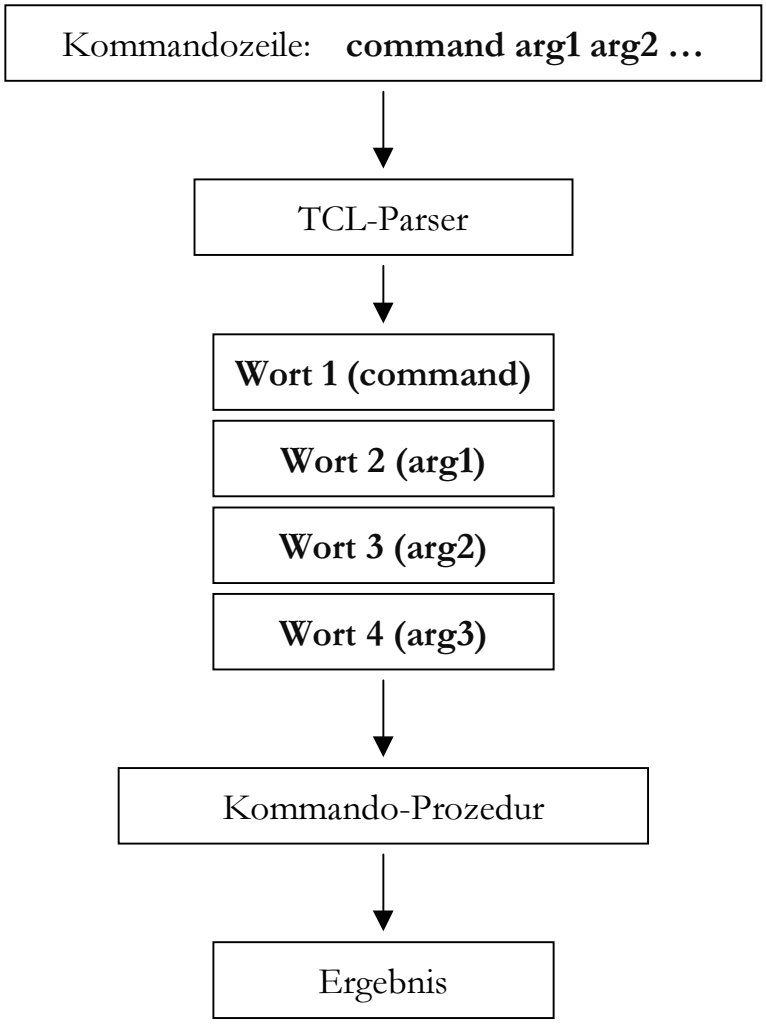
TCL- Eine kurze Einführung

1. Arbeitsweise des tcl-Interpreters

Kommando-Syntax:	**command arg1 arg2 … argN …**
command:	built-in-command
	TCL-Prozedur
Trennzeichen:	Leerzeichen und Tabulatoren
Kommandoabschluß:	Newline oder Semikolon
Kommentarzeichen:	#-Zeichen, muß am Beginn eines Kommandos stehen



Vorgehensweise:

- Ein Durchgang von links nach rechts
- Ergebnis nicht erneut nach Sonderzeichen geprüft
- Zerlegung in einzelne Worte (1.Wort = Kommando, weitere Worte Kommandoargumente)
- Kommando interpretiert die Argumente gegebenenfalls erneut
- Substitution und Gruppierung der Worte
- Rückgabewerte von tcl-Funktionen sind Strings, die im Fehlerfall eine Beschreibung des aufgetretenen Fehlers enthalten.

2. Einfache Variablen

Namen: beliebige Zeichenfolge, Groß- und Kleinschreibung signifikant

Wert: Strings

Datenstrukturen: Listen, assoziative Arrays

Keine Deklaration vor dem Gebrauch!

Grundlegende Operatoren: **set, unset, append, incr, puts**

Grundlegende Operatoren	
Kommando-Syntax	**Beschreibung**
set varName ?value?	Wertzuweisung
unset varName ? varName…varName?	Variable auf „undefiniert“ setzen
append varName value ? value?	Wert an bisherigen Wert anhängen
incr varName ?increment?	Wert wird als Zahl interpretiert und um *increment* verändert
puts ?-nonewline? ?channel? string	Schreibt eine Zeichenkette mit und ohne abschließendem newline, ohne Angabe von *channel* wird die Ausgabe an die Standardausgabe geschrieben

Beispiel 1: einfache Variablen

```
tcl>set a 3
tcl>puts $a
3
tcl>append a 4
34
tcl>incr a -4
30
tcl>unset a
tcl>
```

3. Substitution

Arten der Substitution

1. Variablen und Ersetzungen ($)
2. Kommando-Substitution ([…])
3. backslash-Substitution (\)

3.1. Variablen und Ersetzungen ($)

Das $-Zeichen vor einer Variablen führt zu einer Wertersetzung dieser Variablen.

Auf den Wert einer Variablen kann also mit dem vorangestellten $-Zeichen zugegriffen werden.

Variablensubstitution wird auch innerhalb von:

- Worten
- Doppelapostroph-Gruppierungen
- Kommando-Substitutionen

durchgeführt.

Beispiel 2: Variablensubstitution

```
tcl>set var1 abc
tcl>set var2 $var1
tcl>puts $var2
```

```
abc
tcl>set var3 ${var1}def
tcl>puts $var3
abcdef
tcl>set var4 ABC$var3
tcl>puts $var4
ABCabcdef
tcl>set var5 ABC${var3}DEF
tcl>puts $var5
ABCabcdefDEF
tcl>
```

3.2. Kommando-Substitution ([…])

Anwendung: Verschachtelung von Kommandos

- verschachtelte Kommandos werden durch […] begrenzt
- vom tcl-Interpreter wird alles innerhalb von […] als ein Befehl interpretiert
- das umgebende Kommando wird durch die Ersetzung der […] mit dessen Ergebnis neu interpretiert
- besteht das Ergebnis aus mehreren Worten, so wird es als ein Wort interpretiert, also der Leerraum ignoriert
- beliebige Verschachtelungstiefe
- Auflösung der […] erfolgt von links nach rechts

Beispiel 3: Kommando-Substitution

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set a 4
```

```
puts { -> set a 4 }
set b [expr $a * 2 ]
puts { -> set b [expr $a * 2 ] }
puts "b= $b"

set b [set x 8; expr $a * $x]
puts { -> set b [set x 8; expr $a * $x] }
puts "b = $b"

set len [expr [string length Hallo] + 20]
puts { -> set len [expr [string length Hallo] + 20] }
puts "len = $len"
exit 0
```

Ausgabe:

```
 -> set a 4
 -> set b [expr $a * 2 ]
b= 8
 -> set b [set x 8; expr $a * $x]
b = 32
 -> set len [expr [string length Hallo] + 20]
len = 25
```

3.3. backslash-Substitution (\)

Aufgabe:

- Schutz gewisser Zeichen vor Interpretation, beispielsweise \[

Beispiel 4:

```
tcl>puts "\[a,b,c\]"
[a,b,c]
```

- verleiht gewissen Zeichen Sonderbedeutung

Backslash-Ersetzung	
Backslash-Sequenz	**Ersetzung**
\a	Bell (0x7)
\b	Backspace (0x8)
\f	Form Feed (0xc)
\n	Newline (0xa)
\r	Carriage Return (0xd)
\t	Horizontaltabulator (0x9)
\v	Vertikaltabulator (0xb)
\ddd	Bis zu dreistelliger Oktalwert `tcl>set escape \033`
\xhh	Hexadezimalwert (beliebiger Länge) `tcl>set escape \0x1b`
\newline	Einzelnes Leerzeichen
\newline space	Einzelnes Leerzeichen

\c	Ersetzt durch c, falls c keines der Zeichen von oben
	`tcl>puts "\[ abc \]"`
	`[ abc ]`

- Aufhebung des Zeilenumbruchs als Kommandoabschluß

Beispiel 5:

```
tcl>set gesamteLaenge [expr [string \
length "abcdefghijk"] + \
=>[string length "lmnopqrstuvwxyz"]]
```

```
tcl>puts $gesamteLaenge
26
```

- nur ein Durchlauf der Interpretation

Beispiel 6:

```
tcl> set price \$100
$100
tcl> set costs ${price}Dollar
$100Dollar
```

4. Gruppierung

Arten der Gruppierung

1. Mit Doppelapostrophen " … "
2. Mit geschweiften Klammern { … }

4.1. Gruppierung mit Doppelapostrophen " … "

Bedeutung:

- Wort- und Kommadotrenner verlieren an Sonderbedeutung, d.h. white space, newline und ; werden zu „normalen“ Zeichen
- Substitutionen (Variablen-, Kommando- und Backslash-Substitutionen) finden statt
- Anwendung beispielsweise in den Kommados format und puts

Beispiel 7: Gruppierung mit Doppelapostrophen " … "

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
set titel "Programmieren in Tcl/Tk"
set preis 35.90
puts [format "%-30s\t%-10s" Titel "Preis (\$)"]
```

```
puts "-----------------------------------------"
puts [format "%-30s\t%5.2f" $titel $preis]
puts [format "%-30s\t%5.2f" "Das backslash(\\)-Buch"
0.99 ]
exit 0
```

