Universität Ulm

Sektion Angewandte Informationsverarbeitung

Leiter: Prof. Dr. Franz Schweiggert

Vorlesung (4/2)

UNIX-basierte Implementierung kleiner Datenbanken I

Prof. Dr. Franz Schweiggert
und Mitarbeiter

Tel.: 0731/502-3570, 3571
email: swg@mathematik.uni-ulm.de
web: http://www.mathematik.uni-ulm.de/sai/

Inhalt

INHALT

Vorwort

- □ Kleine Datenbanken - z.B. Persönliches Telefonverzeichnis, Vereinsverwaltung, Tagungsorganisation, Textverwaltung, Literatur, Adreßbuch, ...
- □ Kleine Datenbanken - i.w. zum eigenen Gebrauch für den UNIX-Kenner

Daneben: Umgang mit UNIX, den Tools für viele andere Zwecke wie Textanalyse, Datenanalyse, ...

- Man muß nicht immer mit "Kanonen" (kommerzielle DB wie ORACLE, INGRES & Co.) auf "Spatzen" schießen
- Verwaltung von Daten in UNIX-Dateien (Textdateien) hat Vorteile: wir bleiben im ASCII-Bereich, keine speziellen Sonderzeichen wie manche DB sie ablegen, freier Zugang für eigene Programme

Verwendet werden UNIX-Shell-Tools und UNIX-Shellprogrammierung - doch (leider) ist Shell nicht gleich Shell, es gibt mindestens vier zwar eher wenig verschiedene, aber doch verschiedene Shells:

- fast überall und klassisch: die **Bourne**-Shell
- weiterentwickelt aus der Bourne-Shell: die **Korn**-Shell
- eher etwas exotisch: die **C**-Shell
- gängig weil GNU: die **bash**-Shell - the GNU **B**ourne-**A**gain **SH**ell

```
bash$ type sh
sh is /bin/sh
bash$ type ksh
ksh is /bin/ksh
bash$ type csh
type: csh: not found
bash$ type bash
bash is /usr/bin/bash
```

Relationale DB

im Schnelldurchgang

1. Tabellen, Relationen

1.1 Begriffe

Viele Daten im "täglichen Leben" sind in Form von Tabellen organisiert: Steuertabellen, Gehaltstabellen, Telefonverzeichnisse, ...

Partner						
Name	Vorname	Institution	Straße	PLZ	Wohnort	Telefon
Bauer	Karl					07303/3347
Spillner	Dr. Andreas	Uni	PF 330 440	W-2800	Bremen 33	0421/2182449
Zanger	Dr. Cornelia	TU	Mommsenstr. 13	OD-8027	Dresden	
Pesch	Dr. Herbert		Arabellastr. 5/917	81925	München	089/9102286
Koelsch	Dr. Raimund		Perlacher Str. 21	8000	München 90	089/65107120
Sonnig	Elmar		Baumreute 10	7257	Ditzingen	07152/56203
Wallmueller	Ernest	ATAG AG	Kanalstr. 31	CH-8152	Glattbrugg	01-810-9909
Wilken	Ernst		Hoervelsinger Weg 25	D-89081	Ulm	0731/9650-0
Burr	Barbara	URZ Stg	Allmandring 30	7000	Stuttgart 80	0711/1319203
Schmid	Werner	GPS	Hoervelsinger Weg	7900	Ulm	0731/966570
		Ciba-Geigy	R-1008.8.05	CH-4002	Basel	+41-61/6973419
		IBM Ulm				1515-0
Schreiner	Prof.Dr. A.	Uni	Albrechtstr. 28	D-4500	Osnabrueck	0541/969-2483
Krallmann	Prof.	TU Berlin				030/31473261

Als (mathematische) Relation:

Partner $\subseteq$ Name$\times$Vorname$\times$Institution$\times$Straße$\times$PLZ$\times$Wohnort$\times$Telefon

Partner =

{ (Bauer, Karl, , , , , 07303/3347),
(Spillner, Dr. Andreas, Uni, PF 330 440, W-2800, Bremen 33, 0421/2182449),
(Zanger, Dr. Cornelia, TU, Mommsenstr. 13, OD-8027, Dresden,),
(Pesch, Dr. Herbert, , Arabellastr. 5/917, 81925, München, 089/9102286),
(Koelsch, Dr. Raimund, , Perlacher Str. 21, 8000, München 90, 089/65107120),
(Sonnig, Elmar, , Baumreute 10, 7257, Ditzingen, 07152/56203),
(Wallmueller, Ernest, ATAG AG, Kanalstr. 31, CH-8152, Glattbrugg, 01-810-9909),
(Wilken, Ernst, , Hoervelsinger Weg 25, D-89081, Ulm, 0731/9650-0),
(Burr, Barbara, URZ Stg, Allmandring 30, 7000, Stuttgart 80, 0711/1319203),

(Schmid, Werner, GPS, Hoervelsinger Weg, 7900, Ulm, 0731/966570),
(, , Ciba-Geigy, R-1008.8.05, CH-4002, Basel, +41-61/6973419),
(, , IBM Ulm, , , 1515-0),
(Schreiner, Prof.Dr. A., Uni, Albrechtstr. 28, D-4500, Osnabrueck, 0541/969-2483),
(Krallmann, Prof., TU Berlin, , , , 030/31473261)
}

In einem (Modula-2) Programm als Liste

```
CONST TextLaenge = 30;

TYPE
        Text: ARRAY [0 .. Textlaenge] OF CHAR;

        PartnerPointer = POINTER TO PartnerRec;
        PartnerRec = RECORD
                        Name: Text;
                        Vorname: Text;
                        Institution: Text;
                        Strasse: Text;
                        PLZ: ARRAY [0 .. 8] (*???*) OF CHAR;
                        Wohnort: Text;
                        Telefon: Text
                    END;

VAR partner: PartnerPointer;
```

Zeilen der Tabelle / Tupel der Relation / Elemente der Liste:

- Beschreibungen von realen, gedanklich-abgrenzbaren "Objekten"
- abstrakt - Darstellung nur der "jeweils" relevanten Aspekte
  → Sicht, Interpretation, **Modell**
- Aspekte (Eigenschaften, Merkmale oder **Attribute**)
- Beschreibung so, daß je zwei Elemente in der Abstraktion verschieden sind
- Reihenfolge der Merkmale (Relation!) spielt keine Rolle, ihre Bedeutung ist durch
  – die Bezeichnung der Spalte in der Tabelle
  – die Bezeichnung der Menge innerhalb des kartesischen Produkts
  – die Bezeichnung des Komponententyps in der RECORD-Vereinbarung
  gegeben.

- Menge der aus Sicht der Realität möglichen Werte eines Attributs: **Wertebereich des Attributs** (→ Daten-TYP)
- nicht alle möglichen Kombinationen der Attributwerte sind aus der Sicht der Realität möglich (→ Integritätsbedingungen)
- Attributwerte sind elementar, besitzen aus der Sicht der Interpretation keine relevante Substruktur
  Beispiel: Attribut "Hobbies", "Sprachkenntnisse", ...

  → **Erste Normalform** (**1NF**), *First-Normal-Form*

Tabelle / Relation / Liste

- Homogene Struktur - Zusammenfassung von gleichartigen Objekten (bezüglich der Abstraktion, der Sicht, der Interpretation)
  → **Klassenbildung**
- Aufzufassen als "Objekt-Typ" oder als "Objekt-Klasse" oder als "Container" zur Aufnahme gleichartiger Objekte (Objekte als "Instanzen" des Typs)
- Aufzufassen auch als Menge von relevanten Objekten zum Zeitpunkt **t**
- Menge → Reihenfolge der Nennung darf keine Rolle spielen, es gibt aus Sicht der Interpretation kein erstes oder letztes Objekt - wenn doch, so ist dieses eine Eigenschaft und muß als solches beschrieben werden.
- Tabellen / Relation / Listen tragen einen Namen (**Klassenbezeichnung**)
- Korrekter Name des Attributs durch Angabe der Klassenbezeichnung gefolgt von der Angabe des lokal eindeutigen Attributnamens:
  Partner.Name, **Partner.Telefon**, ...

Klasse - Objekt (Instanz), Typ - Variable, Tabelle - Zeile, Relation - Tupel, ...

- Noch einmal: Alle Objekte einer Klasse sind verschieden
- Identifikation eines konkreten Objekts (Instanz) zu einem beliebigen Zeitpunkt:
  Schlüssel (*key*): identifizierende, minimale Attributkombination
  minimal: Weglassen eines Attributs der Kombination nur auf Kosten der Eindeutigkeit (Rest nicht mehr identifizierend)
  u.U. gibt es mehrere Schlüssel - verwendet man primär einen davon, so spricht man von diesem als **Primärschlüssel** (*primary key*)

- **1. Integritätsregel** (**Objektintegrität**): für jedes Objekt ist der Wert jedes Schlüsselattributs definiert (angegeben)!
  in obiger Partner-Tabelle NICHT gegeben!!! (Modellbildung???)
- Selektion eines konkreten Objekts einer Klasse:
  *<Klassenname>***.***<Schluessel>* = SchlüsselWert

Im allgemeinen ist ein realer Sachverhalt nicht durch eine einzige Tabelle zu beschreiben, es gibt mehrere durch gedankliche Abgrenzung entstandene Tabellen, zwischen denen aber eine Beziehung besteht:

Beispiel: Prüfungsverwaltung

Relevant sind (aus meiner Sicht!):

– Datum und Zeit der Prüfung,

– Name, Matrikelnummer, Studienrichtung, Telefonnummer des Prüflings, Note (Datenschutz!)

– die Fächerkombination,

– Name des Beisitzers

☞ eine Tabelle ?

☞ es soll vorkommen, daß ein Student eine Prüfung wiederholen muß (neues Datum, neue Uhrzeit, anderer Beisitzer)

☞ ich habe mit Studenten auch andere Kontakte (Diplomarbeit) - diese sollen also auch aufgenommen werden, wenn noch kein Prüfungstermin feststeht, von diesen sollen dann auch weitere Attribute bekannt sein (z.B. Thema / Beginn der Arbeit)

☞ Beisitzer sind (angenommen immer) Mitarbeiter von SAI, von denen ich weitere Informationen brauche (Einstellungsdatum, Vertragsdauer, Arbeitsgebiet, u.a.)

Alles in eine Tabelle???

→ sehr große Tabelle, bei der viele Zeilen in vielen Spalten keinen Eintrag haben oder sich Einträge häufig wiederholen (**Redundanz!**)

☞ Was ist, wenn mir morgen einfällt, meine Literatur mitaufzunehmen und vor allem die Tatsache, daß ich meine eigenen Bücher an Diplomanden ausleihe? (**Flexibilität!**)

→ mehrere Tabellen: Bezug zwischen den "zusammengehörenden" Zeilen in diesen verschiedenen Tabellen → **Wertgleichheit in Spalten, von Attributen** (Welche???)

Ausschnitt:

Student	MatNr	Name	usw
	111999	Meier	...
	222888	Huber	...
	333777	Schulz	...
	444666	Kuhn	...

Prüfung	Datum	Zeit	MatNr	etc
	15.5.95	10.00	111999	...
	15.5.95	11.00	444666	...
	16.5.95	10.00	222888	

- ◻ Wenn ich den Namen des Studenten wissen will, der am 15.5.95 um 10.00 Prüfung hat: gehe über die Kombination (15.5.95,10.00) in die Tabelle *Prüfung*, nimm den Wert von *MatNr* in dieser Zeile, gehe damit in die Tabelle *Student*, ...
- ◻ In der Tabelle *Student* bildet das Attribut *MatNr* einen Schlüssel, in *Prüfung* die Kombination (*Datum,Uhrzeit*)
- ◻ Das Attribut *MatNr* in der Relation *Prüfung* wird als **Fremdschlüssel** bezeichnet, es dient zur Herstellung der Beziehung von *Prüfung* zu *Student*, stellt einen "Verweis" (*link*) auf eine andere Tabelle dar.
- ◻ **2. Integritätsregel** (**Beziehungsintegrität**): Die Menge der Fremdschlüsselwerte muß stets eine Teilmenge der zugehörigen Werte in der definierenden Tabelle sein, also hier:
  Prüfung.MatNr $\subseteq$ *Student.MatNr*
  Als Mengen interpretiert, zu jedem Zeitpunkt

1.2 View’s

Was ist, wenn mit meinen Tabellen Dritte arbeiten dürfen: Sollen sie alles sehen dürfen? Sollen sie nur eine eingeschränkte Sicht auf die Tabellen einnehmen können? Sie sollen aber das, was sie sehen dürfen, ebenfalls in Tabellenform sehen:

☞ **View**’s sind spezielle Sichten auf das zugrundeliegende Gesamtschema der Tabellen, sie sind ebenfalls Tabellen, allerdings nur sog. **virtuelle Tabellen** - ihr Inhalt existiert nur zum Zeitpunkt ihrer Benutzung

☞ Einfügen, Verändern, Löschen von Tupeln in virtuellen Tabellen ist problematisch, da u.U. keine 1-1-Abbildung auf die Tupel in den Basistabellen gegeben ist!

Unterscheide:

☞ (Basis-) Tabellen, wie sie "im Rechner" abgelegt sind, verwaltet werden
Anforderungen:

- Flexibilität
  Ändern von Datensätzen, von Strukturen; Hinzufügen von neuen Tabellen; Auswertung nach unterschiedlichsten Kriterien; ...
- Redundanzfreiheit
  Wiederholtes Speichern identischer Informationen birgt die Gefahr der Integritätsverletzung (Inkonsistenz in sich, hin zur Realität)

☞ Tabellen, die dem Nutzer Informationen darstellen (i.a. spezielle Sichten)
Anforderungen:

- Übersichtlichkeit
- Kompaktheit
- u.a.m.

1.3 Sichtenintegration

In der Entstehung: erst die Gesamtsicht und dann die Detailsichten oder erst die Detailsicht und dann die Gesamtsicht???

Eine Gesamtsicht (**konzeptionelles Schema**) liegt dann vor, wenn sich aus ihr alle Detailsichten (**View**s) "ableiten" lassen.

☞ Kleine Datenbanken: nur wenige Detailsichten, daher Gesamtsicht relativ leicht zu ermitteln (s.u.), i.a. wenige Relationen

☞ Große Datenbanken (Unternehmen): Viele Detailsichten (Management, Vertrieb, Produktion, Konstruktion, Qualitätswesen, ...) - diese sind verhältnismäßig leicht zu ermitteln - daraus ist durch Integration eine Gesamtsicht, aus der wieder alle Detailsichten abzuleiten sind, zu erstellen (nicht trivial!).

1.4 Etwas Relationale Algebra

- Ableiten von Detailsichten (Abfrageoperatoren):

Sei R eine Relation mit den Attributen $A_1, A_2, ..., A_n$,

kurz: R: rel($A_1, A_2, ..., A_n$)

und R^t die Menge der Tupel der Relation R zum Zeitpunkt t.

– **Projektion** (*project*-Operator):

  Sei $\Gamma \subseteq \{A_1, A_2, ..., A_n\}$

  Dann ist die Projektion von R auf Γ eine Relation über den Attributen aus Γ, i.Z.:

  R.Γ : rel(Γ)

  $R^t.\Gamma$ ist die Menge (!) der Tupel aus R^t, in denen die Werte zu den Attributen, die nicht in Γ vorkommen, gestrichen sind (Mengeneigenschaft!)

– **Selektion** (*select*-Operator):

  Sei Γ wie oben; desweiteren sei auf dem Wertebereich von Γ eine Ordnung (Relation) Θ definiert (=, <, >, >=, ...); sei weiterhin γ ein möglicher Wert der Kombination Γ.

  Dann ist die Selektion von R bezüglich (Θ,γ) eine Relation wie R, die aber nur die Tupel $\rho \in R^t$ enthält, für die gilt:

  $(\rho,\gamma) \in \Theta$

  i.Z.: $R^t(\Theta,\gamma) = \{\rho \in R^t \mid (\rho,\gamma) \in \Theta\}$

  (Auswahl von Tupeln, die einer bestimmten Bedingung genügen)

– **Verbund** (*join*-Operator):

Seien R: rel(U), S: rel(V) zwei Relationen, U,V zwei Mengen von Attributen.

Seien $U_i \in U$, $V_j \in V$ zwei semantisch vergleichbare Attribute (Gleichheit macht Sinn) - der Einfachheit der formalen Darstellung wegen sei der Datentyp von U_i und V_j identisch.

Dann ist der Verbund von R und S bzgl. (U_i,V_j) eine Relation über $(U \cup V)-\{U_i\}$ (oder über $(U \cup V)-\{V_j\}$),

i.Z.: $R[U_i,V_j]$

mit $R^t[U_i,V_j]=\{(r.(U-\{U_i\}),s) \mid r \in R,\ s \in S \text{ und } r.U_i = s.V_j\}$

<u>Anm.:</u> Eine Verallgemeinerung der Verbunds über mehr als ein Attribut kann leicht vorgenommen werden!

Diese drei Operatoren genügen theoretisch, alle Ableitungen herzustellen!

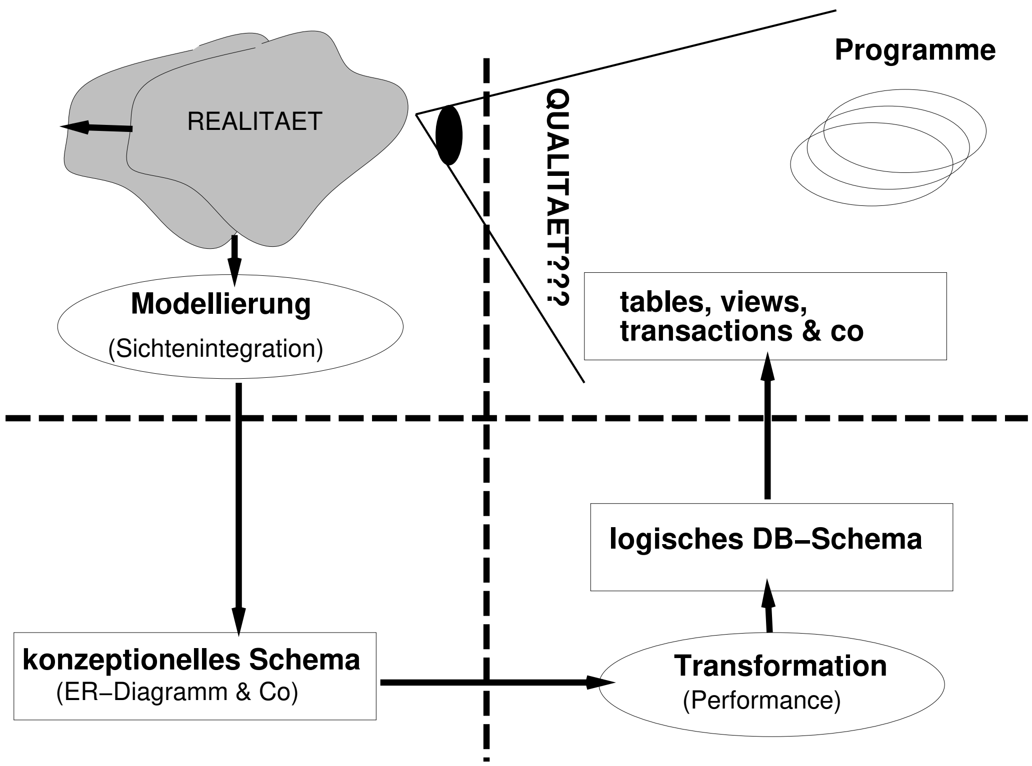
1.5 Zusammenfassung

- Die Begriffe Tabelle und Relation sind identisch - Tabellen betonen mehr den Darstellungsaspekt, Relationen mehr den mathematischen Gedanken
- Eine relationale Datenbank ist eine Menge von gleichberechtigten Tabellen
- Jede Tabelle repräsentiert eine Klasse von realen oder gedanklich konstruierten Objekten
- Tabellen werden nur über ihren Namen identifiziert
- Den Zeilen einer Tabelle entsprechen in der relationalen Sprechweise Tupel, Elemente oder Datensätze
- Jede Zeile beschreibt in abstrakter Form ein Objekt
- Die Zeilen einer Tabelle sind paarweise verschieden
- Die Reihenfolge, in der die Zeilen in der Tabelle stehen, sagt nichts über die Bedeutung der Zeile

- Die Spalten einer Tabelle definieren Attribute oder Merkmale der Objekte
- Die Reihenfolge der Spalten spielt keine Rolle
- Die Spalten besitzen einen lokal eindeutigen Namen (Attributbezeichner)
- Jede Spalte besitzt einen Datentyp (Wertebereich)
- Zugriff und Auswahl von Tabelleninhalten erfolgen ausschließlich über logische Bedingungen (Prädikate), die mit Tabellen- / Attributnamen und Attributwerten definiert werden
- Schlüssel sind Attribute oder Attributkombinationen - ein Schlüssel ist identifizierend und minimal
- Jede Tabelle besitzt mit dem sog. Primärschlüssel (*primary key*) einen besonders ausgezeichneten Schlüssel
- Fremdschlüssel sind Verweise (*links*) auf Objekte in anderen Tabellen und repräsentieren eine reale oder gedanklich konstruierte Beziehung zwischen Objekten (Objektklassen)
- Fremdschlüssel sind nicht notwendig Schlüssel im Sinne von identifizierend und minimal
- Es gibt Tabellen, die Objektklassen repräsentieren, und Tabellen, die Beziehungen realisieren (Kett-Tabellen, Kett-Relationen)
- Objekt- und Beziehungsintegrität sind elementare Integritätsbedingungen
- Die Verwendung restriktiver Wertebereiche für Attribute unterstützt die Sicherung der Integrität
- Semantische Integritätsbedingungen definieren zusätzliche Anforderungen an den Datenbestand
- Relationale Datenbanken betrachten nur Tabellen in 1. Normalform: alle Attribute sind elementar

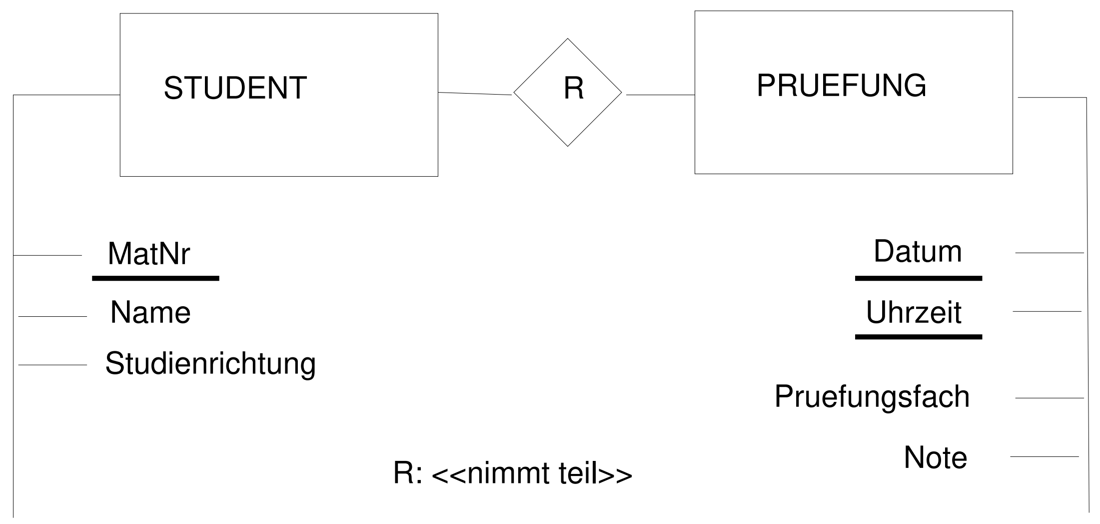
2. ER-Diagramme

- Grafische Notation zur abstrakten Datenmodellierung, losgelöst von Realisierungsdenken (kein Denken in Tabellen, Umsetzung in Tabellen s.u.)
- Von der Realität zur Menge der Basis-Tabellen:



Modellierungselemente:

- ◻ Rechtecke bezeichnen Objektklassen (Objekt = Entität, Wesenseinheit, *entity*)
- ◻ Vebindungen zwischen Rechtecken über eine Raute bezeichnen Beziehungen zwischen Objektklassen (Beziehung zwischen Objektklassen = Relation zwischen Objektklassen, *relationship*)
- ◻ *Entity-Relationship*-Diagramme



- Unterstrichene Attribute stellen zusammen einen Schlüssel dar
- Die Beziehung R kann aufgefaßt werden als

  $R \subseteq STUDENT \times PRUEFUNG$

2.1 Woher kommen die Objektklassen?

- Antwort auf die Frage: WORÜBER werden Informationen benötigt?
- Welche "realen", gedanklich abgrenzbaren Objekte sind von Interesse?
- Wie lassen sich diese beschreiben (Attribute, Abstraktion)?
- Lassen sich über den erfaßten Objekten Klassen bilden ("Äquivalenzrelation")?
- Wie ist die Beziehung zwischen definierten Objektklassen (siehe Klassifikation von Beziehungen)? Oftmals ist das, was man als Klasse definiert, bei genauer Betrachtung besser als Beziehung modellierbar: die PRUEFUNG ist u.U. als Beziehung zwischen STUDENT, BEISITZER, PROFESSOR und PRUEFUNGSFACH besser modelliert (was heißt schon "besser"?)
- Tja, Modellieren ist eine "Kunst" → Erfahrung, Weitblick, Intuition, ...

2.2 Funktionale Abhängigkeiten, Normalformen

Normalformen helfen bei der Modellierung, aber auch bei der Festlegung von Tabellen (s.u.)

Formale Grundlage:

Funktionale Abhängigkeit:

Sei R: rel(U) gegeben, seien A, B $\subseteq$ U (Attributkombinationen);

Dann heißt **B funktional abhängig von A**, i.Z. $A \rightarrow B$, g.d.,w. gilt:

> zu jedem Zeitpunkt t und für je zwei $r_1, r_2 \in R^t$ mit $r_1.A=r_2.A$
>
> ist auch $r_1.B=r_2.B$
>
> (A-Wert bestimmt B-Wert)

B heißt **voll funktional abhängig** von A, i.Z. $A \overset{\bullet}{\rightarrow} B$ g.d., w. gilt:

> für alle A´ mit $\emptyset \neq A' \subset A$: $A' \not\rightarrow B$

Definition eines Schlüssels:

$S \subseteq U$ heißt Schlüssel, g.d., w. gilt: $S \overset{\bullet}{\rightarrow} U$

Attribute, die in einem Schlüssel vorkommen, heißen **Schlüsselattribute** (identifizierende Attribute), alle anderen **Nicht-Schlüsselattribute** (beschreibende Attribute)

Es gilt (A, B, C $\subseteq$ U):

- $A \subseteq B \Rightarrow B \rightarrow A$
- $A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$
- $A \rightarrow B \Rightarrow A \cup C \rightarrow B \cup C$

Funktionale Abhängigkeiten kommen wie die Attribute aus der jeweiligen Realität! Sie stellen, falls gegeben, ebenfalls Integritätsbedingungen dar!

2. Normalform (2NF)

- ◻ Die Attribute einer Objektklasse sollen sich auf das Objekt als Ganzes (identifiziert durch den / die Schlüssel) und nicht auf Teile davon (auf Teile eines Schlüssels) beziehen

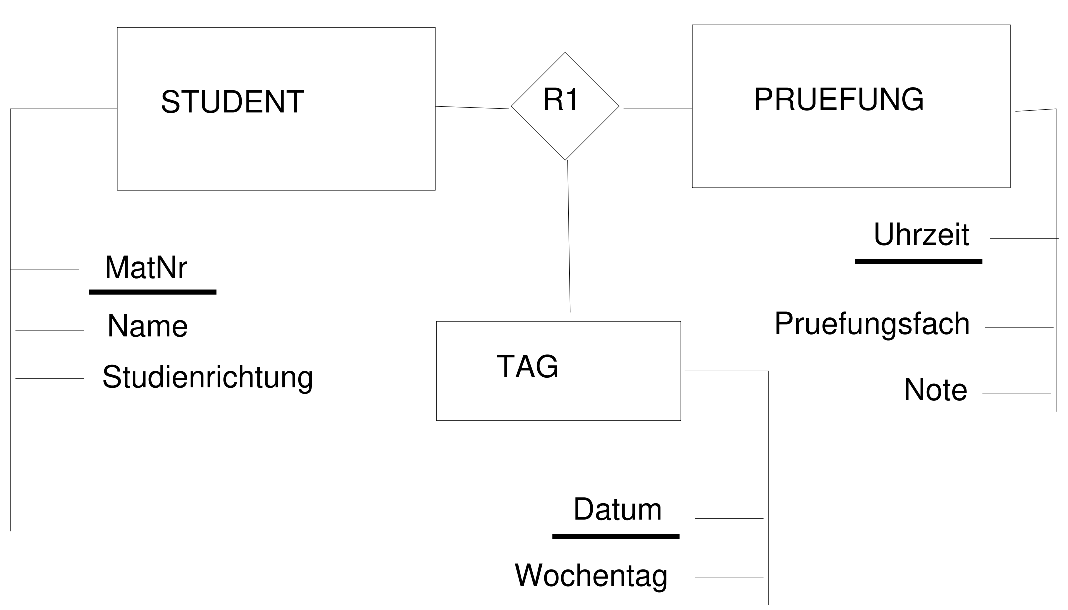
Formal:

- ◻ Für alle Nicht-Schlüsselattribute NS $\in$ U und alle Schlüssel S gilt:
  $S \twoheadrightarrow NS$

Beispiel: Würden wir bei der PRUEFUNG neben *Datum, Uhrzeit* als weiteres Attribut auch den *Wochentag* ablegen, so wäre die 2. Normalform verletzt, denn
DATUM $\rightarrow$ WOCHENTAG, d.h. WOCHENTAG ist nicht voll funktional abhängig vom Schlüssel (DATUM, UHRZEIT)!

Wo ist das Problem?

- Der Benutzer soll ruhig den Wochentag bei Datum und Uhrzeit sehen!
- Soll aber der Wochentag wirklich bei Datum und Uhrzeit gehalten werden? $\rightarrow$ **Redundanz** (Gefahr von Integritätsverletzungen); dieses Beispiel scheint eher trivial, da sich der Wochentag schließlich aus dem Datum errechnen läßt und daher sowieso nicht in der Datenbank abgelegt werden sollte; in anderen Fällen trifft dies nicht zu!
- Ein Tag ist ein eigenes Objekt, mit Schlüssel DATUM und Attribut WOCHENTAG vom Typ (Montag, Dienstag, Mittwoch, ...) und sollte dann auch so modelliert werden:



Transitive Abhängigkeit:

Eine Attributkombination $B \subseteq U$ heißt transitiv abhängig von einer Attributkombination $A \subseteq U$, i.Z. $A \gtrdot B$, g.d., w. gilt:

es gibt ein $C \subseteq U$: $C \neq \emptyset$, $C \not\supseteq B$ und $A \to C \to B$ und $C \not\to A$.

3. Normalform (3NF)

- □ Alle Nicht-Schlüsselattribute machen eine unmittelbare (direkte) und nicht nur eine mittelbare (indirekte) Aussage über das zu beschreibende Objekt!

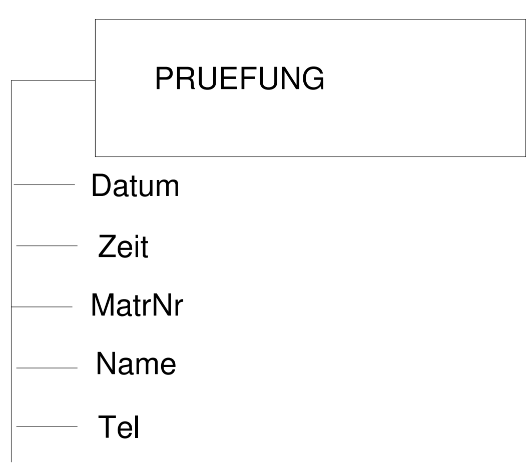
Formal:

- □ Für alle Nicht-Schlüsselattribute $NS \in U$ und für alle Schlüssel $S \subseteq U$ gilt:
  $S \not\gtrdot NS$

Es gilt:

- □ 3NF => 2NF

Beispiel: Wir könnten uns bei der Modellierung der Prüfung nur mit einem einzigen Objekttyp begnügen:



Hier gilt:

- (Datum,Zeit) ist einziger Schlüssel

- 2. Normalform gelte als erfüllt, d.h. es kann z.B. vorkommen (meine "Realität), daß ein Student an einem Tag zwei Prüfungen bei mir macht. Wenn dies ausgeschlossen wäre, so wäre die 2. Normalform verletzt, denn es würde dann gelten:
  Datum $\rightarrow$ MatNr, d.h. (Datum, Zeit) $\bullet\!\!\!\rightarrow$ (MatNr wäre verletzt!
  Aber auch im ersten Fall (2NF als erfüllt vorausgesetzt) ist die 3NF wohl (in meiner Sicht der Realität) verletzt:
  Es gilt:
  (Datum,Zeit) $\rightarrow$ MatrNr $\rightarrow$ Name, MatrNr $\not\rightarrow$ (Datum,Zeit)
  { MatrNr } $\neq \emptyset$, { MatrNr } $\not\supseteq$ { Name }

Wo ist das Problem?

☞ Wie oben: Redundanz!

☞ Das, was "transitiv abhängt", ist vermutlich immer ein eigenes Objekt und sollte auch so modelliert werden!

Arten von Attributen einer Objektklasse

- rein beschreibend Attribute
- identifizierende Attribute

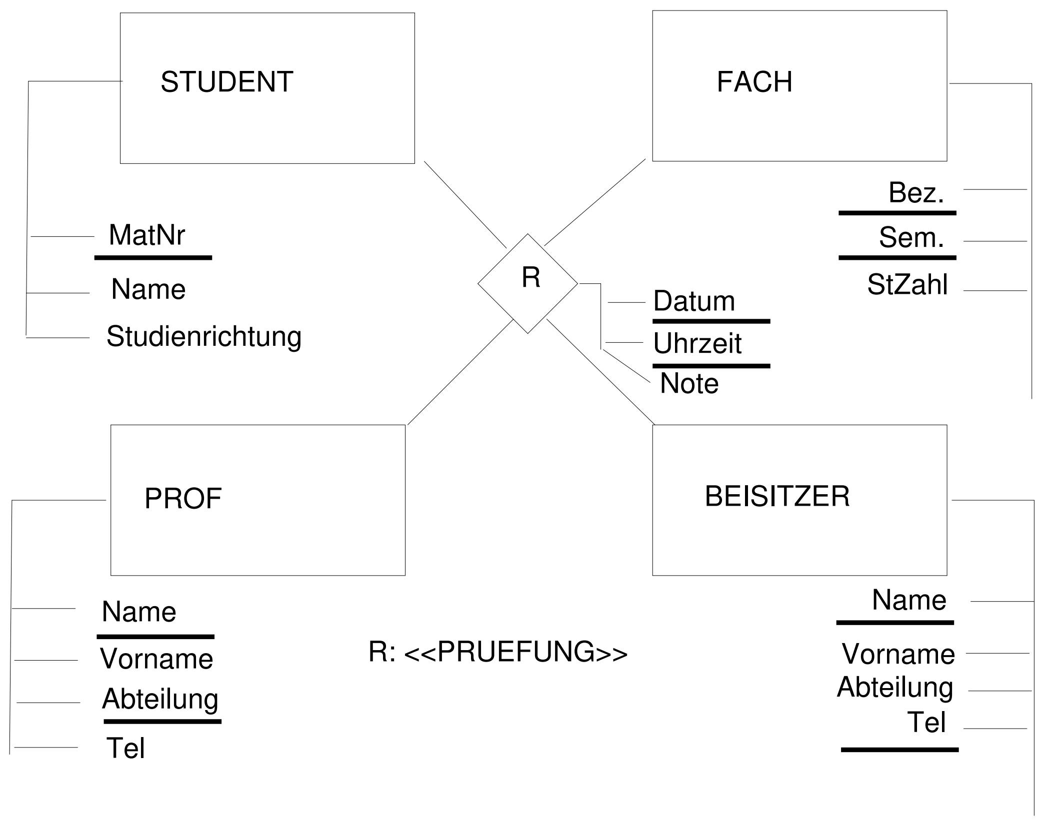
2.3 Klassifikation von Beziehungen

❶ Stelligkeit (Grad der Beziehung):
Die Anzahl der an einer Beziehung beteiligten Objektklassen (oben: R ist zweistellig, grad(R)=2; R1 ist dreistellig, grad(R1)=3)

Anm.:

- Bei mehrstelligen Beziehung ist es i.a. schwer, einen passenden Namen für die Beziehung zu finden. Bei zweistelligen Beziehungen drückt der Name meist eine "Richtung" der Beziehung auf ("ist_Mitglied_von").
- Mehrstellige Beziehungen können oft über 2-stellige Beziehungen aufgelöst werden - dies geht aber nicht immer ohne Informationsverlust!
- In der Praxis treten meist 2-stellige Beziehungen auf, doch machen mehrstellige Beziehungen durchaus Sinn:

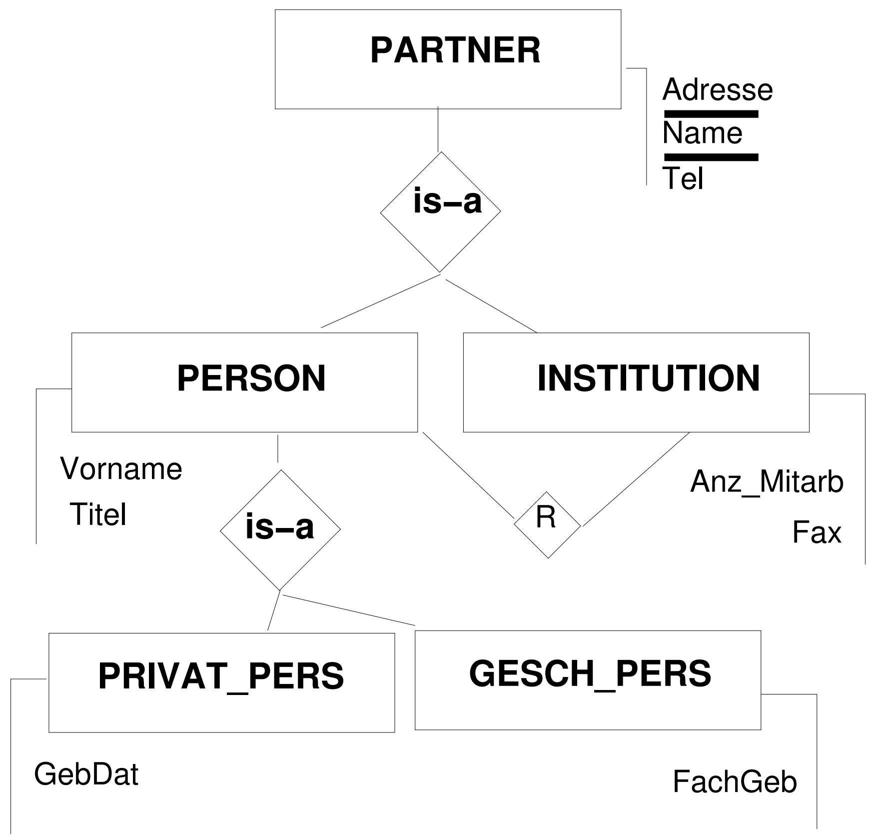
  Eine Prüfung kann betrachtet werden als eine Beziehung zwischen einem Studenten, dem Prof, dem Beisitzer und einem Prüfungsfach:



❷ Beziehungen mit Attributen
siehe oben!

❸ *is-a*-Beziehungen (Generalisierung, Spezialisierung)

Hinweis: in Modula-2: Datentyp *variant record*, in C: *union*



R: <<Mitglied von>>

Dieses Diagramm läßt bezgl. Interpretation viele Fragen offen!

Anm.: Hinter der *is-a*-Beziehung steckt das Konzept der Typ-Erweiterung in OO-Sprachen:

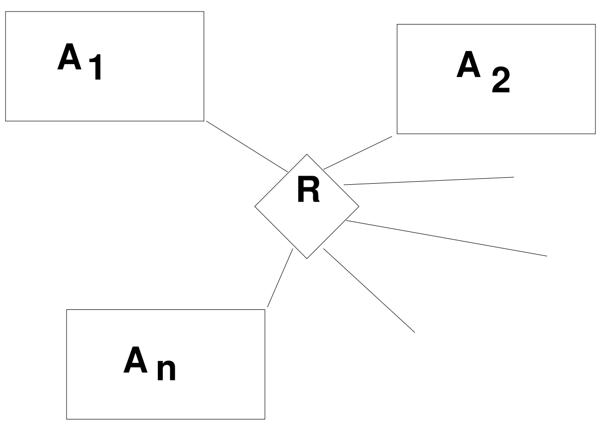
```
TYPE Partner = RECORD
                Adresse: (*Typ*);
                Name: (*Typ*);
                Tel: (*Typ*);
               END;

TYPE Person = RECORD (Partner)
                Vorname: (*Typ*);
                Titel: (*Typ*);
              END;

TYPE PrivatPers = RECORD (Person)
                    GebDat: (*Typ*);
                  END;
```

❹ *is-part-of*-Beziehung ("besteht aus")
z.B.: Stücklisten, Konfigurationslisten

❺ rekursive Beziehung
Objektklasse steht mit sich selbst in Beziehung

❻ Komplexitätsgrade:
wie oft nimmt ein Objekt an einer Beziehung mindestens bzw. höchstens teil?

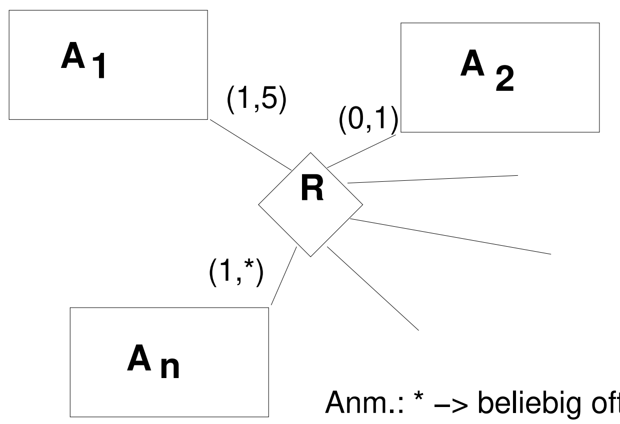
Sei $R \subseteq A_1 \times A_2 \times \bullet\bullet\bullet \times A_n$ eine n-stellige Beziehung,



für $A \in \{A_1, A_2, \bullet\bullet\bullet, A_n\}$ ist:

$\text{kgrad}(A, R) := (\min_{a \in A}(|\{r \in R^t | r.A = a\}|), \max_{a \in A}(|\{r \in R^t | r.A = a\}|))$

Notation im ER-Diagramm beim jeweiligen Objekt als $(\min_i, \max_i)$



verkürzte (1,c,m,mc) Notation:

$kgrad_o(A_i, R)$	$kgrad(A_i, R)$
1	(1,1)
c	(0,1)
m	(1,*)
mc	(0,*)

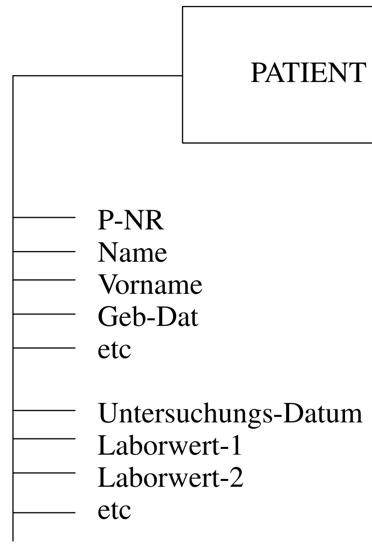
Anm.: Komplexitätsgrade stellen ebenfalls Integritätsbedingungen dar!

2.4 Dependent / Independent Enities

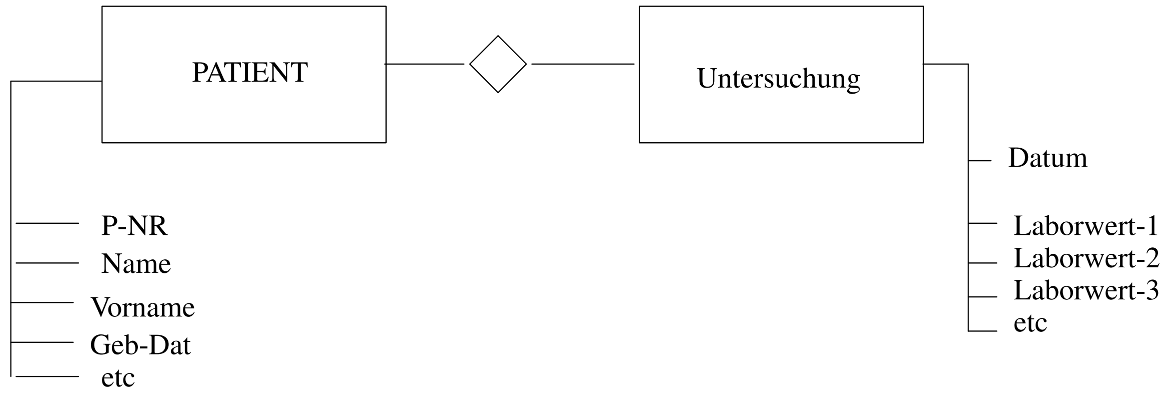
Beispiel:

Für ein Informationssystem in einer Klinik ist ein zentraler Entity-Typ der ‘‘Patient’’, beschrieben durch eine Patientennummer (Schlüssel) und durch weitere Attribute (die sog. Stammdaten). Kommt ein Patient zu ambulanten Untersuchungen, fallen pro Untersuchung verschiedene Daten an. Ein solche Untersuchung kann als eigener Entity-Typ betrachtet werden, der allerdings nur existieren kann, wenn der zugehörige Patient als Entity existiert:

Modellierung als **ein** Enity-Typ:

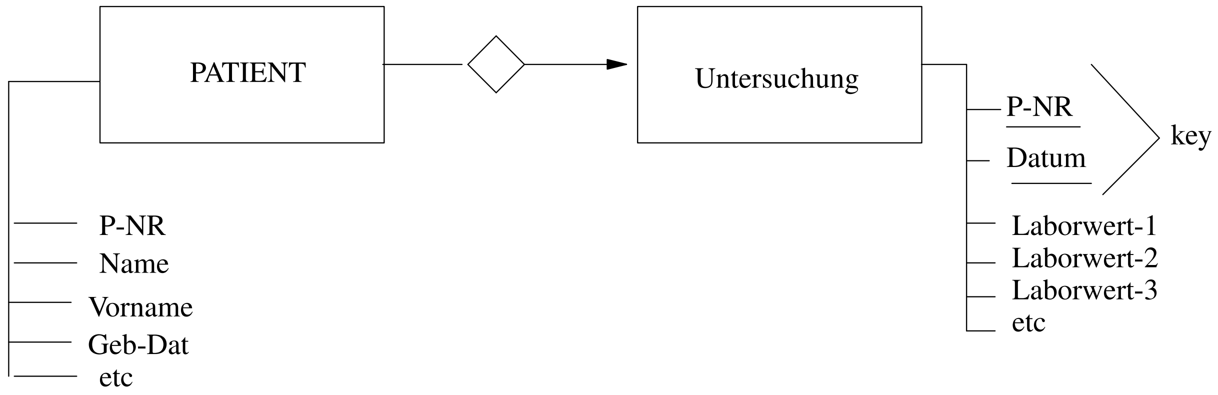


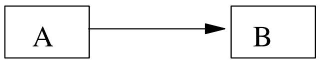
Alternativ:



Diese Modellierung drückt die Existenzabhängigkeit nicht aus. Ausserdem stellt sich die Frage nach dem Schlüssel von ‘‘Untersuchung’’.

Besser:

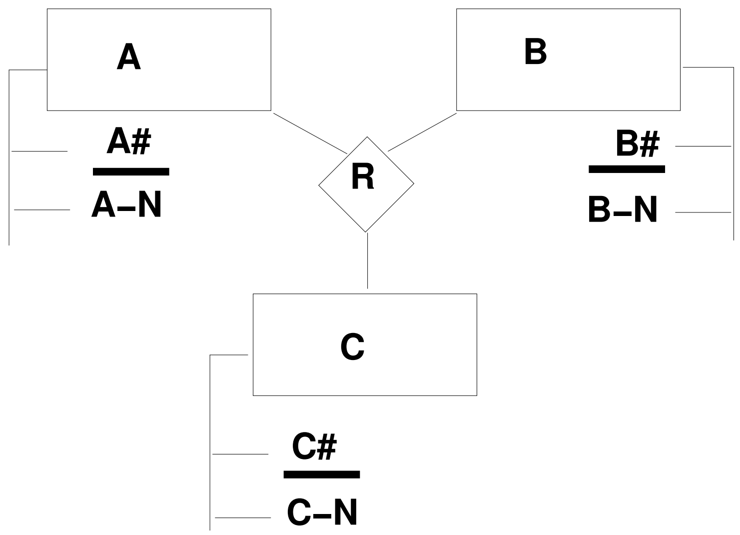


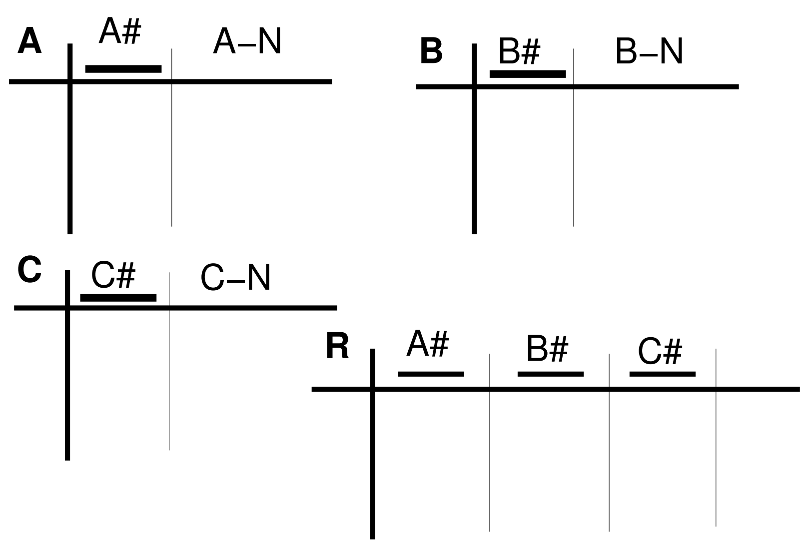


B haengt von A ab

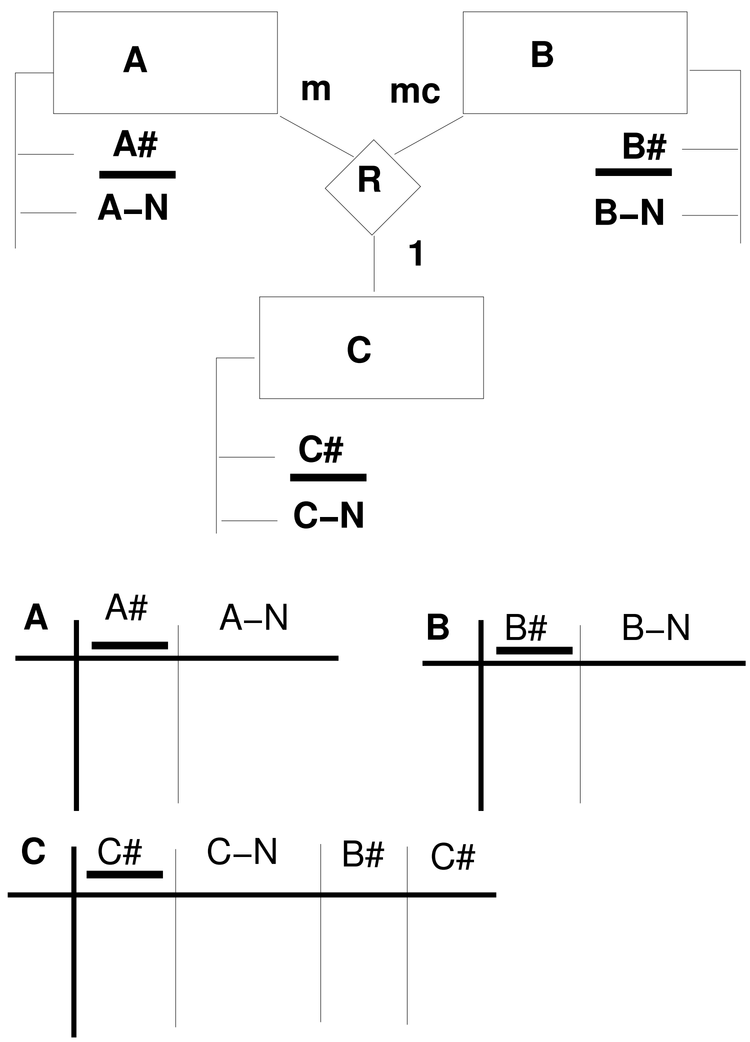
2.5 Umsetzung in Tabellen

- Objektklassen werden Tabellen (PERFORMANCE???)
- Beziehungen können i.P. immer als eigene Tabellen realisiert werden, in denen die Primärschlüssel der beteiligten Objektklassen als Attribute eingetragen sind:





☞ Ist allerdings der Komplexitätsgrad $kgrad_o$ einer der beteiligten Objektklassen **1**, so kann die Beziehung auch durch Aufnahme der Primärschlüssel der anderen Objektklassen in eben diese Tabelle realisiert werden:



Frage:

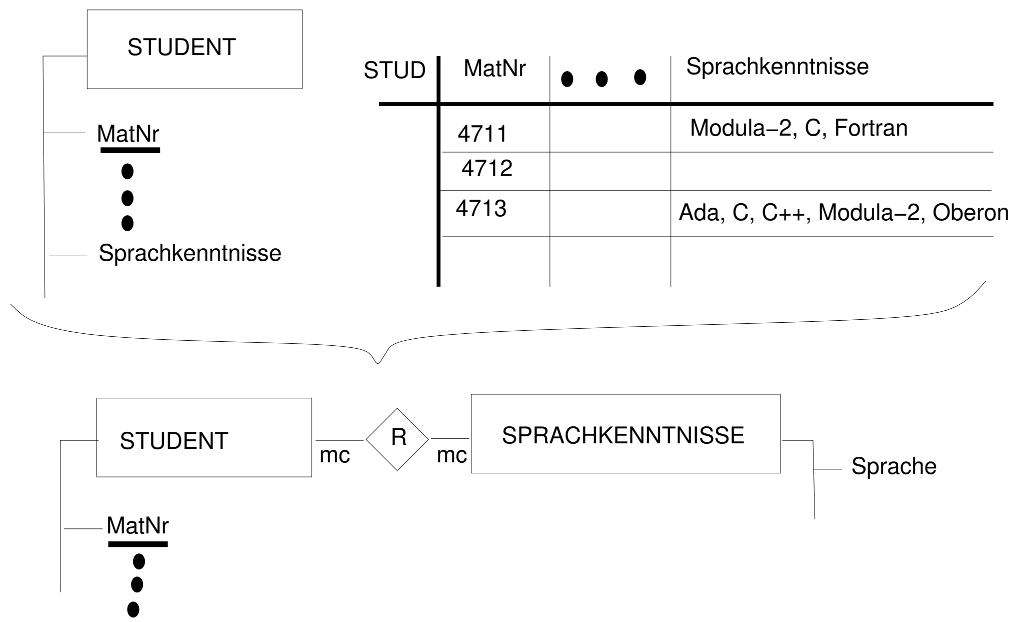
Warum sollte man das in anderen Fällen nicht tun?
(Hinweis: Normalformen!)

□ *is-a*-Beziehungen:

Alle Unter-Klassen werden in die Tabelle der Ober-Klasse "gezogen"?

Unterklassen wie Oberklasse werden je eine Tabelle, ein gemeinsamer Primärschlüssel, mit einem *tag-field* wird der Weg von der Oberklasse in die Unterklasse "gewiesen"?

□ strukturierte Attribute (*non-first-normalform*, NFNF, NF^2)



2.6 Umsetzung in UNIX-Text-Dateien

□ Tabelle → Datei (Folge von Zeichen!)

□ Dateiname = Tabellenname

Aber:

- Wie sollen die Spalten voneinander unterschieden (getrennt) werden?
  z.B. */etc/passwd* → Doppelpunkt
  Ein Zeichen, das nie innerhalb eines Attributwertes vorkommt; Kandidaten: der "Lattenzaun" (**#** - Vorsicht: Kommentarzeichen), das "Dächchen" (**^**), das *pipe*-Symbol (**|**), u.a.

- Wie werden die Tupel voneinander getrennt? Kandidaten: das *newline* (**\n**), eine Leerzeile (zwei aufeinanderfolgende *newline*s)

- Wie erfolgt der Bezug von Attribut- resp. Spaltennamen zu den zugehörigen Werten?

