrechte fehlen.

Nun werden wir ein kleines Programm entwickeln, daß einen Dateinamen einliest und dann Informationen über die Datei ausgibt.

```
# Code Anfang

puts "Bitte geben Sie einen Dateinamen ein: "
set name [gets stdin]

if { [file exists $name] == 1 } {
  puts ""
  puts "Datei-Informationen \[$name\]"
  puts ""
  set c [file extension $name]
  puts "Dateiendung: $c"
  set c [file isdirectory $name]
  if { $c == 1 } { puts "Verzeichnis: ja" }
  else { puts "Verzeichnis: nein" }
  set c [file size $name]
  puts "Dateigroesse: $c"
  set c [file type $name]
  puts "Dateityp: $c"
  set c [file writable $name]
  if { $c == 0 } {puts "Schreibschutz: ja" }
  else { puts "Schreibschutz: nein" }
}

# Code Ende
```

Wie man sieht, läßt sich auch diese Aufgabe sehr kurz lösen. Es geht bestimmt noch kürzer, aber für uns als Einstieg soll es reichen.

Damit wären wir auch schon am Ende vom Abschnitt Datei-Behandlung angekommen, auch an dieser Stelle wieder die Aufforderung zu experimentieren, aber auch der Hinweis anfangs nur auf eine Diskette zu schreiben oder von dort zu lesen, nicht daß nacher noch irgendwelche wichtigen Daten gelöscht werden, nur weil man statt *r w* schrieb. Das wäre erstens schade und zweitens nicht wieder gut zu machen. Also Achtung beim Öffnen von Dateien und ganz wichtig auch das *Schließen* nicht vergessen.

Im nächsten Abschnitt werden wir uns ganz kurz mit Strings (Zeichenketten) beschäftigen, wie man Zeichenketten formatiert ausgibt und Strings manipuliert.

1.7 Zeichenketten

Bevor einige grundlegende Möglichkeiten kennen lernen Strings zu verändern, sollten man vielleicht klären, was ein String ist. Ein String ist eine Aneinanderreihung von Elementen der ASCII-Tabelle. (Sie kennen die ASCII-Tabelle nicht? Dann schauen Sie mal bitte im Informatikteil ihres Tafelwerkes nach (so fern dieses einen hat).) Hier wird der Zeichensatz des PCs für Europa und Amerika festgelegt. Ein String ist also eine Aneinanderreihung aus Buchstaben, Zahlen

*format, incr,
string, split*

und anderen Zeichen. Daher auch der treffliche deutsche Name Zeichenkette.

Mehr wollen wir über Zeichenketten theoretisch auch gar nicht wissen. Das Thema dieses Abschnittes ist einmal die formatierte Ausgabe von Strings und deren Manipulation.

Als erstes wollen wir die formatierte Ausgabe behandeln. Mit dieser ist es möglich die Ausgabe der Strings auf dem Bildschirm unterschiedlich zu gestalten. Der *format*-Befehl (keine Angst, er hat eine völlig andere Funktion als der DOS-Befehl) orientiert sich hierbei am *sprintf*-Befehl aus *ANSI C* und ist praktisch identisch mit ihm. Der *format*-Befehl hat nur untergeordnete Bedeutung in Tcl, trotzdem soll er hier beschrieben werden.

Eine Anwendungsmöglichkeit ist die folgende: Sie möchten eine Werte-Tabelle ausgeben und möchten, daß die Zahlen bündig untereinander stehen. Dies kann man mit dem *puts*-Befehl nicht ohne weiteres erreichen, aber mit dem *format*-Befehl. Der *format*-Befehl hat folgenden Aufbau:

`format TEXT Argumente`

Der Text *TEXT* enthält die Formatierungs-Anweisungen. Diese werden durch die Werte der Argumente ersetzt. Dazu dieses Beispiel: Es soll eine formatierte Tabelle der Werte $f(x) = x^3, x \in [0, 10]$ ausgegeben werden:

```
# Code Anfang

puts "   x       x**3"

for {set i 0} { $i <= 10 } { incr i } {
    puts [format "%4.1f %10.1f" $i [expr pow($i,3)]]
}

# Code Ende
```

Dieses Beispiel reserviert für das erste Argument 4 Zeichen von denen eins eine Kommastelle ist und 10 Stellen für das zweite Argument, von denen auch wieder eine für eine Kommastelle reserviert ist. Es gibt folgende *format*-Codes: *%d* für Ganzzahlen, *%f* für gebrochene Zahlen, *%x* für Hexadezimalzahlen, *%e* für Zahlen in Exponentialschreibweise und *%c* für einzelne Zeichen. Dazu ein abschließendes Beispiel, die Ausgabe der ASCII-Tabelle, welche wir der Einfachheit halber in die Datei *ascii.tab* ausgeben.

```
# Code Anfang

set fileId [open "ascii.tab" w]

puts $fileId " Integer   ASCII   Integer   ASCII   Integer   ASCII"
for { set i 32 } { $i <= 254 } { incr i +4 } {
    puts $fileId [format "%3d    %c    %3d \
%c    %3d    %c    %3d    %c" $i $i \
[expr $i + 1] [expr $i + 1] [expr $i + 2]\
```

```
[expr $i + 2] [expr $i + 3] [expr $i + 3] \  
[expr $i + 4] [expr $i + 4]]  
}
```

```
close $fileId
```

```
# Code Ende
```

Noch ein paar Worte zum Quellcode: Er enthält einige bisher nicht besprochene Neuerungen. Dort wäre zum ersten die Verwendung von `\`. Dieses Zeichen muss benutzt werden, wenn ein Kommando über eine Zeilenlänge hinaus geht. Es teilt dem Interpreter mit, daß der Befehl in der nächsten Zeile witergeht.

Eine andere Neuerung bezieht sich auf den *incr*-Befehl. Bisher wurde nur besprochen, daß dieser eine Variable um 1 erhöht. Er kann auch um mehr als 1 erhöhen und sogar die Werte verringern. Es gilt dann folgende Syntax:

```
incr Variablenname +Zahl oder  
incr Variablenname -Zahl
```

Der Rest dürfte bekannt sein. Das soll nun endgültig zum *format*-Befehl reichen. Als Nächstes wollen wir uns um die Stringmanipulation kümmern. Aber auch hier werde ich nur die grundlegendsten Möglichkeiten erläutern. Zeichenketten-Manipulationen geschehen mit Hilfe des Befehles *string* gefolgt von einem Parameter und einem String oder einer einen String enthaltenden Variablen. Folgende Parameter möchte ich an dieser Stelle vorstellen:

compare	vergleicht zwei Strings miteinander
length	gibt die Länge des Strings zurück (Anzahl der Zeichen)
tolower	wandelt alle Buchstaben des Strings in Kleinbuchstaben um
toupper	wandelt alle Buchstaben in Großbuchstaben um

Diese Parameter wollen wir an einigen Beispielen testen:

```
# Code Anfang
```

```
set textk "hier ist alles klein"  
set textg "HIER IST ALLES GROSS"  
  
set c [string compare $textk $textg]  
puts "Ergebnis des Vergleiches: $c"  
  
set c [string length $textk]  
puts "L"ange von textk: $c"  
  
set c [string toupper $textk]  
puts "Ergebnis nach toupper: $c"  
  
set c [string tolower $textg]  
puts "Ergebnis nach tolower: $c"  
  
# Code Ende
```

Das Ergebnis von *string compare* ist numerisch. 1 bedeutet String1 ist größer String2, -1 bedeutet String1 ist kleiner als String2 und 0 bedeutet String1 ist gleich String2. Alle diese Relationen beziehen sich auf eine lexiographische Betrachtungsweise. Zur Verdeutlichung ein paar Beispiele:

```
# Code Anfang

set a [string compare "Mayer" "Maier"]
set b [string compare "Meier" "Meier"]
set c [string compare "Meier" "Meyer"]

puts "$a $b $c"

# Code Ende
```

Im ersten Vergleich ist String1 größer als String2, da das „y“ unterschiedlich ist und später im Alphabet auftaucht. In Vergleich 3 ist String1 kleiner, da „i“ vor „y“ steht.

Als letzten Befehl für die Bearbeitung von Zeichenketten möchte ich den *split*-Befehl vorstellen. Mit ihm ist es möglich eine Zeichenkette beim Auftreten eines bestimmten Zeichens (Zeichenfolge) aufzuteilen (zu splitten). Die Syntax ist auch wieder recht einfach:

```
split $String $Trennung
```

\$String ist die Zeichenkette, die nach den in *\$Trennung* stehenden Zeichen durchsucht wird und bei ihrem Auftreten an diesen Stellen aufgeteilt wird. Standardmäßig voreingestellt ist das Leerzeichen. Ein Beispiel:

```
# Code Anfang

split "abcde/fghi/jklmn" "/"

# Code Ende
```

Hier erhalten wir eine Liste mit 3 Einträgen (Elementen). Was Listen sind und was man damit machen kann, erkläre ich später. Die Elemente sind „abcde“, „fghi“ und „jklmn“. So kann man auch komplexere Eingaben in einfache Elemente zerlegen. In späteren Beispielen werden wir davon Gebrauch machen.