Ausgabe:

```
Titel                          Preis ($)
-----------------------------------------
Programmieren in Tcl/Tk        35.90
Das backslash(\)-Buch           0.99
```

Variante des obigen Beispiels:

```
puts [format "%-30s\t%5.2f" {Das backslash(\)-Buch}
```

4.2. Gruppierung mit geschweiften Klammern { … }

- Sämtliche Worte verlieren an Sonderbedeutung

Beispiel 8:

```
tcl>puts [format "%-30s\t%5.2f" {Das backslash(\)-Buch}
0.99 ]
Das backslash(\)-Buch           0.99
```

- Substitutionen (Variablen-, Kommando- und Backslash-Substitutionen) finden nicht statt

Beispiel 9:

```
tcl>puts [format "%-30s\t%5.2f" {Titel: $titel} $preis]
Titel: $titel                  35.90
```

Anwendung:

Übergabe ganzer tcl-Skripte als Argument an das Kommando – dieses wertet Sonderzeichen aus, oftmals durch Rückführung des Arguments an den Tcl-Interpreter

Beispiel 10: Übergabe ganzer tcl-Skripte als Argument an das Kommando

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set i 1; set prod 1
while { $i <= 5 } {
        set prod [ expr $prod * $i ]
        incr i
}
puts "5! = $prod"
exit 0
```

Ausgabe:

```
5! = 120
```

4.3. Unterschied der Gruppierungsarten

Der Unterschied der beiden Grupperiungsarten besteht in der Durchführung der Substitution oder dem Ignorieren.

Beispiel 11: Unterschied der Gruppierungsarten

```
tcl>set s Hallo
tcl>puts  "Die Laenge von $s ist [string length $s]."
Die Laenge von Hallo ist 5.
tcl>puts  {Die Laenge von $s ist [string length $s].}
Die Laenge von $s ist [string length $s].
```

4.4. Gruppierung vor Substitutuion

Ein verschachteltes Kommando wird als Folge von Zeichen behandelt. Beim Bestimmen der Argumente für das Hauptkommando wird das verschachtelte Kommando in die umgebende Gruppe eingebaut.

Das folgende Beispiel zeigt wie die Kommandosubstitution die Gruppierung berührt.

Im ersten Beispiel ist das Argument von puts $x+$y=[expr $x + $y], die Leerzeichen innerhalb des verschachtelten Kommandos werden ignoriert.

Beispiel 12: Gruppierung vor Substitutuion

```
tcl>puts $x+$y=[expr $x + $y]
7+9=16
```

Arbeitsschritte des tcl-Interpreters:

1. Die Anweisung wird einmal zu Zwecken der Variablen - und Kommandosubstitution durchgegangen, dabei wird $x und $y durch 7+9 ersetzt.
2. Die anschließend erkannten […] führen zu einem rekursiven Aufruf des Interpreters zur Auswertung der verschachtelten Anweisung.
3. Dabei werden wiederum $x und $y substituiert bevor expr aufgerufen wird.
4. Schließlich wird [expr $x + $y] durch das Ergebnis ersetzt.
5. puts bekommt also als Argument 7+9=16.

Das Beispiel zeigt also, dass die Gruppierung vor der Substitution ausgeführt wird. Durchgeführte Substitutionen beeinflussen die Gruppierung nicht.

[…] gruppieren nicht, sondern das verschachtelte Kommando ist ein Element der umgebenden Gruppe. Folglich ist es nicht notwendig verschachtelte Kommandos zu gruppieren.

Das folgende Beispiel zeigt die Konkatenation der Ergebnisse zweier Anweisungen. Die Ergebnisse werden konkateniert, da sich kein Leerzeichen zwischen den Befehlen befindet.

Beispiel 13: Konkatenation der Ergebnisse zweier Anweisungen

```
tcl>set x 7; set y 9
tcl>set z [expr $x+$y][expr $x-$y]
tcl>puts $z
16-2
```

4.5. Zusammenfassung zu Gruppierung und Substitution

- Das Dollarzeichen erzwingt die Variablensubstitution. Referenzen auf Variable können in beliebige Strings eingebettet sein, Erkennung durch ${varname}
- Ein Dollarzeichen gefolgt von einem Leerzeichen wird als literales $-Zeichen interpretiert.

Beispiel 14 :

```
tcl>set waehrung $
tcl>puts $waehrung
$
```

- Eckige Klammern erzwingen die Kommado-Substitution. Alles zwischen den [ ] wird durch das Ergebnis des Kommandos ersetzt. Klammern können geschachtelt werden, die Bearbeitung erfolgt von links nach rechts.
- Mit dem Backslash wird Einzelzeichen-Substitution erzwungen.
- Substitutionen können an beliebigen Stellen vorgenommen werden, außer sie sind durch { } verhindert.
- Gruppierungen durch { } verhindern jegliche Interpretation, auch die Zerlegung in Worte (Interpretation von Leerraum). { } können geschachtelt werden.
- Gruppierung mit Doppelapostrophen erlaubt Substitution.
- **Grupperiung vor Substitution**. Durchgeführte Substitutionen beeinflussen die Gruppierung nicht.
- Nur ein Durchlauf der Substitution wird durchgeführt, bevor ein Kommando aufgerufen wird. Das heißt das Ergebnis der Substitution wird nicht erneut bewertet. Das aufgerufene Kommando kann allerdings eine erneute Substitution durchführen (rekursiver tcl-Aufruf).
- Nach einer schließenden geschweiften Klammer und einer öffnenden geschweiften Klammer sollte i.a. eine Worttrenner (Leerraum) stehen. Da die Klammern nur der Grupperiung dienen und damit ein Worttrenner (Leerzeichen) zur Trennung der beiden Anweisungen fehlt.

Beispiel 15:

```
tcl>if {$x > 1} {puts "x = $x"}
x = 7
```

```
tcl>if {$x > 1}{puts "x = $x"}
Error: extra characters after close-brace
```

- Leerzeichen zwischen eckigen Klammern werden nicht benötigt, da diese für die Kommadosubstitution gebraucht werden. Der Interpreter betrachtet alles zwischen den Klammern als zu der aktuellen Gruppe zugehörend.

Beispiel 16:

```
set x [cmd1][cmd2]
```

- Werden Doppelapostroph zur Gruppierung verwendet, so wird damit der Effekt der geschweiften Klammern aufgehoben. Substitutionen werden innerhalb von Doppelapostrophen überall durchgeführt.

Beispiel 17:

```
tcl>set x x_value
tcl>set y "y_value + {$x} !"
tcl>puts $y
y_value + {x_value} !
```

- Ein Doppelapostroph ist nur signifikant nach Leerraum.

Beispiel 18:

```
set x ab"cd"
tcl>puts $x
ab"cd"
```

- Newlines und Semikolons werden bei der Gruppierung nicht als Worttrenner bzw. Kommadoabschluß interpretiert.

Beispiel 19:

```
tcl>set zeilen "Das ist die erste Zeile.
Und das die zweite Zeile.
Und nun die dritte Zeile usw."
```

```
=>=>tcl>puts $zeilen
Das ist die erste Zeile.
Und das die zweite Zeile.
Und nun die dritte Zeile usw.
```

5. Numerische Ausdrücke

Der tcl-Interpreter berechnet nicht selbst die mathematischen Ausdrücke, sondern das Kommando expr.

Kommando-Syntax: **expr argument { argument }**

Die Argumente von expr werden konkateniert und als ein Argument an expr weitergegeben.

5.1. Operanden:

- Dezimalzahlen (erste Ziffer keine 0)

Beispiel 20:

```
tcl>expr 9/3
3
```

- Oktalzahlen (erste Ziffer 0)

Beispiel 21:

```
tcl>expr 011/03
3
```

- Hexadezimalzahlen (führendes 0x)

Beispiel 22:

```
tcl>expr 0x9/0x3
3
```

- Reals (Genauigkeit mit built-in Variable tcl_precision bestimmt)

Beispiel 23:

```
tcl>expr 1/3

0

tcl>expr 1/3.0

0.333333333333

tcl> set tcl_precision 15

15

tcl>expr 1/3.0

0.333333333333333
```

5.2. Operatoren:

- Wahrheitswerte (Boolean) 0 ≙ False, 1 ≙ True

Operanden (Reihenfolge mit abnehmendem Vorrang)		
Syntax	**Ergebnis**	**Zulässige Operanden**
-a	Negativer Wert von a	int, real
!a	logisches NOT: 1 wenn a Null, 0 sonst	int, real
~a	Bitweises Komplement von a	int
a$b	Produkt aus a und b	int, real
a/b	Division a durch b	int, real
a%b	Rest nach Division a durch b	int
a+b	Summe aus a und b	int, real
a-b	Differenz aus a und b	int, real
a<<b	Links-Shift von a um b Bits	int
a>>b	Arithmetischer Rechtsshift a um b Bits	int
a<b	1 falls a kleiner b, sonst 0	int, real, string
a>b	1 falls a größer b, sonst 0	int, real, string
a<=b	1 falls a kleiner oder gleich b, sonst 0	int, real, string
a>=b	1 falls a größer oder gleich b, sonst 0	int, real, string

a==b	1 falls a gleich b, sonst 0	int, real, string
a!=b	1 falls a ungleich b, sonst 0	int, real, string
a~b	Bitweises UND von a und b	int
a^b	Bitweises exclusives ODER von a und b	int
a|b	Bitweises inclusives ODER von a und b	int
a&&b	Logisches UND von a und b: 1 falls a und b beide ungleich 0, 0 sonst	int, real
a||b	Logisches ODER von a und b: 1 falls a oder b ungleich 0, 0 sonst	int, real
a?b:c	Auswahl: wenn a (wahr), dann b, sonst c	a: int, real

5.3. Mathematische Funktionen

Mathematische Funktionen	
Funktion	**Ergebnis**
abs(x)	Absolutbetrag von x
acos(x)	Arcuscosinus von $x \in (0,\pi)$
asin(x)	Arcussinus von $x \in (-\pi/2,+\pi/2)$
atan(x)	Arcustangens von $x \in (-\pi/2,+\pi/2)$
atan2(x,y)	Arcustangens von $x/y \in (-\pi/2,+\pi/2)$
ceil(x)	Kleinste Integer >= x
cos(x)	Cosinus von x (x im Bogenmaß)
cosh(x)	Cosinus Hyperbolicus von x
double(i)	Integer i als Real
exp(x)	e^x
floor(x)	Größte Integer <= x
fmod(x,y)	Reeller Rest bei ganzz. Division von x durch y
hypot(x,y)	Quadratwurzel von x^2+y^2
ins(x)	Ganzzahliger Anteil von
log(x)	natürlicher Logarithmus von x

loglO(x)	Logarithmus zur Basis 10 von x
pow(x,y)	x^Y
round(x)	Ganzzahliger Wert durch Rundung
sin(x)	Sinus von x (x im Bogenmaß)
sinh(x)	Sinus Hyperbolicus von x
sqrt(x)	Quadratwurzel von x
tan(x)	Tangens von x (x im Bogenmaß)
tanh(x)	Tangens Hyperbolicus von x

Beispiel 24:

```
tcl>set x 2
tcl>set y 3
tcl>expr 2*pow($x,$y)
16.0
```

6. ARGV, File I/O, Enviroment (env)

6.1. Kommandozeilenargumente

- **argc**: Anzahl der Argumente inklusiv des Programmnamens
- **argv0:** Tcl-Programmname/Kommandoname
- **argv:** Liste der Kommandozeilen-Argumente

Beispiel 25: Kommandozeilenargumente

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
puts "command name: $argv0"
puts "There are $argc arguments: $argv"
puts "The 1st argument is: [lindex $argv 0] "
```

```
puts "The 2nd argument is: [lindex $argv 1] "
puts "The 3rd argument is: [lindex $argv 2] "
```

Ausgabe:

```
>args.tcl 1 2 3 4 5
command name: args.tcl
There are 5 arguments: 1 2 3 4 5
The 1st argument is: 1
The 2nd argument is: 2
The 3rd argument is: 3
```

6.2. Umgebungsvariable env

- **env**: Array, dessen Elemente die Werte der Umgebungsvariablen des Prozesses sind und das mit Strings (Namen der Umgebungsvariablen) indiziert wird.

Beispiel 26: gezieltes Abfragen einer Umgebungsvariable

```
tcl> puts "The home directory is: $env(HOME)"
The home directory is: d:\work
```

Beispiel 27: Abfragen aller Umgebungsvariablen

```
tcl> set args [lsort [array names env]]
tcl> foreach x $args { \
tcl> puts [format "%s = %s" $x $env($x)] }
```

Ausgabe: Gibt eine Liste aller gesetzten Umgebungsvariablen und deren Werte aus.

6.3. File I/O

Grundlegende Operatoren: **open, gets, read, puts, close**

„Standard-Filedeskriptoren“: **stdin, stdout, stderr**

Grundlegende File I/O -Operatoren	
Kommando-Syntax	**Beschreibung**
open filename ?access? ?permissions?	Liefert einen Filedeskriptor zurück *access*: Zugriffsart (siehe Tabelle Skript 183) *permissions*: Dateiattribute (siehe Tabelle Skript 184)
gets channel ?varname?	Liest zeilenweise aus *channel* in die angegebene Variable *varname* ; zurückgeliefert wird –1 bei Dateiende und sonst die Anzahl der gelesenen Bytes
read ?-nonewline? Channel ?numBytes?	Liest (blockweise) *numBytes* oder alles mit oder ohne abschließendem newline;
puts ?-nonewline? ?channel? string	Schreibt eine Zeichenkette mit und ohne abschließendem newline, ohne Angabe von *channel* wird die Ausgabe an die Standardausgabe geschrieben;
close channel	Schließt Filedeskriptor

Die Standard-Filedeskriptoren sind bereits geöffnet und können mit close geschlossen werden.

Beispiel 28: File I/O: open & gets

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set PHONEBOOK "phones"
puts stderr "Phonebook is: $PHONEBOOK"
```

```
set f [open $PHONEBOOK r]

gets $f line
puts stdout $line

close $f
exit 0
```

Ausgabe:

```
Phonebook is: phones
Ralf : 5857
```

Beispiel 29: Lesen von Tastatureingaben

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

proc getnext {f} {
    gets $f line
    return $line
}

set PHONEBOOK "phones"
set f [open $PHONEBOOK r]
```

```
puts "Next? (y/n) "
gets stdin answer
while { ! [string compare "$answer" "y"] } {
   puts [getnext $f]
   puts "Next? (y/n) "
   gets stdin answer
}
close $f
exit 0
```

Ausgabe:

```
>getnext
Next? (y/n)
y
Ralf : 5857
Next? (y/n)
y
Balser : 3558
Next? (y/n)
n
```

Beispiel 30: Gesamte Datei auslesen

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
set PHONEBOOK "phones"
```

```
set f [open $PHONEBOOK r]

while { [gets $f line] >= 0} {
    puts stdout $line
}
close $f
```

Ausgabe:

```
>getall
Ralf : 5857
Balser : 3558
...
Schweiggert : 3570
Stanglow : 3574
```

Beispiel 31: Kopieren von Datei1 -> Datei2

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set PHONEBOOK "phones"

set f_in [open $PHONEBOOK r]
set f_out [open XXX w]
```

```
while { [gets $f_in line] >= 0} {
    puts $f_out $line
}
close $f_in
close $f_out
```

7. Strings

Grundlegende Operatoren: **string, append, format, scan, regexp, regsub und glob**

7.1. Das string-Kommando

Kommando-Syntax: **string operation stringvalue otherargs**

- *operation* entscheidet was *string* tut
- das zweite Argument ist ein String
- weitere Argumente können je nach *operation* folgen
- manche Operationen erlauben die Angabe von Indexbereichen, gezählt wird von 0 an, *end* gibt das letzte Zeichen an

Beispiel 32:

```
Tcl> string range abcdefgh 3 end
defgh
```

Das string-Kommando	
string compare strl str2	lexikographischer Vergleich von *str1* mit *str2*: kleiner (-1), gleich (0) oder größer (I)
string first strl str2	liefert Index in *str2*, wo zum ersten Mal *str1* beginnt; wenn nicht enthalten, - 1
string index str charInd	liefert das Zeichen, welches in *str* an der durch *charInd* bezeichneten Stelle steht.
string last strl str2	liefert die Position des am weitesten rechts stehenden Teilstrings von *str2*, welcher mit *str1* übereinstimmt

string length str	liefert die Anzahl von Zeichen in *str*
string match pattern str	Wenn *str* zur von *pattern* definierten Sprache gehört, wird 1 zurückgegeben, ansonsten 0. Verglichen wird nach den → *glob*-Regeln.
string range str i j	liefert den Teilstring von *str*, der im Bereich i bis j liegt; das erste Zeichen hat Index 0, das letzte kann mit *end* angegebenen werden
string tolower str	liefert *str* in Kleinbuchstaben
string toupper str	liefert *str* in Großbuchstaben
string trim str ?chars ?	Alle Zeichen am Anfang und am Ende von str, welche in *chars* angegeben sind, werden entfernt, Rest wird zurückgegeben. Voreinstellung für *chars* ist *white space*.
string trimleft str .?chars?	Gleich wie *string trim*, allerdings werden nur die führenden Zeichen entfernt.
string trimright str .?chars?	Gleich wie *string trim*, allerdings werden nur die am Schluß stehenden Zeichen entfernt.
string wordend str ix	Liefern den Index des Zeichens in *str* nach dem Wort, das das Zeichen bei Index *ix* enthält.
string wordstart str ix	Liefert den Index in *str* des ersten Zeichens in dem Wort, das das Zeichen bei Index *ix* enthält.

7.2. Strings und expressions

Der Vergleich zweier Strings auf Identität kann mit

- string compare oder
- mit dem ==-Operator (expr-Kommando)

erfolgen.