① Die erste Zeile der Datei (Kopfzeile) enthält die Spaltennamen (gleicher Trenner wie für die Attributwerte eines Tupels

② Die Spaltennamen kommen in eine *header*-Datei, die Datensätze liegen in der Datei mit Endung **.dat**, die Kopfzeile liegt in der enstprechenden Datei mit Endung **.attr**

③ In einer Datei liegt die Beschreibung der gesamten Datenbank *Data Dictionary*) in einer bestimmten Form (Sprache!), z.B.:

```
Datei1
        Attr1
        Attr2
        Attr3
        u.s.w.
Datei2
        Attr1
        Attr2
        etc
View1
        attr1
        attr2
        u.s.f.
        !Erzeugungskommando1
        !Erzeugungskommando2
```

In EBNF:

```
DataDictionary = TableDef { TableDef } { ViewDef }
TableDef = TableName "\n" AttrDef
AttrDef =  WhiteSpace AttrName "\n" { WhiteSpace AttrName "\n" }
ViewDef = TableDef Command { Command }
Command = WhiteSpace "!" CommandName "\n"
WhiteSpace = (" " | "\t") { (" "|"\t") }
AttrName = letter { (letter | digit) }
```

TableName : zulässiger UNIX-Datename
CommandName : ausführbares UNIX-Kommando

2.7 Zusammenfassung

- Im Rahmen der Informationssystementwicklung wird zunächst das konzeptionelle Datenmodell entwickelt

- Das konzeptionelle Datenmodell ist abstrakt und berücksichtigt keine EDV- oder gar implementierungstechnischen Details
- ER-Modelle sind spezielle konzeptionelle Datenmodelle, die sich mit Objektklassen und ihren Beziehungen untereinander beschäftigen
- Entity-Typen beschreiben Objekte (Entities) gleicher Struktur (und "Natur")
- Die Struktur wird durch Attribute (Eigenschaften) festgelegt
- Formal können wir ein Entity als Element des Kreuzprodukts der Attributwertebereiche auffassen
- Beziehungen (Beziehungstypen) können durch Grad, Art (z.B. *is-a*) oder Komplexitätsgrad charakterisiert werden; sie können selbst Attribute haben
- Formal kann eine Beziehung (Beziehungstyp) als Kreuzprodukt über den beteiligten Entity-Typen und über ihre eigenen Attribute aufgefaßt werden
- Komplexitätsgrade sind spezielle Integritätsbedingungen und helfen bei der Umsetzung in Tabellen
- Datenmodellierung ist subjektiv!
- Zu einem vollständigen und umfassenden ER-Modell gehört die Definition von Entity-Typen, Beziehungs-Typen, Attributen, Wertebereichen/Datentypen, Komplexitätsgraden, weiteren semantischen Integritätsbedingungen, zulässigen Operationen, Festlegung von Transaktionen, Festlegung der Reaktion auf Integritätsverletzungen (u.U. noch mehr)

3. Beispiel-Tabellen

Partner						
Name	Vorname	Institution	Straße	PLZ	Wohnort	Telefon
Bauer	Karl					07303/3347
Spillner	Dr. Andreas	Uni	PF 330 440	W-2800	Bremen 33	0421/2182449
Zanger	Dr. Cornelia	TU	Mommsenstr. 13	OD-8027	Dresden	
Pesch	Dr. Herbert		Arabellastr. 5/917	81925	München	089/9102286
Koelsch	Dr. Raimund		Perlacher Str. 21	8000	München 90	089/65107120
Sonnig	Elmar		Baumreute 10	7257	Ditzingen	07152/56203
Wallmueller	Ernest	ATAG AG	Kanalstr. 31	CH-8152	Glattbrugg	01-810-9909
Wilken	Ernst		Hoervelsinger Weg 25	D-89081	Ulm	0731/9650-0
Burr	Barbara	URZ Stg	Allmandring 30	7000	Stuttgart 80	0711/1319203
Schmid	Werner	GPS	Hoervelsinger Weg	7900	Ulm	0731/966570
		Ciba-Geigy	R-1008.8.05	CH-4002	Basel	+41-61/6973419
		IBM Ulm				1515-0
Schreiner	Prof.Dr. A.	Uni	Albrechtstr. 28	D-4500	Osnabrueck	0541/969-2483
Krallmann	Prof.	TU Berlin				030/31473261

PERSON								
PersId	Name	Vorname	Anrede	Institution	Strasse	Plz	Ort	Tel
4711	Biertaler	Peter J.	Herr	BS&S	Usambaraweg 11	9876	Glockendorf	0815/10007
4712	Oha	Alfred V.	Herr	BS&S	Usambaraweg 11	9876	Glockendorf	0815/10008
4713	Koernigen	Brian W	Herr	BS&S	Usambaraweg 11	9876	Glockendorf	0815/10008
4714	Reichle	Dennis M	Herr	BS&S	Usambaraweg 11	9876	Glockendorf	0815/10009
4715	Stanglow	Werner	Herr	Uni Ulm	Helmholtzstr. 18	7900	Ulm	0731/5023572
4716	Rahn	Uwe	Herr Prof.	FH Nixstadt	Bitstr. 7	9999	Nixstadt	0999/9999
4717	Ross	Detlev	Herr	HalComp Inc.	Ringstr. 1	7000	Stuttgart 1	0711/123456
4718	Schweiggert	Franz	Herr Prof. Dr.	Uni Ulm	Helmholtzstr. 18	7900	Ulm	0731/5023570
4719	Ruebe	Berta	Frau Dr.	XSoft AG	Hahnweg 13	8000	Muenchen 21	
4720	Blau	Bernd	Herr	SPG GmbH	Am Bach 3	3000	Hannover 3	0511/546-1234
4721	Rot	Ulf	Herr		Sieberweg 15	7030	Boeblingen	07031/98765
4722	Winter	Ralf	Herr Dr.	Xplus GmbH	Eschbuehel 2	5100	Aachen	0241/4098861
4723	Maier	Urs	Herr	CoreDump AG	Steinstr. 3	2800	Bremen 33	0421/442812
4724	Meyer	Ute	Frau Dr.med.	MedSoft GmbH	Seehof 5	2000	Hamburg 1	040/606320
4725	Meier	Ulf	Herr		Kohlring 3	2000	Hamburg 4	
4726	Bedchaus	Karin	Frau	Steuerbuero Maus	Telefonstr. 1	6500	Mainz	06131/54321
4727	Abs	Karl	Herr	Softikus AG	Auweg 3	7918	Au	07303/5857
4728	Stange	Inge	Frau Prof. Dr.	Techn. Uni.	Postfach 1240	3010	Magdeburg	
4729	Enter	Jo	Herr Dr.	ElektroBank AG	Postfach 131313	8000	Muenchen 60	
4730	Emu	F.J.	Herr Dr.h.c.	SSS e.V.	Straussring 3	3000	Hannover 1	0511/4321645
4731	Streit	Max	Herr		Eselsberg 4	7900	Ulm	
4732	Dann	Else	Frau		Bugstr. 3	4500	Osnabrueck	0541/7720969
4733	List	C.	Herr	CRG mbH	Pascalweg 1	2800	Bremen 1	0421/244218
4734	Mai	Karl	Herr Dr.		Finkenweg 19	7900	Ulm	0731/12534
4735	Maerz	August	Herr	Maerz&Partner	Einsteinweg 18	7900	Ulm	0731/523571
4736	Zank	Werner	Herr Dr.	Softikus AG	Auweg 3	7918	Au	0731/5857
4737	Schon	Gerda	Frau	Softness GmbH	Einsteinring	8000	Muenchen 20	089/980321
4738	Mair	Josef	Herr	IEJ VaG	Virenweg 17	8500	Nuernberg 80	0911/5112323

BEITRAEGE			
BeitrId	Titel	Eingang	Ok
1201	UNIX in der Kommunalverwaltung	12.1.93	+
1202	/Qbx - ein interaktives grafisches Abfrage-Tool	18.12.92	+
1203	CASE und die klassischen UNIX-Tools	01.01.93	+
1204	ER-Design unter X11 und OSF/Motif	3.2.93	+
1205	Verteilte Datenbanken in einer homogen UNIX-Umgebung	12.2.93	-
1206	Die Shell - eine 4GL?	13.2.93	+
1207	COBOL oder C - Ein Erfahrungsbericht	15.1.93	-
1208	FORTRAN2000 - Ende oder neuer Anfang?	13.1.93	-
1209	OBERON vs C++	12.2.93	+
1210	UNIX - und die Software wird besser?	14.1.93	-
1211	Evolutionaere Software-Entwicklung mit UNIX-Tools	10.1.93	+
1212	Was kommt nach UNIX? Ein Blick ins Jahr 2000	16.1.93	-
1213	Sanierung grosser C-Programme	21.1.93	-
1214	Von C zu C++ - wirklich ein Fortschritt?	8.1.93	-
1215	Objektorientiertes Datenmanagement	12.1.93	+
1234	Awk - Really awkward?	1.02.93	-
1235	UNIX - Wohin geht die Reise?	24.12.92	-
1237	C - Ein moderner Makro-Assembler	30.02.93	-
1236	Die Shell - Perle oder Miesmuschel?	1.04.93	-

Autoren	
PersId	BeitrId
4711	1234
4713	1234
4713	1235
4712	1234
4715	1236
4714	1237
4715	1215
4718	1210
4726	1201
4729	1201
4713	1202
4724	1203
4728	1204
4728	1205
4730	1206
4726	1207
4727	1207
4723	1208
4728	1209
4721	1211
4722	1211
4721	1212
4719	1213
4720	1213
4717	1214

AKTIVITAETEN				
AktId	Beschreibung	Datum	Status	PersId
3501	Schwerpunkte definieren, Termin und Ort fuer Tagung festlegen	15.10.92	ok	4718
3502	Termin,Raum fuer 1.Komitee-Sitzung festlegen	17.10.92	ok	4718
3503	Protokoll der 1. Komitee-Sitzung verschicken	30.10.92	ok	4700
3504	Entwurf des Call for Papers (CfP) erstellen	5.11.92	ok	4717
3505	Entwurf an Komitee-Mitgl., u.A.w.g bis 25.11.92	6.11.92	ok	4700
3506	Aenderungen einarbeiten und CfP fertigstellen	27.11.92	ok	4718
3507	Druckauftrag fuer CfP erteilen	29.11.92		
3508	CfP versenden an Verteiler X	11.12.92		
3509	Eingegangene Beitraege sortieren	5.3.93		
3510	Wertungstabelle erstellen	11.3.93		
3511	Einladung zur Komitee-Sitzung	12.3.93		
3512	Annahme-/Ablehnungsbriefe versenden	20.3.93		
3513	Programm erstellen	22.3.93		
3514	Programm zum Druck geben	24.3.93		
3515	Programm versenden	30.3.93		
3516	Anmeldungsliste erstellen und stetig fuehren			
3517	Tagungsbeitraege zum Tagungsband zusammenstellen	30.4.93		
3518	Tagungsband drucken	30.5.93		
3519	Anmeldebestaetigungen versenden	15.6.93		
3520	Namensschilder fuer Referenten, Teilnehmer u. Kom.Mitgl.	15.6.93		
3521	Tagungsmappen vorbereiten	15.6.93		
3522	Raum vorbereiten	17.6.93		
3523	Pausenimbiss bestellen	17.6.93		

LIEFERANTEN				
LiefId	Name	Strasse	Plz	Ort
7900	Papier Schneider	Neue Gasse 72	7900	Ulm
7901	Druckerei Freiberg	Ulmenstrasse 17	7910	Neu-Ulm
7902	Buchhandlung Fiebig	Einsteinstrasse 13	7900	Ulm
7903	Buecher Hofheimer	Hirschweg 12	7900	Ulm
7904	Party Service	Augsburger Str. 1	7950	Biberach
7905	H&S-Shop	Egonstr. 134	7000	Stuttgart 10
7906	Computer Shop	Birkenweg 12	7900	Ulm
7907	Verlagshaus Freitag	Einstein-Weg 34	8000	Muenchen 81
7908	Schwabenblatt	Hauptstr. 23	7918	Illertissen
7909	Falken Buerobedarf	Kirchplatz 2	7913	Senden

Vorab etwas zur Shell

1. Welche Shell?

Vielleicht gehören auch Sie zu den Leuten, deren erster Weg morgens zu Ihrem Unix-Rechner führt, um sich einzuloggen. Sie geben dazu ihren Login-Namen und Ihr geheimes Paßwort ein und warten dann darauf, daß Ihr Rechner Sie mit dem Prompt[1] zur Eingabe von Kommandos auffordert:[2]

```
login: stanglow
Password:
laborix$
```

Tatsächlich ist es nicht Ihr Rechner, der mit Ihnen kommuniziert, sondern die Shell.Die Shell, besser eine Shell (siehe Vorwort), finden Sie auf allen UNIX-System als Kommandointerpreter. Darunter verstehen wir ein Programm, das Befehle von Ihnen entgegen nimmt, sie nach gewissen Regeln interpretiert und daraufhin Aktionen des Betriebssystems veranlaßt. Da auf vielen Rechnern mehrere Shells unterstützt werden, von denen Ihr Systemadministrator eine als Voreinstellung für Sie ausgewählt hat, sollten sie zunächst sicherstellen, daß Sie die gleiche Shell verwenden wie wir, denn sicherlich beabsichtigen Sie, die Beispiele in diesem Skript sofort praktisch am Rechner nachzuvollziehen.

```
laborix$ echo $0
ksh5
laborix$ exec /bin/sh
laborix$ SHELL=/bin/sh
laborix$ export SHELL
laborix$ echo $0
/bin/sh
laborix$ echo $SHELL
/bin/sh
laborix$
```

1) Hier ist der Prompt *laborix$*, also *Rechnername*$. Häufig wird auch nur *$* ausgegeben.
2) Ausgaben des Rechners sind in alle Beispielen in `Courier`, Benutzereingaben zusätzlich **`fett`** gesetzt.

$0	ist eine Variable, die den Namen enthält, unter dem eine Shell oder ein Shellskript aufgerufen wurde
echo	ist ein Programm, daß seine Argumente am Bildschirm ausgibt. Vor der Ausführung jedes Kommandos interpretiert die Shell jedoch erst einmal die gesamte Eingabzeile. Sie ersetzt dabei Audrücke wie *$0* oder *$SHELL* durch den Wert der Variablen
=	weist einen Text (auf der rechten Seite) an eine Shellvariable (auf der linken Seite) zu - Achtung: keine Blanks vor / nachdem =!
SHELL	ist eine Shellvariable, die manche Programm benutzen, um weitere Shells zu erzeugen
export	sorgt dafür, daß die angegebenen Variablen auch in den sogenannten *Subshells* bekannt werden. Darunter sind alle Shells zu verstehen, die von der aktuellen Shell als zusätzliches Programm (neuer Prozeß, s.u.) gestartet werden
exec	ersetzt die aktive Shell (*ksh5*) durch das als Argument angegebene Programm (*/bin/sh*)
IFS	Shell-Variable (*internal field separator*) - enthält das / die Zeichen, das / die als Worttrenner fungiert / fungieren; oft nur das Blank; der *tab* hat oft eine Sonderbedeutung *Datenamen-Expansion)

Sie arbeiten nun mit der sogenannten *Bourne-Shell,* die Sie ziemlich sicher auf jedem UNIX-System als */bin/sh* finden. Soweit es reine Shell-Konstrukte betrifft, sollten die folgenden Beispiele deshalb auch auf Ihrem Rechner funktionieren. Dagegen kann es vorkommen, daß Sie das eine oder anderere Programm auf Ihrem Rechner vermissen oder dort andere Optionen unterstützt werden, als wir hier verwenden. Ziemlich sicher[3)] wird auch die Ausgabe des einen oder anderen Kommandos von den gezeigten Beispielen abweichen oder. Aufschluß erhalten Sie im Zweifel durch Ausprobieren oder vom Manual, das die auf Ihrem Rechner verfügbaren Kommandos und ihre Handhabung erläutert und ein untentbehrliches Hilfmittel für Ihre Arbeit mit UNIX ist. Selbstverständlich gibt *man* auch Informationen über sich selbst aus:

```
laborix$ man man
MAN(1)                   USER COMMANDS                    MAN(1)


NAME
     man - display reference manual pages; find  reference  pages
     by keyword
```

3) Es sei denn, Sie arbeiten auf einer Sun-4/65 unter SunOS 4.1.2, der Umgebung, in der die Beispiele entwickelt wurden.

```
SYNOPSIS
     man [-] [-t] [-M path [-T macro-package  [[section  title
     ...] ...
     man [-M path] -k keyword ...
     man [-M path] -f filename ...

DESCRIPTION
     man displays information from the reference manuals.  It can
     display  complete  manual pages that you select by title, or
     one-line summaries selected either by keyword  (-k),  or  by
     the name of an associated file (-f).
:q
```

2. Ein- / Ausgabe - Erste Schritte mit UNIX

Nachdem ihre UNIX-Welt mit der Einstellung der *richtigen* Shell jetzt in Ordnung ist, besteht Ihre zweite Amtshandlung darin, für das heutige Tagwerk einen neuen Katalog - ein *directory* - einzurichten.

```
laborix$ ls -FC
c++-demo/  gawk/      litdb/     mfg/       proj/      sem/
News/      ded*       hash/      m2/        news/      punkte
laborix$ mkdir neu
laborix$ cd neu
laborix$ ls
laborix$ pwd
/export/home/stanglow/neu
laborix$
```

ls -FC	listet die auf dem aktuellen Katalog verfügbaren Dateien auf und markiert Unterkataloge mit einem Schrägstrich (/) sowie ausführbare Programme mit einem Stern (*)
mkdir	erzeugt einen neuen Katalog
cd	wechselt zu einem Katalog
pwd	zeigt den aktuellen Katalog an

Als nächstes möchten Sie sich vielleicht darüber informieren, wer von Ihrer Abteilung bereits im Hause ist. Sie verwenden dazu das Kommando *who*:

```
laborix$ who
borchert ttyp0   May  3 13:08   (ariel:0.0)
suobermu ttyp1   May  5 14:07   (main01.rz.uni-ul)
borchert ttyp2   May  3 10:49   (ariel:0.0)
wald     ttyp3   Apr 14 08:46   (scix4)
framstag ttyp4   May  5 14:09   (main01:S.5)
borchert ttyp5   Apr 30 14:49   (titania)
wald     ttyp8   May  3 11:36   (hippo)
monika   ttyp9   Apr 28 08:24   (scix3)
grabert  ttypc   May  3 11:38   (orest)
```

```
wald     ttypd   Apr 30 15:27   (hippo)
borchert ttypf   Apr 27 19:20   (ariel:0.0)
monika   ttyq1   May  5 11:46   (scix3)
swg      ttyqe   May  5 10:41   (ariadne)
schoene  ttyr0   May  5 13:46   (scix7)
stanglow ttyr2   May  5 10:35   (helios)
martin   ttyr3   May  5 13:53   (castor)
laborix$
```

Offensichtlich können wir aus der Ausgabe von *who* doch nicht schließen, wer bereits da ist, und wer nicht. Einige ihrer Kollegen und Kolleginnen melden sich nämlich am Abend nicht ab und andere sind über das Netz[4)] von verschiedenen Rechnern aus sogar mehrmals auf Ihrem Abteilungsrechner angemeldet.

Auf einem größeren Rechner ist die Ausgabe von *who* unter Umständen länger als eine Bildschirmseite. Um dann trotzdem alle Zeilen lesen zu können, lenken wir die Ausgabe in eine Datei, und schauen sie uns anschließend in Ruhe mit *vi*, *pg*, *less* oder *more* an.

```
laborix$ who > anmeldungen
laborix$ pg anmeldungen
borchert ttyp0   May  3 13:08   (ariel:0.0)
suobermu ttyp1   May  5 14:07   (main01.rz.uni-ul)
borchert ttyp2   May  3 10:49   (ariel:0.0)
wald     ttyp3   Apr 14 08:46   (scix4)
framstag ttyp4   May  5 14:09   (main01:S.5)
borchert ttyp5   Apr 30 14:49   (titania)
stanglow ttyp6   May  5 12:56   (helios)
wald     ttyp8   May  3 11:36   (hippo)
monika   ttyp9   Apr 28 08:24   (scix3)
grabert  ttypc   May  3 11:38   (orest)
wald     ttypd   Apr 30 15:27   (hippo)
borchert ttypf   Apr 27 19:20   (ariel:0.0)
monika   ttyq1   May  5 11:46   (scix3)
:q
laborix$
```

Mit *pg* (paginate) kann eine Datei seitenweise betrachtet werden. Mehr oder weniger gleichen Zweck erfüllen *more* or *less*, die allerdings etwas komfortabler in ihrer Handhabung sind. Sofern diese Programme bei Ihnen verfügbar sind, erhalten Sie nähere Informationen über diese Tools mit dem schon erwähnten Manual-Kommando *man.*

4) Für die Behandlung der Shell spielt es keine Rolle, ob wir übers Netz arbeiten oder nicht.

Viele typische UNIX-Tools erhalten ihre Eingabe von der Tastatur (**stdin**) und schreiben ihre Ausgabe auf den Bildschirm (**stdout**), sofern der Benutzer dieses Verhalten nicht durch Optionen oder Ausgabeumlenkung verändert. Zu diesen Kommandos gehört auch *sort*, das solange Eingabe verarbeitet, bis der Benutzer am Anfang einer Zeile mit der Tastenkombination <Control><D>[5] das Eingabeende markiert. Die Eingabezeilen werden dann nach dem ASCII-Code sortiert und wieder am Bildschirm ausgegeben.

```
laborix$ sort
zeile 1
Zeile 3
Zeile 2
 Zeile 0
^D
 Zeile 0
Zeile 2
Zeile 3
zeile 1
laborix$
```

Für die Sortierung sind Blanks, Tabs und sonstige Sonderzeichen offensichtlich genauso relevant wie Groß- und Kleinschreibung.[6] Alle Zeichen werden streng nach ihrem ASCII-Code[7] bewertet, und der ist für Blanks nun einmal kleiner als für Buchstaben und für große Buchstaben geringer als für kleine.

Doch nun zurück zur Ausgabeumlenkung mit dem Winkel, die in der skizzierten Form für alle UNIX-Kommandos funktioniert. Analog dazu gibt es eine Eingabeumlenkung mit dem Winkel nach links, die die Shell dazu veranlaßt, die Eingabe für *sort* oder ähnliche Kommandos aus einer Datei statt von der Tastatur zu lesen.

```
laborix$ who > anmeldungen
laborix$ sort < anmeldungen > sortiert
laborix$ head -5 sortiert
borchert ttyp0   May  3 13:08   (ariel:0.0)
borchert ttyp2   May  3 10:49   (ariel:0.0)
borchert ttyp5   Apr 30 14:49   (titania)
borchert ttypf   Apr 27 19:20   (ariel:0.0)
framstag ttyp4   May  5 14:09   (main01:S.5)
laborix$
```

5) Im folgenden steht immer **^X** für die Tastenkombination <Control><X>.
6) Mit entsprechenden Optionen kann das Sortierverhalten geändert werden.
7) Eine ASCII-Tabelle erhalten Sie am Rechner mit *man ascii.*

>	lenkt die Ausgabe eines Kommandos in eine Datei; falls diese bereits existiert, wird sie auf die Länge NULL verkürzt; existiert sie noch nicht, wird sie angelegt
<	bewirkt die Eingabe aus einer Datei
head -5	gibt die ersten fünf Zeilen der Eingabe aus

3. *Pipes & Filters*

Wir haben im letzten Beispiel die Ausgabe von *who* in *anmeldungen* umgelenkt, diese Datei als Eingabe für *sort* verwendet, dessen Ausgabe in *sortiert* abgelegt und uns mit *head* die ersten fünf Zeilen des Ergebnisses anzeigen lassen. Den gleichen Effekt hätten wir auch ohne Zwischenlagerung in den diversen Dateien erzielen können. Mit den *Pipelines* bietet UNIX nämlich eine Möglichkeit, die Ausgabe eines Kommandos sofort als Eingabe für ein anderes Kommando zu verwenden.

```
laborix$ who | sort | head -5
borchert ttyp0   May  3 13:08   (ariel:0.0)
borchert ttyp2   May  3 10:49   (ariel:0.0)
borchert ttyp5   Apr 30 14:49   (titania)
borchert ttypf   Apr 27 19:20   (ariel:0.0)
framstag ttyp4   May  5 14:09   (main01:S.5)
laborix$
```

Kommandos wie *sort* oder *head* bezeichnen wir als Filter, weil sie aus aus ihrem Eingabestrom beliebig viele oder alle Zeilen herausfiltern, bearbeiten und dann wieder in den Ausgabestrom schreiben. Ein weiteres Beispiel für einen solchen Filter ist *grep*, das aus der Eingabe alle Zeilen heraussucht, die ein als Argument übergebenes Suchmuster enthalten.
Mit *grep* haben sie etwa die Möglichkeit festzustellen, ob ihr Chef bereits am Rechner angemeldet ist oder nicht. Wenn er sich zudem als rühmliche Ausnahme ihrer Abteilung abends immer vom Rechner ab- und morgens zunächst wieder dort anmeldet, finden Sie so sogar heraus, ab wann mit einer Chef-Visite zu rechnen ist.

```
laborix$ who | grep swg
swg      ttyqe   May  5 10:41   (ariadne)
laborix$
```

4. Shellskripte

Da es natürlich auf die Dauer lästig ist, bei der Suche nach Ihrem Chef solange immer wieder die gleiche Zeile einzutippen, bis er dann tatsächlich von *grep* gefunden wird, sollten Sie vielleicht ein kleines Shellskript schreiben, das diese Aufgabe dann für Sie unter einem möglichst kurzen Namen erledig.

```
laborix$ cat > chef
who | grep swg
^D
laborix$ chef
chef: execute permission denied
laborix$ ls -l
total 10
-rw-rw-r--   1 stanglow sai         1172 May  5 14:25 anmeldungen
-rw-rw-r--   1 stanglow sai           15 May  5 15:02 chef
-rw-rw-r--   1 stanglow sai         1172 May  5 14:37 sortiert
laborix$ chmod u+x chef
laborix$ chmod g-w chef
laborix$ ls -l chef
-rwxr--r--   1 stanglow sai           15 May  5 15:02 chef
laborix$ chef
swg      ttyqe   May  3 10:41   (ariadne)
laborix$
```

cat	ohne Argumente liest seine Eingabe von der Tastatur und gibt sie wieder aus, hier wegen der Umlenkung in die Datei *chef*
ls -l	listet die vorhandenen Dateien mit zusätzlichen Informationen aus. Die Zeichen 2 bis 9 jeder Zeile geben dabei die für die Datei vergebenen Rechte an. Jeweils in Dreiergruppen wird angezeigt, ob der Eigentümer (**u**ser), die Gruppe (**g**roup) oder der Rest der Welt (**o**ther) eine Datei lesen (**r**ead), schreiben bzw verändern (**w**rite) oder ausführen (e**x**ecute) dürfen.
chmod	verändert die Zugriffsrechte einer Datei. **u+x** setzt (**+**) für den Eigentümer (**u**) die Ausführungsrechte (**x**), **g-w** nimmt (**-**) der Gruppe (**g**) die Schreibrechte (**w**).

Um ein Shellskript wie *chef* aufzuführen, werden Lese- und Ausführungsrechte benötigt. Deshalb ging der erste Versuch schief und erst beim zweiten Male führt die Shell das Programm in einer Subshell aus. Die Wirkung von *chef* ist dabei die gleiche, wie bei der direkten Eingabe der Kommandos am Terminal.

5. Kontrollstrukturen, Exit-Status und Kommandozeilenargumente

Unser erstes Shellskript ist natürlich noch nicht perfekt. Wenn unserer Suche erfolgreich war, möchten wir vielleicht noch einen netten Text ausgeben. Glücklicherweise ist die Shell nicht nur ein Kommandointerpreter, sondern auch eine Programmiersprache mit Kontrollstrukturen, die wir sogar interaktiv nutzen können.

```
laborix$ if who | grep swg
> then
>       echo Chef ist da.
> fi
swg      ttyqe   May  3 10:41   (ariadne)
Chef ist da.
laborix$
```

if *cmd* **then** *cmds1* **else** *cmds2* **fi**

ist die If-Anweisung der Shell. Die Schlüsselworte müssen am Anfang einer Zeile stehen, lediglich Blanks und Tabs dürfen vorausgehen. Solange die Anweisung nicht abgeschlossen ist, fordert die Shell mit dem Sekundärprompt **>** weitere Eingabe vom Benutzer.

Die If-Anweisung funktioniert wie erwartet. Wenn das Kommando, hier also die Suche nach *swg* erfolgreich war, wird der Then-Zweig ausgeführt und das *echo* ausgeführt. Kontrollstrukturen verwenden wir meistens natürlich nicht interaktiv, sondern in Shellskripten:

```
laborix$ cat chef2
if who | grep swg > /dev/null
then
        echo "Chef ist da."
        exit 0
else
        exit 1
fi
laborix$ chef2
Chef ist da.
laborix$ echo $?
0
laborix$
```

Da wir in *chef2* selber eine Ausgabe generieren, benötigen wir die Ausgabe von *grep* nicht und leiten sie gleich in den großen **UNIX-Mülleimer** */dev/null.* Für die Bewertung der If-Bedingung wird nämlich nicht die Ausgabe eines Kommandos verwendet, sondern nur sein sogenannter *Exit-Status.* Per Konvention steht dabei **0 für Erfolg**, **jede andere Zahl für Mißerfolg**. Normalerweise bleibt der Exit-Status eines Programms unsichtbar. Für das zuletzt ausgeführten Programm wird er jedoch in *$?* abgelegt und kann mit *echo* sichtbar gemacht werden. Der Exit-Status unserer Shellskriptes wurde mit *exit* festgelegt, einer Anweisung, die ein Shellskriptes beendet und der wir optional den gewünschten Exit-Status als Argument mitgeben können.

Eigentlich liegt es nahe, unser Shellskript nicht nur nach einer bestimmten, sondern nach beliebigen Personen suchen zu lassen. So wie wir *grep* eine Suchmuster als Argument erhält, sollte unser Skript den gesuchten Login-Name übergeben bekommen. Die Realisierung dieser Erweiterung ist problemlos, weil die Argumente der Kommandozeile in einem Shellskript *$1* bis *$9* zur Verfügung stehen.

```
laborix$ cat suche
if who | grep $1 > /dev/null
then
        echo $1 ist angemeldet.
        exit 0
else
        exit 1
fi
laborix$ suche swg
swg ist angemeldet.
laborix$ suche mueller
laborix$ echo $?
1
laborix$
```

Unser Programm erlaubt nun die Suche nach beliebigen Personen und liefert einen Exit-Status, der den Erfolg oder Mißerfolg der Suche angibt. Letzteres ist die Voraussetzung dafür, daß wir dieses Skript sinnvoll in weitere Skripte integrieren können.

```
laborix$ cat suche2
while sleep 30
do
        if suche $1
        then
                exit 0
        fi
done
laborix$ suche2 swg
swg ist angemeldet.
laborix$ suche2 mueller
^C
laborix$
```

while *cmd* **do** *cmds1* **done**

Auch bei der While-Scheife der Shell ist die auszuwertende Bedingung ein Kommando, dessen Erfolg oder Mißerfolg bestimmt, ob der Schleifenkörper ausgeführt wird oder nicht.

sleep	suspendiert eine Shell für die angegebene Dauer in Sekunden von der Ausführung
ˆC	bricht ein laufendes Programm ab, anschließend meldet sich wieder die Shell.

Da *sleep* praktisch immer Erfolg liefert, sucht unser Skript solange nach *$1* bis *suche* den Exit-Status 0 liefert. Daraus resultieren zwei Schönheitsfehler. Zum einen legt sich das Skript vor der ersten Suche erst einmal 30 Sekunden schlafen und zum zweiten blockiert es unser Eingabeterminal solange, bis sich der gesuchte Benutzer anmeldet oder bis wir mit **ˆC** eine Abbruch forcieren. Das erste Problem lösen wir mit einer anderen Schleifenbedingung, das zweite wollen wir noch hier einmal zurückstellen.

```
laborix$ cat suche3
while true
do
        if who | grep $1 > /dev/null
        then
                echo $1 ist angemeldet.
                exit 0;
        else
                sleep 30
        fi
done
laborix$ suche3 swg
swg ist angemeldet.
laborix$ suche3 mueller
^C
laborix$
```

Wie der Name schon vermuten läßt, liefert die Anweisung *true* immer den Exit-Status 0, ist also prädistiniert für die Programmierung einer Endlosschleife, wie sie auch in *suche3* verwendet wird. Solche Endlosschleifen sind allerdings mit besonderer Vorsicht zu genießen. Ohne das *sleep* im Else-Teil würde es wahrscheinlich nicht einmal 30 Sekunden dauern, bis ein erboster Kollege vor Ihrer Tür steht, und sich darüber beschwert, daß ihr Skript durch die permanente Ausführung von *who* und *grep* praktisch die gesamte CPU-Leistung Ihres Rechners verbraucht.

6. !@#$ˆ&*()-+=|\‘{}[]:;’"<>.?

Wir haben jetzt schon eine ganze Reihe typischer Shell-Elemente kennengelernt: Metazeichen wie in der Überschrift[8)], Kontrollstrukturen, wie *if* oder *while,* Anweisungen mit speziellen Effekten wie *exit* oder *exec* und dazu noch Shellvariable und Dienstprogramme wie *grep* oder *cat*. Vieleicht denken Sie, daß es nun an der Zeit ist, sich eine kleine Datei mit Stichworten zu den verschiedenen Shell-Features anzulegen. Zusammen mit einem Shellskript, das Informationen aus der Datei extrahiert, hätten sie dann eine kleine Hilfedatenbank zur Verfügung.

```
laborix$ cat shelltab
sort    alphabetisch sortieren
echo    Argumente ausgeben
=       Zuweisung an Shell-Variable
grep    Muster suchen
cat     Dateien anzeigen
cut     Dateien projezieren
exit    Shellskript beenden
>       Ausgabeumlenkung
<       Eingabeumlenkung
$0      Name eines Shellskripts
$?      Exitstatus des letzten Kommandos
|       Pipeline-Symbol
laborix$ cat hilfe
if grep $1 shelltab
then
        :
else
        echo Keine Informationen ueber $1.
fi
laborix$ hilfe echo
echo    Argumente ausgeben
laborix$ hilfe =
=       Zuweisung an Shell-Variable
laborix$ hilfe tr
Keine Informationen ueber tr.
laborix$
```

8) Tatsächlich haben alle Zeichen in der Überschrift, sei es nun einzeln oder in Kombination, spezielle Bedeutungen für die Shell.

Das scheint ja hervorragend zu funktionieren. In unserer noch nicht ganz vollständigen Datei *shelltab* findet *grep* die gewünschten Einträge. Der Doppelpunkt ist eine leere Anweisung, die wir hier benötigen, um eine syntaktisch korrekte If-Anweisung zu erhalten. Wir brauchen keine Aktionen im Then-Teil, da *grep* ja bereits bei Erfolg eine Zeile aus der Datei *shelltab* ausgibt. Unsere Freude währt jedoch nur kurz, denn offensichtlich kann unsere kleines Programm keine Informationen über die Sonderzeichen der Shell ausgeben.

```
laborix$ hilfe $0
Keine Informationen ueber /bin/sh.
laborix$ hilfe >
syntax error: `newline or ;' unexpected
laborix$ hilfe |
> ^C
laborix$ hilfe $?
Keine Informationen ueber 2.
laborix$
```

Das Problem ist offensichtlich. Wir wollten Zeichen oder Zeichenfolgen mit Sonderbedeutung für die Shell als Argumente an unser Programm übergeben. Bevor das Skript jedoch zur Ausführung kommt, liest die die Shell die Kommandozeile und bewertet eben diese Zeichen bzw. reagiert auf sie.

- Die Shell erkennt *$0* und substitiert diese Variable durch ihren eigenen Programmnamen */bin/sh*, der dann an *hilfe* übergeben wird.
- Der Winkel wird überhaupt nicht ans Programm übergeben, da er für die Shell das Signal zur Ausgabeumlenkung ist. Dazu fehlt ihr allerdings ein Dateiname und sie bemängelt deshalb einen Syntaxfehler.
- Nach dem Pipelinesymbol erwartet die Shell noch ein weiteres Kommando, das die Ausgabe von *hilfe* auffangen soll. Sie gibt deshalb den Sekundärprompt aus.
- Das vorausgehende Kommando terminierte durch ^C, einem Fehler der zum Exit-Status 2 führte. Bei der Ausführung der letzten Beispielzeile wurde *$?* durch genau diesen Wert substituiert.

Wir benötigen also eine Möglichkeit, unsere Argumente vor der Bewertung durch die Shell zu schützen. Die einfachste und sicherste Methode ist dabei die Verwendung von Apostrophen.

```
laborix$ hilfe '$0'
$0      Name eines Shellskripts
laborix$ hilfe '|'
|       Pipeline-Symbol
laborix$ hilfe 'Dateien anzeigen'
grep: anzeigen: No such file or directory
shelltab:cat    Dateien anzeigen
```

```
shelltab:cut    Dateien projezieren
Keine Informationen ueber Dateien anzeigen.
laborix$
```

Die Apostrophen haben gewirkt, in den beiden ersten Fällen sogar so, wie wir es erwartet haben. Dann straft sich sich unser Skript aber selbst Lügen und produziert außerdem noch eine unerwartete Fehlermeldung. Der Grund dafür liegt natürliche an der Quotierung.[9] Der eingeschlossenen Text wird vor jeder Interpretation durch die Shell bewahrt. Selbst die Sonderbedeutung das Newlines am Zeilenende, das normalerweise ein Kommandoende markiert, oder - wie in unserem Fall - die Bedeutung des Blanks als Wort- und damit als Argumenttrenner wird aufgehoben. Aus diesem Grund bekommt *hilfe* nur ein, allerdings aus zwei Worten bestehendes Argument übergeben.

Vor der Bewertung von *grep $1* substituiert die Shell den Parameter *$1* und führt dann die Anweisung *grep Dateien anzeigen shelltab* aus. Das erste Argument betrachtet *grep* als Suchmuster, die übrigen als Eingabedateien. So kommt es zur Fehlermeldung, weil eine Datei *anzeigen* nicht existiert. Das dritte Argument *shelltab,* wie dann zwar wie gewohnt bearbeitet, aufgrund des vorherigen Fehlers terminiert *grep* jedoch mit einem Exit-Status ungleich Null, so daß der Else-Zweiges der If-Anweisung ausgeführt wird.

Wir müssen also verhindern, daß in unserem Shellskript bereits aus *$1* zwei Argumente für *grep* werden. In *hilfe2* verwenden wir deshalb wieder Apostrophen, um die Wirkung von Blanks als Argumenttrenner zu unterdrücken. Außerdem sollen sie in der modifizierten Fehlermeldung des Else-Teils die Interpretatation der Winkel zu verhindern.

```
laborix$ cat hilfe2
if grep '$1' shelltab
then
        :
else
        echo * Keine * Informationen ueber '>$1<'.
fi
laborix$ hilfe2 'Dateien anzeigen'
anmeldungen chef chef2 chef3 hilfe hilfe2 shelltab sortiert suche suche2
suche3 Keine anmeldungen chef chef2 chef3 hilfe hilfe2 shelltab sortiert
suche suche2 suche3 Informationen ueber >$1<.
laborix$
```

Das Ergebnis ist nicht überzeugend. Die Quotierung verhindert *jede* Interpretation, also auch die von *$1.* Bei *echo* haben wir zudem vergessen, die Sterne ebenfalls in Hochkommas einzuschließen. Darum hat die Shell dieses Metazeichen zu einer Liste der Dateien auf dem aktuellen Katalog expandiert.

9) Quotierung ist ein sehr beliebter, im Deutschen aber eigentlich falscher Sammelbegriff [Duden86] für die Verwendung der verschiedenen *quotation marks* (’ " ‘) in der Shell.

Glücklicherweise kennt die Shell nicht nur eine, sondern vier Möglichkeiten, die Interpretation von Text zu verändern.

’...’	verhindert *jede* Bewertung des eingeschlossenen Textes.
"..."	verhindert die Bewertung des eingeschlossenen Textes. Lediglich $-Variablen, die Rückwärtshochkommas **‘ ‘** und der Backslash \ werden erkannt.
‘...‘	führen zur Ausführung des eingeschlossenen Textes als Kommando. Der gesamte Ausdruck wird dann durch die Ausgabe des Kommandos ersetzt.
\ c	hebt die Sonderbedeutung von *c* auf.

Wir werden noch detailierter auf die Metazeichen der Shell und die Kommandosubstitution eingehen müssen, wollen uns jedoch im Moment mit einer Korrektur des Hilfeskripts begnügen.

```
laborix$ cat hilfe3
if grep "$1" shelltab
then    :
else
        echo \* Keine \* Informationen ueber ">$1<".
fi
laborix$ hilfe3 'Dateien anz'
cat     Dateien anzeigen
laborix$ hilfe3 \>
>       Ausgabeumlenkung
laborix$ hilfe3 \'
* Keine * Informationen ueber >'<.
laborix$
```

7. Quotierung und Shellvariable

Zum Abschluß wollen wir uns noch einige Beispiele zur Quotierung und dabei insbesondere zur Substitution eines Textes durch die Ausgabe eines Kommandos anschauen. Ziel des folgenden Beispiels ist es, in einer Shellvariable die Studen der aktuellen Uhrzeit abzulegen.

```
laborix$ DATUM=date                                                #( 1)
laborix$ echo $DATUM                                               #( 2)
date
laborix$ DATUM=`date`                                              #( 3)
laborix$ echo $DATUM                                               #( 4)
Fri May 7 09:57:19 MET DST 1993
laborix$ echo "$DATUM"                                             #( 5)
Fri May  7 09:57:19 MET DST 1993
laborix$ echo `date`                                               #( 6)
Fri May 7 09:57:55 MET DST 1993
laborix$ echo `date` | tr ' ' '\11'                                #( 7)
Fri     May     7       09:58:39        MET     DST     1993
laborix$ echo `date` | tr ' ' '\11' | cut -f4                      #( 8)
09:59:36
laborix$ echo `date` | cut -f4 -d' '                               #( 9)
10:00:40
laborix$ echo `date` | cut -f4 -d' ' | cut -f1 -d:                 #(10)
10
laborix$ STUNDEN=`echo `date` | cut -f4 -d' ' | cut -f1 -d:`   #(11)
sh: syntax error: `|' unexpected
laborix$ STUNDEN=`echo \`date\` | cut -f4 -d' ' | cut -f1 -d:` #(12)
laborix$ echo $STUNDEN                                             #(13)
10
laborix$
```

(1) Zuweisung an die Shellvariable *DATUM*.

(2) Ausgabe der Variable. Da bei der Zuweisung die rechte Seite normalerweise nicht als Kommando, sondern nur als Text aufgefaßt wird, enthält die Variable den Text *date*.

(3) Bei der Bewertung der Zeile substituiert die Shell *'date'* durch die Ausgabe des Kommandos.

(4) *$DATUM* wird substituert und *echo* erhält sieben durch ein oder mehrere Blanks getrennte Argumente, die es dann jeweils mit einem Blank als Zwischenraum ausgibt.

(5) Hier wird in *$DATUM* zwar die Shellvariable ersetzt, durch die umgebenden Anführungsstiche wird jedoch nur ein Argument an *echo* übergegen, das vor der 7 ein Blank enthält.

(6) *date* wird durch die Ausgabe von *date* ersetzt, die dann als Argumente an *echo* übergeben wird. Diese Anweisung hat den gleichen Effekt wie (3) und (4).

(7) **tr** *c1 c2* ersetzt im Eingabestrom jedes Zeichen *c1* durch *c2.* Die Zeichen können optional mit einem Backslash und ihrem ASCII-Wert als maximal dreistellige Oktalzahl spezifiziert werden. Hier wird jedes Blank durch ein Tab (ASCII-Wert 9, Oktal 11) e"rsetzt.

(8) **cut** ist ein weiterer nützlicher Filter, der aus den Zeilen des Eingabestroms die mit **-f** angegebenen Spalten herausschneidet. *cut* geht normalerweise davon aus, das die Spalten durch Tabs voneinander getrennt sind.

(9) Mit **-d***c* kann der Spaltentrenner (delimiter) von *cut* auf *c,* in unserem Beispiel ein Blank umdefiniert werden. Die Quotierung ist notwendig, da sonst die Shell das Blank als Worttrenner interpretiert.

(10)
Aus dem Resultat von (9) wird durch ein weiteres *cut* das erste Feld, die Stunden, herausgefiltert. Feldtrenner ist dabei der Doppelpunkt.

(11)
Wie bei (3) soll jetzt das Resultat von (10) an *STUNDEN* zugewiesen werden. Shell-Quotes können aber nicht geschachtelt werden, deshalb versucht die Shell zwei Kommandos auszuführen: *echo* und |*cut -f4* @...@". Ein Pipelinesymbol am Kommandobeginn ist jedoch ein Syntaxfehler.

(12)
Hier werden die inneren Rückwärtshochkommas geschützt. Die Shell ersetzt zunächst *date*\\ durch *date* und führt dann die Pipeline in einer Subshell aus. Bei dieser Ausführung wird dann *date* durch die Datumsausgabe ersetzt und und an *echo* übergeben. Das Resultat der Pipeline insgesamt (vgle. (10)) wird schließlich an *STUNDEN* zugewiesen.

(13)
illustriert den Erfolg von (12).

Sie fragen sich vielleicht, warum in den Beispielen (6) bis (12) nicht ein einfaches *date* als Beginn der Pipeline ausreicht. Der Grund liegt in dem Blank, das von *date* vor einstelligen Tagen ausgegeben wird. Diese Blank brächte die Spaltenzählung von *cut* durcheinander, so daß die Anweisungen nur nach dem 10. jeden Monats funktionieren würde. Bei dem Umweg über *echo* fällt dieses Blank dagegen weg.

Sie fragen sich vielleicht auch, ob die Numerierung der Kommandos nicht doch nachträglich in das Protokoll eingefügt wurde. Die Antwort sei dahingestellt, sie können jedoch sicher sein, daß die Numerierung keinen Einfluß auf das Ergebnis der Anweisungen hat. Durch das Fluchtsymbol (#) werden in der Shell nämlich Kommentare eingeleitet, die jeweils bis zum Zeilenende reichen.

Damit sind wir am Ende unserer kleine Reise durch die Welt von UNIX und seiner Shell angelangt und wollen uns und unserem Rechner zunächst einmal eine kleine Ruhepause gönnen. Wir signalisieren also unserer Login-Shell mit ˆD (oder **exit**) das Ende der Eingabe vom Terminal und melden uns damit ab.

```
laborix$ ^D
login:
```

8. 0 - 1 - 2 : *stdin - stdout - stderr*

Wir haben bereits in unserem kleinen Crash-Kurs gesehen, daß UNIX-Dienstprogramme wie *sort* zwar im einfachsten Fall ihre Eingabe von der Tastatur lesen und ihre Ausgabe auf den Bildschirm schreiben, daß es aber auch verschiedene Möglichkeien gibt, dieses Verhalten zu verändern.

```
laborix$ sort                      #(1)
Zacharias
Auto
^D
Auto
Zacharias
laborix$ cat in
Zacharias
Auto
laborix$ sort < in > out           #(2)
laborix$ cat out
Auto
Zacharias
laborix$ sort -o out in            #(3)
laborix$ cat out
Auto
Zacharias
laborix$
```

Jedem Programm eines UNIX-Systems stehen für die Ein- und Ausgabe standardmäßig zunächst einmal drei Kommunikationskanäle zur Verfügung:

- Kanal 0: die Standardeingabe (*stdin*),
- Kanal 1: die Standardausgabe (*stdout*),
- Kanal 2: die Standarddiagnose (*stderr*).

Wie wir am Aufruf (1) erkennen, sind alle Kanäle per Voreinstellung mit der Tastatur bzw. dem Bildschirm verbunden. Dies gilt insbesondere auch für die Standarddiagnose, die wir jedoch für den Augenblick noch einmal zurückstellen wollen.

Die standardmäßige Geräteverbindung der Kanäle mit dem Terminal kann, wie in (2) durch die Verwendung von Winkeln durch Dateiverbindungen ersetzt werden. Für diese Umlenkungen ist die Shell verantworlich, denn bevor sie ein Programm ausführt bewertet sie die die Kommandozeile und insbesondere interpretiert sie dabei die Winkel als Signal zur Ein-/Ausgabeumlenkung. Die Kommandos selbst bemerken davon nichts[10)] und verhalten sich genauso, als wenn die Kanäle mit dem Terminal verbunden sind.

Fast alle UNIX-Tools akzeptieren Argumente, wobei die meisten Kommandos, die im Zusammenhang mit Datenbankanwendungen eine Rolle spielen haben, als

cmd [*options*] [*files*]

aufgerufen[11)] werden. Welche Optionen und Argumente im einzelnen zulässig sind, ist unter dem Stichwort Synopsis[12)] auf der Manualseite des jeweiligen Kommandos dargestellt. Im Fall von *sort* finden wir dort folgende, hier verkürzt wiedergegebene Beschreibung:

sort [**-rdn**] [**-o** *output*] [*files*]

Die Argumente von *sort* in unserem Beispiel (3) entsprechen dieser Synopsis und bewirken eine Veränderung der Arbeitsweise dieses Tools. Es liest nun nicht mehr die Standardeingabe, sondern die Datei *in* und schreibt seine Ausgabe statt auf die Standardausgabe in die Datei *out.* Obwohl (2) und (3) den gleichen Effekt haben, haben wir es mit zwei qualitativ völlig unterschiedlichen Anweisungen zu tun:

---

10) Theoretisch kann ein Programm mit einigen Klimmzügen *doch* herausbekommen, ob seine Ein- oder Ausgabe umgelenkt wird. Die wenigsten System-Kommandos machen sich jedoch diese Mühe.

11) In Syntaxdarstellungen markieren wir optionale Teile in eckigen Klammern []. Kommandonamen und konkrete Optionen, sind **fett**, variable Elemente des Aufrufs *kursiv* dargestellt. UNIX-typisch, aber etwas inkonsequent, weil formal nicht eindeutig, verwenden wir [**-**$c_1c_2...c_n$], um eine Menge von einbuchstabigen Optionen $c_1,c_2,...,c_n$, zusammenzufassen, die einzeln oder kombiniert beim Aufruf des Kommandos verwendet werden können.

12) knappe Zusammenfassung

(2) Die Shell bewertet die Anweisung und findet dabei zwei Winkel. Sie sorgt dafür, daß die Standardeingabe für *sort* mit der Datei *in*, die Standardausgabe mit der Datei *out* verbunden wird und ruft dann das Programm *ohne* Argumente auf. Deshalb liest bzw. schreibt *sort* auf die, allerdings umgelenkten, Standardkanäle.

(3) Die Shell findet bei der Bewertung der Kommandozeile keine Winkel und nimmt deshalb keine Umlenkung vor. Sie ruft also *sort* mit den Argumenten *-o out in* auf. Das Programm verwendet daraufhin die Dateien *in* und *out* für Ein- und Ausgabe. Die Standardkanäle sind zwar weiterhin mit dem Terminal verbunden, werden von *sort* jedoch nicht verwendet.

Das folgende Beispiel illustriert noch einmal, daß eine Anweisung mit Umlenkung (2) das Verhalten der Shell, eine Anweisung mit Optionen (3) das Verhalten von *sort* verändert:

```
laborix$ sort < in1
sh: in1: cannot open
laborix$ sort > out1
sh: out1: cannot create
laborix$ sort -o out1 in
sort: can’t create out1: Permission denied
laborix$ sort in1
sort: can’t open in1: No such file or directory
```

Alle vier Anweisungen scheitern, weil die Datei *in1* nicht existiert und *out1* nicht beschrieben werden kann. In den beiden ersten Fällen ist es die Shell, die dieses feststellt und deshalb *sort* erst gar nicht aufruft. Die beiden letzten Anweisungen machen der Shell keine Probleme, da sie Bedeutung der Argumente nicht kennt. Somit wird *sort* tatsächlich aufgerufen, terminiert aber sofort mit einer Fehlermeldungen, da die Dateien nicht existieren oder nicht schreibbar sind.

Mit diesem Beispiel kommen wir jetzt auch zu den Fehlermeldungen und damit zum dritten Kommunikationskanal, der Standarddiagnose. Dieser Kanal wird von den Tools benutzt, um Fehlermeldungen auszugeben. UNIX unterstützt also zwei Ausgabekanäle: *stdout* für die reguläre Ausgabe und *stderr* für Fehlermeldungen, Warnungen und sonstige Diagnoseausgaben.

```
laborix$ cat in in1              # (1)
Zacharias
Auto
cat: cannot open in1: No such file or directory
laborix$ cat in in1 > out        # (2)
cat: cannot open in1: No such file or directory
laborix$ cat out
Zacharias
Auto
laborix$ cat in in1 2>out        # (3)
```

```
Zacharias
Auto
laborix$ cat out
cat: cannot open in1: No such file or directory
laborix$ cat in in1 1>out 2>err   # (4)
laborix$ cat out
Zacharias
Auto
laborix$ cat err
cat: cannot open in1: No such file or directory
laborix$
```

Beide Ausgabekanäle sind voneinander vollig unabhängig, auch wenn sie per Voreinstellung beide mit dem Terminal (1) verbunden sind. Die Unabhängigkeit der beiden Kanäle zeigt sich erst im Zusammenhang mit der Umlenkung. Es kann dabei wie in (2) und (3) nur einer beiden Kanäle umgelenkt werden, während der andere mit dem Terminal verbunden bleibt. Es ist jedoch wie in (4) auch möglich, beide Kanäle gleichzeitig, aber in verschiedene Dateien zu lenken. Um die Kanäle zu differenzieren, wird dem Winkel die Kanalnummer ohne Zwischenraum vorangestellt. Dies gilt genauso für die Ein- wie auch die Ausgabeumlenkung:

[n]<

Umlenkung von Eingabekanal *n* (mit Voreinstellung 0, also *stdin*).

[n]>

Umlenkung von Ausgabekanal *n* (mit Voreinstellung 1, also *stdout*)

Wir wollen hier nicht erörtern, was für $n \geq 3$ passiert,[13] sondern uns stattdessen lieber mit den "normalen" Umlenkungen beschäftigen.

---

13) Der interessierte Leser lese das Manual zu *fdopen* übersetze folgendes Programm und experimentiere etwas mit *a.out 3<in 4>out* herum:

```
# include <stdio.h>
main()
{    FILE *in3 = fdopen(3,"r"),           /* Kanal 3 zum Lesen oeffnen     */
           *out4 = fdopen(4,"w");         /* Kanal 4 zum Schreiben oeffnen */
     int c;
     if (in3 && out4)
     {       while ((c=fgetc(in3))!=EOF)
                     fputc(c,out4);
             fclose(in3); fclose(out4); exit(0);
     }
     else
             fprintf(stderr,"Cannot open file descriptors"), exit(1);
}
```

9. Gefahr im Verzug

Wie das folgende Beispiel zeigt, ist die Ausgabeumlenkung ist nicht ganz ungefährlich:

```
laborix$ cat in out             # (1)
a
b
a
b
laborix$ sort out > out         # (2)
laborix$ cat out                # (3)
laborix$ cat in > in            # (4)
cat: input/output files 'in' identical
laborix$ cat in                 # (5)
laborix$
```

(1) Zwei Dateien mit je 2 Zeilen.

(2) Der Winkel ist für die Shell das Signal zur Eingabeumlenkung. Sie erzeugt *out* neu, bzw. verkürzt die Datei auf die Länge 0, bevor das Kommando ausgeführt werden. Damit ist die Eingabedatei für *sort* leer, folglich wird auch keine Ausgabe erzeugt.

(3) Da *out* jetzt leer ist, erfolgt keine Ausgabe.

(4) Auch hier wird *in* zerstört, bevor *cat* aufgerufen wird. Wenn das Kommando merkt[14)], daß die Ein- und Ausgabedatei gleich sind, ist es zu spät.

(5) Nach der vorherigen Anweisung ist *in* jetzt leer.

Es bleibt also festzuhalten, daß die Shell Dateien, die als Ziel einer einfachen Ausgabeumlenkung angegeben werden, in jedem Fall zerstört. Wollen wir nacheinander mehrere Ausgaben in die gleiche Datei lenken, ohne den alten Inhalt zu zerstören, müssen wir statt eines einfachen, einen doppelten Winkel verwenden.

☞ **cmd n>> file**
veranlßt die Shell, dafür zu sorgen, die Ausgabe von *cmd* über Kanal *n* (Voreinstellung wieder n=1) an die Datei *file* anzuhängen.

Auch hierzu ein kleines Beispiel:

14) Als eines von wenigen Kommandos macht *cat* sich offensichtlich die Mühe festzustellen, wohin seine Standardausgabe umgelenkt werden.

```
laborix$ cat count_users
date >> PROTOCOL             # (1)
who | wc -l >> PROTOCOL      # (2)
echo >> PROTOCOL             # (3)
laborix$ count_users
laborix$ count_users
laborix$ cat PROTOCOL
Thu May 27 15:12:22 MET DST 1993
      29

Thu May 27 15:12:39 MET DST 1993
      31

laborix$
```

Unsr Shellskript schreibt bei jedem Aufruf an das Dateiende von *PROTOCOL* die Ausgabe von *date* (1) fügt dann die Anzahl der angemeldeten Benutzer (2) und noch eine Leerzeile an (3).

- **wc** [**-lwc**] [*files*]

zählt die Anzahl der Zeilen (-l), Worte (-w) oder Zeichen (-c) in den angegebenen Dateien bzw. in der Standardeingabe. Diese kommt in unsererm Beispiel weder vom Terminal noch aus einer Datei, sondern aus einer Pipeline, die mit der Ausgabe von *who* gefüllt wird.

10. Am Mischpult - Ausgabekanäle zusammenfassen

Vielleicht möchten Sie einmal Fehlermeldungen und Standardausgabe eines Kommandos in eine gemeinsame Datei umlenken. Die einfachen Lösungen mit den Winkeln funktionieren nicht, wie (1) und (2) in folgendem Protokoll beweisen.

```
laborix$ cat xy in
cat: cannot open xy: No such file or directory
b
a
laborix$ cat xy in >out 2>out    # (1)
laborix$ cat out
b
a
 cannot open xy: No such file or directory
laborix$ cat xy in >out 2>>out   # (2)
laborix$ cat out
b
a
 cannot open xy: No such file or directory
laborix$
```

In beiden Fällen ist die Fehlermeldung erstens unvollständig und zweites erscheint sie nicht an der richtigen Stelle. Was im Einzelfall bei Anweisungen wie (1) und (2) tatsächlich in einer Datei landet, bleibt dem Zufall überlassen, die Ergebnisse sind aber in keinem Fall verläßlich. Um die Standard- und Diagnoseausgabe zu mischen, benötigen wir eine weitere Art der Umlenkung:

n*>&*m

- die Ausgabe über Kanal *n* wird mit Kanal *m* gemischt.

```
laborix$ cat in 1>&2     # (1)
b
a
laborix$ cat xy 2>&1     # (2)
cat: cannot open xy: No such file or directory
laborix$
```

Auf diese Weise können wir reguläre Ausgaben zu Fehlermeldungen machen (1) und umgekehrt (2). Daß dies wie gewünscht funktioniert können wir an obigen Beispielen noch nicht erkennen, wir müssen dazu beide Kanäle gemeinsam in eine Datei umlenken.

```
laborix$ cat xy in >out 2>&1        # (1)
laborix$ cat out
cat: cannot open xy: No such file or directory
b
a
laborix$ cat xy in 2>&1 >out        # (2)
cat: cannot open xy: No such file or directory
laborix$ cat out
```

```
b
a
laborix$
```

Anweisung (1) mischt wie gewünscht die Ausgabe über beide Kanäle. Die Standardausgabe wird auf *out* umgelenkt, und der Kanal 2 wird zur umgelenkten Standardausgabe hinzugemischt. Wir erkennen aber auch an (2), daß die Reihenfolge bei der gemeinsamen Umlenkung eine Rolle spielt. Da die Shell eine Zeile von links nach rechts liest, lenkt sie zunächst die Standarddiagnose auf die Standardausgabe um, die zu diesem Zeitpunkt aber noch mit dem Terminal verbunden ist. Erst dann wird die Standardausgabe umgelenkt, was aber keinen Effekt mehr auf die bereits mit dem Terminal verbundene Standarddiagnose mehr hat. Aus diesem Grunde erscheint die Fehlermeldung am Bildschirm und nur die Standardausgabe wandert in die Datei.

Zugegeben: der Sinn solcher Umlenkungen ist nicht gerade offensichtlich. Im Zusammenhang mit der Shell-Programmierung werden wir jedoch eine noch ganze Reihe von Beispielen sehen, wo die Umlenkung eines Ausgabekanals auf einen anderen notwendig gebraucht wird.