Im nächsten Abschnitt wird die Verwendung von Prozeduren erläutert. Wie und warum man Prozeduren schreibt und noch einiges mehr.

1.8 Prozeduren

Prozeduren bieten die Möglichkeit einmal entwickelte Algorithmen in vielen Programmen wiederzuverwenden ohne diese noch einmal neu schreiben zu müssen. Sie bieten ausserdem die Möglichkeit den Quellcode eines Programmes übersichtlicher erscheinen zu lassen, was die Fehlersuche und eine Erweiterung der Programme erleichtert. Auch reduzieren sie den Arbeitsaufwand. Nehmen Sie

proc, return,
source

zum Beispiel die Prozedur *prim* aus der Datei *math.tcl*. Sie ist schon etwas länger als alle Beispiele, die wir bisher behandelt haben. Sie überprüft ob eine Zahl eine Primzahl ist oder nicht. Stellen Sie sich nun einmal vor, Sie würden ein Programm schreiben, in dem Sie an mehreren Stellen Zahlen darauf prüfen müssen, ob es Primzahlen sind. Sie müssten dann den Quelltext der Prozedur *prim* jedesmal an diesen Stellen neu einfügen. Das führt zu einer Unübersichtlichkeit und damit auch zu einer höheren Fehleranfälligkeit. Deshalb legen wir diesen Algorithmus als Prozedur ab und können ihn dann ganz bequem an den geforderten Stellen über seinen Namen wie einen gewöhnlichen Tcl-Befehl aufrufen.

Eine Prozedur hat einen ganz bestimmten Aufbau, nämlich folgenden:

```
proc PName { Parameterliste } {  
    Anweisungen  
}
```

Das Wörtchen *proc* teilt dem Interpreter mit, daß es sich bei dem folgenden um eine Prozedurdefinition handelt. *PName* ist der Name mit dem Sie die Prozedur später aufrufen wollen. *Parameterliste* enthält die Parameter, die der Prozedur übergeben werden sollen. Zur Veranschaulichung ein kleines Beispiel:

```
# Code Anfang  
  
proc Hallo { name } {  
  
    puts "Guten Tag $name!"  
}  
  
Hallo Andreas  
  
# Code Ende
```

Mit diesem Code legen wir eine Prozedur *Hallo* an, die einen Parameter (*name*) erwartet. Mit dem Aufruf *Hallo Andreas* führen wir den Befehl *Hallo* aus und übergeben ihm den Parameter *Andreas*. Wir sind jetzt also in der Lage eigene Befehle zu schaffen. Als nächstes wollen wir dieses noch ein wenig an Beispielen festigen.

Als erstes wollen wir einen Befehl schaffen, der uns mitteilt ob eine Zahl *z* Teiler der natürlichen Zahl *p* ist. Dabei möchte ich Ihnen an dieser Stelle einen weiteren mathematischen Operator vorstellen, die *Modulo-Division*. Sie wird durch das Zeichen *%* ausgedrückt. Was es mit der Modulo-Division auf sich hat? Bewegen wir uns ganz kurz zurück in die Grundschule. Wenn hier Aufgaben der Division (z.B. $9 : 4$) nur mit Hilfe der natürlichen Zahlen zu lösen waren, so behalf man sich in dem man schrieb „*Neun dividiert durch 4 ist 1 Rest 5.*“ Und dieser Rest interessiert jetzt. Bei einer *Modulo-Division* erhält man als Ergebnis den ganzzahligen Rest einer *Integer-Division*. Was ist jetzt wieder eine *Integer-Division*? Bei dieser Art der Division sind Dividend und Divisor Ganzzahlen und das Ergebnis auch. Es kommt hier also nicht zu gebrochenen Zahlen sondern zu

einer Ganzzahl und einem ganzzahligen Rest. (Im Beispiel: 1 ist die Ganzzahl und 5 der ganzzahlige Rest). Zum Üben nun ein paar Aufgaben:

$$14 : 5, 27 : 3, 14 : 23, 40 : 7, (2 + 4) : 3$$

Als Lösung für die erste Aufgabe erhält man 2 mit dem Rest 4, da $2 \cdot 5 + 4 = 14$. Mit dieser Hilfe dürften die anderen Aufgaben kein Problem mehr darstellen. Für unseren Befehl nutzen wir den Befehl folgendermaßen: Eine (Ganz-)Zahl ist Teiler einer anderen, wenn bei der Modulo-Division ein Rest von Null das Ergebnis ist. Wir prüfen also ob das Ergebnis Null ist und geben in Abhängigkeit davon eine Meldung aus.

Code Anfang

```
proc teiler { p z } {  
  
    set erg [expr $p % $z]  
  
    if { $erg == 0 } {  
        puts "$z ist Teiler von $p."  
    } else {  
        puts "$z ist nicht Teiler von $p."  
    }  
}  
  
teiler 14 5  
teiler 27 3  
teiler 2 12
```

Code Ende

Nachdem wir die Prozedur *teiler* erstellt haben, können wir sie wie einen Tcl-Befehl aufrufen. Wie Sie sehen, liefert die Prozedur korrekte Ergebnisse. Was ist aber, wenn wir Ergebnisse an eine Variable übergeben wollen? Dies muss auf anderem Weg erfolgen, da die gewohnte Anweisung

```
set Variable [ Befehl Parameter ]
```

so nicht funktioniert. Der Grund hierfür ist, daß unsere Prozedur keine Werte zurückliefert, wie zum Beispiel eine Funktion. Einer Funktion wird ein Wert übergeben und sie liefert den Funktionswert. Wenn es also gewünscht ist, daß eine Prozedur Werte an Variablen zurückgibt, wir also das Ergebnis der Prozedur in einer Variablen speichern wollen, so ist es notwendig, die Prozedur dahingehend zu modifizieren, daß Werte zurückgegeben werden.

Wie eigentlich alles in Tcl lässt sich auch dies leicht realisieren. Tcl stellt für die Rückgabe von Werten einen eigenen Befehl zur Verfügung. Dieser trägt den Namen *return*. Der *return*-Befehl verlässt die innerste aller Prozeduren (oder die Prozedur) mit dem übergebenen Wert als Rückgabewert. Mit *return* läßt sich immer nur ein Wert zurückgeben, ähnlich einer Funktion in C, wenn man dort nicht mit Zeigern arbeitet. Will man mehr als einen Wert zurückgeben, so

bietet es sich an mit globalen Variablen zu arbeiten und diese nach dem Prozeduraufruf auszulesen.

Um die Funktionsweise von Prozeduren noch zu verdeutlichen, ein weiteres Beispiel an dieser Stelle: Sie erinnern sich noch an das Programm *mul.tcl* mit dem wir die y -te Potenz der Zahl x durch wiederholte Multiplikation berechneten. Dieses wollen wir jetzt a) in eine Prozedur umwandeln und b) das Ergebnis in einer Variablen sichern. Es folgt jetzt der bereits veränderte Programmcode, der originale ist in *mul.tcl* zu finden.

Code Anfang

```
proc potenz { x y } {  
    set erg 1.0  
  
    for { set i 1 } { $i <= $y } { incr i } {  
        set erg [expr $erg*$x]  
    }  
  
    return $erg  
}
```

```
set Ergebnis [ potenz 2 3 ]
```

```
puts "Das Ergebnis ist $Ergebnis."
```

Code Ende

Mit der Anweisung *return \$erg* wird das Ergebnis der Prozedur an eine Variable weitergereicht, mit der dann ganz normal weitergearbeitet werden kann. Wenn wir unsere selbstgeschriebene Prozedur in Berechnungen verwenden wollen, so ist folgender Weg zu wählen:

```
set Variable [expr [PName] Op [R]]
```

Wobei *PName* der Name unserer Prozedur ist, *Op* das Operationszeichen und *R* sämtliche andere Rechenoperationen beschreibt. Oder bezogen auf unser Beispiel (wir addieren einfach zum Wert unserer Prozedur einfach 6, zerlegt in die Multiplikation $2*3$):

```
set c [expr [potenz 2 3] + [expr 2*3]]
```

Wie man sieht sind Prozeduren sehr nützlich. Tcl bietet die Möglichkeit Programme auch modular aufzubauen. Das bedeutet, daß Sie nicht alle Prozeduren in die Programmdatei schreiben müssen, sondern auf mehrere Dateien verteilen können und diese dann in das Programm einbinden können. Was das bedeutet? Nehmen wir mal an man möchte ein Programm schreiben, das physikalisch-chemische Berechnungen ausführt. Dafür benötigen wir Formeln aus dem Bereich der Chemie und der Physik, die (angenommen) in einer Tcl-Datei auf unserem Rechner schlummert. Desweiteren nehmen wir an, daß es sich um eine Vielzahl von Formeln und Dateien handelt, so daß es nicht sinnvoll ist jede

einzelne Formel in unser Programm zu kopieren. Tcl bietet hierfür eine komfortable Lösung: Man kann mit Hilfe des *source*-Befehls diese Dateien in unser Programm einfach einbinden (wie der *#include*-Mechanismus in C).

