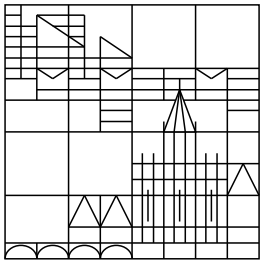


Universität Konstanz

UNIX & Co. in 111 Schritten

Karsten Weihe

Konstanzer Schriften in Mathematik und Informatik
Nr. 31, April 1997

ISSN 1430–3558

Universität Konstanz
Postfach 5560/D 188, 78434 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften

UNIX & Co. in 111 Schritten

Karsten Weihe[1]

Lehrstuhl für praktische Informatik I
(Algorithmen und Datenstrukturen)

31/1997
Universität Konstanz
Fakultät für Mathematik und Informatik

Zusammenfassung

Dies ist eine Zusammenstellung aller schriftlichen Unterlagen für Vorlesungen und Rechnerübungen zum Thema UNIX/X11-Window-System/Internet, die der Autor im Wintersemester 1995/96 durchgeführt und im Wintersemester 1996/97 nach umfassender Überarbeitung wiederholt hat. In beiden Semestern war dies der erste Teil einer kombinierten Vorlesung zu UNIX und C++, in dem der C++-Teil auf dem UNIX-Teil aufbaute, der UNIX-Teil aber auch allein besucht werden konnte. Kern der Unterlagen ist eine Sammlung von Arbeitsblättern, die insgesamt 111 kleine praktische Aufgaben nebst Lösungen umfassen und als Leitfaden zum selbständigen Aneignen des Stoffes dienen.

Einführung

Die Veranstaltung zu UNIX/X11/Internet umfaßte sechs Vorlesungstermine und acht Übungstermine am Rechner. Sie richtete sich an Hörerinnen und Hörer aller Fachrichtungen ohne Vorkenntnisse und wurde auch von Studierenden (und wissenschaftlichen Bediensteten) verschiedenster Richtungen sowie von Gasthörern in Anspruch genommen.

Einerseits war der Kurs zwar sehr praxisorientiert („*learning by doing*“). Andererseits war das Hauptziel nicht die Kenntnis einer Vielzahl von UNIX- und sonstigen Kommandos, sondern — am Beispiel UNIX & Co. — die Vermittlung von praxisorientiertem Hintergrundwissen und allgemeiner Kompetenz im Umgang mit Rechnern, Betriebssystemen, Software und Netzwerken.[2] Besonders eingeübt wurden dabei zwei Fertigkeiten: die eigenständige Beschaffung von Detailinformationen über Systemspezifika mit Hilfe von typischen, didaktisch eher unbedarften Online-Hilfen („*man pages*“) und die Fähigkeit, mit den alltäglichen Unzulänglichkeiten von Softwaresystemen und mit unvorhergesehenen Situationen aller Art fertigzuwerden („*...when you can't find your system administrator*“). Da die Audienz sowohl in

[1]Universität Konstanz, Fakultät für Mathematik und Informatik, Postfach 5560/D188, 78434 Konstanz, karsten.weihe@uni-konstanz.de, http://www.informatik.uni-konstanz.de

[2]Auch der Spaß an der Materie sollte nicht zu kurz kommen, wovon die (nicht als solche gekennzeichneten) „Belohnungsaufgaben“ am Ende der ersten sechs Arbeitsblätter zeugen.

den Vorkenntnissen als auch in den Erwartungen an diesen Kurs sehr heterogen war, mußte bei der Behandlung von zugrundeliegenden Prinzipien (z.B. formale Sprachen) weitgehend auf eine abstrakte, formalere Sicht verzichtet werden.

Entgegen der konventionellen Vorgehensweise, bei der die Übungen die Vorlesung begleiten und nachbereiten, wurde jedes Thema zuerst in praktischer Form in den Rechnerübungen behandelt, und die Vorlesung griff das Thema jeweils im nachhinein auf, um es etwas systematischer zu diskutieren und die Hintergründe zu beleuchten. Dahinter steht die Überlegung, daß der Umgang mit praktischen Dingen und das unmittelbare Erleben der Funktionsweise ein intuitives Verständnis der Materie vermittelt, das in Vorlesungen nicht erreicht werden kann (auf dem die Vorlesung ihrerseits aber gut aufbauen kann). Diese Konzeption ist von der Audienz sehr gut angenommen worden: Bei der Evaluierung wurde auch die Beurteilung dieser Konzeption erfragt, und weit überwiegend wurde das bestmögliche Urteil („*sehr empfehlenswert auch für andere Veranstaltungen*“) abgegeben.

Zu jedem Übungstermin am Rechner wurde ein Arbeitsblatt nebst Lösungsblatt als ASCII-Text auf dem Rechner bereitgestellt. Jedes dieser Blätter ist einem speziellen Thema gewidmet und sollte während des jeweiligen Übungstermins alleine oder in Gruppen selbständig, aber unter Betreuung des anwesenden Dozenten am Rechner bearbeitet werden. Jede der insgesamt 111 Aufgaben behandelt ein spezifisches Detail und ist als Anleitung zur selbständigen Erarbeitung dieses Details gedacht. Der Umfang der Arbeitsblätter ist so bemessen, daß sie nicht unbedingt während der Übungstermine zu schaffen waren, sondern zusätzliche Bearbeitungszeit außerhalb der Termine erfordern. Die Bearbeitung der Übungsaufgaben wurde nicht abgeprüft (dazu sind die Aufgaben auch zu einfach, da sie Anleitungs- und nicht Prüfungscharakter haben), sondern am Ende des Semesters wurde der Lernerfolg in einer mündlichen Einzelprüfung (mit der Möglichkeit zur Wiederholung bei einer mangelhaften Leistung) am Rechner bewertet.

Die folgenden Unterlagen sind zwangsläufig sehr spezifisch auf die Konfiguration des Systems zugeschnitten, auf dem die praktischen Übungen abgehalten wurden. Sie sind daher in der vorliegenden Form nicht für die Verwendung auf anderen Systemen geeignet (und viele Details sind schon jetzt — nach wenigen Monaten! — veraltet). Diese Zusammenstellung soll daher auch nur die inhaltliche, didaktische und ablauftechnische Konzeption, die hinter diesem Kurs stand, exemplarisch vorstellen und richtet sich an Lehrende, nicht an Lernende. Die Unterlagen sollten detailliert genug sein, um als Vorlage für einen ähnlichen Kurs zu dienen, wobei eben diverse technische Details modifiziert werden müssen.

Erfahrungen

1. Auch bei der Evaluierungsfrage nach der praktischen Umsetzung der Konzeption wurde überwiegend die Bestnote („*reibungslos*“) bzw. die zweitbeste Note aus insgesamt fünfen vergeben, das heißt auch aus der Sicht der TeilnehmerInnen ist die Konzeption tatsächlich „machbar“.

2. Durch die Detailliertheit der Aufgaben auf den Arbeitsblättern und der Lösungen konnten die Hörerinnen und Hörer die Arbeitsblätter im wesentlichen tatsächlich selbständig bearbeiten. Dies hat für viele Leute die Teilnahme am Kurs überhaupt erst möglich gemacht, weil diese selbständige Bearbeitung natürlich auch außerhalb der angesetzten Termine stattfinden kann. In kaum einem Studiengang ist dieser Kurs als Wahlpflicht- oder Zusatzfach anrechnungsfähig, das heißt er stellt für die Mehrzahl der TeilnehmerIn-

nen eine zusätzliche zeitliche Belastung dar, die mit den anderen besuchten Lehrveranstaltungen kaum zeitlich zu koordinieren sind (von den zeitlichen Belastungen der wissenschaftlichen Bediensteten, die den Kurs besucht haben, ganz zu schweigen). Zudem war es für eine Reihe von TeilnehmerInnen nicht möglich, kontinuierlich während des Semesters zu arbeiten, das heißt eine permanente Leistungskontrolle durch regelmäßige bewertete Übungsblätter hätte vielen Leuten den Erwerb eines Scheins unmöglich gemacht. Dies betraf einerseits Studierende mit umfangreichem Arbeitspensum während des Semesters, die sich auf UNIX erst während der Semesterferien konzentrieren konnten. Andererseits gab es eine Reihe von Späteinsteigern — im wesentlichen Studierende, die erst durch Mundpropanganda von diesem Kurs erfahren haben, und Diplomanden, die im Laufe des Semesters ihre Diplomarbeit fertiggestellt hatten und erst danach Zeit für einen solchen Kurs erübrigen konnten.

3. Bei der Evaluierung wurde auch speziell gefragt, wie die Gestaltung der praktischen Übungstermine beurteilt wurde, gerade auch im Vergleich zu Frontalveranstaltungen mit Projektor oder Panel, in denen der Dozent oder die Dozentin die einzelnen Schritte vorführt und die TeilnehmerInnen diese Schritte simultan nachvollziehen. Auch hier wurde weit überwiegend die Bestnote („*sehr empfehlenswert auch für andere Veranstaltungen*“) vergeben.

4. Dadurch, daß die TeilnehmerInnen weitgehend selbständig gearbeitet haben, war es mir möglich, die Veranstaltung ohne Assistenz durchzuführen und dennoch einer nicht ganz kleinen Audizenz[3] eine vernünftige Betreuung anzubieten, weil ich meine Arbeitskraft konzentriert dort einsetzen konnte, wo Betreuung wirklich not tat. Ein hilfreicher Nebeneffekt dieser konzentrierten Betreuung war die permanente intensive Rückkopplung.

5. Allerdings war diese Möglichkeit zur intensiven punktuellen Betreuung auch absolut notwendig. Durch die Komplexität von UNIX & Co. bietet jede noch so kleine Abweichung von dem roten Faden, der durch die Arbeitsblätter vorgegeben war, eine Fülle von Fallstricken, in denen sich AnfängerInnen hoffnungslos verstricken können. Ein großer Teil der Betreuung bestand daher einfach in „Feuerlöschaktionen“. Da die Arbeitsblätter zwangsläufig mehr oder weniger in Umgangssprache formuliert sind, sind TeilnehmerInnen auch öfters ohne Absicht von diesem roten Faden abgewichen, einfach weil sie ein entscheidendes Detail der Aufgabenstellung mißverstanden oder schlicht übersehen haben.

   Interessanterweise haben diese Probleme auch einige Bugs im System und in der Konfiguration zu Tage gebracht.

6. Diese Feuerlöscharbeiten hatten andererseits zwei positive Effekte: Erstens sind solche verfahrenen Situationen ausgezeichnete Lehrbeispiele, anhand deren man gut allgemeinere Strategien zur Lösung von unvorhergesehenen Problemen demonstrieren kann — vom zielgerichteten, effizienten Durchstöbern von Manual Pages bis hin zu Detektivspielereien beim Einkreisen des Fehlverhaltens. Daß diese unvorhergesehenen Probleme oft genug auch für mich zunächst einmal eine ernsthafte Herausforderung darstellten und ich selbst erst zusammen mit dem oder der jeweils Betroffenen eine Lösung erarbeiten mußte, hat sicher zum Transfer von Kompetenz beigetragen. Zweitens ist meine

[3] Im Wintersemester 1996/97 haben mehr als dreißig Leute einen Schein erworben. Eine weitere Anzahl hat den Kurs nebenher aus Interesse mitgemacht, ohne die Arbeit in einen Schein investieren zu wollen. Insgesamt haben sich mehr als siebzig Leute für den Kurs angemeldet.

generelle Erfahrung (auch bei anderen praktischen Themen wie C++), daß Fehler oft ein viel tieferes Verständnis in das Wesen der Konzepte ermöglichen als eine perfekt funktionierende Demonstration.[4]

7. Allerdings erforderte die Erstellung der Arbeitsblätter und Lösungen *sehr* viel Zeit in der Vorbereitung. Das Hauptproblem war die Zerlegung eines so komplexen, nichtlinearen Systems wie UNIX & Co. in eine lineare Folge von 111 elementaren Einheiten, so daß man zum Verständnis einer Einheit nur die vorhergehenden Einheiten benötigt, nicht die nachfolgenden. Bei der konkreten Ausarbeitung kamen andauernd unvorhergesehene Abhängigkeiten zwischen einzelnen Konzepten (auch zyklische!) zum Vorschein, die aufwendige Umorganisationen notwendig machten. Viele dieser Probleme tauchten sogar erst beim Austesten der Befehlseingaben auf, das heißt penibles, zeitaufwendiges Austesten aller Aufgaben ist ein absolutes Muß.

8. Gerade beim Austesten der Aufgaben gab es im Wintersemester 1996/97 etliche böse Überraschungen. Es ist einfach unglaublich, wieviele kleine, unscheinbare Details sich innerhalb eines Jahres ändern können! Besonders böse waren etliche kleine Details, die die Abhängigkeiten zwischen den einzelnen Aufgaben geändert haben und daher wieder Umorganisationen erforderlich machten.

9. Durch Koordinationsprobleme, die mit diesem Kurs nichts zu tun haben, gab es zwei Gruppen von TeilnehmerInnen mit verschiedenen Login-Shells (*tcsh* und *bash*). Für einen UNIX-Kurs, der naturgemäß sehr intensiv auf Shells und damit zusammenhängende Themen eingeht, bedeutet das einen erheblichen Mehraufwand, zumal die Default-Konfigurationen für die beiden Shells auf unserem System auch nicht hundertprozentig übereinstimmen (was wohl auch kaum sinnvoll ginge). Um aus der Not eine Tugend zu machen, habe ich die Flucht nach vorn angetreten und bin das Thema auf dem siebten und achten Arbeitsblatt offensiv angegangen. Schließlich paßt es ja auch sehr gut unter die oben formulierte allgemeine Zielsetzung, mit alltäglichen Problemen umgehen zu lernen.

10. In den mündlichen Rücksprachen am Ende des Kurses habe ich mehr Wert auf praktische Kompetenz im Umgang mit dem Rechner als auf Wissensinhalte gelegt. Die TeilnehmerInnen sollten mit Hilfe der Manual Pages Eingabebefehle mit Optionen und Argumenten konstruieren, um kleine Aufgaben zu lösen. Diese Aufgaben waren so gestellt, daß ihre Lösung auch ein gewisses Hintergrundwissen und intuitives Verständnis erfordert.

    Zum Beispiel habe ich oft Aufgaben gestellt, die auf einen Aufruf von `grep` mit einem Regular Expression hinauslaufen. Dies erfordert nicht nur Vorwissen und Verständnis in bezug auf Regular Expressions, sondern auch ein klares, scharfes gedankliches Trennen der verschiedenen beteiligten Ebenen, denn viele Sonderzeichen in Regular Expressions müssen vor der Interpretation durch die Shell geschützt werden. Die genaue Unterscheidung kann im Einzelfall ein kniffliges Durchdenken der Situation erfordern, zum Beispiel wenn ein Sonderzeichen weder von der Shell noch von `grep` interpretiert werden soll.

    Nach meiner Beobachtung hat der Kurs der überwiegenden Mehrzahl der TeilnehmerInnen, die eine Rücksprache mitgemacht haben, die hierfür notwendigen Denkmuster vermittelt (auch Leuten aus eher mathematikfernen Studiengebieten).

[4] Einige Fehler (Features?) sind auch systemimmanent und waren nicht zu verbergen. Die 8. und 9. Aufgabe auf dem 8. Arbeitsblatt sind ein krasses Beispiel, wo ich versucht habe, aus der Not eine Tugend zu machen.

Übersicht

Die Unterlagen bestehen aus den Arbeitsblättern, den Lösungsblättern und den Folien zu den Vorlesungen. Das erste Arbeitsblatt wird durch ein kurzes Einstiegsblatt eingeleitet, mit dem die TeilnehmerInnen sich selbständig einloggen können. Jedes Lösungsblatt ist unmittelbar hinter dem entsprechenden Arbeitsblatt eingefügt. Arbeitsblätter und Folien sind chronologisch geordnet und daher vermischt, da die Vorlesungs- und Übungstermine vermischt waren. Den Abschluß bilden praktische Aufgaben, mit denen die TeilnehmerInnen ihren Übungsschein zu einem Praktikumsschein aufwerten konnten. Alle diese Unterlagen sind auch on-line verfügbar:

ftp://ftp.informatik.uni-konstanz.de/pub/algo/personal/weihe/unixkurs.tar.Z

Die Termine im Überblick:

1. Vorlesung: Einführung und Organisatorisches
2. Rechnerübung: erste Schritte
3. Rechnerübung: der Editor *emacs*
4. Rechnerübung: Email
5. Rechnerübung: Files und Directories
6. Vorlesung: Abarbeitung von Kommandos, Aufbau von UNIX, Basiskonzepte wie Files, Prozesse, Bereiche
7. Rechnerübung: „Kommandos für Profis“, insbesondere zur Informationsbeschaffung
8. Vorlesung: Informationsbeschaffung unter UNIX
9. Rechnerübung: weiteres zur Arbeit mit Files
10. Rechnerübung: Shells
11. Vorlesung: formale Sprachen
12. Rechnerübung: Shell-Programmierung
13. Vorlesung: Shell-Programmierung
14. Vorlesung: Netzwerke

Achtung!

Diese Unterlagen sind nach *didaktischen*, nicht nach *systematischen* Zielsetzungen erstellt. Sie enthalten daher grobe Vereinfachungen und sogar Passagen, die von einem eher systematischen Standpunkt aus als problematisch anzusehen sind. Durch die „Übersetzung“ nach LaTeX könnten sich kleine Fehler eingeschlichen haben.

Ich bin für Feedback jeder Art dankbar!

Vorlesung und Praktikum

UNIX/C++

Wintersemester 1996/97

Universität Konstanz

Fakultät für Mathematik und Informatik

Karsten Weihe

Inhalte:

1. **Semesterhälfte:** Umgang mit Computern, lokalen Computerverbunden und weltweiten Computernetzen.

   Am Beispiel UNIX–Workstationnetz des Bereichs Informatik und Internet.

2. **Semesterhälfte:** Einführung in die UNIX–nahe Programmiersprache C++.

Zielsetzungen des Kurses:

- großer Schritt zum UNIX *Power–User*
- *sinnvoller* Umgang mit Internet
- Grundlagen von C++

Geplante Fortsetzung: wie werde ich C++ *Power–User*

Betrieb: jeder Termin ist

- entweder Vorlesung
- oder Übung am Computer im Lehre–Pool Informatik.

$\Rightarrow \approx 50 : 50$

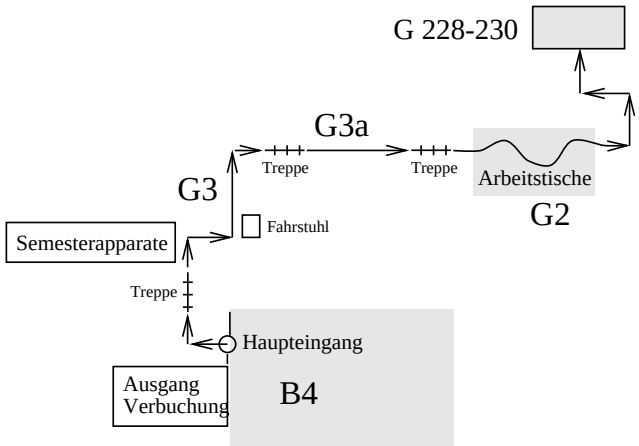
Termine:

- Montag 16.00–18.00:
  F 426 oder Lehre–Pool
- Donnerstag 16.00–18.00:
  D 301 oder Lehre–Pool

Achtung: nächste zwei Termine im Lehre–Pool

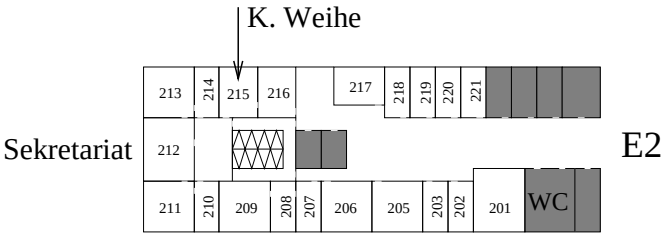
Lehre–Pool: G 228–230.

Achtung: nur über Bibliothek erreichbar!



Übungen am Computer:

- selbständige Bearbeitung von Übungsaufgaben
- Betreuer sind zu Terminen anwesend und können helfen
- Übungsaufgaben können auch außerhalb der Termine bearbeitet werden (ohne anwesende Betreuer)
- Guter Rat (fast!) jederzeit: E 215
- schriftliche Kursunterlagen: E 212



Übungs- und Praktikumsschein:

- Übungsschein über 2 SWS für UNIX
- Übungsschein über 2 SWS für C++
- oder Übungsschein über 4 SWS für beide Hälften zusammen
- zusätzlich Praktikumsschein möglich
- oder auch einfach: *just for fun*

Scheinkriterien:

Übungsschein: Übungsaufgaben bearbeiten und am Ende mündliche Rücksprache erfolgreich bestehen

Praktikumsschein: zusätzliche Hausaufgaben (Programmieraufgaben) bearbeiten und am Computer abnehmen lassen

Zugang zu Computer:

- Benutzerkennung. Bsp.: *weihe*, *muellerr*, *muellerf*.
- Persönliches, geheimes Passwort.
- Benutzerkennung und Passwort abholen: Termin am Donnerstag oder im E 215

Wichtig:

- Passwort **unbedingt** geheimhalten
- **niemals** Passwort an andere weitergeben, auch nicht an gute Freunde oder Betreuer
- möglichst nicht aufschreiben
- falls doch: niemandem zeigen
- sich nicht beim Eintippen des Passworts über die Schulter schauen lassen

Selbst gewähltes Passwort

1. Übungsblatt enthält Aufgabe, das ausgegebene Passwort durch ein selbst ausgedachtes zu ersetzen

⇒ **1. Hausaufgabe:**
gutes Passwort ausdenken (max. 8 Zeichen)

Schlechte Passwörter:

- Sehr kurze Passwörter (≤ 6 Zeichen).
- Wörter der deutschen (oder einer anderen) Sprache.
- Eigennamen.
- Mit persönlichen Daten (z.B. Geburtsdatum, Matrikelnummer).
- Mit normaler Groß- und Kleinschreibung.
- Ohne Sonderzeichen.
- Schwierig zu merken

Gutes Passwort: tOteh@se

Informatik an der Uni Konstanz

- Flur E2
- Drei Arbeitgruppen:
  - Algorithmen und Datenstrukturen (Prof. Wagner)
  - Datenbank- und Informationssysteme (Prof. Scholl)
  - neu: Softwareentwicklung (Prof. Pree)
- Zyklus Informatik im Grundstudium: Start mit Informatik I (Pree)
  - Erlernen einer praxisnahen Programmiersprache
  - Programmiermethodik
  - Wie ist ein Computer aufgebaut
  - Algorithmen

Weitere Veranstaltungen der Informatik:

- Theoretische Informatik = Informatik III (Scholl)
- Arbeitskreis objekt–orientiertes Programmieren in JAVA (Brandes)
- Algorithmen und Datenstrukturen (Wagner)
- Kombinatorische Optimierung (Wagner)
- Objekt–orientierte Datenbanken (Scholl)
- Deklarative Programmierung (Scholl)

Was kann ein Computer?

1. Antwort:

- Umfangreiche Rechnungen blitzschnell durchführen.
- Riesige Datenmengen speichern und verwalten.
- Technische Vorgänge steuern.
- Texte und Zeichnungen druckfertig aufbereiten.
- Expertenwissen speichern und daraus eigenständig Schlußfolgerungen ziehen.
- Weltweite Kommunikation und Information (Stichwort Internet).
- Reale Prozesse vorab simulieren und analysieren.
- ...

2. Antwort:

- Einzelne Nullen und Einsen speichern.
- Hin- und her kopieren.
- Nach formalen Logikregeln verknüpfen.