11. *tee* - Ausgabe nach *stdout* und in *file*

```
bash$ man tee
TEE(1L)                                                    TEE(1L)


NAME
       tee  - read from standard input and write to standard out-
       put and files

SYNOPSIS
       tee  [-ai]   [--append]   [--ignore-interrupts]   [--help]
       [--version] [file...]

DESCRIPTION
       This  manual  page  documents the GNU version of tee.  The
       tee command copies the standard input to the standard out-
       put and any files given as arguments.

       If  a  file being written to does not already exist, it is
       created.  If a file being written to already  exists,  the
       data  it previously contained is overwritten unless the -a
       option is used.
:q
bash$ cat xx
grep: anzeigen: No such file or directory
shelltab:cat    Dateien anzeigen
shelltab:cut    Dateien projezieren
Keine Informationen ueber Dateien anzeigen.

bash$ cat xx | tee file | wc -l
      4

bash$ cat file
grep: anzeigen: No such file or directory
shelltab:cat    Dateien anzeigen
shelltab:cut    Dateien projezieren
Keine Informationen ueber Dateien anzeigen.
bash$
bash$ cat xx | sort | tee xx.s | wc -l
      4
bash$ cat xx.s
Keine Informationen ueber Dateien anzeigen.
grep: anzeigen: No such file or directory
shelltab:cat    Dateien anzeigen
shelltab:cut    Dateien projezieren
bash$
```

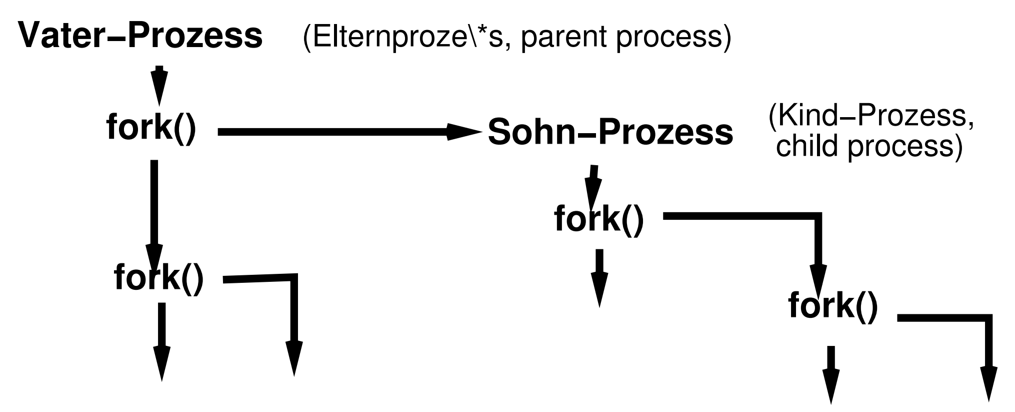
12. Da Capo - Tabellarisch

cmd	*args*	meaning
who		angemeldete Benutzer anzeigen
date		Datum und Uhrzeit ausgeben
vi	*file*	Datei *file* editieren
pg	*file*	Datei *file* seitenweise anzeigen.
less	*file*	Datei *file* seitenweise anzeigen.
more	*file*	Datei *file* seitenweise anzeigen.
man	*cmd*	Online-Manual über *cmd* anfordern
cat	*[files]*	Dateien *files* nacheinander auf Standardausgabe ausgeben.
sort	*[files]*	gesamten Inhalt aller Dateien *files* sortieren
tee	*file*	stdin nach stdout und angegebenes *file* kopieren
	–	als Argument von **cat**, **sort** und einigen anderen Kommandos als spezieller Dateiname für *Standardeingabe* interpretiert

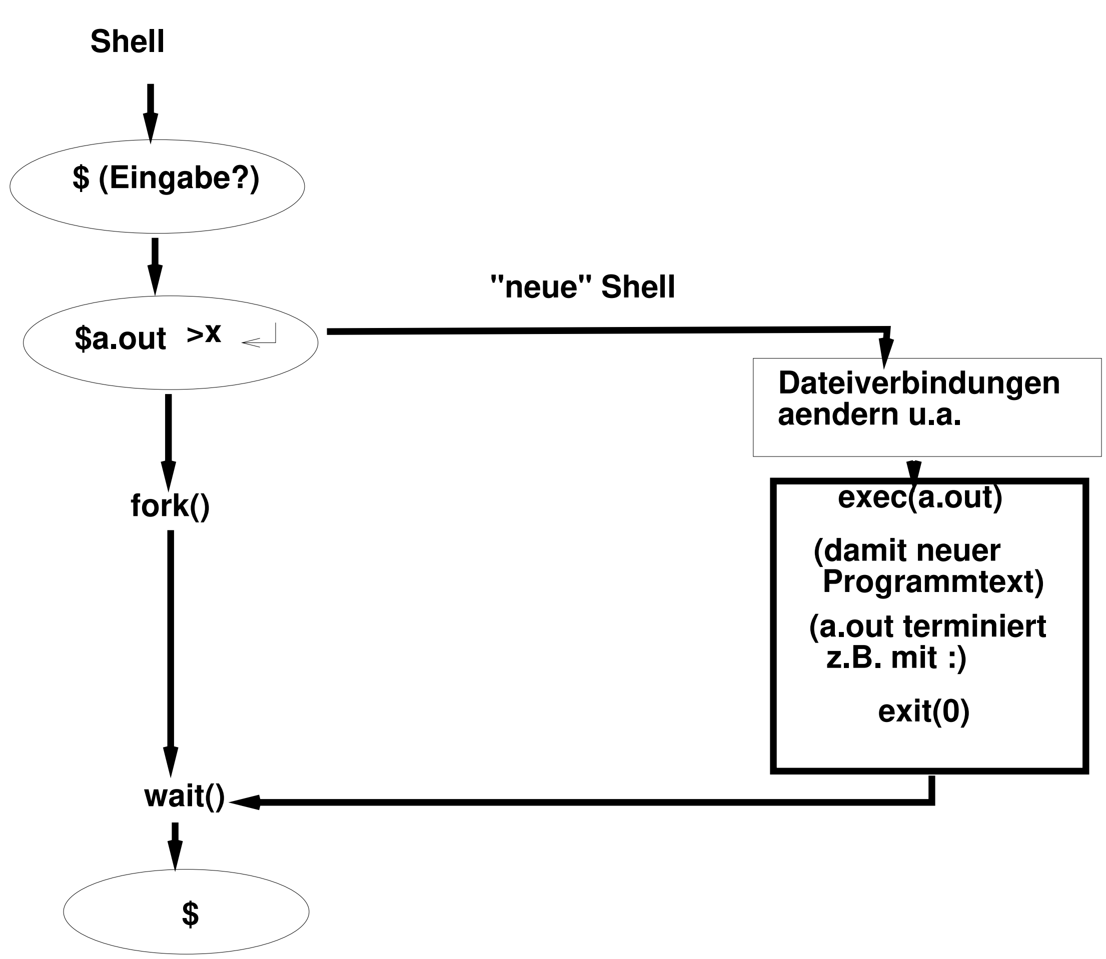
#		Kommentar bis zum Zeilenende einleiten
cmd	**> file**	Standardausgabe (*stdout*) von *cmd* auf *file* umlenken
cmd	**1> file**	dto.
cmd	**2> file**	Standarddiagnose (*stderr*) von *cmd* auf *file* umlenken
cmd	**< file**	Standardeingabe (*stdin*) von *cmd* aus *file* lesen
cmd	**0< file**	dto.
cmd	**>> file**	Standardausgabe von *cmd* an *file* anhängen
cmd	**1>> file**	dto.
cmd	**2>> file**	Standarddiagnoseausgabe von *cmd* an *file* anhängen

13. Prozesse - *builtins*

- ☐ Hierarchie und Vererbung von Prozessen



Shell-Kommando:



Prozesse vererben Teile ihrer Umgebung an ihre Kindprozesse:

– die Standard-Dateiverbindungen
– den aktuellen Katalog
– Umgebungsvariablen

Auch die Regeln für die Weitergabe von Informationen folgen streng der hierarchischen Vererbungslehre. Elternprozesse können ihren Kindern Teile ihrer Umgebung als Kopie mitgeben. Die umgekehrte Richtung ist nicht möglich. Prozesse können Daten untereinander nur über **IPC**-Mechanismen (*interprocess communication*) austauschen.

Eine Ausnahme bildet der Beendigungsstatus (kleine ganze Zahl), den ein Prozeß bei seiner Beendigung an seinen Erzeugerprozeß zurückgeben kann. Der Beendigungsstatus 0 signalisiert einen erfolgreichen Ablauf des Prozesses (UNIX -Konvention); ein von Null verschiedener Wert ist das Zeichen für eine nicht erfolgreiche Beendigung und wird in der Regel mit einer Fehlermeldung auf der Standard-Fehlerausgabe *stderr* verbunden.

□ built-in

```
bash$ type who
who is /usr/bin/who
bash$ type cd
cd is a shell builtin
bash$ type echo
echo is a shell builtin
bash$ type date
date is /bin/date
bash$ type cat
cat is /usr/bin/cat
bash$ type test
test is a shell builtin
bash$ type export
export is a shell builtin
bash$ type ls
ls is /bin/ls
bash$ type exit
exit is a shell builtin
bash$ type sort
sort is /usr/bin/sort
bash$
```

Reguläre Ausdrücke

1. Formale Sprachen

Die im folgenden dargestellten Tools leben sehr stark von sog. regulären Ausdrücken, die vorab allgemein behandelt werden sollen.

Reguläre Ausdrücke sind nach /Schöning92/ zunächst spezielle Formeln, mit denen Sprachen definiert werden können - die sog. regulären Sprachen. Wenn wir uns also mit regulären Ausdrücken beschäftigen müssen, sollten wir erst einen (kurzen) Blick auf **formale Sprachen** im allgemeinen werfen.

Eine (formale) Sprache ist zunächst einmal über einem endlichen Alphabet - dem **Vokabular** - aufgebaut und ist zu verstehen als eine Teilmenge aller endlichen Folgen von Elementen aus diesem Vokabular. Da eine solche Sprache i.a. sehr viele (unendlich viele) Elemente (Sätze der Sprache) enthält, ist eine endliche Beschreibung notwendig. Dazu werden sog. Grammatiken verwendet.

Zur Notation einer Grammatik benötigt man selbst wieder eine Sprache, die dann als **Meta-Sprache** bezeichnet wird. Die Symbole dieser Meta-Sprache sind dann die **Meta-Symbole**. Sehr häufig werden Grammatiken in Form von "Ersetzungssystemen" definiert; man gibt Regeln an, wie Folgen von Meta-Symbolen sukzessive ersetzt werden können, bis schließlich eine Folge von Symbolen aus dem Vokabular der zu definierenden Sprache erreicht oder "produziert" ist. Dabei werden Symbole aus dem Vokabular der zu definierenden Sprache nicht ersetzt und somit auch als **Terminal-Symbole** bezeichnet. Symbole der Meta-Sprache werden ersetzt und heißen in dem Zusammenhang dann **Non-Terminal-Symbole**. In der Darstellung können die Symbole z.B. dadurch unterschieden werden, daß man Non-Terminalsymbole fettgedruckt setzt, die Terminalsymbole normal setzt oder daß man Non-Terminalsymbole in spitze Klammern setzt oder daß man die Terminalsymbole in Doppelapostrophen klammert oder ...

Die Grammatik selbst wird mittels **(Produktions-)Regeln** definiert, wofür unterschiedliche Notationen verwendet werden (z.B. **EBNF** - Extended Backus Naur Form). Wir verwenden hier folgende Notation:

linke Seite → *rechte Seite*

Formal ist eine Grammatik ein 4-Tupel bestehend aus:

(1) endliche Menge der Terminalsymbole

(2) endliche Menge der Non-Terminalsymbole (disjunkt zu obiger Menge)

(3) endliche Menge von Produktionsregeln, wobei die linke Seite eine nichtleere endliche Folge und die rechte Seite eine endliche (auch leere) Folge von jeweils Terminal- und Non-Terminalsymbolen ist; Terminalsymbole auf linken Seiten stehen in derselben Reihenfolge wieder auf der rechten Seite (s.u.: Ersetzung!). Auf der rechten Seite kann auch "nichts" stehen - dazu verwendet man als Terminalsymbol ε.

(4) ein als Startsymbol ausgezeichnetes Element der Non-Terminalsymbole, wobei bei mindestens einer Produktionsregel die linke Seite aus diesem Startsymbol besteht.

Damit kann ein **Ableitungsbegriff** erklärt werden: ausgehend vom Startsymbol kann durch wiederholte Anwendung von Regeln sukzessive eine Symbol-Folge (aus Terminal- und/oder Non-Terminalsymbolen) abgeleitet werden; die Anwendung einer Produktionsregel bedeutet dann, in der bislang erzeugten Symbol-Folge (beim Start nur aus dem Startsymbol bestehend) eine Teilfolge zu suchen, die mit der linken Seite einer Produktionsregel übereinstimmt und diese durch die zugehörige rechte Seite zu ersetzen. Wenn eine Folge solcher Ersetzungen in einer Folge von Terminalsymbolen endet, so wird diese als (syntaktisch korrekter) Satz der Sprache bezeichnet. Terminalsymbole werden nicht ersetzt (siehe (3)).

Liegt also eine beliebige Folge von Terminalsymbolen vor, so kann auf grundsätzlich zwei Arten festgestellt werden, ob diese ein Satz der Sprache ist oder nicht:

- Ausgehend vom Startsymbol wird versucht, diese Folge abzuleiten (**top-down**). Dies ist i.a. weder eindeutig noch muß es überhaupt gelingen.
- Beginnend bei der vorgelegten Folge versucht man die Produktionsregeln "von rechts nach links" anzuwenden, mit dem Ziel, die Folge so auf das Startsymbol zu reduzieren (**bottom-up**). Auch dies ist i.a. weder eindeutig noch muß es überhaupt gelingen.

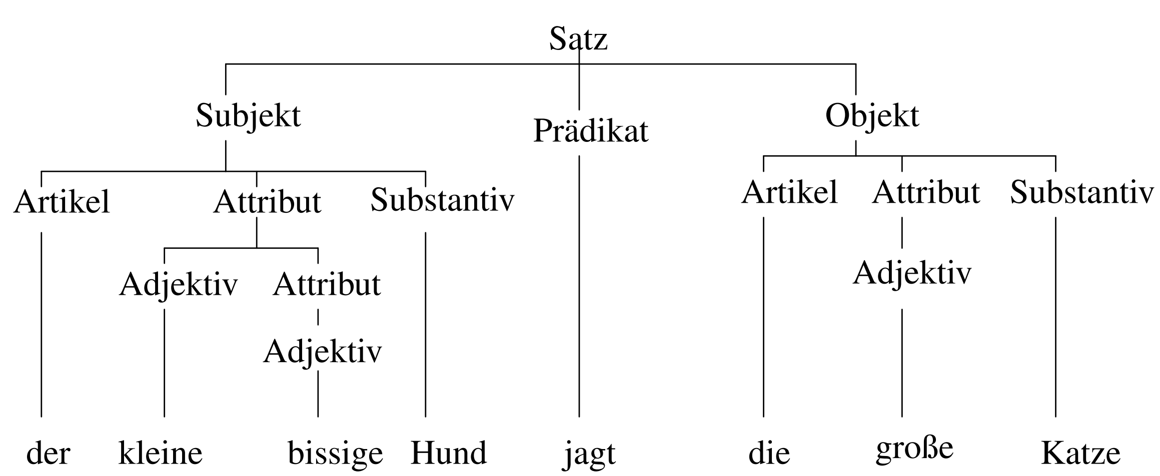
Beispiel (aus /Schöning92/):

Terminalsymbole	ε, der, die, das, kleine, bissige, große, Hund, Katze, jagt
Non-Terminalsymbole	Satz, Subjekt, Prädikat, Objekt, Artikel, Substantiv, Attribut, Adjektiv,
Startsymbol	Satz

Produktionsregeln:

Satz	→	Subjekt Prädikat Objekt
Subjekt	→	Artikel Attribut Substantiv
Artikel	→	ε
Artikel	→	der
Artikel	→	die
Artikel	→	das
Attribut	→	ε
Attribut	→	Adjektiv
Attribut	→	Adjektiv Attribut
Adjektiv	→	kleine
Adjektiv	→	bissige
Adjektiv	→	große
Substantiv	→	Hund
Substantiv	→	Katze
Prädikat	→	jagt
Objekt	→	Artikel Attribut Substantiv

Mit dieser Grammatik wird eine Sprache definiert, zu der der Satz *der kleine bissige Hund jagt die große Katze* gehört, da er mit folgenden Regelanwendungen abgeleitet werden kann:



In Anlehnung an die Notation der EBNF (s.o) läßt sich die Notation der Regeln etwas vereinfachen, in dem man zusätzliche Metasymbole einführt. Diese gehören streng genommen zu einer "Meta-Meta-Sprache", da sie dazu dienen, die Meta-Sprache zur Beschreibung der eigentlichen Sprache zu definieren:

- Kommen mehrere Regeln mit gleicher linker Seite vor, heißt dies *die eine, die andere kann angewendet werden* (im Sinne eines ODER); dafür schreibt man
  linke Seite → *rechte Seite*$_1$ **|** *rechte Seite*$_2$ **|** *rechte Seite*$_3$

Der senkrechte Strich **|** drückt also eine "alternative Ersetzung" aus und ist in obengenanntem Sinn ein "Meta-Meta-Symbol".

- Zwei Regeln der Form
  A → ε
  A → a
  bedeuten: A wird durch nichts (leeres Wort ε) oder durch a ersetzt
  die Ersetzung ist also **optional**; dies wird ausgedrückt durch
  A → [a]

In gleichem Sinn wie oben ist das Paar eckiger Klammern **[ ]** ein "Meta-Meta-Symbol".

- Vier Regeln der Form
  A → ax
  A → aBx
  B → b
  B → bB
  bedeuten: *A wird durch a, gefolgt von einer beliebige (auch leere) Folge von b's, gefolgt von x ersetzt;* dies wird ausgedrückt durch
  A → a { b } x

Auch das Paar der geschweiften Klammern **{ }** ist somit ein "Meta-Meta-Symbol".

Anhand der Art der Produktionsregeln lassen sich Sprachen klassifizieren, z.B. in einer sog. Chomsky-Hierarchie (nach /Bauer71/, /Bauer92/) wie folgt:

- **Typ 0 (Phasenstrukturgrammatik):**
  alle Produktionen sind von der Form
  a → b
  mit a nichtleere Folge von Non-Terminalsymbolen, b beliebige (auch leere) Folge

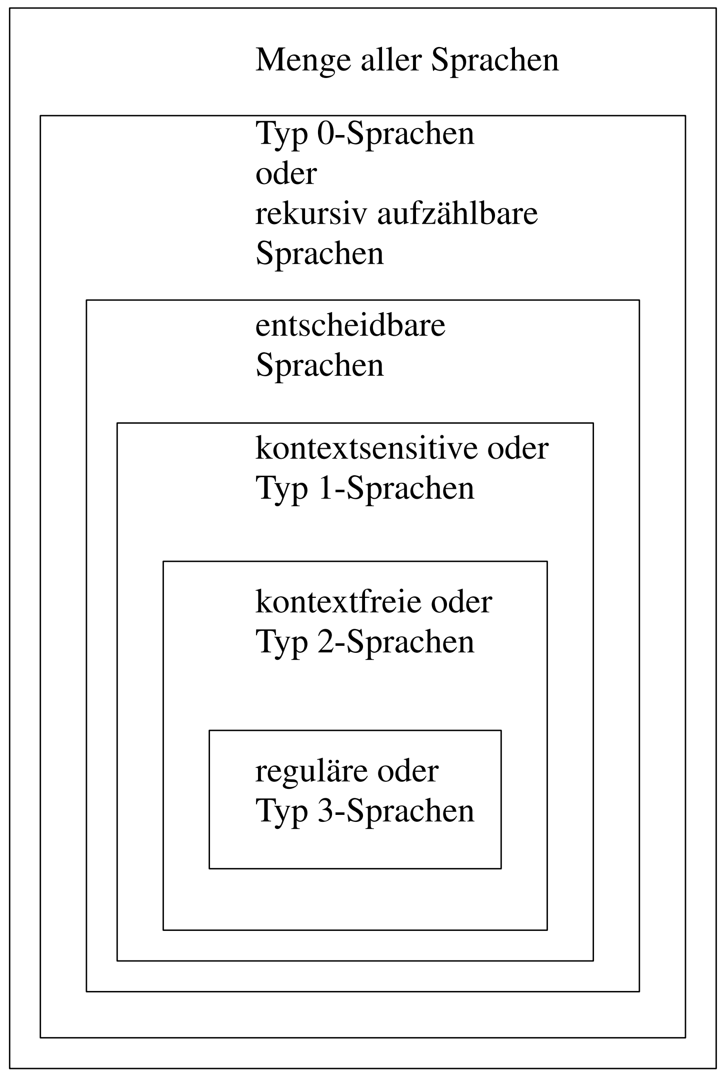
- **Typ 1 (kontextsensitiv):**
  Typ 0 - mit der Einschränkung, daß alle Regeln von der Form
  $uAv \rightarrow ubv$
  sind, mit u,v beliebige Folgen von Non-Terminalsymbolen, A ein Non-Terminalsymbol und b eine beliebige, aber nicht leere Symbolfolge
- **Typ 2 (kontextfrei):**
  Typ 1 - mit der Einschränkung, daß alle Regeln von der Form
  $X \rightarrow b$
  sind, mit X ein Non-Terminalsymbol und b eine nichtleere Folge von Symbolen
- **Typ 3 (regulär):**
  Typ 2 - mit der Einschränkung, daß alle Regeln von der Form
  $X \rightarrow bY$
  $X \rightarrow b$
  sind, mit X und Y je ein Non-Terminalsymbol, b eine Folge von Terminalsymbolen

Liegt nun eine Grammatik für eine Sprache vor, so muß bei Vorliegen einer Folge von Terminalsymbolen ("Wort", "Satz") entschieden werden, ob diese Folge zur so definierten Sprache gehört (syntaktisch korrekt ist) oder nicht. Dies wird i.a. als **Wortproblem** bezeichnet. Präziser kann diese Fragestellung wie folgt formuliert werden:

☞ Gibt es einen Algorithmus, der bei Eingabe der Grammatik und einer Folge von Terminalsymbolen in **endlicher Zeit** entscheidet, ob die Folge zur Sprache gehört oder nicht?

Satz: Das *Wortproblem* ist für Typ 1-, Typ 2-, Typ 3-Sprachen entscheidbar!

Das folgende Bild gibt einen Überblick über den Zusammenhang der Sprachen:



Anm.: Viele Programmiersprachen (z.B. Modula-2) werden mittels kontextfreier Grammatiken definiert, wobei die Menge der zugehörigen korrekten Programme eigentlich nicht kontextfrei sind - Bedingungen wie Typverträglichkeiten lassen sich nicht durch kontextfreie Grammatiken ausdrücken. Die Grammatik allein definiert die Korrektheit nicht. Mehr dazu in entsprechenden Fachbüchern (z.B. [Schöning92])

2. Reguläre Ausdrücke und formale Sprachen

Damit wären wir bei den regulären Sprachen. Was fehlt, ist der Zusammenhang zu regulären Ausdrücken:

Die Definition regulärer Ausdrücke erfolgt induktiv (siehe [Schöning92]):

- Die leere Menge ist ein regulärer Ausdruck
- Das leere Wort (ε) ist eine regulärer Ausdruck
- Jedes Terminalsymbol ist ein regulärer Ausdruck
- Sind α und β reguläre Ausdrücke, so auch αβ (Konkatenation), (α **|** β) (Auswahl) sowie **{** α **}** (Folge)
  Im letzten Fall verwendet man, wie wir das im Folgenden auch tun werden, die Schreibweise **(** α **)*** mit dem Meta-Symbol ***** zusätzlich zu den Meta-Symbolen **()**; damit können die geschweiften Klammern fortan wieder für Mengen verwendet werden.

Der Zusammenhang zu **regulären Sprachen** ergibt sich wie folgt ([Schöning92]):

Seien A, α, β reguläre Ausdrücke und bezeichne **L(A)** die von A definierte Sprache (Menge von korrekten Symbolfolgen); dann gilt:

- Ist A = { } (leere Menge), so ist L(A) = { }
- Ist A = ε (leeres Wort) so ist L(A) = {ε}
- Ist A = a (a Terminalsymbol), so ist L(A) = {a}
- Ist A = α β so ist L(A) = L(α) L(β)
  (das kartesische Produkt mit dem Konkatenationsoperator von L(α) und L(β))
- Ist A = (α | β) so ist L(A) = L(α) ∪ L(β)
- Ist A = (α)* , so ist L(A) = L(α)*

Beispiel: Durch den regulären Ausdruck (0|(0|1)*00) wird folgende Sprache beschrieben:

Vokabular = {0,1}
S → 0 | B.
B → C 0
C → D 0
D → ε
D → D 1
D → D 0

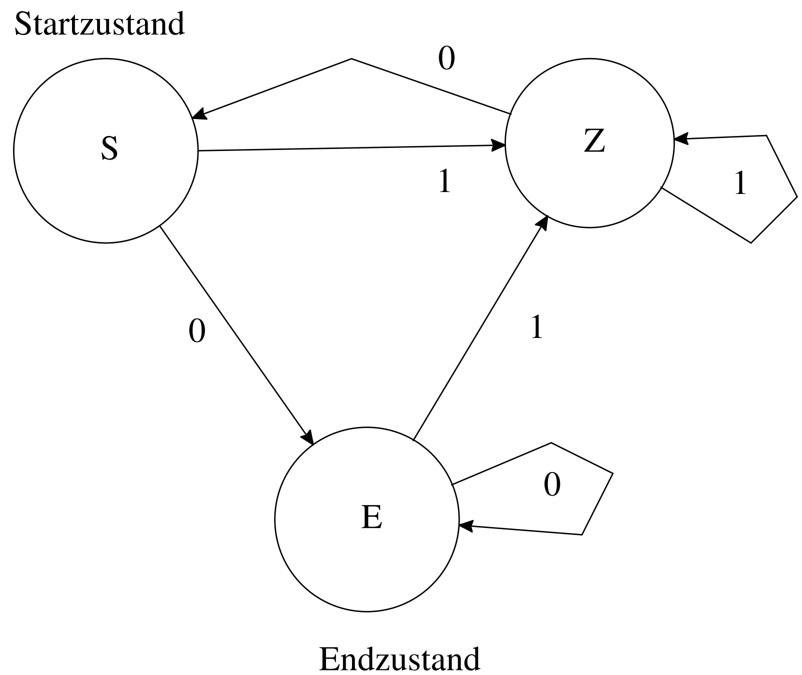
Mit obigen Regeln läßt sich die Sprache auch so ermitteln:

$$
\begin{aligned}
L(\ (0|(0|1)^*00)\ ) \quad &= \quad L(0) \cup L(\ (0|1)^*00\ ) \\
&= \quad L(0) \cup (\ L(\ (0|1)^*\ )\ L(0)\ L(0)\ ) \\
&= \quad L(0) \cup (\ L((0|1))^*\ L(0)\ L(0)\ ) \\
&= \quad L(0) \cup (\ (L(0) \cup L(1)\ )^*\ L(0)\ L(0)\ )
\end{aligned}
$$

Eine wichtige Eigenschaft regulärer Sprachen ist die, daß sie auch mittels **endlicher Automaten** beschrieben werden können. Es gilt:

Satz: Jede durch endliche Automaten erkennbare Sprache ist regulär (Typ 3) und zu jeder regulären Sprache gibt es einen endlichen Automaten, der diese Sprache erkennt.

Wir wollen die Theorie hier nicht weiter strapazieren - die Definition endlicher (deterministischer) Automaten kann z.B. [Schöning92] entnommen werden. Damit obiger Satz aber nicht zu leer im Raum steht, soll für das letzte Beispiel ein zugehöriger Automat angegeben werden.



Der Automat "liest" eine Folge von Eingabesymbolen (hier: 0 | 1), beginnend beim Startzustand und wechselt wie angegeben abhängig vom aktuell gelesenen Symbol den Zustand; ist er am Ende (Zeichenfolge vollständig gelesen) im End-Zustand, so ist die Symbolfolge korrekt.

Reguläre Ausdrücke werden wir im Folgenden u.a. dazu verwenden, aus irgendwelchen Texten (oder Datenbeständen) gewisse Zeichenfolgen herauszufiltern. Dazu werden die gewünschten Zeichenfolgen (z.B.: alle Worte, die mit einem A beginnen, dem ein E oder ein e folgt) mittels eines regulären Ausdrucks beschrieben ("A(E|e)"), wobei wir die Worttrennung (also Zeichenfolgen ohne Leerraum) hier einem "Tool" überlassen. Dieses "Tool" hat dann das *Wortproblem* zu lösen: für jedes Wort im Text muß festgestellt werden, ob es zu der durch diesen regulären Ausdruck definierten Sprache gehört oder nicht.

Abschließend wird die obige Notation für reguläre Ausdrücke so erweitert, wie sie die (meisten) unserer Tools verwenden. Wir stehen dabei insbesondere vor dem Problem, daß wir Zeichen sowohl als Meta-Zeichen wie auch als Terminal-Zeichen unterscheiden müssen. Ebenso wollen wir bei den Terminalsymbolen gewisse nicht-darstellbare Zeichen ("Escape-Sequenzen") haben. Schließlich ist die "ge-Oder-te" - Aufzählung von Terminalzeichen, die in der ASCII-Tabelle hintereinanderstehen, sehr mühsam - einfacher ist es, zu sagen, daß ein Zeichen aus einem Bereich gemeint ist.

Die folgende Tabelle gibt eine Übersicht:

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 Aufzählung 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	r1 oder r2
(r1)(r2)	r1 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

Metazeichen sind also: **() [] \ . ^ + * ?**

Ein Fragezeichen als Terminalsymbol muß demnach entweder als **\?** oder als **[?]** (Aufzählung) definiert werden. Ein *Backslash (\)* muß demnach als **** definiert werden. Die Zeichen **^ $** definieren abhängig von der Zielsetzung oder Eigenschaften des jeweiligen Tools Anfang / Ende einer Zeile oder eines Wortes oder ... (siehe spätere Ausführungen zu den Tools).

Form	Bedeutung
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tabulator
\ddd	ddd Oktalwert eines Zeichens (\007 für "bell")
\c	das Zeichen c als Terminalzeichen

Beispiele:

L(A*) = { ε, A, AA, AAA, ...}

L(AB*C) = { AC, ABC, ABBC, ABBBC, ... }

L(AB?C) = { AC, ABC }

L((AB)*C) = { C, ABC, ABABC, ... }

L([0-9]+) = { Folgen von Ziffern, mindestens jedoch eine }

L((\+|-)?[0-9]+\ .?[0-9]*) ⊃ { 2.3, +23, -23.001 }

L([+-]?[0-9]+[.]?[0-9]*) wie zuvor

L([A-Za-z_][A-Za-z_0-9]*): C-Identifier

Statt zu sagen, daß eine Zeichenfolge zu der von einem regulären Ausdruck definierten Sprache gehört, sagt man oft auch *die Zeichenfolge paßt zum Ausdruck ;* im Englischen wird dies auch als *the regular expression matches the string* augedrückt.

Unix-Tools

1. Übersicht

Ein Unix-System enthält eine ganze Reihe von Tools, die wir für unsere Zwecke sehr gut verwenden können. Teilweise waren sie in den vorherigen Abschnitten bereits verwendet worden - hier wollen wir sie einzeln und etwas ausführlicher besprechen. Aufgrund der Mächtigkeit des einen oder anderen Werkzeugs werden nicht alle Details angegeben; falls Sie mehr wissen wollen, so sollten Sie doch einmal das **man**-Kommando "bemühen".

Viele dieser Kommando's lesen im *default*-Fall von der Standardeingabe und schreiben in die Standardausgabe; sie dienen dazu, den Datenstrom aus der Standardeingabe nach gewissen Regeln zu verarbeiten und das Ergebnis dieser Verarbeitung auszugeben. Sie stellen somit eine Art **Filter** dar. Sehr häufig erfolgt ihre Anwendung in einer Pipeline.

Die folgende Tabelle gibt eine erste Übersicht:

Name	Anwendung
sort	nach gewissen Aspekten sortieren
uniq	doppelte Zeilen herausfinden
cut	eine Art *project*
paste	Dateien zeilenweise zusammenfügen
join	zwei Dateien bezüglich Wertgleichheit in Spalten verknüpfen
grep	nach einem String über einen eingeschränkten regulären Ausdruck suchen
egrep	wie *grep* doch volle reguläre Ausdrücke
fgrep	wie *grep* doch nur feste Strings als "Suchmuster"
sed	**s**treamline **ed**itor für nicht-interaktives Editieren (Filter)
tr	**tr**anslate - Zeichen ersetzen (Filter)

2. *cut*

Dieses Kommando ist immer dann nützlich, wenn es darum geht, aus einem Datenstrom "Bereiche" von Zeichen in Zeilen zu extrahieren. In unserem Zusammenhang ist es eine Art *project.*

Aufrufformen:

cut -c*charlist [filenames]*
cut -d*sep* **-f***colnumbers [filenames]*

Mit der Option **-c** kann angegeben werden, die "wievielten" Zeichen einer Zeile herausgefiltert werden sollen. Mit der Option **-d** kann ein Feldtrenner angegeben werden, die darauffolgende Option **-f** spezifiziert dann die Nummern der Spalten, die ausgegeben werden sollen.

Einige Beispiele sollen den Nutzen von *cut* demonstrieren:

- Wir wollen wissen, welcher Wochentag heute ist.

Die grundlegende Information hierzu liefert das Kommando **date**:

siehe auch **man date**

Der Wochentag erscheint hierbei in der üblichen englischen Abkürzung in den ersten drei Zeichen der Ausgabe von *date.* Mit *cut* können wir diese herausschneiden:

```
pey$ cat 1.scr
date | cut -c1-3
pey$ 1.scr
Tue
```

Da wir mit strukturierten Datensätzen arbeiten, insbesondere mit variabel langen Feldern innerhalb von Zeilen, die durch einen definierten Trenner separiert sind, ist mit dieser Option wenig anzufangen. Dazu verwenden wir die Optionen **-d** zur Festlegung des Trenners und **-f** zur Angabe der gewünschten Spaltennummern.

- Wir wollen aus unserer Datei *Person.all* die zweite und neunte Spalte *(Name, Tel)* herausschneiden - damit das Ergebnis nicht zu lang wird, beschränken wir uns auf die ersten 4 Zeilen (Kommando **head -4**):

```
ariadne$ head -4 Person.all | cut -d: -f2,9
Name:Tel
Biertaler:0815/10007
Oha:0815/10008
Koernigen:0815/10008
```

Zu erwähnen ist, daß als *default* des Feldtrenners der *tab* fungiert.

3. *sort*

Das Sortieren von Datensätzen dürfte zu den häufigsten Aufgaben im Zusammenhang mit Datenbanken gehören - wenn die Ausgabe mehr als ein Tupel umfaßt, wollen wir diese nach irgendeinem Kriterium sortiert sehen.

Das *sort*-Kommando hat eine Vielzahl von Möglichkeiten, die wir hier um den Rahmen nicht zu sprengen nicht im Detail behandeln wollen (Hinweis: **man sort**). Einige Beispiele mögen genügen. Dabei verwenden wir jeweils nur Teile unserer Dateien (i.w. die Datei *Person.all* projiziert auf die zweite Spalte: Datei *cut*), die wir diesmal mit dem Kommando **tail** von "hinten her abschneiden".

- Die letzten 5 Datensätze der Projektion von *Person.all* auf *Name* sortiert ausgeben:

```
# 1.scr
cut -d: -f2 Person.all | tail -5 | sort
```

- Mit der Option **-u** können wir Duplikate - also identische Zeilen in der Eingabe für *sort* unterdrücken.
- Mit der Option **-r** können wir absteigend sortiert ausgeben.
- Wenn wir das Ergebnis in einer Datei haben wollen, so können wir entweder die Standardausgabe in eine Datei umlenken oder mit der Option **-o** die Datei als Argument definieren.
- Wenn wir die Datei durch den sortierten Inhalt ersetzen wollen, so ist mit **sort datei > datei** nichts zu machen. Die *datei* wäre anschließend leer (Warum wohl?). Mit der Option **-o** geht es allerdings!
- Zahlen kann man natürlich auch sortieren - entweder nach alphabetischer Ordnung (was man bei Zahlen meist nicht will) oder numerisch mit der Option **-n**.
- Oft will man in gleicher Form sortieren wie dies z.B. bei Lexika's üblich ist: nur Buchstaben, Ziffern und Leerraum sind hierbei signifikant. Dies erledigt *sort* mit der Option **-d**.
- Soll in den Vergleich nicht die ganze Zeile einbezogen werden, sondern nur gewisse Felder, so kann mit der Option **-t** definiert werden, was als Feldtrenner gesehen werden soll. Der *default*-Feldtrenner ist *white space.*

- Will man etwa ab der ersten Spalte sortieren, also die erste Spalte nicht ins Sortierkriterium einbeziehen, so hilft die Option **+n** (in unserem Fall wäre n=1). *n* gibt die Zahl der Felder (Spalten), die vom Anfang her gesehen beim Sortieren **nicht** berücksichtigt werden:

```
ariadne$ cat zahlen
   5    -105
-105     105
   2      99
 115     -10
   3      25
 234      23
ariadne$ sort +1 zahlen
 234      23
   3      25
   2      99
 115     -10
-105     105
   5    -105
ariadne$ sort +1n zahlen
   5    -105
 115     -10
 234      23
   3      25
   2      99
-105     105
ariadne$ sort +1 -n zahlen
   5    -105
 115     -10
 234      23
   3      25
   2      99
-105     105
ariadne$
```

Man kann dies auch so interpretieren, daß die Felder von 0 an durchnumeriert sind und die Zahl angibt, ab welchem Feld sortiert werden soll.

- Das Gegenstück zum Letztgenannten ist, nur Felder bis zu einem bestimmten Feld in den Vergleich einzubeziehen. Dies kann mit der Option **-** gefolgt von einer Zahl geschehen. Dies macht nur Sinn in Kombination mit der Option **+**:

```
ariadne$ cat Person.part
4734:Mai:Karl
4734a:Mai:Anton
4735:Maerz:August
4736:Zank:Werner
4737:Schon:Gerda
4738:Mair:Josef
```

```
ariadne$ sort -t: +1 -2 Person.part
4735:Maerz:August
4734:Mai:Karl
4734a:Mai:Anton
4738:Mair:Josef
4737:Schon:Gerda
4736:Zank:Werner

ariadne$ sort -t: +1 Person.part
4735:Maerz:August
4734a:Mai:Anton
4734:Mai:Karl
4738:Mair:Josef
4737:Schon:Gerda
4736:Zank:Werner
ariadne$
```

Der Datensatz mit *PersId* = *4734a* wurde zu Demonstrationszwecken eingefügt.

4. *uniq*

Dieses Kommando dient i.w. dazu, hintereinanderstehende identische Zeilen in einer Datei herauszufinden. Einige Beispiele mögen dies erläutern:

```
ariadne$ cat names
Anton
Rudi
Astrid
Werner
Berta
Astrid
Astrid

ariadne$ uniq names names.u   #Ergebnis nach names.u

ariadne$ uniq names
Anton
Rudi
Astrid
Werner
Berta
Astrid

ariadne$ sort names | uniq
Anton
Astrid
Berta
Rudi
Werner
```

```
ariadne$ sort -u names   #letzteres kann sort alleine
Anton
Astrid
Berta
Rudi
Werner

ariadne$ sort names | uniq -d   #nur Duplikate ausgeben
Astrid

ariadne$ sort names | uniq -c   #Haeufigkeiten angeben
   1 Anton
   3 Astrid
   1 Berta
   1 Rudi
   1 Werner
ariadne$
```

5. *paste*

paste ist im Prinzip das Gegenstück zu *cut*: statt Zeilen "auseinanderzuschneiden" fügt es Zeilen aus verschiedenen Dateien zusammen.

Aufruf:

paste *filename1 filename2 ...*
paste -d*sep filename1 filename2*
paste -s *file*

Um die Standardeingabe anzusprechen, ist für *filename* ein **-** zu setzen. Das Zusammenfügen erfolgt zeilenweise, wobei die Zeilen jeweils durch ein *tab*-Zeichen getrennt werden. Mit der Option **-d** kann ein anderer Trenner definiert werden. Die Option **-s** sorgt dafür, daß die Zeilen einer Datei zusammengefügt werden (s.u.).

- Wir trennen unsere Datei *Person.all* (die ersten vier Zeilen) zuächst auf in zwei Dateien: *Namen* enthält jeweils *Name, Vorname* (Spalte 2 und 3), *Ort* enthält jeweils *Plz, Ort* (Spalte 7 und 8). Mit *paste können wir diese beiden Dateien wieder zusammenfügen:*

```
ariadne$ head -4 Person.all | cut -d: -f2,3 > Namen
ariadne$ head -4 Person.all | cut -d: -f7,8 > Ort

ariadne$ cat Namen
Name:Vorname
Biertaler:Peter J.
Oha:Alfred V.
Koernigen:Brian W

ariadne$ cat Ort
```

```
Plz:Ort
9876:Glockendorf
9876:Glockendorf
9876:Glockendorf

ariadne$ paste Namen Ort
Name:Vorname    Plz:Ort
Biertaler:Peter J.      9876:Glockendorf
Oha:Alfred V.   9876:Glockendorf
Koernigen:Brian W       9876:Glockendorf

ariadne$ paste -d+ Namen Ort
Name:Vorname+Plz:Ort
Biertaler:Peter J.+9876:Glockendorf
Oha:Alfred V.+9876:Glockendorf
Koernigen:Brian W+9876:Glockendorf
ariadne$
```

• Was passiert, wenn eine Datei mehr Zeilen als die andere hat?

```
ariadne$ head -6 Person.all | cut -d: -f9 > Tel

ariadne$ paste -d+ Namen Ort Tel

Name:Vorname+Plz:Ort+Tel
Biertaler:Peter J.+9876:Glockendorf+0815/10007
Oha:Alfred V.+9876:Glockendorf+0815/10008
Koernigen:Brian W+9876:Glockendorf+0815/10008
++0815/10009
++0731/5023572
ariadne$
```

Wie man sieht, wird entsprechend mit "Leerzeilen" aufgefüllt.

• Mit der Option **-s** können die Zeilen einer Datei aneinandergefügt werden:

```
ariadne$ head -4 Autor.all | paste -d+ -s -
PersId:BeitrId+4711:1234+4713:1234+4713:1235
```

6. *join*

Während *paste* ein "stupides" Zusammenfügen zweier oder mehrerer Dateien darstellt, ist *join* wirklich ein *join* von "Relationen", also eine Verknüpfung der Zeilen, die gewisse Eigenschaften gemeinsam haben.

Aufruf:

join *[options] file1 file2*

join verknüpft, wenn keine Optionen angegeben sind, die beiden Dateien *file1* und *file2* über Wertgleichheit in der jeweils ersten Spalte. Soll eine der Dateien die Standardeingabe sein, so ist als "Name" **-** anzugeben. Der *default*-Trenner ist eingabeseitig *white space* und ausgabeseitig ein Leerzeichen. Beide Dateien müssen bezüglich der Verbundspalte aufsteigend (nach ASCII) sortiert sein.

Die Optionen:

-a*n*	In Ergänzung zur normalen Ausgabe wird eine zusätzliche Zeile für die "nicht passenden" Zeilen aus *file1 (n=1)* resp. *file2 (n=2)* erzeugt.
-e *string*	Leere Ausgabefelder werden durch *string* ersetzt
-j*n m*	Verbund über die *m*-te Spalte von *file1 (n=1)* resp. *file2 (n=2).*
-t*sep*	Das Zeichen *sep* dient als Feldtrenner

Um dies zu erläutern verwenden wir zwei einfache Dateien, bei denen der jeweilige Effekt "ins Auge springt":

```
ariadne$ cat file1
a:1:x
a:2:y
b:3:z
a:4:x
b:5:z
ariadne$ cat file2
a:x:1
a:y:2
a:z:3
b:z:4
c:y:5
d:x:6
ariadne$
```

- Verbund über die jeweils erste Spalte - dazu müssen beide Dateien nach der ersten Spalte sortiert werden:

```
ariadne$ sort file1 > file1a
ariadne$ sort file2 > file2a
ariadne$ join -t: file1a file2a
a:1:x:x:1
a:1:x:y:2
a:1:x:z:3
a:2:y:x:1
a:2:y:y:2
a:2:y:z:3
a:4:x:x:1
a:4:x:y:2
a:4:x:z:3
b:3:z:z:4
b:5:z:z:4
ariadne$
```

- Dasselbe etwas modifiziert:

```
ariadne$ sort file2 > file2a
ariadne$ sort file1 | join -t: - file2a
a:1:x:x:1
a:1:x:y:2
a:1:x:z:3
a:2:y:x:1
a:2:y:y:2
a:2:y:z:3
a:4:x:x:1
a:4:x:y:2
a:4:x:z:3
b:3:z:z:4
b:5:z:z:4
ariadne$
```

- Verbund über die dritte Spalte von *file1* und über die zweite Spalte von *file2* (vorher sortieren):

```
ariadne$ sort -t: +2 file1 > file1a
ariadne$ sort -t: +1 -2 file2 > file2a
ariadne$ cat file1a
a:1:x
a:4:x
a:2:y
b:3:z
b:5:z
ariadne$ cat file2a
a:x:1
d:x:6
a:y:2
c:y:5
a:z:3
b:z:4
ariadne$ join -t: -j1 3 -j2 2 file1a file2a
```

```
x:a:1:a:1
x:a:1:d:6
x:a:4:a:1
x:a:4:d:6
y:a:2:a:2
y:a:2:c:5
z:b:3:a:3
z:b:3:b:4
z:b:5:a:3
z:b:5:b:4
ariadne$
```

7. *grep*-Familie

grep, egrep und *fgrep* durchsuchen die angegebenen Dateien (oder die Standardeingabe) nach dem Vorkommen des ebenfalls angegebenen Strings resp. nach dem Vorkommen einer Zeichenfolge, die zu dem angegebenen regulären Ausdruck "paßt". Jede Zeile, die eine entsprechende Zeichenfolge enthält, wird an die Standardausgabe kopiert.

grep erlaubt in etwas eingeschränkter Form reguläre Ausdrücke wie das später zu behandelnde Tool *sed.*

egrep erlaubt volle reguläre Ausdrücke

fgrep erlaubt nur die Angabe fester Zeichenfolgen.

Bei der Angabe von regulären Ausdrücken ist darauf zu achten, daß gewisse Meta-Symbole auch für die Shell Bedeutung haben - eine entsprechende Quotierung ist angebracht! Die Zeilenlänge bei *grep* ist beschränkt (1024), zu lange Zeilen werden "abgeschnitten".

Die folgende Tabelle gibt einige Optionen an (mehr: **man grep**):

-c	Anzahl der "Treffer" ausgeben und nicht die jeweiligen Zeilen
-i	ignoriere Groß-/Kleinschreibung
-l	Nur die Namen der Dateien ausgeben, die einen "Treffer" enthalten
-n	vor jeder Zeile die relative Zeilennummer ausgeben
-v	die Zeilen suchen, die keinen "Treffer" enthalten
-f *file*	*egrep*: *file* enthält den regulären Ausdruck *fgrep*: *file* enthält durch *newline* getrennt die zu suchenden Zeichenfolgen
-e *expression*	Nützlich, wenn ein Suchbegriff mit einem **-** beginnt

Die folgenden Beispiele sollen das eine oder andere demonstrieren:

- Die Namen aller Professoren aus *Person.all*:

```
ariadne$ grep Prof Person.all | cut -d: -f2,4
Rahn:Herr Prof.
Schweiggert:Herr Prof. Dr.
Stange:Frau Prof. Dr.
```

oder:

```
pey$ egrep 'Prof\.' Person.all | cut -d: -f2,4
Rahn:Herr Prof.
Schweiggert:Herr Prof. Dr.
Stange:Frau Prof. Dr.
pey$
```

- In welchen Dateien kommt die *PersId* vor?

```
pey$ grep -l PersId $HOME/udb1/relDb/*.all
/home/swg/udb1/relDb/Aktiv.data
/home/swg/udb1/relDb/Autor.data
/home/swg/udb1/relDb/Person.data
```

- Bei welchen Personen in *Person.data* ist keine Telefonnummer eingetragen?

```
pey$ egrep -v '[0-9]+$' $HOME/udb1/relDb/Person.data
PersId:Name:Vorname:Anrede:Institution:Strasse:Plz:Ort:Tel
4719:Ruebe:Berta:Frau Dr.:XSoft AG:Hahnweg 13:8000:Muenchen 21:
4725:Meier:Ulf:Herr::Kohlring 3:2000:Hamburg 4:
4728:Stange:Inge:Frau Prof. Dr.:Techn. Uni.:Postfach 1240:3010:Magdeburg:
4729:Enter:Jo:Herr Dr.:ElektroBank AG:Postfach 131313:8000:Muenchen 60:
4731:Streit:Max:Herr::Eselsberg 4:7900:Ulm:
pey$
```

- Bei welchen Personen aus *Person.data* fehlt bei mindestens einer Spalte der Eintrag?
  Wenn in der ersten Spalte - so beginnt die Zeile mit Leerraum gefolgt von einem Doppelpunkt
  Wenn in der letzten Spalte - so endet die Zeile mit einem Doppelpunkt gefolgt evt. von Leerraum
  Ansonsten: zwei aufeinanderfolgende Doppelpunkte evt. mit Leeraum dazwischen

  Den zugehörigen regulären Ausdruck schreiben wir in eine Datei:

```
pey$ cat 5.pattern
(^[ \t]*:)|(:[ \t]*:)|(:[ \t]*$)
pey$ egrep -f 5.pattern $HOME/udb1/relDb/Person.data
4719:Ruebe:Berta:Frau Dr.:XSoft AG:Hahnweg 13:8000:Muenchen 21:
4721:Rot:Ulf:Herr::Sieberweg 15:7030:Boeblingen:07031/98765
4725:Meier:Ulf:Herr::Kohlring 3:2000:Hamburg 4:
4728:Stange:Inge:Frau Prof. Dr.:Techn. Uni.:Postfach 1240:3010:Magdeburg:
4729:Enter:Jo:Herr Dr.:ElektroBank AG:Postfach 131313:8000:Muenchen 60:
4731:Streit:Max:Herr::Eselsberg 4:7900:Ulm:
4732:Dann:Else:Frau::Bugstr. 3:4500:Osnabrueck:0541/7720969
```

```
4734:Mai:Karl:Herr Dr.::Finkenweg 19:7900:Ulm:0731/12534
pey$
```

Es sollte klar sein, daß mit diesen Tools ohne Rücksicht auf die Satzstruktur gesucht wird. Sollte man also z.B. nach einem Namen suchen, der gleichzeitig auch als Ortsbezeichnung vorkommt, so wird auch diese Zeile mit ausgewählt - außer man konstruiert ein entsprechendes Muster für *egrep*.

8. *sed*

Das in diesem Kapitel angesprochene Tool werden wir zwar selten benötigen, es kann aber trotzdem sehr nützlich sein - zum Beispiel bei

- Editieren von Dateien, die zu groß sind für interaktives Editieren
- Editieren von Dateien mit einer vorab bekannten Folge von Editierbefehlen (interaktiv zwar möglich, aber vielleicht etwas mühsam)
- effizientem Durchführen mehrfacher globaler Editierbefehle in einem Durchgang

Der *sed* (**s**treamline **ed**itor) kopiert im *default*-Fall die Standardeingabe zeilenweise in einen internen Puffer (gewisse Ausnahmen sind möglich), führt dort ggf. Editierbefehle aus und schreibt diesen Puffer dann auf die Standardausgabe. Er wird wie folgt aufgerufen:

sed *[* **-n***] [* **-e** *cmd] [* **-f** *cmdfile] [files]*

Die Optionen:

-n	nur die Zeilen ausgegeben, die explizit mit dem **p**rint-Befehl ausgewählt wurden
-e *cmd*	*cmd ist ein Editierkommando (s.u.)*
-f *cmdfile*	*cmdfile* enthält die Editierbefehle

Ist keine **-f** Option gesetzt und nur ein Editier-Kommando *cmd* enthalten, so kann vor diesem das **-e** entfallen.

Das Format eines *cmd's* sieht wie folgt aus:

*[addr [***,** *addr]] function [arguments]*

Alle Editier-Anweisungen werden übersetzt und sequentiell in der angegebenen Reihenfolge angewandt - der Input einer Anweisung ist der Output der vorangehenden Anweisung.

Die Auswahl der zu editierenden Zeilen erfolgt über die Adreßangaben *addr.* Dies sind entweder **Zeilennummern** oder in **Schrägstriche** (**//**) eingeschlossene reguläre Ausdrücke.

- Ist keine Adresse angegeben, so wird die zugehörige Funktion auf jede Zeile angewandt.
- Ist nur eine Adresse angegeben, so wird die Funktion auf die dadurch ausgewählten Zeilen angewandt.
- Zwei Adressen bestimmen einen Bereich .

Da die Metasymbole etwas anders definiert sind als wir dies zuvor dargestellt haben, werden sie in folgender Tabelle explizit dargestellt:

^	Zeilenanfang
$	Zeilenende
\n	ein "eingebettetes" *newline,* nicht das abschließende *newline* (wird hier nicht weiter benötigt und damit auch nicht näher beschrieben)
	beliebiges Zeichen, nicht jedoch das abschließende *newline*
*	wie bisher
\{d$_1$, d$_2$\}	Häufikeit des Wiederholens des vorangegangenen Ausdrucks; die erste Ziffer sagt "mindestens so oft", die zweite "höchstens so oft"
[...]	wie bisher
[^...]	wie bisher
\(...\)	zum "Zwischenspeichern" der entsprechenden Zeichenfolge in einem Register (numeriert von 0 bis 9) zur späteren Wiederverwendung
d	*d* eine Ziffer (Zahl von 0 bis 9), um den Inhalt eines Registers anzusprechen (siehe obige Zeile)
//	Der "leere" Ausdruck steht für den zuletzt übersetzten regulären Ausdruck

Funktionen

sed kennt eine Vielzahl von Funktionen, die wir hier alle im Einzelnen nicht darstellen wollen. Einige davon sind:

Kommando	Erläuterung	max. Adressen
d	**d**elete: Löschen der Zeile und nächste Zeile lesen	2
n	**n**ext: Aktuelle Zeile ausgeben und nächste Zeile lesen, weiter bei den auf **n** folgenden Editierkommandos	2
a *text*	**a**ppend: Anfügen von *text* an die ausgewählte Zeile; *newline's* in *text* müssen durch einen Backslash geschützt werden; Ende von *text* ist das erste ungeschützte *newline;* nach der Ausführung von **a** wird die Zeile mit dem angefügten *text* ausgegeben	1
i *text*	**i**nsert: Einfügen von *text* bevor die ausgewählte Zeile ausgegeben wird; Behandlung von *newline* wie bei **a**	1
c *text*	**c**hange: Ausgewählte Zeile(n) löschen und durch *text* ersetzen. Wird ein Bereich ausgewählt, so werden die entsprechenden Zeilen alle gelöscht, aber nur eine Kopie von *text* ausgegeben. Behandlung von *newline* wie bei **a**	2
p	**p**rint: Ausgeben ohne evt. folgende Kommandos auszuführen	2
w *filename*	Ausgabe der ausgewählten Zeile(n) in die angegebene Datei; zwischen **w** und *filename* steht genau ein Leerzeichen! Existiert die angegebene Datei, wird sie überschrieben! Folgende Editier-Kommandos wirken nicht!	2
r *filename*	**r**ead: Inhalt der Datei *filename* wird gelesen und an die ausgewählte Zeile angefügt. Wieder genau ein Leerzeichen zwischen **r** und *filename!*	1

Einige einfache Beispiele mögen dies erläutern.

- Alle Beiträge zum Thema "UNIX" auswählen:

```
pey$ sed -n /U[Nn][Ii][Xx]/p Beitrag.data

1201:UNIX in der Kommunalverwaltung:12.1.93:+
1203:CASE und die klassischen UNIX-Tools:01.01.93:+
1205:Verteilte Datenbanken in einer homogen UNIX-Umgebung:12.2.93:-
1210:UNIX - und die Software wird besser?:14.1.93:-
1211:Evolutionaere Software-Entwicklung mit UNIX-Tools:10.1.93:+
1212:Was kommt nach UNIX? Ein Blick ins Jahr 2000:16.1.93:-
1235:UNIX - Wohin geht die Reise?:24.12.92:-
pey$
```

- Alle Personen auswählen, deren Name aus genau drei Buchstaben besteht:

```
ariadne$ sed -n '/^[0-9]*:[A-Z][a-z]\{2,2\}:/p' Person.all
4712:Oha:Alfred V.:Herr:BS&S:Usambaraweg 11:9876:Glockendorf:0815/10008
4721:Rot:Ulf:Herr::Sieberweg 15:7030:Boeblingen:07031/98765
4727:Abs:Karl:Herr:Softikus AG:Auweg 3:7918:Au:07303/5857
4730:Emu:F.J.:Herr Dr.h.c.:SSS e.V.:Straussring 3:3000:Hannover 1:0511/4321645
4734:Mai:Karl:Herr Dr.::Finkenweg 19:7900:Ulm:0731/12534
9999:Ulm:Maria:Frau::Pfauengasse 12:8000: Muenchen 1:089/123456
```

- Nach dem Datensatz mit *PersId = 4738* einen neuen Datensatz einfügen:

```
pey$ cat 3.script
/^4738/a\
8888:Test:Anton:Dr.:::::9999/88888

pey$ sed -f 3.script $HOME/udb1/relDb/Person.data | tail -4

4736:Zank:Werner:Herr Dr.:Softikus AG:Auweg 3:7918:Au:0731/5857
4737:Schon:Gerda:Frau:Softness GmbH:Einsteinring:8000:Muenchen 20:089/980321
4738:Mair:Josef:Herr:IEJ VaG:Virenweg 17:8500:Nuernberg 80:0911/5112323
8888:Test:Anton:Dr.:::::9999/88888
pey$
```

Eine Funktion soll speziell herausgegriffen werden: **Substitution** (Ändern von Zeilen). Diese hat folgende Grundstruktur:

s/*muster***/***ersatztext***/***flags*

Hierbei können maximal zwei Adressen zur Auswahl von Zeilen angegeben werden. Das Muster kann genauso aufgebaut sein wie in den Adreßdefinitionen mit dem Unterschied, daß die Begrenzer (s.o.: **/**) auch beliebige Zeichen sein dürfen (i.a. wird man dennoch die Schrägstriche dafür nehmen). Die Ersetzung findet im *default*-Fall nur beim ersten "Treffer" statt. Mit dem *flag* **g**(lobal) werden alle nicht überlappenden "Treffer" durch den Ersatztext ersetzt; der ersetzte Text selbst wird nicht weiter bearbeitet.

Der Ersatztext ist kein regulärer Ausdruck, d.h. die Meta-Symbole der regulären Ausdrücke haben hier keine spezielle Bedeutung. Dafür gibt es andere Zeichen mit Spezialbedeutung. Diese sind:

& wird ersetzt durch die Zeichenfolge, die zu der von *muster* definierten Sprache gehört

\d **d** ist eine Ziffer; **\d** wird ersetzt durch den Inhalt des **d**-ten Registers, der durch das Konstrukt **\(...\)** im *muster* "gefüllt" wurde.

Einige Beispiele mögen dies näher erläutern:

- Das Wort "UNIX" für die *troff*-Verarbeitung fettgedruckt setzen:

```
pey$ cat 4.script
# Den Treffer selbst ansprechen:
# "UNIX" fuer troff fettgedruckt

s/U[Nn][Ii][Xx]/\\fB&\\fR/gp

pey$ sed -n -f 4.script $HOME/udb1/relDb/Beitrag.data

1201:\fBUNIX\fR in der Kommunalverwaltung:12.1.93:+
1203:CASE und die klassischen \fBUNIX\fR-Tools:01.01.93:+
1205:Verteilte Datenbanken in einer homogen \fBUNIX\fR-Umgebung:12.2.93:-
1210:\fBUNIX\fR - und die Software wird besser?:14.1.93:-
1211:Evolutionaere Software-Entwicklung mit \fBUNIX\fR-Tools:10.1.93:+
1212:Was kommt nach \fBUNIX\fR? Ein Blick ins Jahr 2000:16.1.93:-
1235:\fBUNIX\fR - Wohin geht die Reise?:24.12.92:-
pey$
```

- Vertauschen der Spalten in *Autor.all* (nur die ersten drei ausgeben)

```
pey$ sed -n '1,3s/\(.*\):\(.*\)/\2|\1/p' $HOME/udb1/relDb/Autor.data
BeitrId|PersId
1234|4711
1234|4713
pey$
```

- Die letzten 20 Zeichen der ersten drei Zeilen in *Person.all* löschen (durch "nichts" ersetzen):

```
pey$ sed -n '1,3s/.\{20\}$//w Pers.kurz' $HOME/udb1/relDb/Person.data
pey$ cat Pers.kurz
PersId:Name:Vorname:Anrede:Institution
4711:Biertaler:Peter J.:Herr:BS&S:Usambaraweg 11:9876:Gl
4712:Oha:Alfred V.:Herr:BS&S:Usambaraweg 11:9876:Gl
pey$
```

In alle Spalten, die keinen Eintrag haben, fünf "X" zur Markierung eintragen:

```
pey$ cat 7.script
1,$s/:[ \t]*:/:XXXXX:/gp
1,$s/^[ \t]*:/XXXXX:/p
1,$s/:[ \t]*$/:XXXXX/p

pey$ sed -n -f 7.script $HOME/udb1/relDb/Person.data

4719:Ruebe:Berta:Frau Dr.:XSoft AG:Hahnweg 13:8000:Muenchen 21:XXXXX
4721:Rot:Ulf:Herr:XXXXX:Sieberweg 15:7030:Boeblingen:07031/98765
4725:Meier:Ulf:Herr:XXXXX:Kohlring 3:2000:Hamburg 4:
4725:Meier:Ulf:Herr:XXXXX:Kohlring 3:2000:Hamburg 4:XXXXX
4728:Stange:Inge:Frau Prof. Dr.:Techn. Uni.:Postfach 1240:3010:Magdeburg:XXXXX
4729:Enter:Jo:Herr Dr.:ElektroBank AG:Postfach 131313:8000:Muenchen 60:XXXXX
4731:Streit:Max:Herr:XXXXX:Eselsberg 4:7900:Ulm:
4731:Streit:Max:Herr:XXXXX:Eselsberg 4:7900:Ulm:XXXXX
4732:Dann:Else:Frau:XXXXX:Bugstr. 3:4500:Osnabrueck:0541/7720969
4734:Mai:Karl:Herr Dr.:XXXXX:Finkenweg 19:7900:Ulm:0731/12534
```

Hier sieht man sehr deutlich die Wirkungsweise des *sed* wie auch der **p**rint-Anweisung.