Er hat diese Syntax:

```
source "Dateiname"
```

Das Resultat ist, daß Tcl diese Datei in das Hauptprogramm einliest und die dort vorhandenen Prozeduren auch im Hauptprogramm zugänglich macht. Diese Technik wollen wir mit Hilfe der mitgelieferten Datei *math.tcl* demonstrieren. *math.tcl* enthält verschiedene geläufige mathematische Funktionen und Lösungsalgorithmen. Im Beispiel wollen wir die Nullstelle der linearen Funktion $y = 4 * x - 4$ berechnen. Die benötigte Funktion ist in *math.tcl* definiert. Wir brauchen diese Datei also einfach nur einbinden und die Funktion mit den Parametern aufrufen.

```
# Code Anfang
```

```
source "math.tcl"
```

```
set Ergebnis [lin 4.0 -4.0]
```

```
puts "Nullstelle bei x= $Ergebnis"
```

```
# Code Ende
```

Dies kürzt unser Programm doch erheblich. Wenn wir jetzt zum Beispiel einen Fehler in der Ausgabe bemerken, wissen wir gleich, daß dieser nur im Hauptprogramm sein kann, da *lin* aus *math.tcl* nur rechnet und keine Ausgabe macht.

Damit möchte ich den Abschnitt „Prozeduren“ beenden. Sie wissen jetzt, wie Prozeduren aufgebaut sind, wie man Werte zurückgibt und welche Möglichkeit es gibt, ein Programm übersichtlicher zu gestalten und es modular aufzubauen.

Im nächsten Abschnitt werden wir uns mit Listen beschäftigen, denen in Tcl viel Raum eingerichtete wurde. Auch hier werden Sie einen Überblick über die Möglichkeiten erhalten, die Ihnen die Listen bieten.

1.9 Listen

Jeder von uns kennt Listen, zum Beispiel eine Einkaufsliste. Sie ist eine Aufzählung von Dingen, die man bei einem Einkauf zu besorgen hat. Eine Liste ist also eine Ansammlung von Zeichenketten. Tcl kennt natürlich auch Listen, hier ist es aber egal, ob es sich beim Listeninhalt um Namen, Adressen oder Zahlen handelt, da Tcl ja alles erstmal als String auffasst. Mit Listen und deren Manipulation werden wir uns in diesem Abschnitt beschäftigen.

lappend
index,
length,
lsearch, *lsort*,
join, *concat*,
foreach

Eine Liste erzeugt man am einfachsten, indem man die Listeneinträge einer Variablen zuweist. Das erfolgt so ähnlich wie die Zuweisung eines Wertes an eine Variable. Eine Liste wird durch zwei geschweifte Klammern begrenzt. { öffnet die Liste und } schließt sie. Somit ist der Aufbau einer Liste der Folgende:

```
set liste { Element1 Element2 ... Elementn }
```

Mit dem folgenden Beispiel legen wir eine Liste mit Namen an. Wir nennen diese bezeichnenderweise *Namen*:

```
# Code Anfang
```

```
set Namen { Hans Rita Fritz Gitte Klaus Anne Karl Julia }
```

```
# Code Ende
```

Schön, jetzt haben wir eine Liste, was aber kann man damit tun? Eine ganze Menge. Tcl stellt eine Menge Befehle zur Listenmanipulation zur Verfügung. Es folgt jetzt eine Aufzählung der wichtigsten Befehle:

<code>concat liste1 liste2 ... liste3</code>	verbindet alle Listen zu einer resultierenden Liste
<code>join liste TString</code>	verbindet alle Elemente der Liste mit TString untereinander
<code>lappend liste Variable</code>	fügt Variable in Liste ein
<code>lindex liste Zahl</code>	liefert das Element, das an Stelle Zahl liegt (0 = 1. Element)
<code>list liste</code>	gibt alle Elemente der liste aus
<code>llength liste</code>	gibt die Anzahl der Elemente zurück
<code>lsearch -exact liste LMuster</code>	sucht in der Liste nach dem Muster LMuster, der Parameter <i>-exact</i> ist frei wählbar
<code>lsort Parameter liste</code>	sortiert die Liste nach der in Parameter stehenden Bedingung, mögliche Werte für Parameter sind: <i>-ascii</i> , <i>-integer</i> , <i>-real</i> , <i>-increasing</i> (aufsteigend), <i>-decreasing</i> (absteigend); standardmäßig gilt <i>-ascii</i> und <i>-increasing</i>

Die Verwendung dieser Befehle wird jetzt an einem Beispiel verdeutlicht:

```
# Code Anfang
```

```
set liste1 {1 5 2 9 3 6 8 0}
```

```
list $liste1
```

```
set liste_s [lsort -integer -increasing $liste1]
```

```
puts $liste_s
```

```
set liste2 {a b c d e f g h}
```

```
puts $liste2
```

```
set laenge [llength $liste2]
```

```
puts "L"ange von Liste 2: $laenge"
```

```
set result [concat $liste1 $liste2]
```

```
puts $result
```

```
set neu [lappend liste2 i j k l]
```

```
# Code Ende
```

Mit Hilfe der Erlarungen der einzelnen Befehle und den Ausgaben des Programmes, sollte es moglich sein die Anwendung und Funktionsweise der Befehle zu erfassen.

An dieser Stelle soll noch auf die bereits im Abschnitt *Kontrollstrukturen* erwahnte *foreach*-Schleife eingegangen werden. Die *foreach*-Schleife wird wie folgt benutzt:

```
foreach Var Liste {
  Anweisungen
}
```

Die Variable *Var* nimmt nacheinander die Werte, die in *Liste* stehen an und fuhrt dann die angegebenen Anweisungen aus. Die *foreach*-Schleife wird also so oft durchlaufen, wie *Liste* Elemente enthalt. Zur Verdeutlichung ein Beispiel:

```
# Code Anfang
```

```
foreach i {1 2 3 4 5 6 7 8 9 10} {
  puts "$i \t $i*$i"
}
```

```
foreach i {-4 1.2 "Ein String"} {
  puts "$i"
}
```

```
foreach i {1 2 3 4} {
  puts "Guten Tag!"
}
```

```
# Code Ende
```

Warum diese Schleife an dieser Stelle eingefuhrt wurde? Weil sie sich sehr gut fur die Arbeit mit Listen eignet. Zum Beispiel wird aus folgender *for*-Schleife

```
# Code Anfang
```

```
set liste {"Andi" "Andreas" "Andrew"}
for {set i 0} {$i <= [expr [llength $liste] - 1]} {incr i} {
  puts [string toupper [lindex $liste $i]]
}
```

```
# Code Ende
```

unter Verwendung der *foreach*-Schleife ubersichtlicher:

```

# Code Anfang

set liste {"Andi" "Andreas" "Andrew"}

foreach i $liste {
    puts [string toupper $i]
}

# Code Ende

```

An diesem Punkt wollen wir ein kleines Programm stricken, daß wir mal als CD-Verwaltung bezeichnen werden und folgende Eigenschaften besitzen soll:

- Einlesen einer (vielleicht) vorhandenen Datei cds.xxx
- Speicherung der Einträge in der Datei in eine Liste
- Möglichkeit zur Eingabe neuer CD-Titel
- Löschen von einzelnen Titeln
- Sortierung der resultierenden Liste
- Speicherung des Ganzen wieder in cds.xxx

Das Programm ist zu finden in der Datei *cdlib.tcl* und besteht aus den fünf Prozeduren *main*, *einlesen*, *schreibe*, *eingabe* und *ldelete*. Der Quelltext ist ausführlich dokumentiert und sollte somit recht gut zu verstehen sein.

Die Prozedur *ldelete* ist die am schwierigsten zu verstehende Prozedur in diesem Programm. Sie wird benötigt um CD-Titel aus der Liste zu entfernen. Darum hier gesondert der kommentierte Quelltext:

```

proc ldelete { liste eintrag } {

    # Stelle suchen, an der das zu loeschende Element
    # steht, wenn das Element nicht existiert wird -1
    # zurueckgegeben
    set index [lsearch -exact $liste $eintrag]

    # Eintrag existiert
    if { $index != -1 } {
        if { $index == 0 } {
            # Eintrag ist das erste Element der Liste
            # neue Liste wird gebildet, die das 2. bis
            # letzte Element enthaelt

            for { set i 1 } { $i <= [expr [llength $liste] - 1] } \
                {incr i} {
                set neu [lappend neu [lindex $liste $i]]
            }
            # Liste wird sortiert ...