Von Nullen und Einsen zu vielfältigen Anwendungen

- Betriebssystem: empfängt Befehle vom Benutzer und führt sie aus

  Bei uns: UNIX (Solaris)
- Ausführung eines Befehls

  $\Rightarrow$ zugehöriges Programm wird aufgerufen
- Fenstersystem: erlaubt bequemeren Umgang (Klicken statt Tippen)

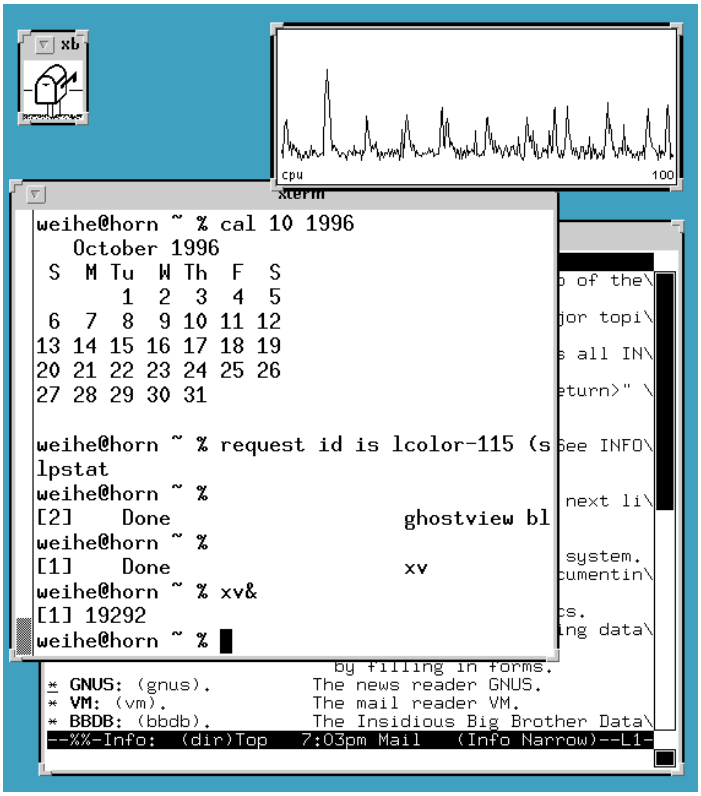
  Bei uns: X11
- Programmiersprache: um neue Programme zu schreiben

  In diesem Kurs: C++

What's in a name?

- *UNIX*: MULTICS knapp verfehlt
- *X11* : X folgt auf W(indow) im Alphabet
- *C++*: BCPL $\rightarrow$ B $\rightarrow$ C $\rightarrow$ C++

Fenstersystem:



UNIX–Kommandos:

```
% cal 10 1996

   October 1996
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

% cal 10 96

   October 96
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

% CAL 10 96

CAL: Command not found.
```

Wichtig: UNIX nimmt alles wörtlich und sehr genau!

Programmiersprache C++

```
#include<iostream.h>

int binom_coeff (int n, int k)
{
   if (n<0 || k<0 || n<k) return 0;
   int result = 1;
   for (int i=1; i<=k; i++)
    {
      result *= n-i+1;
      result /= i;
    }
   return result;
}

void pascal_triangle (int n)
{
   for (int i=0; i<=n; i++)
   {
      for (int j=0; j<=i; j++)
         cout << " " << binom_coeff (i, j);
      cout << endl;
   }
}

int main ()
{
  pascal_triangle (10);
}
```

Neues Programm erzeugen und ausprobieren:

```
% g++ pascal.cc -o pascal
% pascal
 1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1
 1 5 10 10 5 1
 1 6 15 20 15 6 1
 1 7 21 35 35 21 7 1
 1 8 28 56 70 56 28 8 1
 1 9 36 84 126 126 84 36 9 1
 1 10 45 120 210 252 210 120 45 10 1
```

Erläuterungen:

- erst Übersetzung in Nullen und Einsen durch Programm *g++*
- dann Ausführung
- Kommando = Name des Programms = „pascal“

Exkurs: Hierarchie von Abstraktionsebenen

Betriebssystem	Unbewußte Fertigkeiten (motorisch, kommunikativ...)
Computer	Lebewesen
CPU- und Peripheriekomponenten	Organe
Integrierte Schaltkreise	Zellkomplexe
Logische Bausteine	Zellen
Transistoren u.ä	Zellbausteine
Moleküle	
Atome	

Achtung: Vergleich hinkt massiv!

Natürliche Zahlen im Computer darstellen: **Dualcode** mit fester Stellenzahl

0	=	00000000
1	=	00000001
2	=	00000010
3	=	00000011
4	=	00000100
5	=	00000101
6	=	00000110
7	=	00000111
8	=	00001000
9	=	00001001
10	=	00001010
31	=	00011111
32	=	00100000
255	=	11111111

Wichtig: nur begrenzte Anzahl von Zahlen darstellbar!

Zeichen im Computer darstellen: **ASCII–Code**

ASCII = American Standard Code for Information Interchange

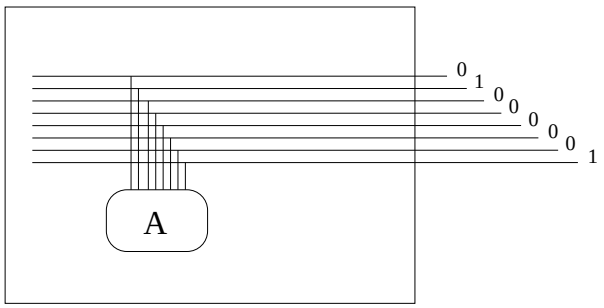
Jedem Zeichen wird eine Zahl zugeordnet, die mit acht Bit (= 1 Byte) darstellbar ist, also $0 \ldots 255 = 2^8 - 1$.

$\Rightarrow$ Zeichen wird als Dualdarstellung der ASCII-Kennummer im Computer gespeichert.

Beispiele:

'a'	=	97	=	01100001 } fortlaufend
...				
'z'	=	122	=	01111010 }
'A'	=	65	=	01000001 } fortlaufend
...				
'Z'	=	90	=	01011010 }
'0'	=	48	=	00110000 } fortlaufend
...				
'9'	=	57	=	00111001 }
'!'	=	33	=	00100001
'?'	=	63	=	00111111

Zeichen von Tastatur eingeben:

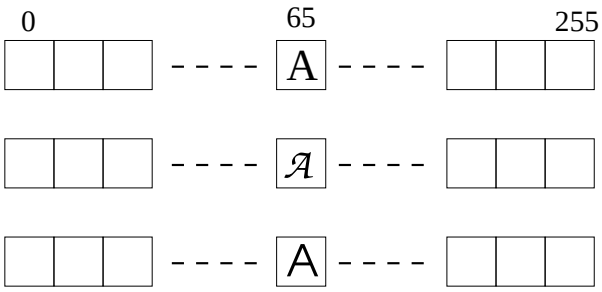


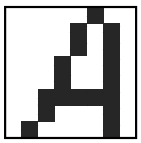
Tastatur

Zeichen auf Bildschirm schreiben:

Computer speichert intern Pixel–Matrix aus Nullen und Einsen für jeden Buchstaben. Zugriff über ASCII–Kennummer von 'A' (65).

Für verschiedene Schriftarten werden verschiedene Sammlungen von Pixel–Matrizen gespeichert.





Computernetz:

1. Alle Rechner des Lehre–Pools sind untereinander vernetzt
   ⇒ Man kann an jedem Computer auf die gleichen (eigenen) Daten zugreifen.
2. Lehre–Pool ist mit Rechenzentrum vernetzt
   ⇒ Zentrale Dienste wie z.B. *koala* sind von Lehre–Pool aus benutzbar.
3. Rechenzentrum hängt am Internet
   ⇒ Weltweit sind Dienste aller Art abrufbar.

Dienste:

- *Email*: kostenlose Briefe in alle Welt innerhalb von Minuten
- *News*: weltweite Diskussionforen zu allen denkbaren Interessensgebieten
- *World Wide Web*:
  - Schnelle Informationsbeschaffung
  - Surfen auf dem Netz :-)

UNIX/C++–Praktikum Wintersemester 1996/97

Praktisches Aufgabenblatt

17. Oktober 1996

Bearbeiten Sie die folgenden Aufgaben soweit wie möglich während dieses Termins. Aufgaben, die Sie nicht schaffen, sollten Sie auf jeden Fall bis zum nächsten Termin nachgearbeitet haben. Erfolgreiche Bearbeitung wird nicht von uns nachgeprüft, ist aber für den weiteren Erfolg wichtig.

Aufgabe I: Mauszeiger

Neben der Tastatur liegt eine „*Maus*“ auf einer speziellen Unterlage. Bewegen Sie die Maus ein bißchen auf der Unterlage hin und her. Was sich daraufhin auf dem Bildschirm bewegt, ist der „*Mauszeiger*“. Wenn Sie den in die hellere Fläche unten rechts bewegen, ändert sich seine Gestalt.

Aufgabe II: Einloggen

Tippen Sie Ihre Benutzerkennung ein. Was Sie eingetippt haben, muß in der Zeile nach „*login:*“ auf dem Bildschirm sichtbar werden. (Vor „*login:*“ steht in der Zeile noch der Name des Rechners, an dem Sie arbeiten.) Erscheinen die Zeichen nicht dort, fragen Sie bei uns nach.

Falls Sie Fehler gemacht haben, korrigieren Sie die Fehler, indem Sie mit der Taste „*Backspace*“ alles bis zum ersten Fehler wieder löschen. Sobald Ihre Benutzerkennung korrekt dasteht, tippen Sie auf die Taste „*Return*“.

Als nächstes tippen Sie das erste Zeichen Ihres Passwortes ein. Das Zeichen darf dabei nicht auf dem Bildschirm erscheinen. Tut es das doch, fragen Sie bei uns nach. Ansonsten tippen Sie den Rest Ihres Passwortes ebenfalls ein. Nachdem Sie alle Zeichen eingetippt haben, drücken Sie wieder „*Return*“.

Aufgabe III: wieder Mauszeiger

Jetzt müßte viel Wirbel passiert sein, und Sie sehen schließlich ein paar weiße (und ein eher schwarzes) „*Fenster*“ mit grauem Rand auf einem blauen Untergrund. Bewegen Sie den Mauszeiger hin und her und beobachten Sie die Änderung seiner Gestalt.

Wenn im weiteren davon die Rede ist, in ein „*Fenster*“ zu gehen, ist genauer gemeint, den Mauszeiger in die weiße (bzw. schwarze) Fläche dieses Fensters zu bewegen.

Aufgabe IV: Ausloggen

Gehen Sie in das Fenster „*Console JUST for OUTPUT*“. Tippen Sie nun die Zeichenfolge „*logout*“ ein. Diese sechs Zeichen erscheinen hinter dem Prozentzeichen, getrennt durch ein Leerzeichen („*Blank*“ oder „*Space*“ genannt). Drücken Sie nun die Taste „*Return*“. Nach etwas Wirbel sind Sie jetzt wieder aus dem Computer heraus; Sie haben sich ausgeloggt. Loggen Sie sich nun wieder ein wie in Aufgabe II beschrieben.

Aufgabe V: Aufgaben 1–14 bearbeiten

Gehen Sie in das unterste Fenster, wo oben in der Mitte des grauen Randes „*xterm*“ steht. Tippen Sie nun die folgenden mysteriösen Zeichen ein:

„*less ~weihek/Aufgabe1*“

Wenn Sie etwas falsch gemacht haben, gibt es eine kurze englische Fehlermeldung. Ansonsten gibt es einen freundlichen deutschen Text. Bearbeiten Sie die Aufgaben in diesem Text der Reihe nach.

Wichtig: Vergessen Sie nicht, sich am Ende wieder auszuloggen, wenn Sie für heute fertig sind!!!

1 Erste Schritte

Hallo,

willkommen zum UNIX/C++-Praktikum im Wintersemester 1996/97!

Der folgende Text enthält ein paar Aufgaben, mit denen man die ersten Schritte im Umgang mit dem Rechner machen kann. Wir empfehlen, sich Notizen zu den Aufgaben zu machen und diese Notizen im nachhinein noch einmal durchzugehen. Wenn Sie mit Aufgaben nicht klarkommen oder das Gefühl haben, die Aufgabe nicht richtig verstanden zu haben, dann fragen Sie uns.

Wenn Sie fertig sind, gehen Sie mit dem Mauszeiger in dieses Fenster hier, wo dieser Text steht, und tippen ein `q` ein (q=quit).

1. Aufgabe: Text hoch und runter

Gehen Sie in dieses Fenster und benutzen Sie den Abwärtspfeil rechts unten auf der Tastatur, um ein bißchen mehr von diesem Text zu sehen.

Benutzen Sie dann den Aufwärtspfeil, um wieder an den Anfang zurückzukommen.

Tippen Sie nun ein paar Mal auf die Taste `d`. Was hat das für einen Effekt? Tippen Sie nun solange auf `u`, bis Sie wieder den Anfang sehen.

Diesen Effekt nennt man in Computer-Deutsch runter und rauf „scrollen“.

Frage: Von welchem englischen Wortpaar leiten sich vermutlich die Kürzel `d` und `u` ab?

2. Aufgabe: Fenster spielen

Auf dem Bildschirm sollten 5 Fenster sein: `xb`, `xclock`, `console` und 2-mal `xterm`. Ein oder zwei Fenster sind teilweise von den anderen verdeckt.

Gehen Sie auf ein teilweise verdecktes Fenster, und drücken Sie zweimal die Taste `Front`. Gehen Sie nun in ein unverdecktes Fenster, und tippen Sie noch einmal `Front`. Wozu ist `Front` also da?

Gehen Sie mit dem Mauszeiger auf den grauen Balken mit dem Schriftzug `xterm` (nicht auf das kleine Viereck mit dem Dreieck darin!). Der Mauszeiger muß ein schräger Pfeil werden. Drücken Sie nun die linke Taste herunter, und bewegen Sie die Maus ein kleines Stück in irgendeine Richtung. Lassen Sie dann die Taste wieder los. Was ist passiert?

Gehen Sie nun mit dem Mauszeiger auf einen der vier Randwinkel eines Fensters. Wenn Sie den Randwinkel getroffen haben, wird der Mauszeiger ein Kreis mit Mittelpunkt. Drücken Sie nun die linke Maustaste herunter, und bewegen Sie die Maus ein wenig hin und her mit heruntergedrückter Taste. Was passiert, wenn Sie die Taste wieder loslassen?

3. Aufgabe: Fenstermenüs

Gehen Sie mit dem Mauszeiger auf das kleine Viereck mit dem Dreieck darin in einem der Fenster oben links. Klicken Sie nun die rechte Maustaste kurz an. Gehen Sie mit dem Mauszeiger auf den Schriftzug `Full Size`, und klicken Sie die linke Maustaste an. Wiederholen Sie das, nur klicken Sie jetzt `Restore Size` statt `Full Size` an. Wiederholen Sie die ganze Prozedur mit einem anderen Fenster.

Wählen Sie jetzt `Close` in diesem Menü, und versuchen Sie danach das Fenster wieder so herzustellen, wie es vorher war.

Gehen Sie wieder auf das Viereck mit dem Dreieck oben links, klicken die linke Maustaste statt

der rechten an, und holen Sie sich wie eben das Fenster wieder.

Gehen Sie nun in das andere Fenster `xterm` (nicht dieses!), und schließen Sie das Fenster, indem Sie wieder das Viereck oben links mit rechter Maustaste anklicken und nun `Quit` wählen.

Klicken Sie noch ein letztes Mal dieses Viereck bei irgendeinem Fenster mit der rechten Maustaste an, gehen Sie mit dem Mauszeiger irgendwohin in den Hintergrund, und klicken Sie die linke Maustaste an. Ist es also notwendig, eine Option auszuwählen, wenn man aus Versehen dieses Menü angeklickt hat?

Freispiel: Spielen Sie noch ein bißchen auf diese Art mit Ihren Fenstern herum, aber benutzen Sie nicht die Option `Quit`!

4. Aufgabe: einfache Kommandos

Bewegen Sie den Mauszeiger in den Hintergrund, klicken Sie die rechte Maustaste an, gehen Sie auf das kleine Dreieck rechts neben `Programs`, lassen weiterhin die Maustaste gedrückt, und lassen Sie sie erst auf `xterm...` wieder los. In dem jetzt neugeöffneten Fenster `xterm` setzen Sie nun ein paar Kommandos ab. Nach jedem der folgenden Kommandos müssen Sie jeweils die Taste `Return` drücken, um das Kommando abzuschicken. Sehen Sie sich die Ausgabe jedes dieser Kommandos an, und versuchen Sie sie zu verstehen, bevor Sie das nächste Kommando absetzen.

Achtung: Bedenken Sie, daß unsere Computer nur Englisch verstehen!

```
cal
cal 1996
cal 10 1996
cal 10 96
cal 13 1996
cal blabla
Cal
Cal 10 1995
CAL
date
factor 360
```

5. Aufgabe: Interaktives Kommando

Viele Kommandos sind nicht gleich wieder zu Ende, sondern „kommunizieren“ mit dem Benutzer. Daß Sie mit diesem Programm „sprechen“ und nicht mit UNIX selbst, erkennen Sie daran, daß der Prompt fehlt. Der Prompt sollte bei Ihnen momentan lauten:

Benutzername `@` *Rechnername* `~ <Zahl> %`

Ein solches interaktives Programm ist z.B. `bc`. Geben Sie einmal folgende Zeilen nacheinander im `xterm` ein, jeweils abgeschlossen mit `Return`:

```
bc
17+4
(1+2)*(17+8)
2^3
2^4
1/10
5/3
sqrt(10)
```

```
scale=10
1/10
5/3
sqrt(10)
quit
```

Was ist der Effekt der Zeile `scale=10`, und welcher Wert für `scale` ist voreingestellt?

6. Aufgabe: Programm mit eigenem Fenster

Geben Sie das Kommando `xcalc` ein. Spielen Sie ein bißchen mit den Funktionen von `xcalc` herum, indem Sie die Knöpfe in diesem Fenster mit der Maus drücken, so wie man auf die Knöpfe eines wirklichen Taschenrechners drücken würde. Wiederholen Sie danach diese Spielereien, indem Sie nun die Rechnungen mit der Tastatur schreiben, während der Mauszeiger im `xcalc`-Fenster ist, also z.B. `3+4=`

Was ist mit dem `xterm`, in dem Sie `xcalc` aufgerufen haben?

Schließen Sie danach das Fenster `Calculator` wieder, wie oben beschrieben mit Menüoption `Quit`.

7. Aufgabe: Programmaufruf mit `&`

Anstelle von `xcalc` geben Sie jetzt `xcalc&` ein. Geben Sie danach irgendeinen anderen Befehl in demselben `xterm` ein. Was passiert, wenn Sie noch einmal `xcalc` aufrufen, ohne vorher das alte `xcalc` mit `quit` zu beenden?

8. Aufgabe: Email

Tippen Sie jetzt das Kommando `elm` ein. Das Programm `elm` stellt Ihnen jetzt zwei Fragen. Auf beide Fragen hin tippen Sie ein `y` OHNE `Return`. Danach sehen Sie ein Menü mit mehreren Emails. Mit den beiden Pfeiltasten für hoch und runter können Sie sich eine Email aussuchen. Suchen Sie sich eine aus, und drücken Sie `Return`. Jetzt können Sie die Mail lesen.

Unten am Rand sehen Sie entweder einen Doppelpunkt oder eine Erklärung, wie Sie aus der Mail wieder herauskommen. Im zweiten Fall folgen Sie dieser Erklärung einfach. Im ersten Fall war die Mail länger als das Fenster, und der `elm` hat `less` aufgerufen, um die Mail anzuzeigen. Wie Sie aus `less` herauskommen, wissen Sie inzwischen.

Lesen Sie auch die anderen Mails. Beachten Sie ihren Inhalt!

Welchen Sinn haben die einzelnen Spalten in der Auflistung aller eingegangenen Mails?

Was ist mit dem Fenster namens `xb` in der Zwischenzeit passiert? Welchen Zweck könnte das Fenster also haben?

Wenn Sie wieder das Menü aller eingegangenen Mails sehen, sehen Sie unten auch ein paar weitere Erklärungen. Nutzen Sie diese Information, um den `elm` zu verlassen. Dabei stellt der `elm` wieder eine Frage. Beantworten Sie die fürs erste einfach mit `n`.

9. Aufgabe: `whatis` und `apropos`

Geben Sie folgende Zeilen im `xterm` ein:

```
whatis cal
whatis date
```

```
whatis bc
whatis xcalc
whatis xterm
whatis elm
```

Welche Aufgabe hat also das Kommando `whatis`?

Geben Sie jetzt folgende Zeile ein:

```
apropos calculator
```

Welche Aufgabe hat demgemäß das Kommando `apropos`?

10. Aufgabe: lange Programmausgaben

Nun geben Sie folgenden Befehl ein:

```
apropos date
```

Am linken Rand des `xterm` ist ein vertikaler Balken, der unten grau und oben weiß ist. Gehen Sie auf den grauen Teil, drücken Sie die mittlere Maustaste herunter, und schieben Sie den Mauszeiger mit heruntergedrückter Taste nach oben. Was passiert?

Sie können sich das Ergebnis von `apropos date` auch durch das Programm `less` anzeigen lassen. Dazu geben Sie ein:

```
apropos date | less
```

Was passiert, wenn Sie nun wie eben wieder an diesem Balken links herumspielen?

Mit `q` kommen Sie auch hier wieder aus `less` heraus.

11. Aufgabe: ManPages

Mit dem Kommando

```
man cal
```

bekommt man Informationen über das Kommando `cal`, die sogenannte „Manual Page“ (ManPage).

ACHTUNG: Das Kommando `man` nutzt zur Anzeige wieder das Programm `less`, das Sie aufgerufen haben, um diesen Text zu lesen. Deshalb kommen Sie auch auf die gleiche Art aus der ManPage wieder heraus: mit einem `q` für `quit`. Auch in der ManPage können Sie mit den beiden Pfeiltasten und mit den Tasten `d` und `u` arbeiten.

Finden Sie mit Hilfe der ManPage zu `cal` heraus, welche Tage einmal ausgelassen wurden, um die Schaltjahrsrechnung, die außer Tritt geraten war, wieder anzupassen.

Suchen Sie in der ManPage zu `bc` die Stelle, wo beschrieben steht, mit welchem Wert `scale` voreingestellt ist.

Finden Sie mit Hilfe der ManPage zu `less` heraus, wieviel durch `u` oder `d` „gescrollt“ wird.

Was hat die Ausgabe von `whatis` *Kommando* mit der ManPage von *Kommando* zu tun?

Gibt es einen Zusammenhang zwischen den Ausgaben von `whatis` und den Ausgaben von `apropos`?

12. Aufgabe: `koala`

Geben Sie folgende Zeilen in einem `xterm`-Fenster ein:

```
koala
info
info suche
```

Lesen Sie sich die Ausgaben genau durch, und versuchen Sie das Buch zu finden, das im Titel „UNIX“, „C“ und „Internet“ enthält. Was für einen Ausleihstatus hat das Buch?

Lassen Sie sich nun durch `koala` alle Bücher auflisten, an denen ein gewisser oder eine gewisse D. Ritchie mitgewirkt hat. Wie lautet der Vorname des Autors D. Ritchie, der im Zusammenhang mit dem UNIX/C++-Praktikum interessant ist?

Beenden Sie danach die Verbindung zum `koala`-System wieder. (Wie? Steht unter `info`!)

13. Aufgabe: `letters`

Das Programm `letters` ist ein Spiel, mit dem man seine Fingerfertigkeit beim Tippen verbessern kann. Wenn Sie `letters` aufrufen, fallen Wörter vom Himmel, und man muß jedes Wort korrekt abtippen, bevor es unten ist. Macht man zwischendurch einen Fehler, muß man das Wort von vorne anfangen. Ein einziges Wort dürfen Sie durchkommen lassen. Beim zweiten Wort ist das Spiel zu Ende, und man erhält eine Punktzahl.