- Modifikation so, daß jede geänderte Zeile nur einmal ausgegeben wird:

```
pey$ cat 8.script
1,$s/:[ \t]*:/:XXXXX:/g
1,$s/^[ \t]*:/XXXXX:/
1,$s/:[ \t]*$/:XXXXX/
/XXXXX/p

pey$ sed -n -f 8.script $HOME/udb1/relDb/Person.data
4719:Ruebe:Berta:Frau Dr.:XSoft AG:Hahnweg 13:8000:Muenchen 21:XXXXX
4721:Rot:Ulf:Herr:XXXXX:Sieberweg 15:7030:Boeblingen:07031/98765
4725:Meier:Ulf:Herr:XXXXX:Kohlring 3:2000:Hamburg 4:XXXXX
4728:Stange:Inge:Frau Prof. Dr.:Techn. Uni.:Postfach 1240:3010:Magdeburg:XXXXX
4729:Enter:Jo:Herr Dr.:ElektroBank AG:Postfach 131313:8000:Muenchen 60:XXXXX
4731:Streit:Max:Herr:XXXXX:Eselsberg 4:7900:Ulm:XXXXX
4732:Dann:Else:Frau:XXXXX:Bugstr. 3:4500:Osnabrueck:0541/7720969
4734:Mai:Karl:Herr Dr.:XXXXX:Finkenweg 19:7900:Ulm:0731/12534
XXXXX:Ulm:Maria:Frau:XXXXX:Pfauengasse 12:8000: Muenchen 1:089/123456
pey$
```

Dies sollte einen Eindruck der Leistungsfähigkeit des *sed* vermitteln; es wurde einiges nicht dargestellt, der geneigte Leser, sofern seine Neugier hierdurch geweckt, kann über das **man**-Kommando leicht mehr erfahren.

9. *tr*

tr dient zum "Übersetzen" von Zeichen. Das generelle Format lautet:

tr *from_chars to_chars*

Ersetzen einzelner Zeichen durch andere kann auch *sed.* Anders ist es beim Umsetzen ganzer Bereiche:

- Alle Namen in Großbuchstaben umsetzen

```
pey$ head -5 $HOME/udb1/relDb/Person.data | cut -d: -f2 | tr '[a-z]' '[A-Z]'
NAME
BIERTALER
OHA
KOERNIGEN
REICHLE
pey$
```

- Die Namen kodieren durch eine zyklische Verschiebung der Buchstaben:

```
pey$ head -5 $HOME/udb1/relDb/Person.data | cut -d: -f2  #vorher
Name
Biertaler
Oha
Koernigen
Reichle

pey$ head -5 $HOME/udb1/relDb/Person.data | cut -d: -f2 | \
>  tr '[A-Y]Z' '[B-Z]A' | tr '[a-y]z' '[b-z]a'
Obnf
Cjfsubmfs
Pib
Lpfsojhfo
Sfjdimf
pey$
```

- Leerzeichen in *newline's* umsetzen:

```
pey$ date | tr ' ' '\12' | head -3
Wed
Jul
23
pey$
```

- Mit der Option **-s** können bei der "Übersetzung" entstandene Zeichenduplikate unterdrückt werden: alle Doppelpunkte in *newline's* umsetzen, mehrfache *newline's* unterdrücken:

```
pey$ tail -1 $HOME/udb1/relDb/Person.data | tr -s : '\12'
4738
Mair
Josef
Herr
IEJ VaG
Virenweg 17
8500
```

```
Nuernberg 80
0911/5112323
pey$
```

- Mit der Option **-d** können einzelne Zeichen im Eingabedatenstrom gelöscht werden: alle Leerzeichen entfernen

10. Ausblick

Die hier dargestellten Tools sollen für unsere Zwecke zunächst genügen. Natürlich gibt es viele weitere sehr nützliche Hilfsmittel:

- Textverarbeitung zur Reportgestaltung: **troff**, **tex**, **tbl**, **grap**,...
- Oberflächengestaltung: **Tcl**, **Tk**
- u.v.m

Shell-Programmierung

1. Shell-Skripte

Beispiel:

```
pey$ cat status
#
# status -- prints: date, number of user's logged on,
#                and current working directory
#

echo The current date and time are:
date            # Datum, Zeit

echo
echo The number of users on the system is:
who | wc -l

echo
echo The current working directory is:
pwd
pey$ status
The current date and time are:
Wed Jul 23 09:23:15 MEST 1997

The number of users on the system is:
      3

The current working directory is:
/home/swg/udb1/shprog/10
pey$
```

2. Shell-Variablen

2.1 Benutzer-definiert

variable_name = (letter | "_"){ letter | "_" | digit }

assignment = variable_name"="value

- Kein Leerzeichen vor und nach dem "=" !!!
- *value* wird grundsätzlich als Zeichenfolge gesehen, keine Datentypen !!!
- Der Wert einer Shellvariablen wird durch Voranstellen des Dollarzeichens **$** angesprochen (s.u.)

Beispiele:

```
pey$ cat 1.scr
anzahl=12
echo Es sind $anzahl Flaschen im Raum!
wert1=10
wert2=wert1
echo $wert2
wert2=$wert1
echo $wert2
echo $wert2
x=*
echo $x
pey$ 1.scr
Es sind 12 Flaschen im Raum!
wert1
10
10
1.scr status status.se x
pey$
```

Abarbeitung der Kommandozeile *echo $x* im letzten Beispiel:

1. Für die Variable x wird der Wert, also *, gesetzt
2. Die Zeile wird erneut berarbeitet, der * wird durch alle Dateinamen ersetzt
3. Ausführung von *echo* wird veranlaßt, als Argument wird die Liste der Dateinamen übergeben

```
pey$ cat 2.scr
cmd=cut
option1=-d:
option2=-f2,3
file=$HOME/udb1/relDb/Person.data
cmd1=head
option3=-3
$cmd $option1 $option2 $file | $cmd1 $option3
```

```
pey$ 2.scr
Name:Vorname
Biertaler:Peter J.
Oha:Alfred V.
pey$
```

Mehrere Worte:

```
pey$ cat 3.scr
string="Hier sind Blanks      und tabs         ."
echo $string
string='Hier sind Blanks    und tabs           .'
echo $string

pey$ 3.scr
Hier sind Blanks und tabs .
Hier sind Blanks und tabs .
pey$
```

2.2 Das *${variable}*-Konstrukt

```
pey$ cat 4.scr
file=1.se
ls $file
ls $file1
ls ${file}1
pey$ 4.scr
1.se
1.scr 1.se 2.scr 2.se 3.scr 3.se
4.scr 4.se status status.se x
ls: 1.se1: No such file or directory (Anm.: stderr)
pey$
```

Was ist hier passiert? Die Folge von Buchstaben und Ziffern nach dem **$** wird als Variablenname interpretiert; *file1* hat keinen Wert (*null value*), d.h. die Shell startet das *ls*-Kommando **ohne** Argumente!

Die geschweiften Klammern sind nur notwendig, wenn nach dem Variablenname ein Buchstaben, eine Ziffer oder ein Underscore folgt !

2.3 Arithmetik mit Shell-Variablen - *expr*

Im Gegensatz zu höheren Programmiersprachen kennt die Shell keine Datentypen. Werte von Shell-Variablen sind Folgen von Zeichen. Die Shell selbst bietet keine Möglichkeiten, Arithmetik mit Zahl-Werten zu betreiben. Dazu gibt es ein spezielles Programm: **expr**.

Beispiele:

```
pey$ cat 5.scr
i=1
i=$i+1
echo $i
expr $i + 1
i=1
expr $i + 1
expr 1 + 2
expr 1+2
pey$ 5.scr
1+1
expr: non-numeric argument (Anm.: stderr)
2
3
1+2
pey$
```

expr erwartet jeden Operator und jeden Operanden (ganze Zahlen) als eigenes Argument, deshalb also die Wirkung des letzten Beispiels oben.

expr erkennt folgende Operatoren: **+** für Addition, **-** für Subtraktion, ***** für die Multiplikation (Vorsicht: die Shell sieht * auch!), **/** für die ganzzahlige Division und **%** für die Restbildung. Vorrang wie üblich: Punkt vor Strich !

```
pey$ cat 6.scr
expr 5 * 3
expr 5 \* 3
expr "5 * 3"
i=1
i=`expr $i + 1`
echo $i
pey$ 6.scr
expr: syntax error (Anm.: stderr)
15
5 * 3
2
pey$
```

expr kann noch ein bißchen mehr, doch das bisherige reicht für die folgenden Aufgaben (näheres ist im Manual zu finden; Stichwort: der **:**-Operator).

3. Argument-Übergabe

3.1 Positionsvariable

Die Argumente eines Shell-Programmes werden von der Shell automatisch in speziellen Variablen gespeichert: das erste Argument in 1, das zweite in 2, u.s.w. Diese Variable werden auch als Positions-Parameter bezeichnet. Die Wert-Zuweisung erfolgt, nachdem die Shell ihre übliche Kommando-Zeilen-Bearbeitung (I/O-Umlenkung, Variablen-Substitution, Dateinamen-Substitution) erledigt hat.

```
pey$ cat telnr
cut -d: -f2,3,9 $HOME/udb1/relDb/Person.data | tr ':' ' ' | grep $1
pey$ telnr Schweiggert
Schweiggert Franz 0731/5023570
pey$
```

Im Positions-Parameter 0 wird der Kommando-Name abgelegt, 1 bis 9 enthalten die ersten 9 Argumente in angegebener Reihenfolge. Die Variable **$#** gibt die Anzahl der Argumente auf der Kommandozeile:

```
pey$ cat argtest
echo $# Argument\(e\) uebergeben
echo 1.Arg. = :$1:
echo 2.Arg. = :$2:
echo 3.Arg. = :$3:
echo Kommandoname = :$0:
pey$ chmod +x argtest

pey$ argtest
0 Argument(e) uebergeben
1.Arg. = ::
2.Arg. = ::
3.Arg. = ::
Kommandoname = :./argtest:
pey$ argtest ab c d e
4 Argument(e) uebergeben
1.Arg. = :ab:
2.Arg. = :c:
3.Arg. = :d:
Kommandoname = :./argtest:
pey$ argtest "a b c" d
2 Argument(e) uebergeben
1.Arg. = :a b c:
2.Arg. = :d:
3.Arg. = ::
ommandoname = :./argtest:
pey$
```

3.2 Die *$*-Variable*

Diese Variable bezieht sich auf *alle* übergebenen Argumente. Dies ist dann nützlich, wenn eine unbestimmte oder variable Anzahl von Argumenten behandelt werden soll:

```
pey$ cat argtest1
echo $# Argumente wurden uebergeben
echo Dies sind :$*:
pey$ chmod +x argtest1
pey$ argtest1 a b c d e f g h i j k l m n
14 Argumente wurden uebergeben
Dies sind :a b c d e f g h i j k l m n:
pey$
```

Das obige Beispielprogramm *telnr* hat noch einen kleinen Nachteil:

```
pey$ telnr Schweiggert
Schweiggert Franz 0731/5023570
pey$ telnr "Schweiggert Franz"
grep: Franz: No such file or directory
pey$
```

Eine kleine Änderung von *telnr:*

```
pey$ cat telnr1
cut -d: -f2,3,9 $HOME/udb1/relDb/Person.data | tr ':' ' ' | grep "$1"
pey$ telnr1 Mai
Maier Urs 0421/442812
Mai Karl 0731/12534
Mair Josef 0911/5112323
pey$ telnr1 "Mai Karl"
Mai Karl 0731/12534
pey$
```

☞ Um die folgenden Beispiele etwas einfacher zu halten, "verkürzen" wir unsere Datei *Person.data* auf die Attribute *Name, Vorname, Tel* und nennen diese *Person.tels* ("Telefonbuch"). Ebenso wird obiges Programm *telnr* darauf zurechtgemacht.

- Beispiel: Jemanden ins Telefonbuch aufnehmen

```
pey$ cat telnr2
cat $HOME/udb1/relDb/Person.tels | tr ':' ' ' | grep "$1"

pey$ cat add1
#
# Name, Vorname, Tel in Person.tels aufnehmen
#
echo "$1:$2:$3" >> $HOME/udb1/relDb/Person.tels

pey$ add1 Borchert Andreas 3572
pey$ telnr2 Borchert
```

```
Borchert Andreas 3572

pey$ add1 Grabert Matthias F. 3575
pey$ telnr2 Grabert
Grabert Matthias F.

pey$ add1 Grabert "Matthias F." 3575
pey$ telnr2 Grabert
Grabert Matthias F. 3575
Grabert Matthias F.
pey$
```

Bei dieser Form werden die neuen Datensätze jeweils am Ende angefügt. Sollte das Telefonbuch stets sortiert sein, so wird dies durch folgende Modifikation erreicht:

```
#
# Name, Vorname, Tel in Person.tels aufnehmen
#
echo "$1:$2:$3" >> Person.tels
sort -o Person.tels Person.tels
```

- Beispiel: Jemanden aus dem Telefonbuch entfernen

```
pey$ cat rem1
#
# jemanden aus Person.tels entfernen
#
grep -v "$1" $HOME/udb1/relDb/Person.tels > /tmp/Person.tels
mv /tmp/Person.tels $HOME/udb1/relDb/Person.tels

pey$ telnr2 Grabert
Grabert Matthias F. 3575
Grabert Matthias F.
pey$ rem1 Grabert
pey$ telnr2 Grabert
pey$
```

Diese Form hat noch einen großen Nachteil. Würden wir als Argument einen Namen angeben, zu dem es zwei (verschiedene) Einträge gibt, so würden beide entfernt werden!

Falls mehrere Einträge betroffen sind, sollte das Programm zumindest nachfragen, bevor alle gelöscht werden. Das wird in einer späteren Version betrachtet.

3.3 Das *shift*-Kommando

Das *shift*-Kommando bewirkt einen Links-Shift der Belegung der Positionsparameter; bei der ersten Ausführung von *shift* wird das, was zuvor in $2 abgelegt war, in $1 abgelegt, das von $3 in $2, u.s.w. Die alten Werte sind unwiederbringlich verloren, $# wird entsprechend dekrementiert.

Ein Programm zum Test von *shift*:

```
pey$ cat shifttest
echo $#:$*
shift
echo $#:$*
shift
echo $#:$*
shift
echo $#:$*
shift
echo $#:$*
pey$ shifttest "a b" c d e f
5:a b c d e f
4:c d e f
3:d e f
2:e f
1:f
pey$
```

shift kann als Argument eine Zahl haben - mit **shift 3** können z.B. 3 (mehrere) *shift's* ausgeführt werden.

Es gibt einige weitere **$**-Variable, die wir im Zusammenhang mit anderen Dingen kennenlernen werden.

4. Entscheidungen / Verzweigungen

4.1 *if*

Generelle Form der **if**-Anweisung:

if *test_command*

then

command
command
command

fi

Das *test_command* (beliebiges Kommando, nicht nur das *test*-Kommando) wird ausgeführt, sein **Exit Status** (!) wird geprüft; ist er Null, werden die zwischen *then* und *fi* stehenden Kommandos ausgeführt, ansonsten werden sie übergangen.

4.2 Der Exit-Status

Wenn unter UNIX ein Programm seine Ausführung beendet, wird ein "Exit Status" an das System zurückgegeben. Dieser Status ist eine Zahl, die üblicherweise anzeigt, ob das Programm erfolgreich zum Ende kam oder nicht. Per Konvention sagt ein Exit Status **ungleich 0** "nicht erfolgreich".

Der Exit Status einer Pipeline ist der Exit Status des letzten Kommandos. In

who | *grep swg*

wird der Exit Status von *grep* als Exit Status der Pipeline genommen.

4.3 Die *$?*-Variable

In dieser Variablen legt die Shell automatisch den Exit Status des zuletzt ausgeführten Kommandos ab:

- Beispiel: Feststellen, ob eine bestimmte Person in unserer Datenbang eingetragen ist; wenn ja, entsprechende Nachricht ausgeben, wenn nein, keine Nachricht.

```
pey$ cat is_in
if grep $1 $HOME/udb1/relDb/Person.tels
then
        echo "$1 ist eingetragen"
fi

pey$ is_in Mai
Maier:Urs:0421/442812
Mai:Karl:0731/12534
Mair:Josef:0911/5112323
Mai ist eingetragen
```

```
pey$
```

Dieses Programm hat noch einige unschöne Eigenheiten. Die Ausgabe von *grep* erscheint (nicht tragisch), die Suche nach Sub-Pattern zeigt weitere Personen an, nach denen nicht gefragt war.

Modifikation:

```
pey$ cat is_in1
if grep $1 $HOME/udb1/relDb/Person.tels > /dev/null
then
        echo "$1 ist eingetragen"
fi

pey$ is_in1 Mai
Mai ist eingetragen
pey$
```

4.4 Das *test*-Kommando

Format

Damit können Bedingungen im *if*-Kommando getestet werden. Das generelle Format ist:

test *expression*

String-Operatoren

Beispiel: *test $name = Schweiggert*

Zunächst ist festzuhalten, daß *test* alle seine Argumente sehen muß; in diesem Fall also *$name, Schweiggert* und **=** als separate Argumente, d.h. sie müssen durch Leerraum getrennt sein.

Es ist zweckmäßig, die Variable *name* in Apostrophen einzuschließen:

```
pey$ name=
pey$ test $name = Huber
test: =: unary operator expected
pey$ test "$name" = Huber
pey$ echo $?
1
pey$
```

Übersicht:

Operator	Exit Status 0, falls
string1 **=** *string2*	*string1* identisch mit *string2*
string1 **!=** *string2*	*string1 nicht identisch mit string2*
string	*string* ist nicht null (nicht leer)
-n *string*	*string* ist nicht null (und muß von *test* "gesehen" werden
-z *string*	*string* ist null (und muß von *test* "gesehen" werden

Beispiele:

```
pey$ blanks="            "
pey$ test $blanks
pey$ echo $?
1
pey$ test "$blanks"
pey$ echo $?
0
pey$
```

Im ersten Fall wird *$blanks* durch Leerraum ersetzt, dieser wird auf der Kommandozeile von *test* von der Shell "weggefressen", *test* sieht überhaupt kein Argument.

Der **-n** Operator kann als Test auf Länge ungleich 0, der **-z** als Test auf Länge gleich 0 angesehen werden:

```
pey$ leer=
pey$ nichtleer=etwas
pey$ test -n "$nichtleer"
pey$ echo $?
0
pey$ test -n "$leer"
pey$ echo $?
1
pey$ test -z "$nichtleer"
pey$ echo $?
1
pey$ test -z "$leer"
pey$ echo $?
0
pey$
```

Alternatives Format für *test*

Statt

test *expression*

kann man auch

[*expression* **]**

schreiben.

Beispiel:

```
pey$ name=swg
pey$ if [ "$name" = swg ]
> then
>     echo "Hallo, Herr Schweiggert"
> fi
Hallo, Herr Schweiggert
pey$ [-z "$name" ]
bash: [-z: command not found
pey$ [ -z "$name" ]
pey$ echo $?
1
pey$
```

Nach der öffnenden und vor der schließenden eckigen Klammer muß (natürlich) Leerraum kommen.

Integer-Operatoren

Operator	Exit Status 0, falls
int1 **-eq** *int2*	*int1 ist gleich int2*
int1 **-ge** *int2*	*int1* ist größer / gleich *int2*
int1 **-gt** *int2*	*int1* ist größer *int2*
int1 **-le** *int2*	*int1* ist kleiner / gleich *int2*
int1 **-lt** *int2*	*int1* ist kleiner *int2*
int1 **-ne** *int2*	*int1* ist ungleich *int2*

Beispiel:

```
pey$ x1="005"
pey$ [ "$x1" = 5 ]
pey$ echo $?
1
pey$ [ "$x1" -eq 5 ]
pey$ echo $?
0
pey$
```

Das *test*-Kommando führt die Interpretation einer Zeichenfolge als Integer beim Erkennen eines entsprechenden Operators durch.

Datei-Operatoren

Operator	Exit Status ist 0, falls
-d *file*	*file* existiert und ist ein Directory
-f *file*	*file* existiert und ist ein geöhnliche Datei
-r *file*	*file* existiert und kann gelesen werden
-w *file*	*file existiert und kann beschrieben werden*
-s *file*	*file* existiert und hat Länge > 0
-x *file*	*file* existiert und ist ausführbar

Verknüpfung von Ausdrücken

☞ Logische Negation: **!**

Beispiele:

```
pey$ [ -r telnr ]
pey$ echo $?
0
pey$ [ ! -r telnr ]
pey$ echo $?
1
pey$
```

☞ Logische Konjunktion: **-a**

Beispiel:

```
pey$ i=5
pey$ [ $i -gt 3 -a $i -lt 10 ]
pey$ echo $?
0
pey$
```

Der **-a**-Operator hat geringeren Vorrang als die Integer-Vergleichsoperatoren, so daß sich eine Klammerung erübrigt. Falls durch Klammerung ein anderer Vorrang hergestellt werden soll, so ist darauf zu achten, daß die runden Klammern für die Shell eine Sonderbedeutung haben:

```
pey$ i=5
pey$ [ \( $i -gt 3 \) -a \( $i -lt 10 \) ]
pey$ echo $?
0
pey$
```

Anm.: die runden Klammern dienen zur Zusammenfassung von Kommandos zur Ausführung durch eine **Subshell**:

```
pey$ pwd
/home/swg/udb1/shprog/10
pey$ echo $$
2457
pey$ ( cd ../../tools; pwd; echo $$)
/home/swg/udb1/tools
2457
pey$ echo $$
2457
pey$ pwd
/home/swg/udb1/shprog/10
pey$
```

In obigem Beispiel sind zwei Dinge anzumerken:

1. Die Variable **$$** enthält die Nummer des Prozesses.
2. Die in runden Klammern gesetzte Folge von Kommando's wird von einer sog. **Sub**-Shell ausgeführt - diese ist ein neuer Prozeß - die mit *echo $$* ausgegebene Prozeßnummer ist die von der aufrufenden Shell; dies liegt daran, daß die aufrufende Shell die Kommandozeile vorher bearbeitet, also **$$** erst bewertet und danach die Subshell kreiert. Die Subshell führt das der Wechsel wirkt aber nicht auf die aufrufende Shell!

☞ Die logische Disjunktion: **-o**

Anm.: Das UND (**-a**) hat höheren Vorrang als das ODER (**-o**).

4.5 Das *if-then-else*-Konstrukt

if *test_command*

then

command
command
command

else

command
command
command

fi

Ist der Exit Status von *test_command* **0**, werden die Kommandos zwischen **then** und **else**, ist er **ungleich 0**, die zwischen **else** und **fi** ausgeführt.

Modifikation des *is_in*-Programms:

```
pey$ cat is_in2
#
# feststellen, ob eine Person eingetragen ist
#

DB=$HOME/udb1/relDb/Person.tels
export DB

# richtige Anzahl von Argumenten?
if [ "$#" -ne 1 ]
then
        echo "Incorrect number of arguments" >&2
        echo "Usage: $0 name" >&2
        exit 1
else
        name="$1"
        if grep "$name" $DB > /dev/null
        then
                echo "$name ist eingetragen"
        else
                echo "$name ist nicht eingetragen"
        fi
fi
pey$ is_in2
Incorrect number of arguments
Usage: ./is_in2 name
pey$ is_in2 Mai
Mai ist eingetragen
pey$ is_in2 Nobody
Nobody ist nicht eingetragen
pey$
```

Anm.: Fehlermeldungen gehören nicht in die Standardausgabe - deshalb wird die Ausgabe von *echo* (geht an die Standardausgabe) in die Diagnoseausgabe umgelenkt: **>&2**

4.6 Das *exit*-Kommando

Damit wird ein Programm beendet und ein definierbarer Exit-Wert zurückgegeben. Generelles Format:

exit *n*

n ist der gewünschte Exit-Wert. Ist keiner angegeben, so wird der Exit Status des zuletzt ausgeführten Kommandos genommen. Eingabe von *exit* direkt auf Shell-Ebene bewirkt ein "log off".

Modifikation des *rem*-Programms - Eintrag aus *Person.tels* entfernen:

```
pey$ cat rem2
#
# jemanden aus Person.tels entfernen
#
DB=$HOME/udb1/relDb/Person.tels
export DB

if [ "$#" -ne 1 ]
then
        echo "Incorrect number of arguments" >&2
        echo "Usage: $0 name" >&2
        exit 1
fi

grep -v "$1" $DB > /tmp/rem$$
mv /tmp/rem$$ $DB
pey$ rem2
Incorrect number of arguments
Usage: ./rem2 name
pey$ rem2 Hubert Kempter
Incorrect number of arguments
Usage: ./rem2 name
pey$ rem2 Emu
pey$ echo $?
0
pey$
```

4.7 Das *elif*-Konstrukt

if *test_command1*
then
 command
 command
elif *test_command2*
then
 command
 command
elif *test_commandN*
then
 command
 command
else

command
command

fi

Eine weitere Modifikation des *rem*-Programms, bei dem überprüft wird, ob mehr als ein Eintrag getroffen wird:

```
pey$ cat rem3
#
# jemanden aus Person.tels entfernen
#
DB=$HOME/udb1/relDb/Person.tels
export DB

if [ "$#" -ne 1 ]
then
        echo "Incorrect number of arguments" >&2
        echo "Usage: $0 name" >&2
        exit 1
fi

name="$1"

# Wieviele Treffer?
matches=`grep "$name" $DB | wc -l`

# Nur ein Treffer - loeschen
# Mehr als ein Treffer - Meldung
if [ $matches -gt 1 ]
then
        echo "Mehr als ein Eintrag"
elif [ $matches -eq 1 ]
then
        grep -v "$name" $DB > /tmp/rem$$
        mv /tmp/rem$$ $DB
        echo "$name geloescht!"
else
        echo "$name nicht enthalten"
fi

pey$ rem3
Incorrect number of arguments
Usage: ./rem3 name
pey$ add1 King Luigi 0000
pey$ add1 King Luigi 1111
pey$ rem3 King
Mehr als ein Eintrag
pey$ rem3 Luigi:0000
Luigi:0000 geloescht!
pey$
```

Anm.: Der Doppelpunkt im letzten *rem*-Aufruf ist notwendig, weil in unserer Datei *Person.tels* dieses Zeichen als Feldtrenner fungiert - wir werden uns dazu später noch etwas überlegen.

4.8 Das *case*-Kommando

Generelle Form:

case *value* **in**
    pat_1) *command*
        command
        command **;;**
    pat_2) *command*
        command
        command **;;**
    pat_N) *command*
        command
        command **;;**
esac

Das Wort *value* wird sukzessive mit den Mustern pat_1, pat_2, ... pat_N verglichen; wird eine Übereinstimmung festgestellt, so werden die zugehörigen Kommandos bis zum **;;** ausgeführt.

Beispiel:

```
#
# Uebersetzen der englischen Kuerzel fuer
# Wochentage in die deutsche "Langfassung"
#

if [ "$#" -ne 1 ]
then
        echo "Usage: $0 day" >&2
        exit 1
fi

case "$1" in
        Sun     )       echo Sonntag ;;
        Mon     )       echo Montag ;;
        Tue     )       echo Dienstag ;;
        Wed     )       echo Mittwoch ;;
        Thu     )       echo Donnerstag ;;
        Fri     )       echo Freitag ;;
        Sat     )       echo Samstag ;;
        *       )       echo Unbekannt ; exit 1 ;;
esac
exit 0
```

Anm.: Daß die Ausführung in einer *pipeline* so nicht gehen kann, sollte klar sein!

Beispiel: Datum in deutscher Form für Reports erzeugen

```
pey$ cat ger_datum
tag=`date +%a`
case "$tag" in
        Sun     )       echo -n Sonntag ;;
        Mon     )       echo -n Montag ;;
        Tue     )       echo -n Dienstag ;;
        Wed     )       echo -n Mittwoch ;;
        Thu     )       echo -n Donnerstag ;;
        Fri     )       echo -n Freitag ;;
        Sat     )       echo -n Samstag ;;
        *       )       echo Unbekannt ; exit 1 ;;
esac
echo -n ", "
date +%d.%m.%Y
exit 0

pey$ ger_datum
Mittwoch, 23.07.1997
pey$
```

Die Muster in der *case*-Struktur können genauso aufgebaut sein wie bei der Dateinamen-Substitution: **?** für ein beliebiges einzelnes Zeichen, ***** für beliebige Zeichenfolgen, **[...]** für in den Klammern spezifizierte einzelne Zeichen. Der Stern ***** wird daher i.a. als "Auffangfall" verwendet (s.o.).

Zum *echo*-Kommando: *echo* fügt ans Ende automatisch ein *newline* an. Bei manchen Systemen kann das mit der Option **-n**, bei manchen mit **\c** als letztem Zeichen unterdrückt werden.

Beispiel: Feststellen des Zeichentyps

```
pey$ cat chartype
#
# Klassifizieren von Zeichen
#
if [ "$#" -ne 1 ]
then
        echo "Usage: $0 char" >&2
        exit 1
fi

# Nur ein einzelnes Zeichen?

char="$1"
```

```
number=`echo "$char" | wc -c`

if [ "$number" -ne 1 ]
then
        echo "Bitte nur ein einzelnes Zeichen" >&2
        exit 2
fi

# Klassifizieren

case "$char" in
        [0-9]   )       echo Ziffer ;;
        [a-z]   )       echo Kleinbuchstabe ;;
        [A-Z]   )       echo Grossbuchstabe ;;
        \?      )       echo Fragezeichen ;;
        ?       )       echo Sonderzeichen ;;
        *       )       echo Da stimmt was nicht ;;
esac
pey$ chartype a
Bitte nur ein einzelnes Zeichen
pey$
```

Ursache: echo-Kommando! (s.o)

4.9 Fehlersuche / Debugging

Um obigen Fehler zu finden, führen wir das Programm in einer Subshell mit der Option **-x** aus:

```
pey$ sh -x chartype a
+ [ 1 -ne 1 ]
+ char=a
++ echo a
++ wc -c
+ number=      2
+ [       2 -ne 1 ]
+ echo Bitte nur ein einzelnes Zeichen
Bitte nur ein einzelnes Zeichen
+ exit 2
pey$
```

4.10 Entweder-Oder-Kombination von Fällen

Beispiel von oben, bei Buchstaben keine Unterscheidung in Groß-/Kleinbuchstaben:

```
#
# Klassifizieren von Zeichen
#
if [ "$#" -ne 1 ]
then
        echo "Usage: $0 char" >&2
        exit 1
fi

# Nur ein einzelnes Zeichen?

char="$1"

# Klassifizieren und Ueberpruefen:

case "$char" in
        [0-9]           )       echo Ziffer ;;
        [a-z] | [A-Z]   )       echo Buchstabe ;;
        ?               )       echo Sonderzeichen ;;
        *               )       echo Bitte nur ein Zeichen eingeben ;;
esac
```

In diesem Fall hätte man natürlich statt **[a-z] | [A-Z]** auch **[a-zA-Z]** schreiben können.

4.11 Das leere Kommando :

Die Shell erwartet an vielen Stellen die Angabe eines Kommandos; das leere Kommando dient dazu, dieses zu befriedigen, auch wenn an dieser Stelle nichts getan werden soll.

4.12 bedingte Ausführung: && und ||

Wenn ein Kommando $command_2$ in Abhängigkeit der Erfolgs eines anderen Kommandos $command_1$ ausgeführt werden soll, so läßt sich das statt

```
if command1
then
        command2
fi
```

auch kürzer schreiben als

```
command1 && command2
```

Wenn $command_1$ mit einem Exit Status 0 endet, dann wird $command_2$ ausgeführt, sonst nicht.

Wenn ein Kommando $command_2$ nur ausgeführt werden soll, wenn ein anderes Kommando $command_1$ nicht erfolgreich war, so läßt sich dies schreiben als

```
command1 || command2
```

Wenn also *command*$_1$ mit einem Exit Status ungleich 0 endet, so wird *command*$_2$ ausgeführt, sonst nicht. Dies ließe sich auch so formulieren:

```
if command1
then
        :
else
        command2
fi
```

5. Schleifen

Die Shell kennt wie die meisten höheren Programmiersprachen drei Arten von Schleifen:

- die **for**-Schleife
- die **while**-Schleife
- die **until**-Schleife

5.1 Die *for*-Schleife

Format:

for *var* **in** *word*$_1$ *word*$_2$ ... *word*$_N$
do
command
command
command
done

Beispiel:

```
pey$ cat for1
for i in 1 2 3 4 5
do
        echo -n $i
done
echo

pey$ for1
12345
pey$
```

Oder:

```
pey$ cat for2
for file in for*
do
        echo -n "$file : "
        head -1 $file
done

pey$ for2
for1 : for i in 1 2 3 4 5
for1.se : pey$ cat for1
for2 : for file in for*
pey$
```

Oder:

```
pey$ cat for3
for file in for[1-3]
do
        echo -n "$file : "
        head -1 $file
done

pey$ for3
for1 : for i in 1 2 3 4 5
for2 : for file in for*
for3 : for file in for[1-3]
pey$
```

Wenn die zu bearbeitenden Dateien in einer Datei *filelist* aufgelistet sind:

```
pey$ filelist
pey$ cat filelist
for1
for2
for3
for4

pey$ cat for4
for file in `cat filelist`
do
        echo "$file : "
        head -1 $file
        echo "-----------------------------------"
done

pey$ for4
for1 :
for i in 1 2 3 4 5
----------------------------------
for2 :
for file in for*
```

```
----------------------------------
for3 :
for file in for[1-3]
----------------------------------
for4 :
for file in `cat filelist`
----------------------------------
pey$
```

Sollen die zu bearbeitenden Dateien als Argumente übergeben werden können, könnte dies wie folgt geschehen:

```
pey$ cat for5
for file
do
        echo -n "$file : "
        head -1 $file
        echo "---------------------------------------------"
done
pey$ for5 for?
for1 : for i in 1 2 3 4 5
---------------------------------------------
for2 : for file in for*
---------------------------------------------
for3 : for file in for[1-3]
---------------------------------------------
for4 : for file in `cat filelist`
---------------------------------------------
for5 : for file
---------------------------------------------
pey$
```

5.2 Die Shell-Variablen $* und $@

Beispiel: Übergebene Argumente ausgeben

```
pey$ cat argtest2
echo Anzahl der uebergebenen Args: $#
for arg in $*
do
        echo $arg
done

pey$ argtest2 'a b' c
Anzahl der uebergebenen Args: 2
a
b
c
pey$
```

Hier wurden 2 Argumente, einmal **a b** und zum anderen **c** an *argtest2* übergeben; **$*** in der *for*-Anweisung wurde durch **a b c** ersetzt, so daß die *for*-Schleife eben dreimal durchlaufen wurde.

Für diesen Fall hilft die Shell-Variable **$@**: sie hat zunächst die gleiche Wirkung wie **$***; wird sie aber quotiert verwendet, also "$@", so erfolgt die Umsetzung nach "$1", "$2", u.s.w.

```
pey$ cat argtest3
echo Anzahl der uebergebenen Args: $#
for arg in "$@"
do
        echo $arg
done

pey$ argtest3 'a b' c
Anzahl der uebergebenen Args: 2
a b
c
pey$
```

Was passiert, wenn man **$*** quotiert?

```
pey$ cat argtest4
echo Anzahl der uebergebenen Args: $#
for arg in "$*"
do
        echo $arg
done

pey$ argtest4 'a b' c
Anzahl der uebergebenen Args: 2
a b c
pey$
```

Und **$@** ohne Quotierung?

```
pey$ cat argtest5
echo Anzahl der uebergebenen Args: $#
for arg in $@
do
        echo $arg
done

pey$ argtest5 'a b' c
Anzahl der uebergebenen Args: 2
a
b
c
pey$
```

Das Konstrukt *for var in $@* läßt sich auch einfacher haben:

```
for var
do
        command
        ...
        command
done
```

Man beachte: kein **in**

Damit geht die Shell durch alle Argumente der Kommandozeile.

Beispiel:

```
pey$ cat argtest6
echo Anzahl der uebergebenen Args: $#
for arg
do
        echo $arg
done

pey$ argtest6 'a b' c
Anzahl der uebergebenen Args: 2
a b
c
pey$
```

5.3 Die *while*-Schleife

Format:

while *test_command*
do
command
command
done

Ist der Exit Status von *test_command* **0**, so wird der Schleifenrumpf ausgeführt, danach wird *test_command* erneut ausgeführt, u.s.w.

Beispiel: eine einfache Zählschleife

```
pey$ cat while1
i=1
while [ $i -le 4 ]
do
      echo "Wert von i: $i"
      i=`expr $i + 1`
done

pey$ while1
Wert von i: 1
Wert von i: 2
```

```
Wert von i: 3
Wert von i: 4
pey$
```

Beispiel: Ausgabe der Argumente auf der Kommandozeile

```
pey$ cat pr_args
i=1
while [ "$#" -ne 0 ]
do
        echo -n "$i.tes Argument: "
        echo "$1"
        shift
        i=`expr $i + 1`
done

pey$ pr_args a b c 'd e' f
1.tes Argument: a
2.tes Argument: b
3.tes Argument: c
4.tes Argument: d e
5.tes Argument: f
pey$
```

5.4 Die *until*-Schleife

Diese entspricht in ihrer Semantik der **REPEAT-UNTIL**-Konstruktion von Modula-2.

Format:

until *test_command*
do
        command
        command
done

Beispiel: Warten, bis ein bestimmtes Ereignis eintritt (sich irgendein Benutzer angemeldet hat) und dann Nachricht (optional per *mail*)

```
#
# Warten bis sich ein Benutzer angemeldet hat
# Optional per mail mitteilen
#
if [ "$1" = -m ]
then
        send=TRUE
        shift
else
        send=FALSE
fi
```

```
if [ "$#" -eq 1 ]
then
        :
else
        echo "Usage: $0 [-m] username" >&2
        echo "-m: Message via mail" >&2
        exit 1
fi

user="$1"

# alle 60 sec nachsehen
until who | grep "^$user " > /dev/null
do
        sleep 60
done

# user ist jetzt angemeldet
echo "Hallo, $user" | mail $user
if [ "$send" = FALSE ]
then
        echo "$user ist jetzt da"
else
        # eigenen Benutzernamen feststellen
        sender=`who am i | cut -c9-16`
        echo "$user ist jetzt da" | mail $sender
fi
```

5.5 Abbrechen von Schleifen: *break - break n*

Mit Erreichen von *break* wird die unmittelbar umgebende Schleife beendet, *break n* beendet die **n** unmittelbar umschließenden Schleifen

5.6 Endlos-Schleifen

```
while true
do
        ...
done
```

```
until false
do
        ...
done
```

```
while :
do
        ...
done
```

- **true** tut nichts außer einen Exit Status von 0 zu liefern;
- **false** tut nichts außer einen Exit Status ungleich 0 zu liefern;
- das leere Kommando **:** tut auch nichts, außer einen Exit Status von 0 zu liefern.

5.7 Vorzeitig am Schleifenbeginn fortfahren: *continue* - *continue n*

Ausführen von Schleifen im Hintergrund:

```
pey$ cat for6
for i in for?
do
        echo -n "$i : "
        head -1 $i
        echo "------------------------"
done &

pey$ for6
for1 : pey$ for i in 1 2 3 4 5
------------------------
for2 : for file in for*
------------------------
for3 : for file in for[1-3]
------------------------
for4 : for file in `cat filelist`
------------------------
for5 : for file
------------------------
for6 : for i in for?
------------------------

pey$
```

5.8 I/O-Umlenkung bei Schleifen

Eingabe-Umlenkung (im Zusammenhang mit Lesen von Daten dazu mehr) wirkt auf alle Kommandos in der Schleife, die von der Standard-Eingabe Daten lesen. Ausgabe-Umlenkung erfolgt nach *done* und wirkt, sofern keine spezielle Umlenkung, ebenfalls auf alle Kommandos in der Schleife.

Beispiele:

```
pey$ cat for7
echo "pid = $$"
for i in 1 2 3
do
        echo "wohin geht $i ?" > /dev/null
        echo "und wohin geht $i ?" > ./x
        echo "und wohin schliesslich $i?"
done > /tmp/tst$$
echo "/tmp/tst$$:"
cat /tmp/tst$$
echo x:
cat x

pey$ for7
pid = 3743
/tmp/tst3743:
und wohin schliesslich 1?
und wohin schliesslich 2?
und wohin schliesslich 3?
x:
und wohin geht 3 ?
pey$
```

Anm.: Dieses Beispiel zeigt (noch einmal, s.o.) einen Weg auf, eindeutige Dateinamen zu erzeugen: *name*$$

Für die Bearbeitung von Optionen gibt es ein nützliches Kommando: **getopts**

Dazu später mehr (siehe auch `man`).

6. Eingabe

Format:

read *var1 var2 ...*

Wirkung:

Die Shell liest eine Zeile von der Standardeingabe und weist der ersten Variablen das erste Wort zu, der zweiten das zweite, usw. Sind weniger Variable augelistet als Worte auf der Eingabezeile, so wird der letzten Variable der Rest der Zeile zugewiesen.

Beispiele:

```
pey$ cat auswahl
clear # Schrim loeschen

echo "Waehlen Sie aus:"
echo -e "\tTelefonieren \t(T)"
echo -e "\tHeimgehen \t(H)"
echo -e "\tBiergarten \t(B)"
echo -n -e "\t\tEingabe: \t"

read wahl
case $wahl in
        [Tt] )  echo "Telefonieren" ;;
        [Hh] )  echo "Heimgehen" ;;
        [Bb] )  echo "Biergarten" ;;
        *    )  echo -e "Falsche Wahl \a" ;;
esac
# -e -> erlaubt Interpretation von Escape-Sequenzen
# \a -> Tonzeichen

pey$ auswahl
Waehlen Sie aus:
        Telefonieren    (T)
        Heimgehen       (H)
        Biergarten      (B)
                Eingabe:        a
Falsche Wahl ^G
pey$

pey$ cat lies
read name vname tel
echo "Name: $name"
echo "Vorname: $vname"
echo "Telefon: $tel"

pey$ echo Mueller Hans 3577 | lies
Name: Mueller
```

```
Vorname: Hans
Telefon: 3577
pey$

pey$ cat lies1
tail -2 $HOME/udb1/relDb/Person.tels | tr ':' ' ' | \
while read name vname tel
do
        echo "Name: $name"
        echo "Vorname: $vname"
        echo "Telefon: $tel"
        echo "--------------------"
done

pey$ lies1
Name: Borchert
Vorname: Andreas
Telefon: 3572
--------------------
Name: King
Vorname: Luigi
Telefon: 1111
--------------------
pey$
```

Zuviele Variable:

```
pey$ cat lies2
tail -2 $HOME/udb1/relDb/Person.tels | tr ':' ' ' | \
while read name vname tel ort beruf
do
        echo "Name: $name"
        echo "Vorname: $vname"
        echo "Telefon: $tel"
        echo "Wohnort: $ort"
        echo "Beruf: $beruf"
        echo "--------------------"
done

pey$ lies2
Name: Borchert
Vorname: Andreas
Telefon: 3572
Wohnort:
Beruf:
--------------------
Name: King
Vorname: Luigi
Telefon: 1111
Wohnort:
```

```
Beruf:
--------------------
pey$
```

Zuwenig Variable:

```
pey$ cat lies3
tail -2 $HOME/udb1/relDb/Person.tels | tr ':' ' ' | \
while read name vname
do
        echo "Name: $name"
        echo "Vorname: $vname"
        echo "--------------------"
done

pey$ lies3
Name: Borchert
Vorname: Andreas 3572
--------------------
Name: King
Vorname: Luigi 1111
--------------------
pey$
```

Der Effekt ist sichtbar: Die letzten Worte der Eingabezeile werden an die letzte genannte Variable zugewiesen!

Beispiel: Informationen über Datenbank-Tabellen

```
pey$ cat db_info
#
# Ausgabe von Informationen zu einer Tabelle:
#
# Spaltennamen, Anzahl Datensaetze
# Aufbau der Tabelle:
# erste Zeile: Spaltennamen
# Spalten durch : getrennt

table="$1"
echo
echo "Informationen zur Tabelle $table"
echo "======================================================"
echo

read kopf <"$table"
echo "Satzstruktur:"
echo "$kopf"
echo
echo "Spaltennamen:"
echo "-------------"
kopf=`echo "$kopf" | tr ":" "   "`
for attr in $kopf
do
        echo $attr
```

```
done
echo
anzahl=`cat $table | wc -l`
anzahl=`expr $anzahl - 1` # ohne Kopfzeile
echo -n "Anzahl der Datensaetze: "
echo $anzahl
echo

pey$ db_info $HOME/udb1/relDb/Person.data

Informationen zur Tabelle /home/swg/udb1/relDb/Person.data
==========================================================

Satzstruktur:
PersId:Name:Vorname:Anrede:Institution:Strasse:Plz:Ort:Tel

Spaltennamen:
-------------
PersId
Name
Vorname
Anrede
Institution
Strasse
Plz
Ort
Tel

Anzahl der Datensaetze: 28

pey$
```

Als nächstes Beispiel soll eine Projektion einer Tabelle auf durch Namen spezifizierte Spalten entwickelt werden:

Aufruf: **project** *tablename colnames*

```
pey$ cat project
#
# Die Tabelle ist wie folgt aufgebaut:
# erste Zeile enthaelt die Spaltennamen
# die Spalten sind durch : getrennt
#
if [ $# -lt 2 ]
then
        echo "Usage: $0 tablename attrnames"
        exit 1
fi

table=$1
shift
# die verbleibenden Argumente sind Spaltennamen
# existiert die Tabelle ?
if [ ! -r $table ]
```

```
then
        echo "Tabelle $table existiert nicht"
        exit 2
fi

read kopf <$table
# den : als Trenner durch Leerraum ersetzen

kopf=`echo $kopf | tr ':' ' '`

# Optionen fuer das cut-Kommando ermitteln
cutoption="-d: -f"
for attr
do
        i=1
        for col in $kopf
        do
                if [ $attr = $col ]
                then
                        cutoption="$cutoption$i,"
                fi
                i=`expr $i + 1`

        done
done

if [ "$cutoption" = '-d: -f' ]
then
        echo "Attributnamen sind unzulaessig"
        exit 1
fi
cut $cutoption $table
```

```
pey$ project $HOME/udb1/relDb/Person.data Name Tel | head -4
Name:Tel
Biertaler:0815/10007
Oha:0815/10008
Koernigen:0815/10008
pey$
```

Das Suchen nach einem Datensatz in einer Tabelle über einen spezifizierten Attributwert könnte i.P. mit *grep* erfolgen. Dies hat allerdings den Nachteil, daß auf die Satzstruktur keine Rücksicht genommen wird.

Auch das kann mit einem einfachen Programm besser (aber leider nur wesentlich langsamer) gemacht werden:

```
pey$ cat suche
# selektiert die Datens\*atze aus einer Tabelle ($1)
# fuer die ein Attributwert einen gegebenen Wert hat ($2 $3)
# Aufruf: $0 tablename attributname attributvalue

table=$1
attr=$2
val=$3
read kopf <$table
kopf=`echo $kopf | tr ':' '     '`

i=1
for col in $kopf
do
        if [ "$col" = "$attr" ]
        then
                break
        fi
        i=`expr $i + 1`
done
k=1
cat $table | while read satz
do
        if [ $k -eq 1 ]  #Kopfzeile
        then
                echo "$satz"
                echo
        else

                wert=`echo $satz | cut -d: -f$i`
                if [ "$wert" = "$val" ]
                then
                        echo "$satz"
                fi
        fi
        k=`expr $k + 1`
done

pey$ suche $HOME/udb1/relDb/Person.data Name Oha

PersId:Name:Vorname:Anrede:Institution:Strasse:Plz:Ort:Tel

4712:Oha:Alfred V.:Herr:BS&S:Usambaraweg 11:9876:Glockendorf:0815/10008
pey$
```

Dies läßt sich durch Vorschalten des *grep* beschleunigen:

```
pey$ cat suche2
# selektiert die Datens\*atze aus einer Tabelle ($1)
# fuer die ein Attributwert einen gegebenen Wert hat ($2 $3)
# Aufruf: $0 tablename attributname attributvalue

table=$1
attr=$2
val=$3
read kopf <$table
echo "$kopf"

kopf=`echo $kopf | tr ':' '     '`

i=1
```

```
for col in $kopf
do
        if [ "$col" = "$attr" ]
        then
                break
        fi
        i=`expr $i + 1`
done
cat $table | grep "$val" | while read satz
do

        wert=`echo $satz | cut -d: -f$i`
        if [ "$wert" = "$val" ]
        then
                echo "$satz"
        fi
done
```

```
pey$ suche2 $HOME/udb1/relDb/Person.data Name Oha

PersId:Name:Vorname:Anrede:Institution:Strasse:Plz:Ort:Tel
4712:Oha:Alfred V.:Herr:BS&S:Usambaraweg 11:9876:Glockendorf:0815/10008
pey$
```

Beide Programme lesen eine Tabelle, die als erstes Argument angegeben sein muß. Damit lassen sie sich weder kombinieren (in einer Pipeline) noch läßt sich I/O-Umlenkung nutzen.

Modifizierte Version proj von project:

```
pey$ cat proj
#
# Die Tabelle aus stdin ist wie folgt aufgebaut:
# erste Zeile enthaelt die Spaltennamen
# die Spalten sind durch : getrennt
#
if [ $# -eq 0 ]
then
        echo "Usage: $0 attrnames"
        exit 1
fi

tmp=/tmp/table$$
if [ -f $tmp ]
then
        echo "Temporaere Datei ex. bereits" >&2
        exit 1
fi

while read satz
do
        echo "$satz" >> "$tmp"
done

read kopf < "$tmp"
# den : als Trenner durch Leerraum ersetzen
```

```
kopf=`echo $kopf | tr ':' ' '`
# Optionen fuer das cut-Kommando ermitteln
cutoption="-d: -f"
for attr
do
        i=1
        for col in $kopf
        do
                if [ $attr = $col ]
                then
                        cutoption="$cutoption$i,"
                fi
                i=`expr $i + 1`

        done
done
if [ "$cutoption" = '-d: -f' ]
then
        echo "Attributnamen sind unzulaessig"
        rm $tmp
        exit 1
fi
cut $cutoption $tmp
rm $tmp
exit 0
```

```
pey$ cat $HOME/udb1/relDb/Person.data | proj Name Tel | head -3
Name:Tel
Biertaler:0815/10007
Oha:0815/10008
pey$
```

7. Environment

Beispiel:

```
pey$ cat vartest1
echo :$x:
pey$ x=aaaa
pey$ echo :$x:
:aaaa:
pey$ vartest1
::
pey$
```

Dieser Effekt erklärt sich dadurch, daß die Ausführung des Scripts *vartest* über eine Subshell (eine von der login-Shell "aufgerufene" Shell erfolgt und daß die in einer Shell vereinbarten Variablen **lokal** sind:

```
pey$ cat vartest2
x=xxxx
echo :$x:
pey$ x=aaaa
pey$ vartest2
:xxxx:
pey$ echo :$x:
:aaaa:
pey$
```

Immer, wenn eine neue Shell ausgeführt wird, hat sie ihre eigene Umgebung (Environment).

7.1 Export von Variablen in Subshells

export *variables*

Damit werden Variable an Subshells im Sinne von "Wertparameterübergabe", also als Kopie weitergegeben.

Beispiel (Unterschied zwischen lokalen und exportierten Variablen):

```
pey$ cat vartest3
echo x = $x
echo y = $y

pey$ x=1
pey$ y=2
pey$ vartest3
x =
y =
pey$ export y
pey$ vartest3
x =
y = 2
```

```
pey$ cat vartest4
x=50
y=100
pey$ vartest4
pey$ echo $x ; echo $y
1
2
pey$
```

Wenn in einer Subshell eine von der "Vater-Shell" exportierte Variable verändert wird, so wird unter deren Namen eine neue lokale Variable angelegt. Benutzung dieser Variablen bezieht sich dann im folgenden auf die lokale Variable.

Durch einen Export kann eine Subshell weiterhin Variable an von ihr initiierte Subshells weiterreichen:

```
pey$ cat vartest5
x=50
y=5
z=1
export z
vartest6

pey$ cat vartest6
echo x = $x
echo y = $y
echo z = $z

pey$ vartest5
x =
y =
z = 1
pey$
```

Bei der Ausführung von *vartest5* (Subshell) wird die von der "Vatershell" (in dem Fall die login shell) exportierte Variable y (s.o.) in die Umgebung der Subshell (*vartest5*) kopiert; die Wertzuweisungen in *vartest5* bestimmen lokale Variable (x,y,z); z wird exportiert.

Der Aufruf von *vartest6* erzeugt eine neue Subshell, in deren Umgebung die exportierten Variablen y (mit Orginalwert 2) und z kopiert werden.

Ein expliziter Export von y in *vartest5* gibt den modifizierten Wert (also die lokale Version von y) weiter:

```
pey$ cat vartest5a
x=50
y=5
z=1
export y z
vartest6

pey$ cat vartest6
echo x = $x
echo y = $y
echo z = $z

pey$ vartest5a
x =
y = 5
z = 1
pey$
```

7.2 Zusammenfassung

1. Variablen, die nicht exportiert werden, sind lokal und für Subshells unsichtbar.
2. Exportierte Variable werden samt Wert in die Umgebung der Subshells kopiert und werden bei Änderungen als lokale Variable behandelt. Diese Änderungen wirken nicht auf die Variable in der "Vater-Shell".
3. Wenn eine Subshell explizit eine Variable exportiert, werden Änderungen dieser Variablen mitexportiert. Wird die Variable nicht exportiert, so wird nicht der geänderte Wert weitergegeben.
4. Ein Export bezieht sich auch auf "Subshells von Subshells von ..."
5. Variable können jederzeit (vor / nach Wertzuweisung) exportiert werden.

Der *export* ohne Argumentangabe zeigt die Variablen an, die von der aktuellen Shell exportiert werden.

7.3 Spezielle Shellvariable

- **PS1** und **PS2**: Primär- und Sekundär-Prompt

```
ariadne$ echo ":$PS1:"
:ariadne$ :
ariadne$ echo ":$PS2:"
:> :
ariadne$
```

- **HOME**: "Heimatkatalog", "Login-Katalog"

```
ariadne$ echo $HOME
/home/laborix/swg
```

Nach Möglichkeit nicht ändern, *cd* "vertraut darauf!

- **PATH**: Spezifiziert die Kataloge, in denen die Shell nach Kommandos sucht
- **CDPATH**: Spezifiziert ähnlich wie *PATH* eine Liste von Katalogen, die von *cd* durchsucht werden.

7.4 Das Punkt-Kommando .

Format: *file*

Wirkung: führt den Inhalt des angegebenen Files in der aktuellen Shell aus, so als ob sie in der aktuellen Shell direkt eingegeben worden wären.

Beispiel:

```
pey$ pwd
/home/swg/udb1/shprog/13

pey$ cat cd_dbdir
DB=$HOME/udb1/relDb
cd $DB
pwd

pey$ cd_dbdir
/home/swg/udb1/relDb

pey$ pwd
/home/swg/udb1/shprog/13

pey$ echo $DB

pey$ . cd_dbdir
/home/swg/udb1/relDb

pey$ pwd
/home/swg/udb1/relDb

pey$ echo $DB
/home/swg/udb1/relDb
pey$
```

7.5 Das *exec*-Kommando

Format: **exec** *program*

Wirkung: Ersetzen des aktuellen Programmes (Prozesses) durch eine neues (*program*).

Beispiel:

```
ariadne$ cat dbexec
PS1="DB: "
export PS1
exec /bin/sh
ariadne$ dbexec
DB:
DB:
<ctrl-D>
ariadne$
```

Die Subshell *dbexec* wird ersetzt durch eine neue Shell (*bin/sh*).

7.6 I/O-Umlenkung und Subshells

Wenn bei Kommandos wie **if** oder **for** Ausgabe-Umlenkung erfolgt, so werden diese in Pseudo-Subshells ausgeführt.

Beispiel:

```
ariadne$ for i in a b c; do echo $i; done
a
b
c
ariadne$ echo $i
c
ariadne$
```

UND:

```
ariadne$ for k in a b c; do echo $k; done >/dev/null
ariadne$ echo $k
c
ariadne$
```

Im zweiten Fall wird die *for*-Anweisung wegen der Umlenkung in einer Pseudo-Subshell ausgeführt, Wertveränderungen sind in der Vater-Shell wirksam.

Wie sieht es bei der Eingabe-Umlenkung aus? Beispiel:

```
pey$ cat wcl
 # counts number of lines in a file

file=$1
count=0

while read line
do
        count=`expr $count + 1`
done < $file

echo $count

pey$ wcl wcl
12
pey$ wc -l wcl
     12 wcl
pey$

pey$ count=0
pey$ while read line
>> do
>>      count=`expr $count + 1`
>> done < wcl
pey$ echo $count
12
pey$
```

Aber:

```
pey$ count=0
pey$ cat wcl |
> while read line
> do
>      count=`expr $count + 1`
> done
pey$ echo $count
0
pey$
```

☞

7.7 Umdefinieren von *stdin / stdout*

Mit **exec <** *file* wird Standard-Eingabe von *file* genommen, mit **exec >** *file* wird *file* als Standardausgabe genommen,

mit **exec < /dev/tty** wird *stdin* wieder zur Standardeingabe (sinngemäß auch für die Standardausgabe).

7.8 Die (...) und {...} Konstrukte

Mit **(...)** können Kommandos (durch Semikolon getrennt) zusammengefaßt werden und zur Ausführung durch eine Subshell gebracht werden. Mit **{...}** werden Kommandos ebenfalls zusammengefaßt (um sie z.B. gemeinsam in den Hintergrund zu schicken), die Ausführung erfolgt aber durch die aktuelle Shell.

Beispiele:

```
ariadne$ x=50
ariadne$ (x=100)
ariadne$ echo $x
50
ariadne$ { x=100; }
ariadne$ echo $x
100
ariadne$
```

Anm.: Nach **{** muß ebenso wie vor **}** ein Blank stehen, das letzte Kommando muß mit ; abgeschlossen sein. In Verbindung mit I/O-Umlenkung wird ebenfalls mit einer Subshell gearbeitet.

8. Mehr über Parameter

8.1 *${parameter:-value}*

Wirkung: Wenn *parameter* nicht null ist, erfolgt Ersetzung des Wertes von *parameter,* ansonsten die Ersetzung von *value.*

Beispiel:

```
ariadne$ a=20
ariadne$ echo ${a:-30}
20
ariadne$ b=
ariadne$ echo ${b:-111}
111
ariadne$
```

8.2 *${parameter:=value}*

Wenn *parameter* einen Wert hat, passiert weiter nichts; wenn nicht, wird *value* substituiert und gleichzeitig an *parameter* zugewiesen.