Nachteil bei der Auswertung mit dem ==-Operator ist die Umwandlung in Zahlen, falls möglich und die Zurückwandelung, falls ein string-Vergleich entdeckt wurde.

Zum string-Vergleich string compare verwenden!

Beispiel 33:

```
Tcl> if {[string compare $str1 $str2] == 0 } { .... }
```

7.3. string match –Kommando

Kommando-Syntax: **string match pattern str**

Muster (*glob*-Regeln):

- *: beliebige Zeichenfolge
- ?: beliebiges Einzelzeichen
- […]: Aufzählung von Zeichen
  (Achtung […] dienen auch der Kommandosubstitution, also in {…} setzen)

Rückgabewerte: 1, falls *str* zur von *pattern* definieren Sprache gehört

0, sonst.

Beispiel 34: string match –Kommando

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

puts "Ergebnis von \[string match a* alpha\]: \
   [ string match a* alpha]"
puts "Ergebnis von \[string match a? alpha\]: \
   [ string match a? alpha]"
puts "Ergebnis von \[string match ?? AA\]: \
   [ string match ?? AA]"
puts "Ergebnis von \[string match a*b alpha\]: \
   [ string match a*b alpha]"
puts "Ergebnis von \[string match \{\[abc\]*\} alpha\]:
\
   [ string match {[abc]*} alpha]"
```

```
set pat {[abc]*X}
puts "set pat \{\[abc\]*X\}"
puts "pat=$pat"
puts "Ergebnis von \[string match \$pat alTreX\]: \
   [string match $pat alTreX]"
puts "Ergebnis von \[string match \{*\\*\} abc*\]: \
   [string match {*\*} abc*]"
puts "Ergebnis von \[string match \{\\\\?\} \{\\a\}\]:
\
   [string match {\\?} {\a}]"
```

Ausgabe:

```
Ergebnis von [string match a* alpha]:  1
Ergebnis von [string match a? alpha]:  0
Ergebnis von [string match ?? AA]:  1
Ergebnis von [string match a*b alpha]:  0
Ergebnis von [string match {[abc]*} alpha]:  1
set pat {[abc]*X}
pat=[abc]*X
Ergebnis von [string match $pat alTreX]:  1
Ergebnis von [string match {*\*} abc*]:  1
Ergebnis von [string match {\\?} {\a}]:  1
```

7.4. format–Kommando

Formatierung von Zeichenketten entsprechend der gegebenen Formatspezifikation.

Komando-Syntax: **format spec_arg value1 value2 …**

- spec_arg enthält Literale, die im Ergebnis enthalten sind wie dargestellt und Schlüsselworte zur Formatierung
- spec_arg wird in Doppelapostrophen geklammert, da Leeraum enthalten sein kann, backslash-Substitutionen durchgeführt werden sollen, beispielsweise \t,\n, und spezielle Zeichen ($) substituiert werden sollen.
- Format-Schlüsselworte beginnen mit einem %, gefolgt von Format-Flags und abschließend Konvertierungszeichen
- Angabe i$ (mit i eine Zahl) wird das i-te korrespondierende Argument genommen, gezählt wird ab 1

Beispiel 35: format–Kommando

```
tcl>set position 3

tcl>puts [format "%${position}\$s" eins zwei drei vier]
```

Ausgabe:

```
drei
```

Konvertierungszeichen	
d	dezimale Ganzzahl
u	dezimale Ganzzahl ohne Vorzeichen
i	dezimale Ganzzahl; das Argument kann eine Zahl im hexadezimalen oder oktalen Format sein;
o	oktale Zahl ohne Vorzeichen
x oder X	hexadezimale Zahl (Klein-oder Großzeichenformat)
c	einzelnes Zeichen
s	Zeichenkette
f	dezimale Gleitpunktzahl im Format a.b
e oder E	dezimale Gleitpunktzahl im wissenschaftlichen Format a.bE+-c.
g oder G	dezimale Gleitpunktzahl im Format %f oder %e, je nachdem welches kürzer ist;

Formatierungszeichen	
-	Ausrichtung des Feldes links
+	Vorzeichen angeben, auch +.
space	Falls ein Zahl kein Vorzeichen hat, wird Leerraum vorangestellt
0	mit Nullen auffüllen
#	Einer Oktalzahl 0, eine Hexadezimalzahl 0x voranstellen, Dezimalpunkt grundsätzlich angeben, hintere Nachkomma-Nullen nicht entfernen (%g)

Beispiel 36: format–Kommando

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
set language 2
puts [format "%${language}\$s" one eins un uno]
puts "[ format "%x" 20 ] als Hexzahl"
puts "[format "%#x" 20 ] als Hexzahl mit Praefix"
set maxLength 8
puts [format "%*s|%s" $maxLength 10 10]
puts [format "%-*s|%s" $maxLength 10 10]
```

Ausgabe:

```
eins
14 als Hexzahl
0x14 als Hexzahl mit Praefix
      10|10
10      |10
```

7.5. scan–Kommando

Komando-Syntax: **scan string format var ?var? ?var? …**

- Formatierungszeichen wie bei format-Kommando mit den Ausnahmen:
  - kein %u (unsigned integer)
  - keine Feldweite bei %c
- zusätzlich gibt es eine Mengennotation. Mit [ ] kann eine Menge an Zeichen bestimmt werden. Diese Menge gibt die Zeichen an, welche in die Variable kopiert werden sollen.

Beispiel 37: scan–Kommando

```
#! /bin/sh

# the next line restarts using tclsh \

exec tclsh "$0" "$@"


puts [ scan "123 456 abc" "%d %d %s" x y s ]

puts "x=$x, y=$y, s=$s"

puts [ scan abcd {%[ab]} res ] ; #{} ist hier wichtig

puts "res=$res"

set res ""

puts [scan abcdEFGhij {%[a-z]} res] ; \

#[] haben Bedeutung

puts "res=$res"

set res ""

puts [scan XYabcdEFGhij {%[^a-z]} res] ; \

# vom Anfang her

puts "res=$res"
```

```
set res ""
puts [scan abcdEFGhij {%[A-Z]} res] ; # vom Anfang her
puts "res=$res"
```

Ausgabe:

```
>scan.tcl
3
x=123, y=456, s=abc
1
res=ab
1
res=abcd
1
res=XY
0
res=
```

7.6. regexp–Kommando und reguläre Ausdrücke

- Musterbestimmung mit regulären Ausdrücken
- Extraktion von „Teilzeichenketten“ aus einer Zeichenfolge passend zu einem vorgegebenem Muster

Beispiele siehe Welch Seite 25/26

Reguläre Ausdrücke

c	Terminalzeichen c
\c	Escapesequenz, oder falls c Meta-Zeichen, c als Terminalsymbol
^	Anfang (Zeile, Wort) - Meta-Zeichen
$	Ende (Zeile, Wort) - Meta-Zeichen
.	beliebiges Einzel-Zeichen - Meta-Zeichen
[abc…]	ein Terminalzeichen aus der Aufzahlung zwischen []
[^abc…]	alle Zeichen als Terminalzeichen außer den Aufgezählten in [ ]
[a-f]	alle Zeichen zwischen a und f nach ASCII-Ordnung als Terminalzeichen
[^a-f]	alle Zeichen außer denen zwischen a und f als Terminalzeichen
r1 | r2	rl oder r2
(r1)(r2)	rl gefolgt von r2
(r)*	beliebige viele r's hintereinander (auch keins)
(r)+	beliebig viele, aber mindestens ein r
(r)?	ein oder kein r
(r)	Klammern dienen der Gruppierung - Meta-Zeichen

Regel: first, then longest

Komando-Syntax: **regexp ?flags? pattern sting ?match sub1 sub2 …?**

- flags:
  -nocase Großbuchstaben in string werden als Kleinbuchstaben betrachtet
  -indices zwei Indices geben den Bereich an, in dem in string gesucht werden soll
  -- beginnt das pattern mit einem -,
  so kann damit flags von pattern getrennt werden

- pattern:
  Vorsicht: Einige Metasymbole haben für tcl Bedeutung, deshalb das Muster in {…} setzen oder in "…" und die tcl-Sonderzeichen mit backslash schützen.

- Rückgabewerte: Falls ein Teilstring von string zu der von pattern erzeugten Sprache gehört, wird 1, sonst 0 zurückgliefert. Im ersten Fall wird der Teilstring in der Variable match abgelegt; die restlichen Variablen werden mit den Teilstrings von string belegt, die zu den Teilmustern von pattern „passen“.