Danach sehen Sie die „Top Ten“.

Aufgabe: unter die Top Ten zu kommen. :-)

14. Aufgabe: Antworten

Wenn Sie alle Aufgaben erledigt haben, sehen Sie sich mit

```
less ~weihe/unix96/Antworten1
```

die Antworten zu den wichtigsten Fragen oben an. Vergleichen Sie sie mit Ihren eigenen Antworten.

Nicht vergessen: hinterher ausloggen!!!

1 Antworten zum ersten Aufgabenblatt

1. Aufgabe: Text hoch und runter

`d` = down, `u` = up

2. Aufgabe: Fenster spielen

`Front` sorgt dafür, daß das Fenster, in dem sich gerade der Mauszeiger befindet, nach vorne geholt wird, das heißt unverdeckt ist, oder wieder in den Hintergrund geschoben wird.

Die einzelnen Teile des grauen Randes eines Fensters haben unterschiedliche Funktionen, die man durch die linke Maustaste aktivieren kann. Der Namensbalken oben verschiebt das Fenster, die vier Eckwinkel ändern die Größe.

3. Aufgabe: Fenstermenüs

Ein Menü ist eine Liste von Möglichkeiten (Optionen genannt), aus der man wählen kann.

Das graue Viereck mit Dreieck drin bietet ein Menü, mit dem das Fenster ebenfalls geändert werden kann.

Durch Anklicken von `Close` verschwindet das Fenster vom Bildschirm, und statt dessen erscheint am Bildschirmrand ein kleines Fenster (Icon). Auf diesem Fenster kann man dasselbe Menü durch die rechte Maustaste bekommen und damit das Fenster wieder zurückbekommen.

Durch Iconifizieren kann man seinen Bildschirm etwas aufräumen.

Die linke Maustaste anstelle der rechten wählt sofort die oberste Option aus (also `Close`).

Durch Mausklick außerhalb des Fensters kann man das Menü wieder loswerden, wenn man es sich anders überlegt hat.

4. Aufgabe: einfache Kommandos

UNIX unterscheidet Groß- und Kleinbuchstaben.

Es gibt Kommandos mit und solche ohne zusätzliche Argumente. Z.B. hat `date` kein Argument, während `cal` ein oder zwei Argumente bekommen kann. Argumente müssen Sinn machen, sonst gibt es eine Fehlermeldung.

Das `MET` in der Ausgabe von `date` steht für middle-European time.

Das Kommando `factor` erwartet eine natürliche Zahl und zerlegt sie in Primfaktoren.

5. Aufgabe: Interaktives Kommando

`bc` ist ein flexibler Taschenrechner, dem man auch komplizierte Formeln geben kann.

`scale=10` hat den Effekt, daß die arithmetische Genauigkeit der Rechnungen zehn Nachkommastellen beträgt. Voreingestellt ist `scale=0`, das heißt rein ganzzahlige Rechnungen.

6. Aufgabe: Programm mit eigenem Fenster

`xcalc` ist praktisch ein Taschenrechner. Man kann aber auch einfache Formeln direkt eingeben.

Das `xterm` ist durch den Aufruf von `xcalc` blockiert und wird erst wieder verwendbar, wenn `xcalc` beendet wird.

7. Aufgabe: Programmaufruf mit `&`

Hängt man an ein Kommando ein `&` an, dann läuft das entsprechende Programm im Hintergrund ab und blockiert nicht das `xterm`.

Grobe Faustregel: Programme, die eigene Fenster öffnen, sollte man in den Hintergrund schicken, andere nicht.

8. Aufgabe: Email

Die einzelnen Spalten der Auflistung aller Mails haben folgende Bedeutung: Nummer (sortiert nach Zeitpunkt der Abschickung), Datum, Absender, Anzahl Zeile und „Überschrift".

Das Fenster `xb` zeigt an, ob Mail für einen Benutzer eingegangen ist, nachdem dieser das letzte Mal Mail gelesen.

9. Aufgabe: `whatis` und `apropos`

`whatis` bekommt einen Kommandonamen als Argument und gibt dafür eine Kurzbeschreibung aus.

`apropos` bekommt einen Suchbegriff als Argument und sucht nach Kurzbeschreibungen, die den Suchausdruck enthalten.

10. Aufgabe: lange Programmausgaben

Der weiß-graue Balken links an einem Fenster ist der sogenannte „Scroll-Bar". Das `xterm`-Fenster hat viel mehr Inhalt, als man im Fenster selbst auf einen Blick sehen kann. Das Verhältnis zwischen grau und weiß im Scroll-Bar zeigt, wie groß der ausgeblendete Anteil ist.

Mit dem Scroll-Bar kann man im Fenster rauf und runterscrollen, um sich einen anderen Ausschnitt des Textes anzusehen.

Das Programm `less` belegt nur die alleruntersten Zeilen des `xterm`-Fensters für seine Zwecke. Diese Zeilen werden überschrieben. Mit den Scroll-Möglichkeiten, die `less` anbietet (Pfeiltasten, `d`, `u`) scrollt man im von `less` angezeigten Text hin und her, mit der Scroll-Möglichkeit des `xterm`-Fensters (Scroll-Bar) scrollt man dagegen im Inhalt des `xterm`-Fensters hin und her.

11. Aufgabe: ManPages

3.-13. September 1752 sind ausgelassen worden.

Voreinstellung von `scale` im `bc` siehe Antwort zu Aufgabe 5.

`u` und `d` scrollen im `less` die halbe Größe des Fensters.

Die Ausgabe von `whatis` gibt im wesentlichen den Abschnitt `NAME` der ManPage aus.

Die Ausgabe von `apropos` zu einem Kommando ist genau die Ausgabe, wenn man `whatis` mit diesem Kommando aufruft.

12. Aufgabe: `koala`

Das Programm `koala` stellt eine Verbindung zu dem zentralen Rechner in der Bibliothek her, auf dem das Programm läuft, das man auch an den `koala`-Terminals in der Bibliothek benutzen kann. Das Programm ist zwar dasselbe, aber die sogenannte „Schnittstelle“ ist eine andere.

Das Buch, dessen Titel „UNIX“, „C“ und „Internet“ enthält, findet man mit der Suchanfrage

```
*su ti=unix c internet
```

Alle Bücher von allen Leuten namens D. Ritchie bekommt man mit

```
*su au=ritchie,d?
```

Der Vorname, nach dem gefragt war, lautet Dennis.

2 Heutiges Generalthema: `Emacs`

Bearbeiten Sie wie beim letzten Mal die folgenden Aufgaben. Es ist für den weiteren Erfolg im Praktikum nicht notwendig, daß Sie alle Aufgaben bearbeiten. Aber es ist sinnvoll, Aufgaben, die Sie nicht innerhalb des Termins geschafft haben, später nachzubearbeiten.

Sie sollten das File Antworten2 **unbedingt** erst dann lesen, wenn Sie alle Aufgaben hier bearbeitet haben, sonst geht der Übungseffekt verloren.

1. Aufgabe: File `.emacs`

Jetzt legen Sie zum ersten Mal ein *File* (eine *Datei*) selbst an. Allerdings basteln Sie den Inhalt nicht selbst zusammmen, sondern übernehmen ihn einfach aus einem anderen File. Geben Sie dazu das folgende Kommando ein:

```
cp ~weihe/.emacs ~
```

2. Aufgabe: Texte basteln

Geben Sie das Kommando `emacs dummy&` ein. Das Wort `dummy` ist der Name des Files, das wir jetzt erstellen. (Wiederholung vom 1. Übungsblatt: Wozu war noch das `&` gut?)

Schreiben Sie mit der Tastatur ein paar Zeilen in diesem Fenster voll. Benutzen Sie dabei zunächst **nur** die weißen Tasten, nicht die hellgrauen. Ausnahme: Eine neue Zeile machen Sie durch die hellgraue Taste `Return` auf.

Das schwarze Rechteck, an dem das jeweils nächste eingetippte Zeichen erscheint, heißt *Cursor*.

Bewegen Sie den Mauszeiger irgendwo in die Mitte Ihres Textes und drücken Sie die linke Maustaste herunter. Was passiert mit dem Cursor?

Spielen Sie an allen vier Pfeiltasten herum, um den Cursor in Ihrem Text beliebig hin und her zu bewegen.

Positionieren Sie den Cursor in der Mitte irgendeiner Zeile und drücken Sie `Return`. Was passiert? Wenn Sie jetzt ein Zeichen eingeben, was passiert mit dieser Zeile?

Positionieren Sie den Cursor mit dem Mauszeiger hinter irgendein Zeichen und drücken Sie die `Backspace`-Taste. Was passiert?

Klicken Sie mit der linken Maustaste das `Edit`-Menü an und gehen Sie auf `Undo`. Was passiert, wenn Sie auf diese Art wiederholt auf `Undo` gehen? Wozu ist `Undo` also da?

3. Aufgabe: File speichern, `emacs` verlassen, File wieder öffnen

Klicken Sie mit dem Mauszeiger (linke Maustaste) das `emacs`-Menü `File+` an und gehen Sie auf `Save Buffer`. Rufen Sie nun `less dummy` in einem `xterm` auf. Was ist das Ergebnis?

Gehen Sie nun im Menü `Files` des `emacs`-Fensters auf `Exit Emacs`.

Rufen Sie erneut `emacs dummy&` auf. Was ist das Ergebnis?

4. Aufgabe: neues File eröffnen

Gehen Sie im `Files`-Menü des `emacs` auf `Open File...`. Dann erscheint in der untersten Zeile eine *Tilde* `~` und ein Slash `/`. Löschen Sie den Slash (nicht die Tilde!) und tippen Sie dann

ein (nebst Return). Was ist das Ergebnis? Was steht in der untersten Zeile? Was passiert, wenn Sie versuchen, den Inhalt des Buffers zu ändern?

Gehen Sie noch einmal auf Open File.... Jetzt wird ~weihe/ angezeigt statt ~/. Das ist ein Service vom emacs, nämlich daß immer der Bereich angezeigt wird, in dem das letzte File geöffnet wurde. Tippen Sie nun Au und drücken dann die Taste Tab. Was passiert? Tippen Sie nun eine 2 und Return. Welchen Service bietet also die Tab-Taste beim Öffnen von Files?

5. Aufgabe: Größere Manövrierschritte

Gehen Sie im File+-Menü mehrmals hintereinander auf Scroll forward und danach wieder auf Scroll backward.

Wiederholung von Aufgabe1: Was heißt „Scrollen“?

Gehen Sie nun im File+-Menü auf Goto end of buffer und danach auf
Goto beginning of buffer.

Gehen Sie nun im File+-Menü auf Goto line.... Dann erscheint in der untersten Zeile Goto line: . Tippen Sie 100 nebst Return ein. Wo sind Sie jetzt im File gelandet?

6. Aufgabe: Größere Manipulationen eines Files

Öffnen Sie wieder Ihr File dummy im emacs.

Selektieren Sie ein paar aufeinanderfolgende Zeichen in einer Zeile, indem Sie mit dem Mauszeiger auf das erste Zeichen gehen, die linke Maustaste herunterdrücken und mit herunter gedrückter Maustaste hinter das letzte Zeichen gehen, das Sie selektieren wollen. Gehen Sie nun im Edit-Menü auf Copy.

Jetzt positionieren Sie den Cursor an eine völlig andere Stelle in einer anderen Zeile und gehen im Edit-Menü auf Paste Most Recent. Was ist der Effekt?

Wiederholen Sie diese ganze Prozedur mit einer anderen Zeichensequenz, aber nun gehen Sie auf Cut anstelle von Copy. Was ist der Unterschied? Wie würden Sie also einen größeren Textabschnitt löschen?

Der emacs hat sogar eine Liste von „alten“ Inhalten des Paste-Buffers. Gehen Sie im Edit-Menü auf Select and Paste und wählen Sie durch Anklicken einen alten Paste-Buffer-Inhalt aus. Was passiert, wenn Sie nun auf Paste gehen?

Wiederholen Sie die ganze Prozedur mit einer Zeichenkette, die nun nicht mehr in einer Zeile steht, sondern über mehrere Zeilen geht, und die drei Zeichen vor Zeilenende zu Ende ist. Woran erkennt man einen Zeilenumbruch in der Liste aller alten Paste-Buffer? Warum wird das so gemacht?

7. Aufgabe: Text suchen und ersetzen

Kopieren Sie das File Aufgabe1 aus dem Bereich von weihe wie in der ersten Aufgabe oben beschrieben und öffnen Sie Ihre Kopie im emacs. Gehen Sie im Search-Menü auf Search und geben Sie daraufhin cal ein (nebst Return). Was passiert? Was passiert, wenn Sie diese Prozedur wiederholen? Was passiert, wenn Sie nach wiederholtem Search auf Search Backwards statt dessen gehen?

Gehen Sie wieder an den Anfang des Files zurück. (Mir fallen mindestens fünf grundverschiedene

Möglichkeiten hierzu ein. Ihnen auch? Manche dieser Möglichkeiten sind allerdings nicht sehr schlau.)

Gehen Sie nun im `File+`-Menü auf `Always Replace`. Nun erscheint unten der Text `Replace-string:` Tippen Sie zuerst

```
cal
```

nebst `Return` und danach

```
california
```

nebst `Return`. Was hat dieses Kommando nun mit Ihrem File `Aufgabe1` angestellt? Was hat es insbesondere mit den Zeichenketten `Cal` und `CAL` angestellt?

Jetzt ersetzen Sie in Ihrem File Aufgabe1 das Wort "'Gehen" durch das Wort „Laufen", aber nur dort, wo es am Anfang eines neuen Absatzes steht, sonst nicht. Dazu gehen Sie wieder an den Anfang Ihrer Kopie des Files Aufgabe1 und dann im `Search`-Menü auf `Query Replace`. Jetzt tippen Sie zuerst `Gehen` nebst `Return` ein, danach `Laufen` nebst `Return`.

Danach springt der Cursor zum ersten Vorkommen von `Gehen`. Nun tippen Sie einen von drei Buchstaben:

`y` = `yes`, wenn dieses Vorkommen von „Gehen" durch „Laufen" ersetzt werden soll.

`n` = `no`, wenn „gehen" stehenbleiben soll.

`q` = `quit`, wenn Sie den Ersetzungsvorgang ganz abbrechen wollen.

Jedesmal, wenn Sie `y` oder `n` eintippen, springt der Cursor zum nächsten Vorkommen von „Gehen". Gibt es keines mehr, steht in der untersten Zeile `done`, und der Ersetzungsprozeß ist zu Ende. Benutzen Sie `Query Replace`, um die oben spezifizierte Aufgabe zu lösen.

8. Aufgabe: mehrere Buffer gleichzeitig

Gehen Sie jetzt im `File+`-Menü auf `Split window horizontally`. Jetzt haben Sie zwei File-Buffer anstelle von einem. Positionieren Sie den Cursor durch den Mauszeiger irgendwo in den rechten Buffer und öffnen Sie darin auf die oben beschriebene Weise das File `dummy` in Ihrem Bereich.

Kopieren Sie die 6. Aufgabe vollständig aus dem Buffer `Aufgabe1` in den Buffer `dummy`, so daß dieser Text zwischen der zweiten und dritten Zeile in `dummy` erscheint und nach oben und unten vom bisherigen Inhalt durch eine Leerzeile getrennt ist. Die eigentliche Kopieraktion geht genauso, wie oben beschrieben.

Schließen Sie den Buffer `Aufgabe1` wieder, so daß nur noch `dummy` das ganze `emacs`-Fenster ausfüllt.

(Wie das geht? Das finden Sie in den Menüs!)

Gehen Sie auf das `Buffers`-Menü und wählen Sie dort Aufgabe1 aus. Was ist also der Sinn dieses Menüs?

9. Aufgabe: `Help`-Menü

Gehen Sie im `Help`-Menü auf `Browse Manuals`. Sie stehen jetzt an der Spitze der `emacs`-eigenen Dokumentationshierarchie. Die einzelnen Punkte in dieser Hierarchie werden `nodes` genannt. Im ersten Absatz ist eine kurze Einhilfe, und darunter ist ein Menü, das mit der Option `Info` beginnt.

Lesen Sie den ersten Absatz und lassen Sie sich die Liste aller INFO Kommandos anzeigen. Wie wählt man also eine Option im Menü unten aus? Wie kommt man aus dieser Liste aller INFO Kommandos wieder zurück auf die erste Seite?

Wählen Sie so die Option `Emacs` aus. Jetzt bekommen Sie eine neue Seite der `emacs`-eigenen Dokumentation, die den `emacs` selbst beschreibt. Diese Dokumentation ist ihrerseits wieder hierarchisch gegliedert, und die Gliederungspunkte auf der nächsten Ebene erscheinen wieder als ein Menü. Benutzen Sie diese Menüpunkte, um folgende Fragen zu beantworten.

- Was sagt das „Glossary“ zur `Echo Area`?
- Was sagt das „Glossary“ zu `Scroll Bar`?
- Tippen Sie mit der mittleren Maustaste auf `Note Scroll Bars`. Was passiert? Tippen Sie jetzt ein `?`, und nutzen Sie die angezeigte Information, um mit einem einzigen weiteren Tastendruck wieder zurück zum „Glossary“-Eintrag für `Scroll-Bar` zu kommen.
- Warum ist der `emacs` – nach Meinung seiner Erfinder – ein „display“, „real-time“, „advanced“, „self-documenting“ und „customizable“ Editor?
- Wie heißt die Zeile, die unten in einem `emacs`-Buffer weiß-auf-schwarz erscheint?

Hinweis: „Important General Concepts“

10. Aufgabe: Kontrollsequenzen im `emacs`

Böse Zungen meinen, `emacs` wäre die Abkürzung für

```
escape-meta-alt-control-
```

`shift`, weil man laufend diese Tasten drücken müßte. (In Wirklichkeit steht `emacs` für `Editing Macros`.) Richtig daran ist, daß man viel mit diesen Tasten arbeiten mußte, als es noch keine Menüs gab; hinter einigen Kommandos in den Menüs ist jeweils in Klammern die Kontrollsequenz aufgeführt, mit der man das Kommando ausführen kann. Richtig ist aber auch, daß man mit Hilfe solcher Kontrollsequenzen noch weit mehr Möglichkeiten hat als mit den Menüs allein.

Gehen Sie nun im `Help`-Menü auf `Command Apropos...`, und tippen Sie `line` ein. Was ist also die Funktion dieses Apropos? Wie läßt sich diese Funktion vergleichen mit der Funktion des UNIX-Kommandos `apropos`, die Sie bei der Bearbeitung von `Aufgabe1` kennengelernt haben?

Wie lautet das Kommando, mit dem man sich die Nummer der Zeile anzeigen lassen kann, in der der Cursor gerade steht? Rufen Sie dieses Kommando einmal auf. Dazu tippen Sie zuerst `Esc` und danach `x` und dann den Kommandonamen.

Versuchen Sie dann analog, ein Kommando zu finden, mit dem man sich mehr Informationen über die aktuelle „Cursor Position“ anzeigen lassen kann. Rufen Sie dieses Kommando genauso auf.

11. Aufgabe: Surfen im Internet

Rufen Sie das Kommando

```
netscape http://www.uni-konstanz.de&
```

auf. Nach kurzer Zeit erscheint ein neues Fenster „Netscape: License Agreement“. Lesen Sie darin ein bißchen herum und akzeptieren Sie dann die Lizenzbedingungen. Warten Sie einen Moment, dann erscheint ein Fenster `Netscape Error`. Drücken Sie auf `OK`. Sie sind jetzt im „World Wide Web (WWW)“, und zwar am Hauptknotenpunkt der Uni Konstanz. Machen Sie das Fenster etwas größer. Der Scroll Bar befindet sich an diesem Fenster auf der rechten Seite.

Jeder unterstrichene Schriftzug führt Sie durch Klick mit der linken Maustaste zu einem weiteren Knoten im WWW. Durch Druck auf `Back` kommen Sie immer einen Schritt wieder zurück. Suchen Sie folgende Informationen:

- Den Busfahrplan der Konstanzer Stadtwerke.
- Den Lageplan der Uni Konstanz und die dazugehörige Orientierungshilfe.
- Die Veranstaltungen im „Kulturladen“.
  (Disclaimer: Ich beziehe keine Provision vom Kulturladen. :-)
- Den Knotenpunkt Ihrer Fakultät.
- Den Knotenpunkt der Fachschaft Ihrer Fakultät.
- Den Knotenpunkt der Informatik.
- Vom Knotenpunkt Informatik aus die Ankündigung des UNIX/C++-Praktikums. (Was gibt es also noch so alles an interessanten Veranstaltungen aus der Informatik?)
- Vom Knotenpunkt Informatik aus die Liste aller Leute, die mit dem Mail-Alias `unix96` angesprochen werden, über den Sie von mir schon Mail bekommen haben.
- Vom Knotenpunkt Informatik aus die Informationsangebote diverser Zeitungen, Zeitschriften und Radiosender.

12. Aufgabe: `netscape` konfigurieren

Gehen Sie im Menü `Options` auf das kleine Viereck neben der Option `Show location` und tippen es mit der linken Maustaste an. Was passiert? Machen Sie es noch einmal. Was passiert jetzt? Machen Sie dasselbe **einmal** mit dem Menüpunkt `Show directory buttons`, und lassen Sie die Directory Buttons von jetzt an ein für allemal weg.

Gehen Sie im Menü `Options` statt dessen auf `General Preferences...`. Sie bekommen jetzt ein Fenster, in dem Sie das Programm `netscape` nach Ihren Vorlieben konfigurieren können.

Unter `Fonts` sehen Sie, daß Proportional Font 12.0 voreingestellt ist. Wenn Sie 12.0 anklicken, erhalten Sie ein Menü von Schriftgrößen. Wählen Sie darin 24.0 aus. Was passiert? Stellen Sie dann den Font Style ein, der Ihnen am meisten behagt.

Gehen Sie zurück auf `Appearance` und ändern Sie die `Home Page location` ab in die Location des zentralen Knotens der Uni Konstanz oder die Location Ihrer Fakultät oder wo immer Sie sonst jeden weiteren Surf im WWW starten lassen wollen. Diese Location sehen Sie im rosa Balken oben, wenn Sie gerade auf diesem Knoten stehen. Beachten Sie, daß Sie exakt diesen Text angeben.

Schließen Sie nun `General Preferences` durch Drücken auf `OK`, und gehen Sie jetzt im Menü `Options` auf `Save Options`. Jetzt erst sind Ihre Änderungen dauerhaft gespeichert, so daß sie auch beim nächsten Aufruf von `netscape` Wirkung haben.

Gehen Sie jetzt aus `netscape` wieder heraus und rufen Sie auf:

```
netscape&
```

Nachdem das Netscape-Logo erscheint, klicken Sie mit der linken Maustaste mitten in das Fenster hinein. Jetzt müßte der WWW-Knoten erscheinen, den Sie unter `Home Page Location` eingetragen haben. Falls nicht, haben Sie etwas falsch gemacht und müssen noch einmal daran herumbasteln.

13. Aufgabe: Weitersurfen im Internet

Gehen Sie jetzt aus der Uni Konstanz heraus und deutschland- oder sogar weltweit auf Suche. Wie Sie vielleicht schon gemerkt haben (und wie etliche Surfer aus ganz Deutschland ebenfalls schon bemerkt haben), eignet sich der Knotenpunkt der Informatik sehr gut als Startrampe - vor allem dank Ulrik Brandes, unserem „Webmaster“ (Applaus!), der dafür die Hauptarbeit geleistet hat.

Merke: Echte Hacker surfen, bis die Bib dichtmacht. :-)

Und sie sprechen auch nicht von Knotenpunkten, sondern von Pages oder URLs (Uniform Resource Locations).

2 Antworten zum Generalthema: emacs

1. Aufgabe: File .emacs

cp ist das Kommando, mit dem man den Inhalt eines Files in ein anderes File kopieren kann (cp=copy).