```

```

        set neu [lsort $neu]
        # ... und zurueck gegeben
        return $neu
    }

if { $index == [expr [llength $liste] - 1] } {
    # der Eintrag ist das letzte Element der Liste
    # neue Liste wird gebildet, die alle bis
    # einschliesslich des vorletzten Elementes enthaelt

    for { set i 0 } { $i <= [expr [llength $liste] - 2] } \
    {incr i} {
        set neu [lappend neu [lindex $liste $i]]
    }
    # Liste sortieren ...
    set neu [lsort $neu]
    # ... zurueck geben
    return $neu
} else {
    # Element steht an Stelle j in der Liste
    # neue Liste wird mit den Elementen 1 bis j-1
    # gefuelllt

    for {set i 0} { $i < $index } {incr i} {
        set neu [lappend neu [lindex $liste $i]]
    }

    # zu der Liste werden noch die Elemente von
    # j+1 bis zum letzten Element angehaengt

    for { set i [expr $index + 1] } \
    {$i <= [expr [llength $liste] - 1]} {incr i} {
        set neu [lappend neu [lindex $liste $i]]
    }
    # sortieren ...
    set neu [lsort $neu]
    # ... zurueckgeben
    return $neu
}
}
}

```

In der Prozedur *main* wird eine Art Menü gestaltet, das den Benutzer unseres Programms die verschiedenen Optionen anbietet und dann die entsprechenden Aktionen ausführt.

Dies ist schon ein recht komplexes Beispiel, das fast alles von dem, was wir bisher behandelt haben, beinhaltet. Mit diesem Beispiel, daß ohne weiteres noch erweitert werden und an andere Sachverhalte angepasst werden kann (zum Beispiel für Bücher), möchte ich den Abschnitt *Listen* beenden.

Im folgenden Abschnitt werden wir uns mit Prozessen befassen, also der Möglichkeit Programme von Tcl aus zu starten und deren Ausgabe zu verwerten.

1.10 Prozesse

exec

Nicht immer ist es möglich oder sinnvoll ein Programm nur mit Tcl-Befehlen zu erstellen. Zum Beispiel, wenn es darum geht, eine Berechnungen durchzuführen. In solchen Fällen kann es günstiger sein mit Tcl die benötigten Werte einzulesen und an ein externes Programm zu übergeben. Dieses Programm verrichtet dann die Arbeit und wir lesen dann (wieder mit Tcl) das Ergebnis ein und können dieses dann weiterverarbeiten.

Tcl bietet die Möglichkeit externe Programme zu starten. Diesen Vorgang nennt man einen *neuen Prozess erzeugen*. Wie dieses geht und welche Möglichkeiten diese Technik bietet, soll in diesem Abschnitt gezeigt werden.

Um einen neuen Prozess zu erzeugen, bedient man sich des Befehles *exec*, mit dieser Syntax:

```
exec Programmname Parameter1 ... ParameterN
```

Dabei unterbricht der Tcl-Interpreter solange die Abarbeitung des Skripts, bis der mit *exec* gestartete Prozess beendet ist (also das Programm seine Aufgabe erledigt hat). Nach Beendigung liefert *exec* alle Ausgaben, die das aufgerufene Programm in die Standardausgabe geschrieben hat. Dazu ein Beispiel (Nr. 1 für Windows, Nr. 2 für Linux):

Beispiel 1:

```
# Code Anfang
```

```
set memory [exec mem]
puts $memory
```

```
# Code Ende
```

Beispiel 2:

```
# Code Anfang
```

```
set memory [exec cat /proc/meminfo]
puts $memory
```

```
# Code Ende
```

Mit diesen Zeilen lässt sich ganz leicht der vorhandene Systempeicher und dessen Belegung feststellen. Wenn das Programm keine Daten in die Standardausgabe schreibt, gibt das Programm eine leere Zeichenketten zurück.

Ein Anwendungsgebiet für Prozesse wäre zum Beispiel folgendes Szenario: Sie wollen ein Programm schreiben, daß sehr viele komplizierte Berechnungen in sehr wenig Zeit vornimmt. Nun handelt es sich bei Tcl um eine interpretierte Sprache, die immer noch etwas langsamer sind als kompilierte Programme. So

wäre es denkbar, daß Sie für die Berechnungen ein eigenes Programm schreiben, dem Sie dann die Parameter übergeben und später dann das Ergebnis wieder einlesen. Damit können Sie möglicherweise viel Zeit sparen. Oder Sie wollen von Tcl auf Geräte zugreifen, deren Benutzung nicht vorgesehen ist. Ein Beispiel wäre hierfür die Ansteuerung der Soundkarte, welche dann über ein externes Programm realisieren können.¹

Die Ausgabe von *exec* kann umgeleitet werden zum Beispiel mit *>ADatei*. Dies schreibt dann die Ausgaben in die neu erzeugte Datei *ADatei*. Es ist auch möglich direkt Daten aus einer mit Tcl geöffneten Datei direkt an das Programm zu übergeben, dies geschieht mit *<@fileId*. Standardmäßig bedeutet *>* eine Umleitung der Standardausgabe und *<* eine Umleitung der Standardeingabe. Mit *|* ist es möglich die Ausgabe des einen Programmes an ein anderes Programm weiterzuleiten. (Ähnlich dem DOS-Befehl *dir *.* | mode*)

Es gibt noch eine weitere Möglichkeit Prozesse zu starten und zwar mit dem bereits bekannten Befehl *open*. Damit legen wir Kanäle (Pipes) an über die wir dann mit *gets* und *puts* auf die Programme wie auf normale Dateien zugreifen können. Dazu ein kleines schematisches Beispiel:

```
set Pipe [open {|Befehl } Zugriffsart]
```

Mit *puts Variable \$Pipe* kann dann der Wert der Variablen bequem an das Programm übergeben werden.

Mit diesen kurzen Erläuterungen möchte ich den Abschnitt „Prozesse“ auch wieder beenden.

1.11 Verschiedenes

after, time

In diesem Abschnitt stelle ich kurz ein paar Befehle vor, die nützlich sind aber nicht so richtig in die anderen Abschnitte gepasst haben. Dort wären zum einen der *time*-Befehl, der *after*-Befehl sowie einige Escape-Sequenzen.

Den Anfang macht diesmal der *after*-Befehl. Dieser ermöglicht eine Unterbrechung des Programmablaufs für eine bestimmte Zeit. Der *after*-Befehl hat folgende Syntax:

```
after Zahl
```

Der *after*-Befehl unterbricht für *Zahl* Millisekunden den Programmablauf. Dies lässt sich für kleine Spielereien wie zum Beispiel verzögerte Ausgaben nutzen nutzen:

```
# Code Anfang
```

```
set c [gets stdin]
after 2000
puts $c
```

```
# Code Ende
```

¹Inzwischen gibt es auch Tcl-Erweiterungen mit denen man die Soundkarte ganz einfach ansteuern kann. Mit dem „reinen“ Tcl ist das nicht einfach möglich.

Als nächstes wollen wir uns mit einigen Escape-Sequenzen beschäftigen. Escape-Sequenzen werden durch einen Backslash eingeleitet und werden von einem Buchstaben gefolgt. Folgende Escape-Sequenzen gibt es:

<pre>\a</pre>	gibt eine Piepton aus
<pre>\n</pre>	bewirkt einen Zeilenumbruch
<pre>\t</pre>	ein horizontaler Tabulator
<pre>\v</pre>	ein vertikaler Tabulator
<pre>\\</pre>	erzeugt einen Backslash

Diese Escape-Sequenzen können und sollen mit dem *puts*-Befehl verwendet werden. Anstatt 5 mal `puts ""` zu schreiben um 5 Leerzeilen zu erhalten, genügt jetzt `puts "\n\n\n\n\n"` zu schreiben. Mit Hilfe des Tabulators lassen sich dann sehr schön Tabellen erstellen:

```
# Code Anfang

for {set i 0} {$i <= 10} {incr i} {
    set erg [expr pow($i,2)]
    puts "$i\t| $erg"
}

# Code Ende
```

Als letzten Befehl in diesem Abschnitt möchte ich den Befehl *time* vorstellen. Mit ihm ist es möglich die Geschwindigkeit eines Systems bei verschiedenen Aufgaben zu testen. (Genauer: Es lässt sich ungefähr feststellen, wie lange der Rechner für eine bestimmte Befehls-Sequenz benötigt.) *time* hat folgende Syntax:

```
time { Befehl(e) } Anzahl
```

Die übergebenen Befehle werden *Anzahl*-mal oft ausgeführt und dabei wird jeweils die Zeit genommen. Zurückgegeben wird ein Durchschnittwert, gebildet aus allen Zeiten. Dies lässt sich dann für einen kleinen Benchmark missbrauchen. Dabei ist zu beachten, daß man ausreichend große Durchlaufzahlen wählt um auch auf schnellen Maschinen noch Werte zu erhalten.

```
# Code Anfang

set varset [time {set a 1000} 100000]
puts $varset
set intdiv [time {expr 4 / 3} 100000]
puts $intdiv
set floatdiv [time {expr 40.0 / 3.7} 100000]
puts $floatdiv

# Code Ende
```

Mi diesen Befehlen möchte ich dann auch die Einführung in die Scriptsprache Tcl beenden. Sie wissen jetzt wie Variablen gesetzt werden, Werte eingelesen und verarbeitet werden können. Sie können weiterhin Dateien öffnen, auslesen,

verändern und Berechnungen durchführen. Mit Hilfe der vorgestellten Kontrollsequenzen und Befehle sind Sie in der Lage eine Vielzahl von Problemen mit Tcl zu bearbeiten. Falls Sie mal einen Befehl brauchen, der hier nicht vorgestellt wurde, empfiehlt es sich einen Blick in die *man*-pages (Unix, Linux, etc.) oder die Online-Hilfe (Windows) zu werfen.