Beispiel 38: reguläre Ausdrücke

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
set str 123abcde56789x
set pat {[0-9]+[a-z]*}
puts "$pat"
puts [regexp $pat $str]
puts [regexp $pat $str match]
puts "match=$match"
set pat {([0-9]+).}
puts "$pat"
puts [regexp $pat $str match sub]
puts "match=$match, submach=$sub"
# HOSTDISPLAY = hostname:display.screen
puts "$env(HOSTDISPLAY)"
#hostname ermitteln:
puts [regexp {([^:]*):} $env(HOSTDISPLAY) match host]
puts "match=$match, HOST=$host"
# oder so:
puts [regexp (\[^:\]*):(.*) $env(HOSTDISPLAY) match
host number]
puts "match=$match, HOST=$host, number=$number"
```

Ausgabe:

```
>regexp.tcl
[0-9]+[a-z]*
1
1
match=123abcde
([0-9]+).
1
match=123a, submach=123
hypatia:0.0
1
match=hypatia:, HOST=hypatia
1
match=hypatia:0.0, HOST=hypatia, number=0.0
```

Beispiel 39: Ein Muster für URLs (Variante 1)

```
Tcl> set url http://www.sun.com:80/index.html
Tcl>regexp {([^:]+)://([^:/]+)(:([0-9]+))?(/.*)} $url\
match protocol server x port path
```

Ausgabe:

```
1
Tcl> set match
http://www.sun.com:80/index.html
Tcl> set protokol
http
```

```
Tcl> set server
www.sun.com
Tcl> set x
:80
Tcl> set port
80
Tcl> set path
/index.html
```

Beispiel 40: Ein Muster für URLs (Variante 2)

```
Tcl> set url http://www.sun.com/index.html
http://www.sun.com/index.html
Tcl> regexp {([^:]+)://([^:/]+)(:([0-9]+))?(/.*)} $url\
match protocol server x port path
```

Ausgabe:

```
1
Tcl> set match
http://www.sun.com/index.html
Tcl> set protokol
http
Tcl> set server
www.sun.com
Tcl> set x
```

```
Tcl> set port
Tcl> set path
/index.html
```

Beispiel 41: Ein Muster für URLs (Variante 3)

```
Tcl> regexp {([^:]+)://([^:/]+)(:([0-9]+))(/.*)} $url\
match protocol server x port path
0
```

Ausgabe:

```
Tcl> set $match
can’t read "match": no such variable
Tcl> set $protocol
can’t read "protocol": no such variable
Tcl> set $server
can’t read "server": no such variable
% set $x
can’t read "ab"c": no such variable
% set $port
can’t read "port": no such variable
% set $path
can’t read "path": no such variable
```

7.7. regsub–Kommando

String-Ersetzung basierend auf regulären Ausdrücken

Komando-Syntax: **regsub ?switches? pattern sting subspec varname**

- switches
  – all alle Vorkommen ersetzen, sonst nur das Erste
  – nocase Großbuchstaben werden zur Untersuchung in Kleinbuchstaben umgewandelt
  -- zur Trennung von *flags* und *pattern*, falls *pattern* mit – beginnt
- Spezialzeichen in subspec
  & ersetzt durch den zu *pattern* passenden Teilstring
  \1 …\9 ersetzt durch die Teilstrings, die zu den entsprechenden Teilmustern in *pattern* passen – Ordnung durch die linken Klammern (
- Rückgabewert Anzahl der Treffer bzw. Ersetzungen
- kopiert string nach varname, wobei Teilstrings entsprechend pattern durch subspec erstezt werden

Beispiel 42 : Spaltentausch

```
#! /bin/sh

# the next line restarts using tclsh \

exec tclsh "$0" "$@"

set line Franz^Schweiggert^Illertissen

puts "vorher: $line"

puts [regsub {([^^]+)\^([^^]+)\^} $line {\2^\1^} \
swapline]

puts "nachher: $swapline"
```

Ausgabe:

```
>regsub1.tcl
vorher: Franz^Schweiggert^Illertissen
1
nachher: Schweiggert^Franz^Illertissen
```

Beispiel 43: Spalten verdoppeln

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
set line Franz^Schweiggert^Illertissen
puts "vorher: $line"
puts [regsub {[^^]+} $line &^& newline]
puts "nachher: $newline"
```

Ausgabe:

```
>regsub2.tcl
vorher: Franz^Schweiggert^Illertissen
1
nachher: Franz^Franz^Schweiggert^Illertissen
```

8. Datenstrukturen

8.1. Einfache Variablen

Noch kurz etwas zu einfachen Variablen. Die Kommandos set und unset wurden schon besprochen. Mit dem Kommando *info exists* kann die Existenz einer Variablen getestet werden.

Komando-Syntax: **info exists variable**

Beispiel 44: Kommando info exists

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
set var {Wert von var}
```

```
set name var
puts "name=$name"
puts "var=$var"
puts "name=[set name]"
puts [info exists name]
puts [info exists x]
unset name
puts [info exists name]
```

Ausgabe:

```
>simplevar.tcl
name=var
var=Wert von var
name=var
1
0
0
```

8.2. Listen

- Strings mit spezieller Interpretation
- Eine Liste hat die gleiche Struktur wie ein Kommando
- Listen sind Strings
- Listenelemente werden durch Leerzeichen getrennt
- Geschweifte Klammern und Doppelapostrophen können zur Gruppierung verwendet werden (Listen als Elemente von Listen)
- Bei großen Datenmengen sind Listen nicht sehr effizient, da eine Liste bei jedem Zugriff analysiert werden muß (→ Arrays).

Listen-Kommandos
list arg1 arg2 ... Erzeugen einer Liste aus den angegebenen Argumenten
concat list {*list*} Konkatenieren von Listen
join list .?joinString? Die Elemente von *list* werden mit *joinString* als Trenner zurückgegeben. Voreinstellung von *joinString* ist das Leerzeichen.
lappend varName value {*value*} *value* wird als Listenelement an die Variable *varName* angehängt. Als Ergebnis wird die neue Liste zurückgegeben. Wenn *varName* noch nicht existiert, so wird die Variable angelegt.
lindex list index Gibt das Element an Stelle *index* zurück. Das erste Element der Liste hat den Index 0.
linsert list index value {*value*} Als Ergebnis wird eine neue Liste mit den zusätzlichen Elementen *value* zurückgegeben; eingefügt wird vor der Stelle *index*.
llength list Liefert die Anzahl der Elemente in *list*
lrange list first last Gibt die Listenelemente von Stelle *first* bis Stelle *last* aus. Wird für *last* der Wert *end* übergeben, so wird der komplette hintere Teil der Liste ausgegeben.
lreplace list first last {*value*} Eine neue Liste, in welcher die Elemente zwischen den Stellen *first* und *last* durch die *value*-Werte ersetzt wurden, wird ausgegeben. Werden keine Argumente für *value* angegeben, so werden die zwischen Stelle *first* und *last* stehenden Argumente ausgelassen.
lsearch ?-exact? ?-glob? ?-regexp? list pattern Rückgabewert ist der Index des ersten Listenelements, welches dem Muster *pattern* entspricht. Wird kein solches Element gefunden, so wird –1 zurückgegeben. Die Art der Suche kann mit den Optionen (Voreinstellung *-glob*) beeinflußt werden. Ist *-glob* gesetzt, wird das Suchmuster nach den Regeln der Dateinamensexpansion der Bourne-Shell bewertet. Bei *-exact* muß *pattern* mit dem Element übereinstimmen und bei *–regexp* wird nach dem regulären Ausdruck *pattern* gesucht.
lsort ?-ascii? ?-integer? ?-real? ?-command command? ?-increasing? ?-decreasing? list Als Ergebnis werden die Elemente von *list* sortiert zurückgegeben. Per Option

kann die Sortierreihenfolge bestimmt werden. Voreinstellung ist *ascii* und *increasing*.
split string ?splitChars? Erzeugt aus *string* eine Liste, wobei die *splitChars* als Trenner benutzt werden.

Beispiel 45: Anlegen einer Liste und Anfügen an eine Liste

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set x 1
set liste [list $x 2 {8 9}]
puts "liste = $liste"
set liste [lappend liste \$ 10 11]
puts "liste = $liste"
set liste [lappend liste \ ]
puts "liste = $liste"
set liste [lappend liste \" ]
puts "liste = $liste"
set liste [lappend liste \] ]
puts "liste = $liste"
set liste [lappend liste \} ]
puts "liste = $liste"
puts "Listenlaenge = [llength $liste]"
```

Ausgabe:

```
liste = 1 2 {8 9}
liste = 1 2 {8 9} {$} 10 11
liste = 1 2 {8 9} {$} 10 11 { }
liste = 1 2 {8 9} {$} 10 11 { } {"}
liste = 1 2 {8 9} {$} 10 11 { } {"} \]
liste = 1 2 {8 9} {$} 10 11 { } {"} \] \}
Listenlaenge = 10
```

Beispiel 46: Anfügen an eine Liste ohne diese vorher anzulegen

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set liste [lappend liste 1 2 {a b} ]
puts "liste = $liste"
```

Ausgabe:

```
liste = 1 2 {a b}
```

Beispiel 47: Unterschied list und concat

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set x {a b c}
```

```
set y {1 2 3 4 5 6}
set z X
set liste [concat $x $y $z]
puts "liste = $liste"
set liste [concat "$x" $y "$z"]
puts "liste = $liste"

# Aber:

set x {a b c}
set y {1 2 3 4 5 6}
set z X
set liste [list $x $y $z]
puts "liste = $liste"
```

Ausgabe:

```
liste = a b c 1 2 3 4 5 6 X
liste = a b c 1 2 3 4 5 6 X
liste = {a b c} {1 2 3 4 5 6} X
```