~weihe ist der Bereich, aus dem das File kopiert werden soll. Das ist der persönliche Bereich des Benutzers mit Kennung „weihe“.

.emacs ist der Name des Files im Bereich von weihe, das kopiert werden soll. Der *Slash* / trennt also den Bereich vom Filenamen selbst.

~ steht für ihren eigenen Bereich. Das kopierte File heißt auch bei Ihnen .emacs.

Das File .emacs ist das sogenannte *Konfigurationsfile* für das Programm namens emacs. Wenn man emacs startet, schaut es zuerst nach, ob es ein File dieses Namens in Ihrem Bereich gibt. Falls ja, liest es dieses File und richtet sich bei allen Voreinstellungen nach dessen Inhalt.

Merkregel: Solche Konfigurationsfiles fangen per Konvention immer mit einem Punkt an.

Ohne diese Kopierarbeit würde der emacs auch funktionieren, wäre aber nicht so komfortabel, wie wir ihn im weiteren benutzen.

2. Aufgabe: Texte basteln

Wiederholung: Das & sorgt dafür, daß das xterm nicht blockiert wird. Als allgemeine Faustregel hatten wir gesagt, daß ein Programm, das ein eigenes Fenster öffnet, mit & gestartet werden soll, also auch der emacs.

Mit Undo kann man sich sehr bequem korrigieren, wenn man sich total „vergaloppiert“ hat.

3. Aufgabe: File speichern, emacs verlassen, File wieder öffnen

Emacs hält ein File in seinem „Buffer“ (Puffer). Mit Save Buffer wird der momentane Inhalt des Buffers in das File zurückgeschrieben, und der alte Inhalt geht verloren.

4. Aufgabe: neues File eröffnen

Man kann in einen laufenden emacs ein neues File laden. Dazu muß man den Namen des Files angeben und auch den Bereich, aus dem das File genommen werden soll (in Aufgabe 4: Bereich ~weihe). Der emacs bietet einige Möglichkeiten, sich dabei lästige Tipparbeit zu sparen.

6. Aufgabe: Größere Manipulationen eines Files

Mit Copy oder Cut speichert man den selektierten Textteil in einem internen Zwischenlager des emacs ab (der sogenannte *Paste-Buffer*). Der Inhalt dieses Zwischenlagers wird durch ein neuerliches Copy oder Cut überschrieben. Durch Paste wird der momentane Inhalt des Paste-Buffers an der momentanen Cursor-Position eingefügt. Will man eine Textpassage löschen, wendet man nur Cut an.

Der emacs speichert die alten Paste-Buffer-Inhalte weiterhin. Im Menü kann man sich jeden alten Inhalt herauspicken. Da jeder alte Inhalt genau eine Zeile im Menü ist, muß Zeilenumbruch irgendwie anders angezeigt werden, daher dieses merkwürdige Zeichen für Zeilenumbruch und die drei Punkte als Abkürzung.

7. Aufgabe: Text suchen und ersetzen

Fünf Möglichkeiten, im `emacs` an den Anfang eines Files zu kommen:

- Scroll backward
- Maustaste mit Aufwärtspfeil
- Goto line: 1
- `Emacs` beenden und neu starten
- File noch einmal in den `emacs` laden, ohne ihn zwischenzeitlich zu beenden.

Im `emacs` kann man eine Textpassage durchgängig durch eine andere im ganzen File ersetzen. Das kann man so einrichten, daß man bei jeder Ersetzung erst gefragt wird, oder - wenn man genau weiß,was man tut! - daß man ohne Nachfrage alles automatisch ersetzt. In beiden Fällen versucht `emacs` die Groß- und Kleinschreibung einzuhalten.

8. Aufgabe: mehrere Buffer gleichzeitig

Das `Buffers`-Menü zeigt alle Files an, die momentan vom `emacs` geöffnet sind. Das sind nicht nur die, die momentan im Fenster zu sehen sind, sondern auch alle Files, die früher geöffnet worden sind und jetzt durch später geöffnete Files überdeckt worden sind. Man kann über das `Buffers`-Menü also bequem zwischen Files hin- und herspringen.

Mit `delete window` im `File+`-Menü kann man einen Buffer wieder schließen.

9. Aufgabe: `Help`-Menü

Man kann nähere Informationen über einen der Menüpunkte erhalten, indem man `m` = menu item tippt und danach den Menüpunkt eintippt. Mit `l` = last kann man sich wieder zurückhangeln.

`Node Scroll Bar` ist ein Verweis auf eine umfangreichere, detailliertere Erklärung. Auch hier kommt man mit `l` wieder zurück.

Die Antwort auf die letzte Frage zum `emacs`-Informationssystem findet sich unter `Screen`.

10. Aufgabe: Kontrollsequenzen im `emacs`

Das `apropos` im `emacs` bekommt einen Suchbegriff und listet alle Kommandos auf, die diesen Suchbegriff enthalten.

Mit `what-line` bekommt man die Zeilennummer des Cursors.

Mit `apropos cursor` bekommt man unter anderem `what-cursor-position` angeboten.

3 Generalthema heute: Email selbst schreiben

Bis jetzt haben Sie nur passiv Email (=Electronic Mail) empfangen. Heute werden Sie sich gegenseitig Emails übungshalber schreiben.

1. Aufgabe: Email vorbereiten

Kopieren Sie sich das File elmrc im Unterbereich `.elm` von `weihek` in den eigenen Unterbereich `.elm`.

Wiederholung von Aufgabenblatt 2: Wie kopiert man ein File aus einem Bereich in einen anderen?

Öffnen Sie dieses File im `emacs` und gehen Sie zur Zeile `editor = vi`. Die darunterstehende Zeile ist auskommentiert durch das `#`, d.h. hat keinen Effekt. Kommentieren Sie die Zeile `editor = vi` aus und geben Sie statt dessen der unmittelbar darunterstehenden Zeile einen Effekt.

Erklärung: Zum Email schreiben braucht man einen Editor. Der `emacs` ist einer, der `vi` ein anderer. Der `emacs` ist wesentlich bequemer. Die Setzung auf `emacsclient` anstelle von `emacs` besagt, daß für Email schreiben kein neuer `emacs` aufgerufen wird, sondern nur ein weiterer Buffer in einem schon laufenden `emacs` aufgemacht wird.

2. Aufgabe: PartnerInnen suchen

Suchen Sie sich rechts, links oder gegenüber ein oder zwei Leute, die ebenfalls soweit sind, und mit denen Sie sich jetzt Emails austauschen können.

3. Aufgabe: Email verschicken

Wenn Sie jetzt PartnerInnen gefunden haben, erkundigen Sie sich nach deren Benutzerkennungen und schicken Sie ihnen Mails mit dem `elm`. Vergewissern Sie sich vorher, daß Sie genau einen `emacs` laufen haben.

Dazu rufen Sie `elm` auf und tippen `m` OHNE `Return` ein. Jetzt werden Sie nach der Benutzerkennung gefragt, an die die Mail geschickt werden soll, danach nach einer kurzen Überschrift für die Mail. Geben Sie Ihrer Mail immer eine Überschrift, damit der Adressat seine Mails gut auseinanderhalten kann! Auf die dritte Frage hin (`Cc`) drücken Sie fürs erste einfach `Return`.

Jetzt müßte in diesem `xterm` erscheinen:

```
Invoking editor...Waiting for emacs...
```

und ein leerer Buffer erscheint im `emacs`, wo unten steht:

```
When done with a buffer: type C-x #.
```

Schreiben Sie den Inhalt Ihrer Mail in diesen Buffer, und gehen Sie dann im `File+`-Menü auf `Start or finish server edit` (dahinter steht in Klammern wieder das `emacs`-Kommando `C-x #`). Nun fragt `elm` Sie, ob Sie die Mail

- abschicken (`sent`),
- vergessen (`forget`) oder
- noch einmal im `emacs` anschauen oder redigieren (`edit`) oder

- noch einmal den „Header“ (Adressat, Subject u.ä.) ändern

wollen. Wenn Sie mit dem Inhalt unzufrieden sind, wählen Sie die dritte Möglichkeit. Wenn Sie etwas falsch gemacht haben bei Adressat oder Subject, wählen Sie die vierte. Danach schicken Sie die Mail ab mit `s`. Gehen Sie dann aus dem `elm` heraus.

Beobachten Sie dann, wie die Mail jeweils bei Ihren PartnerInnen ankommt. Woran erkennen Sie, daß die Mail angekommen ist?

4. Aufgabe: auf Email antworten

Wenn Sie jetzt in Aufgabe 3 Email von anderen bekommen haben, antworten Sie darauf. Dazu wählen Sie die Mail, auf die Sie antworten wollen, im Menü aller eingegangenen Mails aus. (Die Zeile ist dann schwarz unterlegt.) Drücken Sie `r` (r=reply) **ohne** `Return`. Auf die Frage `Copy message` geben Sie `y` ein, und das angebotene Subject akzeptieren Sie einfach durch `Return`. Beantworten Sie nun die Mail im `emacs`, indem Sie ein paar Kommentare zwischen den schon vorhandenen Zeilen einfügen und die Mail wie gehabt abschicken. Welchen Sinn mögen die `>` am Anfang einer Zeile haben?

5. Aufgabe: Mail forwarden

Forwarden Sie jetzt eine der Mails von `weihe` an Ihren Partner oder Ihre Partnerin. Dazu drücken Sie `f` (f=forward) anstelle von `r`. Versuchen Sie nun, sich selbst durch den `elm` soweit durchzuhangeln, daß die Mail an diesen Partner oder diese Partnerin abgeschickt wird. Diesmal sagen Sie dem `elm` auf seine Frage hin, daß Sie nicht mehr in die Mail hineinschauen wollen, bevor Sie sie abschicken.

6. Aufgabe: falsche Adresse

Schicken Sie jetzt eine Mail an User `dummy`. Was passiert?

7. Aufgabe: `Cc` und Mehrfachadressaten

Geben Sie jetzt zwei verschiedene Benutzerkennungen an, wenn `elm` Sie fragt, an wen die Mail geschickt werden soll. Trennen Sie diese beiden durch ein Komma. Auf die Frage `Cc:` geben Sie Ihre eigene Benutzerkennung an. Schicken Sie die Mail dann ab. Was passiert? Haben Sie eine Idee, wofür das Kürzel `Cc` stehen mag?

8. Aufgabe: Header manipulieren

Schicken Sie nun eine Mail an `weihek`, aber wenn `elm` Sie fragt, ob Sie die Mail wirklich abschicken wollen, tippen Sie zuerst `h` und dann `s` und basteln noch etwas an Ihrem Subject herum, bevor Sie die Mail endgültig abschicken.

9. Aufgabe: das doppelte `elm`chen

Rufen Sie in zwei verschiedenen `xterm` jeweils parallel `elm` auf. Was passiert beim zweiten Aufruf? Was mag der Grund sein, daß der `elm` zwei parallele Aufrufe abfängt?

10. Aufgabe: Wie ist Mail gespeichert?

Öffnen Sie im `emacs` das File namens `/var/mail/`*benutzerkennung*, wobei Sie für *benutzerkennung* Ihre eigene Benutzerkennung (**nicht** Ihr Passwort!) einsetzen. Was ist der Inhalt dieses Files? Können Sie den Inhalt einer Mail vom Header dieser Mail unterscheiden? Was passiert, wenn Sie für *benutzerkennung* die Benutzerkennung von jemand anders einsetzen? Warum wohl?

Kopieren Sie das File `/var/mail/`*benutzerkennung* (wieder mit Ihrer Kennung) in Ihren eigenen Bereich. (Wie? Steht in File `Aufgabe2`.)

Achtung: Ändern Sie nichts an dieser Kopie in Ihrem Bereich! Das ist Ihre Referenzkopie, mit der Sie hinterher wieder alles so wiederherstellen, wie es vorher war.

Basteln Sie jetzt im Originalfile `/var/mail/`*benutzerkennung* am Inhalt einer Mail (nicht am Header!) herum. Was passiert, wenn Sie Ihre Änderungen abspeichern, danach den `elm` aufrufen und sich diese Mail anschauen?

Was passiert, wenn Sie z.B. die Header-Zeile, die mit `From:` beginnt, bei einer Mail löschen und danach wieder den `elm` aufrufen und sich die Mail anschauen wollen?

Speichern Sie jetzt Ihr File `~/`*benutzerkennung* wieder zurück, um den alten Zustand wiederherzustellen.

11. Aufgabe: eingegangene Mail löschen

Gehen Sie jetzt wieder im `elm` in das Menü aller eingegangenen Mails. Gehen Sie auf jede Mail, die Sie nicht mehr brauchen, und tippen Sie `d`. Damit haben Sie diese Mail zum Löschen vorgemerkt. Wie ändert sich die Anzeige dieser Mail? Kommen Sie mit den Pfeiltasten wieder auf diese Mail zurück?

Tippen Sie jetzt die Nummer einer solchen „deleteten“ Mail ein (nebst `Return`) und dann ein `u`. Was ist jetzt passiert? Ist das Löschen einer Mail also unwiderruflich?

Gehen Sie jetzt aus dem `elm` wieder heraus. Beantworten Sie die Frage, ob Sie die Mails löschen wollen, mit `y` (aber **nur**, wenn Sie die Mails **wirklich** löschen wollen!).

Wenn Sie jetzt wieder in den `elm` gehen, sind die Mails jetzt wirklich gelöscht?

Wofür mögen die Kürzel `d`, `u` und `y` hier stehen? Stehen `u` und `d` für dasselbe wie im `less`?

12. Aufgabe: Protokoll aller ausgegangenen Emails

Oben haben wir gesehen, daß die eingegangene Mail schlicht und einfach in einem File abgespeichert ist (auch „Folder“ genannt). Was sagt die ManPage von `elm` dazu, wie man anstelle von `/var/mail/`*benutzerkennung* einen anderen Folder als Basis benutzen kann?

Benutzen Sie einmal als Basis den Folder `=sent`, d.h. das Wort sent mit einem Gleichheitszeichen davor. Was für Mails werden angezeigt? Sehen Sie sich im `less` einmal das File `~/Mail/sent` an. Wie interpretiert der `elm` also das Gleichheitszeichen?

Wenn man (wie in der 6. Aufgabe) aus Versehen eine falsche Adresse angegeben hat, muß man die ganze Mail noch einmal neu tippen, um sie nun an die richtige Adresse zu schicken? Oder wie kann man sich das Leben einfacher machen mit Hilfe von `=sent`?

Sehen Sie sich Ihr File `~/.elm/elmrc` an. Welche Zeile darin ist wohl verantwortlich dafür, daß alle herausgehenden Mails mitprotokolliert werden? Was müßte man wohl tun, wenn man ausgehende Mails nicht mitprotokollieren lassen möchte?

Benutzen Sie als Folder nun das File `~weihe/unix96/Aufgabe1`. Was passiert und warum?

13. Aufgabe: News lesen

Rufen Sie einmal `xrn&` auf. In dem Fenster, das nun erscheint, tippen Sie zunächst auf `Quit`. Sie sind nun im Hauptmenü des Programms `xrn`. Drücken Sie auf `Subscribe`. In dem erscheinenden Fenster geben Sie

```
de.sci.informatik.misc
```

ein und tippen nun auf `Last`. Was passiert mit der Anzeige im oberen Fenster? Wiederholen Sie das nun mit

```
rec.arts.ascii
```

anstelle von

```
de.sci.informatik.misc.
```

Klicken Sie nun mit der linken Maustaste `rec.arts.ascii` an und danach den Knopf `Read`. Sie lesen jetzt Ihren ersten News-Artikel, und zwar in der Newsgroup `rec.arts.ascii`. Wenn Sie den Artikel gelesen haben, drücken Sie den Knopf `Next unread`. Was passiert im oberen Fenster? Was passiert, wenn Sie auf `Quit` drücken und danach wieder nach `rec.arts.ascii` hineingehen?

Versuchen Sie, den Sinn von möglichst vielen vorkommenden Kürzeln zu raten: Was bedeuten wohl `de`, `sci`, `misc`, `rec`, `arts` und `ascii`?

WARNUNG: Bevor Sie selbst Artikel in irgendeiner Newsgroup „posten“, lesen Sie zuerst **alle** Files im Bereich `~weihe/unix96/news-start` gründlich und handeln Sie auch dementsprechend! (Diese Files habe ich natürlich aus den News abgefischt.) Wer diese Warnung nicht beherzigt, muß damit rechnen, daß er von anderen News-Lesern „angeflamt“ wird, bis er rote Ohren kriegt!

14. Aufgabe: `weihe/news-start`

Sehen Sie sich mit

```
ls ~weihe/unix96/news-start
```

an, was es für Files in diesem Unterbereich gibt. Sehen Sie sich die in diesem Bereich befindlichen Files an und beantworten Sie folgende Fragen:

- kurzlexikon-deutsch: Wozu ist der Knopf `Rot-13` im `xrn`-Fenster da?
- antworten-englisch: Was ist die Bedeutung der Zeichenkombination `:-)` ? Wie heißt sie?
- kurz-lexikon-deutsch, antworten-englisch, so-NICHT-deutsch: Was ist ein Flame? Wenn man unbedingt einen Flame loslassen möchte, wie kennzeichnet man ihn als solchen, um zu zeigen, daß man es doch nicht so böse meinte?
- einfuehrung-englisch: Was sind die FAQs einer Newsgroup?
- antworten-englisch: Was heißt das Kürzel „RTFM“?
- urheberrechte-englisch: Wie heißt der Rechner am Massachusetts Institute of Technology (MIT), vom dem man sich die FAQs vieler Newsgroups herüberholen kann?

15. Aufgabe: eigene Interessen bei News

Gehen Sie wieder in das Menü, wo alle Newsgroups aufgelistet sind, und drücken Sie den Knopf `All groups`. Sie bekommen jetzt eine Liste aller Newsgroups, die die Uni Konstanz „abonniert“.

Scrollen Sie die Liste soweit herunter, bis Sie zu den Gruppen kommen, die mit `sci` beginnen (sci=science). Tippen Sie die Newsgroup zu Ihrer Wissenschaft mit dem linken Mauszeiger an und drücken Sie `Subscribe`. Suchen Sie noch ein paar weitere interessante Newsgroups auf diese Weise heraus. Was passiert, wenn Sie jetzt `Quit` drücken?

HINWEIS: Falls Sie jetzt eine oder mehrere Newsgroups nicht aufgelistet bekommen, obwohl Sie sie ausgewählt haben, liegt das daran, daß `xrn` nur solche Newsgroups in diesem Menü auflistet, in denen momentan auch Artikel vorhanden sind, die noch nicht gelöscht worden sind. Falls Sie beim Lesen einer Newsgroup die Meldung `Download in progress` erhalten, wird diese Newsgroup an der Uni Konstanz nicht abonniert, sondern erst durch Ihre Anforderung von einem anderen Punkt im Internet geladen (in diesem Fall Stuttgart). Sie müssen sich in diesem Fall ein paar Minuten gedulden.

16. Aufgabe: File `.newsrc`

Sehen Sie sich nun das File `.newsrc` an. Was ist der Inhalt? Wie wird der Unterschied zwischen subskribierten und nicht subskribierten Newsgroups angezeigt?

3 Antworten zum Generalthema: Email selbst schreiben

1. Aufgabe: Email vorbereiten

Den Unterbereich `.elm` Ihres eigenen Bereichs hat der `elm` beim ersten Aufruf eingerichtet, als Sie bei der Bearbeitung des ersten Übungsfiles Email gelesen haben.

3. Aufgabe: Email verschicken

Wenn eine Mail eingeht, piept das `xbiff`-Fenster oben rechts. Falls es vorher weiß war, wird es jetzt schwarz, und die Signalfahne geht hoch.

4. Aufgabe: auf Email antworten

Das `>` am Anfang einer Zeile zeigt an, daß Sie diese Zeile aus einer anderen Mail übernommen haben und nur zitieren.

Achtung: Es gibt da keine allgemeingültige Konvention. Andere Mail-Programme zeigen Zitate anders an.

6. Aufgabe: falsche Adresse

Wenn Sie beim Abschicken einer Mail einen nicht-existierenden Benutzer als Adressaten eintragen, kann die Mail nicht ausgeliefert werden. Statt dessen erhält der Absender eine Mail vom Mail-Auslieferungsprogramm, die eine Fehler-Meldung enthält. Der entscheidende Hinweis auf die Art des Fehlers ist in diesem Fall „User unknown“.

7. Aufgabe: `Cc` und Mehrfachadressaten

Mit `Cc` kann man weitere Empfänger einer Mail angeben. Der Sinn besteht darin, daß diese Empfänger nicht direkt angesprochen werden, sondern daß ihnen diese Mail nur „zur Kenntnis“ gegeben wird.

`Cc` steht für „Carbon Copy“, also Kohledurchschlag (entlehnt aus der Zeit, als man Briefe noch mit der Schreibmaschine und nicht am Computer getippt hat).

9. Aufgabe: das doppelte `elm`chen

Der `elm` verweigert zwei simultane Aufrufe. Man muß erst aus dem ersten `elm` aussteigen, bevor man den zweiten `elm` aufrufen kann. Der Grund ist, daß der `elm` die eingegangenen Mails nicht nur lesen, sondern auch verändern (z.B. löschen) kann. Wenn zwei `elms` die Mails manipulieren, ohne voneinander zu wissen, kann das leicht zu Inkonsistenzen führen.

10. Aufgabe: Wie ist Mail gespeichert?

Die Mails eines Benutzers sind einfach in einem File mit einem speziellen Namen gespeichert. Der `elm` kennt den Namen dieses Files.

Mails sind eine persönliche Angelegenheit eines Benutzers. Deshalb ist das File vor fremdem Zugriff geschützt. Dieser Schutzmechanismus und wie man ihn selbst verwenden kann, wird Inhalt eines der nächsten Termine sein.

Z.B. für Benutzer `weihe` lautet der Kopierbefehl:

```
cp /var/mail/weihe ~
```

Zurückspeichern der Referenzkopie (wieder Beispiel `weihe`):

```
cp ~/weihe /var/mail
```

Merkregel: Unter UNIX wird grundsätzlich alles als ein ganz normales File gespeichert.

11. Aufgabe: eingegangene Mail löschen

Man kann Mails zum Löschen vormerken. Wenn man aus dem `elm` herausgeht, wird man noch einmal gefragt, ob man die Mail wirklich löschen möchte. Dies ist die letzte Gelegenheit, es sich noch einmal anders zu überlegen. Danach ist die Mail unwiderbringlich verloren.

Das Kürzel `y` steht für `yes`, `d` und `u` stehen für `delete` und `undelete`.

12. Aufgabe: Protokoll aller ausgegangenen Emails

Die ManPage zu `elm` sagt, daß man mit

`elm -f` *folder*

den `elm` mit Basis *folder* aufrufen kann.

Achtung: Dies war eine wichtige Aufgabe. Wenn Sie sie nicht selbst beantworten konnten, schauen Sie noch einmal in der ManPage nach und überlegen sich, wie Sie auch alleine darauf hätten kommen können.

Mit

```
elm -f =sent
```

wird ein spezieller Folder verwendet. Dieser Folder enthält die Mails, die man selbst abgeschickt hat.

Das Gleichheitszeichen wird vom elm als Ihr Unterbereich `~/Mail/` interpretiert.

Wenn man eine Mail an die falsche Adresse geschickt hat, kann man z.B. mit `elm -f =sent` wieder darauf zugreifen.

Die Zeile `copy = ON` ist dafür verantwortlich, daß alle herausgehenden Mails mitprotokolliert werden. Die Zeile `copy = OFF` hat dagegen keinen Effekt, weil sie „auskommentiert“ ist. Das `#` am Anfang der Zeile sagt, daß diese Zeile auskommentiert ist. Um Mitprotokollieren zu verhindern, muß man also die Zeile auskommentieren und die Zeile `copy = OFF` einkommentieren.