Im nächsten Teil werden wir einen Blick auf das Toolkit Tk werfen, mit dem in Verbindung mit Tcl sehr leicht grafische Oberflächen programmiert werden können.

2 Das Tk-Toolkit

2.1 Einleitung

In diesem Teil der Einführung in die Programmierung mit Tcl/Tk werden wir uns nun mit der grafischen Programmierung beschäftigen. Während im ersten Teil nur Konsolen-Anwendungen, das heißt Programme, die ihre Ausgaben nur im Textmodus machten, entwickelt wurden, werden wir jetzt Programme für die grafischen Benutzeroberflächen der einzelnen Betriebssysteme schreiben.

*wish, cget,
configure,
destroy*

Wie schon mehrfach erwähnt wurde, eignet sich die Verbindung aus der Scriptsprache Tcl und dem Toolkit Tk besonders für diese Aufgabe. Ich möchte hier nur 2 Gründe anführen weshalb das so ist: zum einen sind die Befehle die Tk bereitstellt für sich *plattformunabhängig*, das heißt sie sind auf jedem System verfügbar, auf dem der Tk-Interpreter lauffähig ist. Zum anderen ist die einfache Syntax von Tcl in Tk weitergeführt worden. Tk stellt eine Vielzahl von vorgefertigten Bildschirmobjekten (*Widgets*) zur Verfügung, die kaum Wünsche offen lassen und ohne weiteres mit einander kombinierbar sind.

Für die Darstellung von grafischen Elementen (Programmfenster, Buttons, etc.) wird ein spezieller Interpreter benutzt, die *windowing shell*, kurz *wish*. Alle Programme in diesem Teil müssen mit diesem Interpreter ausgeführt werden. Bei Benutzung von *tclsh* kommt es zu Fehlern, da die *wish* Befehle bereit stellt, die *tclsh* nicht kennt. Der Interpreter wird so aufgerufen:

```
wish Dateiname
```

Wie bei der *tclsh* kann der Name hier abweichen, beispielsweise durch Anhängen einer Versionsnummer (z.B. *wish82*).

Bevor wir mit der Programmierung und der Vorstellung der einzelnen Widgets anfangen, an dieser Stelle ein paar Worte zur Fensterstruktur in Tcl/Tk. Standardmäßig wird nach dem Aufruf der *wish* ein Fenster geschaffen: das Root-Window, das in Tk durch den Punkt (.) repräsentiert wird. Wenn wir nun ein anderes Element in dieses Fenster packen wollen, so lautet dessen Name:

```
.name
```

Wollen wir etwas in dem Element *name* schaffen, so wird der Name des zu erschaffenden Elements durch einen Punkt getrennt an dieses angehängt:

```
.name.neu
```

Für alle folgenden Elemente gilt die gleiche Vorgehensweise.
Im Allgemeinen gilt das ein Widget in der folgenden Weise aufgerufen werden kann:

```
widget Pfadname [Option1 Wert1 [...]]
```

Eine Option beginnt mit einem Minuszeichen (-) an das sich gleich der Name der Option anschließt. Die eckigen Klammern bedeuten hier, daß diese Optionen und Werte *optional* sind, das heißt sie sind nicht zwingend notwendig für diesen Befehl. Gleichzeitig wird ein neuer Befehl geschaffen, der diesen Namen trägt und über diesen können wir dann auf das Element (Widget) zugreifen. Mit dem Befehl *destroy* kann dieser Befehl wieder gelöscht werden:

```
destroy .name.neu
```

Im Programm können die Optionen über folgenden Aufruf geändert werden:

```
Pfadname configure Option1 Wert1 [...]
```

Wenn man den Wert einer Option auslesen will, benutzt man die Funktion *cget*:

```
Pfadname cget Option
```

Im folgenden Abschnitt werden wir die grundlegendsten Widgets betrachten und die Möglichkeiten, die sie uns bieten. In den beiden darauf folgenden Abschnitten werden wir stärker auf die Bearbeitung von Texten und dem Zeichnen/Darstellen von Bildern etc. beschäftigen.

Das klingt alles recht kompliziert, ist es aber im Grunde nicht, wie man sehr bald feststellen wird. Hat man einmal das Prinzip verstanden, steht einem nichts mehr im Weg. Dies wird im nächsten Abschnitt klarer, in dem wir einige der einfacheren Widgets kennenlernen, u. a. den Button.

2.2 Grundlegende Widgets

2.2.1 Klick mich! – Das *button*-Widget

Jetzt werden wir unser erstes Widget kennen lernen: das *button*-Widget. Es erzeugt einen einzelnen Button, der die folgenden Optionen besitzt (die wichtigsten):

button, pack, bell, exit

-bg Farbe	Hintergrundfarbe des Buttons in der Form (#xxxxxx), $x \in (0..9, a..f)$
-command Anweisungen	führt die Anweisungen aus, wenn der Button gedrückt wird
-fg Farbe	Vordergrundfarbe des Buttons in der Form (#xxxxxx), $x \in (0..9, a..f)$
-font	ändert Schriftart der Beschriftung des Buttons
-height	Höhe des Buttons
-relief	bestimmt Effekt des Rahmens des Buttons, möglich sind <i>raised, groove, sunken, flat, ridge</i>
-text	Beschriftung des Buttons
-width	Breite des Buttons

Um jetzt mal an einem Beispiel zu sehen, wie ein Widget erschaffen wird, programmieren wir jetzt einen beschrifteten Button, der wenn wir ihn anklicken das Programm beendet.

```
# Code Anfang
```

```
button .b1 -text "Hallo!!!" -command { exit }  
pack .b1
```

```
# Code Ende
```

Zu diesem Beispiel noch 2 Bemerkungen: mit dem Befehl *exit* wird das Programm sofort beendet und kann optional noch einen Wert an das System zurückgeben. Die andere Bemerkung betrifft die letzte Zeile:

```
pack .b1
```

Dieser Befehl ist dafür zuständig, daß wir auch etwas sehen auf unserem Bildschirm! Löschen Sie ihn einmal und starten Sie das Programm noch einmal. Sie erhalten nur ein leeres Fenster. Was der Befehl alles bewirkt und wie man ihn benutzt, erfahren wir später genauer.

Man kann natürlich nicht nur den Befehl *exit* aufrufen. Nach der *-command* können alle gültigen Tcl/Tk-Befehle aufgerufen werden. Möchte man mehr als einen Befehl ausführen lassen, so setzt man sie wie gehabt in die geschweiften Klammern { und }. Um die Übersicht zu behalten, empfehle ich solche Befehle in einer Prozedur zusammenzufassen und dann die Aktionen über den Prozedurnamen aufzurufen.

```
#Code Anfang
```

```
proc aktion {} {  
    bell  
    after 1000  
    exit  
}
```

```
button .b1 -text "Bing!!!" -command { aktion }  
pack .b1
```

```
# Code Ende
```

Beim Drücken des Buttons in diesem Beispiel ertönt ein Signal (mit Hilfe von *bell*) und nach circa einer Sekunde wird das Programm beendet.

Zum Abschluss zum Thema Button erstellen wir in einer Schleife 5 Buttons mit unterschiedlicher Farbe. Man beachte hier, daß auch die Namen der Widgets über Variablen gebildet werden können (hier erzeugen wir die Buttons *.b1* bis *.b5*).

```

# Code Anfang

set color {"#ffffff" "#abcdef" "#aaaaaa" "#446234" "#000000"}

foreach i {1 2 3 4 5} {

    button .b$i -bg [lindex $color [expr $i - 1]] -text \
        "Hallo!!!" -command { exit }
    pack .b$i
}

# Code Ende

```

Damit möchte ich diesen Abschnitt beenden. Wir wissen jetzt wie man Buttons erzeugt und sie mit Befehlen erknüpft. Als nächstes Widget werden wir uns des *frame*-Widgets annehmen.