Beispiel 48: Zugreifen auf verschiedene Listenelemente

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
```

```
set liste [list {1 2} a b c ]
puts "liste = $liste"
puts "Anzahl Elemente: [llength $liste]"
puts "Das erste Element (Index 0): [lindex $liste 0]"
puts "Das letzte Element: [lindex $liste end]"
puts "von Index 1 bis Ende: [lrange $liste 1 end]"
puts "Das vorletzte Element: [lindex $liste [expr \
[llength $liste] - 2]]"
```

Ausgabe:

```
liste = {1 2} a b c
Anzahl Elemente: 4
Das erste Element (Index 0): 1 2
Das letzte Element: c
von Index 1 bis Ende: a b c
Das vorletzte Element: b
```

Beispiel 49: Einfügen und ersetzen von Elementen in einer Liste

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set liste [list {1 2} a b c]
puts "Liste: $liste"
puts "Einfuegen: [linsert $liste 0 X Y]"
```

```
puts "Liste: $liste"
set liste [linsert $liste 0 X Y]
puts "Liste: $liste"
set liste [lreplace $liste 0 1 0]
puts "Liste: $liste"
set liste [lreplace $liste 2 end X]
puts "Liste: $liste"
set liste [lreplace $liste 2 2 XXX]
puts "Liste: $liste"
```

Ausgabe:

```
Liste: {1 2} a b c
Einfuegen: X Y {1 2} a b c
Liste: {1 2} a b c
Liste: X Y {1 2} a b c
Liste: 0 {1 2} a b c
Liste: 0 {1 2} X
Liste: 0 {1 2} XXX
```

Beispiel 50: Suchen und Löschen in einer Liste

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set liste [list dies sind die Elemente]
puts "Liste: $liste"
```

```
puts "lsearch -exact \$liste die : [lsearch -exact
$liste die]"
puts "lsearch \$liste die* : [lsearch $liste die*]"
puts "lsearch -regexp \$liste {^s.*d\$} : [lsearch -
regexp $liste ^s.*d\$]"

proc ldelete {liste wert } {
   set ix [lsearch -exact $liste $wert ]
   if {$ix >=0} {
       return [lreplace $liste $ix $ix]
   } else {
       return $liste
   }
}
puts "Loeschen: [ldelete $liste die]"
puts "Liste: $liste"
set liste [ldelete $liste die]
puts "Liste: $liste"
```

Ausgabe:

```
Liste: dies sind die Elemente
lsearch -exact $liste die : 2
lsearch $liste die* : 0
lsearch -regexp $liste {^s.*d$} : 1
Loeschen: dies sind Elemente
```

```
Liste: dies sind die Elemente
Liste: dies sind Elemente
```

Beispiel 51: Liste sortieren

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set liste { {Mathias Etter} {Matthias Grabert}
   {Andreas F. Borchert} {Franz Schweiggert} }

puts "Sortiert:"
puts "[lsort $liste]"

proc famcmp {a b} {
   set aLast [lindex $a [expr [llength $a] -1]]
   set bLast [lindex $b [expr [llength $b] -1]]
   set res [string compare $aLast $bLast]
   if {$res != 0} {
       return $res
   } else {
       return [string compare $a $b]
   }
}
```

```
puts "Nach FamNamen sortiert:"
puts "[lsort -command famcmp $liste]"
```

Ausgabe:

```
Sortiert:
{Andreas F. Borchert} {Franz Schweiggert} {Mathias
Etter} {Matthias Grabert}
Nach FamNamen sortiert:
{Andreas F. Borchert} {Mathias Etter} {Matthias
Grabert} {Franz Schweiggert}
```

Beispiel 52: split-Kommando

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set DB_FILE "student"
set SEP ^
set f [open $DB_FILE r] ; #zum Lesen oeffnen

gets $f line
set list [split $line $SEP]
puts [format "%-10s | %-16s | %-10s" [lindex $list 1] \
                    [lindex $list 2] \
                    [lindex $list 0] ]
```

```
puts [format "%30s" ----------+------------------+---\
----- ]
while { [gets $f line] > 0 } {
   set list [split $line $SEP]
   puts [format "%-10s | %-16s | %-10s" [\
   lindex $list 1] \
                  [lindex $list 2] \
                  [lindex $list 0] ]
}
```

Ausgabe:

```
Name       | Fachrichtung     | Login
-----------+------------------+---------
Maier      | WiMa             | gonzales
Huber      | WiMa             | joerg
Schroeder  | Mathematik       | mg
Meyer      | Informatik       | nobody
Schmidt    | WiMa             | ralf
Huber      | Informatik       | speedy
```

Beispiel 53: split und join-Kommando

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"
```

```
set DB_FILE "student"
set SEP ^
set NEWSEP @

set f [open $DB_FILE r] ; #zum Lesen oeffnen

gets $f line
set list [split $line $SEP]
set x [list [lindex $list 1] [lindex $list 2] ]
puts [join $x $NEWSEP]
while { [gets $f line] > 0} {
   set list [split $line $SEP]
   set x [list [lindex $list 1] [lindex $list 2] ]
   puts [join $x $NEWSEP]
}
```

Ausgabe:

```
Name@Fachrichtung
Maier@WiMa
Huber@WiMa
Schroeder@Mathematik
Meyer@Informatik
Schmidt@WiMa
Huber@Informatik
```

8.3. (Assoziative) Arrays

Definition: **set arr(index) value**

- Arrays sind Variablen deren Index ein String ist, also assoziative Arrays.
- Intern werden Arrays in einer hash-Tabelle verwaltet.
- Der Index kann jeden String-Wert annehmen und auch das Ergebnis einer Variablen- oder Kommandosubstitution sein. Vorsicht bei Leerzeichen im Index.

Beispiel 54: Die Tücke mit den Leerzeichen

```
Tcl> set my_arr(Index des Arrays) Inhalt
wrong # args: should be "set varName ?newValue?"
Tcl> set {my_arr(Index des Arrays)} Inhalt
Inhalt
Tcl> info exists my_arr
1
Tcl> unset my_arr
```

oder so:Index als Ergebnis einer Substitution

```
Tcl> set index {Index des Arrays}
Index des Arrays
Tcl> set my_arr($index) Inhalt
Inhalt
```

- Auch Arraynamen können das Ergebnis einer Substitution sein.

Beispiel 55:

```
Tcl> set name my_arr
my_arr
```

```
Tcl> set ${name}(Index1) {Wert1}
Wert1
Tcl> set x $my_arr(Index1)
Wert1
Tcl> set y ${name}(Index1)
my_arr(Index1)
Tcl> set x [set ${name}(Index1)]
Wert1
```

- Namenskonflikte treten auf, wenn eine Variable als „normale“ Variable und als array verwendet wird.

Beispiel 56:

```
Tcl> set my_arr 123
```

Ausgabe:

```
can’t set "my_arr": variable is array
```

- Elemente werden mit lappend angehängt.

Beispiel 57:

```
Tcl> lappend my_arr(Index2) Wert2
```

Ausgabe:

```
Wert2
```

Kommando-Syntax: **array operation arrname ?otherargs?**

Array-Kommando	
array exists arrname	1 wenn *arrname* ein Array-Name
array get arrname	Liefert liste aus den Paaren (Indexname, Wert)
array set arrname list	Initialisiert das Array *arrname* aus der Liste *list*; diese muß das Format haben, wie es *array get* liefert!
array anymore arrname id	Rückgabewert ist 1, falls noch weitere Elemente bei der durch *id* spezifizierten Suche zu bearbeiten sind, ansonsten 0
array donesearch arrname id	Die durch *id* spezifizierte Suche wird beendet, jeglicher Zustand der Suche wird verworfen. Rückgabewert ist ein Leerstring Rückgabewert ist eine Liste aller Indizes des Arrays, oder jener, die zu *pattern* passen
array names arrname .?pattern?	Rückgabewert ist eine Liste aller Indices des Arrays, oder jener, die zu *pattern* (→ string match) passen.
array nextelement arrname id	Gibt den Namen des nächsten Feldelementes von *arrname*, bezogen auf den durch *id* bezeichneten Suchvorgang, zurück. Ist die Suche beendet, wird ein Leerstring zurückgegeben
array size arrname	Rückgabewert ist Elementanzahl des Arrays
array startsearch arrname	Ein Suchdurchlauf über alle Feldelemente von *arrname* wird initialisiert. Rückgabewert ist ein eindeutiger Bezeichner für diesen Durchlauf. Dieser wird für die Kommandos *array anymore, array donesearch, array nextelement* verwendet

Durchlaufen eines Arrays:**foreach index [array names arrname] { Anweisungen}**

Beispiel 58: Durchlaufen eines Arrays

```
Tcl> foreach index [array names my_arr] \
{puts $my_arr($index)}
```