Wiederholung: Welche Zeilen haben Sie schon an einem früheren Termin ein- bzw. auskommentiert?

Verwendet man ein File als Basis, der keine Mails enthält (so wie `~weihe.unix96/ Aufgabe1`), dann erkennt `elm` das und gibt eine Fehlermeldung aus.

13. Aufgabe: News lesen

Sie haben in dieser Aufgabe zwei News-Gruppen „abonniert“, `de.sci.informatik.misc` und `rec.arts.ascii`. Das bedeutet nichts anderes als daß `xrn` Ihnen diese News-Gruppen bei je-

dem Aufruf zum Lesen bereitstellt. Dieses Abonnement ist kostenlos, und Sie werden nirgendwo registriert (außer in Ihrem eigenen File `.newsrc`; siehe Aufgabe 16).

News sind ein weltweites Forum zum Austausch von Informationen und Meinungen. Damit das Forum überschaubar bleibt, ist es in tausende von einzelnen Newsgroups zergliedert. Die Newsgroups sind nach Themen gegliedert, und in einer Newsgroup „treffen" sich alle, die an diesem Thema interessiert sind.

Die News-Groups sind hierarchisch gegliedert. Bspw. bedeutet `de.sci.informatik.misc`

`de`:	deutschsprachige Newsgroup
`sci`:	scientific = wissenschaftlich
`informatik`	Spezifizierung der Wissenschaft, um die es in der News-Group geht.
`misc`	miscellaneous = vermischtes

und rec.arts.ascii bedeutet

`rec`:	recreational = alles, was mit Freizeit zu tun hat
`arts`:	die schönen Künste
`ascii`:	Kunst mit Hilfe des ASCII-Zeichensatzes

Artikel, die in den News „gepostet" wurden, werden nur ein paar Tage lang aufgehoben. Danach werden sie gelöscht und sind nicht mehr zugreifbar (die Postings einiger Newsgroups werden auch archiviert).

14. Aufgabe: `weihe/news-start`

`Rot-13` macht denText unleserlich, indem er jeden Buchstaben zyklisch um 13 Stellen im Alphabet verschiebt. Damit ist der Text natürlich nicht wirklich verschlüsselt. Wenn man es schon nicht lassen kann (man kann es lassen!), Texte im Internet zu verbreiten, die irgendwie anstößig sind, ist es guter Stil, ihn vor dem Verschicken mit `Rot-13` unleserlich zu machen und eine (nicht unleserlich gemachte) Warnung an den Anfang zu stellen, wer sich vielleicht alles dadurch beleidigt fühlen könnte.

`:-)` ist der Smiley. Kurz gesagt bedeutet er: alles nicht so ernst gemeint, wie es vielleicht klingt.

Ein Flame ist ein ziemlich aggressiv und beleidigend formuliertes Posting. Um einen Flame abzuschwächen, kann man den beleidigenden Teil in eine Klammer `FLAME ON ... FLAME OFF` stellen.

Die FAQs einer News-Group sind eine Sammlung von Antworten zu häufig in dieser News-Group gestellten Fragen: FAQ = frequently asked question

RTFM = read the f*ing manual. Wenn man dies als Antwort auf eine Frage erhält, hätte man sich die Frage durch Nachschlagen auch selbst beantworten können.

Der Rechner am MIT heißt konsequenterweise auch „rtfm" und ist im Internet ansprechbar durch `rtfm.mit.edu`.

4 Generalthema heute: Files und Directories

Allgemeiner Hinweis: In den folgenden Aufgaben werden Sie oft gebeten, in die ManPages hineinzuschauen. Sie sollten diese Aufgabenteile unbedingt sorgfältig durcharbeiten, denn ManPages werden das wichtigste Thema bei den Rücksprachen sein! Falls Sie einige der Aufgaben mit ManPages nicht ohne File `Antworten4` lösen können, sollten Sie sich auf alle Fälle hinterher überlegen, wie Sie vielleicht doch von alleine auf die Lösung gekommen wären.

Sie sollten sich nicht entmutigen lassen, falls Sie den Inhalt von ManPages nicht wirklich verstehen. Die Kunst des ManPage-Lesens besteht darin, alles Unverständliche zu überspringen und irgendwie durch eine Mischung aus sorgfältigem Suchen und Intuition die Informationen herauszuziehen, die Sie brauchen.

1. Aufgabe: Suchen im `less`

Sehen Sie sich die ManPage zu `less` an und beantworten Sie sich folgende Fragen:

- Wie kann man im `less` vorwärts nach einer Zeichenkette suchen?
- Wie geht es rückwärts?
- Wie kann man die letzte Suche mit einem Tastendruck wiederholen?
- Wie kann man bei der Wiederholung die Suchrichtung umkehren (d.h. vorwärts-rückwärts) ?

Hinweis: Die Antworten auf diese Fragen werden Ihnen im Laufe dieses Termins und an allen folgenden Terminen die Arbeit erleichtern. Denken Sie insbesondere daran, daß auch die Programme `elm` und `man` Text im `less` anzeigen.

2. Aufgabe: Folien anschauen

Rufen Sie das Kommando

```
ghostview ~weihe/unix96/folien/96.10.14.ps &
```

auf. Was sehen Sie nun? Wie können Sie sich die weiteren Folien dieses Vortrags ansehen? Wie können Sie mit Hilfe der Liste aller Seitenzahlen links neben dem Text auf eine spezielle Seite, die Sie interessiert, gezielt zugreifen?

ACHTUNG: Bedenken Sie, daß Ausdrucken der Folien Ihr knapp bemessenes Kontingent (100 Seiten) rasch reduziert!

Stattdessen werde ich beizeiten alle Folien in Papierform zugänglich machen.

Sehen Sie sich nun

```
~weihe/unix96/folien/96.10.14.ps
```

auch einmal „ganz normal“ im `less` an. Finden Sie die Kommandos, die die Zeichenkette „1. Semesterhälfte“ auf der zweiten Folie erzeugt haben?

Rufen Sie das Kommando `file` mit dem Argument

```
~weihe/unix96/folien/96.10.14.ps
```

auf. Wie heißt also die Sprache, in der Text formuliert sein muß, damit das Programm `ghostview` solche hübschen Anzeigen liefert?

Bemerkung: Allgemein ist es Konvention, daß der Name eines Files, das in dieser Sprache abgefaßt ist, auf `ps` endet. Logisch, oder?

Bemerkung: Das File `96.10.14.ps` ist durch ein anderes Programm erstellt worden aus einem anderen File, das von mir in der Sprache LaTeX geschrieben wurde. LaTeX ist wesentlich einfacher! (Wie einfach LaTeX ist: siehe Aufgaben 13-15 unten.)

3. Aufgabe: Kommando file

Sehen Sie sich die ManPage zum Kommando `file` an und versuchen Sie herauszufinden, wodurch `file` erkannt hat, daß es sich um PostScript handelt. Sie müssen sich dazu ein in der ManPage angegebenes File ansehen und darin die Zeile zu PostScript suchen.

Wenn Sie diese Zeile gefunden haben, sehen Sie sich noch einmal im `less` den Anfang von

```
~weihe/unix96/folien/96.10.14.ps
```

an und vergewissern Sie sich, daß das File tatsächlich wie angegeben als PostScript-File identifizierbar ist.

Überlisten Sie das Kommando `file`, indem Sie im `emacs` ein File `dummy` erstellen, das für `file` wie ein PostScript-File aussieht, in Wirklichkeit aber gar keines ist.

4. Aufgabe: `ls`

Geben Sie nacheinander folgende Kommandos ein. Nachdem Sie das erste Kommando eingegeben haben, spielen Sie kurz ein bißchen mit den vier Pfeiltasten herum, um zu sehen, wie Sie sich bei den weiteren Kommandos (und auch allgemein in Zukunft) Tipparbeit sparen können.

Kommandos im einzelnen:

```
ls ~weihe
ls ~weihe -a
ls ~weihe -l
ls ~weihe a
ls ~weihe l
ls ~weihe -la
ls ~weihe -al
ls ~weihe -l -a
ls ~weihe -a -l
ls ~weihe -s
ls ~weihe -als
```

Wozu dient wohl hier das Minuszeichen?

Schauen Sie sich mit `man ls` an, was die Optionen `a, l` und `s` bedeuten. Hat es einen Effekt, in welcher Reihenfolge man diese Optionen angibt oder ob man sie zusammen oder getrennt angibt?

Die ManPage zu `ls` gibt auch eine Option an, mit der man die Ausgabe nach der Größe der Files sortieren kann. Welche Option ist das? Prüfen Sie Ihre Antwort nach, indem Sie `ls ~weihe` aufrufen mit zwei Optionen zugleich: Option `l` und die von Ihnen gefundene Option.

5. Aufgabe: Wildcards

Das ist das englische Wort für den Joker beim Kartenspiel. Ein treffenderer Ausdruck wäre „Platzhalter“.

Geben Sie folgende Kommandos ein:

```
ls ~weihe/unix96/Aufg*
ls ~weihe/unix96/Aufgabe?
ls ~weihe/unix96/Aufg?
ls ~weihe/unix96/Auf?abe1
ls ~weihe/unix96/A*1
ls ~weihe/unix96/*1
ls ~weihe/unix96/A*be?
ls ~weihe/unix96/Aufg*abe1
```

Was ist also die Funktion von `?` und `*` bei Eingaben im `xterm`-Fenster? Worin unterscheiden sich `*` und `?` ?

6. Aufgabe: `cd`

Sie gehen nun mit `cd` Schritt für Schritt aus Ihrem Heimatbereich (Home Directory) in den Heimatbereich von User `weihe`. Tippen Sie dazu die folgenden Kommandos ein:

```
cd ..
cd ..
cd inf
cd weihe
```

Jetzt gehen Sie wieder in einem Schritt von diesem Bereich zurück in Ihre eigene Home Directory:

```
cd ~
```

Gehen Sie wieder schrittweise vorwärts und in einem Schritt zurück, aber nun rufen Sie jedesmal zwischendurch `pwd` auf:

```
pwd
cd ..
pwd
cd ..
pwd
cd inf
pwd
cd weihe
pwd
cd ~
pwd
```

Beantworten Sie sich aus der ManPage zu `pwd` folgende Fragen:

- Was ist eine Directory?
- Was hat man sich unter der Current Directory bzw. Working Directory vorzustellen?

- Wieso spricht man von einer „hierarchischen“ Organisation aller Directories?
- Was bedeutet offenbar .. bei den obigen Kommandos?

7. Aufgabe: `pushd` und `popd`

Wenn Sie wieder sicher in Ihrem eigenen Bereich angelangt sind, geben Sie das Kommando

```
pushd ~weihek/.elm
```

ein. In welcher Directory sind Sie jetzt? Geben Sie nun

```
pushd ~weihe
```

ein. Wo sind Sie nun? Geben Sie jetzt zweimal `popd` ein und beobachten Sie nach jedem Mal, in welcher Directory Sie nun wieder sind.

Wiederholen Sie die ganze Prozedur, nur daß Sie nun zweimal `pushd` anstelle von `popd` tippen. Was ist der Unterschied? (Falls Sie absolut keinen Unterschied feststellen, geben Sie noch ein paar Mal `pushd` ein.)

Schauen Sie in der ManPage zu `pushd` folgendes nach: Angenommen, Sie haben fünf Directories auf diesem Stapel, mit welchem Argument muß man `pushd` aufrufen, um die unterste Directory zur obersten zu machen?

8. Aufgabe: `mkdir` und `cp`

Gehen Sie jetzt wieder in Ihre Home Directory.

Geben Sie jetzt das Kommando `mkdir Aufg` ein. Schauen Sie sich mit `ls -l` an, was der Effekt war. Schauen Sie in der ManPage zu `ls` nach, was das allererste Zeichen in der Zeile zu `Aufg` besagt. Was tut `mkdir` also? Schauen Sie in der ManPage zu `mkdir` nach, wofür `mkdir` die Abkürzung ist.

Geben Sie das Kommando

```
cp ~weihe/unix96/Aufg* Aufg
```

ein. Was sagt jetzt `ls Aufg`?

Jetzt sollten Sie in allen Details verstehen, was die mysteriösen Kopierbefehle in den ersten drei Aufgabenblättern bewirkt haben?

9. Aufgabe: `mv`

Gehen Sie in die Directory `Aufg` hinein und erzeugen Sie jetzt - während Sie in `~/Aufg` stehen! - eine Directory `~/Aufg2`. Geben Sie dann das Kommando

```
mv Aufgabe{1,2,3} ~/Aufg2
```

ein. Prüfen Sie jetzt mit `ls` nach, was nunmehr der Inhalt von `~/Aufg` und `~/Aufg2` ist. Das Ergebnis wirft einige Fragen auf:

- Was ist die Aufgabe des Kommandos `mv`? Schauen Sie in der ManPage von `mv` nach, für welches englische Wort `mv` die Abkürzung ist.

- Wie interpretiert `xterm` offenbar Argumente, die Paare von geschweiften Klammern enthalten?
- Wozu ist offenbar das Komma innerhalb der geschweiften Klammern da?

WARNUNG: Wenn Sie `mv` benutzen, vergewissern Sie sich VORHER, daß damit kein schon existierendes File überschrieben wird oder – falls doch – daß Sie den alten Inhalt dieses Files nicht mehr brauchen.

10. Aufgabe: `rm` und `rmdir`

Gehen Sie nun wieder in Ihre Home Directory und geben Sie jetzt folgende zwei Kommandos ein:

```
rm Aufg/Aufgabe*
rm ~/Aufg2/Aufgabe1
```

Prüfen Sie jetzt wieder mit `ls` den Inhalt von `~/Aufg` und `~/Aufg2`. Was tut also das Kommando `rm`? Schauen Sie in der ManPage zu `rm` nach, für welches englische Wort `rm` die Abkürzung ist.

Geben Sie folgende zwei Kommandos ein:

```
rmdir Aufg
rmdir Aufg2
```

Was sagt jetzt `ls`? Ist es sinnvoll, daß das zweite Kommando keinen Effekt für `Aufg2` hatte? Rufen Sie nun das Kommando

```
rm -r Aufg2
```

auf. Was ist jetzt der Effekt? Schauen in der ManPage zu `rm` nach, was Option `r` tut. Was mag das Wort „recursively“ bedeuten?

WARNUNG: Wenn Sie selbst einmal `rm` mit Option `r` benutzen wollen, vergewissern Sie sich VORHER, ob das, was Sie damit löschen würden, wirklich nur das ist, was Sie löschen wollten! Probieren Sie lieber vorher `ls -R` aus. (Was tut Option `R` von `ls`?)

Frage: Wieso wäre `rm -rf ~/*` der absolute Super-GAU?

WARNUNG: Jemandem anders ein File namens `-rf *` oder ähnlich irgendwo hinzustellen in der Hoffnung, daß derjenige `rm` darauf anwendet, ist ein krimineller Akt und wird entsprechend verfolgt!

11. Aufgabe: Symbolic Links

Gehen Sie wieder in Ihre Home Directory.

Geben Sie nun das Kommando

```
ln -s ~weihe/unix96/Aufgabe1 Auf
```

ein. Was sagt nun `ls -l`? Was besagt insbesondere das erste Zeichen in der Zeile zu `Auf`? Und was bedeutet der Pfeil und die Angabe nach dem Pfeil am Ende der Zeile?

Geben Sie nun das Kommando

```
diff ~weihe/unix96/Aufgabe1 Auf
```

ein. Schauen Sie dann in der ManPage zu `diff` nach, was die Aufgabe von `diff` ist und was die „Ausgabe“ bei obigem Aufruf zu bedeuten hatte.

12. Aufgabe: `tty`

Rufen Sie das Kommando `tty` auf. Was sagt die ManPage zu `tty`, was für ein File `tty` ausgibt? Rufen Sie `tty` in mehreren verschiedenen `xterm`-Fenstern auf.

Gehen Sie in die Directory, in der die ausgegebenen Files stehen (es sollte immer dieselbe sein). Wenden Sie `ls -l` auf eines dieser Files an. Es müßte sich um einen Symbolic Link handeln. Wenden Sie `ls -l` auf das File hinter dem Pfeil an. Was für ein Zeichen steht jetzt in der ersten Spalte der Ausgabe?

Diese Phänomene werden in File `Antworten4` kurz erklärt und in der Vorlesung dann systematisch behandelt.

13. Aufgabe: LaTeX und `xdvi`

Machen Sie in Ihrer Home Directory eine Directory `tex` auf und kopieren Sie das File

```
~weihe/others/misc/sample.tex
```

dort hinein. Gehen Sie selbst dort hinein und geben Sie folgende Kommandos ein:

```
latex sample.tex
xdvi sample.dvi &
```

Sehen Sie sich das File `sample.tex` auch im `less` an und vergleichen Sie das, was Sie im `less` sehen, mit dem, was Sie im `xdvi`-Fenster sehen.

Ändern Sie Ihre Kopie von `sample.tex` so ab, daß bei jedem Aufruf von `LaTeX sample` immer automatisch das Datum des jeweiligen Tages erscheint anstelle des festgesetzten Datums.

14. Aufgabe: `dvips -o`

Geben Sie jetzt die Kommandos

```
dvips sample.dvi -o sample.ps
ghostview sample.ps &
```

ein. Wie ist also das File

```
~weihe/unix96/folien/96.10.14.ps
```

entstanden?

15. Aufgabe: selbst LaTeX spielen

Basteln Sie wiederholt ein wenig an Ihrer Kopie von `sample.tex` herum, lassen Sie zwischendurch immer wieder einmal `LaTeX sample` laufen und schauen Sie sich das Ergebnis jeweils mit `xdvi` an. Müssen Sie vor jedem `LaTeX` aus `xdvi` herausgehen oder können Sie `xdvi` einfach die ganze Zeit offenlassen?

4 Antworten zum Generalthema: Files und Directories

1. Aufgabe: Suchen im `less`

Man sucht im `less` vorwärts nach einer Zeichenkette, indem man zuerst ein `/` eingibt, danach die Zeichenkette und schließlich `Return`.

Zum Rückwärtssuchen tippt man `?` statt `/`.

Die letzte Suche wiederholt man mit `n`. Um die Suchrichtung umzukehren, tippt man statt dessen `N`.

2. Aufgabe: Folien anschauen

Mit den Knöpfen („Buttons“) auf der rechten Seite des Fensters kann man im `ghostview` vor- und zurückblättern. Eine Seite direkt ansteuern kann man mit dem mittleren Mausknopf auf den Seitenzahlen am linken Rand.

Mit Vorwärtssuche nach `Semesterh` findet man zwei Zeilen, die unter anderem die Kommandos

```
y(1.)76 b(Semesterh)m(\177)-110 b(alfte:)80
```

enthalten. Offenbar sind diese Kommandos irgendwie für die Zeichenkette „1. Semesterhälfte“ verantwortlich.

Die Sprache heißt PostScript.

3. Aufgabe: Kommando `file`

`man file` sagt, daß das File namens `/etc/magic` benutzt wird, um den Typ eines Files herauszufinden. In diesem File findet sich genau eine Zeile, die das Wort „PostScript“ enthält. Der Kommentar am Anfang des File `/etc/magic` sagt, daß die Zeichen in der dritten Spalte, also `!%`, gesucht werden, um ein File als PostScript-File zu identifizieren.

Ein File `dummy`, dessen einziger Inhalt `!%` ist, wird von `file` fälschlich als PostScript-File identifiziert.

4. Aufgabe: `ls`

Durch die beiden Pfeiltasten für „hinauf“ und „herunter“ kann man in den vorherigen Kommandos herumblättern und so einfach ein vorhergehendes Kommando noch einmal aufrufen. Man kann auch noch daran herumbasteln, bevor man es aufruft.

Das Minuszeichen sorgt dafür, daß die nachfolgenden Buchstaben als Optionen genommen werden und nicht z.B. als Namen des aufzulistenden Bereiches.

Optionen von `ls`:

- a: alle Files in der Directory. Ohne diese Option listet `ls` keine Files auf, die mit einem . beginnen. Warum das so ist, wird in der Vorlesung erklärt werden.
- l: Auflistung von umfangreicher Information für jedes einzelne File, z.B. Besitzer, Zugriffsrechte, Zeitpunkt der letzten Änderung.
- s: Auflistung der Größe aller Files in KiloBytes. 1 KiloByte = 1024 Zeichen.

Es ist egal, ob man die Optionen zusammen oder getrennt angibt. Die Reihenfolge ist ebenfalls egal.

Die gesuchte Option ist `S`. Laut ManPage kann man auch `--sort=size` tippen.

5. Aufgabe: Wildcards

`?` steht als Platzhalter für genau ein Zeichen. Bspw. bekommt man mit `ls Auf?abe?` alle Files aufgelistet, die genau sieben Zeichen enthalten, wovon die ersten drei `Auf` sind und die Zeichen Nr. 5-7 `abe` sind. Im Unterschied dazu ist `*` nicht ein Platzhalter für genau ein Zeichen, sondern für eine beliebig lange Zeichenkette. Wie das letzte Kommando zeigt, kann diese Zeichenkette auch leer sein.

6. Aufgabe: `cd`

Eine Directory (Verzeichnis, Ordner) ist ein Bereich, in dem man Files ablegen kann, sofern man die Berechtigung dazu hat. Jeder User hat seinen eigenen Heimat-Bereich (Home Directory), in dem er diese Berechtigung hat. Die eigene Home Directory kann man mit `~` ansprechen. Die Home Directory eines anderen Users, z.B. `weihe`, spricht man mit `~weihe` an.

Die Current Directory oder auch Working Directory ist die Directory, in der `xterm` sich gerade „befindet“. Mit `cd` ändert man die Current Directory.

Eine Directory kann nicht nur Files, sondern ihrerseits auch wieder Directories enthalten. Z.B. enthält die Directory `/home` die Directory `/home/student`, und `/home/student` enthält wiederum die Home-Directories aller Teilnehmerinnen und Teilnehmer des UNIX-Praktikums.

Mit `..` spricht man die Directory an, die in der Hierarchie genau eins höher ist.

7. Aufgabe: `pushd` und `popd`

Die Kommandos `pushd` und `popd` organisieren Directories auf einem „Stapel“. Die Current Working Directory ist immer die oberste auf dem Stapel. Wenn man nie `pushd` aufruft, ist das die einzige Directory auf dem Stapel, und man merkt von diesem Stapel gar nichts. Mit `pushd directory` packt man `directory` oben auf den Stapel, mit `popd` entfernt man die oberste Directory vom Stapel. Mit `pushd` ohne Argument tauscht man die beiden Directories aus, die auf dem Stapel zuoberst liegen. D.h. wenn man andauernd `pushd` ohne Argument aufruft, springt man immer zwischen zwei Directories hin und her. Mit `pushd +4` macht man die unterste Directory von fünfen zur obersten.

8. Aufgabe: `mkdir` und `cp`

Das erste Zeichen bei der Ausgabe durch `ls -l` gibt den Typ eines Eintrags an. Das `d` in der Zeile zu `Aufg` bedeutet, daß `Aufg` eine Directory ist. Das Kommando `mkdir` erzeugt also neue Directories (mkdir = make directory).

Nach dem Kopierbefehl `cp ~weihek/Aufg* Aufg` enthält Directory `Aufg` Kopien aller Files, auf die `~weihek/Aufg*` paßt.

9. Aufgabe: `mv`

Das Kommando zur Erzeugung von `~/Aufg2` lautet einfach: `mkdir ~/Aufg2`.

Das Kommando `mv` ist unter anderem dazu da, ein File von einer Directory in eine andere zu verschieben (mv=move).

Wenn ein Argument einen Ausdruck der Form `Aufgabe{1,2,3}` enthält, konstruiert `xterm` daraus eine Liste von Argumenten: `Aufgabe1 Aufgabe2 Aufgabe3`. D.h. das Kommando

.mv Aufgabe1,2,3 /Aufg2.

hat die Files `Aufgabe1`, `Aufgabe2` und `Aufgabe3` nach `~/Aufg2` verschoben.