2.2.2 Fensterteilung – Das *frame*-Widget

frame

Das *frame*-Widget ist ein sehr einfaches Widget. Es ist im Grunde nichts anderes als ein farbiges Rechteck, dem man noch einen 3D-Effekt zuweisen kann. Es wird häufig benutzt um Elemente in einem Fenster anzuordnen, also für Dekorationszwecke. Es hat folgende Syntax:

```
frame fName [Optionen]
```

Als Optionen stehen hier zur Verfügung:

-relief	sorgt für das Aussehen (siehe <i>button</i>)
-borderwidth	Einheit bestimmt die Breite des Randes
-bg	Farbe des Frames
-width	Breite des Frames
-height	Höhe des Frames

An dieser Stelle etwas zu den Einheiten, die zulässig sind für die Angabe von Breite und Höhe von Widgets. Neben der normalen Pixel-Angabe ist es möglich auch Zentimeter, Zoll, Milimeter und Vielfache von $\frac{1}{72}$ Zoll anzugeben. Dafür muss lediglich an den Zahlenwert einer der folgenden Buchstaben angehängt werden:

c	Zentimeter (cm)
m	Milimeter (mm)
i	Zoll (inch)
p	$\frac{1}{72}$ Zoll

Wollen wir mal zwei kleine Beispiele dafür ausprobieren. Hier wird die Option *bg* verwendet, damit auch etwas gesehen werden kann.

```

# Code Anfang

frame .f1 -bg #ffffff -height 10c -width 7c
pack .f1

```

```
frame .f2 -bg #000000 -height 0.5i -width 2i
pack .f2
```

```
# Code Ende
```

Dies erschafft ein 10 Zentimeter hohes und 7 Zentimeter breiten Frame (weiß) und ein Frame, daß 2 Zoll breit und ein halbes Zoll hoch ist. Warum die Frames so angeordnet wurden und wie man dies verändern kann, erfahren wir später. Damit soll es genug für dieses Widget sein. Jetzt wollen wir uns das *entry*-Widget ansehen.

2.2.3 Her mit den Eingaben! – Das *entry*-Widget

Das *entry*-Widget ist ein einzeiliges Textfeld, das man nutzen kann um Eingaben in Programmen zu machen. Dabei stehen folgende Optionen zur Verfügung:

entry, insert, delete

-width	Länge der Eingabezeile
-textvariable Var	speichert Eingabe in der Variablen Var

Die Eingabe kann die Länge der Eingabezeile überschreiten, dann muss man aber mit Hilfe der Cursortasten (erstmal) durch die Zeile scrollen um den Rest der Eingabe zu sehen. Der in *Var* gespeicherte Wert kann wie jede andere Variable genutzt werden. Eine Besonderheit hat das *entry*-Widget: Jedes Zeichen kann über einen Index angesprochen werden, der die Form *x* hat, wobei *x* die Werte 0 bis zur Länge der Eingabe um 1 verringert annehmen kann. Das heißt bei einer Länge von 24 Zeichen kann *x* die Werte 0, 1, 2, . . . , 23 annehmen. Hier muss man nur darauf achten auch die Leerzeichen und ähnliches mitzuzählen.

Es ist möglich Text in eine Eingabezeile einzufügen bzw. zu löschen. Dies geschieht mit den Befehlen *insert* und *delete*. Der Aufruf von *insert* sieht folgendermaßen aus:

```
.ename insert Index Text
```

Hierbei ist *Index* ein gültiger Index. Die Indexmarke *insert* steht für die aktuelle Cursorposition. Die Marke *end* ist der Index der auf das letzte eingegebene Zeichen folgt. Der folgende Befehl fügt also Text an der Cursorposition ein:

```
.ename insert insert Text
```

Dazu ein kleines Beispiel:

```
# Code Anfang
```

```
entry .e1 -width 15 -textvariable eingabe
pack .e1
```

```
.e1 insert insert "Hallo"
```

```
# Code Ende
```

Hier wird der Text „Hallo“ in das Entry-Widget eingefügt. Der *delete*-Befehl arbeitet in ähnlicher Weise:

```
.ename delete IStart [IEnde]
```

Dies löscht die Zeichen von IStart bis direkt vor IEnde (rechtsseitig offenes Intervall). Fehlt IEnde wird nur das Zeichen an IStart entfernt. Das soll das folgende Beispiel verdeutlichen: Wir schreiben in das *Entry*-Widget wieder den Text „Hallo“ und schneiden danach alle Zeichen vor und nach den „ll“ ab:

```
# Code Anfang
```

```
entry .e1 -width 15 -textvariable eingabe
pack .e1
```

```
.e1 insert insert "Hallo"
.e1 delete 0 2
.e1 delete 2
```

```
# Code Ende
```

Zum Schluß noch eine Anmerkung: Bei der Bestimmung der Indizes ist darauf zu achten, daß diese sich nach jedem Einfüge- bzw. Löschvorgang ändern: Während das „o“ im Ausgangstext den Index 4 hat, hat es nach dem Entfernen der ersten beiden Buchstaben den Index 2. Würde man jetzt die zweite Anweisung mit dem Index 4 ausführen, würde nicht das Gewünschte passieren. Das „o“ wäre immer noch vorhanden.

2.2.4 Das Kind braucht einen Namen! – Das *label*-Widget

label

Das gerade beschriebene *entry*-Widget hat einen Nachteil: Man sieht ihm nicht an, welche Eingaben es erwartet. Wir könnten natürlich die entsprechende Aufforderung in das Widget einfügen:

```
# Code Anfang
```

```
entry .e1 -width 20 -textvariable name
pack .e1
```

```
.e1 insert 0 "Namen eingeben!"
```

```
# Code Ende
```

Dieses Vorgehen hat zwei Nachteile: zum einen sieht es nicht besonders gut aus, zum anderen muss der Benutzer bevor er eine Eingabe machen will erst umständlich unseren Text entfernen. Dies kann man mit der Benutzung des *label*-Widgets umgehen. Es ist dem *frame*-Widget sehr ähnlich. Es besitzt folgende zusätzliche Optionen:

-text	gibt die Beschriftung aus
-textvariable VAR	in VAR enthaltener Wert wird ausgegeben
-font String	ändert die Schriftart in der die Beschriftung erfolgt

Um den Unterschied zwischen *-text* und *-textvariable* zu verdeutlichen, lassen Sie das folgende Programm einmal laufen. Sie werden die Ausgabe „Hi“ erhalten.

Danach entfernen Sie bitte das Kommentarzeichen aus der letzten Zeile und lassen das Programm noch einmal laufen.

```
# Code Anfang
```

```
label .l1 -bg #FFFFFF -text "Hi" -textvariable name  
pack .l1
```

```
# set name "Hallo"
```

```
# Code Ende
```

Jetzt ist die Ausgabe „Hallo“. Die Beschriftung ist also überschrieben worden. Dabei spielt es keine Rolle, ob die Anweisung `set name "Hallo"` am Ende oder am Anfang des Programms ausgeführt wird. Also Obacht! Die Option `font` hat einen recht komplexen Aufbau, sie soll daher an dieser Stelle nicht genau erläutert werden. Um sie dennoch benutzen zu können nur soviel:

```
--Name-Wichtung-Neigung-Abst-Stil-*Groesse-*-*-*-*-*■
```

Hierbei bedeutet *Name* eine Gruppe von gleichartigen Schriften wie zum Beispiel *Courier*, *Helvetica* oder *Times*. *Wichtung* steht für die ypographische Gewichtung, kann *medium* oder *bold* (fett) annehmen. Mit *Neigung* wird das Aussehen maßgeblich beeinflusst. Für aufrechte Schrift setzt man *r*, für kursive Schrift *i* und für „schiefe“ Schrift setzt man *o*. Mit *Abst* setzt man den Abstand der Zeichen zueinander. Zur Verfügung stehen hier *normal*, *condensed* und *narrow*. Mit der Option *Stil* kann man zusätzliche Stile einstellen wie zum Beispiel *serif*. Möchte man nichts eintragen so läßt man es einfach frei bzw. setzt den nächsten Bindestrich. Mit der Option *Größe* setzt man die Größe der Schrift in Punkten. Möchte man eine 12 Punkt-Schrift so trägt man hier 120 ein, also das Zehnfache. Die Angabe in Punkten hat den Vorteil, daß das Aussehen der Schrift dann unabhängig von der Auflösung des Anwenders ist, d. h. die Schrift hat immer die gleiche Größe. Dazu ein Beispiel:

```
# Code Anfang
```

```
set my_font "--Times-bold-i-narrow--*200-*-*-*-*-*"
```

```
label .l1 -text "Hallo" -font $my_font  
pack .l1
```

```
# Code Ende
```

Dieses Beispiel gibt den Text in kursiver, fetter, 20-Punkt großer Times-Schrift aus mit einem wenig Raum zwischen den Buchstaben. Kann Tk einmal eine Schrift nicht finden, so versucht es eine Ausgleichsschriftart zu finden. Laut Literatur sollten die oben angegebenen Schriften aber auf jedem System zu finden sein. Bei einer Schriftart wie Verdana sieht es dagegen anders aus. Im nächsten Abschnitt werden wir uns dann mit einem Widget beschäftigen, daß dem `label`-Widget sehr ähnlich ist, aber trotzdem anders: das `message`-Widget.