Ausgabe:

```
Wert1
Wert2
Inhalt
```

Beispiel 59: Join über zwei Dateien

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set FILE1 "student"
set FILE2 "verbrauch"
set SEP ^
set pattern "^(\[^$SEP\]*)\\$SEP\(.*)\$" ; \
# 1. Attr / Rest

set f1 [open $FILE1 r] ; #zum Lesen oeffnen
set f2 [open $FILE2 r] ; #zum Lesen oeffnen

gets $f1 attr1 ; # Attributzeile weglesen
gets $f2 attr2 ; # Attributzeile weglesen

while { [gets $f1 line] > 0 } {
    regexp $pattern "$line" match login rest
```

```
    set studArr($login) "$rest$SEP"
}
while { [gets $f2 line] > 0 } {
    regexp $pattern "$line" match login rest
    if { [info exist studArr($login) ] } {
        append studArr($login) $rest
    }
}

foreach index [array names studArr] {
    # keine Seiten eingetragen -> $SEP am Zeilenende:
    if { [ regexp "\\$SEP\$" $studArr($index) match ] \
        == 0 } {
        puts "$index$SEP$studArr($index)"
    }
}
```

Ausgabe:

```
mg^Schroeder^Mathematik^123
speedy^Huber^Informatik^1234
nobody^Meyer^Informatik^4
gonzales^Maier^WiMa^12
```

Beispiel 60: mehrdimensionale Arrays

```
#! /bin/sh
```

```
# the next line restarts using tclsh \
exec tclsh "$0" "$@"

set dim 5

set i 0
while { $i < $dim } {
    set j 0
    while { $j < $dim } {
        set arr($i,$j) [expr $i+$j+2]
        incr j
    }
    incr i
}

puts "Matrix:"
set i 0
while { $i < $dim} {
    set j 0
    while { $j < $dim } {
        puts -nonewline "$arr($i,$j) "
        incr j
    }
```

```
    incr i
    puts ""
}

puts "Diagonale:"
set i 0
while {$i < $dim } {
    puts -nonewline "$arr($i,$i) "
    incr i
}
puts ""
```

Ausgabe:

```
Matrix:
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
6 7 8 9 10
Diagonale:
2 4 6 8 10
```

9. Kontrollstruktur-Kommandos

Kontrollstrukturen: **break, continue, eval, for, foreach, if return, switch, while,**

9.1. Übersicht

Kommandos wie if, for und while beinhalten Boolsche Ausdrücke, sie greifen auf das expr-Kommando zurück.

Kontrollstrukturen	
break	Beendet die innerste Schleife.
continue	Setzt die innerste Schleife mit der nächsten Iteration fort.
eval arg {arg}	Verbindet die Argumente mit Leerzeichen und führt das dadurch entstehende Skript aus. Das Ergebois wird zurückgegeben.
for init test reinit body	Entspricht der for-schleife in C. init, reinit und body sind gültige Tcl-Skripte, test ein Ausdruck, welcher vor jedem Schleifendurchlauf ausgewertet wird. Ergibt sich dabei 0, also False, wird die Schleife beendet.
foreach varname list body	Die Elemente von list werden sukzessive durchlaufen, der jeweilige Wert wird an die Variable varname zugewiesen und das Tcl-Skript body wird ausgeführt. Ein leerer String wird zurückgegeben.
if test body {elseif test body} ?else body?	Entspricht dem if - elseif - else Konstrukt in C.
source fileName	Das in der Datei fileName stehende Tcl-Skript wird eingelesen und ausgeführt. Gibt das Ergebnis des Skriptes zurück. Steht innerhalb des Skriptes das return-Kommando wird seine Ausführung beendet.
switch ?options? string pattern body {pattern body} switch ?options? string {pattern body {pattern body}}	Vergleicht string nacheinander mit den patterns und führt bei Übereinstimmung den jeweiligen body als Tcl-Skript aus. Als Ergebnis wird dann die Ausgabe von body zurückgegeben. Ist als letztes pattern default angegeben, so entspricht es immer dem Suchkriterium. Als Optionen sind -exact, -glob, -regexp oder -- zur Anzeige, daß die Optionen beendet sind zugelassen.
while test body	Wenn der Ausdruck test wahr ist, wird body ausgeführt und mit der erneuten Bewertung von test fortgefahren. Teminiert, wenn test mit Null (0) bewertet wird. Gibt einen leeren String zurück.

9.2. if-Kommando

Kommandosyntax: **if boolean then body1 ((elseif body)? else body2)?**

Beispiel 61:

```
if {$x == 0} {
   puts "Division durch Null!"
} else {
   puts [expr $y/$x]
}
```

9.3. switch-Kommando

Kommandosyntax: **switch flags value {pat1 body1 pat2 body2 pat3 body3 …}**

- *flags*: bestimmen wie *value* dem Muster entspricht
  – exact: exakte Übereinstimmung mit einem der Muster
  – glob (glob-Regeln siehe string match)
  – regexp Musterbeschreibung mit regulären Ausdrücken
  -- zur Anzeige, dass die Optionen beendet sind
- passen mehrere Muster , so wird nur der body des ersten passenden Falles abgearbeitet
- default-Fall:
  - falls *value* keinem der Muster entspricht,
  - muss immer als letzte Alternative ausgeführt sein
- Kommentare in switch-Anweisungen müssen dort stehen, wo der tcl-Interpreter auch ein Kommando erwarten würde. Sie dürfen nicht auf derselben Stufe wie die *patterns* stehen, sonst werden sie als Kommandos interpretiert.

Beispiel 62: Kommentare in switch-Anweisungen

```
switch -- $value {
    #dieser Kommentar verursacht Verwirrungen
    pattern {#hier ist der Kommentar \
                richtig angebracht}
```

```
}
```

- Vorsicht mit Klammerung und Substitution

Beispiel 63: Variante 1

```
while { [gets stdin monat] >= 0 } {


  switch -exact -- $monat {
   jan { puts Januar }
   feb { puts Februar }
   mar { puts Maerz }
   apr { puts April }
   may { puts Mai }
   jun { puts Juni }
   jul { puts Juli }
   aug { puts August }
   sep { puts September }
   oct { puts Oktober }
   nov { puts November }
   dec { puts Dezember }
   default { puts "Ungueltige Angabe" ; exit 1 }
  }
}
```

Beispiel 64: Variante 2

```
while { [gets stdin monat] >= 0 } {


  switch -exact -- $monat \
   jan { puts Januar } \
   feb { puts Februar } \
   mar { puts Maerz } \
   apr { puts April } \
   may { puts Mai } \
   jun { puts Juni } \
   jul { puts Juli } \
   aug { puts August } \
   sep { puts September } \
   oct { puts Oktober } \
   nov { puts November } \
   dec { puts Dezember } \
   default { puts "Ungueltige Angabe" ; exit 1 }
}
```

Beispiel 65: switch mit regulären Ausdrücken

```
set z1 0
set z2 0
set z3 0
set anfang A
```

```
while { [gets stdin eingabe ] >= 0 } {
    switch -regexp -- $eingabe \
        ^$anfang { incr z1 } \
        {^[0-9]+$} { incr z2 } \
        default { incr z3 }
}
puts "Anzahl mit $anfang am Anfang: $z1"
puts "Anzahl Zahlen: $z2"
puts "Anzahl andere: $z3"
```

9.4. foreach-Kommando

Kommandosyntax: **foreach loopVar valueList commandBody**

LoopVar Name einer Variablen, für jedes Element der Liste wird der *commandBody* ausgeführt.

Beispiel 66:

```
set liste {1 2 3 4 5 6 7 8 9}
set summe 0
foreach i $liste {
    set summe [ expr $summe + $i ]
}
puts "Summe: $summe"
```

9.5. for-Kommando

Kommandosyntax: **for initial test final body**

- *initial* Initialisierung der Schleife
- *test* Boolsche Bedingung, die geprüft wird und falls diese einen true-Wert liefert wird der body ausgeführt
- *final* wird nach der Body-Ausführung ausgeführt

Beispiel 67:

```
for {set i 0} {$i < 10} {incr i 3} {
    lappend myList $i
}
set myList
```

Ausgabe:

```
0 3 6 9
```

9.6. while -Kommando

Kommandosyntax: **while booleanExpr body**

Beispiel 68:

```
set anzahlZeilen 0; set anzahlZeichen 0
while {[gets stdin line] >=0} {
    incr anzahlZeilen
    incr anzahlZeichen [string length $line]
}
```

10. Prozeduren

Zugehörige Kommandos: **proc, rename, global, upvar**

10.1. Syntax

Kommando-Syntax: **proc name params body**

- Die Prozedur *name* wird zu einem Kommando, welches der tcl-Interpreter versteht.
- Nach der Definition kann *name* wie jedes andere tcl-Kommando verwendet werden.
- Ergebnis ist das des letzten in *body* ausgeführten Kommandos oder ein durch *return* definierter *string*
- Rekursionen sind möglich
- Prozeduren haben einen eigenen Namensraum, es kann also eine Variable und eine Prozedur den gleichen Namen haben.
- Zahl aktueller und formaler Parameter müssen nicht übereinstimmen. Der letze formale parameter muss dann *args* heißen, sonst gibt's einen Fehler!

Beispiel 69:

```
proc listArgs { a b c args } {
   puts "first:     $a"
   puts "second: $b"
   puts "third:    $c"
   puts "fourth: $args"
}
```

Gültige Aufrufbeispiele:

```
listArgs 1 2 3 4          a=1, b=2, c=3, args =4
listArgs 1 2              a=1, b=2
listArgs 1 2 3            a=1, b=2, c=3
listArgs 1 2 3 4 5 6      a=1, b=2, c=3, args = 4 5 6
```

- Parameter können mit *default*-Werten vorbelegt werden

Beispiel 70:

```
proc compute { a { b 2 } { c 10 } } {
    puts -nonewline "($a + $b) * $c gibt: "
        expr ( $a + $b ) * $c
}
```

- Prozeduren können verschachtelt werden, sind aber immer global sichtbar. Die Verschachtelung dient also nur der Verständlichkeit.
- *call-by-value* (→*upvar*-Kommando)
- Variablen, die außerhalb einer Prozedur vereinbart sind, sind in der Prozedur nicht sichtbar (→*global*-Kommando)

10.2. Lebensdauer

Jede Prozedur hat einen lokalen Bereich für ihre Variablen. Deshalb ist die Lebensdauer lokaler Variablen auf die Ausführung der Prozedur beschränkt. Wird eine Variable innerhalb und außerhalb einer Prozedur mit dem gleichen Namen verwendet, so haben diese keinen Einfluss aufeinander.

Beispiel 71: Lebensdauer von Variablen

```
Tcl> set a 123
Tcl> proc proc1 {a} { \
set a 42 ; return $a }
Tcl> proc1 1
42
Tcl> puts $a
123
```

10.3. Sichtbarkeit: global-Kommando

Im globalen Gültigkeitsbereich sind alle Variablen und Prozeduren enthalten. Dieser Gültigkeitsbereich liegt ausserhalb des Gültigkeitsbereichs der Prozeduren. Will man innerhalb einer Prozedur auf eine globale Variable zugreifen, so muss diese explizit

„importiert“ werden. Undefinierte Vraiablen werden dadurch definiert. Durch den Import in eine Prozedur wird die Variable nur in dieser Prozedur bekannt, in anderen Prozeduren muss dies wiederholt werden.

Kommando-Syntax: **global var1 var2 …**

Beispiel 72: Zufallszahlengenerator

```
proc randomInit {seed} {
   global rand
   set rand(ia)        9301; #Multiplier
   set rand(ic)      49297; #Constant
   set rand(im)   233280; #Divisor
   set rand(seed)   $seed; #Last result
}

proc random {} {
   global rand
   set rand(seed) \
       [expr ($rand(seed)* $rand(ia) + $rand(ic)) %
$rand(im)]
   return [expr $rand(seed)/double($rand(im))]
}
```

Die globalen Variabeln, welche in den Prozeduren benötigt werden sollten in einem Array zusammengefasst werden. Dann genügt jeweils ein Import in den Prozeduren. Das Management der globalen Variablen wird dadurch vereinfacht.

10.4. upvar-Kommando: call-by-reference

Kommando-Syntax: **upvar ?level? vrName localvar**

- Übergabe des Namens einer Variablen, nicht des Wertes (call-by-reference)
- upvar-Kommando assoziiert eine lokale Variable mit einer Variablen eines anderen Bereichs des tcl-Stacks
- level hat als default-Wert 1, #0 bezieht sich auf den globalen Gültigkeitsbereich
- mit #number kann absolut auf stack-Bereich Bezug genommen werden
- Namensübergabe ist zulässig bei skalaren Variablen und Array-Elementen, die lokale Variable ist dann skalar. Bei der Übergabe eines gesamten Arrays wird die lokale ebenfalls als Array betrachtet.

Beispiel 73:

```
proc do { varName } {
    upvar varName var
    puts $varName
    set var 5
}
set x 1
do x
puts $x
```

Ausgabe:

```
x
1
```

Beispiel 74:

```
proc do_again { varName } {
    upvar $varName var
    puts $varName
```

```
    set var 5
}
set x 1
do_again x
puts $x
```

Ausgabe:

```
x
5
```

Beispiel 75: Übergabe von Arrays (nach Welch)

```
proc push { stack value } {
    upvar $stack st
    if ![info exists st(top) ] {
        set st(top) 0
    }
    set st($st(top)) $value
incr st(top)
}

proc pop { stack } {
    upvar $stack st
    if ![info exists st(top) ] {
        return {}
```

```
    } else {
        incr st(top) -1
        set x $st($st(top))
        unset st($st(top))
        return $x
    }
}

push stack 1
push stack 2
push stack 3
puts [pop stack]
puts [pop stack]
push stack a
push stack b
puts [pop stack]
puts [pop stack]
puts [pop stack]
```

Ausgabe:

```
3
2
b
```

```
a
1
```

11. Das eval-Kommando

Kommando-Syntax: **eval arg1 arg2 …**

- expliziter Aufrauf des tcl-Interpreters
- zusätzliche Interpretation (Substitution)
- dynamische Konstruktion von Kommandos

Beispiel 76:

```
Tcl> set text "Hallo hier kommt Text!"
Hallo hier kommt Text!
Tcl> set cmd [list puts $text]
puts { Hallo hier kommt Text!}
Tcl> unset text
Tcl> $cmd
invalid command name "puts { Hallo hier kommt Text!}"
Tcl> eval $cmd
Hallo hier kommt Text!
```

Beispiel 77: list erhält die Listenstruktur, concat nicht!

```
Tcl> set x {1 2}
1 2
Tcl> set y [concat $x 3]
1 2 3
Tcl> set z [list $x $y]
```

```
{1 2} {1 2 3}
Tcl> set w [concat $x $y]
1 2 1 2 3
```

Beispiel 78: statt mit list mit concat

```
Tcl> set text "Hallo hallo!"
Hallo hallo!
Tcl> set cmd [concat puts $text]
puts Hallo hallo!
Tcl> eval $cmd
can not find channel named "Hallo"
```

- eval mit mehr als einem Argument

Beispiel 79:

```
set file stderr
set text "Hallo, Tcl'Fan's"
set cmd puts
eval $cmd $file $text
```

Ausgabe:

```
Error: bad argument "Tcl'Fan's": should be "nonewline"
    while executing
"puts stderr Hallo, Tcl'Fan's"
    ("eval" body line 1)
    invoked from within
```

```
"eval $cmd $file $text"
```

Beispiel 80:

```
set file stderr
set text {Hallo,\ Tcl’Fan’s}
set cmd puts
eval $cmd $file $text
```

Ausgabe:

```
Hallo, Tcl’Fan’s
```

1. ARBEITSWEISE DES TCL-INTERPRETERS 1

2. EINFACHE VARIABLEN 2

3. SUBSTITUTION 3

3.1. VARIABLEN UND ERSETZUNGEN ($) 3
3.2. KOMMANDO-SUBSTITUTION ([…]) 4
3.3. BACKSLASH-SUBSTITUTION (\) 5

4. GRUPPIERUNG 7

4.1. GRUPPIERUNG MIT DOPPELAPOSTROPHEN " … " 7
4.2. GRUPPIERUNG MIT GESCHWEIFTEN KLAMMERN { … } 8
4.3. UNTERSCHIED DER GRUPPIERUNGSARTEN 9
4.4. GRUPPIERUNG VOR SUBSTITUTUION 10
4.5. ZUSAMMENFASSUNG ZU GRUPPIERUNG UND SUBSTITUTION 11

5. NUMERISCHE AUSDRÜCKE 13

5.1. OPERANDEN: 13
5.2. OPERATOREN: 14
5.3. MATHEMATISCHE FUNKTIONEN 15

6. ARGV, FILE I/O, ENVIROMENT (ENV) 16

6.1. KOMMANDOZEILENARGUMENTE 16
6.2. UMGEBUNGSVARIABLE ENV 17
6.3. FILE I/O 17

7. STRINGS 22

7.1. DAS STRING-KOMMANDO 22
7.2. STRINGS UND EXPRESSIONS 23
7.3. STRING MATCH –KOMMANDO 24
7.4. FORMAT–KOMMANDO 25
7.5. SCAN–KOMMANDO 28
7.6. REGEXP–KOMMANDO UND REGULÄRE AUSDRÜCKE 29
Regel: first, then longest 30
7.7. REGSUB–KOMMANDO 34

8. DATENSTRUKTUREN 36

8.1. EINFACHE VARIABLEN 36
8.2. LISTEN 37
8.3. (ASSOZIATIVE) ARRAYS 49

9. KONTROLLSTRUKTUR-KOMMANDOS 55

9.1. ÜBERSICHT 56
9.2. IF-KOMMANDO 57
9.3. SWITCH-KOMMANDO 57
9.4. FOREACH-KOMMANDO 60
9.5. FOR-KOMMANDO 61
9.6. WHILE -KOMMANDO 61

10. PROZEDUREN 61

10.1. SYNTAX 62
10.2. LEBENSDAUER 63
10.3. SICHTBARKEIT: GLOBAL-KOMMANDO 63
10.4. UPVAR-KOMMANDO: CALL-BY-REFERENCE 64

11. DAS EVAL-KOMMANDO 68