10. Aufgabe: `rm` und `rmdir`

`rm` = remove files

`rmdir` = remove directory

Der zweite Aufruf von `rmdir` hatte keinen Effekt, sondern hat nur eine Fehlermeldung ergeben. Das dient zum Schutz von Daten vor versehentlichem Löschen, und das ist sicher sinnvoll.

r = recursively bedeutet, daß das Kommando nicht nur Files in einer Directory löscht, sondern auch in allen Unterdirectories dieser Directory. Man nennt solche Durchläufe „Rekursionen".

Option `R` bei `ls` listet laut ManPage nicht nur die Files in einer Directory auf, sondern auch die Files in allen Unterdirectories.

Mit `rm -r ~/*` löscht man sich den Inhalt der eigenen Home Directory und aller Unterdirectories, also den gesamten eigenen Datenbestand. Gibt man noch Option `f` dazu, macht `rm` das skrupellos ohne irgendwelche Nachfragen.

VORSICHT: Verlassen Sie sich nicht darauf, daß `rm` ohne Option `f` irgendwelche Fragen stellt, bevor die Files gelöscht werden. Sie können das nur mit Option `i` erzwingen.

11. Aufgabe: Symbolic Links

Das erste Zeichen in der Ausgabe von `ls -l` zu `Auf` zeigt an, daß es sich um einen File-Typ handelt, den wir noch nicht behandelt haben: einen symbolischen Verweis auf ein anderes File (Symbolic Link). Das File `Auf` ist also nur ein Verweis für das File `~weihe/unix96/Aufgabe1`. Dieses File ist nach dem Pfeil am Ende der Zeile mit aufgeführt.

Das Kommando `diff` erwartet zwei Files als Argumente und gibt die Unterschiede zwischen beiden Files aus. Da `diff` in der Aufgabe keine Ausgabe liefert, ist der Inhalt beider Files identisch. Das ist auch nicht überraschend, da das zweite Argument ein Verweis auf das erste Argument ist.

Wenn man mit `cd` in eine Directory geht, die in Wirklichkeit nur ein Symbolic Link für eine andere Directory ist, dann betritt man mit `cd` diese andere Directory. Mit `cd ..` kommt man dann von dieser anderen Directory aus eine Stufe höher in der Hierarchie.

12. Aufgabe: `tty`

`tty` gibt den Namen eines Files aus. Dieses File ist ein Beispiel für eine generelle UNIX-Philosophie: Alles ist ein File. Directories sind Files, Symbolic Links sind Files, sogar die Eingabe von der Tastatur ist ein File, und der Name dieses Files wird durch `tty` ausgegeben.

In der ersten Spalte der Ausgabe von `ls -l` steht immer der Typ eines Files. Hier steht ein `c`. Das ist das Kürzel für einen anderen File-Typ, den wir noch nicht kennen: zeichenorientiertes Geräte-File (c=character=Zeichen).

Die Vorlesung wird diesen Stoff systematisch aufarbeiten.

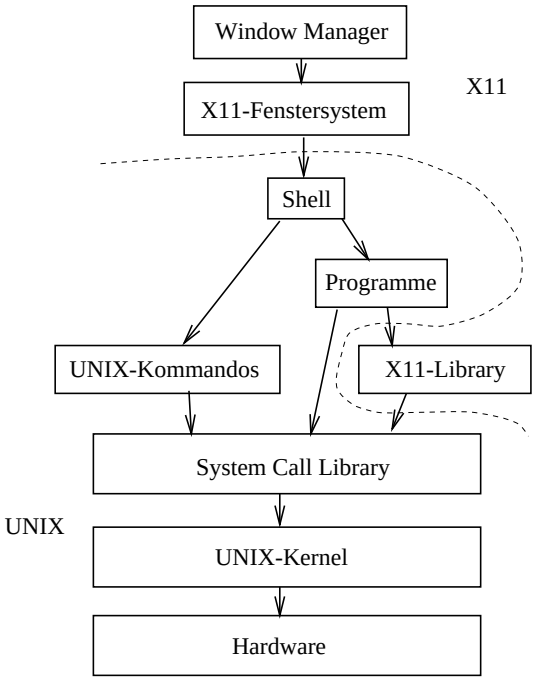
UNIX/C–Praktikum

Donnerstag, 31.10.1996

Heute: Grundsätzliches zu

- Abarbeitung von Kommandos,
- Aufbau von UNIX,
- Basiskonzepte wie Files, Prozesse, Bereiche.

Aufbau der Systemsoftware



1. Szenario: Aufruf eines Kommandos im Xterm–Fenster:

- Benutzer positioniert Mauszeiger auf Xterm–Fenster.
- X11–Fenstersystem erhält eingegebene Zeichen.
- X11 berechnet aus Position des Mauszeigers, in welchem Fenster er steht (hier: Xterm).
- X11 leitet eingegebene Zeichen an Prozeß weiter (hier: Shell), der in diesem Fenster läuft.
- Shell versucht, eingegebene Zeichen als Name eines Programms (+Zusatzargumente) zu interpretieren.
- Falls erfolgreich, gibt Shell an Kernel Auftrag, Prozeß mit dazugehörigem Programm zu starten.
- Wenn Prozeß terminiert, wird aufrufender Prozeß (Shell) von Terminierung in Kenntnis gesetzt.

Eingabedaten für Prozeß:

- Prozeß öffnet Fenster

  ⇒ Eingabedaten in diesem Fenster werden an Prozeß weitergereicht.

- Prozeß wird ohne & gestartet

  ⇒ Eingabedaten in Xterm–Fenster werden nicht an Shell, sondern an diesen Prozeß weitergereicht.

  Shell wird schlafengelegt und erst bei Terminierung des Prozesses wieder aufgeweckt.

Interpretation der Eingabe:

Wdh.: Eingabe von Tastatur besteht aus ASCII–Code.

ASCII–Code enthält Klein- und Großbuchstaben, Ziffern, Sonderzeichen und nichtschreibbare Zeichen.

Wichtige Beispiele für nichtschreibbare Zeichen:

- Newline (Return–Taste, ASCII 10).
- Backspace (ASCII 8).
- Delete (Del–Taste, ASCII 127)
- Tabulator (Tab–Taste, ASCII 9).
- Bell (Glocke, ASCII 7).

Grundlegende Philosophie: Jeder Prozeß interpretiert Eingabe nach Gutdünken!

Insb. nichtschreibbare Zeichen werden oft verschieden interpretiert.

2. Szenario: Knopf drücken im xrn–fenster

- Benutzer positioniert Mauszeiger auf Knopf im xrn–Fenster und klickt Maustaste an.
- X11 erhält Information, daß Maustaste angeklickt wurde.
- X11 berechnet aus Position des Mauszeigers, daß ein Knopf „gedrückt“ wurde, und auch welcher in welchem Fenster.
- X11 informiert xrn-Prozeß, der in diesem Fenster läuft, davon, welcher Knopf gedrückt wurde.
- xrn-Prozeß reagiert in der Weise wie er programmiert wurde.

  Bsp.: Druck auf Knopf „Kill Author“ fügt Autor des aktuell angezeigten Postings in persönliches „Kill File“ ein.

3. Szenario: Window Manager

Window Manager beherrscht

- das den ganzen Bildschirm umfassende, dunkle Hintergrundfenster,
- den grauen Rand jedes anderen Fensters.

⇒ *Events* (Tastatureingaben und Maus–Klicks) in Hintergrundfenster und auf Rand von Fenstern werden von X11 an Window Manager weitergereicht.

⇒ Window Manager hat u.a. folgende Funktionen:

- Gestaltung des Randes jedes Vordergrundfensters.
- Menüs im Hintergrundfenster.
- Fenstermenü im grauen Rand eines Fensters oben links.
- Änderung der Koordinaten eines Fensters (auch: Iconifizieren).

UNIX–Kernel

Generelle Aufgabe: Details der internen Organisation und der Hardware verwalten und vor Ebenen, die auf Kernel aufbauen, verstecken.

- Zugriff auf Files,
- Prozesse starten, laufenlassen und beenden,
- Kommunikation zwischen CPU und Peripherie,
- Kontrolle von Benutzerrechten.

Attribute von Benutzern:

- Benutzerkennung: weihek
- Passwort: ???
- Hauptgruppe (student)
- eine oder mehrere Nebengruppen (unixc)
- eigener Heimbereich: /home/student/weihek

```
weihe@hoyren ~ % finger weihek
Login name: weihek              In real life: Karsten Weihe
Directory: /home/student/weihek  Shell: /usr/local/bin/tcsh
Never logged in.
New mail received Tue Oct 22 14:19:23 1996;
  unread since Thu Oct 17 13:29:28 1996
No Plan.
```

```
weihe@hoyren ~ % groups weihek
weihek : student unix96
```

Attribute von Files

- Name.
- eindeutige ID.
- Besitzer.
- Gruppe (meist Hauptgruppe des Besitzers).
- Größe.
- Zeit des letzten Zugriffs:
  - Access time: letztes Mal Inhalt gelesen oder verändert.
  - Modification time: letztes Mal Inhalt verändert.
  - Change time: letztes Mal Attribute verändert.
- Zugriffsrechte.

Zugriffsrechte:

- Leserecht,
- Schreibrecht,
- Ausführrecht

jeweils getrennt für

- Besitzer selbst,
- Gruppe des Files,
- alle anderen Benutzer.

Ausführrecht: nur bedeutsam bei Programmen, nämlich das Recht, das Programm auszuführen.

```
weihe@hoyren unix/termin-11-02 % ls -li ~weihek/Aufgabe1
149786 -rwxr--r--  1 weihek student 12027 Oct 19 15:39 Aufgabe1
```

149786 ID des Files.

rwxr−−r−− Zugriffsrechte:

- Erste drei Zeichen: Rechte des Besitzers weihek.
- Zweite drei Zeichen: Rechte der anderen Mitglieder der Gruppe student.
- Dritte drei Zeichen: Rechte aller anderen Benutzer.

Innerhalb Dreierkolonne:

- Erstes Zeichen: Leserecht (r=read).
- Zweites Zeichen: Schreibrecht (w=write).
- Drittes Zeichen: Ausführrecht (x=execute).

weihek Besitzer.

student Gruppe.

12027 Größe des Files in Bytes.

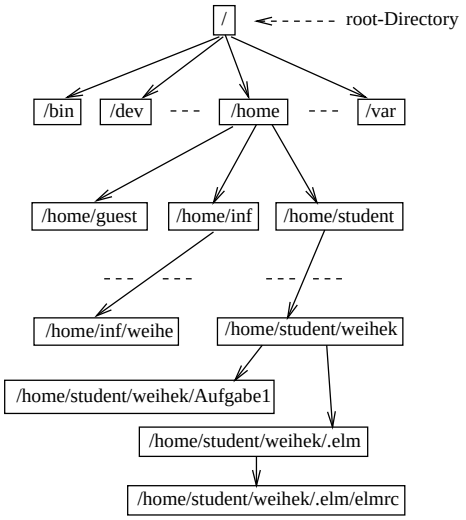
Oct 19 15:39 letzte Änderung des Inhalts (Modification Time).

Aufgabe1 Name des Files.

Bereiche (Directories):

- Jedes File ist in einer Directory gespeichert.
- Directories sind hierarchisch organisiert.

Ausschnitt aus Directory–Baum:



Absoluter Pfadname:

Beispiel: /home/student/weihek/Aufgabe1 ist absoluter Pfadname des Files Aufgabe1 von User weihek.

⇒ Lesen mit

```
weihe@konstanz ~ % less /home/student/weihek/Aufgabe1
```

Abkürzung für Home–Directory:

```
weihe@konstanz ~ % less ~weihek/Aufgabe1
```

Abkürzung für eigene Home–Directory:

```
weihek@konstanz ~ % less ~/Aufgabe1
```

Beachte: `weihek` statt `weihe`!

Relativer Pfadname:

```
weihe@konstanz ~ % ls ~weihek/Aufg*
  /home/student/weihek/Aufgabe1
  /home/student/weihek/Aufgabe2
  /home/student/weihek/Aufgabe3
  /home/student/weihek/Aufgabe4

weihe@konstanz ~ % cd ~weihek

weihe@konstanz /home/student/weihek % ls Aufg*
  Aufgabe1  Aufgabe2  Aufgabe3  Aufgabe4

weihe@konstanz /home/student/weihek % cd .elm

weihe@konstanz /home/student/weihek/.elm % ls ../Aufg*
  ../Aufgabe1  ../Aufgabe3
  ../Aufgabe2  ../Aufgabe4

weihe@konstanz /home/student/weihek/.elm % cd ../..

weihe@konstanz /home/student % ls weihek/Aufg*
  weihek/Aufgabe1  weihek/Aufgabe3
  weihek/Aufgabe2  weihek/Aufgabe4
```

Current Working Directory:

- Jeder Prozeß hat eigene *Current Working Directory*, die auch geändert werden kann.
- Bei jedem neuen Xterm (d.h. Shell) ist Current Working Directory anfangs Home Directory des Benutzers.
- Die Current Working Directory einer Shell ändert man durch Kommando cd.
- Argument von cd ist entweder absoluter oder relativer Pfadname der neuen Directory (relativ zur alten).

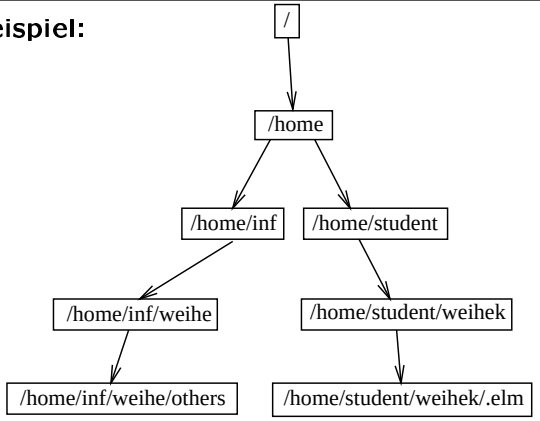
Relativer Pfadname:

Mit .. kommt man eine Stufe höher.

Ist Directory namens „*Dir*“ unmittelbar in Current Working Directory enthalten, ist „*Dir*“ relativer Pfadname dieser Directory.

Kette: durch / miteinander verbunden.

Beispiel:



Schritt für Schritt:

```
weihe@hoyren ~/others % cd ..
weihe@hoyren ~ % cd ..
weihe@hoyren /home/inf % cd ..
weihe@hoyren /home % cd student
weihe@hoyren /home/student % cd weihek
weihe@hoyren /home/student/weihek % cd .elm
weihe@hoyren /home/student/weihek/.elm %
```

In einem Sprung:

```
cd ../../../student/weihek/.elm
```

Generelle Philosophie: Daten und Datenträger aller Art sind Files.

Insbesondere:

- ASCII–Files: Files, deren Inhalt Texte sind.
- Programme: Files, deren Inhalt Maschinenbefehle sind.
- Directories.
- Externe Geräte.
- Symbolic Links.

Files aller Arten haben dieselben Attribute.

Weiteres Fileattribut: Filetyp.

Symbolic Links:

```
weihe@konstanz ~ % ln -s ~weihek/Aufgabe1 Aufg1

weihe@konstanz ~ % ls -l Aufg1
lrwxrwxrwx   1 weihe  inf  29 Oct 29 13:22 Aufg1
               -> /home/student/weihek/Aufgabe1

weihe@konstanz ~ % diff Aufg1 /home/student/weihek/Aufgabe1

weihe@konstanz ~ %
```

Also: Symbolic Link ist zusätzlicher Name für schon existierendes File. Es sind z.B. äquivalent:

```
weihe@konstanz ~ % less Aufg1

weihe@konstanz ~ % less /home/student/weihek/Aufgabe1
```

Filetyp: Kenntlich durch ersten Buchstaben in der Ausgabe von „*ls -l*“ (*l*=*long listing*).

```
weihek@konstanz ~ % ls -l Aufgabe1
-rwxr--r--  1 weihek  student  12027 Oct 19 15:39 Aufgabe1

weihek@konstanz ~ % ls -l .elm
drwxr-xr-x  2 weihek  student  512   Oct 19 13:38 .elm

weihek@konstanz ~ % tty
/dev/pts/0

weihek@konstanz ~ % ls -ld /dev
drwxrwxr-x  16 root   sys      5120  Oct 13 07:31 /dev

weihek@konstanz ~ % ls -ld /dev/pts
drwxrwxr-x   2 root   sys      1024  Aug 28 11:00 /dev/pts

weihek@konstanz ~ % ls -l /dev/pts/0
lrwxrwxrwx   1 root   root     28    Aug 28 10:10 /dev/pts/0
                                  -> /devices/pseudo/pts@0:0

weihek@konstanz ~ % ls -l /devices/pseudo/pts@0:0
crw--w--w-   1 weihe    inf       24,   0 Oct 29 13:17
                                     /devices/pseudo/pts@0:0
```

Directory als File: Inhalt = Sammlung der IDs aller in der Directory enthaltenen Files.

Zugriffsrechte bei Directories:

- Leserecht: Kommandos wie „ls“ sind erlaubt.
- Schreibrecht: Man darf Files einfügen und löschen.
- Ausführrecht: Man darf Directory betreten (z.B. mit „cd“).

Warnung: Falls Schreibrechte von File weggenommen, aber für Directory des Files immer noch vorhanden, kann man File trotzdem ändern:

- File in eine eigene Directory kopieren.
- Ursprüngliches File löschen.
- Kopie nach Gusto ändern.
- Geänderte Kopie in ursprüngliche Directory zurückspeichern.

Spur: File hat jetzt anderen Besitzer.

5 Generalthema heute: Kommandos für Profis

ACHTUNG: Die heutigen Themen sind eines der Herzstücke des UNIX-Teils!

1. Aufgabe: `man man`

Natürlich hat auch das Kommando `man` eine Man Page. Sehen Sie sich diese Man Page an und überlegen Sie sich folgende Fragen:

- Was mag der *search path* `MANPATH` sein?
- Was ist mit Sektionen gemeint?

Suchen Sie nun die Optionen von `man`, die folgendes tun:

- Alle Sektionen auflisten, in denen zu einem Kommandonamen eine Man Page zu finden ist.
- Die Man Page in einer spezifischen Sektion gezielt ansteuern.

In welchen Sektionen gibt es also Man Pages zu `man`? Steuern Sie die zweite Man Page zu `man` gezielt an und beantworten Sie daraus folgende Fragen:

- Wofür ist der NAME-Teil einer Man Page da?
- Was steht in der SYNOPSIS?
- In welchem Teil steht, welche Fehler im Kommando bekannt sind?

2. Aufgabe: Synopsis

In der Beschreibung von SYNOPSIS, die Sie in der 1. Aufgabe heute nachgeschlagen haben, finden Sie auch ein paar Erläuterungen zu den in SYNOPSIS verwendeten Sprachkonstrukten. Sehen Sie sich die Man Page zum Kommando `wc` an und beantworten Sie sich selbst, welche der folgenden Kommandos laut Synopsis zulässig sind. Überlegen Sie sich die Antworten, BEVOR Sie die Kommandos ausprobieren, und vergleichen Sie dann.

```
wc ~weihe/unix96/Aufgabe1
wc ~weihe/unix96/Aufgabe1 ~weihe/unix96/Aufgabe2
wc ~weihe/unix96/Aufgabe?
wc -l ~weihe/unix96/Aufgabe1
wc -cl ~weihe/unix96/Aufgabe1
wc -lc ~weihe/unix96/Aufgabe1
wc -lcl ~weihe/unix96/Aufgabe1
```

Wiederholung von `Aufgabe4`: Was bedeutet `~weihe/unix96/Aufgabe?` ?

Ist `wc` ohne Optionen und Argumente auch zulässig? Probieren Sie:

```
% wc
bli
bla
blu
<Control-d>
```

3. Aufgabe: Pipes

Gehen Sie nach `~weihe/unix96` und geben Sie einmal folgende Kommandos ein. Beobachten Sie die Ausgabe jedes einzelnen Kommandos.

```
ls -l | grep Auf
ls -l | grep auf
ls -l | grep -i auf
ls -l | grep -v Auf
ls -l | grep Auf | grep -v be1 | grep -v be2
```

Schauen Sie in der Man Page zu `grep` nach, welche Aufgabe das Kommando hat. Was tun die Optionen `i` und `v`? Was bedeutet das englische Wort „case“ in Zusammenhang mit Option `i`?

Was ist offenbar der Sinn des Zeichens `|` bei Eingabe von Kommandos in einem `xterm`?

4. Aufgabe: `ps`

Geben Sie folgende Kommandos ein:

```
ps
ps -e
ps -l
ps -f
ps -ef
ps -elf
ps -elf | less
```

Finden Sie nun in der Man Page zu `ps` den Abschnitt, in dem die einzelnen angezeigten Spalten aufgelistet sind und ihre jeweilige Bedeutung beschrieben wird.

5. Aufgabe: Prozeßtabelle zurechtschneidern

Um mit solchen riesigen Tabellen gut zu arbeiten, kann man gut `grep`, `head`, `tail` und `cut` verwenden.

Geben Sie folgende Kommandos ein:

```
ps -elf | head -1
ps -elf | head -3
ps -elf | tail -10l
ps -elf | cut -b1-21
ps -elf | grep root
ps -elf | grep 0:00
ps -elf | cut -b1-21 | head -15
ps -elf | grep root | grep 0:00
ps -elf | grep 0:00 | cut -b1-21
ps -elf | cut -b1-21 | grep 0:00
```

Schauen Sie in den Man Pages zu `head`, `tail` und `cut` nach, wie diese Kommandos ihre jeweiligen Eingaben modifizieren. Warum haben die letzten beiden Kommandos unterschiedliche Ergebnisse?

6. Aufgabe: `kill`

Gehen Sie in Ihre Home Directory und rufen Sie auf:

```
emacs dummyneu&
```

Tippen Sie ein paar längere Zeilen mit beliebigem Inhalt, aber speichern Sie das File NICHT ab.

Suchen Sie mit `ps` in der Prozeßtabelle die Prozeß-ID (ID = Identifikation) von `emacs dummyneu` und geben Sie folgendes Kommando ein:

`kill -STOP` *Prozeß-ID*

Was passiert jetzt, wenn Sie in diesem `emacs` weitere Zeichen tippen? Geben Sie jetzt das Kommando

`kill -CONT` *Prozeß-ID*

mit derselben Prozeß-ID ein. Jetzt müßten die zuletzt getippten Zeichen erscheinen. Ansonsten ist der `emacs` jetzt hoffentlich wieder ganz der alte.

Geben Sie jetzt das Kommando

`kill -HUP` *Prozeß-ID*

ein. Das `emacs`-Fenster müßte jetzt weg sein. Rufen Sie erneut `emacs dummyneu&` auf. Was steht in der untersten Zeile des neuen `emacs`-Fensters? Geben Sie das dort angegebene Kommando im `emacs` ein, um das File wieder herstellen zu lassen:

erst *Esc*, dann `x`, dann `recover-file`, zweimal *Return* und schließlich `yes` *Return*

Jetzt müßte `dummyneu` wieder so dargestellt werden, wie es zuallerletzt aussah.

Tippen Sie wieder ein bißchen im File `dummyneu` herum, aber speichern Sie es wieder nicht ab. Suchen Sie nun die Prozeß-ID des neuen `emacs` mit `ps` und geben Sie nun mit dieser Prozeß-ID ein:

```
kill -KILL Proze\3-ID
```

Rufen Sie wieder `emacs dummyneu&` auf. Bekommen Sie die letzte Version von `dummyneu` wieder so wie eben zurück?

Achtung: Wenn Sie mit Kommando `kill` einen Prozeß „abschießen“, verwenden Sie `-KILL` nur dann, wenn `-HUP` versagt hat und eh nichts mehr zu retten ist!

7. Aufgabe: `man kill`

Sehen Sie sich die Man Page zu `kill` an.