Beispiel:

```
ariadne$ zeig=
ariadne$ ${zeig:=pwd}
/home/laborix/swg/dbskript/shprog
ariadne$ echo $zeig
pwd

ariadne$ xx=
ariadne$ ${xx:=12}
12: not found

ariadne$ xx=who
ariadne$ ${xx:=pwd}
swg      console Dec 30 09:14
swg      ttyp0   Dec 30 09:15
swg      ttyp1   Dec 30 09:17
swg      ttyp2   Dec 30 09:18
swg      ttyp3   Jan 22 18:24
ariadne$
```

□

8.3 Das Null-Kommando :

Beispiel:

```
ariadne$ PHONEBOOK=
ariadne$ : ${PHONEBOOK:=$HOME/phones}
ariadne$ echo $PHONEBOOK
/home/laborix/swg/phones
ariadne$
```

8.4 *${parameter:?value}*

Beispiel:

```
ariadne$ PHONEBOOK=
ariadne$ : ${PHONEBOOK:?"No Phonebook file"}
PHONEBOOK: No Phonebook file
ariadne$ : ${PHONEBOOK:?}
PHONEBOOK: parameter null or not set
ariadne$
```

Wenn *parameter* nicht null und ein *value* angegeben ist, so substituiert die Shell den Wert; ist er null und ein Wert angegeben, so schreibt die Shell den Wert nach Standard-Error und terminiert (nicht die login-Shell).

Ist kein *value* angegeben, so s.o.

8.5 *${parameter:+value}*

Substitution findet nur statt, wenn *parameter* nicht null ist:

```
ariadne$ opt=T
ariadne$ echo options: ${opt:+"list mode"}
options: list mode
ariadne$ opt=
ariadne$ echo options: ${opt:+"list mode"} :
options: :
ariadne$
```

8.6 Das *set*-Kommando

Dies dient einmal zum Setzen von Shell-Optionen, zum Anzeigen von allen existierenden Variablen in der jeweiligen Umgebung und zum **Neu-Definieren der Positionsparameter.**

- die **-x** Option (trace)

Beispiel:

```
ariadne$ set -x
ariadne$ x=*
x=*
ariadne$ echo $x
+ echo add add.old add1 db ... vartest2 vartest3 wcl
add add.old add1 db  ... vartest2 vartest3 wcl

ariadne$ cmd=wc
cmd=wc
ariadne$ ls | $cmd -l
+ ls
+ wc -l
      57
ariadne$ set +x
+ set +x
ariadne$ ls | wc -l
      57
ariadne$ set
CDPATH=
DISPLAY=:0
EDITOR=/usr/bin/vi
FRAMEBUFFER=/dev/fb
HELPPATH=/usr/openwin/lib/help
HOME=/home/laborix/swg
ICON_PATH=/usr/local/lib/x11/icons
IFS=
 ...
LOGNAME=swg
NEWSSERVER=2252096030.2000;ariadne
PATH=:/home/laborix/swg/bin: ...
PS1=ariadne$
PS2=>
SAIBIN=/home/laborix/sai/bin
SAIHOME=/home/laborix/sai
SHELL=/bin/sh
TERM=sun-cmd
USER=swg
WINDOW_PARENT=/dev/win0
WMGR_ENV_PLACEHOLDER=/dev/win3
cmd=wc
x=*
ariadne$
```

•

Zuweisung neuer Werte an Positionsvariable:

Beispiele:

```
ariadne$ set a b cd
ariadne$ echo :$1: :$2: :$3:
:a: :b: :cd:
ariadne$ echo $#
3
ariadne$ for args do echo $args; done
a
b
cd

ariadne$ cat words
# Worte auf einer Zeile zaehlen

read line
set $line
echo "$# Worte waren es:"
for i
do
        echo $i
done
ariadne$ words
Dies sind die eingegebenen Worte
5 Worte waren es:
Dies
sind
die
eingegebenen
Worte
ariadne$
```

- Die **--** Option:

Beispiel:

```
ariadne$ words
-Grade sind sehr unangenehm
words: -Grade: bad option(s)

ariadne$ words

CDPATH=
DISPLAY=:0
EDITOR=/usr/bin/vi
 ...
line=
x=*
0 Worte waren es:
ariadne$
```

Im ersten Fall wurde nach der *read*-Anweisung das Kommando *set* ausgeführt. Nach der Substitution von *line* sieht die Kommando-Zeile wie folgt aus:

```
set -Grade sind sehr unangenehm
```

set sieht also ein "-"-Zeichen und hält dies verständlicherweise für eine Option.

Im zweiten Fall sieht *set* eine white-Space-Zeile oder eine leere Zeile (s.o.)

Mit der "--" Option unterläßt *set* jegliche Interpretation der folgenden Zeichen:

```
set -- $line
```

set ist ebenfalls ein sog. built-in-Kommando. Mehr zu *set:* **man set**

8.7 Die *IFS*-Variable

IFS: Internal Field Seperator

Achtung: Vor Veränderung die alten Werte merken (manche Programme vertrauen auf white space als Trenner).

```
ariadne$ echo "$IFS" | od -b
0000000 040 011 012 012
0000004
ariadne$ read line
        Dies ist eine Zeile mit fuehrenden Blanks
ariadne$ echo "$line"
Dies ist eine Zeile mit fuehrenden Blanks
ariadne$ IFSOLD="$IFS"
ariadne$ IFS="
> "
ariadne$ read line
        Dies ist eine Zeile mit fuehrenden Blanks
ariadne$ echo "$line"
        Dies ist eine Zeile mit fuehrenden Blanks
ariadne$ line="Uni Ulm:SAI:Prof.Dr.F.Schweiggert:"
ariadne$ IFS=:
ariadne$ set $line
ariadne$ for args do echo $args ; done
Uni Ulm
SAI
Prof.Dr.F.Schweiggert
ariadne$ IFS="$IFSOLD"
ariadne$
```

bei den neueren Shells ist der ‘tab’ nicht mehr Trenner, sondern dient der Dateinamenexpansion!

8.8 *unset*

Dient zum Entfernen von Variablen-Definitionen im Environment

9. Weitere Kommandos

9.1 *eval*

Format: **eval** *command_line*

Der Effekt ist, daß die Shell die Kommandozeile **zweimal** untersucht: einmal zur Bestimmung der Argumente von *eval,* dann zur Ausführung von *eval.*

```
ariadne$ pipe="|"
ariadne$ ls $pipe wc -l
ls: |: No such file or directory
ls: wc: No such file or directory
ls: -l: No such file or directory

ariadne$ eval ls $pipe wc -l
      59
ariadne$
```

Die Shell sucht nach Pipes und I/O-Umlenkung **bevor** Variablensubstitution durchgeführt wird. Im ersten Fall erkennt die Shell das Zeichen "|" daher nicht, "|", "wc" und "-l" werden *ls* als Argumente zugeführt.

Mit *eval* wird die Zeile letztlich zweimal durchsucht und das "|" erkannt.

eval ist besonders nützlich, wenn in einem Shell-Programm Kommandos als Variablenwerte aufgebaut werden. Wenn die Variablen Zeichen (wie ;, | &, >, < oder Quotes) enthalten, die von der Shell direkt gesehen werden müssen (nicht als Ergebnis einer Variablen-Substitution, so kann dies über *eval* erreicht werden.

Beispiel: Ein Programm *last,* daß das letzte übergebene Argument (max. 9) anzeigt

```
ariadne$ cat last
eval echo \$$#
ariadne$ last a b c d e f g
g
ariadne$

ariadne$ cat argpos
arg=5
eval echo \$$arg
ariadne$ argpos a b c d 'e f' g h
e f
ariadne$
```

Pointer

```
ariadne$ x=100
ariadne$ ptrx=x
ariadne$ eval echo \$$ptrx
100
ariadne$ eval $ptrx=50
ariadne$ echo $x
50
ariadne$
```

9.2 *wait*

Format: **wait** *process_id*

Wirkung: Wird ein Programm mit **&** im Hintergrund ausgeführt, so wird dies von einer Subshell als Abkömmling der Vater-Shell erledigt. Mit *wait* wartet der Vater auf die von ihm erzeugte Subshell (*process_id* ist die Prozeßnummer des Prozesses, auf dessen Beendigung gewartet wird).

Läuft nur ein Prozess im Hintergrund, so genügt *wait* ohne Angabe der Prozeßnummer.

9.3 Die $!-Variable

In dieser Variablen speichert die Shell die Prozeß-Id des letzten in den Hintergrund geschickten Kommandos.

Beispiel:

```
ariadne$ prog1 &
3333
ariadne$ pid1=$!
 ...
ariadne$ prog2
4444
ariadne$ pid2=$!
 ...
ariadne$ wait $pid1
```

9.4 Signale: *trap*

Format: **trap** *commands signals*

Durch Drücken z.B. der DELETE- oder BREAK-Taste wird ein Signal an den laufenden Prozess geschickt, auf das dieser auch gezielt reagieren kann (statt zu beenden).

Signal	Generated for
0	Exit from the shell
1	Hangup
2	Interrupt (e.g. DELETE key
3	Quit
4	Illegal Instruction
.	...
.	...
.	...
13	Write to a pipe without a process to read it
14	Alarm timeout
15	Software termination signal (kill)

Beispiel:

```
 ...
 trap "rm $WORKDIR/work1$$ $WORKDIR/res$$; exit " 1 2
 ...
```

Sobald das *trap*-Kommando in diesem Shellprogramm ausgeführt ist, werden die angegeben Datei automatisch gelöscht, wenn Signal 1 (*hangup*) oder Signal 2 (Interrupt) empfangen werden. Das *exit* ist notwendig, denn ohne würde das Programm "munter" an dem Punkt weiter machen, an dem es unternrochen wurde.

In obigem Beispiel würde wegen der angegeben Quotierung die Shell beim Erreichen des *trap*-Kommandos die Zeile sofort interpretieren (Variablensubstitution). Soll dies erst zu dem Zeitpunkt geschehen, an dem die Signale empfangen werden, müßte wie folgt quotiert werden:

```
trap 'rm $WORKDIR/work1$$ $WORKDIR/res$$; exit ' 1 2
```

9.5 *trap* ohne Argumente

Beispiel:

```
ariadne$ trap 'echo logged off at `date` >> $HOME/logoffs' 0
ariadne$ trap
0: echo logged off at `date` >>$HOME/logoffs
ariadne$ <ctrl-d>
 ...
```

Erneutes login:

```
ariadne$ cat $HOME/logoffs
logged off at Fri Jan 24 11:41:16 MET 1992
ariadne$
```

9.6 Signale ignorieren

In der Form **trap "" 2** (also kein Kommando) wird das Interrupt-Signal ignoriert. So können Teile, in denen ein Interrupt unerwünscht ist, geschützt werden.

Das Ignorieren eines Signals überträgt sich auf alle erzeugten Subshells. Ist allerdings ein Kommando angegeben, so haben die Subshells den trap-Defaultwert.

9.7 Zurücksetzen von *trap*s

```
trap 1 2
```

Damit wird die Aktion bei Empfangen von Signal 1 oder 2 auf den Default-Wert zurückgesetzt.

□

9.8 Mehr zu I/O

Beispiele: (zur Erinnerung: Filedeskriptor 0 steht für *stdin,* 1 für *stdout,* 2 für *stderr)*

- Umlenken von *stderr* auf Datei:

  ```
  command 2>file
  ```

- *stdout* auf *stderr* lenken:

  ```
  command >& 2
  ```

- Vorübergehendes Umlenken von *stdin* aus Datei:

  ```
  exec < file
  ```

- Dito für *stdout*

  ```
  exec > file
  ```

- Dito für *stderr*

  ```
  exec 2> errors
  ```

9.9 In-Line Input Redirection

Beispiel:

```
command << word
```

Die Shell benutzt die folgenden Zeilen bis zur Zeile, die nur *word* enthält, als Standardeingabe.

```
ariadne$ wc -l <<ENDOFINPUT
> dies ist die erste,
> dies die zweite,
> dies die dritte Zeile
> ENDOFINPUT
       3

ariadne$ cat mailmsg
mail $* <<END-OF-MSG

Attention please
        Next week ...
        ....

END-OF-MSG

ariadne$ cat <<Text
> $HOME
> irgendwas
> `date`
> Text
/home/laborix/swg
irgendwas
Fri Jan 24 12:22:00 MET 1992

ariadne$ cat <<\Text
> $HOME
> irgendwas
> `date`
> Text
$HOME
irgendwas
`date`
```

Mit dem Backslash wird die Shell veranlaßt, die Zeile bis zur angegebenen Stelle nicht zu interpretieren.

```
ariadne$ cat <<END
>       mit tab eingerueckt
>       dito
> END
mit tab eingerueckt
dito
ariadne$
```

Im Folgenden sollen diese Konzepte an einem Beispiel gezeigt werden. Es geht darum, verschiedene Dateien in einem Archiv-File (z.B. zum Versenden via mail) zu verpacken und zwar so, daß es "sehr leicht" wieder "ausgepackt" werden kann.

```
ariadne$ cat shar
#
# Program to create a shell archive from a set of files
#

echo "#"
echo "# to restore, type sh archive"
echo "#"

for file
do
        echo
        echo "echo Extracting $file"
        echo "cat >$file <<\END-OF-DATA"
        cat $file
        echo "END-OF-DATA"
done

ariadne$ ls var*
vartest   vartest1  vartest2  vartest3

ariadne$ shar var* >archive

ariadne$ rm var*

ariadne$ cat archive
#
# to restore, type sh archive
#

echo Extracting vartest
cat >vartest <<\END-OF-DATA
echo :$x:
END-OF-DATA

echo Extracting vartest1
cat >vartest1 <<\END-OF-DATA
```

```
x=30
echo :$x:
END-OF-DATA

echo Extracting vartest2
cat >vartest2 <<\END-OF-DATA
u=10
y=20
z=30
export z
vartest3
END-OF-DATA

echo Extracting vartest3
cat >vartest3 <<\END-OF-DATA
echo y = $y
echo z = $z
echo u = $u

END-OF-DATA

ariadne$ sh archive
Extracting vartest
Extracting vartest1
Extracting vartest2
Extracting vartest3

ariadne$ ls var*
vartest   vartest1  vartest2  vartest3

ariadne$ cat vartest
echo :$x:
ariadne$
```

10. Funktionen

Format: **name() { command; ...; command; }**

name ist der Funktionsname, die runden Klammern zeigen der Shell, daß eine Funktion definiert wird, die Kommando's zwischen den geschweiften Klammern bilden den Funktionsrumpf. Das Blank nach **{** ist notwendig, das Semikolon vor **}** ist notwendig, falls das letzte Kommando in derselben Zeile wie **}** steht.

Beispiel:

```
ariadne$ nf () { ls | wc -l; }
ariadne$ nf
      62
ariadne$
```

Die Ausführung von Funktionen ist schneller als die von Shell-Skripten in Dateien (Suchen nach Dateien auf Platte, Öffnen von Dateien, Übertragen in Hauptspeicher entfällt). Daher bietet es sich an, gewisse Funktionsdefinitionen im unterzubringen und sie zur Verfügung zu haben nach dem *login.*

Beispiel:

```
#
# telmen        a program to select, add, and remove someone
#               from phones (the phonebook)
#
#               if there are arguments it's assumed a look up is ment
#
#               necessary commands are: select, add, rem (remove)
#

# Functions:

add () {

        # add commands
}

rem () {

        # rem commands
}

# and so on

# the main programm with function calls
```

Entfernen von Funktionsdefinitionen:

```
unset <function name>
```

return - Kommando

Wird innerhalb einer Funktion das terminiert, sondern auch das rufende Shell-Programm. Zur Beendigung der Funktion allein dient das **return**-Kommando:

Format: **return n**

Der Wert **n** ist der Return-Status der Funktion. Wenn er weggelassen wird, ist dies der EXIT-Status des zuletzt ausgeführten Kommandos. Der Return-Status kann genauso abgefragt werden wie der Exit-Status: **$?, if, while, until**

11. *type*

Dieses dient zur Feststellung, ob ein Kommando eine Funktion, ein Shell-Programm oder ein built-in-Kommando ist.

Beispiel:

```
ariadne$ nu () { who | wc -l; }
ariadne$ type nu
nu is a function
nu(){
who  | wc -l
}
ariadne$ type pwd
pwd is a shell builtin
ariadne$ type cd
cd is a shell builtin
ariadne$ type cat
cat is /usr/5bin/cat
ariadne$
```

12. Zusammenfassendes Beispiel

telmen:

```
#!/bin/sh
# -------------------- telmen -------------------------------

# a program to select, add, and remove someone
# from phones (the phonebook) as well as allow changes
#
# if there are arguments it’s assumed a select is meant
#
# necessary commands are: getentry, add, rem, change,
#       display, listall, tel_update, get_adr
#

# Set PHONEBOOK / OLDPHONEBOOK to point to the phonebook file
# if not yet defined and export it for use in subshells

: ${PHONEBOOK:=./phones}
: ${OLDPHONE:=./phones.old}

# Field Separator
SEP=’^’

# : -> leeres Kommando, erzwingt Bewertung der Argumente
# ${parameter:=value} -> Skript 8.2, S. 138
# Wenn parameter einen Wert hat, passiert weiter nichts;
# wenn nicht, so wird value substituiert und gleichzeitig
# an parameter zugewiesen

export PHONEBOOK
export OLDPHONE
export SEP

if [ ! -f "$PHONEBOOK" ]
then
   printf "%s does not ex. or is not an ordinary file!\n" "$PHONEBOOK"
   printf "Should I create it for you (y/n)? "
   read answer

   # read -> Skript 6., S. 123 ff
   # liest von der Standardeingabe eine Zeile und weist Wort
   # fuer Wort an die Variablen der read-Anweisung zu

   if [ "$answer" != y ]
   then
        exit 1
   fi
   > $PHONEBOOK || exit 1  # exit if creation fails
fi

# if arguments are supplied, then do a select

if [ $# -ne 0 ]
then
        name="$1"
        get_adr "$name" # get_adr:  another shell script
```

```
        exit 0
fi

# set trap on interrupt (DEL-key) to continue the loop

trap "continue" 2

# trap -> Skript 9.4 S. 144
# trap comands signals
# Signale abfangen

#
# Loop until user selects ‘exit’
#

while true # loop forever
do
        clear
        # Display menue

        printf ’
        Would you like to:

                (1)     Look someone up
                (2)     Add someone to the phonebook
                (3)     Remove someone from the phonebook
                (4)     Change an entry
                (5)     List all
                (6)     Update with new file
                (7)     Exit

        Please select one of the above (1-7): ’

        # Read and process selection

        read choice
        printf "\n"

        case "$choice" in
                1)      printf "Enter name to look up: "
                        read name

                        if [ -z "$name" ]
                        then
                                printf "select ignored\n"
                        else
                                getentry "$name"
                        fi ;;

                2)      cp $PHONEBOOK $OLDPHONE
                        add ;;


                3)      printf "Enter name to be removed: "
                        read name
                        cp $PHONEBOOK $OLDPHONE

                        if [ -z "$name" ]
                        then
                                printf "Removal ignored\n"
```

```
                        else
                                rem "$name"
                        fi ;;

                4)      printf "Enter name to change: "
                        read name
                        cp $PHONEBOOK $OLDPHONE
                        if [ -z "$name" ]
                        then
                                printf "Change ignored\n"
                        else
                                change "$name"
                        fi ;;

                5)      listall ;;

                6)      tel_update ;;

                7)      exit 0 ;;

                *)      printf "Bad choice\007\n"
                        printf "Press enter\n"
                        read enter ;;
        esac
done
```

get_adr:

```
# ----------------- get_adr --------------
# list an entry from the phonebook 'phones'
#

name=$1
grep "$name" $PHONEBOOK >/tmp/matches$$
# $$ -> Prozess-ID

# -s : file existiert und hat Laenge > 0

if [ ! -s /tmp/matches$$ ]
then
   printf "Can't find %s in the phone book\n" "$name"
   printf "Press enter to continue"
   read enter </dev/tty
   # </dev/tty: falls wer auch immer stdin
   # vom Terminal wegegebogen hat!
else
   #
   # Display each of the matching entries
   #
   IFS="$SEP"
   # fuer diesen Prozess wird der IFS umdefiniert
   # -> Aufsplitten einer Zeile
   cat /tmp/matches$$ | while read line
   do
```

```
      set $line
      for line in "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9"
      do
      printf "%s\n" "$line"
      done
   done | more
fi
rm /tmp/matches$$
```

add:

```
# ------------------- add ----------------------------------
# jemanden in phones aufnehmen
#

printf "Type in the new entry, "
printf "each line one of your fields\n"
printf "When done, type just a single "
printf "RETURN on the line\n"

first=
entry=
firstfield=
while true
do
   printf ">> "
   read line

   if [ -n "$line" ]
   then
      if [ -z "$firstfield" ]
      then
         entry="$line"
         firstfield=no
      else
         entry="$entry$SEP$line"
      fi
        # who (first field) has to be added?
      if [ -z "$first" ]
      then
         first="$line"
      fi
   else
      break
   fi
done

printf "%s\n" "$entry" >> $PHONEBOOK
sort -o $PHONEBOOK $PHONEBOOK
printf "\n%s has been added\n" "$first"
printf "Press enter to continue\n"
```

```
read enter </dev/tty
```

getentry:

```
# -------------- getentry --------------
# select someone form the phone book
#

name="$1"
grep "$name" $PHONEBOOK >/tmp/matches$$

if [ ! -s /tmp/matches$$ ]
then
   printf "Can't find %s in the phone book\n" "$name"
   printf "Press enter to continue"
   read enter </dev/tty
else
   #
   # Display each of the matching entries
   #

   while read line
   do
      display "$line"
   done < /tmp/matches$$
fi
rm /tmp/matches$$
```

select ist ein System Call! **display:**

```
# --------------- display -------------------
#
# Display an entry from the phonebook `phones'
#

printf "\n------------------------------------------------\n"

entry=$1
IFS="$SEP"
set $entry

for line in "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9"
do
        printf "\t\t\t\t\t\t|\r| %s\n" "$line"
done

printf "|\t\to\t\to\t\t|\n"
printf "-----------------------------------------------\n\n"
printf "Press enter\n"
```

```
read enter < /dev/tty
```

rem

```
# ----------------- rem -----------
#
# Remove someone from the phonebook
#

name=$1

#
# Get matching entries
#
grep "$name" $PHONEBOOK > /tmp/matches$$

if [ ! -s /tmp/matches$$ ]
then
   printf "Can't find %s in the phonebook\n" "$name"
   printf "Press enter to continue"
   read enter </dev/tty
   exit 1
fi

#
# Display matching entries and confirm removal
#

exec </tmp/matches$$

while read line
do
   display "$line"
   printf "Remove this entry (y/n)? "
   read answer </dev/tty

   if [ "$answer" = y ]
   then
      break
   fi
done

rm /tmp/matches$$

if [ "$answer" = y ]
then
   if grep -v "^$line$" $PHONEBOOK > /tmp/phones$$
   then
      mv /tmp/phones$$ $PHONEBOOK
      printf "selected entry is removed\n"
```

```
   else
      printf "Nothing removed\n"
   fi
   printf "Press enter to continue"
   read enter </dev/tty
fi
```

change:

```
# ------------- change --------------------------
#
# change an entry in the phonebook
#

name=$1

#
# Get matching entries and save in temp file
#

grep "$name" $PHONEBOOK >/tmp/matches$$

if [ ! -s /tmp/matches$$ ]
then
   printf "Can't find %s in the phonebook\n" "$name"
   printf "Press enter to continue"
   read enter </dev/tty
   exit 1
fi

#
# Display matching entries one at a time and confirm changes
#

exec < /tmp/matches$$      # reassign stdin
# exec -> Skript 7.5 (S. 135), 7.7 (S.136)

while read line
do
   display "$line"
   printf "Change this entry (y/n) ? "
   read answer < /dev/tty

   if [ "$answer" = y ]
   then
      break
   fi
done

rm /tmp/matches$$

if [ "$answer" != y ]
then
   exit 0
fi

#
# start up editor on confirmed entry
#

exec < /dev/tty      # reassign stdin for editing
printf "%s" "$line" | tr "$SEP" '\012' > /tmp/ed$$
EDITOR=vi
printf "Enter changes with %s\n" "$EDITOR"
trap "" 2      #Don't abort if DEL hit while editing
$EDITOR /tmp/ed$$

#
# Remove old entry now and insert new one
```

```
#
grep -v "^$line$" $PHONEBOOK > /tmp/phones$$
{ tr '\012' "$SEP" < /tmp/ed$$; printf "\n"; } | sed 's/\'"$SEP"'$//' >> /tmp/phones$$
# last printf was to put back trailing new line translated by tr

sort /tmp/phones$$ -o $PHONEBOOK
rm /tmp/ed$$ /tmp/phones$$
```

listall:

```
# --------------- listall -----------------
# list all entries
# Get the first and the last fields,
# presumably names and numbers

gawk '
BEGIN { FS='\""$SEP"\"' }
      { print $1 " : " $NF }
' $PHONEBOOK | more
printf "Press enter to continue"
read enter </dev/tty
```

tel_update

```
printf "Give new filename - it must be located in current directory\n"
read file
if [ ! -f $file ]
then
        printf "your file doesn't exist!\n"
        exit 1
fi
cp $PHONEBOOK $OLDPHONE
cat $file >> $PHONEBOOK
sort -u -o $PHONEBOOK $PHONEBOOK
exit 0
```

12.1 Anhang: SHELL BUILTIN COMMANDS

(nach dem Manual der GNU bash unter Linux)

12.2 :

: [*arguments*]

No effect; the command does nothing beyond expanding *arguments* and performing any specified redirections. A zero exit code is returned.

12.3 ., source

. *filename* **[***arguments***]**

source *filename* [*arguments*]

Read and execute commands from *filename* in the current shell environment and return the exit status of the last command executed from *filename*. If *filename* does not contain a slash, pathnames in **PATH** are used to find the directory containing *filename*. The file searched for in **PATH** need not be executable. The current directory is searched if no file is found in **PATH**. If any *arguments* are supplied, they become the positional parameters when *file* is executed. Otherwise the positional parameters are unchanged. The return status is the status of the last command exited within the script (0 if no commands are executed), and false if *filename* is not found.

12.4 alias

alias [*name*[=*value*] ...]

Alias with no arguments prints the list of aliases in the form *name*=*value* on standard output. When arguments are supplied, an alias is defined for each *name* whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution when the alias is expanded. For each *name* in the argument list for which no *value* is supplied, the name and value of the alias is printed. **Alias** returns true unless a *name* is given for which no alias has been defined.

12.5 bg

bg [*jobspec*]

Place *jobspec* in the background, as if it had been started with **&**. If *jobspec* is not present, the shell's notion of the *current job* is used. **bg** *jobspec* returns 0 unless run when job control is disabled or, when run with job control enabled, if *jobspec* was not found or started without job control.

12.6 bind

bind [**-m** *keymap*] [**-lvd**] [**-q** *name*]

bind [**-m** *keymap*] *-f filename*

bind [**-m** *keymap*] *keyseq:function-name*

Display current **readline** key and function bindings, or bind a key sequence to a *readline* function or macro. The binding syntax accepted is identical to that of *.inputrc*, but each binding must be passed as a separate argument; e.g., '"\C-x\C-r":re-read-init-file'. Options, if supplied, have the following meanings:

-m *keymap*
Use *keymap* as the keymap to be affected by the subsequent bindings. Acceptable *keymap* names are *emacs, emacs-standard, emacs-meta, emacs-ctlx, vi, vi-move, vi-command,* and *vi-insert. vi* is equivalent to *vi-command*; *emacs* is equivalent to *emacs-standard*.
-l
List the names of all *readline* functions
-v
List current function names and bindings
-d
Dump function names and bindings in such a way that they can be re-read
-f *filename*
Read key bindings from *filename*
-q *function*
Query about which keys invoke the named *function*
The return value is 0 unless an unrecognized option is given or an error occurred.

12.7 break

break [*n*]
Exit from within a **for**, **while**, or **until** loop. If *n* is specified, break *n* levels. *n* must be >= 1. If *n* is greater than the number of enclosing loops, all enclosing loops are exited. The return value is 0 unless the shell is not executing a loop when **break** is executed.

12.8 builtin

builtin *shell-builtin* [*arguments*]
Execute the specified shell builtin, passing it *arguments*, and return its exit status. This is useful when you wish to define a function whose name is the same as a shell builtin, but need the functionality of the builtin within the function itself. The **cd** builtin is commonly redefined this way. The return status is false if *shell-builtin* is not a shell builtin command.

12.9 cd

cd [*dir*]
Change the current directory to *dir*. The variable **HOME** is the default *dir*. The variable **CDPATH** defines the search path for the directory containing *dir*. Alternative directory names are separated by a colon (**:**). A null directory name in **CDPATH** is the same as the current directory, i.e., '**.**'. If *dir* begins with a slash (**/**), then **CDPATH** is not used. An

argument of **-** is equivalent to **$OLDPWD**. The return value is true if the directory was successfully changed; false otherwise.

12.10 command

command [**-pVv**] *command* [*arg* ...]

Run *command* with *args* suppressing the normal shell function lookup. Only builtin commands or commands found in the **PATH** are executed. If the **-p** option is given, the search for *command* is performed using a default value for **PATH** that is guaranteed to find all of the standard utilities. If either the **-V** or **-v** option is supplied, a description of **command** is printed. The **-v** option causes a single word indicating the command or pathname used to invoke *command* to be printed; the **-V** option produces a more verbose description. An argument of **--** disables option checking for the rest of the arguments. If the **-V** or **-v** option is supplied, the exit status is 0 if *command* was found, and 1 if not. If neither option is supplied and an error occurred or *command* cannot be found, the exit status is 127. Otherwise, the exit status of the **command** builtin is the exit status of *command*.

12.11 continue

continue [*n*]

Resume the next iteration of the enclosing **for**, **while**, or **until** loop. If *n* is specified, resume at the *n*th enclosing loop. *n* must be >= 1. If *n* is greater than the number of enclosing loops, the last enclosing loop (the 'top-level' loop) is resumed. The return value is 0 unless the shell is not executing a loop when **continue** is executed.

12.12 declare, typeset

declare [**-frxi**] [*name*[=*value*]]

typeset [**-frxi**] [*name*[=*value*]]

Declare variables and/or give them attributes. If no *name*s are given, then display the values of variables instead. The options can be used to restrict output to variables with the specified attribute.

-f

Use function names only

-r

Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements.

-x

Mark *name*s for export to subsequent commands via the environment.

-i

The variable is treated as an integer; arithmetic evaluation (see

ARITHMETIC EVALUATION) is performed when the variable is assigned a value.

Using '+' instead of '-' turns off the attribute instead. When used in a function, makes *name*s local, as with the **local** command. The return value is 0 unless an illegal option is encountered, an attempt is made to define a function using "-f foo=bar", one of the *name*s is not a legal shell variable name, an attempt is made to turn off read-only status for a readonly variable, or an attempt is made to display a non-existant function with -f.

12.13 dirs

dirs [**-l**] [**+/-n**]

Display the list of currently remembered directories. Directories are added to the list with the **pushd** command; the **popd** command moves back up through the list.

+n

displays the *n*th entry counting from the left of the list shown by **dirs** when invoked without options, starting with zero.

-n

displays the *n*th entry counting from the right of the list shown by **dirs** when invoked without options, starting with zero.

-l

produces a longer listing; the default listing format uses a tilde to denote the home directory.

The return value is 0 unless an illegal option is supplied or *n* indexes beyond the end of the directory stack.

12.14 echo

echo [**-neE**] [*arg* ...]

Output the *arg*s, separated by spaces. The return status is always 0. If **-n** is specified, the trailing newline is suppressed. If the **-e** option is given, interpretation of the following backslash-escaped characters is enabled. The **-E** option disables the interpretation of these escape characters, even on systems where they are interpreted by default.

\a alert (bell)
\b backspace
\c suppress trailing newline
\f form feed
\n new line
\r carriage return
\t horizontal tab

\v vertical tab
**** backslash
\nnn the character whose ASCII code is *nnn* (octal)

12.15 enable

enable [**-n**] [**-all**] [*name* ...]

Enable and disable builtin shell commands. This allows the execution of a disk command which has the same name as a shell builtin without specifying a full pathname. If **-n** is used, each *name* is disabled; otherwise, *name*s are enabled. For example, to use the **test** binary found via the **PATH** instead of the shell builtin version, type 'enable -n test'. If no arguments are given, a list of all enabled shell builtins is printed. If only **-n** is supplied, a list of all disabled builtins is printed. If only **-all** is supplied, the list printed includes all builtins, with an indication of whether or not each is enabled. **enable** accepts **-a** as a synonym for **-all**. The return value is 0 unless a *name* is not a shell builtin.

12.16 eval

eval [*arg* ...]

The *arg*s are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of the **eval** command. If there are no *arg*s, or only null arguments, **eval** returns true.

12.17 exec

exec [[**-**] *command* [*arguments*]]

If *command* is specified, it replaces the shell. No new process is created. The *arguments* become the arguments to *command*. If the first argument is **-**, the shell places a dash in the zeroth arg passed to *command*. This is what login does. If the file cannot be executed for some reason, a non-interactive shell exits, unless the shell variable **no_exit_on_failed_exec** exists, in which case it returns failure. An interactive shell returns failure if the file cannot be executed. If *command* is not specified, any redirections take effect in the current shell, and the return status is 0.

12.18 exit

exit [*n*]

Cause the shell to exit with a status of *n*. If *n* is omitted, the exit status is that of the last command executed. A trap on **EXIT** is executed before the shell terminates.

12.19 export

export [**-nf**] [*name*[=*word*]] ...

export -p

The supplied *name*s are marked for automatic export the environment of subsequently executed commands. If the **-f** option is given, the *name*s refer to functions. If no *name*s are given, or if the *-p* option is supplied, a list of all names that are exported in this shell is printed. The *-n* option causes the export property to be removed from the named variables. An argument of **--** disables option checking for the rest of the arguments. **export** returns an exit status of 0 unless an illegal option is encountered, one of the *name*s is not a legal shell variable name, or **-f** is supplied with a *name* that is not a function.

12.20 fc

fc [**-e** *ename*] [**-nlr**] [*first*] [**last**]

fc -s [*pat*=*rep*] [*cmd*]

Fix Command. In the first form, a range of commands from *first* to *last* is selected from the history list. *First* and *last* may be specified as a string (to locate the last command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If *last* is not specified it is set to the current command for listing (so that **fc -l -10** prints the last 10 commands) and to *first* otherwise. If *first* is not specified it is set to the previous command for editing and -16 for listing.

The **-n** flag suppresses the command numbers when listing. The **-r** flag reverses the order of the commands. If the **-l** flag is given, the commands are listed on standard output. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the **FCEDIT** variable is used, and the value of **EDITOR** if **FCEDIT** is not set. If neither variable is set, *vi* is used. When editing is complete, the edited commands are echoed and executed.

In the second form, *command* is re-executed after each instance of *pat* is replaced by *rep*. A useful alias to use with this is ‘‘r=fc -s’’, so that typing ‘‘r cc’’ runs the last command beginning with ‘‘cc’’ and typing ‘‘r’’ re-executes the last command.

If the first form is used, the return value is 0 unless an illegal option is encountered or *first* or *last* specify history lines out of range. If the **-e** option is supplied, the return value is the value of the last command executed or failure if an error occurs with the temporary file of commands. If the second form is used, the return status is that of the command re-executed, unless *cmd* does not specify a valid history line, in which case **fc** returns failure.

12.21 fg

fg [jobspec]

Place *jobspec* in the foreground, and make it the current job. If *jobspec* is not present, the shell's notion of the *current job* is used. The return value is that of the command placed into the foreground, or failure if run when job control is disabled or, when run with job control enabled, if *jobspec* does not specify a valid job or *jobspec* specifies a job that was started without job control.

12.22 getopts

getopts *optstring name* [*args*]

getopts is used by shell procedures to parse positional parameters. *optstring* contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. Each time it is invoked, **getopts** places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable **OPTIND**. **OPTIND** is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, **getopts** places that argument into the variable **OPTARG**. The shell does not reset **OPTIND** automatically; it must be manually reset between multiple calls to **getopts** within the same shell invocation if a new set of parameters is to be used.

getopts can report errors in two ways. If the first character of *optstring* is a colon, *silent* error reporting is used. In normal operation diagnostic messages are printed when illegal options or missing option arguments are encountered. If the variable **OPTERR** is set to 0, no error message will be displayed, even if the first character of *optstring* is not a colon.

If an illegal option is seen, **getopts** places ? into *name* and, if not silent, prints an error message and unsets **OPTARG**. If **getopts** is silent, the option character found is placed in **OPTARG** and no diagnostic message is printed.

If a required argument is not found, and **getopts** is not silent, a question mark (**?**) is placed in *name*, **OPTARG** is unset, and a diagnostic message is printed. If **getopts** is silent, then a colon (**:**) is placed in *name*

and **OPTARG** is set to the option character found.
getopts normally parses the positional parameters, but if more arguments are given in *args*, **getopts** parses those instead. **getopts** returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs.

12.23 hash

hash [**-r**] [*name*]
For each *name*, the full pathname of the command is determined and remembered. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is printed. An argument of **--** disables option checking for the rest of the arguments. The return status is true unless a *name* is not found or an illegal option is supplied.

12.24 help

help [*pattern*]
Display helpful information about builtin commands. If *pattern* is specified, **help** gives detailed help on all commands matching *pattern*; otherwise a list of the builtins is printed. The return status is 0 unless no command matches *pattern*.

12.25 history

history [*n*]

history -rwan [*filename*]
With no options, display the command history list with line numbers. Lines listed with a * have been modified. An argument of *n* lists only the last *n* lines. If a non-option argument is supplied, it is used as the name of the history file; if not, the value of **HISTFILE** is used. Options, if supplied, have the following meanings:

-a Append the ‘‘new’’ history lines (history lines entered since the beginning of the current **bash** session) to the history file

-n Read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current **bash** session.

-r Read the contents of the history file and use them as the current history

-w Write the current history to the history file, overwriting the history file's contents.
The return value is 0 unless an illegal option is encountered or an error occurs while reading or writing the history file.

12.26 jobs

jobs [**-lnp**] [*jobspec* ...]

jobs -x *command* [*args* ...]

The first form lists the active jobs. The **-l** option lists process IDs in addition to the normal information; the **-p** option lists only the process ID of the job's process group leader. The **-n** option displays only jobs that have changed status since last notified. If *jobspec* is given, output is restricted to information about that job. The return status is 0 unless an illegal option is encountered or an illegal *jobspec* is supplied.

If the **-x** option is supplied, **jobs** replaces any *jobspec* found in *command* or *args* with the corresponding process group ID, and executes *command* passing it *args*, returning its exit status.

12.27 kill

kill [**-s sigspec** | **-sigspec**] [*pid* | *jobspec*] ...

kill -l [*signum*]

Send the signal named by *sigspec* to the processes named by *pid* or *jobspec*. *sigspec* is either a signal name such as **SIGKILL** or a signal number. If *sigspec* is a signal name, the name is case insensitive and may be given with or without the **SIG** prefix. If *sigspec* is not present, then **SIGTERM** is assumed. An argument of **-l** lists the signal names. If any arguments are supplied when **-l** is given, the names of the specified signals are listed, and the return status is 0. An argument of **--** disables option checking for the rest of the arguments. **kill** returns true if at least one signal was successfully sent, or false if an error occurs or an illegal option is encountered.

12.28 let

let *arg* [*arg* ...]

Each *arg* is an arithmetic expression to be evaluated (see **ARITHMETIC EVALUATION**). If the last *arg* evaluates to 0, **let** returns 1; 0 is returned otherwise.

12.29 local

local [*name*[=*value*] ...]

For each argument, create a local variable named *name*, and assign it *value*. When **local** is used within a function, it causes the variable *name* to have a visible scope restricted to that function and its children. With no operands, **local** writes a list of local variables to the standard output. It is an error to use **local** when not within a function. The return status is 0 unless **local** is used outside a function, or an illegal *name* is supplied.

12.30 logout

logout

Exit a login shell.

12.31 popd

popd [**+/-n**]

Removes entries from the directory stack. With no arguments, removes the top directory from the stack, and performs a **cd** to the new top directory.

+n removes the *n*th entry counting from the left of the list shown by **dirs**, starting with zero. For example: ‘‘popd +0’’ removes the first directory, ‘‘popd +1’’ the second.

-n removes the *n*th entry counting from the right of the list shown by **dirs**, starting with zero. For example: ‘‘popd -0’’ removes the last directory, ‘‘popd -1’’ the next to last.

If the **popd** command is successful, a **dirs** is performed as well, and the return status is 0. **popd** returns false if an illegal option is encountered, the directory stack is empty, a non-existent directory stack entry is specified, or the directory change fails.

12.32 pushd

pushd [*dir*]

pushd +/-n

Adds a directory to the top of the directory stack, or rotates the stack, making the new top of the stack the current working directory. With no arguments, exchanges the top two directories and returns 0, unless the directory stack is empty.

+n Rotates the stack so that the *n*th directory (counting from the left of the list shown by **dirs**) is at the top.

-n Rotates the stack so that the *n*th directory (counting from the right) is at the top.

dir adds *dir* to the directory stack at the top, making it the new current working directory.
If the **pushd** command is successful, a **dirs** is performed as well. If the first form is used, **pushd** returns 0 unless the **cd** to *dir* fails. With the second form, **pushd** returns 0 unless the directory stack is empty, a non-existant directory stack element is specified, or the directory change to the specified new current directory fails.

12.33 pwd

pwd

Print the absolute pathname of the current working directory. The path printed contains no symbolic links if the **-P** option to the **set** builtin command is set. See also the description of **nolinks** under **Shell Variables** above). The return status is 0 unless an error occurs while reading the pathname of the current directory.

12.34 read

read [**-r**] [*name* ...]

One line is read from the standard input, and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. Only the characters in **IFS** are recognized as word delimiters. If no *name*s are supplied, the line read is assigned to the variable **REPLY**. The return code is zero, unless end-of-file is encountered. If the **-r** option is given, a backslash-newline pair is not ignored, and the backslash is considered to be part of the line.

12.35 readonly

readonly [**-f**] [*name* ...]

readonly -p

The given *name*s are marked readonly and the values of these *name*s may not be changed by subsequent assignment. If the **-f** option is supplied, the functions corresponding to the *name*s are so marked. If no arguments are given, or if the **-p** option is supplied, a list of all readonly names is printed. An argument of **--** disables option checking for the rest of the arguments. The return status is 0 unless an illegal option is encountered, one of the *name*s is not a legal shell variable name, or **-f** is supplied with a *name* that is not a function.

12.36 return

return [*n*]

Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed in the function body. If used outside a function, but during execution of a script by the **.** (**source**) command, it causes the shell to stop executing that script and return either *n* or the exit status of the last command executed within the script as the exit status of the script. If used outside a function and not during execution of a script by **.**, the return status is false.

12.37 set

set [**--abefhkmnptuvxldCHP**] [**-o** *option*] [*arg* ...]

-a Automatically mark variables which are modified or created for export to the environment of subsequent commands.

-b Cause the status of terminated background jobs to be reported immediately, rather than before the next primary prompt. (Also see **notify** under **Shell Variables** above).

-e Exit immediately if a *simple-command* (see **SHELL GRAMMAR** above) exits with a non-zero status. The shell does not exit if the command that fails is part of an *until* or *while* loop, part of an *if* statement, part of a **&&** or **||** list, or if the command's return value is being inverted via **!**.

-f Disable pathname expansion.

-h Locate and remember function commands as functions are defined. Function commands are normally looked up when the function is executed.

-k All keyword arguments are placed in the environment for a command, not just those that precede the command name.

-m Monitor mode. Job control is enabled. This flag is on by default for interactive shells on systems that support it (see **JOB CONTROL** above). Background processes run in a separate process group and a line containing their exit status is printed upon their completion.

-n Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored for interactive shells.

-o *option-name*
The *option-name* can be one of the following:

allexport
Same as **-a**.

braceexpand
The shell performs brace expansion (see **Brace Expansion** above). This is on by default.

emacs
Use an emacs-style command line editing interface. This is enabled by default when the shell is interactive, unless the shell is started with the **-nolineediting** option.

errexit
Same as **-e**.

histexpand
Same as **-H**.

ignoreeof
The effect is as if the shell command ‘IGNOREEOF=10’ had been executed (see **Shell Variables** above).

interactive-comments
Allow a word beginning with **#** to cause that word and all remaining characters on that line to be ignored in an interactive shell (see **COMMENTS** above).

monitor
Same as **-m**.

noclobber
Same as **-C**.

noexec
Same as **-n**.

noglob
Same as **-f**.

nohash
Same as **-d**.

notify
Same as **-b**.

nounset
Same as **-u**.

physical
Same as **-P**.

posix
Change the behavior of **bash** where the default operation differs from the Posix 1003.2 standard to match the standard.

privileged
Same as **-p**.

verbose
Same as **-v**.

vi Use a vi-style command line editing interface.

xtrace
Same as **-x**.

If no *option-name is supplied, the values of the current options are printed.*

-p Turn on *privileged* mode. In this mode, the **$ENV** file is not processed, and shell functions are not inherited from the environment. This is enabled automatically on startup if the effective user (group) id is not equal to the real user (group) id. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

-t Exit after reading and executing one command.

-u Treat unset variables as an error when performing parameter expansion. If expansion is attempted on an unset variable, the shell prints an error message, and, if not interactive, exits with a non-zero status.

-v Print shell input lines as they are read.

-x After expanding each *simple-command*, **bash** displays the expanded value of **PS4**, followed by the command and its expanded arguments.

-l Save and restore the binding of *name* in a **for** *name* [in **word**] command (see **SHELL GRAMMAR** above).

-d Disable the hashing of commands that are looked up for execution. Normally, commands are remembered in a hash table, and once found, do not have to be looked up again.

-C The effect is as if the shell command ‘noclobber=’ had been executed (see **Shell Variables** above).

-H Enable **!** style history substitution. This flag is on by default when the shell is interactive.

-P If set, do not follow symbolic links when performing commands such as **cd** which change the current directory. The physical directory is used instead.

-- If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the *args*, even if some of them begin with a **-**.

- Signal the end of options, cause all remaining *arg*s to be assigned to the positional parameters. The **-x** and **-v** options are turned off. If there are no *arg*s, the positional parameters remain unchanged.

The flags are off by default unless otherwise noted. Using + rather than - causes these flags to be turned off. The flags can also be specified as options to an invocation of the shell. The current set of flags may be found in **$-**. After the option arguments are processed, the remaining *n* *arg*s are treated as values for the positional parameters and are assigned, in order, to **$1**, **$2**, **...** **$***n*. If no options or *arg*s are supplied, all shell variables are printed. The return status is always true unless an illegal option is encountered.

12.38 shift

shift [*n*]

The positional parameters from *n*+1 ... are renamed to **$1 ...**. Parameters represented by the numbers **$#** down to **$#**-*n*+1 are unset. If *n* is 0, no parameters are changed. If *n* is not given, it is assumed to be 1. *n* must be a non-negative number less than or equal to **$#**. If *n* is greater than **$#**, the positional parameters are not changed. The return status is greater than 0 if *n* is greater than **$#** or less than 0; otherwise 0.

12.39 suspend

suspend [**-f**]

Suspend the execution of this shell until it receives a **SIGCONT** signal. The **-f** option says not to complain if this is a login shell; just suspend anyway. The return status is 0 unless the shell is a login shell and **-f** is not supplied, or if job control is not enabled.

12.40 test

test *expr*

[*expr* **]**

Return a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression *expr*. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. Each operator and operand must be a separate argument. If *file* is of the form /dev/fd/*n*, then file descriptor *n* is checked.

-b *file*
True if *file* exists and is block special.

-c *file*
True if *file* exists and is character special.

-d *file*
True if *file* exists and is a directory.

-e *file*
True if *file* exists.

-f *file*
True if *file* exists and is a regular file.

-g *file*
True if *file* exists and is set-group-id.

-k *file*
True if *file* has its ‘‘sticky’’ bit set.

-L *file*
True if *file* exists and is a symbolic link.

-p *file*
True if *file* exists and is a named pipe.

-r *file*
True if *file* exists and is readable.

-s *file*
True if *file* exists and has a size greater than zero.

-S *file*
True if *file* exists and is a socket.

-t *fd*
True if *fd* is opened on a terminal.

-u *file*
True if *file* exists and its set-user-id bit is set.

-w *file*
True if *file* exists and is writable.

-x *file*
True if *file* exists and is executable.

-O *file*
True if *file* exists and is owned by the effective user id.

-G *file*
True if *file* exists and is owned by the effective group id.

file1 **-nt** *file2*
True if *file1* is newer (according to modification date) than *file2*.

file1 **-ot** *file2*
True if *file1* is older than *file2*.

file1 **-ef** *file2*
True if *file1* and *file2* have the same device and inode numbers.

-z *string*
True if the length of *string* is zero.

-n *string*

string
True if the length of *string* is non-zero.

string1 **=** *string2*
True if the strings are equal.

string1 **!=** *string2*
True if the strings are not equal.

! *expr*
True if *expr* is false.

expr1 **-a** *expr2*
True if both *expr1* AND *expr2* are true.

expr1 **-o** *expr2*
True if either *expr1* OR *expr2* is true.

arg1 **OP** *arg2*
OP is one of **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**. These arithmetic binary operators return true if *arg1* is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than *arg2*, respectively. *arg1* and *arg2* may be positive integers, negative integers, or the special expression **-l** *string*, which evaluates to the length of *string*.

12.41 times

times

Print the accumulated user and system times for the shell and for processes run from the shell. The return status is 0.

12.42 trap

trap [**-l**] [*arg*] [*sigspec*]

The command *arg* is to be read and executed when the shell receives signal(s) *sigspec*. If *arg* is absent or **--**, all specified signals are reset to their original values (the values they had upon entrance to the shell). If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. *sigspec* is either a signal name defined in <*signal.h*>, or a signal number. If *sigspec* is **EXIT** (0) the command *arg* is executed on exit from the shell. With no arguments, **trap** prints the list of commands associated with each signal number. The **-l** option causes the shell to print a list of signal names and their corresponding numbers. An argument of **--** disables option checking for the rest of the arguments. Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created. The return status is false if either the trap name or number is invalid; otherwise ttrraapp returns true.

12.43 type

type [**-all**] [**-type** | **-path**] *name* [*name* ...]

With no options, indicate how each *name* would be interpreted if used as a command name. If the **-type** flag is used, **type** prints a phrase which is one of *alias, keyword, function, builtin*, or *file* if *name* is an alias, shell reserved word, function, builtin, or disk file, respectively. If the name is not found, then nothing is printed, and an exit status of false is returned. If the **-path** flag is used, **type** either returns the name of the disk file that would be executed if *name* were specified as a command name, or nothing if **-type** would not return *file*. If a command is hashed, **-path** prints the hashed value, not necessarily the file that appears first in **PATH**. If the **-all** flag is used, **type** prints all of the places that contain an executable named *name*. This includes aliases and functions, if and only if the **-path** flag is not also used. The table of hashed commands is not consulted when using **-all**. **type** accepts **-a**, **-t**, and **-p** in place of **-all**, **-type**, and **-path**, respectively. An argument of **--** disables option checking for the rest of the arguments. **type** returns true if any of the arguments are found, false if none are found.

12.44 ulimit

ulimit [**-SHacdfmstpnuv** [*limit*]]

ulimit provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The value of *limit* can be a number in the unit specified for the resource, or the value **unlimited**. The **H** and **S** options specify that the hard or soft limit is set for the given resource. A hard limit cannot be increased once it is set; a soft limit may be increased up to the value of the hard limit. If neither **H** nor **S** is specified, the command applies to the soft limit. If *limit* is omitted, the current value of the soft limit of the resource is printed, unless the **H** option is given. When more than one resource is specified, the limit name and unit is printed before the value. Other options are interpreted as follows:

-a all current limits are reported

-c the maximum size of core files created

-d the maximum size of a process's data segment

-f the maximum size of files created by the shell

-m the maximum resident set size

-s the maximum stack size

-t the maximum amount of cpu time in seconds

-p the pipe size in 512-byte blocks (this may not be set)

-n the maximum number of open file descriptors (most systems do not allow this value to be set, only displayed)

-u the maximum number of processes available to a single user

-v The maximum amount of virtual memory available to the shell

An argument of **--** disables option checking for the rest of the arguments. If *limit* is given, it is the new value of the specified resource (the **-a** option is display only). If no option is given, then **-f** is assumed. Values are in 1024-byte increments, except for **-t**, which is in seconds, **-p**, which is in units of 512-byte blocks, and **-n** and **-u**, which are unscaled values. The return status is 0 unless an illegal option is encountered, a non-numeric argument other than **unlimited** is supplied as **limit**, or an error occurs while setting a new limit.

12.45 umask

umask [**-S**] [*mode*]

The user file-creation mask is set to *mode*. If *mode* begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by **chmod**(1). If *mode* is omitted, or if the **-S** option is supplied, the current value of the mask is printed. The **-S** option causes the mask to be printed in symbolic form; the default output is an octal number. An argument of **--** disables option checking for the rest of the arguments. The return status is 0 if the mode was successfully changed or if no *mode* argument was supplied, and false otherwise.

12.46 unalias

unalias [**-a**] [*name* ...]

Remove *name*s from the list of defined aliases. If **-a** is supplied, all alias definitions are removed. The return value is true unless a supplied *name* is not a defined alias.

12.47 unset

unset [**-fv**] [*name* ...]

For each *name*, remove the corresponding variable or, given the **-f** option, function. An argument of **--** disables option checking for the rest of the arguments. Note that **PATH**, **IFS**, **PPID**, **PS1**, **PS2**, **UID**, and **EUID** cannot be unset. If any of **RANDOM**, **SECONDS**, **LINENO**, or **HISTCMD** are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless a *name* does not exist or is non-unsettable.

12.48 wait

wait [*n*]

Wait for the specified process and return its termination status. *n* may be a process ID or a job specification; if a job spec is given, all processes in that job's pipeline are waited for. If *n* is not given, all currently active child processes are waited for, and the return status is zero. If *n* specifies a non-existant process or job, the return status is 127. Otherwise, the return status is the exit status of the last process or job waited for.

Tcl/Tk - Einführung

1. Grundlegendes

Tcl = Tool Command Language ("Tickl")

Tk = Toolkit

Tcl → Shell - Kommandointerpreter (bash, Bourne-Shell, Korn-Shell, C-Shell)

Tcl + Tk = wish (windowing shell)

Für UNIX entwickelt, portiert auf DOS, Windows, OS/2, NT, MacIntosh

2. Literatur und Bezugsquellen

- John K. Ousterhout: *Tcl and the TK Toolkit*. Addison-Wesley 1994
- Brent B. Welch: *Practical Programming in Tcl and Tk*. Prentice Hall 1995

Anm.: Viele der folgenden Tabellen und Beispielprogramme sind entstanden im Rahmen der Diplomarbeit von Jürgen Schindler, die mit dessen freundlicher Genehmigung direkt übernommen wurden.

World Wide Web:

- http://cuiwww.unige.ch/eao/www/TclTk.html
  offizielle Tcl/Tk Homepage im WWW
- http://www.smli.com/research/tcl
  Tcl/Tk Projektseite der Fa. Sun Microsystems
- http://www.sco.com/IXI/of_interest/tcl/Tcl.html
- http://web.cs.ualberta.ca/˜wade/Auto/Tcl.html

URL’s (*Universal Resource Location*):

- ftp://ftp.aud.alcatel.com/tcl
- ftp://syd.dit.csiro/au/pub/tk
- ftp://src.doc.ic.ac.uk/packages/tcl/
- ftp://ftp.ibp.fr/pub/tcl
- ftp://ftp.luth.se/pub/unix/tcl
- ftp://ftp.switch.ch/mirror/tcl

- ftp://sterling.com/programming/languages/tcl
- ftp://ftp.sunet.se/pub/lang/tcl
- mailto://ftpmail@ftp.sunet.se
- ftp://ftp.cs.columbia.edu/archives/tcl
- ftp://ftp.uni-paderborn.de/pub/unix/tcl
- ftp://sunsite.unc.edu/pub/languages/tcl
- ftp://ftp.funet.fi/pub/languages/tcl/

ftp:
anonymous ftp - *user name: anonymous / passwd: email*-Adresse
Directory nach Hostname

Newsgroups: *comp.lang.tcl*

- Grundlage: GNU Public License

3. Idee von Tcl

- Einfacher, aber erweiterbarer Kommando-Interpreter
- string-orientierte Sprache
- wenige grundlegende Konstrukte, einfache Syntax
- Erweiterung erfolgt durch "Hinzubinden" neuer Kommandos als *built-in*-Kommandos
- Ausführung von Kommandos nicht in neuen Prozessen, sondern als Funktionsaufrufe (die u.U. allerdings *fork*en können)

Basis-Shell (Basis-Kommando-Interpreter):

- **tclsh** oder **tcl** oder **tsh**
  (**/usr/bin/tclsh** oder **/usr/local/bin/tclsh**,
  entsprechend für *tcl*) - im Folgenden ist i.a. der erste Pfad eingestellt

Um Funktionen von *Tk* zur Gestaltung von Benutzeroberflächen unter Unix X11 erweiterte Shell:

- **wish** oder **wishx** oder **wsh** (Windowing-Shell),
  /usr/bin/wish(x), /usr/local/bin/wish(x)
  im Folgenden ist i.a. der erste Pfad eingestellt)

Tcl - Erste Schritte

1. Ausführung

- Interaktiv über Kommandozeile
  Lies Zeile - Bewerte Zeile (Gruppierung + Substitutionen) - Gib Ergebnis aus
- Interaktiv durch Auswertung von Kommandos aus einer Datei
  → Kommando *source* mit Datei als Argument
- Start eines tcl-Skripts von einer Unix-Shell (bash, bourne, ksh):
  `pey$: tcl -f command_file`
- als ausführbare Datei (Shell-Skript - *chmod +x*)
  Erste Zeile des ausführbaren Skripts ist:
  #! /usr/bin/tcl
  resp. (je nachdem wie der Pfad gesetzt ist)
  #! /usr/local/bin/tcl

  bei *windowing shell* - erste Zeile:
  `#!/usr/local/bin/wish(x)`

  → Pfad !!!
  → Version ??? (evt. *wish -f*)

→ **man**

- Syntax:

  Tcl-Skript = { cmd (newline | semicolon | eof) }
  cmd = cmd_name [one_arg] {further_arg}

- **Optional** (siehe oben **[**one_arg**]**) wird bei *tcl* angegeben als: **?**one_arg**?**

Jedes Kommando besteht also aus einer Folge von Worten (Zeichenfolge ohne Leerraum) - *tcl* zerlegt die Kommandozeile zuerst in eine Folge von Worten (→ Gruppierung), wobei bestimmte Zeichen eine Sonderbedeutung haben (von links nach rechts interpretiert).