2.2.5 Das *message*-Widget

message

Wie bereits angesprochen, ist das *message*-Widget dem *label*-Widget ähnlich. Der Unterschied besteht darin, daß dieses Widget mehrere Zeilen Text aufnehmen kann und über zwei zusätzliche Optionen verfügt:

-width	Länge einer Zeile
-justify	Ausrichtung des Textes in den Zeilen möglich: <i>left</i> , <i>center</i> , <i>right</i>

Wird keine Zeilenlänge angegeben, so bricht Tk den Text selbständig um. Dazu ein Beispiel:

Code Anfang

```
set my_text "Dies ist ein kleiner Text um die \
Faehigkeiten des Widgets zu zeigen."
```

```
message .msg1 -text $my_text -width 180
pack .msg1
```

Code Ende

Dieses Widget ist gut geeignet um zum Beispiel Dialogboxen zu gestalten (ein kleiner Vorgriff):

Code Anfang

```
frame .f1
frame .f2
pack .f1 -side top
pack .f2 -side top
```

```
message .f1.msg1 -text "Eine kleine Dialogbox!"
pack .f1.msg1
button .f2.b1 -text "Klick mich!" -command { exit }
pack .f2.b1
```

Code Ende

Wie man sieht, kann man schon mit wenigen Befehlen eine Dialogbox erstellen. Zugegeben sie ist nicht die Schönste und wenn man das Fenster vergrößert passieren seltsame Sachen, aber das Prinzip dürfte klar sein. Was wir genau tun? Als erstes schaffen wir zwei Frames, dann ein *message*-Widget im oberen Widget *.f1* und ein *button*-Widget im zweiten Frame, der unter dem ersten Frame positioniert wurde. Mehr zum Anordnen von Frames und anderen Widgets im Abschnitt *Layoutmanager*.

2.2.6 Das *listbox*-Widget

listbox

Tk stellt für die Darstellung von Listen das *listbox*-Widget zur Verfügung. Mit ihm können Auswahl-Listen erstellt werden. Dazu das folgende kleine Beispiel:

```
# Code Anfang

listbox .l1
pack .l1

.l1 insert end "Hallo Liste!"

# Code Ende
```

Das Widget hat folgende Syntax:

```
listbox .lname [Optionen]
```

Es stellt diese Optionen zur Verfügung:

-bg	Hintergrundfarbe
-font	Schriftart ändern
-fg	Vordergrundfarbe (= Schriftfarbe)
-height	Höhe
-width	Breite
-relief	Aussehen (s. <i>button</i>)

Wie man schon im Beispiel sehen kann, gibt es auch hier Indexmarken und zwar wie gehabt eine Zahl von Null an bis *end*, was das Ende der Liste markiert. Eine besondere Marke ist der Index *active*, das das ausgewählte Element der Liste bezeichnet. Mit den Befehlen *insert* können Einträge hinzugefügt werden, mit *delete* wieder entfernt. Die Syntax ist die selbe wie beim *entry*-Widget:

```
.lname insert Index Eintrag
.lname delete IStart IEnde
```

Desweiteren stehen folgende Befehle zur Verfügung:

activate Index	setzt das aktive Element auf Index
get I1 [I2]	gibt Liste mit den Elemente I1 bis I2 zurück
see Index	sorgt dafür, daß Element an der Stelle Index sichtbar ist
size	liefert die Anzahl der Elemente in der Listbox

Versuchen Sie doch mal, das Programm, daß im Abschnitt „Listen“ entwickelt wurde, dahingehend zu modifizieren, daß es die Datei „cde.xx“ einliest und die Einträge als Listbox ausgibt. (Tip: Lesen Sie die Einträge in eine Liste und arbeiten Sie diese dann mit einer *foreach*-Schleife ab.)

Damit möchte ich die Vorstellung der grundlegenden Widgets, die Tk zur Verfügung stellt abschließen. Probieren Sie die Widgets zu manipulieren, damit Sie ein Gefühl für die Syntax und Möglichkeiten bekommen. Im nächsten größeren Abschnitt werden wir uns endlich darum kümmern, wie wir unsere Widgets richtig positionieren können und ihnen beibringen, wie sie sich bei Veränderungen der Fenstergröße etc. verhalten sollen.

2.3 Die Layoutmanager

*pack, grid,
place*

In diesem Abschnitt soll es nun darum gehen, daß unsere Programme auch aus mehreren Widgets bestehen können und das es unsere Aufgabe ist, dafür zu sorgen, daß es auch gut aussieht bzw. benutzbar ist. Dabei unterstützen uns die Layoutmanager. Sie sorgen für das Layout (Aussehen) unseres Programms. Also wenn wir wollen, daß in unserem Programm ein Button in der linken unteren Ecke positioniert wird, so erschaffen wir nur den Button und sagen dann „Layoutmanager, positioniere den Button links unten“. Das war es dann auch schon. Der Layoutmanager kümmert sich um alles Weitere. Tk stellt mehrere Layoutmanager zur Verfügung, die über die Befehle *pack*, *grid* und *place* aufgerufen werden können. Jeder der drei Manager verfolgt dabei einen anderen Ansatz zur Darstellung. Natürlich können auch mehrere Manager in einem Programm benutzt werden, die stellt kein Problem dar. Als erstes möchte ich den Layoutmanager *pack* vorstellen, der uns in den vorhergegangenen Beispielen schon begegnet ist.

2.3.1 Der Layoutmanager *pack*

Der Layoutmanager *pack* ermöglicht uns das zeilen- bzw. spaltenweise Anordnen von Widgets. Dabei geht *pack* immer vom äußeren Rand des Widgets/Fensters in dem das neue Widget platziert werden soll in die Mitte des Widgets. Das heißt bei der Anordnung wird immer eine gesamte Zeile bzw. Spalte des Widgets/Fensters für das Widget, daß wir platzieren wollen, reserviert wird. Damit wird die Platzierung mehrerer Widgets nebeneinander bzw. untereinander schwierig. Als Ausweg bietet sich hier die Benutzung von *frame*-Widgets.

pack hat folgende Syntax:

```
pack .wname1 [.wname] [Optionen]
```

Wie man sieht, muß dem *pack*-Befehl der Name mindestens eines Widgets übergeben werde, es können also auch mehrere Widgets auf einmal platziert werden. Für diesen Befehl gibt es mehrere Optionen, die in folgender Tabelle aufgelistet sind:

-anchor	Ausrichtung der Widgets
-expand	Ausdehnung des Widgets
-fill	füllt vorhandenen Platz in vorgegebener Richtung
-ipadx	internes Padding in x-Richtung
-ipady	internes Padding in y-Richtung
-padx	externes Padding in x-Richtung
-pady	externes Padding in y-Richtung
-side	Seitenposition des Widgets

Die Option *anchor* kann mehrere Werte annehmen, die sich an den Himmelsrichtungen orientieren. Für eine linksbündige Ausrichtung des Widgets nimmt beispielsweise „w“ für Westen, „e“ wäre dann rechtsbündig und so weiter.

Code Anfang

```
button .b1 -text "linksbuendig" -command { exit }  
pack .b1 -anchor w
```

```
button .b2 -text "rechtsbuendig" -command { exit }
pack .b2 -anchor e
```

Code Ende

Sollte der Effekt nicht gleich offensichtlich sein, einfach das Fenster vergrößern. Hier sieht man auch ein Problem, des *pack*-Befehls. Er hat den zweiten Button gleich in eine neue Zeile gepackt und nicht neben den ersten.

Mit *expand* wird festgelegt, ob sich das Widget auch in y-Richtung vergrößern darf, wenn sich der zur Verfügung stehende Platz ändert. Gültige Werte hierfür sind *yes* und *no*.

Code Anfang

```
button .b1 -text "linksbuendig" -command { exit }
pack .b1 -anchor w -expand no
button .b2 -text "rechtsbuendig" -command { exit }
pack .b2 -anchor e -expand yes
```

Code Ende

Läßt man dieses Programm laufen und vergrößert das Fenster, so sieht man nur, daß sich die beiden Widgets den Platz im Fenster teilen, mehr nicht. Die *expand*-Option gewinnt an Bedeutung in Verbindung mit der *fill*-Option. Die gibt an in welche Richtung sich das Widget ausdehnen soll, wenn mehr Platz zur Verfügung steht. Als mögliche Werte gibt es hier *x*, *y* und *both* für beide Richtungen.

Code Anfang

```
button .b1 -text "linksbuendig" -command { exit }
pack .b1 -anchor w -expand no -fill both
button .b2 -text "rechtsbuendig" -command { exit }
pack .b2 -anchor e -expand yes -fill both
```

Code Ende

Hier sieht man, obwohl sich das Widget *.b1* in beide Richtungen ausdehnen soll, dehnt es sich nur in x-Richtung, da mit *-expand no* die Ausdehnung in y-Richtung unterbunden wurde. Der zweite Button hat diese Einschränkung nicht und dehnt sich in beide Richtungen aus.