- Was ist die Aufgabe des Kommandos `kill`?
- Was passiert, wenn man nur die Prozeß-ID und keine Option angibt?
- In `man kill` finden Sie einen Verweis auf eine andere Man Page, in der alle Signale aufgelistet und kurz erklärt werden. Was bedeuten also `STOP`, `CONT`, `HUP` und `KILL`?

8. Aufgabe: `grep` und Man Pages

Gehen Sie mit `cd` nach `/usr/man/man1` und geben Sie folgendes Kommando ein:

```
grep -i hop *
```

Verständnisfragen:

- Was ist wohl ein `Hop`?
- Welche Files sind durch obiges Kommando durchge„grep“t worden?

9. Aufgabe: `find`

Geben Sie folgende Kommandos ein und beobachten Sie die Ausgaben:

```
find ~weihe -name Aufgabe1
find ~weihe/unix96 -name Aufgabe1
find ~weihe/unix96 -type f
find ~weihe/unix96 -type f -exec ls -l {} \;
find ~weihe/unix96 -type f -exec wc -l {} \;
find ~weihe/unix96 -type f -exec wc -l {} \; -print
```

Sehen Sie sich nun die Man Page zu `find` an. Verständnisfragen:

- Was ist der Sinn des Kommandos `find`?
- Das erste Argument von `find` muß eine Directory sein (`path` in der Synopsis genannt). Was macht `find` mit dieser Directory und ihren Subdirectories?
- Was bewirkt `-name` ?
- Was bewirkt `-type f` ?
- Was bewirkt `-print` ?
- Was bewirkt `-exec ... \;` ?
- Was bedeutet `{}` hinter `-exec`?
- Im Abschnitt zu `-exec` in der Man Page zu `find` steht, daß der Ausdruck hinter `-exec` eigentlich nur durch ein Semikolon begrenzt wird. Welcher Satz weist darauf hin, daß dem Semikolon noch ein `\` vorangestellt werden muß?
- Welcher Teil des Kommandos

  ```
  find ~weihe/unix96 -type f -exec wc -l {} \; -print
  ```

  ist in der Synopsis als „expression“ bezeichnet?

9. Aufgabe: `find` und `grep` und Man Pages

Gehen Sie nun nach `/usr/man` und geben Sie folgendes Kommando ein:

```
find . -type f -exec grep -si hop {} \; -print
```

Hinweis: Die Ausgabe des Kommandos nimmt viel Zeit in Anspruch. Wenn es Ihnen zuviel wird, tippen Sie einfach `Control-c` im `xterm`. Es kommt in dieser Aufgabe nicht auf die gesamte Ausgabe an.

Beantworten Sie sich folgende Verständnisfragen mit Hilfe der Man Pages zu `find` und `grep`:

- Was ist der Exit Status von `grep`?
- Wann ergibt daher `-exec` in obigem Beispiel den Wahrheitswert „wahr“?
- Die Namen welcher Files werden also durch `-print` ausgegeben?
- Was würde passieren, wenn man `-print` vor `-exec` schreibt und nicht ans Ende? Warum?
- Welcher Teil der Ausgabe stammt von `find` und welcher von `grep`?
- Zusammenfassend: Wie läßt sich in einem kurzen Satz der Effekt der obigen Kommandozeile beschreiben?

11. Aufgabe: Wildcards an Kommandos weiterreichen

Wiederholung von Aufgabe4: Was ist die Funktion der Wildcard `*`?

Geben Sie folgende Kommandos ein:

```
find ~weihe/unix96 -name *.ps -print

find ~weihe/unix96 -name \*.ps -print

find ~weihe/unix96 -name \*.p\s -print

find ~weihe/unix96 -name "*.ps" -print

find ~weihe/unix96 -name '*.ps' -print
```

Verständnisfragen:

- Wer wertet offenbar im ersten Kommando die Wildcard aus: `find` oder die Shell?
- Was macht daher das `\` im zweiten Kommando?
- Hat das `\` einen Effekt, wenn es z.B. vor einen ganz normalen Buchstaben gestellt wird?
- Was für einen Effekt haben wohl `"..."` und `'...'` ?

Achtung: Verwechseln Sie nicht `'...'` mit `` `...` ``. Letzteres hat eine andere Bedeutung.

12. Aufgabe: Whitespaces

Geben Sie folgende Kommandos ein:

```
grep folgende Kommandos ~weihe/unix96/Aufgabe5

grep "folgende Kommandos" ~weihe/unix96/Aufgabe5

grep 'folgende Kommandos' ~weihe/unix96/Aufgabe5
```

Verständnisfragen:

- Warum gibt das erste Kommando auch Zeilen aus, in denen die Zeichenkette `Kommandos` nicht vorkommt?
- Wie läßt sich die erste Zeile in der Ausgabe des ersten Kommandos erklären?
- Warum steht in der Ausgabe des ersten Kommandos jeweils der Filename vorneweg? Unter welchen Umständen wird der Filename vorneweg geschrieben?

Hinweis: Es steht versteckt in der Beschreibung einer Option in der Man Page, die Sie mit `man grep` erhalten. Es steht auch sehr deutlich und auffällig in einer anderen Man Page zu `grep`.

13. Aufgabe: Anführungszeichen mit `grep` suchen

Geben Sie jetzt folgende Kommandos ein:

```
grep -i "hop" ~weihe/unix96/Aufgabe5

grep -i \"hop\" ~weihe/unix96/Aufgabe5
```

Warum ergibt sich ein Unterschied?

14. Aufgabe: `xfig`

Geben Sie das Kommando `xfig&` ein und malen Sie mit den Möglichkeiten, die `xfig` bietet, einfach drauflos. Versuchen Sie, wirklich möglichst alle Möglichkeiten, die `xfig` bietet, auszuprobieren.

Geben Sie auch Texte ein, die deutsche Umlaute und scharfes S enthalten. Wie das geht, steht in der Man Page zu `xfig`.

Speichern Sie nun Ihr Gemälde in einem File namens `test.fig` ab. Außerdem exportieren Sie es in „Portrait Mode“ in ein File namens `test.eps`, das in Encapsulated PostScript kodiert ist.

Fügen Sie nun im File `sample.tex`, das Sie aus `~weihek/Aufgabe4` kennen, folgendes ein:

Am Anfang direkt unter der `documentclass`-Zeile eine weitere Zeile mit Inhalt:

```
\usepackage{epsfig}
```

Am Ende direkt über der Zeile `\end{document}` die folgende Zeile:

```
\epsfig{file=test.eps, width=10cm}
```

Wenden Sie `latex` wie in `~weihek/Aufgabe4` an. Was ist das Resultat im `xdvi`?

5 Antworten zum Generalthema: Kommandos für Profis

1. Aufgabe: `man man`

Der Suchpfad `MANPATH` ist eine Variable, die alle Directories enthält, in denen das Kommando `man` nach Man Pages sucht.

Genauer: Die Man Pages stehen in den einzelnen Subdirectories dieser Directories.

Mit dem Kommando `echo $MANPATH` bekommt man alle diese Directories angezeigt. Probieren Sie's.

Man Pages sind in Sektionen unterteilt. Die Standard-Kommandos von UNIX gehören alle zur Sektion 1. Sektion 5 enthält einige Tabellen und tabellenähnliche Dokumentation. Die Sektionen 2 und 3 sind eher für den C++-Teil interessant.

Alle diese Themen werden in der Vorlesung noch systematisch behandelt werden.

Gesuchte Optionen: `l` und `s`, also:

```
% man -l man
man (1) -M /usr/man
man (5) -M /usr/man
% man -s 5 man
```

Der NAME-Teil erklärt kurz und bündig (eine Zeile), was das Kommando tut, das in der Man Page beschrieben wird.

Die Synopsis gibt an, wie man das Kommando verwenden darf, das heißt vor allem Optionen und Argumente.

Bekannte Fehler stehen im BUGS-Teil. Ein Fehler in einem Programm heißt üblicherweise „Bug".

2. Aufgabe: Synopsis

Alle Kommandos sind zulässig. Bei `wc -lc ...` und `wc -lcl ...` greift noch eine Regel, die sehr schlecht erklärt ist: `-cClw` bedeutet: Ein Minus gefolgt von einer beliebigen Zeichenkette, die sich nur aus den Zeichen `c`, `C`, `l` und `w` zusammensetzt, aber diese Zeichen können beliebig häufig und in beliebiger Reihenfolge auftreten.

`~weihe/unix96/Aufgabe?` steht für alle Filenamen, die mit `~weihe/unix96/Aufgabe` beginnen und dann noch ein einzelnes weiteres Zeichen haben.

Bei Kommando `wc` ohne Argumente erwartet `wc` Eingaben vom `xterm` (in der Man Page „standard input" genannt) und zählt die Zeichen, Wörter und Zeilen, die man eingibt.

Merkregel: Das Ende der Eingabe von „Standard Input" wird immer durch *Control-d* angezeigt.

3. Aufgabe: Pipes

Mit `grep` kann man sich alle Vorkommen einer Zeichenkette in einem File anzeigen lassen, genauer: alle Zeilen, in denen die Zeichenkette vorkommt. Mit Option `i` ist dabei Groß- und Kleinschreibung innerhalb der Zeichenkette egal. Durch Option `v` werden umgekehrt genau die Zeilen ausgegeben, die die Zeichenkette nicht enthalten.

Das englische Wort „case" bezieht sich hier auf Groß- und Kleinschreibung (upper/lower case).

Das Zeichen `|` verknüpft zwei Kommandos zu einem. Die Ausgabe des ersten Kommandos wird nicht angezeigt, sondern gleich als Eingabe an das zweite Kommando weitergereicht. Mit Fachwörtern

formuliert: Standard Output des ersten Kommandos wird Standard Input des zweiten. In der Man Page zu `grep` steht oben in DESCRIPTION, daß von Standard Input gelesen wird, wenn wie hier kein Filename als Argument für `grep` angegeben wird. Also sucht `grep` in der Ausgabe von `ls -l` nach `Auf`, `auf` usw.

4. Aufgabe: `ps`

Die einzelnen Spalten der Ausgabe werden unter DISPLAY FORMATS beschrieben.

5. Aufgabe: Prozeßtabelle zurechtschneidern

Mit `head` kann man sich die ersten Zeilen, mit `tail` die letzten Zeilen einer mehrzeiligen Ausgabe anzeigen lassen. Mit `cut` kann man statt dessen einzelne Spalten ausfiltern.

Das vorletzte und das letzte Kommando haben verschiedene Ergebnisse, weil `grep` auf verschiedene Eingaben angewendet wird: Beim letzten Kommando ist die Spalte, in der 0:00 vorkommt, schon ausgefiltert, so daß `grep` sie gar nicht zu Gesicht bekommt.

6. Aufgabe: `kill`

Mit dem Kommando `kill` kann man offenbar das Verhalten laufender Prozesse vom einem `xterm` aus beeinflussen, z.B. kurzfristig stoppen (STOP), wieder starten (CONT) oder ganz beenden (HUP und KILL). Der `emacs` fängt HUP ab und erstellt erst eine Sicherheitskopie jedes nicht abgespeicherten Files, bevor er sich selbst beendet. Wenn man später dieses File wieder in einen `emacs` einladen will, wird erst geprüft, ob es eine solche Sicherheitskopie gibt, und man wird darauf hingewiesen. (Diese Sicherheitskopie heißt übrigens `#dummyneu#`.) Bei KILL tut er das nicht, weil Signal KILL grundsätzlich nicht abgefangen werden kann.

7. Aufgabe: `man kill`

Die oben für die 6. Aufgabe gegebene Antwort auf die Frage, was die Aufgabe von `kill` ist, läßt sich mit Hilfe der Man Page klarer fassen: `kill` schickt ein Signal an den Prozeß, dessen ID als Argument angegeben wird.

Wenn man kein Signal spezifiziert, wird Signal TERM genommen.

Die Man Page ist signal(5), also Aufruf: `man -s 5 signal`.

8. Aufgabe: `grep` und Man Pages

Ein Hop ist im Prinzip der Transport einer Mail von einem lokalen Netz zum nächsten. Die Anzahl von Hops, die eine Mail durchlaufen kann, ist beschränkt, weil man sonst aus Versehen leicht Mails unendlich lange auf Zykeln herumschicken könnte.

Die durchge„grep“ten Files sind alle Files in der Directory /usr/man/man1.

9. Aufgabe: `find`

Mit `find` kann man - ausgehend von einer Directory - diese Directory sowie alle ihre Unterdirectories und Unterunterdirectories und ... durchsuchen und eine Operation auf alle Files darin anwenden, die eine bestimmte Eigenschaft erfüllen. Wird keine Operation spezifiziert, so wird einfach der Name des Files ausgegeben.

Durch `-name Aufgabe1` wird dafür gesorgt, daß nur die Files mit Namen `Aufgabe1` in Betracht kommen.

Durch `-type f` kommen nur die Files vom Typ `f`, das heißt normale Files (keine Directories, Symbolic Links etc.) in Betracht.

Durch `-print` werden die Namen der Files ausgegeben.

Option `-exec` ist dafür da, eine Operation zu spezifizieren, die auf alle in Betracht kommenden Files angewendet wird. Zum Beispiel wird durch `-exec wc -l {} \;` die Zahl der Zeilen in jedem dieser Files protokolliert.

Das `{}` hinter `-exec` ist der Platzhalter für den Namen des Files.

Der letzte Satz in der Beschreibung von `-exec` weist darauf hin, daß das Semikolon auch für die Shell (`xterm`) eine Bedeutung hat, so daß man die Shell mit `\` dazu zwingen muß, das Semikolon einfach an `find` weiterzureichen anstatt es selbst auszuwerten.

Laut Man Page steht `expression` für die Gesamtheit aus `options`, `tests` und `actions` (suchen Sie nach der Überschrift EXPRESSION). Unter TESTS steht, daß `-type f` ein Test ist, und unter ACTIONS steht, daß `-exec ... \;` und `-print` Actions sind. Also ist `expression`:

```
-type f -exec wc -l {} \; -print
```

10. Aufgabe: `find` und `grep` und Man Pages

Jedes UNIX-Kommando hat einen Exit Status, der am Ende der Abarbeitung des Kommandos an das aufrufende Programm zurückgereicht wird. Wurde das Kommando ganz normal aus dem `xterm` aufgerufen, bekommt die Shell den Exit Status zu sehen, und wenn man keine besonderen Vorkehrungen trifft, „vergißt“ die Shell ihn einfach wieder, und er hat keinen Effekt. In der 10. Aufgabe ist `grep` nicht direkt, sondern indirekt durch `find` aufgerufen worden, und `find` wertet den Exit Status von `grep` aus: `-exec` ergibt für ein File wahr genau dann, wenn `grep` Exit Status 0 hat, und das ist genau dann der Fall, wenn die gesuchte Zeichenkette mindestens einmal im File gefunden wurde.

Mit `-print` werden also im Kommandoaufruf in der 10. Aufgabe alle „normalen“ Files in `/usr/man` und darunterliegenden Directories ausgegeben, die die Zeichenkette `hop` in beliebiger Groß- und Kleinschreibung enthalten.

Würde `-print` vor `-exec` stehen, dann würden alle normalen Files in diesen Directories durch `-print` ausgegeben, weil erst nach Abarbeitung von `-print` der Filter `-exec ... \;` angewendet würde.

11. Aufgabe: Wildcards an Kommandos weiterreichen

Das Wildcard hat für die Shell eine Sonderbedeutung, nämlich als Platzhalter für eine beliebig lange Zeichenkette in Namen von Files, Directories etc.

Wenn ein Zeichen eine Sonderbedeutung für die Shell hat, kann man ihm allgemein diese Sonderbedeutung nehmen, indem man ein `\` unmittelbar davor schreibt. Dann wird das Zeichen von der Shell wie jedes andere behandelt, d.h. als Teil des Arguments an das Kommando weitergereicht. Der `\` selbst wird nicht an das Kommando weitergereicht, sondern von der Shell „verschluckt“.

Schreibt man ein `\` vor ein Zeichen, das keine Sonderbedeutung hat, wird der `\` ignoriert. Das Zeichen `\` heißt Backslash und ist nicht zu verwechseln mit dem Slash `/`.

Schließt man eine ganze Zeichenkette in Hochkommas ('Quotes') oder Anführungszeichen ("Double Quotes") ein, so hat das im Prinzip denselben Effekt, als wenn man ein Backslash vor jedes einzelne

Zeichen geschrieben hätte. (Die Unterschiede werden später genauer behandelt.) Die umgekehrten Anführungszeichen heißen übrigens 'Back Quotes'.

12. Aufgabe: Whitespaces

Auch das Leerzeichen (Blank) hat für die Shell eine Sonderbedeutung. Hier bricht sie nämlich die Kommandozeilen in einzelne Zeichenketten auf, und diese Zeichenketten reicht sie einzeln als Argumente an das Kommando weiter. In der ersten Kommandozeile bekommt `grep` also drei Argumente: `folgende`, `Kommandos` und `~weihe/unix96/Aufgabe5`. Laut Man Page zu grep wird `Kommandos` als Filename interpretiert.

Der erste Aufruf von `grep` sucht also die Zeichenkette `folgende` in den Files `Kommandos` und `~weihe/unix96/Aufgabe5`. Da es das File `Kommandos` nicht gibt, wird eine Fehlermeldung ausgegeben. Das ist die erste Zeile der Ausgabe. Zugleich werden aus dem zweiten File, also in `~weihe/unix96/Aufgabe5`, alle Zeilen mit `folgende` ausgegeben.

Wenn man es nicht durch Angabe geeigneter Optionen anders macht, dann wird der Name des Files genau dann vor jeder Zeile ausgegegeben, wenn `grep` mehr als ein File als Argument bekommen hat. Dies ist leider nicht sehr gut dokumentiert. Ein Hinweis findet sich in der üblichen Man Page in der Beschreibung der Option `-h`. Man kann es auch exakt nachlesen:

```
% man -l grep
grep (1)        -M /usr/local/man
grep (1)        -M /usr/man
% man -M /usr/man grep
```

Hier zeigt sich das generelle Problem, daß Dokumentation niemals ganz perfekt ist!

13. Aufgabe: Anführungszeichen mit `grep` suchen

In der ersten Kommandozeile werden die Anführungszeichen durch die Shell behandelt: Sie bedeuten, daß die Shell alles innerhalb von ... wörtlich an das Programm `grep` als Argument weiterreicht, ohne Zeichen mit Sonderbedeutung irgendwie gesondert zu behandeln. Das ist in diesem Fall egal, da innerhalb von ... sowieso kein besonderes Zeichen steht.

In der zweiten Kommandozeile hingegen haben auch die Anführungszeichen keine Sonderbedeutung mehr für die Shell und werden mit an `grep` weitergereicht. Daher sucht das zweite Kommando nach `"hop"`, während das erste nur nach `hop` sucht (jeweils mit beliebiger Groß- und Kleinschreibung; wodurch wurde das noch erreicht?).

UNIX/C++–Praktikum

Donnerstag, 7.11.1996

Heute:

- Prozesse unter UNIX,
- Informationsquellen für UNIX, X11 und Internetzugänge,
- insbesondere *Man Pages*.

Weitere Aufgabe für UNIX–Kernel: Prozesse verwalten

- Mehrere Prozesse können gleichzeitig ablaufen,
- auch Prozesse von verschiedenen Benutzern.

Prozeßverwaltung durch Kernel:

- CPU hat nur einen (oder ein paar) Prozessor(en).
- Jedem Prozeß wird reihum CPU für kurze Zeitspanne zugeteilt.
- Kernel verteilt CPU–Zeit auf alle Prozesse „gerecht“,
- beachtet dabei aber verschiedene Prioritäten von Prozessen.

Attribute von Prozessen:

- Name.
- Besitzer.
- Eindeutige Prozeß–ID (PID).
- Prozeß–ID des erzeugenden Prozesses (PPID=Parent Process ID).
- Priorität.
- Anfangszeitpunkt.
- Bisheriger Verbrauch von CPU–Zeit.
- Current Working Directory.
- Status, z.B.:

  Running: Läuft auf einer CPU.

  Runnable: Ist in der Warteschlange.

  Sleeping: Wartet auf einen Event.

  Zombie: Prozeß ist beendet, aber aufrufender Prozeß hat (noch) nicht Terminierung bestätigt.

Daemons: Prozesse, die meist

- beim Hochfahren des Systems gestartet werden,
- nie gestoppt werden, solange das System läuft,
- User root gehören.

$\Rightarrow$ Man ist nie allein auf dem Rechner.

Einer der Gründe (nicht der einzige) dafür, daß man Workstations **nicht** ausschaltet.

Beispiele für Daemons:

- UNIX–Kernel,
- X11–Server,
- Mail–Daemon.

Prozeßtabelle (Ausschnitt):

```
weihe@horn ~ % ps -elf | head -1 | cut -b 7-25,76-

    UID   PID  PPID    TIME COMD

weihe@horn ~ % ps -elf | tail -20l | cut -b 7-25,76-

brandes 12089 12052    0:00 xbiff -geometry +0-0
stoltze  8929  8913    0:00 tail -f /var/adm/messages
   root  7477 13016    0:02 <defunct>
 derman 12550 12529    0:01 xbiff -geometry -175+0
  weihe 13638 13636    0:02 -tcsh
 nobody 14274   504    0:00 <defunct>
   janz 13241     1    0:00 /usr/local/bin/tcsh /home/xinf/janz
   root 10248 13016    0:02 <defunct>
   root 14777   381    0:00 lpNet
   root 17366     1    0:01 /usr/lib/ncs/bin/glbd -create -firs
   root  7466 13016    0:02 <defunct>
koeppel 10267 10265    0:01 xterm -name chkintr -title horn -fn
 derman 12627 12547    0:01 /usr/openwin/bin/cm
   root 11876   495    0:00 /usr/openwin/bin/xdm -server /usr/o
 derman 12549 12529    0:00 xterm -geometry +150+300
   root  6399 13016    0:02 <defunct>
   root 13636   321    0:01 in.rlogind
 derman 12570 125292   0:00 rlogin horn
 derman 12551 12529    0:00 xclock -geometry -0+0
brandes 12104 12094    0:00 rlogin horn
```

Erzeugung eines Prozesses:

Elter–Prozeß $\rightarrow$ Kind–Prozeß.

1. System Call „*fork*“ von Elter an Kernel.

   $\Rightarrow$ Kernel richtet Kind ein, der bis auf Prozeß–ID exakte Kopie des Elter ist.

   $\Rightarrow$ Zunächst läuft gleiches Programm in Elter und Kind und ist im gleichen Zustand. Auch Attribute der Prozesse sind exakt gleich, z.B. Current Working Directory (Ausnahme: PID).

2. System Call „*exec*“ von Kind an Kernel mit Namen des auszuführenden Programms.

   $\Rightarrow$ Kernel überschreibt Programm von Kind mit auszuführendem Programm.

Vorteil: Kind–Prozeß erbt alle Attribute des Elter–Prozesses (kann sie auch teilweise überschreiben).

Prozeß beenden oder Status ändern: **Signale**

- Prozesse können einander Signale schicken.
- Für jedes Signal gibt es Default–Reaktion.
- Programme können Default–Reaktion durch spezifische Reaktion überschreiben.

Wichtige Signale:

HUP (Hangup) Wird beim Ausloggen an alle noch laufenden Prozesse geschickt.
Default–Reaktion: Terminierung.
Emacs–Reaktion: erst Sicherheitskopien aller offenen Files ziehen, dann terminieren.

STOP *Default*–Reaktion: Prozeß wird schlafengelegt.

CONT (Continue) *Default*–Reaktion: Schlafengelegter Prozeß wird wieder aufgeweckt.

KILL *Default*–Reaktion: Terminierung.
Besonderheit: Default–Reaktion kann nicht überschrieben werden.

WINCH (Window Change) Wird von Window Manager an Prozeß geschickt, wenn Größe des Fensters verändert wird.