Kommando: *built-in*-Kommando, Tcl-Prozedur, UNIX-Kommando

Erweitert um
- Syntax für **Gruppierung** (mehrere Worte als ein Argument)
- Syntax für **Substitution** (Variable, geschachtelte Kommandos)

Argumente:
grundsätzlich Strings, außer bei Substitutionen von Tcl nicht interpretiert
Evaluierung von Variablen oder Kommando's ist explizit zu erzwingen

- Start der (einer) *Tcl*-Shell zur interaktiven Nutzung:

```
pey$ tcl
tcl>  # ein Kommando
tcl>  # noch ein Kommando
tcl>  # ...
tcl>  # und noch ein's
tcl>  exit
pey$
```

- **Prompt** von *tcl*: **tcl>** oder **%** oder was auch immer
- **Kommentare:**
  erstes *non-white-space* Zeichen ist **#**: Rest der Zeile ist Kommentar
  nach einem Kommando, abgeschlossen mit einem Semikolon, folgt ein **#**: Rest der Zeile ist Kommentar

2. Übersicht

Zentrale tcl-builtin's	
append	Wert einer Variablen verlängern (Leeraum wird nicht angefügt
array	Gibt Zustand eines Arrays und durchsucht die Elemente
break	Beendigung einer Schleife
catch	Fehler abfangen
cd	Katalog wechseln
close	Offene I/O-Verbindung schließen
concat	Argumente konkatenieren mit Leeraum als Trenner. Listen zusammenfügen
continue	Mit nächster Schleifeniteration weiter
error	Fehlermeldung ausgeben und ggf. terminieren (−> *catch*)
eof	Auf EOF prüfen
eval	Argumente konkatenieren und als Kommando auswerten
exec	neuen Prozeß erzeugen (*fork*) und ein UNIX-Kommando damit ausführen
exit	Prozeß terminieren
expr	Mathematischen Ausdruck auswerten
file	Dateisystem durchsuchen
flush	Interne Puffer einer I/O-Verbindung leeren
for	Schleifen-Kommando (→ *for*-Statement in C)
foreach	Schleife über eine Liste von Werten
format	String formattieren (→ sprintf in C)
gets	Eine Zeile der Eingabeverbidnung lesen

Zentrale tcl-builtin's	
glob	Muster auf Dateinamen expandieren
global	Globale Variable deklarieren
history	Steuerung der Kommandozeilen-Historie
if	Bedingtes Kommando - mit *else* und *elseif*
incr	Inkrementieren einer Variablen um ganzzahligen Wert
info	Abfrage des Status des *tcl*-Interpreters
join	Konkatenieren von Listenelementen mit einem gegebenen Trenner
lappend	Elemente am Ende einer Liste hinzufügen
lindex	Listenelement mit gegebenem Index holen
linsert	Elemente in eine Liste einfügen
list	Erzeugen einer Liste aus gegebenen Argumenten
llength	Anzahl der Elemente einer Liste bestimmen
lrange	Bereich von Listenelementen holen
lreplace	Listenelemente ersetzen
lsearch	Listenelement zu gegebenem Muster suchen
lsort	Liste sortieren
open	Datei oder Prozeß-Pipeline für I/O öffnen
pid	Prozeß-ID bestimmen
proc	Definition einer *tcl*-Prozedur
puts	Ausgabe eines Strings an I/O-Verbindung
pwd	Aktuellen Katalog ausgeben

Zentrale tcl-builtin's	
read	Blöcke von Zeichen aus I/O-Verbindung lesen
regexp	Suchen nach Strings zu einem regulären Ausdruck
regsub	Ersetzung basierend auf regulärem Ausdruck
rename	Namen eines *tcl*-Kommandos ändern
return	Wert aus Prozedur zurückgeben und diese terminieren
scan	→ *sscanf* in C
seek	File Offset setzen
set	Wertzuweisung an Variable
source	Kommando's in einer Datei ausführen
split	String in Liste zerlegen
string	Operationen auf Strings
subst	Substitution ohne Kommando-Evaluierung
switch	Mehrfachfall-Unterscheidung
tell	Aktueller FileOffset bestimmen
time	Ausführungszeiten eines Kommandos bestimmen
trace	Variablenveränderungen überwachen
unknown	Behandlung unbekannter Kommandos
unset	Variablen auf "undefiniert" setzen
uplevel	Kommando in anderem Gültigkeitsbereich (*scope*) ausführen
upvar	Referenz auf Variable in anderem Gültigkeitsbereich
while	Schleifen-Kommando

3. Erste Beispiele

```
pey$ cat ex1.tcl
#! /usr/bin/tcl
set a "Hallo, Tcl-Fan's"
# Kommando set zur Zuweisung eines
# string-Wertes (Hallo, Tcl-Fan's)
# an die Variable a

puts $a;  #Ausgabe des Wertes der Variablen a

pey$ tcl -f ex1.tcl
Hallo, Tcl-Fan's

pey$ chmod +x ex1.tcl
pey$ ex1.tcl
Hallo, Tcl-Fan's
pey$
```

oder interaktiv:

```
pey$ tcl
tcl>   set a "Hallo, Tcl-Fan's"
tcl>
tcl>   puts $a
Hallo, Tcl-Fan's
tcl>
tcl>   puts a
a
tcl>   # also $ vor einer Variablen liefert
tcl>   # deren Wert
tcl>   exit
pey$
```

- oder interaktiv aus einer Datei

```
pey$ cat ex2.tcl
set a "Hallo, Tcl-Fan's"
# Kommando set zur Zuweisung eines string-Wertes
# (Hallo, Tcl-Fan's) an die Variable a, Gruppierung
# in Doppelapostrophen macht trotz
# Leerraum EIN Argument!

puts $a;  #Ausgabe des Wertes der Variablen a

pey$ tcl
tcl> source ex2.tcl
Hallo, Tcl-Fan's
tcl>  exit
pey$
```

- Ausführung von Kommando’s als neuer Prozeß: `fork - execute`

```
pey$ tcl
tcl>  exec ps
  PID TTY STAT  TIME COMMAND
   58 v01 S     0:00 -bash
   o o o
   77 v01 S     0:00 /usr/openwin/bin/olvwm
   82 v01 S     0:00 olwmslave
   84 pp0 S     0:00 -bash
   91 v01 S     0:00 /usr/bin/X11/xclock
  206 pp0 S     0:00 script
  207 pp0 S     0:00 script
  208 pp1 S     0:00 bash -i
  209 pp1 S     0:00 tcl
  210 pp1 R     0:00 ps

tcl>  exec ls
ex1.tcl
ex2.tcl
kap.1

tcl>  exec ls-l
couldn’t find "ls-l" to execute

tcl>  exec ls -l
total 35
-rwxr-xr-x   1 swg      sai           179 Sep 19 19:29 ex1.tcl
-rwxr-xr-x   1 swg      sai           160 Sep 20 08:32 ex2.tcl
-rw-r--r--   1 swg      sai          2382 Sep 20 08:37 kap.1

tcl>  exec ls -l kap.*
ls: kap.*: No such file or directory
tcl>  # dazu spaeter mehr
tcl>  exit
pey$
```

- **Rückgabewerte** von Tcl-Funktionen sind Strings, die im Fehlerfall eine Beschreibung des aufgetretenen Fehlers enthalten:

```
pey$ tcl
tcl>  set a 5
tcl>  puts $a
5
tcl>  set a
5
tcl>  set a 9 12
wrong # args: should be "set varName ?newValue?"
tcl>  set a
5
```

```
tcl>  exit
pey$
```

- Etwas Rechnen:

```
pey$ tcl
tcl>  expr 2+ 2 ; #Leerraum???
4
tcl>  expr 2 + 2-5
-1
tcl>  expr (2*4)-(1 +2) ; #Klammern
5
tcl>  expr 2 << 4 ; #Linksshift
32
tcl>  expr 32 >> 3 ; #Rechtshift
4
tcl>  expr (32 >> 3) * 3;
12
tcl>  expr 2 < 3
1
tcl>  expr 3 < 2
0
tcl>  #also: 0 ist FALSE und 1 ist TRUE
tcl>  expr (3 < 2) || 2 <= 10
1
tcl>  exit
pey$
```

Anm.: Die arithmetischen Ausdrücke sind in sehr enger Anlehnung an die *expressions* in ANSI C!

☞ Da *tcl* für sich allein zwar nützlich ist, im Zusammenhang mit *wish* "spannend" wird, werden weitere einführende Beispiele unter *wish* dargestellt.

4. Die Windowing-Shell *wish*

- Erweiterung von *tcl* durch Kommando's von *Tk*
- Erzeugung grafischer Oberflächen
- Nach dem Aufruf von *wish* erscheint am Bildschirm ein kleines, leeres Window (siehe unten), danach erwartet *wish* Eingabe von Kommandos von der Standardeingabe (wie *tcl*)

Beispiel:

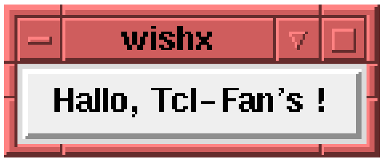
```
pey$ wishx
wishx>  cat hallo1.wsh
button .a -text "Hallo, Tcl-Fan's !" \
-background green -command exit
pack .a
wishx>  source hallo1.wsh
wishx>  exit
pey$
```

erzeugt wird das Window:

Oder interaktiv:

```
pey$ wishx
wishx>  button .a -text "Hallo, Tcl-Fan's!" \
-background green -command exit
.a
wishx>  pack .a
wishx>  exit
pey$
```

erzeugt wird das Window:

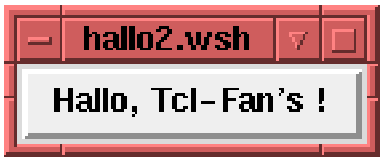


oder so:

```
pey$ cat hallo2.wsh
#!/usr/bin/wishx
button .a -text "Hallo, Tcl-Fan's !" \
-background green -command exit
pack .a

pey$ chmod +x hallo2.wsh
pey$ hallo2.wsh
pey$
```

Achtung: hier muß bei manchen Versionen die Option **-f** angegeben werden!

☞ erzeugt wird das Window:



☞ Klick mit der linken Maustaste (Maustaste 1) auf den Text "Hallo, Tcl-Fan's!" terminiert *wish* und das Window verschwindet

Erläuterungen zu diesem Beispiel:

- die Kommandozeile mit *button* (aus *TK*) enthält 8 Worte:

button	*.a*	*-text*	*Hallo, Tcl-Fan's !*	*-background*	*green*	*-command*	*exit*

- die mit *pack* enthält zwei Worte:

pack	*.a*

- Das Grundgerüst für grafische Oberflächen in Tk ist das *widget*, ein Window mit speziellem Erscheinungsbild und Verhalten. Dies können sein: *buttons*, *menus*, *scrollbars* (*widget*-Klassen)
- *widgets* sind hierarchisch angeordnet analog zum UNIX-Dateisystem. Die *root* heißt statt "Schrägstrich" (**/**) "Punkt" (**.**). Also heißt das erste Window darunter **.a**, dessen Kind z.B. **.a.b**
- das *button*-Kommando bekommt als erstes Argument den Namen des zu erzeugenden *buttons*, die weiteren Argumente treten paarweise auf und steuern die Gestaltung und das Verhalten (Konfiguration). In unserem Fall:

-text	*Hallo, Tcl-Fan's!*	Text, der im *button* erscheinen soll
-background	*green*	Hintergrundfarbe
-command	*exit*	das Kommando, das beim Anklicken ausgeführt werden soll

- Das *pack*-Kommando erzeugt das *widget* am Schirm (mehr dazu später → *geometry manager*))
- Das *destroy*-Kommando (`destroy .a`) entfernt das *widget*

Weitere Beispiele (Telefonverzeichnis):

```
pey$ cat ./phonebook/phones
Balser : 3558
Beschorner : 3552
Borchert : 3572
Boremski : 3566
Etter : 3601
Fiedler : 3507
Gessner : 3555
Grabert : 3573
Hellwig : 3540
Kratz : 3559
Maier : 3510
Mueller : 3571
Pittelkow : 3502
Rieder : 3525
Schweiggert : 3570
Stanglow : 3574
pey$
```

- *listboxes*

```
pey$ cat ./phonebook/list1.all
#!/usr/bin/wishx

set PHONEBOOK "/export/home/swg/tclSkr/ex1/phonebook/phones"

listbox .phonelist
pack .phonelist
set f [open $PHONEBOOK r] ; #Datei zum Lesen oeffnen

#Zeile fuer Zeile lesen:
while { [gets $f line] >= 0} {
        .phonelist insert end $line
}
close $f; #Datei schliessen

pey$ exit
```

Dieses Programm erzeugt folgendes Window:

Erläuterungen (vorab):

- das Kommando *open* öffnet die Datei "phones" zum Lesen ("r") und liefert eine Art Filedeskriptor zurück
- Das *while*-Kommando (!) hat zwei Argumente: Bedingung + Aktion(en)
- Das *gets*-Kommando liest aus dem "Filedeskriptor" (siehe *open*) eine Zeile in eine Variable (*line*)
- Das *close*-Kommando schließt den "Filedeskriptor".

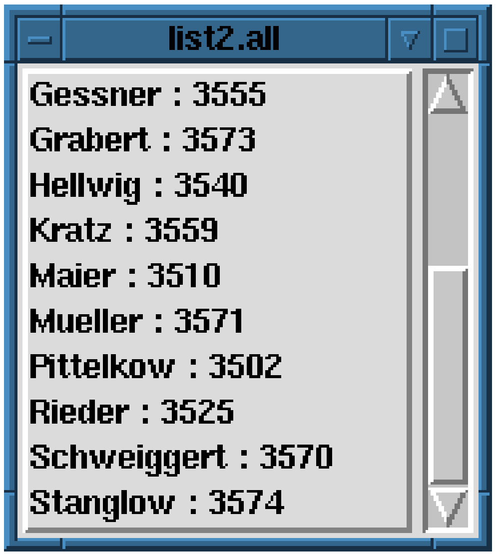
☐ *listbox & scrollbar*

```
pey$ cat ./phonebook/list2.all
#!/usr/bin/wishx

set PHONEBOOK "~/tclSkr/ex1/phonebook/phones"

listbox .phonelist -relief raised \
-borderwidth 2 -yscrollcommand ".scroll set"
pack .phonelist -side left
scrollbar .scroll -command ".phonelist yview"
pack .scroll -side right -fill y

set f [open $PHONEBOOK r]

while { [gets $f line] >= 0} {
        .phonelist insert end $line
}
close $f
```

```
pey$
```

erzeugt wird:

durch Verschieben der *scrollbar*:



□ Ereignisse an Windows binden (*event binding*)

```
pey$ cat ggt.wsh
#! /usr/bin/wishx

#eine Tcl-Prozedur
proc ggt { x y } {


  if {  $x <= 0 || $y <=0 } { return "zwei nat. Zahlen verlangt!" }
```

```
  while { $x != $y } {
        if { $x > $y } { set x [expr $x - $y] }
        if { $y > $x } { set y [expr $y - $x] }
  }
return $x
}

label .label1 -text "GGT von"
entry .x -width 6 -relief sunken -textvariable x
label .label2 -text "und"
entry .y -width 6 -relief sunken -textvariable y
label .label3 -text "ist"
label .result -textvariable result
pack .label1 .x .label2 .y .label3 .result -side left \
      -padx 2m -pady 4m
bind .x <Return> {set result [ggt $x $y]}
bind .y <Return> {set result [ggt $x $y]}

pey$
```

Der Aufruf erzeugt zunächst:

Anklicken der beiden Eingabefelder und Eingeben von Zahlen, nach zweiter Zahl *return*:

Gibt man nach Anklicken des zweiten Feldes sofort *Return*:

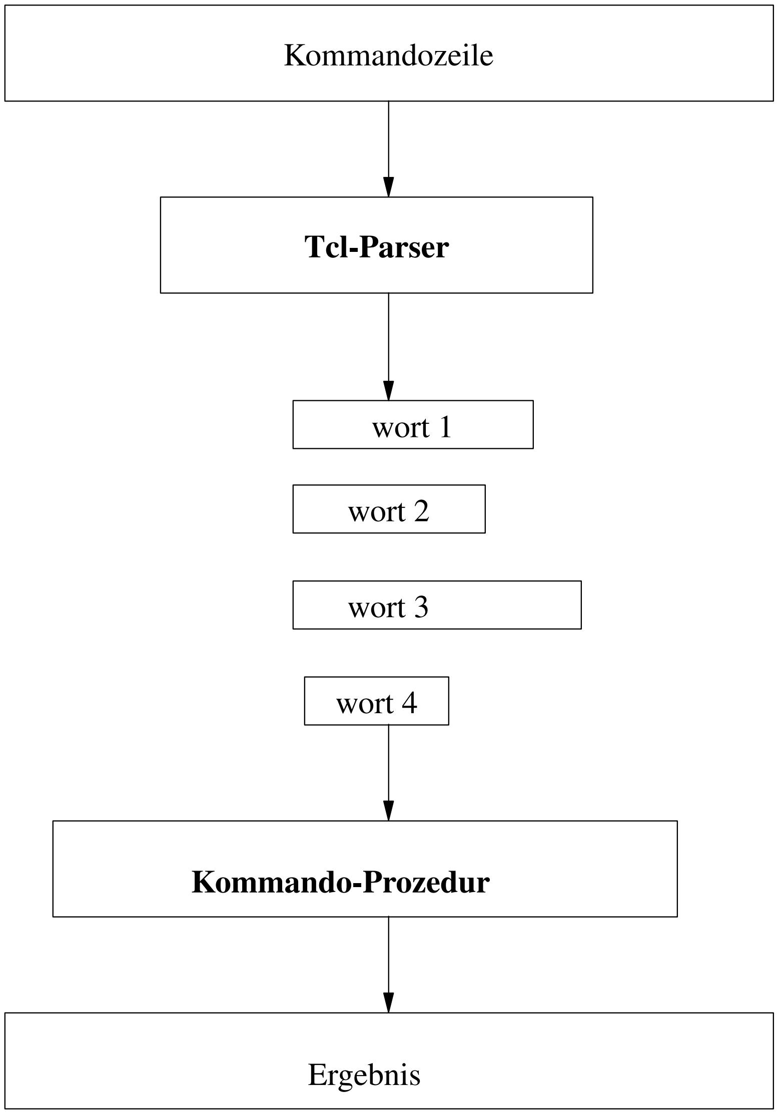
Dasselbe passiert, wenn man beim ersten Feld *return* eingibt!

Anmerkungen:

- Diese Anwendung besteht aus 6 *widgets*:
  - zwei Eingabefelder (Kommando *entry*) mit Bezeichnung x, y (*relief sunken*)
  - vier Textfelder (Kommando *label*), drei davon mit festem Text, das letzte (*result*) mit Text als Wert einer Variablen (*result*)
- Das Kommando *pack* ordnet die *widgets* von links nach rechts an
- Das Kommando *bind* bindet die Benutzerschnittstelle an die Prozedur *ggt*. Es hat drei Argumente:
  - Das erste Argument bezeichnet ein *widget*
  - Das zweite spezifiziert ein Ereignis (Drücken der Maustaste - Einfachklick, Doppelklick, Eingabe von *return*, ...)
  - Das dritte definiert, welches Tcl-Skript ausgeführt werden soll, wenn das Ereignis im angegebenen *widget* eintritt

tcl im Detail

1. Arbeitsweise



Jedes Zeichen wird vom *tcl*-Interpreter nur einmal von links nach rechts gelesen, Ergebnis wird nicht erneut nach Sonderzeichen untersucht. Dabei wird die Zerlegung in Worte vorgenommen. Das erste Wort wird dann als Name des Kommandos interpretiert, die weiteren werden diesem als Argumente durchgereicht! Dieses Kommando interpretiert die Argumente ggf. erneut.

- Substitution:
  Variablen (**$**, Kommandos (**[...]** - geschachtelt möglich), *backslashes*
- Gruppierung mehrerer Worte zu einem Argument:
  Doppelapostroph-Klammerung (**" ... "**) und Klammerung in geschweifte Klammern (**{ ... }**)

2. Arbeiten mit Variablen

- Namen von Variablen: beliebige Zeichenfolge;
  praktisch: der C-Konvention folgen

Wert von Variablen: Strings

- Einfache Variablen
- Vektoren
- assoziative Array's

Grundlegende Operationen auf (einfachen) Variablen:

- *set varName ?value?*
  Wertzuweisung / Wertdarstellung
- *unset varName ?varName ... varName?*
  Auf "undefiniert" setzen
- *append varName value ?value?*
  bisherigen Wert "verlängern"
- *incr varName ?increment?*
  Wert als Zahl interpretiert um *increment* verändern

Beispiele:

```
pey$ tcl
tcl>  set "xyz #!" {ein text}
tcl>  set "xyz #!"
ein text
tcl>  append "xyz #!" {mit fuenf worten}
tcl>  set "xyz #!"
ein textmit fuenf worten
tcl>  set a 5
tcl>  set a
5
tcl>  incr a 2
tcl>  set a
7
tcl>  unset a "xyz #!"
tcl>  set a
can’t read "a": no such variable
tcl>  exit
pey$
```

3. Substitution

3.1 Variablen und Ersetzungen ($)

Das $-Zeichen vor einem Namen führt zur Wertersetzung dieser Variablen.

```
pey$ var="file1 file2"
pey$ ls $var  #ls bekommt ZWEI Args
ls: file1: No such file or directory
ls: file2: No such file or directory

pey$ tcl
tcl>  set var "file1 file2"
file1 file2
tcl>  ls $var
ls: file1 file2: No such file or directory
child process exited abnormally
tcl>  # ls hat EIN Argument bekommen, naemlich
tcl>  # den Wert einer Variablen
tcl>  exit
pey$
```

- *tcl* liest die Zeile "ls $var", erkennt zwei Worte, nämlich "ls" und das, was sich durch Substitution von "$var" ergibt, reicht also an "ls" ein (!) Argument weiter.

☞ Variablensubstitution wird an beliebigen Stellen vorgenommen, auch innerhalb von Worten, Doppelapostroph-Gruppierung und Kommando-Substitution:

```
pey$ tcl
tcl> set var 1
tcl> set var
1
tcl> set var2 xxx$var1 ; #so nicht!
Error: can't read "var1": no such variable
tcl> set var2 xxx$var ; #so schon
tcl> set var2
xxx1
tcl> set var2 xxx${var}xxx ; #so auch
tcl> set var2
xxx1xxx
tcl> set text Hallo
tcl> set text
Hallo
tcl> puts "Die Laenge von $text ist [string length $text]"
Die Laenge von Hallo ist 5
tcl> exit
pey$
```

3.2 Kommando-Substitution ([...])

- Teil eines Wortes (oder das ganze Wort) wird ersetzt durch das Ergebnis eines Tcl-Scipts; besteht dieses Ergebnis aus mehreren Worten, wird es anschließend als ein Wort betrachtet (Leerraum nicht mehr interpretiert!):

```
pey$ tcl
tcl>  set a 4
tcl>  set b [expr $a * 2 ]
tcl>  set b [
        expr $a * 3 ]
tcl) set b
12

tcl>  set b [set x 8; expr $a * $x]
tcl>  set b
32

tcl>  set b [
        set x 10
        expr $x*$a
]
tcl>  set b
40

tcl> set len [expr [string length Hallo] + 20]
tcl> set len
25

tcl> set pi [expr 2*asin(1.0)]
tcl> set pi
3.14159

tcl> set tcl_precision 15
tcl> set pi [expr 2*asin(1.0)]
tcl> set pi
3.14159265358979
tcl> exit
pey$
```

oder

```
pey$ tcl
tcl>  set var4 1
tcl>  set var[expr 2*2] 100
tcl>  set var4
100
tcl>  exit
pey$
```

3.3 *backslash*-Substitution

- Ähnlich wie bei UNIX-Shells: schützt gewisse Zeichen (z.B. [,]) vor Interpretation, also **\[** wird ersetzt durch **[**.
- Ähnlich wie bei C: gibt gewissen Zeichen eine Sonderbedeutung (z.B.: *newline* → \n)
- Ein Backslash vor dem *return* wird samt den evt. in der folgenden Zeile führenden Lerrraumzeichen durch ein Blank ersetzt.
- Nur "eine Runde" der Ersetzung wird vorgenommen.

```
pey$ tcl
tcl>  set x zwei Worte
wrong # args: should be "set varName ?newValue?"
tcl>  set x zwei\ Worte
tcl>  set x
zwei Worte
tcl>  set x zwei\ Worte\ und\nzwei\ Zeilen
tcl>  set x
zwei Worte und
zwei Zeilen
tcl>  set x\
 zwei\ Worte
tcl>  set x
zwei Worte
tcl>  set preis 1US\$\ kostet\ DM\ 1.47
set preis
1US$ kostet DM 1.47
tcl>
tcl> set dollar \$
tcl> set dollar
$
tcl> set x $dollar
tcl> set x
$
tcl> set one Hallo
tcl> set two Tcl\ Fan's
tcl> set one
Hallo
tcl> set two
Tcl Fan's
tcl> set gesamt [expr [string length $one] + \
=>        [string length $two]]
tcl> set gesamt
14
tcl> exit
pey$
```

Backslash-Ersetzung	
Backslash Sequenz	wird ersetzt durch
\a	Biep (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
\xhh	Hexadezimalwert (beliebig viele Stellen)
newline	einzelnes Leerzeichen
newline space	einzelnes Leerzeichen
\c	ersetzt durch c, falls c keines der Zeichen von oben

4. Gruppierung

4.1 Mit Doppelapostrophen

Ähnlich wie bei UNIX-Shells:

- die Wort- und Kommandotrenner *white space*, *newline* und Semikolon verlieren Sonderbedeutung und werden zu ganz normalen Zeichen
- Variablensubstitution (**$**), Kommando-Substitution (**[ ]**) wie auch *backslash*-Substitution finden statt

```
pey$ tcl
tcl>  set preis 2.15
tcl>   set text "Es kostet \$$preis;
okay? \"NEIN\""
tcl>   set text
Es kostet $2.15;
okay? "NEIN"

tcl>   set a 6
tcl>   set quadrat "$a zum Quadrat ist [ expr $a*$a]\n"
tcl>   set quadrat
6 zum Quadrat ist 36
tcl>   exit
pey$
```

Anwendung: **format-Kommando**

```
pey$ tcl
tcl> set titel "Programmieren in Tcl/Tk"
tcl> set preis 35.90
tcl> puts [format "Buch: %s\t%5.2f" $titel $preis]
Buch: Programmieren in Tcl/Tk   35.90
tcl> exit
pey$
```

4.2 Mit geschweiften Klammern

- Sämtliche Zeichen verlieren ihre Sonderbedeutung

```
pey$ tcl
tcl>  set a 5
tcl>  set quadrat {$a zum Quadrat ist [expr $a*$a]
 oder ?
 }
tcl>  set quadrat
 $a zum Quadrat ist [expr $a*$a]
oder ?

tcl>  set laenge 5.9
tcl>  puts ${laenge}km ; # $laengekm ???
5.9km
tcl>  set {Franz Schweiggert} {Leiter von SAI}
tcl>   set {Franz Schweiggert}
Leiter von SAI
tcl>  set a {die Menge \{1,4,6\} }
tcl>  set a
die Menge \{1,4,6\}
tcl>  set a {die Menge {1,4,6} }
tcl>  set a
die Menge {1,4,6}

tcl>  set b {was passiert mit \n? }
tcl>  set b
was passiert mit \n?
tcl>  set b {was passiert mit\
?}
tcl>  set b
was passiert mit ?

tcl>  # der Unterschied von " ... " und { ... }
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].
```

```
tcl> exit
pey$
```

- Worttrenner, Sonderzeichen wie $ oder \n verlieren Sonderbedeutung!
  → einfache Hochkomma's in der Bourne-Shell
  NB: einfache Hochkomma's wie Rückwärts-Hochkomma's haben in der Tcl-Shell keine Bedeutung!
- Anwendung:
  Übergabe ganzer Tcl-Skripte als Argument an Kommando - dieses wertet Sonderzeichen aus, oftmals durch Rückführung des Arguments an Tcl-Interpreter
  -> verzögerte Auswertung

Beispiel:

```
pey$ tcl
tcl>  cat do_exit
proc do_exit {name} {
        puts "Und tschuess, lieber $name!"
        exit 0
}
tcl>  source do_exit
tcl>  do_exit swg
Und tschuess, lieber swg!
pey$
```

Beispiel:

```
pey$ tcl
tcl>  cat occur_count
proc occur {value list} {
        set count 0
        foreach el $list {
                if $el==$value {
                        incr count
                }
        }
        return $count
}
tcl>  source occur_count
tcl>  occur 1 { 1 11 2 3 1 2 1}
3
tcl> exit
pey$
```

Beispiel:

```
pey$ tcl
tcl> cat fak.tcl
set i 1; set prod 1
while { $i <= 10 } {
        set prod [ expr $prod * $i ]
        incr i
}

tcl> source fak.tcl
tcl> set prod
3628800
tcl> exit
pey$
```

Was passiert, wenn man im letzten Beispiel bei
{ $i <= 10}
die geschweiften Klammern wegläßt oder statt dessen Doppel-Apostrophen setzt? **$i** wird sofort zu **1** bewertet (also vor dem Aufruf des Kommando's *while* - *while* bekommt als Bedingungsargument die Konstante 1 (True), die Bedingung ist also immer wahr -> Endlos-Schleife!!!
wenn möglich, bewertet *tcl* - aufgerufene Kommando's bewerten ggf. erneut!

4.3 Zusammenfassung

- Das Dollarzeichen erzwingt Variablen-Substitution. Variablennamen können beliebig lang sein, Groß-/Kleinschreibung ist signifikant. Referenzen auf Variable können in beliebige Strings eingebettet sein, Erkennung durch *${varname}*
- Eckige Klammern erzwingen Kommando-Substitution. Alles zwischen den eckigen Klammern wird ersetzt durch das Ergebnis des Kommando's. Eckige Klammern können geschachtelt werden, Bearbeitung von links nach rechts.
- Mit dem Backslash wird Einzelzeichen-Substitution erzwungen.
- Substitutionen können an beliebigen Stellen vorgenommen werden, außer sie sind durch geschweifte Klammern verhindert.
- Gruppierung durch geschweifte Klammern verhindern jegliche Interpretation, auch die Zerlegung in Worte (Interpretation von Leerraum). Geschweifte Klammern können geschachtelt werden.
- Gruppierung mit Doppelapostrophen erlaubt Substitution.
- **Gruppierung vor Substitution**. Durchgeführte Substitutionen beeinflussen die Gruppierung nicht!
- Nur "eine Runde" der Substitution wird durchgeführt, bevor ein Kommando aufgerufen wird; d.h. das Ergebnis der Substitution wird nicht erneut bewertet. Das aufgerufene Kommando allerdings kann eine erneute Substitution durchführen (rekursiver *tcl*-Aufruf).

- Nach einer schließenden gescweiften Klammer und einer öffnenden geschweiften Klammer sollte i.a. ein Worttrenner stehen (Lerrraum).
- Ein Doppelapostroph ist nur signifikant nach Leerraum

5. Ausdrücke (numerisch)

- *expr*: Kommando zur Bewertung von Ausdrücken mit Operanden und Operatoren

  Syntax: *expr argument { argument }*
  Ausführung: alle Argumente werden konkateniert und als ein Argument an *expr* durchgereicht

5.1 Operanden

- *integer*:

  Dezimalzahlen (erste Ziffer keine 0): 123
  0129 (nicht zulässig, Oktalzahl)

  Oktalzahlen (Oktalziffern, erste Ziffer eine Null): 0123
  0129 (nicht zulässig, da 9 keine Oktalziffer)

  Hexadezimalzahlen (Hexadezimalziffern, die ersten beiden Zeichen sind 0x): 0x123, 0xfff
- *reals*:
  → ANSI-C
  Beispiele: 3.14, 0.89e+3, 1.0e-10, 4., 3E6

  Genauigkeit: *builtin*-Variable *tcl_precision*

5.2 Operatoren

- Wahrheitswerte: Zahl 0 entspricht FALSE, Zahl <> 0 (**1**) entspricht TRUE (Bei der Bourne-Shell: Zahl 0 entspricht TRUE, Zahl <> 0 entspricht FALSE)

- In der folgenden Tabelle stehen die Operatoren mit größerem Vorrang weiter oben, die horizontalen Linien trennen die einzelnen Vorrangstufen. Durch Klammerung kann die Reihenfolge der Bewertung festgelegt werden. Bei gleichem Vorrang der Operatoren wird der Ausdruck von links nach rechts ausgewertet. (Es gelten die gleichen Vorrangregeln wie in ANSI C).

Operanden		
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 vona	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
aˆb	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
int(x)	Ganzzahliger Anteil von
log(x)	natürl. Logarithmus von x
log10(x)	Logarithmus zur Basis 10 von x
pow(x,y)	x^y
round(x)	Ganzz. 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

5.4 Ersetzungen

Beispiel:

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

- Im zweiten Fall führt die *Tcl*-Shell die Substitution aus, bevor das Kommando ausgeführt wird (defacto wird also 2*pow(2,3) ausgeführt
- Im dritten Fall wird dies wegen der geschweiften Klammern verhindert; die Substitution wird bei der Ausführung von *expr* durchgeführt

Beispiel:

```
#!/usr/local/bin/tcl
# kleinste Zweierpotenz groesser oder gleich
# als die als Argument gegebene positive Zahl
# argv Argumentvektor, von 0 an indiziert;
# Index 0: erstes Argument (nicht Kommandoname)
# Kommandoname steht in Variablen argv0

set num [lindex $argv 0]
puts $num

set pot 1
while {$pot < $num } {
        set pot [expr $pot * 2]
}
puts $pot
```

ABER:

```
#!/usr/local/bin/tcl
# kleinste Zweierpotenz groesser oder gleich
# als die als Argument gegebene positive Zahl


set num [lindex $argv 0]
puts $num

set pot 1
while { $pot<$num } {
        set pot [expr $pot * 2]
}
puts $pot
```

☞ Endlosschleife (WARUM???)

6. ARGV, File-I/O, Environment (vorab)

6.1 ARGV: Kommandozeilenargumente

- *argc*: Anzahl Argumente (incl. Programmname)
- *argv0*: Kommandoname (Name des *tcl script*)
- *argv*: Liste der Kommandozeilen-Argumente

Beispiel:

```
pey$ cat args.tcl
#!/usr/bin/tclsh
# Argumentverarbeitung in Tcl (hier drei Argumente):

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] "
#       lindex <listname> <index>

pey$ args.tcl -a b cdef
command name: ./args.tcl
There are 3 arguments: -a b cdef
The 1st argument is: -a
The 2nd argument is: b
The 3rd argument is: cdef
pey$
```

6.2 Environment: Umgebungsvariable

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

Beispiel:

```
pey$ tcl
tcl>  >>>puts "Home directory: $env(HOME)"
Home directory: /home/endlich/swg
tcl>  >>>puts "Operating system: $env(OSTYPE)"
Operating System: Linux
tcl>  >>>puts "sec. prompt: $env(PS2)"
sec. prompt: >
tcl>  exit
pey$
```

6.3 File-I/O

Kommando's *open - gets - puts*
"Filedeskriptoren" → open
Standard-"Filedeskriptoren": stdin, stdout, stderr

- den ersten Datensatz aus einer Datei lesen:

```
pey$ cat ../ex1/phonebook/phones
Ralf : 5857
Balser : 3558
Beschorner : 3552
Borchert : 3572
Boremski : 3566
Etter : 3601
Fiedler : 3507
Gessner : 3555
Grabert : 3573
Hellwig : 3540
Kratz : 3559
Maier : 3510
Mueller : 3571
Pittelkow : 3502
Rieder : 3525
Schweiggert : 3570
Stanglow : 3574

pey$tcl
tcl> cat getfirst

set PHONEBOOK "~/tclSkr/ex1/phonebook/phones"

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

gets $f line

puts stdout $line

tcl> source getfirst
Ralf : 5857
tcl> puts stderr $line
Ralf : 5857
tcl> exit
pey$
```

- Und immer den nächsten:

```
pey$tcl
tcl> cat getnext

set PHONEBOOK "~/tclSkr/ex1/phonebook/phones"

set f [open $PHONEBOOK r]

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

tcl> source getnext

tcl> getnext $f
Ralf : 5857
tcl> getnext $f
Balser : 3558
tcl> getnext $f
Beschorner : 3552
tcl> exit
pey$
```

- Und alle:

```
pey$tcl
tcl> cat getall

set PHONEBOOK "~/tclSkr/ex1/phonebook/phones"

set f [open $PHONEBOOK r]

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

tcl> source getall
Ralf : 5857
Balser : 3558
Beschorner : 3552
Borchert : 3572
Boremski : 3566
Etter : 3601
Fiedler : 3507
Gessner : 3555
Grabert : 3573
Hellwig : 3540
Kratz : 3559
Maier : 3510
```

```
Mueller : 3571
Pittelkow : 3502
Rieder : 3525
Schweiggert : 3570
Stanglow : 3574
tcl> exit
pey$
```

- Und alles in eine Datei "XXX":

```
pey$ls
copyall
pey$tcl

tcl> cat copyall

set PHONEBOOK "~/tclSkr/ex1/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

tcl> source copyall

tcl> cat XXX
Ralf : 5857
Balser : 3558
Beschorner : 3552
Borchert : 3572
Boremski : 3566
Etter : 3601
Fiedler : 3507
Gessner : 3555
Grabert : 3573
Hellwig : 3540
Kratz : 3559
Maier : 3510
Mueller : 3571
Pittelkow : 3502
Rieder : 3525
Schweiggert : 3570
Stanglow : 3574
tcl> exit
pey$
```

7. Strings

7.1 Das *string*-Kommando

Das *string*-Kommando	
string compare str1 str2	lexikographisch von *str1* mit *str2*: kleiner (-1), gleich (0) oder größer (1)
string first str1 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 str1 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	Liefert den Index des Zeichens in *str* nach dem Wort, das das Zeichen bei Index *ix* enthält.
strin 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 auf Identität zweier Strings kann prinzipiell auch mit dem **==** Operator erfolgen.

ABER:

```
pey$ tcl
tcl> set x Hallo
tcl> if $x==Hallo { puts gleich!}
Error: syntax error in expression "Hallo==Hallo"
tcl> if $x=="Hallo" { puts gleich! }
Error: syntax error in expression "Hallo=="Hallo""
tcl> if {$x==Hallo} { puts gleich! }
Error: syntax error in expression "$x==Hallo"
tcl> if {$x=="Hallo"} { puts gleich! }
gleich!
tcl> if { "0xc" == "12" } { puts newline }
newline
tcl> if { [string compare 0xc 12] == 0 } {
=> puts equal
=> } else {
=> puts not_equal
=> }
not_equal

tcl> exit
pey$
```

- Bei der Auswertung des **==** Operators durch das *if*-Kommando (genauer *expr*) muß der String als String (in quotierter Form) ankommen, daher in geschweiften Klammern!
- *expr* versucht zunächst die Operanden in Zahlen zu wandeln (wenn möglich); falls dies nicht geht, werden sie in strings zurückgewandelt (Aufwand)
- also: *string compare*

7.3 *string match* - Kommando

→ Dateinamen-Matching bei Unix-Shell's (*glob*-style)

- *****: beliebige Zeichenfolge
- **?**: beliebige Einzelzeichen
- Aufzählung von Zeichen **[ ]**

```
pey$ tcl
tcl> string match a* alpha
1
tcl> string match a? alpha
0
tcl> string match ?? AA
1
tcl> string match a*b alpha
0
tcl> string match {[abc]*} alpha
1
tcl> set pat {[abc]*X}
tcl> set pat
[abc]*X
tcl> string match $pat alTreX
1
tcl> string match {*\*} abc*
1
tcl> string match {\\?} {\a}
1
tcl> exit
pey$
```

7.4 *append*-Kommando

Syntax:

```
append var_name arg1 arg2 ...
```

- Erstes Argument ist Variable
- Die weiteren Argumente werden konkateniert und an den Wert der Variablen angefügt

7.5 *format*-Kommando

→ *printf*

Syntax:

```
format spec_arg value1 value2 ...
```

- *spec_arg* entspricht dem 1.Argument von *printf*
- *spec_arg* enthält Literale (im Ergebnis enthalten wie dargestellt) und Schlüsselworte, die angeben, wie der zugehörige Wert im Ergebnis konvertiert werden soll
- Schlüsselwort: beginnt mit **%**, gefolgt von Modifikationszeichen und schließt ab mit einem Konvertierungszeichen (%% gibt %)
  im allgemeinsten Fall:
  %(pos_spec)(flags)(fld_width)(prec)(w_length)(conv_char)
- *spec_arg* in Doppelapostrophen geklammert, da Leerraum enthalten sein kann, *backslash* Substitution durchgeführt werden soll (\t, \n) und spezielle Zeichen substituiert werden sollen ($)

Format Flags	
−	Ausrichtung des Feldes links
+	Vorzeichen angeben, auch +
space	Falls eine Zahl kein Vorzeichen hat, stelle Leerzeichen voran
0	Mit Nullen auffüllen
#	Einer Octalzahl 0, einer Hexadezimalzahl 0x voranstellen. Dezimalpunkt grundsätzlich angeben. Hintere Nachkomma-Nullen nicht entfernen (%g)

- *position specifier*
  Eine Angabe **i$** (i eine Zahl) bedeutet, daß als korrespondierendes Argument (*value*) das i-te genommen werden soll (gezählt wird ab 1)

Beispiele:

```
pey$ tcl
tcl> set language 2
tcl> format "%${language}\$s" one eins un uno
eins
tcl> format "%x" 20
14
tcl> format "%#x" 20
0x14
tcl> format "%#06x" 20
0x0014
tcl> format "%# 6x" 20
  0x14
tcl> set maxLength 8
tcl> format "%*s|%s" $maxLength 10 10
```

```
      10|10
tcl> format "%-*s|%s" $maxLength 10 10
10      |10
tcl> exit
pey$
```

7.6 *scan*-Kommando

→ *sscanf* in C

Syntax: scan str format_spec var1 var2 ... kap.LP

- *format_spec* (fast) wie bei *format*-Kommando
  allerdings kein "%u" (*unsigned integer*), bei "%c" keine Feldweite
- zusätzlich eine Mengen-Notation möglich

```
pey$ tcl
tcl> scan "123 456 abc" "%d %d %s" x y s
3
tcl> set x
123
tcl> set y
456
tcl> set s
abc
tcl> scan abcd {%[ab]} res ; #{} ist hier wichtig
1
tcl> set res
ab
tcl> scan abcdEFGhij {%[a-z]} res ; #[] haben Bedeutung
1
tcl> set res
abcd
tcl> scan abcdEFGhij {%[^a-z]} res
0
tcl> set res
abcd
tcl> scan abcdEFGhij {%[A-Z]} res
0
tcl> set res
abcd
tcl> exit
pey$
```

7.7 Reguläre Ausdrücke

Ausgangspunkt: Gegeben ist eine Menge **T** von Symbolen (Menge der Terminal-Symbole). Eine Sprache **L** ist eine Teilmenge der Menge aller möglichen Folgen von Symbolen aus T. Eine solche Menge läßt sich ggf. endlich beschreiben (→ formale Sprachen, Grammatiken, EBNF).

Reguläre Sprachen lassen sich durch reguläre Ausdrücke beschreiben; dazu werden zu den Terminalsymbolen sog. Meta-Symbole eingeführt:

- Die leere Menge ist ein regulärer Ausdruck
- Das leere Wort ist ein regulärer Ausdruck
- Jedes Terminalsymbol ist ein regulärer Ausdruck
- Sind A und B reguläre Ausdrücke, so auch AB (Konkatenation),
  (A | B) (Auswahl) sowie
  { A } (Folge)
  Im letzten Fall verwendet man, wie wir das im Folgenden auch tun werden, die Schreibweise (A)* mit dem Meta-Symbol * zusätzlich zu den Meta-Symbolen (); damit können die geschweiften Klammern fortan wieder für Gruppierung oder Mengen verwendet werden.

Der Zusammenhang zu **regulären Sprachen** ergibt sich wie folgt:

Seien A, B, C reguläre Ausdrücke und bezeichne **L(A)** die von A definierte Sprache (Menge von korrekten Symbolfolgen); dann gilt:

- Ist A = { } (leere Menge), so ist L(A) = { }
- Ist A das leeres Wort so ist L(A) = {<leeres Wort>}
- Ist A = a (a Terminalsymbol), so ist L(A) = {a}
- Ist A = BC so ist L(A) = L(B) L(C)
  (das kartesische Produkt mit dem Konkatenationsoperator von L(B) und L(C))
- Ist A = (B | C), so ist L(A) = L(B) ∪ L(C)
- Ist A = (B)* , so ist L(A) = L(B)*
  Der *****-Operator drückt hier die "Hülle" bezgl. der Konkatenation aus!

Es gilt:

Satz: Jede durch endliche Automaten erkennbare Sprache ist regulär (Typ 3) und zu jeder regulären Sprache gibt es einen endlichen Automaten, der diese Sprache erkennt.

Reguläre Ausdrücke werden wir im Folgenden u.a. dazu verwenden, aus irgendwelchen Texten (oder Datenbeständen) gewisse Zeichenfolgen herauszufiltern. Dazu werden die gewünschten Zeichenfolgen (z.B.: alle Worte, die mit einem A beginnen, dem ein E oder ein e folgt) mittels eines regulären Ausdrucks beschrieben ("A(E|e)"), wobei wir die Worttrennung (also Zeichenfolgen ohne Leerraum) hier einem "Tool" überlassen. Dieses "Tool" hat dann das *Wortproblem* zu lösen: für jedes Wort im Text muß festgestellt werden, ob es zu der durch diesen regulären Ausdruck definierten Sprache gehört oder nicht.

Abschließend wird die obige Notation für reguläre Ausdrücke so erweitert, wie sie die (meisten) UNIX-Tools verwenden. Wir stehen dabei insbesondere vor dem Problem, daß wir Zeichen sowohl als Meta-Zeichen wie auch als Terminal-Zeichen unterscheiden müssen. Ebenso wollen wir bei den Terminalsymbolen gewisse nicht-darstellbare Zeichen ("Escape-Sequenzen") haben. Schließlich ist die "ge-Oder-te" - Aufzählung von Terminalzeichen, die in der ASCII-Tabelle hintereinanderstehen, sehr mühsam - einfacher ist es, zu sagen, daß ein Zeichen aus einem Bereich gemeint ist.

Die folgende Tabelle gibt eine Übersicht:

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 Aufzählung 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	r1 oder r2
(r1)(r2)	r1 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

Metazeichen sind also: **() [] \ . ^ + * ?**

Ein Fragezeichen als Terminalsymbol muß demnach entweder als **\?** oder als **[?]** (Aufzählung) definiert werden. Ein *Backslash* (\) muß demnach als **** definiert werden. Die Zeichen **^ $** definieren abhängig von der Zielsetzung oder Eigenschaften des jeweiligen Tools Anfang / Ende einer Zeile oder eines Wortes oder ...

Statt zu sagen, daß eine Zeichenfolge zu der von einem regulären Ausdruck definierten Sprache gehört, sagt man oft auch *die Zeichenfolge paßt zum Ausdruck ;* im Englischen wird dies auch als *the regular expression matches the string* augedrückt.

Regel: *first, then longest*

7.8 *regexp*-Kommando

Syntax:

```
regexp ?flags? pattern string ?match sub1 sub2 ...?
```

- *pattern*:
  Achtung: Einige Meta-Symbole haben für *tcl* Bedeutung - also z.B. in **{ }** setzen (nichts dazwischen wird von *tcl* erkannt) oder in **" "** setzen und *tcl*-Sonderzeichen mit *backslash* schützen.
- *flags*:

-nocase	Großbuchstaben in *string* werden als Kleinbuchstaben betrachtet
-indices	zwei Indizes geben den Bereich an, in dem in *string* gesucht werden soll
—	Beginnt das *pattern* mit einem **-**, so kann damit *flags* von *pattern* getrennt werden.

- Falls eine Teilstring von *string* zu der von *pattern* erzeugten Sprache gehört, so wird 1, ansonsten 0 zurückgegeben. Im ersten Fall wird der Teilstring in der Variablen *match* abgelegt; die restlichen Variablen werden mit den Teilstrings von *string* belegt, die zu den Teilmustern von *pattern* "passen" (entsprechend der Klammerung "("

Beispiel:

```
pey$tcl
tcl> set str 123abcde56789x
tcl> set str
123abcde56789x
tcl> set pat {[0-9]+[a-z]*}
tcl> set pat
[0-9]+[a-z]*
tcl> regexp $pat $str
1
tcl> regexp $pat $str match
1
tcl> set match
123abcde
tcl> set pat {([0-9]+).}
```

```
tcl> set pat
([0-9]+).
tcl> regexp $pat $str match sub
1
tcl> set match
123a
tcl> set sub
123
tcl>
tcl> set env(DISPLAY)
pey:0.0
tcl> # DISPLAY = hostname:display.screen
tcl> #hostname ermitteln:
tcl>
tcl> regexp {([^:]*):} $env(DISPLAY) match host
1
tcl> set match
pey:
tcl> set host
pey
tcl>
tcl> # oder so:
tcl> regexp (\[^:]*): $env(DISPLAY) match host
1
tcl> set match
pey:
tcl> set host
pey
tcl> exit
pey$
```

7.9 *regsub*-Kommando

→ String-Ersetzung basierend auf regulären Ausdrücken

Syntax:

`regsub ?switches? pattern string subspec varname`

- liefert Anzahl der Treffer / Ersetzungen
- kopiert *string* nach *varname*, wobei Teilstrings entsprechend *pattern* durch *subspec* ersetzt werden
- *switches*

-all	alle Vorkommen ersetzen, sonst nur das Erste
-nocase	Großbuchstaben werden zur Untersuchung in Kleinbuchstaben gewandelt
—	zur Trennung, 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 (

Beispiele:

```
pey$tcl
tcl> set str 123abcde56789x
tcl> set str
123abcde56789x
tcl> set pat {[0-9]+[a-z]*}
tcl> set pat
[0-9]+[a-z]*
tcl> regexp $pat $str
1
tcl> regexp $pat $str match
1
tcl> set match
123abcde
tcl> set pat {([0-9]+).}
tcl> set pat
([0-9]+).
tcl> regexp $pat $str match sub
1
tcl> set match
123a
tcl> set sub
123
tcl>
tcl> set env(DISPLAY)
pey:0.0
tcl> # DISPLAY = hostname:display.screen
tcl> #hostname ermitteln:
tcl>
tcl> regexp {([^:]*):} $env(DISPLAY) match host
1
tcl> set match
pey:
tcl> set host
pey
tcl>
tcl> # oder so:
tcl> regexp (\[^:]*): $env(DISPLAY) match host
1
tcl> set match
pey:
tcl> set host
```

```
pey
tcl> exit
pey$
```

Spalten vertauschen:

```
pey$ tcl
tcl> set line Franz:Schweiggert:Illertissen
tcl> set line
Franz:Schweiggert:Illertissen
tcl>
tcl> regsub {([^:]+):([^:]+):} $line {\2:\1:} swapline
1
tcl> set swapline
Schweiggert:Franz:Illertissen
tcl> exit
pey$
```

1. Spalte verdoppeln

```
pey$ tcl
tcl> set line Franz:Schweiggert:Illertissen
tcl> regsub {[^:]+} $line &:&: newline
1
tcl> set newline
Franz:Franz::Schweiggert:Illertissen
tcl> exit
pey$
```

8. Daten-Strukturen

8.1 Einfache Variable

```
pey$ tcl
tcl> set var {Wert von var}
tcl> set name var
tcl> set name
var
tcl> set var
Wert von var
tcl> set $name
Wert von var
tcl>
tcl> set [set name]
Wert von var
tcl> info exists name
1
tcl> info exists x
0
tcl> exit
pey$
```

Keine Liste, nur eine einfache Variable:

```
pey$ tcl
tcl> set liste {1 2 {8 9} 5 "6 7 8 {10 11}"}
tcl> set liste
1 2 {8 9} 5 "6 7 8 {10 11}"
tcl> set liste1 "1 2 {8 9} {6 7 8 {10 11} }"
tcl> set liste1
1 2 {8 9} {6 7 8 {10 11} }
tcl> exit
pey$
```

8.2 Listen

Übersicht

- Strings mit spezieller Interpretation
- Eine Liste hat gleiche Struktur wie ein Kommando
- Listen sind Strings, Leerraum dient zur Trennung der Elemente
- Geschweifte Klammern und Doppelapostrophen können zur Gruppierung verwendet werden (Listen als Elemente von Listen)
- Bei großen Datenmengen wenig effizient → Array's

Listen-Kommando's
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.

Listen-Kommando's
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 das erste Listenelement, 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.

lappend - llength - lindex **- Kommando's**

```
pey$ tcl
tcl> set x 1
tcl> set liste [list $x 2 {8 9}]
tcl> set liste
1 2 {8 9}
tcl> set liste [lappend liste \$ 10 11]
tcl> set liste
1 2 {8 9} {$} 10 11
tcl> set liste [lappend liste \ ]
tcl> set liste
1 2 {8 9} {$} 10 11 { }
tcl> set liste [lappend liste \" ]
tcl> set liste
1 2 {8 9} {$} 10 11 { } {"}
tcl> set liste [lappend liste \] ]
tcl> set liste
1 2 {8 9} {$} 10 11 { } {"} \]
```

```
tcl> set liste [lappend liste \} ]
tcl> set liste
1 2 {8 9} {$} 10 11 { } {"} \] \}
tcl> llength liste
1
tcl> llength $liste
10
tcl> exit
pey$
```

Mit Sonderzeichen:

```
pey$ tcl
tcl> set x \[
tcl> set x
[
tcl> set liste [list $x \[]
tcl> set liste
{[} {[}
tcl> set liste [lappend liste \[ \] ]
tcl> set liste
{[} {[} {[} \]
tcl> set liste [lappend liste {]} ]
tcl> set liste
{[} {[} {[} \] \]
tcl> lindex $liste 4
]
tcl> lindex $liste 3
]
tcl> lindex $liste 2
[
tcl> exit
pey$
```

Und nochmal:

```
pey$ tcl
tcl> set x {a b}
tcl> set x
a b
tcl> list $x c d
{a b} c d
tcl> exit
pey$
```

In der Anweisung `set x {a b}` dienen die geschweiften Klammern dazu, die Folge ’a b’ als ein Wort an die Variable x zu übergeben - sie werden vom *tcl*-Interpreter bei der Zuweisung entfernt. Daß sie in der Liste dennoch auftauchen, liegt daran, daß sie als ein Listenelement geführt werden, also vom *list*-Kommando wieder hinzugefügt werden.

lappend - Kommando

```
pey$ tcl
tcl> set liste [lappend liste 1 2]
tcl> set liste
1 2
tcl> exit
pey$
```

Falls die Liste, an die angefügt werden soll, noch nicht existiert, wird sie erzeugt!

concat - Kommando

```
pey$ tcl
tcl> set x {a b c}
tcl> set y {1 2 3 4 5 6}
tcl> set z X
tcl> set liste [concat $x $y $z]
tcl> set liste
a b c 1 2 3 4 5 6 X
tcl> set liste [concat "$x" $y "$z"]
tcl> set liste
a b c 1 2 3 4 5 6 X
tcl> exit
pey$
```

Die Argumente von *concat* werden einfach konkateniert!