Mit den beiden Optionen *-ipadx* und *ipady* stellt man das interne Padding des Widgets ein. Was internes Padding bedeutet? Durch diese Anweisung weisen wir dem Widget mehr Platz zu, das dieses dadurch ausfüllt, das es sich selbst vergrößert. Als Verdeutlichung dieses Beispiel:

Code Anfang

```
button .b1 -text "ohne -ipad"
```

```
pack .b1
button .b2 -text "mit -ipad"
pack .b2 -ipadx 10 -ipady 10
```

Code Ende

Wie man sieht umschließt der Button .b1 den Text „ohne -ipad“ recht eng, beim zweiten Button ist zwischen dem Text und dem Rand des Buttons .b2 ein Abstand (10 Pixel in x-, und 10 Pixel in y-Richtung). Im Gegensatz dazu sorgen *-padx* und *-pady* für Abstand zwischen den Widgets:

Code Anfang

```
button .b1 -text "ohne pad"
pack .b1
button .b2 -text "mit pad"
pack .b2 -pady 10
button .b3 -text "auch ohne"
pack .b3
```

Code Ende

Wie man sieht ist der Button .b2, der mit der Option *-pady* positioniert wurde, von den anderen Buttons getrennt. Zwischen ihnen liegt in jede y-Richtung ein Abstand von 10 Pixel. Mit *-padx* und *-pady* sorgt man also für Abstand zwischen den Widgets, während *-ipadx* und *-ipady* das Widget vergrößern.

Als letztes möchte ich die Option *-side* vorstellen. Mit ihr kann man festlegen, wo der Layoutmanager das Widget platzieren soll. Da *pack* nach einem Zeilen/Spaltenprinzip vorgeht, gibt sind hier folgende Werte möglich: *top*, *bottom*, *left* und *right*. Beispielsweise wird ein Widget mit der Option *bottom* immer über das unterste bereits vorhandene Widget gepackt:

Code Anfang

```
button .b1 -text "Button 1"
pack .b1 -side bottom
button .b2 -text "Button 2"
pack .b2 -side bottom
```

Code Ende

Wie man sieht befindet sich der Button .b2 über dem Button .b1, da nach dem Platzieren des ersten Buttons, die unterste Zeile für den Button .b1 reserviert wird. Der Button .b2 wird dann an das Ende des verfügbaren Platzes gepackt und dieser beginnt unmittelbar über dem Button .b1.

Damit möchte ich die Vorstellung des Befehles *pack* abschließen. wie man sieht ist er vielseitig einsetzbar um Widgets auf dem Bildschirm zu positionieren. Im nächsten Abschnitt möchte ich den Befehl *grid* vorstellen, der ein etwas anderes Prinzip zur Darstellung verfolgt.

2.3.2 Der Layoutmanager *grid*

Mit dem Layoutmanager *grid* ist es möglich die Widgets an einem virtuellen Raster auszurichten. Dieses Raster kann aus beliebig vielen Zeilen und Spalten bestehen. Dazu verfolgt *grid* folgendes Prinzip:

	Spalte 1	Spalte 2	...
Zeile 1:	<code>grid Widget1</code>	<code>[Widget2 ...]</code>	<code>[Optionen]</code>
Zeile 2:	<code>grid Widgetn</code>	<code>[Widgeto ...]</code>	<code>[Optionen]</code>

Jedes *grid* übergebene Widget bildet eine eigene Spalte im Raster. Jeder neue *grid*-Befehl erzeugt eine neue Zeile im Raster. Ein Widget erhält in diesem Raster einen Index, der aus Zeilen und Spaltennummer besteht. Außerdem werden noch folgende Optionen bereitgestellt:

<code>-ipadx</code>	siehe <i>pack</i>
<code>-ipady</code>	siehe <i>pack</i>
<code>-padx</code>	siehe <i>pack</i>
<code>-pady</code>	siehe <i>pack</i>
<code>-sticky</code>	anheften an ein anderes Widget
<code>-row</code>	Zeilenzahl zuweisen
<code>column</code>	Spaltenzahl zuweisen

Mit der Option *-sticky* kann man das Widget an ein anderes „anheften“. Es bleibt dann immer an dessen Seite. Als Auswahlmöglichkeit stehen *e*, *n*, *s* und *w* zur Verfügung, die sich wieder an den Himmelsrichtungen orientieren.

Zur Benutzung nun ein kleines Beispiel, doch zuvor noch ein Hinweis: Möchte man eine leere Spalte erzeugen, so schreibt man „x“. Im Beispiel erzeugen wir erst 8 Buttons und ordnen sie dann in 3 Spalten und 3 Zeilen an. Dabei soll das Spalte 2 der zweiten Zeile leer bleiben.

Code Anfang

```
foreach i {11 12 13 21 23 31 32 33} {  
  button .b$i -text "$i"  
}
```

```
grid .b11 .b12 .b13 -padx 5 -pady 5  
grid .b21 x .b23 -padx 5 -pady 5  
grid .b31 .b32 .b33 -padx 5 -pady 5
```

Code Ende

Wie man sieht, ist diese Art der Anordnung mit dem Befehl *grid* sehr einfach. Das soll dazu auch reichen. Die Mischung von *pack* und *grid* lässt fast keine Wünsche mehr offen. Für alle anderen gibt es noch einen weiteren Layoutmanager: *place*.

2.3.3 Widgets platzieren mit *place*

Mit diesem Befehl ist es möglich Widgets *pixelgenau* innerhalb eines anderen Widgets oder Fensters zu platzieren. *place* hat folgende Syntax:

```
place .wname -x xpos -y ypos -anchor {n, s, e, w}
```

xpos und *ypos* bezeichnen Pixelkoordinaten *innerhalb* des Widgets/Fensters. Mit *-anchor* kann man einstellen, welche Seite/Ecke des Widgets an der Position (*xpos*, *ypos*) stehen soll. Beispielsweise steht „se“ für die untere rechte Ecke. Mit folgenden Optionen kann man die Widgets relativ zum Fenster platzieren:

-relx	relative x-Koordinate
-rely	relative y-Koordinate
-relheight	relative Höhe
-relwidth	relative Breite

Sinnvolle Werte legen hier zwischen 0 und 1. Dazu ein kleines Beispiel, das den Button immer in der Mitte des Fensters platziert. Ich lege es so fest, daß die untere linke Ecke an die Position gesetzt wird.

```
# Code Anfang

button .b -text "Button"
place .b -relx 0.5 -rely 0.5 -anchor sw

# Code Ende
```

Aber Achtung! Auf den ersten Blick erscheint die Platzierung mit *place* einfacher, aber auf die Dauer ist das Bestimmen der einzelnen Pixelpositionen mühsam. Auch kann es dazu kommen, daß sich zwei Widget überlappen, wenn zum Beispiel bei den Koordinaten ein Fehler gemacht wurde:

```
# Code Anfang

button .b1 -text "Button1"
place .b1 -x 50 -y 50

button .b2 -text "Button1"
place .b2 -x 60 -y 60
# Code Ende
```

Hier ist also erhöhte Aufmerksamkeit notwendig. Persönlich bevorzuge ich *grid* und *pack*. Damit möchte ich diesen Abschnitt beenden und im nächsten das Prinzip des *Key-Bindings* erklären.

2.4 Das Key-Binding

bind

Jeder von uns weiß, daß man in fast jeder grafischen Anwendung bestimmte Funktionen über Tastenkürzel aufrufen kann. Bekannt sind zum Beispiel STRG-C zum Kopieren, STRG-V zum Einfügen und ALT-X zum Beenden einer Anwendung. Soetwas können wir natürlich auch. Dafür stellt uns Tk den *bind*-Befehl

zur Verfügung, der ein Widget mit einer beliebigen Taste(nkombination) verbindet. Wird diese Tastenkombination dann gedrückt, wird ein vorher festgelegter Befehl ausgeführt. Die Syntax von *bind* ist folgende:

```
bind .Widget <Tastenkombination> Befehl
```

Widget bezeichnet hier das Widget für welches die in den Klammern angegebene Tastenkombination gelten soll. Möchte man eine Kombination für das Fenster „,“ vereinbaren, so schreibt man nur den Punkt:

```
# Code Anfang
```

```
button .b1 -text "Button1"  
pack .b1
```

```
bind . <Control-c> { exit }
```

```
# Code Ende
```

Dieses Programm kann man beenden, indem man die Tastenkombination STRG-C drückt. Bei der Vereinbarung mit *bind* ist auf die Schreibweise der Tasten zu achten: <Control-c> hat eine andere Bedeutung als <Control-C>.

Literatur

- [1] *Tcl und Tk*, John K. Ousterhout, Addison-Wesley, 1995
- [2] *Effektiv Tcl/Tk programmieren*, Mark Harrison u. Michael McLennan, Addison-Wesley, 1998
- [3] *Tcl/Tk In A Nutshell*, Paul Raines u. Jeff Tranter, O'Reilly, 1999
- [4] *Tcl/Tk kurz & gut*, Paul Raines, O'Reilly, 1998