Zum Unterschied dazu *list*:

```
pey$ tcl
tcl> set x {a b c}
tcl> set y {1 2 3 4 5 6}
tcl> set z X
tcl> set liste [list $x $y $z]
tcl> set liste
{a b c} {1 2 3 4 5 6} X
tcl> exit
pey$
```

ALSO:

- *list* und *lappend* erhalten Listenstruktur, während *concat* (oder *" "*) eine "Stufe" der Listenstruktur eliminieren!

llength - lindex - lrange - Kommando

```
pey$ tcl
tcl> set liste [list {1 2} a b c ]
tcl> set liste
{1 2} a b c
tcl> llength liste
1
tcl> llength $liste
4
tcl> lindex $liste 0
1 2
tcl> lindex $liste 1
a
tcl> lrange $liste 1 2
a b
tcl> lrange $liste 1 end
a b c
tcl> exit
pey$
```

linsert - lreplace - Kommando's

```
pey$ tcl
tcl> set liste [list {1 2} a b c]
tcl> set $liste
Error: can't read "{1 2} a b c": no such variable
tcl> set liste
{1 2} a b c
tcl> linsert $liste 0 X Y
X Y {1 2} a b c
tcl> set liste
{1 2} a b c
tcl> set liste [linsert $liste 0 X Y]
tcl> set liste
X Y {1 2} a b c
tcl> set liste [lreplace $liste 0 1 0]
tcl> set liste
0 {1 2} a b c
tcl> set liste [lreplace $liste 2 end X]
tcl> set liste
0 {1 2} X
tcl> set liste [lreplace $liste 2 2 XXX]
tcl> set liste
0 {1 2} XXX
tcl> exit
pey$
```

lsearch - Kommando

liefert Index, wo gesuchtes Element in Liste enthalten, -1 wenn nicht
default pattern matching: glob-style ("Dateimuster")
mode: -exact (genaues Vorkommen), -regexp (regulärer Ausdruck), -glob (default: s.o.)

```
pey$
tcl> set liste [list dies sind die Elemente]
tcl> set liste
dies sind die Elemente
tcl> lsearch -exact $liste die
2
tcl> lsearch $liste die*
0
tcl> lsearch -regexp $liste die.*
0
tcl> lsearch $liste d*
0
tcl> lsearch -regexp $liste d*
0
tcl> proc ldelete {liste wert } {
=>          set ix [lsearch -exact $liste $wert ]
=>          if {$ix >=0} {
=>                  return [lreplace $liste $ix $ix]
=>          } else {
=>                  return $liste
=>          }
=> }
tcl> ldelete $liste die
dies sind Elemente
tcl> set liste
dies sind die Elemente
tcl> set liste [ldelete $liste die]
tcl> set liste
dies sind Elemente
tcl> exit
pey$
```

lsort - Kommando

- Art der Sortierung: *-ascii* (default), *-integer*, *-real*
- Ordnung: *increasing* (default), *decreasing*
- Rückgabe: neue sortierte Liste, eingehende Liste bleibt unverändert
- Eigene Vergleichsfunktionen: *-command yourcompare*

Beispiel:

```
pey$tcl
tcl> set liste { {Mathias Etter} {Matthias Grabert}
=> {Andreas F. Borchert} {Franz. A. Schweiggert} {Axel T. Schreiner}}

tcl> lsort $liste
{Andreas F. Borchert} {Axel T. Schreiner} {Franz. A. Schweiggert}
{Mathias Etter} {Matthias Grabert}

tcl> 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]
=>         }
=> }
tcl> lsort -command famcmp $liste
{Andreas F. Borchert} {Mathias Etter} {Matthias Grabert}
{Axel T. Schreiner} {Franz. A. Schweiggert}
tcl> exit
pey$
```

split - *join* - Kommando's

Dazu nehmen wir die Datei "Person.all" unserer Datenbank:

```
pey$head -2 ../db/Person.all
PersId:Name:Vorname:Anrede:Institution:Tel
4711:Biertaler:Peter J.:Herr:BS&S:0815/10007

pey$ cat get_split
#!/usr/bin/tcl

set DB_FILE "~/tclSkr/db/Person.all"

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

gets $f line
set list [split $line :]
puts [format "%-10s %-10s %-10s" [lindex $list 3] \
                                  [lindex $list 2] \
                                  [lindex $list 1] ]
puts [format "%30s" ------------------------------- ]
```

```
gets $f line
set list [split $line :]
puts [format "%-10s %-10s %-10s\n" [lindex $list 3] \
                                   [lindex $list 2] \
                                   [lindex $list 1] ]

pey$ get_split
Anrede     Vorname    Name
---------------------------------
Herr       Peter J.   Biertaler

pey$
```

→ *join*: das "Inverse" von *split*

```
pey$ cat split_join
#!/usr/bin/tcl

set DB_FILE "~/tclSkr/db/Person.all"

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

gets $f line
set list [split $line :]
set x [list [lindex $list 1] [lindex $list 2] [lindex $list 3]]
puts $x
puts [join $x @]
gets $f line
set list [split $line :]
set x [list [lindex $list 1] [lindex $list 2] [lindex $list 3]]
puts $x
puts [join $x @]

pey$ split_join
Name Vorname Anrede
Name@Vorname@Anrede
Biertaler {Peter J.} Herr
Biertaler@Peter J.@Herr
pey$
```

8.3 (Assoziative) Array's

→ *awk*
→ Array-Indizierung: Strings
→ Belegung eines Elements: `set arrname(index) value`

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
array names arrname ?pattern?	Rückgabewert ist eine Liste aller Indizes des Arrays, oder jener, die zu *pattern* 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*

```
pey$tcl
tcl> set index Andreas
tcl> set SAI($index) Borchert
tcl> set SAI(Andreas)
Borchert

tcl> set SAI($index)
Borchert
```

```
tcl> set SAI 5
Error: can't set "SAI": variable is array

tcl> set SAI("Franz A.") Schweiggert
Error: wrong # args: should be "set varName ?newValue?"
tcl> set {SAI(Franz A.)} Schweiggert
tcl> set index "Franz A."
tcl> set SAI($index)
Schweiggert

tcl> array exists SAI
1
tcl> array get SAI
{Franz A.} Schweiggert Andreas Borchert

tcl> array names SAI
{Franz A.} Andreas

tcl> array size SAI
2
tcl> exit
pey$
```

Durchlaufen eines Array's

`foreach index [array names arrname] { cmd's }`

→ Ordnung der Elemente "willkürlich" (hashing)

- "mehrdimensionale" Array's:

```
pey$ cat matr.tcl
#! /usr/bin/tcl

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 ""

pey$ matr.tcl
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
pey$
```

- Achtung bei Blank's im Index:

```
pey$ cat matr1.tcl
#! /usr/bin/tcl

set dim 5

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

pey$ matr1.tcl
Error: wrong # args: should be "set varName ?newValue?"
    while executing
"set arr($i , $j) [expr $i+$j+2]"
    ("while" body line 3)
```

```
    invoked from within
"while { $j < $dim } {
        # Blanks im Index:
                set arr($i , $j) [expr $i+$j+2]
                incr j
        }"
    ("while" body line 3)
    invoked from within
"while { $i < $dim } {
        set j 0
        while { $j < $dim } {
        # Blanks im Index:
                set arr($i , $j) [expr $i+$j+2]
                incr j
        }
        incr i
}"
    (file "./matr1.tcl" line 6)
pey$
```

Zur Erklärung:

```
pey$ cat matr2.tcl
#! /usr/bin/tcl

set dim 5

set i 0
while { $i < $dim } {
        set j 0
        while { $j < $dim } {
        # Blanks im Index:
                set arr($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 ""

pey$ matr2.tcl
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:
Error: can't read "arr(0,0)": no such element in array
    while executing
"puts -nonewline "$arr($i,$i)..."
    ("while" body line 2)
    invoked from within
"while {$i < $dim } {
        puts -nonewline "$arr($i,$i) "
        incr i
}"
    (file "./matr2.tcl" line 30)
pey$
```

8.4 Erste Anwendungen

Punkte-Auswertung

Struktur der Punkte-Datenbank

```
pey$ cat punkte.l
Vorname(n) Name:MaxUebPkte 120
Jan Meier:10:8:2:0:4:8:0:0:0:0
Andreas L. Huber:0:9:15:20:18:12:0:0:0:0
Rainer Maria Rilke:2:2:10:12:20:0:0:0:0:0
Andrew Prinz:0:0:0:0:0:0:0:0:0:10
Boris Graf:12:6:6:6:8:9:10:12:4:4
Franz Uwe Rahn:2:10:10:10:10:10:10:8:9:10
Ralf H. Wolf:1:2:3:4:5:6:7:8:9:10
Rene Jean Papin:1:1:1:2:5:6:8:0:0:0
Josef Meier:12:12:12:12:12:12:12:12:12:12
pey$
```

- Bestimme die Gesamtpunktezahl - wer weniger als die Hälfte der maximal zu erreichenden Punktezahl hat, wird nicht zur Klausur zugelassen - Ausgabe nach Namen sortiert:

```
pey$ cat summe.tcl
#!/usr/bin/tcl

# Berechnung der Summe der Uebungspunkte
# Feststellen, ob mindestens 50% erreicht
# sortiert ausgeben

proc famcmp {a b} {
# Parameter sind Listen, deren erstes Element
# selbst eine Liste der Form
# {Vorname(n) Familienname} sind
# verglichen wird nach Familienname
# falls gleich wird der gesamte Namen
# zum Vergleich herangezogen

        set aFirst [lindex $a 0]
        set aFirstLast [lindex $aFirst [expr [llength $aFirst] -1] ]
        set bFirst [lindex $b 0]
        set bFirstLast [lindex $bFirst [expr [llength $bFirst] -1] ]
        set res [string compare $aFirstLast $bFirstLast]
        if {$res != 0} {
                return $res
        } else {
                return [string compare $aFirst $bFirst]
        }
}

set DB "~/tclSkr/pktDb/punkte.l"

set f [open $DB r]

#aus Kopfzeile max. Punktzahl holen
# und weglesen
gets $f line
regexp {[0-9]*$} $line maxPkt

while { [gets $f line] >= 0} {
        set liste [ split $line :]
        set name [list [lindex $liste 0] ]
        set punkte [lrange $liste 1 end]
        set pkt_sum [join $punkte +]
        set sum [expr $pkt_sum]
        set bestanden ja; #default
        if { $sum < (0.5*$maxPkt) } {
                set bestanden nein
        }

        lappend all "$name $sum $bestanden"
}

# nach Familienname sortieren:
set all_sorted [lsort -command famcmp $all]

# Ausgabe:
```

```
set i 0
while { $i < [llength $all_sorted] } {
        puts stdout [join [lindex $all_sorted $i] :]
        incr i
}
```

```
pey$ summe.tcl
Boris Graf:77:ja
Andreas L. Huber:74:ja
Jan Meier:32:nein
Josef Meier:120:ja
Rene Jean Papin:24:nein
Andrew Prinz:10:nein
Franz Uwe Rahn:89:ja
Rainer Maria Rilke:46:nein
Ralf H. Wolf:55:nein

pey$
```

- Vielleicht sollten wir dies besser als Filter realisieren. Dazu sind in vorigem Programm die beiden Zeilen
  `set DB "~/tclSkr/tcl/pktDb/punkte.l"`
  `set f [open $DB r]`
  durch die Zeile
  `set f stdin`
  zu ersetzen:

```
pey$ cat punkte.l | sumfilter.tcl
Boris Graf:77:ja
Andreas L. Huber:74:ja
Jan Meier:32:nein
Josef Meier:120:ja
Rene Jean Papin:24:nein
Andrew Prinz:10:nein
Franz Uwe Rahn:89:ja
Rainer Maria Rilke:46:nein
Ralf H. Wolf:55:nein

pey$
```

Daraus die Teilnehmer herauszufiltern, die an der Klausur teilnehmen können, ist jetzt ganz trivial:

```
pey$ cat punkte.l | sumfilter.tcl | egrep 'ja$' | cut -d: -f1
Boris Graf
Andreas L. Huber
Josef Meier
Franz Uwe Rahn

pey$
```

Nachdem natürlich niemand am Zugang zur Klausur gehindert wird, werden die Klausurergebnisse ebenfalls in die Summenpunkteliste eingetragen; diese hat die Struktur, wie sie von den Skripten *summe.tcl* bzw. *sumfilter.tcl* geliefert werden. Sie wird um eine weitere Spalte und eine Kopfzeile ergänzt:

```
pey$ echo "Name:UebPkte:Klausur(j/n):Klausur (100)"  > punkte.sum

pey$ summe.tcl >> punkte.sum

pey$ cat punkte.sum
Name:UebPkte:Klausur(i/n):Klausur (100)
Boris Graf:77:ja
Andreas L. Huber:74:ja
Jan Meier:32:nein
Josef Meier:120:ja
Rene Jean Papin:24:nein
Andrew Prinz:10:nein
Franz Uwe Rahn:89:ja
Rainer Maria Rilke:46:nein
Ralf H. Wolf:55:nein

pey$
```

Vielleicht sollten wir das einmal schön darstellen, indem wir das "Tabellenprogramm" der *groff*-Textverarbeitung verwenden. Dazu brauchen wir eine Datei, die folgende Struktur hat:

```
.TS
.tab(:);
c | c | c | c
l | r | l | r.
<<Kopfzeile>>
_
<<Datenzeile>>
  ...
<<Datenzeile>>
<<Datenzeile>>
.TE
```

Diese können wir auch mit *tcl* produzieren (wir sollten wohl eine allgemeine Lösung anstreben → Übung!):

```
pey$ tcl
tcl> set in [open punkte.sum  r ]
tcl> set out [open punktsum.tbl w]
tcl> puts $out ".TS"
tcl> puts $out "tab(:);"
tcl> puts $out "c | c | c | c"
tcl> puts $out "l | r | l | r."
tcl> gets $in line
tcl> puts $out $line
tcl> puts $out "_"
```

```
tcl> while { [gets $in line] >= 0} {
=>         puts $out $line
=> }
tcl> puts $out ".TE"
tcl> close $out
tcl> cat punktsum.tbl
.TS
tab(:);
c | c | c | c
l | r | l | r.
Name:UebPkte:Klausur(i/n):Klausur (100)
_
Andreas L. Huber:74:ja
Jan Meier:32:nein
Josef Meier:120:ja
Rene Jean Papin:24:nein
Andrew Prinz:10:nein
Franz Uwe Rahn:89:ja
Rainer Maria Rilke:46:nein
Ralf H. Wolf:55:nein
.TE
tcl> exit
pey$
```

Dies ergibt folgende Tabelle:

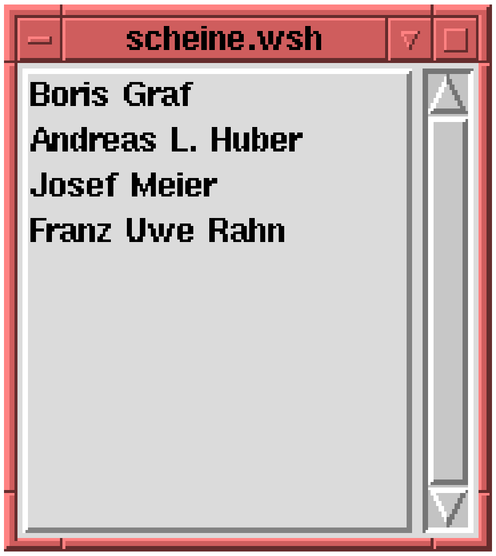
Name	UebPkte	Klausur(j/n)	Klausur (100)
Andreas L. Huber	74	ja	
Jan Meier	32	nein	
Josef Meier	120	ja	
Rene Jean Papin	24	nein	
Andrew Prinz	10	nein	
Franz Uwe Rahn	89	ja	
Rainer Maria Rilke	46	nein	
Ralf H. Wolf	55	nein	

In diese Liste werden nach der Klausur die Punkte eingetragen (eigentlich in die ursprüngliche, was aber hier nicht weiter beachtet werden soll). Unser Programm soll die Liste derer ermitteln, die einen Schein bekommen (Übung mindestens 50% der Punkte - Klausur mindestens 50 Punkte):

```
pey$ cat scheine.tcl
#!/usr/bin/tcl

#  Alle Teilnehmer (Namen), die einen
# Schein erhalten

set DB "~/tclSkr/pktDb/punkte.kl"

set f [open $DB r]
```

```
# Kopfzeile weglesen
gets $f line

while { [gets $f line] >= 0} {
        set liste [ split $line :]
        set name [list [lindex $liste 0] ]
        if { ([string compare [lindex $liste 2] ja] == 0)
                && ([expr [lindex $liste 3] >= 50] ) } {
                puts stdout [join $name " "]
        }
}

pey$ scheine.tcl
Boris Graf
Andreas L. Huber
Josef Meier
Franz Uwe Rahn

pey$
```

Vielleicht sollten wir der Sekretärin die Liste der Studenten, die einen Schein erhalten, etwas schöner darstellen:



zugehörige Quelle:

```
#!/usr/bin/wishx

#  Alle Teilnehmer (Namen), die einen
# Schein erhalten

set DB "~/tclSkr/pktDb/punkte.kl"

listbox .scheinlist -relief raised \
-borderwidth 2 -yscrollcommand ".scroll set"
```

```
pack .scheinlist -side left
scrollbar .scroll -command ".scheinlist yview"
pack .scroll -side right -fill y

set f [open $DB r]
# Kopfzeile weglesen
gets $f line

while { [gets $f line] >= 0} {
        set liste [ split $line :]
        set name [list [lindex $liste 0] ]
        if { ([string compare [lindex $liste 2] ja] == 0)
                && ([expr [lindex $liste 3] >= 50] ) } {
                .scheinlist insert end [join $name " "]
        }
}
```

Einige DB-Operationen

Ausgangstabelle:

PERSON					
PersId	Name	Vorname	Anrede	Institution	Tel
4711	Biertaler	Peter J.	Herr	BS&S	0815/10007
4712	Oha	Alfred V.	Herr	BS&S	0815/10008
4713	Koernigen	Brian W	Herr	BS&S	0815/10008
4714	Reichle	Dennis M	Herr	BS&S	0815/10009
4715	Stanglow	Werner	Herr	Uni Ulm	0731/5023572
4716	Rahn	Uwe	Herr Prof.	FH Nixstadt	0999/9999
4717	Ross	Detlev	Herr	HalComp Inc.	0711/123456
4718	Schweiggert	Franz	Herr Prof. Dr.	Uni Ulm	0731/5023570
4719	Ruebe	Berta	Frau Dr.	XSoft AG	
4720	Blau	Bernd	Herr	SPG GmbH	0511/546-1234
4721	Rot	Ulf	Herr		07031/98765
4722	Winter	Ralf	Herr Dr.	Xplus GmbH	0241/4098861
4723	Maier	Urs	Herr	CoreDump AG	0421/442812
4724	Meyer	Ute	Frau Dr.med.	MedSoft GmbH	040/606320
4725	Meier	Ulf	Herr		
4726	Bedchaus	Karin	Frau	Steuerbuero Maus	06131/54321
4727	Abs	Karl	Herr	Softikus AG	07303/5857
4728	Stange	Inge	Frau Prof. Dr.	Techn. Uni.	
4729	Enter	Jo	Herr Dr.	ElektroBank AG	
4730	Emu	F.J.	Herr Dr.h.c.	SSS e.V.	0511/4321645
4731	Streit	Max	Herr		
4732	Dann	Else	Frau		0541/7720969
4733	List	C.	Herr	CRG mbH	0421/244218
4734	Mai	Karl	Herr Dr.		0731/12534
4735	Maerz	August	Herr	Maerz&Partner	0731/523571
4736	Zank	Werner	Herr Dr.	Softikus AG	0731/5857
4737	Schon	Gerda	Frau	Softness GmbH	089/980321
4738	Mair	Josef	Herr	IEJ VaG	0911/5112323

- Über einen Namen Eintrag auswählen:

```
pey$ cat select1.tcl
#!/usr/local/bin/tcl

set PHONEBOOK "~/tclSkr/db/Person.all"

# one argument expected - name of person
# returns whole record

if { $argc != 1 } {
   puts stderr "usage: $argv0 name"
    exit 1
}
```

```
set name [lindex $argv 0 ]

proc display {line} {
   regsub -all : $line " " new
   puts $new
}


set f [open $PHONEBOOK r]

while { [gets $f line] >= 0} {
   if [regexp -nocase $name $line ] {
      display $line
      exit 0
   }
}
close $f

pey$ select1.tcl
usage: ./select1.tcl name

pey$ select1.tcl Meier
4725 Meier Ulf Herr  Kohlring 3 2000 Hamburg 4

pey$
```

- Über einen Namen nur Telefonnummer auswählen

```
pey$ cat select2.tcl
#!/usr/local/bin/tcl

set PHONEBOOK "~/tclSkr/db/Person.all"

# one argument expected - name of person
# returns name & phonenumber

if { $argc != 1 } {
   puts stderr "usage: $argv0 name"
    exit 1
}

set name [lindex $argv 0 ]

proc tdisplay {line} {
   set list [split $line :]
   set name [lindex $list 1]
   set phone [lindex $list [expr [llength $list] -1 ]]
   if { [string length $phone] <= 0 } {
      set phone "Telefonnummer unbekannt"
   }
```

```
   puts "$name : $phone "
}


set f [open $PHONEBOOK r]

while { [gets $f line] >= 0} {
   if [regexp -nocase $name $line ] {
      tdisplay $line
      exit 0
   }
}
close $f

pey$ select2.tcl
usage: ./select2.tcl name

pey$ select2.tcl Oha
Oha : 0815/10008

pey$ select2.tcl oHA
Oha : 0815/10008

pey$ select2.tcl Meier
Meier : Telefonnummer unbekannt

pey$
```

- Tabelle in Array umwandeln: Indizes sind die PersId’s, Elemente sind Name, Vorname, Anrede

```
pey$ cat makearr1.tcl
#!/usr/local/bin/tcl

set PHONEBOOK "~/tclSkr/db/Person.all"

set f [open $PHONEBOOK r]

while { [gets $f line] >= 0} {
   set list [split $line :]
   set arr([lindex $list 0]) [lrange $list 1 3]
}

foreach index [array names arr] {
   puts "$index : $arr($index) "
}
close $f

pey$ makearr1.tcl
4723 : Maier Urs Herr
4724 : Meyer Ute {Frau Dr.med.}
```

```
4725 : Meier Ulf Herr
4726 : Bedchaus Karin Frau
4727 : Abs Karl Herr
4728 : Stange Inge {Frau Prof. Dr.}
4711 : Biertaler {Peter J.} Herr
4730 : Emu F.J. {Herr Dr.h.c.}
4729 : Enter Jo {Herr Dr.}
4712 : Oha {Alfred V.} Herr
4731 : Streit Max Herr
4713 : Koernigen {Brian W} Herr
4732 : Dann Else Frau
4714 : Reichle {Dennis M} Herr
4733 : List C. Herr
4715 : Stanglow Werner Herr
4734 : Mai Karl {Herr Dr.}
4716 : Rahn Uwe {Herr Prof.}
4735 : Maerz August Herr
4717 : Ross Detlev Herr
4736 : Zank Werner {Herr Dr.}
4718 : Schweiggert Franz {Herr Prof. Dr.}
4737 : Schon Gerda Frau
4719 : Ruebe Berta {Frau Dr.}
4720 : Blau Bernd Herr
4738 : Mair Josef Herr
4721 : Rot Ulf Herr
4722 : Winter Ralf {Herr Dr.}
PersId : Name Vorname Anrede

pey$
```

- wie oben (Element nur Name) und von dort in Liste (gelesen wird von *stdin*)

```
pey$ cat makearr2.tcl
#!/usr/local/bin/tcl

set f  stdin
while { [gets $f line] >= 0} {
   set list [split $line :]
   set arr([lindex $list 0]) [lindex $list 1 ]
}

foreach index [array names arr] {
   puts "$index : $arr($index) "
}
puts "------------------------------------------"
puts "Als Liste:\n"

set list [array get arr]
puts $list
puts "------------------------------------------"
```

```
puts "Einzeln:\n"

set i 0
while { $i < [llength $list] } {
   puts "[lindex $list $i]"
   incr i
}
close $f

pey$ head -3 ../Person.all | makearr2.tcl
4711 : Biertaler
4712 : Oha
PersId : Name
-----------------------------------------
Als Liste:

4711 Biertaler 4712 Oha PersId Name
-----------------------------------------
Einzeln:

4711
Biertaler
4712
Oha
PersId
Name

pey$
```

- Prüfen der Objektintegrität (*PersId* definiert und eindeutig?)

```
pey$ cat checkId.tcl
#!/usr/local/bin/tcl

set PHONEBOOK "~/tclSkr/db/FPerson.all"

set f [open $PHONEBOOK r]

while { [gets $f line] >= 0} {
   set list [split $line :]
   set persId [lindex $list 0]
   if { [string length $persId ] <= 0 } {
      puts stderr "empty key:"
      puts stderr $list
   }
   if [info exists arr($persId)] {
      incr arr($persId)
      if { $arr($persId) > 1 } {
         puts stderr "duplicate:"
         puts stderr $list
```

```
      }
   } else {
      set arr($persId) 1
   }
}

pey$ cat ../FPerson.all
4711:Biertaler:Peter J.:Herr
4712:Oha:Alfred V.:Herr
4713:Koernigen:Brian W:Herr
4714:Reichle:Dennis M:Herr
4713:Stanglow:Werner:Herr
:Rahn:Uwe:Herr Prof.
4717:Ross:Detlev:Herr
4718:Schweiggert:Franz:Herr Prof. Dr.
4719:Ruebe:Berta:Frau Dr.
4720:Blau:Bernd:Herr
4721:Rot:Ulf:Herr

pey$ checkId.tcl
duplicate:
4713 Stanglow Werner Herr
empty key:
{} Rahn Uwe {Herr Prof.}
pey$
```

- Projektion auf Spaltennamen

```
#!/usr/local/bin/tcl

if { $argc == 0 } {
   puts stderr "usage: $argv0 attr attr ..."
   exit 1
}

set DB "../Person.all"

set f [ open $DB r ]

#Welche Attribute  hat die Tabelle?

gets $f head

set headlist [ split $head : ]
#============================================
# Abbildung: Attr.Bezeichnung -> Spaltenummer:
set i 0
while { $i < [llength $headlist] } {
   set headarr([lindex $headlist $i]) $i
   incr i
}
```

```
#==============================================
puts "Kontrollausgabe (hashing!)"
foreach index [array names headarr] {
   puts "$index : $headarr($index)"
}
puts "---------------------------------------"
#=============================================

#Sind die Attr-Argumente zulaessige Attribute?
set i 0
while { $i < $argc} {
   if { [info exists headarr([lindex $argv $i]) ]} {
      incr i
   } else {
      puts stderr "Nichtzul. Attribut: >[lindex $argv $i]<"
      exit 1
   }
}
#=============================================
#Projektion:
while { [gets $f line] >= 0 } {
   set list [ split $line :]
   set i 0
   while { $i < $argc } {
      set spaltenNr $headarr([lindex $argv $i])
      if {$i > 0} {
         puts -nonewline ":"
      }
      puts -nonewline "[lindex $list $spaltenNr]"
      incr i
   }
   puts ""
}
```

Ausführung:

```
pey$ project.tcl
usage: ./project.tcl attr attr ...

pey$ project.tcl Name Tel
Kontrollausgabe (hashing!)
Vorname : 2
Tel : 5
Name : 1
Institution : 4
Anrede : 3
PersId : 0
-------------------------------
Biertaler:0815/10007
Oha:0815/10008
```

```
Koernigen:0815/10008
Reichle:0815/10009
Stanglow:0731/5023572
Rahn:0999/9999
Ross:0711/123456
Schweiggert:0731/5023570
Ruebe:
Blau:0511/546-1234
Rot:07031/98765
Winter:0241/4098861
Maier:0421/442812
Meyer:040/606320

pey$ project.tcl Name xxx PersId
Kontrollausgabe (hashing!)
Vorname : 2
Tel : 5
Name : 1
Institution : 4
Anrede : 3
PersId : 0
-------------------------------
Nichtzul. Attribut: >xxx<

pey$
```

9. Kontrollstruktur-Kommando's

9.1 Übersicht

Kontrollstruktur-Kommando's
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 Ergebnis wird zurückgegeben.
for init test reinit body Entspricht der for-Schleife in C. *init, reinit* und *body* sind gültige Tcl-Skripte, *test* ist 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.

Kontrollstruktur-Kommando's
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 *pattern*s 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. Terminiert, wenn *test* mit Null (0) bewertet wird. Gibt einen leeren String zurück.

- Rückgabewert: der des zuletzt ausgeführten Kommando's
- Kommando's, die Bool'sche Ausdrücke beinhalten (Bedingungen), greifen auf das *expr*-Kommando zurück

9.2 *if*-Kommando

- mindestens zwei Argumente: Bedingung und Ausführungsteil
- die *elseif*-Teile wie auch der *else*-Teil sind weitere Argumente

Beispiele:

```
tcl> set x -8
tcl> if { $x < 0 } { set x [expr -$x] }
8
tcl> puts $x
8
tcl> if { $x > 0 } {
=>          set x [ expr -$x ]
=>          }
-8
tcl> if { $x < 0 }
Error: wrong # args: no script following " $x < 0 " argument
tcl> >>>
```

9.3 *switch*-Kommando

```
pey$ cat switch.tcl
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 }
  }
}

pey$ tcl
tcl> source switch.tcl
jan
Januar
dec
Dezember
xxx
Ungueltige Angabe
```

```
pey$
```

- *default*: dient nur dann als "Auffangfall", falls als letzte Alternative angegeben

Oder so:

```
pey$ cat switch1.tcl
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 }
}

pey$ tcl
tcl> source switch1.tcl
jan
Januar
nov
November
okt
Ungueltige Angabe

pey$
```

Mit regulären Ausdrücken: *-regexp*

Erster Versuch:

```
pey$ cat switch2.tcl
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"

pey$ tcl
tcl> source switch2.tcl
Anne
Error: invalid command name "0-9"
tcl> exit
pey$
```

- Woran liegt's? - Die eckigen Klammern sind sowohl bei regulären Ausdrücken als auch für *tcl* Metasymbole

Neuer Versuch:

```
pey$ cat switch3.tcl
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"

pey$ tcl
tcl> source switch3.tcl
Anne
123
xyz
Alfons
456
 ^d
Anzahl mit A am Anfang: 0
Anzahl Zahlen: 2
Anzahl andere: 3
tcl> exit
pey$
```

- Und wieder nicht! Warum? - Im Ausdruck *^$anfang* wird wegen des Schutzes durch die geschweiften Klammern keine Variablensubstitution vorgenommen!

Noch ein Versuch:

```
pey$ cat switch4.tcl
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"

pey$ tcl
tcl> source switch4.tcl
Anne
123
Alfons
Annette
456
xxx
^d
Anzahl mit A am Anfang: 3
Anzahl Zahlen: 2
Anzahl andere: 1
tcl> exit
pey$
```

Also doch!!!

Eine kleine Änderung (unsichtbar):

```
pey$ cat switch5.tcl
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"

pey$ tcl
tcl> source switch5.tcl
Anne
Error: invalid command name "^[0-9]+$"
tcl> exit
pey$ cat switch5.tcl | od -bc #Ausgabe manuell etwas bereinigt

136 044 141 156 146 141 156 147 040 173 040 151 156 143 162 040
  ^   $   a   n   f   a   n   g       {       i   n   c   r
172 061 040 175 040 134 040 012 011 011 173 136 133 060 055 071
  z   1       }       \      \n  \t  \t   {   ^   [   0   -   9
135 053 044 175 040 173 040 151 156 143 162 040 172 062 040 175
  ]   +   $   }       {       i   n   c   r       z   2       }
040 134 012 011 011 144 145 146 141 165 154 164 040 173 040 151
  \  \n  \t  \t   d   e   f   a   u   l   t       {       i
156 143 162 040 172 063 040 175 012 175 012 160 165 164 163 040
  n   c   r       z   3       }  \n   }  \n   p   u   t   s
```

Da hat sich doch zwischen einem Backslash und dem zu schützenden *newline* ein Leerzeichen eingeschlichen!!!

Und noch eine Möglichkeit:

```
pey$ cat switch6.tcl
set z1 0
set z2 0
set z3 0
set anfang A

while { [gets stdin eingabe ] >= 0 } {
        switch -glob -- $eingabe "
                ${anfang}* { incr z1 }
                X* -
                Y*  { incr z2 }
                default { incr z3 }
        "
}
puts "Anzahl mit $anfang am Anfang: $z1"
puts "Anzahl mit X oder Y am Anfang $z2"
puts "Anzahl andere: $z3"

pey$ tcl
tcl> source switch6.tcl
Anne
Xanten
Ypsilon
Zeiiang
anton
 ^d
```

```
Anzahl mit A am Anfang: 1
Anzahl mit X oder Y am Anfang 2
Anzahl andere: 2
tcl> exit
pey$
```

Sollten innnerhalb des *switch*-Kommando's oben Doppelapostrophen benötigt werden, müssen diese durch Backslash geschützt werden!

- Kommentare nur dort, wo *tcl* den Anfang eines Kommandos erwartet:

So?

```
pey$ cat switch7.tcl
set x #
switch -- $x {
        # Fallunterscheidung:
        ? {puts Fragezeichen}
        # {puts Lattenzaun }
        default { puts sonstiges }
}
pey$ tcl -f switch7.tcl
Error: invalid command name "Fallunterscheidung:"
    while executing
"Fallunterscheidung:"
    ("#" arm line 1)
    invoked from within
"switch -- $x {
        # Fallunterscheidung:
        ? {puts Fragezeichen}
        # {puts Lattenzaun }
        default { puts sonstiges }
}"
    (file "switch7.tcl" line 2)
pey$
```

Oder so?

```
pey$ cat switch8.tcl
set x #
switch -- $x {
        ? { # Fallunterscheidung:
            puts Fragezeichen}
        # {puts Lattenzaun; # Kommentar }
        default { puts sonstiges }
}

pey$ tcl -f switch8.tcl
Lattenzaun
pey$
```

9.4 *foreach*-Kommando

Beispiele:

```
pey$ cat foreach1.tcl
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"

pey$ tcl -f foreach1.tcl
Summe: 45

pey$
```

```
pey$ cat foreach2.tcl
#!/usr/bin/tcl
set wert -1
set outOption 0
set outFile stdout
set inFile stdin

foreach arg $argv {
  switch -regexp -- $arg {
        -([0-9]+)       {scan $arg -%d wert}
        -o              {set outOption 1}
        default         {if { $outOption == 1 } {
                                set outFile $arg
                                set outOption 0
                         } else {
                                set inFile $arg
                         }
                        }
  }
}

puts "Wert: $wert"
puts "outfile: $outFile"
puts "infile: $inFile"

pey$ foreach2.tcl -123 -o output input
Wert: 123
outfile: output
infile: input
pey$
```

Hier kann man sehen, warum die Option -- bei *switch* wichtig ist:

```
pey$ cat foreach3.tcl
#!/usr/bin/tcl
set wert -1
set outOption 0
set outFile stdout
set inFile stdin

foreach arg $argv {
  switch -regexp  $arg {
        -([0-9]+)       {scan $arg -%d wert}
        -o              {set outOption 1}
        default         {if { $outOption == 1 } {
                                set outFile $arg
                                set outOption 0
                         } else {
                                set inFile $arg
                         }
                        }
  }
}

puts "Wert: $wert"
puts "outfile: $outFile"
puts "infile: $inFile"

pey$ foreach3.tcl -123 -o output input
Error: bad option "-123": should be -exact, -glob, -regexp, or --
    while executing
"switch -regexp  $arg {
        -([0-9]+)       {scan $arg -%d wert}
        -o              {set outOption 1}
        default         {if { $outOption == 1 } {
                                set outFile $arg
                                set outOpti ..."
    ("foreach" body line 2)
    invoked from within
"foreach arg $argv {
  switch -regexp  $arg {
        -([0-9]+)       {scan $arg -%d wert}
        -o              {set outOption 1}
        default         {if { $outOption == 1 } {
                                set outFil ..."
    (file "./foreach3.tcl" line 7)
pey$
```

- Vorsicht:

```
pey$ cat foreach4.tcl
set a "Ich bin muede"
set b und

foreach w "$a $b gehe schlafen ([exec date])" {
        puts $w
}
```

```
pey$ tcl -f foreach4.tcl
Ich
bin
muede
und
gehe
schlafen
(Fri
Feb
16
22:00:42
GMT
1996)
pey$
```

Es war wohl so gemeint:

```
pey$ cat foreach5.tcl
set a "Ich bin muede"
set b und

foreach w [list $a $b gehe schlafen ([exec date])] {
        puts $w
}

pey$ tcl -f foreach5.tcl
Ich bin muede
und
gehe
schlafen
(Fri Feb 16 22:01:52 GMT 1996)
pey$
```

9.5 *while*-Kommando

```
pey$ cat while.tcl
set nl 0
set nw 0
set nc 0
while { [gets stdin line] >= 0 } {
        incr nl
        incr nc [string length $line]
        incr nw [ llength [split $line] ]
}

puts "Number Lines: $nl"
puts "Number Words: $nw"
puts "Number Chars: $nc"

pey$ cat text
```

```
Dies ist ein Text
mit drei Zeilen, 12 Worten
und 61 Zeichen.

pey$ cat text | tcl -f while.tcl
Number Lines: 3
Number Words: 12
Number Chars: 58

pey$ wc text
      3      12      61 text
pey$
```

Es sollte klar sein, warum obiges *tcl*-Skript nicht ganz so zählt wie *wc*!

9.6 *for*-Kommando

Beispiel:

```
pey$ cat for.tcl
set prod 1
for { set i 1 } { $i < 10 } { incr i 2 } {
        lappend oddList $i
        set prod [expr $prod * $i ]
}
puts "Liste der ungeraden Zahlen: $oddList"
puts "Produkt dieser Zahlen: $prod"

pey$ tcl -f for.tcl
Liste der ungeraden Zahlen: 1 3 5 7 9
Produkt dieser Zahlen: 945

pey$
```

10. Prozeduren

10.1 Syntax

```
proc name parameters body
```

- Die Prozedur *name* wird zu den Kommando's, die *tcl* "versteht", hinzugefügt (wenn also einmal definiert, so ist *name* so wie jedes andere *tcl*-Kommando zu verwenden)
- Ergebnis: das des letzten in *body* ausgeführten Kommando's oder speziell durch *return*-Kommando definierter String
- Rekursion möglich
- *call-by-value* (→ *upvar*-Kommando)
- Variable, die außerhalb definiert sind, sind in der Prozedur nicht sichtbar (→ *global*-Kommando)
- Prozeduren können zwar geschachtelt werden, sind aber immer global sichtbar! Schachteln macht also wenig Sinn (s.u.)
- Prozeduren haben einen eigenen Namensraum
- Zahl aktueller und formaler Parameter muß nicht übereinstimmen (letzter formaler Parameter muß dann *args* heißen)
- Parameter können mit *default*-Wert vereinbart werden

```
pey$ cat proc1.tcl
proc compute { a { b 2 } { c 10 } } {
        puts -nonewline "($a + $b) * $c gibt: "
        expr ( $a + $b ) * $c
}

pey$ tcl
tcl> source proc1.tcl
tcl> compute 2
(2 + 2) * 10 gibt: 40
tcl> compute 23 4
(23 + 4) * 10 gibt: 270
tcl> compute 1 2 3
(1 + 2) * 3 gibt: 9
tcl> compute
Error: no value given for parameter "a" to "compute"
tcl> exit
pey$
```

Mehr aktuelle als formale Parameter?

```
pey$ cat proc2.tcl
proc listArgs { a { b 2 } { c 10 } d } {
        puts "first: $a"
        puts "second: $b"
        puts "third: $c"
        puts "fourth: $d"
}
```

```
pey$ tcl
tcl> source proc2.tcl
tcl> listArgs one two
Error: no value given for parameter "d" to "listArgs"
tcl> listArgs one two three four
first: one
second: two
third: three
fourth: four
tcl> listArgs one two three four five
Error: called "listArgs" with too many arguments
tcl> exit
pey$
```

Mit *args*:

```
pey$ cat proc3.tcl
proc listArgs { a { b 2 } { c 10 } args } {
        puts "first: $a"
        puts "second: $b"
        puts "third: $c"
        puts "fourth: $args"
}
pey$ tcl
tcl> source proc3.tcl
tcl> listArgs one two
first: one
second: two
third: 10
fourth:
tcl> listArgs
Error: no value given for parameter "a" to "listArgs"
tcl> listArgs one two three four five six
first: one
second: two
third: three
fourth: four five six
tcl> exit
pey$
```

10.2 Umbenennen

Syntax:

```
rename oldname newname
```

- Erweitern einer bestehenden Prozedur (Kommando)
- Verbergen von Prozeduren (Kommando's)
  newname = { }

```
pey$ cat fak1.tcl
# erste nicht sehr robuste Implementierung
# der Fakultaet
proc fak n {
        if { $n ==  1 } {
                return 1
        } else {
                return [expr $n * [fak [expr $n - 1 ] ] ]
        }
}

# Umbenennen, aber Zugriff auf alte Version
# behalten

rename fak fak.old

# Erweiterung

proc fak n {
        if { $n <= 0 } {
                return 1
        } else {
                return [ fak.old $n ]
        }
}

set n 5
puts "$n Fakultaet ist [fak $n]"
set n -5
puts "$n Fakultaet ist [fak $n]"

pey$ tcl -f fak1.tcl
5 Fakultaet ist 120
-5 Fakultaet ist 1

pey$
```

Oder:

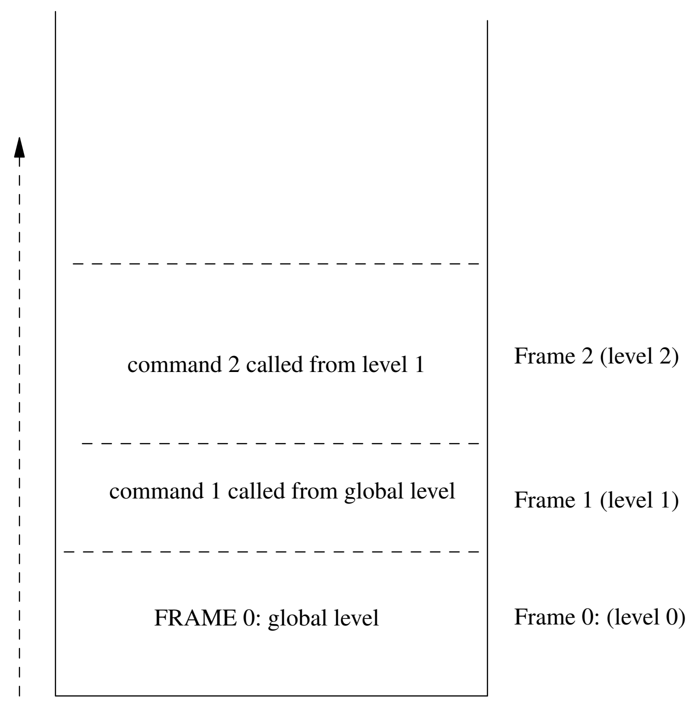
```
rename exec {}
```

→ exec kann nicht mehr aufgerufen werden

10.3 Lebensdauer

Lebensdauer lokaler Variablen beschränkt auf Ausführung der Prozedur

10.4 Sichtbarkeit - *global*-Kommando



- Globaler Gültigkeitsbereich (*top level scope*)
  → alle Prozeduren sind hier enthalten, ebenso alle Variablen außerhalb von Prozeduren
  Will eine Prozedur auf globale Variable zugreifen, müssen diese explizit "importiert" werden (→ *global*)
  Ist eine solche Variable "undefiniert", so wird sie dadurch definiert!

```
pey$ cat random.tcl

# aus Welch:

proc randomInit { seed } {
        global rand ;    # bezieht sich auf alle
                         # Feldelemente
        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 ;    # muss auch hier
                         # 'importiert' werden
```

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

proc randomRange { range } {
        expr int ([random]*$range)
}

puts "Startwert (PID): [randomInit [pid] ] "

for { set i 1 } { $i <= 5 } { incr i } {
        set f1($i) [ random ]
}
for { set i 1 } { $i <= 5 } { incr i } {
        set f2($i) [ randomRange 100 ]
}
for { set i 1 } { $i <= 5 } { incr i } {
        puts [format "%-8.4f   |   %4d" $f1($i) $f2($i)]
}

pey$ tcl -f random.tcl
Startwert (PID): 346
0.0065     |     95
0.9342     |     43
0.7998     |     14
0.1374     |     88
0.2217     |     30
pey$ tcl -f random.tcl
Startwert (PID): 348
0.0863     |      8
0.6060     |     49
0.3909     |     87
0.5675     |     49
0.1142     |     27
pey$
```

- Vermeide viele *global*'s - fasse die globalen, in Prozeduren zu verwendenden Variablen in ein Array zusammen!

10.5 *upvar*-Kommando: *call-by-reference*

Syntax:

```
upvar ?level? varName localVarName
```

- Zugriff auf Variable außerhalb des Kontexts einer Prozedur (anderer Bereich auf dem *tcl*-Stack
- Übergabe des Namens einer Variablen (statt Wert) an Parameter (siehe auch *eval*-Kommando)
- *level* hat als default-Wert 1, bezieht sich also auf 1 Stufe "oberhalb" im *tcl*-Stack (oder "unterhalb", je nach Stack-Organisation)
- mit *#number* kann absolut auf Stack-*frames* Bezug genommen werden: #0 bezieht sich also auf das top-Level, den globalen Gültigkeitsbereich
- Namensübergabe ist zulässig bei skalaren Variablen und Array-Elementen (lokale Variable ist jeweils skalar) sowie bei ganzen Array's (lokale Variable wird dann ebenfalls als Array betrachtet)

Beispiele:

```
pey$ cat proc5.tcl
proc do { varName } {
        upvar varName var
        puts $varName
        set var 5
}

set x 1
do x
puts $x

pey$ tcl -f proc5.tcl
x
1

pey$ cat proc6.tcl
proc do_again { varName } {
        upvar $varName var
        puts $varName
        set var 5
}

set x 1
do_again x
puts $x

pey$ tcl -f proc6.tcl
x
5

pey$
```

ODER:

```
pey$ cat proc5a.tcl
proc do { varName } {
        upvar varName var
        puts $varName
        puts $var
        set var 5
}

set x 1
do x
puts $x

pey$ tcl -f proc5a.tcl
x
Error: can't read "var": no such variable
    while executing
"puts $var..."
    (procedure "do" line 4)
    invoked from within
"do x"
    (file "proc5a.tcl" line 9)

pey$ cat proc6a.tcl
proc do_again { varName } {
        upvar $varName var
        puts $varName
        puts $var
        set var 5
}

set x 1
do_again x
puts $x

pey$ tcl -f proc6a.tcl
x
1
5

pey$
```

- *upvar $varName var*

```
pey$ cat printByName.tcl
proc printByName { varName } {
        upvar $varName var
        puts "$varName = $var"
}

set varname varwert
```

```
printByName varname
set "The name" "The value"
printByName "The name"

pey% tcl -f printByName.tcl
varname = varwert
The name = The value

pey$
```

- Übergabe von Arrays:

```
pey$ cat stack.tcl
# 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]

pey$ tcl -f stack.tcl
3
```

```
2
b
a
1
pey$
```

10.6 Zwei Varianten für Permutationen

```
pey$ cat perm.tcl

proc printFeld { } {
   global feld
   global anzahl
# feld: Feld der zu auszugebenden Objekte [1 .. anzahl]
# anzahl : Anzahl der Elemente, also anzahl oberster Index
   for {set i 1 } { $i <= $anzahl } { incr i } {
   puts -nonewline "$feld($i)  "
   }
   puts " "
}


proc tausche { x y } {
   upvar $x a
   upvar $y b
   set tmp $a
   set a $b
   set b $tmp
}

proc perm { n nr_perm } {
   global feld
   upvar $nr_perm nr

   if { $n == 1 } {
      incr nr
      puts -nonewline [format "%3d. Permutation: " $nr ]
      printFeld
   } else {
      perm [expr $n - 1 ] nr
      for { set i 1 } { $i <= [expr $n - 1] } \
         { incr i } {

         tausche feld($i) feld($n)
         perm [expr $n - 1] nr
         tausche feld($i) feld($n)
      }
   }
}

set anzahl  4
```

```
for { set i 1 } { $i <= $anzahl } { incr i } {
   set feld($i) $i
}

set nr 0

perm $anzahl nr

pey$ tcl -f perm.tcl
  1. Permutation: 1  2  3  4
  2. Permutation: 2  1  3  4
  3. Permutation: 3  2  1  4
  4. Permutation: 2  3  1  4
  5. Permutation: 1  3  2  4
  6. Permutation: 3  1  2  4
  7. Permutation: 4  2  3  1
  8. Permutation: 2  4  3  1
  9. Permutation: 3  2  4  1
 10. Permutation: 2  3  4  1
 11. Permutation: 4  3  2  1
 12. Permutation: 3  4  2  1
 13. Permutation: 1  4  3  2
 14. Permutation: 4  1  3  2
 15. Permutation: 3  4  1  2
 16. Permutation: 4  3  1  2
 17. Permutation: 1  3  4  2
 18. Permutation: 3  1  4  2
 19. Permutation: 1  2  4  3
 20. Permutation: 2  1  4  3
 21. Permutation: 4  2  1  3
 22. Permutation: 2  4  1  3
 23. Permutation: 1  4  2  3
 24. Permutation: 4  1  2  3
pey$
```

Mit weniger *global*'s:

```
pey$ cat perm1.tcl

proc printFeld { feld anz } {
   upvar $feld f
# feld[1 .. anz]: auszugebende Objekte
   for {set i 1 } { $i <= $anz } { incr i } {
   puts -nonewline "$f($i)  "
   }
   puts " "
}


proc tausche { x y } {
   upvar  $x a
```

```
   upvar  $y b
   set tmp $a
   set a $b
   set b $tmp
}

proc perm { feld n nr_perm anz } {
# feld: zu permutierendes Feld
# n : feld[1..n] aktuell zu permutieren
# nr_perm: zaehlt die Permutationen
# anz: feld[1..anz] Gesamtfeld

   upvar $feld f
   upvar $nr_perm nr

   if { $n == 1 } {
      incr nr
      puts -nonewline [format "%3d. Permutation: " $nr ]
      printFeld f $anz
   } else {
      perm f [expr $n - 1 ] nr $anz
      for { set i 1 } { $i <= [expr $n - 1] } \
         { incr i } {

         tausche f($i) f($n)
         perm f [expr $n - 1] nr $anz
         tausche f($i) f($n)
      }
   }
}

set anzahl  4

for { set i 1 } { $i <= $anzahl } { incr i } {
   set feld($i) $i
}

set nr 0

perm feld $anzahl nr $anzahl
```

```
pey$ tcl -f perm1.tcl
  1. Permutation: 1  2  3  4
  2. Permutation: 2  1  3  4
  3. Permutation: 3  2  1  4
  4. Permutation: 2  3  1  4
  5. Permutation: 1  3  2  4
  6. Permutation: 3  1  2  4
  7. Permutation: 4  2  3  1
  8. Permutation: 2  4  3  1
```

```
  9. Permutation: 3  2  4  1
 10. Permutation: 2  3  4  1
 11. Permutation: 4  3  2  1
 12. Permutation: 3  4  2  1
 13. Permutation: 1  4  3  2
 14. Permutation: 4  1  3  2
 15. Permutation: 3  4  1  2
 16. Permutation: 4  3  1  2
 17. Permutation: 1  3  4  2
 18. Permutation: 3  1  4  2
 19. Permutation: 1  2  4  3
 20. Permutation: 2  1  4  3
 21. Permutation: 4  2  1  3
 22. Permutation: 2  4  1  3
 23. Permutation: 1  4  2  3
 24. Permutation: 4  1  2  3
pey$
```

11. *eval*-Kommando

Syntax:

```
eval arg1 arg2 ...
```

- expliziter Aufruf des *tcl*-Interpreters
- zusätzliche Interpretation (Substitution)
- Dynamische Konstruktion von Kommando’s

```
pey$ cat prog1.tcl
set cmd { puts "Hallo, Tcl-Fan's" }

eval $cmd ;      # $ vor cmd fuehrt zur Substitution
                 # eval sorgt fuer erneute Bewertung

set text "Nochmal: Hallo, Tcl-Fan's"
set cmd { puts $text } ; # $ ist geschuetzt!!!
eval $cmd ;      # ???

set text "Und Nochmal: Hallo, Tcl-Fan's"
set cmd { puts $text }
unset text ;     #Warum nicht?!
eval $cmd ;      # ???

pey$ tcl -f prog1.tcl
Hallo, Tcl-Fan's
Nochmal: Hallo, Tcl-Fan's
Error: can't read "text": no such variable
    while executing
"puts $text..."
    ("eval" body line 1)
```

```
    invoked from within
"eval $cmd "
    (file "prog1.tcl" line 13)
pey$
```

Lösung: Kommando mit Argumenten in Liste

```
pey$ cat prog2.tcl
set text "Und Nochmal: Hallo, Tcl-Fan's"
set cmd [ list puts $text ] ; # $ nicht geschuetzt!!!
unset text ;    #Warum nicht?!
eval $cmd ;     # !!!

pey$ tcl -f prog2.tcl
Und Nochmal: Hallo, Tcl-Fan's
pey$
```

- In *set cmd [list puts $text]* wird die Variable *text* erst substituiert, danch erzeugt *list* eine Liste aus zwei Argumenten: *puts* und *Hallo, Tcl-Fan's*.

Statt *list* mit *concat*:

```
pey$ cat prog3.tcl

set text "Und Nochmal: Hallo, Tcl-Fan's"
set cmd [ concat puts $text ] ; # $ nicht geschuetzt!!!
unset text ;    #Warum nicht?!
eval $cmd ;     # !!!

pey$ tcl -f prog3.tcl
Error: wrong # args: should be "puts ?-nonewline? ?fileId? string"
    while executing
"puts Und Nochmal: Hallo, Tcl-Fan's"
    ("eval" body line 1)
    invoked from within
"eval $cmd "
    (file "prog3.tcl" line 5)
pey$
```

- *list* erhält die Listenstruktur, *concat* NICHT!

eval mit mehr als einem Argument

```
eval arg1 arg2 arg3
```

- Zunächst werden die Argumente (als Listen betrachtet) via *concat* konkateniert, danach wird bewertet:

Vier (selbsterklärende ???) Varianten:

Variante 1:

```
pey$ cat prog4a.tcl
set file stderr
set text "Hallo, Tcl'Fan's"
set cmd puts
eval $cmd $file $text

pey$ tcl -f prog4a.tcl
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 $text"
    (file "prog4a.tcl" line 4)
pey$
```

Variante 2:

```
pey$ cat prog4b.tcl
set file stderr
set text "Hallo,\ Tcl'Fan's"
set cmd puts
eval $cmd $file $text

pey$ tcl -f prog4b.tcl
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 $text"
    (file "prog4b.tcl" line 4)
pey$
```

Variante 3:

```
pey$ cat prog4c.tcl
set file stderr
set text {Hallo, Tcl'Fan's}
set cmd puts
eval $cmd $file $text

pey$ tcl -f prog4c.tcl
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 $text"
    (file "prog4c.tcl" line 5)
pey$
```

Variante 4:

```
pey$ cat prog4d.tcl
set file stderr
set text {Hallo,\ Tcl'Fan's}
set cmd puts
eval $cmd $file $text

pey$ tcl -f prog4d.tcl
Hallo, Tcl'Fan's
pey$
```

Mehr dazu später (→ wish)

12. *uplevel*-Kommando

Syntax:

```
uplevel ?level? command
```

siehe zu *level* das Kommando *upvar*

- Ausführung des Kommando's *command* auf angegebenem Stack-Level

Beispiel: eine eigene Kontrollstruktur

```
pey$ cat mydo.tcl
#! /usr/local/bin/tcl

proc do { varName first last body } {
        upvar $varName v
        for { set v $first } { $v <= $last } { incr v } {
                uplevel $body
        }
}

do i 4 10 {
        puts [ format "%3d " $i ]
}
puts ""
puts "--------------------"

set vektor {}

do i 1 5 {
        lappend vektor [expr $i * $i ]
        puts [ format "%3d " $i ]
}
puts ""
set vektor

pey$ tcl
```

```
tcl> source mydo.tcl
  4
  5
  6
  7
  8
  9
 10

-------------------
  1
  2
  3
  4
  5

1 4 9 16 25
tcl> exit
pey$
```

- Erstes Argument von *do*: Schleifenvariable, die das durch die nächsten beiden Argumente bestimmte Intervall durchläuft
- Das vierte Argument bestimmt das Kommando, das bei jedem Durchlauf ausgeführt wird
- Die globale Variable *vektor* wird mit der leeren Liste initialisiert. Mittels *upvar* wird die in der *do*-Prozedur existente Variable an die globale Schleifenvariable i gebunden. Für jeden ihrer Werte wird der Schleifenkörper in der Variablen-Umgebung des Aufrufers aufgerufen.

Das Ganze mit *eval* statt mit *uplevel* ???

```
pey$ cat mydo1.tcl
#! /usr/local/bin/tcl

proc do { varName first last body } {
        upvar $varName v
        for { set v $first } { $v <= $last } { incr v } {
                eval $body
        }
}

do i 4 10 {
        puts [ format "%3d " $i ]
}
puts ""
puts "-------------------"

set vektor {}

do i 1 5 {
        lappend vektor [expr $i * $i ]
```

```
        puts [ format "%3d " $i ]
}
puts ""
set vektor

pey$ tcl
tcl> source mydo1.tcl
Error: can't read "i": no such variable
tcl> exit
pey$
```

Noch etwas "gespielt":

```
pey$ cat mydo2.tcl
#! /usr/local/bin/tcl

proc do { varName first last body } {
        upvar $varName v
        for { set v $first } { $v <= $last } { incr v } {
                uplevel $body
        }
        puts "Fertig!"
}

proc something {} {
        set vektor {}

        do i 2 5 {
                lappend vektor [expr $i * $i ]
                if { $i == 4 } { return $i }
        }
        puts "something"
}

puts "-----------------"
puts [ something ]
puts "-----------------"

pey$ mydo2.tcl
-----------------
something

-----------------
pey$ tcl
tcl> source mydo2.tcl
-----------------
something

-----------------
tcl> exit
pey$
```

- Was bewirkt die *return*-Anweisung in der Prozedur *something*? Offenkundig, nicht das, was man von anderen Kontrollstrukturen wie etwa *for* erwartet:

```
pey$ cat mydo3.tcl
#! /usr/local/bin/tcl

proc do { first last } {
        global vektor
        for { set v $first } { $v <= $last } { incr v } {
                lappend vektor [expr $v * $v ]
                if { $v == 4 } { return $v }
                }
        puts "Fertig!"
}

proc something {} {
        set vektor {}

        return [ do  2 5 ]
        puts "something" ; #macht hier wohl keinen Sinn!

}

puts "-----------"
puts [ something ]
puts "-----------"
pey$ mydo3.tcl
-----------
4
-----------
pey$ tcl
tcl> source mydo3.tcl
-----------
4
-----------
tcl> exit
pey$
```

13. *tcl* und UNIX

13.1 Ausführung von UNIX-Kommando's

Syntax:

```
exec unix-cmd
```

- Ausführung des *unix-cmd* als neuer Prozess

Beispiel:

```
pey$ tcl
tcl> set tcl_anz [exec grep tcl ../kap.1 | wc -l ]
tcl> puts $tcl_anz
     28
tcl> exit
pey$ grep tcl ../kap.1 | wc -l
     28
pey$
```

exec - I/O-Umlenkung	
-keepnewline	als erstes Argument, abschließendes *newline* bleibt erhalten
|	Standardausgabe eines Prozesses via Pipeline in Standardeingabe eines anderen Prozesses
|&	Sowohl Standard- als auch Diagnoseausgabe eines Prozesses via Pipeline an Standardeingabe eines anderen Prozesses
< *filename*	Standardeingabe aus Datei *filename*
<@ *fileId*	Eingabe aus durch *fileId* identifizierter I/O-Verbindung
<< *value*	Eingabe von gegebenem Wert *value*
> *filename*	Standardausgabe in Datei *filename* - Überschreibend
2> *filename*	Diagnoseausgabe in Datei *filename* - Überschreibend
>& *filename*	Sowohl Standard- als auch Diagnoseausgabe in Datei *filename* - Überschreibend
>> *filename*	Standardausgabe an Datei *filename* anfügen
2>> *filename*	Diagnoseausgabe an Datei *filename* anfügen
>>& *filename*	Standard- und Diagnoseausgabe an Datei *filename* anfügen
>@ *fileId*	Standardausgabe in durch *fileId* identifizierte I/O-Verbindung umlenken
2>@ *fileId*	Diagnoseausgabe in durch *fileId* identifizierte I/O-Verbindung umlenken
>&@ *fileId*	Diagnoseausgabe in durch *fileId* identifizierte I/O-Verbindung umlenken

- UNIX-Kommando's sind der *tcl*-Shell automatisch bekannt - dies kann wie folgt verhindert werden:
  set auto_noexec anything

13.2 File-System-Zugriffe

Übersicht

file-Kommando's	
file atime name	*access time* als Dezimalstring
file dirname name	Katalog, in dem *name* enthalten
file executable name	1, wenn Datei ausführbar, 0 sonst
file exists name	1, wenn *name* existiert, 0 sonst
file extension name	liefert den Teil von *name* nach dem (letzten) Punkt
file isdirectory name	1, wenn *name* Katalog, 0 sonst
file isfile name	1, wenn *name* weder Katalog, symbolischer Link noch Geräte-Datei, 0 sonst
file lstat name var	Statusinformationen über Link *name* in *var* ablegen
file mtime name	Liefert Modifikationszeitstempel von *name* zurück
file owned name	1, falls der aktuelle Benutzer die Datei *name* besitzt, 0 sonst
file readable name	1, falls Leserecht gesetzt, 0 sonst
file readlink name	Liefert den Inhalt des symbolischen Links *name*
file rootname name	Liefert alles bis zur letzten Extension von *name* (bis zum letzten ".")
file size name	Größe von *name* (number of bytes)
file stat name var	liefert Status-Informationen der Datei *name* in Array *var*. Definierte Indizes sind *atime, ctime,dev,gid,ino,mode,mtime,nlink,size,type* und *uid*
file tail name	Liefert alle Zeichen nach dem letzten Schrägstrich ("/")
file type name	Liefert Dateityp: *file, directory, characterSpecial, blockSpecial, fifo, link* oder *socket*
file writable name	1, wenn Schreibrecht gesetzt, 0 sonst (Achtung: Schreibweise von *writ(e)able* !!!)

- Empfehlung: **man stat** (oder lstat oder fstat)
- Offen sind: *stdin, stdout, stderr*

Beispiele:

```
pey$ pwd
/home/swg/tclSkr/ex7
pey$ ls -l
total 16
-rw-r--r--   1 swg      users         316 Feb 19 16:14 prog1.tcl
-rw-r--r--   1 swg      users         142 Feb 19 16:20 prog2.tcl
-rw-r--r--   1 swg      users         408 Feb 19 16:26 se.3
-rw-r--r--   1 swg      users         338 Feb 19 16:46 se.4a
-rw-r--r--   1 swg      users         339 Feb 19 16:47 se.4b
pey$ tcl
tcl> set pfad [ exec pwd ]
tcl> puts $pfad
/home/swg/tclSkr/ex7

tcl> set datei se.4a
tcl> puts $datei
se.4a
tcl> set datei1 $pfad
tcl> append datei1 /se.4a
/home/swg/tclSkr/ex7/se.4a
tcl> puts $datei1
/home/swg/tclSkr/ex7/se.4a

tcl> file dirname $datei1
/home/swg/tclSkr/ex7
tcl> file tail $datei1
se.4a
tcl> file exists $datei
1
tcl> file extension $datei1
.4a
tcl> file owned $datei
1
tcl> file readable $datei
1
tcl> file rootname $datei1
/home/swg/tclSkr/ex7/se
tcl> file size $datei
338
tcl> file stat $datei datarray
tcl> foreach ind [array names datarray] {
=>          puts $datarray($ind)
=> }
824748368
824752571
100
1
33188
```

```
file
824748368
500
31798
338
770
tcl> file tail $datei1
se.4a
tcl> file type $datei1
file
tcl> file writable $pfad
1
tcl> exit
pey$
```

Zwei Pfadnamen für dieselbe Datei?

```
pey$ ln -s se.4a new.se.4a
pey$ ls -l
total 19
-rwxr-xr-x   1 swg      users         182 Feb 20 19:01 fileeq.tcl*
lrwxrwxrwx   1 swg      users           5 Feb 20 19:01 new.se.4a -> se.4a
-rw-r--r--   1 swg      users         338 Feb 19 16:46 se.4a
-rw-r--r--   1 swg      users         339 Feb 19 16:47 se.4b
pey$ cat fileeq.tcl
proc fileeq { path1 path2 } {
        file stat $path1 stat1
        file stat $path2 stat2
        expr  $stat1(ino) ==  $stat2(ino) && \
                $stat1(dev) == $stat2(dev)
}

puts [fileeq se.4a new.se.4a]

pey$ tcl -f fileeq.tcl
1
pey$
```

File-I/O, Pipe-I/O

File - I/O	
open what ?access? ?permissions?	Liefert ID für I/O-Verbindung zu *pipeline oder Datei* *access: s.u.* **/ man open**
puts ?-nonewline? ?stream? string	Schreiben von String auf I/O-Verbindung *stream*; *nonewline*: kein *newline* angefügt
gets stream ?varname?	Eine Zeile lesen
read stream ?numBytes?	Liest *numbytes* Bytes, oder ganze Datei
read -nonewline stream	Liest alle Bytes, letztes *newline* wird entfernt
tell stream	Liefert Datei-*offset*
seek stream offset ?whence?	Positionieren; *whence*: *start, current, end*
eof stream	1, wenn EOF, 0 sonst
flush stream	Puffer leeren
close stream	Verbindung schließen

access **Argumente für** *open*	
r	Öffnen zum Lesen - Datei muß existieren
r+	Öffnen zum Lesen und Schreiben - Datei muß existieren
w	Öffnen zum Schreiben - falls Datei existiert, wird sie auf Länge 0 gesetzt - falls sie nicht existiert, erzeugen
w	Öffnen zum Schreiben und Lesen - falls Datei existiert, wird sie auf Länge 0 gesetzt - falls sie nicht existiert, erzeugen
a	Öffnen zum Schreiben - Datei muß existieren - Daten werden angefügt
a+	Öffnen zum Schreiben und Lesen - Datei muß existieren - Daten werden angefügt

access **Argumente für** *open* **- POSIX-Standard**	
RDONLY	Öffnen zum Lesen
WRONLY	Öffnen zum Schreiben
RDWR	Öffnen zum Lesen und Schreiben
APPEND	Öffnen zum Anfügen
CREAT	Erzeugen, falls Datei nicht existiert
EXCL	Wenn zusammen mit *CREAT*, darf Datei noch nicht existieren
NONBLOCK	Kein Blockieren während des *open* → man open
TRUNC	Falls Datei existiert, auf Länge 0 verkürzen

Diese *POSIX-Flags* können auch verknüpft werden:

```
set fileId [ open /tmp/temp { RDWR CREAT } ]
```

- Öffnen zum Lesen und Schreiben (RDWR), erzeugen falls noch nicht existiert (CREAT)

Pipe-I/O:

```
pey$ cat pipe1.tcl
set input [ open "| sort phones " r ]
set contents [split [ read $input ] \n ]
for { set i 0 } { $i <= [ llength $contents ] } \
        { incr i } {
        puts [lindex $contents $i]
}
close $input

pey$ cat phones
Ralf : 5857
Balser : 3558
Beschorner : 3552
Borchert : 3572
Boremski : 3566
Etter : 3601
Fiedler : 3507
Gessner : 3555
Grabert : 3573
Hellwig : 3540
Kratz : 3559
Maier : 3510
Mueller : 3571
Pittelkow : 3502
Rieder : 3525
Schweiggert : 3570
Stanglow : 3574
pey$ tcl -f pipe1.tcl
```

```
Balser : 3558
Beschorner : 3552
Borchert : 3572
Boremski : 3566
Etter : 3601
Fiedler : 3507
Gessner : 3555
Grabert : 3573
Hellwig : 3540
Kratz : 3559
Maier : 3510
Mueller : 3571
Pittelkow : 3502
Ralf : 5857
Rieder : 3525
Schweiggert : 3570
Stanglow : 3574

pey$
```

- Vorsicht bei gleichzeitigem Lesen und Schreiben aus / in *pipeline* (→ *flush*)

gets-Kommando

- Nur ein Argument (ID der Dateiverbindung): gelesen und zurückgeliefert wird eine Zeile - das abschließende *newline* wird entfernt - EOF: leerer String wird geliefert; "leere Zeile" liefert dasselbe → *eof*
- Zwei Argumente: das erste wie oben, das zweite nimmt die gelesene Zeile ohne *newline* auf - Rückgabewert ist die Zahl der gelesenen Bytes (ohne *newline*) - EOF: Rückgabewert -1

read-**Kommando**

- entweder mit der Option *-nenewline* oder mit der Angabe *numBytes*
- *numBytes* nicht angegeben: ganze Datei (oder Rest) lesen
- *nonewline*: abschließendes *newline* der Eingabe wird nicht zurückgeliefert

Ein kleines, sehr einfaches Dialog-Programm:

```
#! /usr/local/bin/tcl
set HOMEDIR $env(HOME)
set PHONEBOOK "$HOMEDIR/sai/skripte/udb1/tcl/db/phones"

proc menu {} {
        exec /usr/bin/clear
        puts [format "\t\t%s\n" "(1) Look for somebody"]
        puts [format "\t\t%s\n" "(2) Add somebody"]
        puts [format "\t\t%s\n" "(3) Delete somebody"]
        puts [format "\t\t%s\n" "(4) List all"]
        puts [format "\t\t%s\n" "(5) Quit"]
        puts -nonewline "Your choice 1/2/3/4/5: "
        set choice [ gets stdin]
        return $choice
}

exec clear

if ![ file isfile $PHONEBOOK ] {
        puts "$PHONEBOOK doesn't exist"
        exit 1
}
set f [ open $PHONEBOOK RDONLY ]
set contents [split [ read $f ] \n ]
close $f

proc listall { { f stdout } } {
        global contents
        for { set i 0 } { $i <= [ llength $contents ] } \
        { incr i } {
        puts $f [lindex $contents $i]
        }
}


proc look { name } {
        global contents
        set ix [lsearch -regexp $contents $name ]
        if { $ix >= 0 } {
                puts [lindex $contents $ix]
        } else {
                puts "$name doesn't exist!"
        }
}

proc delete { list name } {

        set ix [lsearch -regexp $list $name ]
        if { $ix >= 0 } {
                puts "$name is deleted"
```

```
                return [ lreplace $list $ix $ix ]
        } else {
                puts "$name doesn't exist!"
                return $list
        }
}

set choice [menu]
while { $choice >= 1 && $choice <= 4 } {
switch -exact -- $choice {
        1       { puts -nonewline "Enter name: "
                  set name [ gets stdin ]
                  look $name
                }
        2       { puts -nonewline "Enter name: "
                  set name [ gets stdin ]
                  puts -nonewline "Enter phonenumber: "
                  set phone [ gets stdin ]
                  set contents [ linsert $contents 0 "$name : $phone" ]
                }
        3       { puts -nonewline "Enter name: "
                  set name [ gets stdin ]
                  set contents [ delete $contents $name ]
                }
        4       { listall }
        5       {puts "exit"; exit 0 }
        default {puts "Wrong choice"}
}
puts "press any key"
gets stdin
set choice [menu]
}

set f [ open "| egrep {^\[A-Za-z\]+} |  sort > $PHONEBOOK" w  ]
# mit egrep werden Leerzeilen entfernt

listall $f
close $f
```

13.3 Weitere *tcl*-Kommando's

- Datei-Namen-Substitution

	glob - Liefert Liste von Dateinamen
Syntax:	*glob ?flags? pattern ?pattern ...?*
flags:	
-nocomplain	*glob* liefert leere Liste, falls kein Treffer. Wenn nicht angegeben, wird Fehler gesetzt
—	zu benutzen, falls *pattern* mit einem – beginnt
pattern:	
*	beliebige Zeichenfolge
?	beliebiges Einzelzeichen
[*chars* **]**	Aufzählung von Zeichen
{ *a,b, ...* **}**	Irgendeiner der Strings
\c	Zeichen x
~/	Heimatkatalog des Benutzers (*HOME*)
~*user*	Heimatkatalog von *user*

Beispiele:

```
pey$ ls
dial.tcl*    phones       prog4a.tcl   se.2         se.4d
fileeq.tcl*  pipe1.tcl    prog4b.tcl   se.3         se.5
glob.man     prog1.tcl    prog4c.tcl   se.4a        se.6
new.se.4a@   prog2.tcl    prog4d.tcl   se.4b        se.7
packButton*  prog3.tcl    se.1         se.4c        se.8
pey$ tcl
tcl> exec ls * ; #kann ja nicht gehen
Error: ls: *: No such file or directory
tcl> glob * ; #vergleiche mit Ausgabe von ls
prog1.tcl prog2.tcl se.1 prog3.tcl se.2 prog4b.tcl se.3 prog4d.tcl
prog4a.tcl prog4c.tcl new.se.4a se.4a se.4b se.4c packButton se.4d
se.5 se.6 fileeq.tcl pipe1.tcl se.7 phones dial.tcl se.8 glob.man
tcl> glob se*
se.1 se.2 se.3 se.4a se.4b se.4c se.4d se.5 se.6 se.7 se.8
tcl> glob se.?
se.1 se.2 se.3 se.5 se.6 se.7 se.8
tcl> glob [s-t]* ; #wird wohl nicht gehen?
Error: invalid command name "s-t"
tcl> glob \[s-t\]* ; #so wohl doch!
se.1 se.2 se.3 se.4a se.4b se.4c se.4d se.5 se.6 se.7 se.8
tcl> glob {se.2, xy, se.3} ; #???
Error: no files matched glob pattern "se.2, xy, se.3"
tcl> glob \{se.2, xy, se.3\} ; #Leerraum ???
Error: unmatched open-brace in file name
tcl> glob \{se.2\}
se.2
tcl> glob \{se.2,xy,se.3\}
se.2 se.3
tcl> glob \{se.2,xy, se.3\}
```

```
Error: unmatched open-brace in file name ; #Leerraum - s.o.
tcl> glob \{se.2,\ xy,\ se.3\}
se.2
tcl> glob ~/
/home/swg/
tcl> glob ~swg
/home/swg
tcl> glob ~swg/*
/home/swg/dip /home/swg/unixDb /home/swg/soft /home/swg/tclSkr
/home/swg/tmp /home/swg/tqm /home/swg/lit
tcl> exit
pey$
```

- Katalog wechseln: *cd*
- Aktuellen Katalog bestimmen: *pwd*

Beispiel:

```
pey$ tcl
tcl> pwd
/home/swg/tclSkr/ex7
tcl> cd ..
tcl> pwd
/home/swg/tclSkr
tcl> cd ./ex7
tcl> pwd
/home/swg/tclSkr/ex7
tcl> exit
pey$
```

- Terminieren und Exit-Status setzen: *exit ?integer?*
- Prozeß-ID bestimmen: *pid*

Beispiel:

```
pey$ tcl
tcl>set TMPFILE /tmp/tst[pid]
tcl>puts $TMPFILE
/tmp/tst277
tcl>exit
pey$
```

14. Fehler- / Ausnahmebehandlung

14.1 Übersicht

Wenn in *tcl* eine Fehlerbedingung auftritt, wird die Ausführung des aktuellen Kommando's abgebrochen. Ist das Kommando Teil eines Skriptes, so wird auch dieses beendigt. Tritt der Fehler während der Ausführung einer Prozedur auf, so wird diese abgebrochen, genauso ihr Aufrufer und die "darüber stehenden" Aufrufer. Die betreffende Fehlermeldung wird ebenso nach oben durchgereicht. Bei verschachtelten Kommandos ist es daher oft schwer, zu lokalisieren wo der Fehler denn nun genau aufgetreten ist. Dies erleichtert der *tcl*-Interpreter mit der globalen Variablen *errorInfo*. Diese enthält einen sogenannten "Stack-Trace"; darin sind alle zur Zeit des Fehlers aktiven (verschachtelten) Kommandos aufgeführt. Fehler zählen zu der Gruppe der *exceptions* (Ausnahmen). Zu den Exceptions gehören auch *break, continue* und *return*. Exceptions veranlassen Kommando's / Skripte zum Abbruch.

Weitere Information über den Fehler kann der globalen Variablen *errorCode* entnommen werden. Diese enthält eine Liste: das erste Element beschreibt eine allgemeine Fehlerklasse, die weiteren Elemente geben noch mehr Informationen zum speziellen Fehler.

Fehlerbehandlung
catch command ?resultVar? Zum Abfangen von Fehlern während der Ausführung von *command.* In der Variablen *resultVar* wird das Ergebnis von *command* zurückgeliefert oder eine Fehlermeldung *catch* selbst liefert 0, wenn *command* fehlerfrei ablief, 1 wenn Fehler. Termination bei Fehler in der Ausführung von *command* wird so verhindert
error message ?info? ?code? *erzeugt Fehlerbedingung, die zur Termination führt (falls nicht durch catch abgefangen* *message: Fehlertext* *info: Initialisierung der globalen errorInfo-Variablen* *code: Initialisierung der globalen errorCode-Variablen*
return ?-code c? ?-errorinfo i? ?-errorcode ec? string Zur (vorzeitigen) Beendigung einer Prozedur oder zur Rückgabe eines konstanten Wertes aus einer Prozedur *-code c*: c $\in$ { *ok, error, return, break, continue,*<eine integer> } default: *ok* *-code error*: $\rightarrow$ *error*-Kommando *-errorcode*: setzt die globale *errorCode*-Variable *-errorinfo*: setzte die globale *errorInfo*-Variable

14.2 *catch*-Kommando

- Zur Vermeidung des Abbruchs von Kommando-Interpretation (Ausführung von *tcl*-Skripten bei Auftreten von Fehlern
- Ruft den *tcl*-Interpreter rekursiv zur Ausführung des als ersten Arguments angegebenen Kommando’s (Skript’s)
- Liefert immer einen *TCL_XX*-Code - der Rückgabewert des *tcl*-Interpreters nach Abarbeitung des Kommando’s (Skript’s)

- Kein Fehler: **0**, sonst kleine ganze Zahl → */usr/include/tcl.h* (TCL_OK: 0, TCL_ERROR: 1, TCL_RETURN: 2, TCL_BREAK: 3, TCL_CONTINUE: 4)

Beispiele:

```
pey$ tcl
tcl> # zuruecksetzen einer Variablen, falls existent
tcl> set x 3
tcl> set x
3
tcl> unset x
tcl> set x
Error: can't read "x": no such variable
tcl> unset y
Error: can't unset "y": no such variable
tcl> catch "unset y" err
1
tcl> set err
can't unset "y": no such variable
tcl> set y 3
tcl> catch "unset y" err
0
tcl> set err
tcl> exit
pey$
```

bei *return*

```
pey$ tcl
tcl> catch { return "alles okay" } str
2
tcl> set str
alles okay
tcl> exit
pey$
```

tcl-exceptions		
Rückgabewert von *catch*	Beschreibung	abgefangen von
TCL_OK: 0	Normaler Return. String beinhaltet Rückgabewert.	Nicht abfangbar
TCL_ERROR: 1	Fehler. String enthält Fehlermeldung.	*catch*
TCL_RETURN: 2	*return*-Kommando. String enthält Rückgabewert von *procedure* oder *source*-Kommando.	*catch, source*, Prozeduren
TCL_BREAK: 3	*break*-Kommando. String ist leer.	*catch, for, foreach, while*, Prozeduren
TCL_CONTINUE: 4	*continue*-Kommando. String ist leer	*catch, for, foreach, while*, Prozeduren
anderer	Von Benutzer oder Programm definiert	*catch*

Weitere Beispiele:

```
pey$ cat err1.tcl
#! /usr/local/bin/tcl

set list { 1 2 3 4 5 6 }

set sum 0
foreach el $list {
        set sum [ expr $sum + $element ]
}
puts "Summe: $sum"

pey$ err1.tcl
Error: can't read "element": no such variable
    while executing
"expr $sum + $element..."
    invoked from within
"set sum [ expr $sum + $element..."
    ("foreach" body line 2)
    invoked from within
"foreach el $list {
        set sum [ expr $sum + $element ]
}"
    (file "./err1.tcl" line 6)
pey$
```

- Der Fehler ist klar: Variable *element* müsste *el* heißen
- Fehler tritt auf beim Aufruf von *expr*: *tcl*-Parser versucht die Substitution durchzuführen und liefert den Fehler, der stufenweise zurückgereicht wird und sukzessive die Kommandoaufrufe bis hin zum gesamten Skript terminiert.

Also:

```
pey$ cat err2.tcl
#! /usr/local/bin/tcl

set list { 1 2 3 4 5 6 }

set sum 0
catch { foreach el $list {
          set sum [ expr $sum + $element ]
         } } msg
puts [ format "von 'catch' geliefert:\n %s"  $msg ]
puts [ format "\nVariable 'errorInfo':\n\n %s" $errorInfo ]

pey$ err2.tcl
von 'catch' geliefert:
 can't read "element": no such variable

Variable 'errorInfo':

 can't read "element": no such variable
    while executing
"expr $sum + $element..."
    invoked from within
"set sum [ expr $sum + $element..."
    ("foreach" body line 2)
    invoked from within
"foreach el $list {
          set sum [ expr $sum + $element ]
         } "
pey$
```

14.3 *error*-Kommando

- Fehler generieren

```
pey$ cat err3.tcl
#! /usr/local/bin/tcl

if { $argc <= 2 } {
        error "$argc Argumente sind zuwenig"
} else {
        puts "alles okay"
}
```

```
pey$ err3.tcl  1 2 3 4
alles okay

pey$ err3.tcl a b
Error: 2 Argumente sind zuwenig
    while executing
"error "$argc Argumente sind zuwenig""
    invoked from within
"if { $argc <= 2 } {
        error "$argc Argumente sind zuwenig"
} else {
        puts "alles okay"
}"
    (file "./err3.tcl" line 3)
pey$ exit
```

```
pey$ cat err3.tcl
#! /usr/local/bin/tcl

if { $argc <= 2 } {
        error "$argc Argumente sind zuwenig"
} else {
        puts "alles okay"
}

pey$ err3.tcl  1 2 3 4
alles okay

pey$ err3.tcl a b
Error: 2 Argumente sind zuwenig
    while executing
"error "$argc Argumente sind zuwenig""
    invoked from within
"if { $argc <= 2 } {
        error "$argc Argumente sind zuwenig"
} else {
        puts "alles okay"
}"
    (file "./err3.tcl" line 3)
pey$ exit
```

14.4 *return*-Kommando

- Vorzeitige Beendigung einer Prozedur
- Definition eines Rückgabewertes einer Prozedur (*default: Rückgabewert des zuletzt ausgeführten Kommandos)*
- Rückkehr in Ausnahmen:

```
pey$ tcl
tcl> cat err4.tcl
proc go_back { } {
        return -code error -errorinfo "Mag nicht mehr" ret_value
}

catch { go_back } result

tcl> source err4.tcl
1
tcl> set result
ret_value
tcl> set errorInfo
Mag nicht mehr
    invoked from within
"go_back "
tcl> exit
pey$
```

- *-code error*: ähnlich wie Aufruf von *error*-Kommando
- -errorinfo "text" initialisiert die globale Variable *errorInfo* mit "text" (kein Stacktrace - s.o.)

```
pey$ tcl
tcl> cat err5.tcl
proc my_break { } {
        return -code break
}

set i 1
while { $i < 5 } {
        if { $i == 3 } {
                my_break
        }
        puts $i
        incr i
}
tcl> source err5.tcl
1
2
tcl> exit
pey$
```

```
pey$ tcl
tcl> cat err6.tcl
proc my_continue { } {
        return -code continue
}

set i 0
while { $i < 5 } {
        incr i
        if { $i == 3 } {
                my_continue
        }
        puts $i
}
tcl> source err6.tcl
1
2
4
5
tcl> exit
pey$
```

- *default*-Wert: *ok* (*-code* nicht angegeben)

Beispiel: unsere eigene Kontrollstruktur *do*

```
pey$ cat mydoerr.tcl
# modifizierte Version der do-Schleife

proc do { varName first last body } {
        global errorInfo errorCode
        upvar $varName v
        for { set v $first } { $v <= $last } { incr v } {
                set code [ catch {uplevel $body } string ]
                switch -- $code {
                  1 { return -code error -errorinfo $errorInfo \
                          -errorcode $errorCode $string
                    }
                  2 { return -code return $string }
                  3 { return }
                  default { return -code $code $string }
                }

        }
}

pey$
```

- Ausnahmen werden hier genauso behandelt wie bei den Builtin-Schleifenkommando's. Der Schleifenkörper wird innerhalb eines *catch*-Kommandos bearbeitet. Tritt dort keine Ausnahme auf (*code* ist 0) oder die Ausnahme ist ein *continue* (*code* ist 4), fährt *do* mit der nächsten Iteration fort. Wenn ein Fehler (*code* ist 1) oder ein *return* (*code* ist 2) anliegt, wird diese Ausnahme mit dem *return*-Kommando zum Aufrufer weitergeleitet. Ist die Ausnahme ein *break* (*code* ist 3), kehrt *do* auf normale Weise zum Aufrufer zurück.
- Alle anderen Codes werden einfach an den Aufrufer weitergeleitet. Nach der Beendigung des *catch*-Befehles sind die Variablen *errorInfo* und *errorCode* mit der Fehlerbeschreibung belegt. Da beim *return*-Kommando die Optionen *-errorinfo* und *-errorcode* gewählt wurden, werden diese Variablen auch nach oben durchgereicht, das heißt, der Stack-Trace wird mit der ursprünglichen Fehlermeldung initialisiert. Ohne die Optionen würde der Stack-Trace erst mit der Fehlermeldung der *do*-Prozedur beginnen, die genaue Fehlerstelle wäre nicht sofort erkennbar.

15. Fehlersuche

15.1 *info*-Kommando

info-Kommando	
info args procName	Liefert Liste der Argumentnamen der Prozedur *procName*
info body procName	Liefert die Kommando's im Rumpf der Prozedur *procName*
info cmdcount	Liefert die Anzahl der im aktuellen *tcl*-Interpreter ausgeführten Kommandos
info commands ?pattern?	Liefert Liste aller Kommandos des aktuellen Interpreters (Builtin-Kommandos, applikationsdefinierte Kommandos und Prozeduren). Wenn ein Muster angegeben ist, werden nur die Kommandos ausgegeben, welche dem Muster genügen. (die gleichen Regeln wie bei *string match*).
info complete string	Liefert 1, wenn *string* ein vollständiges *tcl*-Kommando enthält
info default procName argName varName	Prüft, ob das Argument *argName* in der Prozedur *procName* einen Defaultwert besitzt. Wenn ja, wird dieser Wert in *varName* abgelegt und *1* zurückgegeben. Ansonsten wird *0* zurückgegeben, *varName* wird nicht verändert.
info exists varName	Falls eine Variable *varName* in der gegenwärtigen Umgebung existiert, wird *1* zurückgegeben, ansonsten *0*.
info globals ?pattern?	Liefert Liste aller gegenwärtig definierten globalen Variablen. Ist *pattern* angegeben, werden nur die Variablen ausgegeben, die dem Muster genügen.

info-Kommando	
info level ?number?	Falls *number* nicht angegeben ist, wird die Nummer des gegenwärtigen *Stacklevels* zurückgegeben. Ist *number* angegeben, wird eine Liste der Namen und Argumente vom Prozeduraufruf in Level *number* zurückgegeben.
info library	Gibt den vollen Pfadnamen des Kataloges zurück, in dem die Standard-*tcl*-Bibliothek liegt.
info locals ?pattern?	Liefert Liste der lokalen Variablen der gegenwärtigen Prozedur oder einen Leerstring, wenn sich der Aufruf in keiner Prozedur befindet. Ist ein Muster *pattern* angegeben, enthält die Liste nur die Variablennamen, welche *pattern* genügen.
info procs ?pattern?	Liefert Liste aller gegenwärtig definierten Prozeduren. Ist *pattern* angegeben, werden nur die Prozedurnamen ausgegeben, die *pattern* genügen.
info script	Wenn gerade ein Skriptfile ausgewertet wird, wird dessen Name zurückgegeben, andernfalls ein Leerstring.
info tclversion	Liefert Versionsnummer des Tcl-Interpreters in der Form *major.minor* . major und minor sind dezimale Strings. minor erhöht sich, wenn die Änderungen zur vorherigen Version kompatibel sind, ist die neue Version inkompatibel zu ihren Vorgängern, erhöht sich major (→ **tcl.h**).
info vars ?pattern?	Eine Liste aller gegenwärtig bekannten (sichtbaren) Variablen wird zurückgegeben. Ist *pattern* angegeben, werden nur die Variablen berücksichtigt, welche dem Muster genügen.

- sichtbare Variable in einer Prozedur: lokale Variable + via *global* und *uplevel* sichtbar gemachte Variable

Beispiel: *info level*

```
pey$ cat perm.tcl
#! /usr/local/bin/tcl

proc printFeld { } {
  global feld
  global anzahl
# feld: Feld der zu auszugebenden Objekte [1 .. anzahl]
# anzahl : Anzahl der Elemente, also anzahl oberster Index
  for {set i 1 } { $i <= $anzahl } { incr i } {
  puts -nonewline "$feld($i)  "
  }
  puts " "
  puts "Prozedur [info procs printFeld ] Level: [info level ]"
}


proc tausche { x y } {
  upvar $x a
  upvar $y b
  puts "Prozedur [info procs tausche ] Level: [info level ]"
  set tmp $a
  set a $b
  set b $tmp
}

proc perm { n nr_perm } {
  global feld
  upvar $nr_perm nr

  puts "Prozedur [info procs perm ] Level: [info level ]"
  if { $nr > 6 } {
    # das reicht !
    return
  }
  if { $n == 1 } {
    incr nr
    puts -nonewline [format "%3d. Permutation: " $nr ]
    printFeld
  } else {
    perm [expr $n - 1 ] nr
    for { set i 1 } { $i <= [expr $n - 1] } \
      { incr i } {

      tausche feld($i) feld($n)
      perm [expr $n - 1] nr
      tausche feld($i) feld($n)
    }
  }
}
```

```
set anzahl  4

for { set i 1 } { $i <= $anzahl } { incr i } {
  set feld($i) $i
}

set nr 0
puts "global level: [ info level ]"
perm $anzahl nr

puts "Prozedur: [ info procs perm ]"
puts "Argumente: [ info args perm ]"
puts "Rumpf: [ info body perm ]"

pey$ perm.tcl
global level: 0
Prozedur perm Level: 1
Prozedur perm Level: 2
Prozedur perm Level: 3
Prozedur perm Level: 4
  1. Permutation: 1  2  3  4
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur perm Level: 4
  2. Permutation: 2  1  3  4
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur tausche Level: 3
Prozedur perm Level: 3
Prozedur perm Level: 4
  3. Permutation: 3  2  1  4
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur perm Level: 4
  4. Permutation: 2  3  1  4
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur tausche Level: 3
Prozedur tausche Level: 3
Prozedur perm Level: 3
Prozedur perm Level: 4
  5. Permutation: 1  3  2  4
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur perm Level: 4
  6. Permutation: 3  1  2  4
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur tausche Level: 3
Prozedur tausche Level: 2
```

```
Prozedur perm Level: 2
Prozedur perm Level: 3
Prozedur perm Level: 4
  7. Permutation: 4  2  3  1
Prozedur printFeld Level: 5
Prozedur tausche Level: 4
Prozedur perm Level: 4
Prozedur tausche Level: 4
Prozedur tausche Level: 3
Prozedur perm Level: 3
Prozedur tausche Level: 3
Prozedur tausche Level: 3
Prozedur perm Level: 3
Prozedur tausche Level: 3
Prozedur tausche Level: 2
Prozedur tausche Level: 2
Prozedur perm Level: 2
Prozedur tausche Level: 2
Prozedur tausche Level: 2
Prozedur perm Level: 2
Prozedur tausche Level: 2
Prozedur: perm
Argumente: n nr_perm
Rumpf:
  global feld
  upvar $nr_perm nr

  puts "Prozedur [info procs perm ] Level: [info level ]"
  if { $nr > 6 } {
    # das reicht !
    return
  }
  if { $n == 1 } {
    incr nr
    puts -nonewline [format "%3d. Permutation: " $nr ]
    printFeld
  } else {
    perm [expr $n - 1 ] nr
    for { set i 1 } { $i <= [expr $n - 1] }  { incr i } {

      tausche feld($i) feld($n)
      perm [expr $n - 1] nr
      tausche feld($i) feld($n)
    }
  }

pey$
```

Mit etwas "Modularisierung":

```
pey$ cat printFeld.tcl
proc printFeld { } {
  global feld
  global anzahl
# feld: Feld der zu auszugebenden Objekte [1 .. anzahl]
# anzahl : Anzahl der Elemente, also anzahl oberster Index
  for {set i 1 } { $i <= $anzahl } { incr i } {
  puts -nonewline "$feld($i)  "
  }
  puts " "
  puts "Prozedur [info procs printFeld ] Level: [info level ]"
  puts "Source: [info script ]"
  puts " "
}

pey$ cat tausche.tcl
proc tausche { x y } {
  upvar $x a
  upvar $y b
  puts "Prozedur [info procs tausche ] Level: [info level ]"
  puts "Source: [info script ]"
  puts "tausche: I was called from '[info level -1 ]'"
  puts " "
  set tmp $a
  set a $b
  set b $tmp
}

pey$ cat permut.tcl
proc perm { n nr_perm } {
  global feld
  upvar $nr_perm nr

  puts "Prozedur [info procs perm ] Level: [info level ]"
  puts "Source: [info script ]"
  puts " "
  if { $nr > 0 } {
    # das reicht !
    return
  }
  if { $n == 1 } {
    incr nr
    puts -nonewline [format "%3d. Permutation: " $nr ]
    printFeld
  } else {
    perm [expr $n - 1 ] nr
    for { set i 1 } { $i <= [expr $n - 1] } \
      { incr i } {
```

```
      tausche feld($i) feld($n)
      perm [expr $n - 1] nr
      tausche feld($i) feld($n)
    }
  }
}

pey$ cat perm1.tcl
#! /usr/local/bin/tcl

source printFeld.tcl
source tausche.tcl
source permut.tcl

set anzahl  4

for { set i 1 } { $i <= $anzahl } { incr i } {
  set feld($i) $i
}

set nr 0
puts "global level: [ info level ]"
perm $anzahl nr

pey$ perm1.tcl
global level: 0
Prozedur perm Level: 1
Source: ./perm1.tcl

Prozedur perm Level: 2
Source: ./perm1.tcl

Prozedur perm Level: 3
Source: ./perm1.tcl

Prozedur perm Level: 4
Source: ./perm1.tcl

  1. Permutation: 1  2  3  4
Prozedur printFeld Level: 5
Source: ./perm1.tcl

Prozedur tausche Level: 4
Source: ./perm1.tcl
tausche: I was called from 'perm 2 nr'

Prozedur perm Level: 4
Source: ./perm1.tcl

Prozedur tausche Level: 4
```

```
Source: ./perm1.tcl
tausche: I was called from 'perm 2 nr'

Prozedur tausche Level: 3
Source: ./perm1.tcl
tausche: I was called from 'perm 3 nr'

Prozedur perm Level: 3
Source: ./perm1.tcl

Prozedur tausche Level: 3
Source: ./perm1.tcl
tausche: I was called from 'perm 3 nr'

Prozedur tausche Level: 3
Source: ./perm1.tcl
tausche: I was called from 'perm 3 nr'

Prozedur perm Level: 3
Source: ./perm1.tcl

Prozedur tausche Level: 3
Source: ./perm1.tcl
tausche: I was called from 'perm 3 nr'

Prozedur tausche Level: 2
Source: ./perm1.tcl
tausche: I was called from 'perm 4 nr'

Prozedur perm Level: 2
Source: ./perm1.tcl

Prozedur tausche Level: 2
Source: ./perm1.tcl
tausche: I was called from 'perm 4 nr'

Prozedur tausche Level: 2
Source: ./perm1.tcl
tausche: I was called from 'perm 4 nr'

Prozedur perm Level: 2
Source: ./perm1.tcl

Prozedur tausche Level: 2
Source: ./perm1.tcl
tausche: I was called from 'perm 4 nr'

Prozedur tausche Level: 2
Source: ./perm1.tcl
tausche: I was called from 'perm 4 nr'
```

```
Prozedur perm Level: 2
Source: ./perm1.tcl

Prozedur tausche Level: 2
Source: ./perm1.tcl
tausche: I was called from 'perm 4 nr'

pey$
```

15.2 *trace*-Kommando

trace-**Kommando**
trace variable name ops command Das durch *command* bezeichnete Kommando wird ausgeführt, sobald auf die Variable *name* in der durch *ops* beschriebenen Form zugegriffen wird. Für *ops* können die Zeichen **r** (*read*), **w** (*write*) oder **u** (*unset*), bzw. Kombinationen daraus angegeben werden. *name* kann sein Name einer skalaren Variablen, eines Array-Elements oder eines ganzen Array's (dann wird *command* bei jedem Elementzugriff aufgerufen). *command* wird aufgerufen als *command name1 name2 op* *name1* = *name* *name* Skalare Variable → *name2* leerer String *name* Array-Element → *name2* Index *name* ganzes Array → i.P. *name2* leerer String *op* → *ops* Ausführungskontext: → **man** Rückgabewert ist im Normalfall ein Leerstring.
trace vdelete name ops command Falls ein *trace* (s.o.) auf die Variable *name* mit den Parametern *ops* und *command* besteht, wird dieser entfernt. Rückgabewert ist ein Leerstring.
trace vinfo name Eine Liste wird ausgegeben, deren Elemente die gegenwärtig für die Variable *name* aktiven Traces enthält. Jedes Listenelement ist eine Unterliste mit den zwei Elementen *ops* und *command*.
→ **man trace**

Beispiel:

```
pey$ cat trace.tcl
#! /usr/local/bin/tcl

trace variable x w printVar
trace variable x uw printMsg
trace variable y(3) w printVar
trace variable y u printMsg
proc printVar { name element op } {
   if { $element != "" } {
      # it's an array
      set name ${name}($element)
   }
   upvar $name z ; # make value accessible with local z
   puts  "Variable $name gesetzt auf $z"
}
proc printMsg { name element op } {
   upvar $name z
   if { [string compare $op "u" ] == 0 } {
      set operation unset
   } else {
      set operation write
   }
   if { $element != "" } {
      puts  "'$operation' auf Element $element von $name"
   } else {
        puts  "'$operation' auf Variable $name"
   }
}

set length 3
for { set i 0 } { $i <= 5 } { incr i } {
   set y($i) [expr $i * $i ]
   puts $y($i)
}

puts [ set x 111 ]
set y($length) 0
unset y(0)
unset x

pey$ trace.tcl
0
1
4
Variable y(3) gesetzt auf 9
9
16
25
'write' auf Variable x
```

```
Variable x gesetzt auf 111
111
Variable y(3) gesetzt auf 0
'unset' auf Element 0 von y
'unset' auf Variable x
pey$
```

- Die beiden Prozeduren werden immer dann aufgerufen, wenn die in den *trace*-Kommando's angegebenen Operation auf die Variablen ausgeführt werden.
- *write trace*: Zunächst wird Variable verändert, danach die entsprechende *trace*-Prozedur aufgerufen und zuletzt das Ergebnis vom *write* (neuer Wert) zurückgeliefert.
- *read-trace*: Aufruf der entsprechenden *trace*-Prozedur vor dem Lesen (→ *trace*-Prozedur könnte also Wert noch verändern)
- Kommt ein *r/w-trace* auf einen Fehler irgendeiner Art, so wird die ge-*trace*-te Operation abgebrochen (z.B.: Implementierung einer *read-only*-Variablen)

Beispiel:

```
pey$ cat trace1.tcl
#! /usr/local/bin/tcl

trace variable myvar w forceInt
trace variable feld w forceInt
proc forceInt { name element op } {
   if { $element != "" } {
      # it's an array
      set name ${name}($element)
      set name_old ${name}_old($element)
      upvar $name x $name_old x_old
   } else {
      upvar $name x ${name}_old x_old
   }
   if ![regexp -- {^[-+]?[0-9]*$} $x] {
      if [info exists x_old ] {
         set x $x_old
      }
      puts stderr "Value must be integer"
      return
   }
   set x_old $x
}

# KORREKTE Initialisierung:
set myvar 0
for { set i 0 } { $i <= 5 } { incr i } {
   set feld($i) 0
}
```

```
# Korrekte Zuweisung:
set myvar -21
puts "-------------"
puts "myvar: $myvar"

# Falsche Zuweisungen:
puts "-------------"
set myvar abc
puts "myvar: $myvar"
puts "-------------"
set feld(3) +-3
puts "feld(3): $feld(3)"
puts "-------------"
pey$ trace1.tcl
-------------
myvar: -21
-------------
Value must be integer
myvar: -21
-------------
Value must be integer
feld(3): 0
-------------
pey$
```

Oder gleich richtig hart:

```
pey$ cat trace2.tcl
#! /usr/local/bin/tcl

trace variable myvar w forceInt
trace variable feld w forceInt
proc forceInt { name element op } {
   if { $element != "" } {
      # it's an array
      set name ${name}($element)
      set name_old ${name}_old($element)
      upvar $name x $name_old x_old
   } else {
      upvar $name x ${name}_old x_old
   }
   if ![regexp -- {^[-+]?[0-9]*$} $x] {
      if [info exists x_old ] {
         set x $x_old
      }
      error "Value must be integer"
   }
   set x_old $x
}

# KORREKTE Initialisierung:
```

```
set myvar 0
for { set i 0 } { $i <= 5 } { incr i } {
   set feld($i) 0
}
# Korrekte Zuweisung:
set myvar -21
puts "-------------"
puts "myvar: $myvar"

# Falsche Zuweisungen:
puts "-------------"
set feld(3) +-3
puts "feld(3): $feld(3)"
puts "-------------"
set myvar abc
puts "myvar: $myvar"
puts "-------------"
```

```
pey$ trace2.tcl
-------------
myvar: -21
-------------
Error: can't set "feld(3)": Value must be integer
    while executing
"set feld(3) +-3"
    (file "./trace2.tcl" line 35)
pey$
```

16. *time*-Kommando

Syntax:

```
time script ?count?
```

- Defaultwert von *count*: 1

Beispiele:

```
pey$ tcl
tcl> time { set a 5 }
84 microseconds per iteration
tcl> time { set a 5 } 100
16 microseconds per iteration
tcl> time { set a 5 } 1000
15 microseconds per iteration
tcl> time { set a 5 } 10000
15 microseconds per iteration
tcl> time { set b(0) 5 }
108 microseconds per iteration
tcl> time { set b(0) 5 } 100
18 microseconds per iteration
tcl> time { set b(0) 5 } 1000
19 microseconds per iteration
tcl> time { set b(0) 5 } 10000
18 microseconds per iteration
tcl> exit
pey$
```

17. *history*-Kommando

Interaktiv eingetippte Kommandos werden von der Tcl-Shell in einer History-Liste abgelegt. Dazu wird jedes Kommando noch zusätzlich mit einer fortlaufenden Nummer versehen.

history-Kommando	
history	kurz für *history info* ohne *count* (s.u.)
history add command ?exec?	*command* zur *history*-Liste hinzufügen und bei *exec* auch ausführen
history change new ?event?	Benennt das durch *event* spezifizierte Kommando um in *new*
history event ?event?	Liefert die durch *event* spezifizierten Kommando's
history info ?count?	Liefert formatierte Liste der letzten *count* Kommando's (ohne *count*: aller)
history keep count	Die Größe der Historyliste wird auf *count* Kommandos eingestellt. Voreinstellung sind 20 Ereignisse.
history nextid	Die Nummer des nächsten Ereignisses, welches in der Historyliste aufgezeichnet wird, wird zurückgegeben.
history redo ?event?	Das Kommando, welches durch *event* spezifiziert wird, wird nochmals ausgeführt.
history substitute old new ?event?	Das Kommando spezifiziert durch *event* wird nochmals ausgeführt. Jedoch wird der durch *old* spezifizierte Teilstring vorher durch den String *new* ersetzt. Bei der Ersetzung werden keine regulären Ausdrücke bzw. Wildcards (Glob Style Pattern Matching) berücksichtigt.
history words selector ?event?	Liefert Listenelemente zu dem *selector* entsprechenden Ereignis. Listenelemente zählen ab **0**, der letzte Eintrag ist **$**; ein Bereich wird mit **a-b** angegeben (z.B. **0-$**)

- Optionen können eindeutig durch Anfangsbuchstaben abgekürzt werden!

history-Kommando - spezielle Syntax (→ *unknown*)	
!!	Letztes Kommando, entspricht *history redo*
!n	wiederhole Kommando mit Nummer n; entspricht *history redo n*
!prefix	Wiederhole das letzte Kommando, das mit *prefix* beginnt
!pattern	Wiederhole letztes Kommando, das *pattern* entspricht
ˆoldˆnew	ersetzte global *old* durch *new* im letzten Kommando; entspricht *history substitute old new*

Mittels des Parameters *event* (in der Beschreibung "Ereignis") kann bei den Kommandos in obiger Tabelle auf ein bestimmtes Ereignis in der Historyliste Bezug genommen werden.

event:
- positive Zahl: laufende Numerierung
- negative Zahl: vom aktuellen Kommando "rückwärts"
- beliebiger String: das Kommando, das in den Anfangszeichen mit dem String übereinstimmt oder das zu der von String (als regulärer Ausdruck) definierten Sprache gehört

Beispiele:

```
pey$ tcl
tcl> set a 9
tcl> set a [expr $a + 1 ]
tcl> history
     1  set a 9
     2  set a [expr $a + 1 ]
     3  history
tcl> !2
11
tcl> history
     1  set a 9
     2  set a [expr $a + 1 ]
     3  history
     4  set a [expr $a + 1 ]
     5  history
tcl> !-2
12
tcl> hi
     1  set a 9
     2  set a [expr $a + 1 ]
     3  history
     4  set a [expr $a + 1 ]
     5  history
     6  set a [expr $a + 1 ]
     7  hi
tcl> set b [hist w 2-$ 6]
```

```
tcl> history
     1  set a 9
     2  set a [expr $a + 1 ]
     3  history
     4  set a [expr $a + 1 ]
     5  history
     6  set a [expr $a + 1 ]
     7  hi
     8  set b {[expr $a + 1 ]}
     9  history
tcl> !8
[expr $a + 1 ]
tcl> set c !8
tcl> puts $c
!8
tcl> set c [!8]
tcl> puts $c
[expr $a + 1 ]
tcl> exit
pey$
```

18. Modularisierung - *Script Libraries*

- kein explizites Modul-Konzept
- *main script* (Haupt-Modul), unterstützende Skripte in Bücherei
- Vorsicht: Namenskonflikte bei Prozeduren und globalen Variablen !!!

18.1 *unknown*-Kommando

- Wenn der *tcl*-Interpreter ein Kommando nicht findet / kennt, wird *unknown* aufgerufen und der unbekannte Kommandoname samt seinen Argumenten übergeben. Was macht *unknown*?
- die *default*-Implementierung steht in einer Datei namens *init.tcl* oder *TclInit.tcl* oder ... im Katalog */usr/tclX/7.4a-p1* oder ... ???
- ist selbst in *tcl* geschrieben, kann also auf eigene Bedürfnisse angepaßt werden

```
pey$ tcl
tcl> puts [info library]
/usr/tclX/7.4a-p1
tcl> ls /usr/tclX/7.4a-p1
TclInit.tcl     help            tcl.tlib
buildidx.tcl    loadouster.tcl  tcl.tndx
tcl> exit
pey$
```

Die Quelle im Orginal:

```
#-----------------------------------------------------------------------------
# TclInit.tcl -- Extended Tcl initialization.
#-----------------------------------------------------------------------------
# $Id: TclInit.tcl,v 5.0 1995/07/25 06:00:03 markd Rel $
#-----------------------------------------------------------------------------

#
# Unknown command trap handler.
#
proc unknown args {
    if [auto_load [lindex $args 0]] {
        return [uplevel 1 $args]
    }
    if {([info proc tclx_unknown2] == "") && ![auto_load tclx_unknown2]} {
        error "can't find tclx_unknown2 on auto_path"
    }
    return [uplevel 1 [tclx_unknown2 $args]]
}

set auto_index(buildpackageindex) {source $tclx_library/buildidx.tcl}

# == Put any code you want all Tcl programs to include here. ==

if !$tcl_interactive return

# == Interactive Tcl session initialization ==

if ![info exists tcl_prompt1] {
    set tcl_prompt1 {global argv0; puts -nonewline stdout [file tail $argv0]>}
}
if ![info exists tcl_prompt2] {
    set tcl_prompt2 {puts -nonewline stdout =>}
```

```
}
```

Beispiel:

```
pey$ cat myunknown.tcl
#! /usr/local/bin/tcl

proc unknown {name args} {
  # name kann Abkuerzung sein:
  set cmds [info commands $name*]
  puts -nonewline stderr "$name $args :"
  if { [llength $cmds ] != 1 } {
    puts stderr "unbekanntes / nicht eindeutiges Kommando \"$name\""
    exit 1
    # statt dessen ggf.: error "text"
  }
  # Kommando eine level hoeher ausfuehren, damit
  # es die gleiche Ausfuehrungsumgebung hat wie
  # das urspruengliche Kommando
  puts "wird ausgefuehrt"
  uplevel $cmds $args
}

lapp list 123 ; #lapp: Abkuerzung fuer lappend
puts "Liste : $list"

set db mydb
mkDb lib $db

pey$ myunknown.tcl
lapp list 123 :wird ausgefuehrt
Liste : 123
mkDb lib mydb :unbekanntes / nicht eindeutiges Kommando "mkDb"
pey$
```

Die *default*-Version:

1) Wenn das unbekannte Kommando eine Prozedur ist, die in einer Datei der Bibliothek zu finden ist, wird die Datei via *source* geladen und erneut gestartet (→ *autoloading*)

2) Wenn ein Programm mit dem Namen des unbekannten Kommando's existiert, wird es via *exec* gestartet; enthält es keine I/O-Umlenkung, werden *stdin, stdout, stderr* auf die entsprechenden Verbindungen der *tcl*-Anwendung umgelenkt - auch interaktive Kommandos wie *more* oder *vi* möglich
nur in interaktiver Nutzung!

Beispiel:

```
pey$ tcl
tcl>puts [info commands ls]

tcl>ls
dial.tcl       myunknown.tcl  prog4d.tcl
tcl>exit
pey$
```

3) siehe obige Tabelle zu Kurzformen von *history*

4) Behandlung eindeutiger Abkürzungen (siehe *myunknown.tcl*) nur in interaktiver Nutzung!

18.2 *autoloading*

- zur "Modularisierung"
- Effizienz: Prozeduren werden erst dann geladen, wenn sie tatsächlich (zum ersten Mal) benötigt werden
- Prozeduren sinnvoll auf mehrere "kleine" Dateien in einem gemeinsamen Katalog verteilen
- Prozeduren, die "exportiert" werden, tragen einen gemeinsamen Präfix, gefolgt von _, gefolgt vom Prozedurenamen (Konvention)
- Glieches gilt für globale Variable
- Index mit *tcl* erzeugen:

  ```
  tcl> auto_mkindex . *.tcl
  ```

  erzeugt Datei *tclIndex*, die später von *tcl* zum Zugriff auf unbekannte Prozeduren (*unknown*) benutzt wird.
- Eine Datei entspricht Hauptmodul und enthält die Pfadangabe zum Index *tclIndex* - siehe unten Datei *tel*

Beispiel:

```
pey$ pwd
/home/swg/tclSkr/db/dblib

pey$ ls
change.tcl   display.tcl  init.tcl     select.tcl
delete.tcl   getin.tcl    insert.tcl   tel*

pey$ cat tel
#! /usr/local/bin/tcl
# Pfad setzen:
set auto_path \
   [linsert $auto_path 0 "~/tclSkr/db/dblib"]
```

```
db_Init
if { $argc == 1 } {
        db_sel [lindex $argv 0]
        db_Close
        exit 0
}

puts [format "\n\n==========================" ]
puts [format "Einfache Version von telman\n" ]
puts [format "==========================\n\n" ]

set repeat 1
while { $repeat } {
  puts [format "\tEinfuegen         (e)\n"]
  puts [format "\tTel.Nr. Suchen    (s)\n"]
  puts [format "\tAlles Suchen      (a)\n"]
  puts [format "\tLoeschen          (l)\n"]
  puts [format "\tKorrigieren       (k)\n"]
  puts [format "\tBeenden           (q)\n"]
  puts -nonewline [format "\tAuswahl: "]
  gets stdin answer
  switch -exact -- $answer {
    e { db_insert }
    s -
    a { puts -nonewline "Gib Namen: "
        gets stdin name
        if ![string compare $answer "s"] {
        } else {
                db_sel "$name"
        }
        puts "press enter"
        gets stdin enter
      }
    l { puts -nonewline "Gib Namen: "
        gets stdin name
        db_delete $name
      }
    k { puts -nonewline "Gib Namen: "
        gets stdin name
        db_change $name
      }
    default { set repeat 0 }
  }
}

db_Close

pey$ cat init.tcl
#!/usr/local/bin/tcl

set PHONEBOOK "/home/swg//tclSkr/db/Person.all"

# Global: db_headarr, db_attrarr, db_headlist, db_datarr
# db_headarr: Index ist Attributname, Wert ist Spaltennummer
# db_attrarr: Index ist Spaltennummer, Wert ist Attributname
# db_datarr: Index ist PersId (key), Wert ist Restliste mit
#       Trenner "blank"
# db_headlist: Liste der Attribute, wie sie in der Kopfzeile
```

```
#       stehen


proc db_Init { } {
  global table
  global PHONEBOOK
  set PHONEBOOK_OLD [file rootname $PHONEBOOK].old
  exec cp $PHONEBOOK  $PHONEBOOK_OLD
  set table [open $PHONEBOOK r]
  getattr $table
  makearr $table
  close $table
}

proc db_Close  { } {
  global table
  global PHONEBOOK
  global db_headlist
  global db_datarr
  set table [open $PHONEBOOK w]
  puts $table [join $db_headlist :]
  foreach ind [array names db_datarr] {
        set list [linsert $db_datarr($ind) 0 $ind]
        puts $table [join $list :]
  }
}

pey$ cat getin.tcl
# initialisiert db_headarr, db_headlist, db_attrarr
proc getattr { f } {
  global db_headarr
  global db_attrarr
  global db_headlist
  gets $f head
  set db_headlist [ split $head : ]
  # Abbildung: Attr.Bezeichnung -> Spaltenummer:
  set i 0
  while { $i < [llength $db_headlist] } {
    set attr [lindex $db_headlist $i]
    set db_headarr($attr) $i
    set db_attrarr($i) $attr
    incr i
  }
}


proc makearr { f } {
  global db_datarr
  while { [gets $f line] >= 0} {
  set list [split $line :]

  # Schluessel pruefen:
  set persId [lindex $list 0]
  if { [string length $persId ] <= 0 } {
    puts -nonewline stderr "empty key: "
    puts stderr $list
    exit 1
  }
```

```
  if [info exists arr($persId)] {
    incr arr($persId)
    if { $arr($persId) > 1 } {
      puts -nonewline stderr "duplicate key: "
      puts stderr $list
      exit 1
    }
  } else {
    set arr($persId) 1
  }

  set db_datarr([lindex $list 0]) [lrange $list 1 end ]
  }
}
```

```
pey$ cat select.tcl
proc getentries { entry } {
# liefert Liste der PersId's zum Suchbegriff "entry"

  global db_datarr
  set list {}
  foreach ind [array names db_datarr] {
        set el $db_datarr($ind)
        if [regexp -nocase $entry $el ] {
                set list [linsert $list end $ind ]
        }
  }
  return $list
}


proc db_sel { name { what "tel" } } {
  global db_datarr
  if ![string length $name] {
        puts "Mindestens ein Zeichen bitte"
        return
  }
  set matches [getentries $name]
  set count [llength $matches ]

  switch $count {
    0 { puts "Kein Eintrag fuer $name" }
    1 { if [string compare $what "tel" ] {
        db_tdisplay [lindex $matches 0]
        } else {
          db_display [lindex $matches 0]
        }
      }
    default {
        for { set i 0 } { $i < $count } { incr i } {
          if [string compare $what "tel" ] {
           db_tdisplay [lindex $matches $i]
          } else {
            db_display [lindex $matches $i]
          }
          if { $i < [expr $count - 1]} {
             puts "Es gibt weitere [expr $count - $i - 1] Eintraege"
             puts "<Return> fuer den Naechsten"
```

```
            puts "<q> fuer Beenden"
            gets stdin answer
            if [string length $answer] {
               return
            }
          }
        }
      }
   }
}

pey$ cat insert.tcl
proc db_insert {} {
  global db_datarr
  global db_headlist
  set notdone 1
  while { $notdone } {
    puts -nonewline "PersId: "
    if { [ gets stdin persId ] == 0 } {
      continue
    }
    if [info exists db_datarr($persId) ] {
      puts "$persId ist bereits eingetragen"
    } else {
      set notdone 0
    }
  }
  set list {}
  set i 1
  while { $i < [llength $db_headlist ] } {
    set notokay 1
    while { $notokay } {
      puts -nonewline "[lindex $db_headlist $i]: "
      gets stdin value
      puts -nonewline "Okay (y/n): "
      gets stdin answer
      if { [string compare $answer "n"] } {
        set notokay 0
      } else {
        continue
      }
    }
    set list [linsert $list end $value]
    incr i
  }
  set db_datarr($persId) $list
  puts "eingetragen: $db_datarr($persId) "
}

pey$ cat delete.tcl
proc db_delete { entry } {
        global db_datarr
        if ![string length $entry] {
                puts "Mindestens ein Zeichen bitte"
                return
        }
        set matches [getentries $entry]
        set count [llength $matches ]
```

```
        switch $count {
          0 { puts "Kein Eintrag fuer $entry" }
          1 { set key [lindex $matches 0]
              db_display $key
              unset db_datarr([lindex $matches 0])
              puts "ist geloescht!"
              puts "press enter"
              gets stdin answer
            }
          default {
              puts "es sind $count Treffer: "
              for { set i 0 } { $i < $count } { incr i } {
                  set key [lindex $matches $i]
                  db_display $key
                  puts -nonewline "Loeschen (j)? "
                  gets stdin answer
                  if { [string compare $answer "j"] == 0 } {
                        unset db_datarr($key)
                  }
                  puts "ist geloescht!"
                  puts "press enter"
                  gets stdin answer

                }
              }
           }
}
```

```
pey$ cat change.tcl
proc changeattr { attr old } {
        set notdone 1
  while { $notdone } {
        puts "Bisheriger Wert von $attr : $old"
        puts ""
        puts "neuer Wert (<return> wenn beibehalten)"
        gets stdin new
        if { [string length $new] > 0 } {
           puts "neuer Wert: $new"
           puts "if okay press enter else <n>"
           gets stdin answer
           if { [string length $answer] > 0 } {
                continue
           } else {
                return $new
           }
        } else {
                return $old
        }

   }
}

proc changelist { list } {
        global db_attrarr
        set i 0
        while { $i < [llength $list] } {
                 set attrname $db_attrarr([expr $i + 1])
                 set old [lindex $list $i]
```

```
                 set new [changeattr $attrname $old ]
                 set list [lreplace $list $i $i $new ]
                 incr i
        }
        return $list
}



proc db_change {entry} {
        global db_datarr
        global db_headlist
        global db_attrarr
        if ![string length $entry] {
                puts "Mindestens ein Zeichen bitte"
                return
        }
        set matches [getentries $entry]
        set count [llength $matches ]
        switch $count {
          0 { puts "Kein Eintrag fuer $entry / -> insert" }
          1 { set key [lindex $matches 0]
              db_display $key
              set list $db_datarr($key)
              set db_datarr($key) [changelist $list ]
               puts "Neuer Eintrag: "
               db_display $key
               puts -nonewline "press enter"
               gets stdin answer
            }
          default {
              puts "es sind $count Treffer: "
              for { set i 0 } { $i < $count } { incr i } {
                  set key [lindex $matches $i]
                  db_display $key
                  puts -nonewline "Korrigieren (j)? "
                  gets stdin answer
                  if { [string compare $answer "j"] == 0 } {
                        set list $db_datarr($key)
                        set db_datarr($key) [changelist $list]
                        puts "Neuer Eintrag: "
                        db_display $key
                        puts -nonewline "press enter"
                        gets stdin answer
                        return
                  }
                  puts "press enter"
                  gets stdin answer

                }
              }
           }
}

pey$ cat display.tcl
# alles auflisten:
proc printAll {} {
  global db_datarr
```

```
  foreach index [array names db_datarr] {
    puts "$index : $db_datarr($index) "
  }
}

# zu gegebenem Schluessel "PersId"
# Name u. Telefonnummer ausgeben
proc db_tdisplay {key} {
  global db_datarr
  set list $db_datarr($key)
  set name [lindex $list 0]
  set vname [lindex $list 1]
  set anrede [lindex $list 2]
  set phone [lindex $list [expr [llength $list] -1 ]]
  if { [string length $phone] <= 0 } {
    set phone "Telefonnummer unbekannt"
  }
  puts [format "\t%s %s %s : %s\n" $anrede $vname $name $phone]
}

#Eintrag ausgeben
proc db_display {key} {
  global db_datarr
  set list $db_datarr($key)
  puts -nonewline "\t[lindex $list 2] "
  puts -nonewline "[lindex $list 1] "
  puts [lindex $list 0]
  set i 3
  while { $i < [llength $list] } {
    puts "\t[lindex $list $i]"
    incr i
  }
}

proc db_ldisplay { key list } {
  global db_headarr
  global db_attrarr
  puts "$db_attrarr(0) : $key "
  set i 0
  while { $i < [llength $list] } {
    set attrname $db_attrarr([expr $i + 1 ])
    puts "\t$attrname : [lindex $list $i]"
    incr i
  }
}

pey$
```

Index erzeugen:

```
pey$ tcl
tcl> auto_mkindex . *.tcl
tcl> ls
change.tcl   display.tcl  init.tcl     tclIndex
delete.tcl   getin.tcl    insert.tcl   select.tcl   tel
tcl> exit
pey$
```

Und ausführen:

```
pey$ tel Emu
        Herr Dr.h.c. F.J. Emu
        SSS e.V.
        0511/4321645

pey$ tel


===========================
Einfache Version von telman

===========================



        Einfuegen         (e)

        Tel.Nr. Suchen    (s)

        Alles Suchen      (a)

        Loeschen          (l)

        Korrigieren       (k)

        Beenden           (q)

        Auswahl: s
Gib Namen: Emu
        Herr Dr.h.c. F.J. Emu : 0511/4321645

press enter

        Einfuegen         (e)

        Tel.Nr. Suchen    (s)

        Alles Suchen      (a)

        Loeschen          (l)

        Korrigieren       (k)

        Beenden           (q)

        Auswahl: a
Gib Namen: Emu
        Herr Dr.h.c. F.J. Emu
        SSS e.V.
        0511/4321645
```

```
press enter

        Einfuegen         (e)

        Tel.Nr. Suchen    (s)

        Alles Suchen      (a)

        Loeschen          (l)

        Korrigieren       (k)

        Beenden           (q)

        Auswahl: k
Gib Namen: Emu
        Herr Dr.h.c. F.J. Emu
        SSS e.V.
        0511/4321645
Bisheriger Wert von Name : Emu

neuer Wert (<return> wenn beibehalten)
Strauss
neuer Wert: Strauss
if okay press enter else <n>

Bisheriger Wert von Vorname : F.J.

neuer Wert (<return> wenn beibehalten)

Bisheriger Wert von Anrede : Herr Dr.h.c.

neuer Wert (<return> wenn beibehalten)

Bisheriger Wert von Institution : SSS e.V.

neuer Wert (<return> wenn beibehalten)

Bisheriger Wert von Tel : 0511/4321645

neuer Wert (<return> wenn beibehalten)

Neuer Eintrag:
        Herr Dr.h.c. F.J. Strauss
        SSS e.V.
        0511/4321645
press enter
        Einfuegen         (e)

        Tel.Nr. Suchen    (s)
```

```
        Alles Suchen      (a)

        Loeschen          (l)

        Korrigieren       (k)

        Beenden           (q)

        Auswahl: a
Gib Namen: Str
        Herr Dr.h.c. F.J. Strauss
        SSS e.V.
        0511/4321645
Es gibt weitere 1 Eintraege
<Return> fuer den Naechsten
<q> fuer Beenden

        Herr Max Streit

press enter

        Einfuegen         (e)

        Tel.Nr. Suchen    (s)

        Alles Suchen      (a)

        Loeschen          (l)

        Korrigieren       (k)

        Beenden           (q)

        Auswahl: q
pey$
```

Anm.: Dieses Programm ist natürlich verbesserungsfähig, es soll nur die Idee der Modularisierung demonstrieren!