Signal schicken von der Shell aus:

1. PID mit Kommando ps herausfinden.
2. kill –*Signalname PID*

Beispiel:

```
weihe@hoyren ~ % ps -e | grep elm
  1561 pts/3    0:01 elm
weihe@hoyren ~ % kill -HUP 1561
weihe@hoyren ~ % ps -e | grep elm
  1561 pts/3    0:01 elm
weihe@hoyren ~ % kill -KILL 1561
weihe@hoyren ~ % ps -e | grep elm
weihe@hoyren ~ %
```

Warnung: Immer erst mit einem „weichen“ Signal wie HUP zuschlagen, nur wenn das nichts hilft, mit KILL!

Informationen on–line

1. Schritt: Namen des Kommandos mit apropos herausfinden

```
weihe@hoyren ~ % apropos calendar
ical          ical (1)         - X based calendar program
cal           cal (1)          - display a calendar
calendar      calendar (1)     - reminder service
difftime      difftime (3c)    - computes the difference
                                 between two calendar times
mktime        mktime (3c)      - converts a tm structure to
                                 a calendar time
cm            cm (1)           - calendar manager, appointment
                                 and resource scheduling tool
cm_delete     cm_delete (1)    - delete appointments from
                                 Calendar Manager database
cm_insert     cm_insert (1)    - insert appointments into
                                 Calendar Manager database
cm_lookup     cm_lookup (1)    - look up appointments from
                                 Calendar Manager database
rpc.cmsd      rpc.cmsd (1)     - calendar manager service
                                 daemon

weihe@hoyren ~ % man cal
```

2. Schritt: ManPage

```
cal(1)                    User Commands                cal(1)

NAME
   cal - display a calendar
SYNOPSIS
   cal [ [ month ] year ]
AVAILABILITY
   SUNWesu
DESCRIPTION
   cal prints a calendar for the specified year.  If a
   month  is also  specified,  a calendar  just for  that
   month is printed.  If neither  is specified,  a calendar
   for the present month is printed.  The calendar produced
   is that for England and the United States.
OPTIONS
   month     The month is a number between 1 and 12.
   year      The year can be between 1 and 9999.
SEE ALSO
   calendar(1)
NOTES
   An unusual calendar is printed for September 1752.  That
   is the  month  11 days were skipped  to make up for lack
   of leap year adjustments.  To see this calendar, type:
        cal 9 1752
   The command cal 83 refers to the year 83, not 1983.  The
   year is always  considered  to  start  in  January  even
   though this is historically naive.
```

3. Schritt: Kommando gemäß ManPage anwenden.

Beispiel: Kalender für November 1994.

Relevante Information in ManPage:

```
SYNOPSIS
   cal [ [ month ] year ]
   ...

NOTES
   ...

   The command cal 83 refers to the year 83, not
   1983.  The year is always considered to start
   in January  even though this is  historically
   naive.
```

Also:

```
weihe@hoyren ~ % cal 11 1994
   November 1994
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Wichtige Kapitel von ManPages:

NAME: Kurzerklärung „auf einen Blick“.

DESCRIPTION: Ausführlichere Erklärung.

OPTIONS: Erklärung aller Optionen und ihrer Argumente.

SYNOPSIS: Formen des Gebrauchs.

DIAGNOSTICS: Fehlermeldungen bei falschem Gebrauch.

BUGS: Fehler, die erkannt, aber noch nicht behoben sind.

FILES: Files, die das Programm benutzt.

SEE ALSO: Verwandte Kommandos.

AUTHOR: Urheber der Bugs (und vom Rest des Programms).

Kapitel NAME:

```
weihe@hoyren ~ % whatis cal
  cal       cal (1)   - display a calendar
```

```
weihe@hoyren ~ % apropos cal
  ...
  cal       cal (1)   - display a calendar
  ...
```

```
weihe@hoyren ~ % man cal

  cal(1)        User Commands     cal(1)

  NAME
     cal - display a calendar
     ...
```

Kapitel SYNOPSIS:

```
SYNOPSIS:
     cal [ [ month ] year ]
```

- Eckige Klammern: optionale Bestandteile.
- Bestandteil in inneren Klammern nur erlaubt, wenn auch Bestandteil in äußeren Klammern da ist.

```
SYNOPSIS:
   bc [ -lws ] [  file ... ]
```

- `-lws` heißt: Möglich sind alle Kombinationen, also -l, -w, -s, -lw, -ls, -ws und -lws.
- „`file ...`“ bedeutet nicht **ein** File, sondern beliebig viele (ohne eckige Klammern: mindestens eines, mglw. mehr).

Sektionen und Untersektionen von Man-Pages:

1. Kommandos: UNIX, X11, Sun–spezifisch, einige weitere.

   Beispiele für Untersektionen von Sektion 1:

   1b: BSD–spezifisch,

   1m: Kommandos für Systemadministration,

   1s: Sun–spezifisch.
2. System Call Library.
3. Funktionen für C–Programme.
4. Allgemein verwendete Fileformate.
5. Spezifische Konstanten für Rechner und System (z.B. ASCII–Zeichensatz, Signal–Tabelle).
6. Spiele und Demonstrationen.
7. Beschreibung von Devices und anderen speziellen Files.

Woher Man Pages?

- Globale Variable „*MANPATH*“.
- Erste in Liste gefundene Man Page wird genommen.

```
weihe@hoyren ~ % echo $MANPATH
  /usr/local/man:/usr/man:/usr/openwin/man:
  /usr/local/X11R6/man:/usr/local/TeX/man:
  /opt/SUNWspro/man

weihe@hoyren ~ % man -l bc
  bc (1)  -M /usr/local/man
  bc (1)  -M /usr/man

weihe@hoyren ~ % ls /usr/man
  cat1    cat3g   cat5    man1b   man3i   man6
  cat1b   cat3i   cat6    man1c   man3k   man7
  cat1c   cat3k   cat7    man1f   man3m   man8
  cat1f   cat3m   cat9    man1m   man3n   man8c
  cat1m   cat3n   cat9e   man1s   man3r   man9
  cat1s   cat3r   cat9f   man2    man3s   man9e
  cat2    cat3s   cat9s   man3    man3t   man9f
  cat3    cat3t   catl    man3b   man3x   man9s
  cat3b   cat3x   catn    man3c   man4    manl
  cat3c   cat4    man.cf  man3e   man4b   mann
  cat3e   cat4b   man1    man3g   man5    windex
```

Einführende Man Pages:

```
weihe@hoyren ~ % man -l intro
  intro (1)       -M /usr/man
  intro (1m)      -M /usr/man
  intro (2)       -M /usr/man
  intro (3)       -M /usr/man
  intro (9f)      -M /usr/man
  intro (9s)      -M /usr/man
  intro (9e)      -M /usr/man
  intro (9)       -M /usr/man
  intro (4)       -M /usr/man
  intro (5)       -M /usr/man
  intro (7)       -M /usr/man
  intro (6)       -M /usr/man
  intro (1)       -M /opt/SUNWspro/man
  intro (3m)      -M /opt/SUNWspro/man
  intro (3f)      -M /opt/SUNWspro/man
```

Sektion direkt auswählen:

```
weihe@hoyren ~ % man -s6 intro
weihe@hoyren ~ % man -s1m intro
weihe@hoyren ~ % man -s1 -M /opt/SUNWspro/man intro
```

Wenn apropos nicht mehr hilft ...

Beispiel: Wofür ist File `~/.plan` gut?

```
weihe@hoyren ~ % apropos plan
```

⇒ Riesenliste, aber das richtige nicht dabei.

```
weihe@hoyren ~ % apropos .plan
```

⇒ Keine Ausgabe.

Also Holzhammermethode:

```
weihe@hoyren ~ % cd /usr/man/man1
weihe@hoyren /usr/man/man1 % grep -s "\.plan" *
  finger.1:.B \s-1$HOME\s0/\|\.plan
  finger.1:Suppress printing of the \f4.plan\f1

weihe@hoyren /usr/man/man1 % man finger
```

```
NAME
  finger - display information about local and remote users

...


DESCRIPTION
  ...

  When one or more username arguments are given, more detailed
  information  is  given  for each username specified, whether
  they are logged in or not.  username must be that of a local
  user,  and  may be a first or last name, or an account name.
  Information is presented in multi-line format as follows:
    o   the user name and the user's full name
    o   the user's home directory and login shell
    o   time the user logged in if currently logged in,  or
        the  time  the user last logged in; and the terminal
        or host from which the user logged in
    o   last time the user received mail, and the last time
        the user read mail
    o   the first line of the $HOME/.project  file,  if  it
    o   the contents of the $HOME/.plan file, if it exists

  ...
```

Wo ist das Programm?

```
weihe@hoyren ~ % which cal
  /usr/bin/cal

weihe@hoyren ~ % whereis cal
  cal: /usr/bin/cal /usr/man/man1/cal.1

weihe@hoyren ~ % echo $PATH
  .:/home/inf/weihe/bin:/usr/local/bin:
  /opt/SUNWspro/bin:/usr/bin:/usr/sbin:
  /usr/ccs/bin:/usr/ucb:/usr/openwin/bin:
  /usr/local/X11R6/bin:/usr/local/TeX/bin:
  /usr/local/hotmetal/bin:/net/frame/bin:
  /usr/local/lib/mocka/bin
```

Shell nimmt erstes Vorkommen von „*cal*“ in den Directories in globaler Variable PATH.

Directory „.“: Current Working Directory.

Eingebaute Kommandos in der Shell:

```
weihe@hoyren ~ % whatis cd
  cd      cd (1)    - shell built-in function to change
                      the current working directory
```

Eingebautes Kommando: Kein eigenständiges Programm, sondern Unterprogramm der Shell, das ohne eigenen Prozeß abgearbeitet wird.

```
weihe@hoyren ~ % echo $SHELL
  /usr/local/bin/tcsh

weihe@hoyren ~ % man tcsh

  ...

  SEE ALSO
     csh(1),  emacs(1),  ls(1),  newgrp(1),  sh(1),
     setpath(1),  stty(1),  su(1), tset(1),  vi(1),
     x(1), access(2), execve(2), fork(2),  killpg(2),
     pipe(2),   setrlimit(2),   sigvec(2),   stat(2),
     umask(2),  vfork(2),  wait(2),  malloc(3),  set-
     locale(3),  tty(4),  a.out(5),  termcap(5),  en-
     viron(7),  termio(7), Introduction to the C Shell
```

```
weihe@hoyren ~ % man csh

  ...

  ...

  ...

  SEE ALSO
     login(1),   ps(1),   sh(1),
     shell_builtins(1),  tset(1B),
     access(2),  exec(2), fork(2),
     pipe(2), a.out(4), environ(4),
     environ(5), ascii(5), termio(7)


weihe@hoyren ~ % man shell_builtins
```

```
  shell_builtins(1)   User Commands   shell_builtins(1)

  NAME
    shell_builtins - shell command interpreter
    built-in functions

  DESCRIPTION
    The shell command interpreters (sh(1), csh(1),
    and  ksh(1)), have special built-in functions
    which are interpreted by the shell as commands.
    Many of these built-in commands are implemented
    by more than one of the shells, and some are
    unique to a particular shell.  These are:

      alias       bg          break       case
      cd          chdir       continue    dirs
      eval        exec        exit        export
      fc          fg          for         foreach
      function    getopts     glob        goto
      hash        hashstat    history     if
      jobs        kill        let         limit
      logout      newgrp      notify      onintr
      popd        print       pushd       read
      readonly    rehash      repeat      return
      select      set         setenv      shift
      source      stop        suspend     switch
      test        times       trap        type
      typeset     ulimit      umask       unalias
      unhash      unlimit     unset       unsetenv
      until       wait        whence      while
```

```
weihe@hoyren ~ % which cd
  cd: shell built-in command.

weihe@hoyren ~ % whereis cd
  cd: /usr/man/man1/cd.1

weihe@hoyren ~ % which koala
  koala:   aliased to
           telnet polydos.uni-konstanz.de 775

weihe@hoyren ~ % whereis koala
  koala:
```

Alias: Einfache Abkürzung für langen, komplizierten Ausdruck.

Interaktive Definition:

- Tcsh:
  `alias koala "telnet polydos.uni-konstanz.de 775"`
- Bash:
  `alias koala="telnet polydos.uni-konstanz.de 775"`

Wdh.: `echo $SHELL` liefert Namen der Shell.

Vordefinierte Aliasse für jedes Xterm:

- Tcsh: File `~/.cshrc` — Ressource–File für Csh und Tcsh.
- Bash: File `~/.bashrc` — Ressource–File für Bash.

Kommando telnet:

```
telnet(1)    User Commands      telnet(1)

NAME
     telnet - user interface to a remote
     system using the TELNET protocol

SYNOPSIS
     telnet [ host [ port ] ]
```

Host: Rechnername auf dem Internet (hier: polydos.uni–konstanz.de).

Port: Spricht einen speziellen Daemon auf Host an (hier: Koala–Daemon auf Port 775).

6 Generalthema heute: weiteres zur Arbeit mit Files

1. Aufgabe: Output-Redirection

Gehen Sie in Ihre Home Directory und geben Sie das Kommando

```
ls -l > xyz
```

ein. Was steht hinterher im File `xyz`? Geben Sie jetzt nacheinander folgende Kommandos ein und schauen Sie sich zwischendurch immer wieder den Inhalt von File `XYZ` an.

```
ps -elf > XYZ
ls -l > XYZ
ls -l >! XYZ
ps -elf >> XYZ
```

Achtung: Wenn sich der Inhalt von `XYZ` ändert, während man sich das File mit `less`, `emacs` oder sonstwas anschaut, ändert sich die Anzeige nicht unbedingt. Nach jedem Kommando muß also `less` bzw. `emacs` neu aufgerufen werden.

2. Aufgabe: `diff`

Kopieren Sie das File `~weihe/unix96/Aufgabe6` in Ihre Home Directory und nehmen Sie an Ihrer Kopie ein paar kleine Änderungen vor: Zeilen löschen, Zeilen einfügen und ein paar Zeilen verändern. Rufen Sie dann auf:

```
diff ~weihe/unix96/Aufgabe6 ~/Aufgabe6
```

In welchem Verhältnis steht die Ausgabe des Kommandos zu Ihren Änderungen? Was bedeuten insbesondere die `<` und `>` am Anfang von Zeilen? Was bedeuten die zusätzlich angegebenen Nummern?

Zwischen den zusätzlichen Nummern finden Sie auch einzelne Kleinbuchstaben (`a`,`c`,`d`...). Versuchen Sie zunächst, aus der Ausgabe des obigen Kommandoaufrufs die Bedeutung zu erraten. In der Man Page zu `diff` finden Sie dazu einen Verweis auf ein anderes Kommando. Schauen Sie sich die Man Page zu diesem Kommando an und versuchen Sie, die Stelle zu finden, an der diese Buchstaben erklärt werden.

Hinweis: Lesen Sie die Man Page nicht konzentriert von vorne bis hinten, sondern versuchen Sie, die richtige Stelle durch ein schnelles Durch„browsen“ zu finden. Ein solcher Lesestil ist wichtig für das effiziente Arbeiten mit Dokumentation aller Art und kann nur durch Übung erreicht werden.

3. Aufgabe: `tr`

Geben Sie folgende Kommandos ein:

```
ps -elf
ps -elf | tr "[a-z][A-Z]" "[A-Z][a-z]"
```

Wiederholung von Aufgabe5: Was macht das Kommando `ps` und was machen die Optionen `e`, `l`, `f`?

Worin unterscheiden sich die Ausgaben der beiden Kommandos? Wie läßt sich der Unterschied in einem Satz zusammenfassen?

Worin unterscheiden sich die Ausgaben der folgenden zwei Kommandos:

```
ps -elf | tr "[0-9]" "[a-j]"
ps -elf | tr "[0-9]" "[a-c]"
```

Geben Sie nun ein:

```
ps -elf | tr -d "[0-3]"
```

Was tut also Option `d`? Vergleichen Sie Ihre Antwort mit der Erklärung in der Man Page.

4. Aufgabe: Input Redirection

Geben Sie nun ein:

```
tr "[a-z]" "[A-Z]" < ~weihe/unix96/Aufgabe6 > ~/Aufgabe6
```

Was steht danach in `~/Aufgabe6`? Was bewirkt also `<` ?

Warnung: Schreiben Sie niemals `<` und `>` (oder `>!` oder `>>`) in einem Kommandoaufruf den Namen desselben Files! Das bewirkt nicht das, was man damit beabsichtigt, sondern dadurch geht der Inhalt des Files verloren.

5. Aufgabe: `ps` und `sort`

Geben Sie folgende Kommandos ein:

```
ls -l ~weihe/unix96
ls -l ~weihe/unix96 | sort
```

Nach welchen Kriterien ist die Ausgabe durch `sort` anscheinend sortiert worden? Schauen Sie in der Man Page von `sort` nach, nach welchen Kriterien `sort` sortiert, wenn keine Optionen angegeben werden.

Allgemeiner Hinweis: Für Verhalten ohne Optionen gibt es im Englischen das Wort *default*.

Was könnte *lexicographic* meinen?

Geben Sie jetzt folgende Kommandos ein:

```
ls -l ~weihe/unix96 | sort +4 -5
ls -l ~weihe/unix96 | sort +5 -6
ls -l ~weihe/unix96 | sort -M +5 -6
ls -l ~weihe/unix96 | sort -r -M +5 -6
```

Welche Spalte(n) erscheint/-en jetzt jeweils sortiert und nach welchen Kriterien? Vergleichen Sie mit den Erklärungen in der Man Page.

6. Aufgabe: Sortieren ohne Blanks

Geben Sie folgende Kommandos ein:

```
ps -e | tail +2 | sort +0 -1
ps -e | tail +2 | sort -b +0 -1
```

Im beiden Fällen ist die Ausgabe nach der gleichen Spalte (nämlich der ersten) sortiert worden, aber nach verschiedenen Kriterien. Was ist das Kriterium beim zweiten Aufruf gewesen?

Schauen Sie in der Man Page zu `sort` nach, was Option `-b` tut bzw. was passiert, wenn man Option `-b` nicht angibt. Können Sie daraus schlußfolgern, ob das Blank eine kleinere oder eine größere ASCII-Nummer hat als die Ziffern 0-9?

Geben Sie folgendes Kommando ein:

```
ps -e | sort
```

Wo findet sich die Kopfzeile der Tabelle in der Ausgabe? Können Sie daraus schlußfolgern, wie sich die ASCII-Nummern von Großbuchstaben im Vergleich zu den ASCII-Nummern von Ziffern einordnen?

7. Aufgabe: numerisch sortieren

Geben Sie folgende Kommandos ein:

```
du ~weihe > ~/du_output
sort +0 -1 ~/du_output | less
sort +0 -1 -n ~/du_output | less
```

Haben Sie eine Idee, wieso das erste Kommando Ausgaben auf den Bildschirm geschrieben hat, wo doch die Ausgaben mit `>` umgelenkt worden sind?

Die Ausgabe ist offenbar wieder nach der ersten Spalte sortiert, aber nicht lexikographisch. Sondern? Schlagen Sie in der Man Page zu `sort` nach, was Option `n` macht.

8. Aufgabe: `cat` und `wc`

Geben Sie folgende Kommandos ein:

```
wc ~weihe/unix96/Aufgabe?
cat ~weihe/unix96/Aufgabe? | wc
```

Wiederholung von `Aufgabe5`: Was macht Kommando `wc`?

Was haben die beiden Ausgaben miteinander zu tun? Sehen Sie sich dazu DESCRIPTION in der Man Page von `cat` an. Was hat `cat` also mit den Files `Aufgabe1`, `Aufgabe2` ... gemacht?

Wie kann man also mit `cat` und `wc` herausfinden, wieviel Platz in Bytes alle Files in einer Directory zusammen belegen?

9. Aufgabe: `cat` und `sort`

Geben Sie folgendes Kommando ein:

```
cat ~weihe/unix96/Aufgabe? | sort -u | less
```

Was ist das Ergebnis?

Geben Sie jetzt folgendes Kommando ein:

```
cat ~weihe/unix96/Aufgabe? | sort | less
```

Jetzt müßte das `xterm` ziemlich leer sein. Suchen Sie im `less` jetzt dennoch die Zeichenkette `Laufen` und scrollen Sie dann ein bißchen hin und her. Schauen Sie in der Man Page zu `sort` nach, was Option `-u` tut. Danach müßte Ihnen eigentlich klar werden, warum die zweite Ausgabe so merkwürdig ist und die erste nicht.

Geben Sie diese beiden Kommandos noch einmal ein, nur ersetzen Sie `less` jetzt jeweils durch `wc`. Sind die beiden Ergebnisse signifikant unterschiedlich?

10. Aufgabe: `grep` und Regular Expressions

Gehen Sie nach `/usr/man/man1` und geben Sie folgendes Kommando ein:

```
grep -si .plan *
```

Werden nur Zeilen ausgegeben, die die Zeichenkette `.plan` enthalten? Was sonst ist allen ausgegebenen Zeilen gemeinsam?

Suchen Sie in der Man Page zu `grep` den Abschnitt über Regular Expressions.

Verständnisfragen:

- Was bedeutet ein `*` innerhalb von Regular Expressions? Wird der `*` von `grep` wirklich genauso interpretiert wie von der Shell?
- Wie muß man `.` und `*` kombinieren, um einen Platzhalter zu konstruieren, der wie `*` in der Shell für eine beliebige Zeichenkette von beliebiger Länge steht?
- Was muß man anstelle von `.plan` schreiben, wenn man sich nur die Zeilen ausgeben lassen will, in denen ein Kleinbuchstabe vor der Zeichenkette `plan` steht?

Probieren Sie es aus. Muß man wieder irgendwelchen Zeichen ihre Sonderbedeutung, die sie für die Shell haben, nehmen?

Geben Sie nun folgende Kommandos ein:

```
grep -si ".plan" *

grep -si \.plan *

grep -si "\.plan" *

grep -si \\.plan *
```

Warum funktionieren die letzten beiden Kommandos anders als die ersten beiden? Würden Sie eher eines der ersten beiden Kommandos oder eher eines der letzten beiden Kommandos wählen, um in den Man Pages nach Erklärungen zum File `.plan` zu suchen?

11. Aufgabe: Backslash pur

Geben Sie folgende Kommandos ein:

```
grep -i \ ~weihe/unix96/Aufgabe5    ( <- gefolgt von Control-c)

grep -i \  ~weihe/unix96/Aufgabe5

grep -i \\ ~weihe/unix96/Aufgabe5

grep -i "\\" ~weihe/unix96/Aufgabe5
```

Können Sie sich aus dem bisher Gesagten das Verhalten der einzelnen Kommandos erklären?

12. Aufgabe: Zugriffsrechte und `chmod`

Legen Sie noch einmal eine Directory `~/Aufg` an und kopieren Sie das File `~weihe/unix96/Aufgabe1` hinein. Sehen Sie sich mit `ls -l` die Zugriffsrechte von `Aufg` und `Aufgabe1` an. Das sind die Zeichen 2-10 in jeder Zeile.

Geben Sie jetzt folgende Kommandos ein, wobei Sie in Ihrer Home Directory stehen. Beobachten Sie dabei, wie sich die Anzeige der Zugriffsrechte durch `ls -l` ändert.

Kommandos im einzelnen:

```
ls -l Aufg
ls -ld Aufg
chmod u-r Aufg
ls -l Aufg
ls -ld Aufg
cd Aufg
cd ..
chmod u-x Aufg
ls -l Aufg
ls -ld Aufg
cd Aufg
chmod g-rx Aufg
ls -ld Aufg
chmod a-rwx Aufg
ls -ld Aufg
chmod u+rwx Aufg
ls -ld Aufg
chmod go+rx Aufg
ls -ld Aufg
```

Verständnisfragen:

- Was macht Option `-d` bei `ls`?
- Der Name `chmod` ist die Abkürzung für „change mode“. Was ist mit mode (also Modus) gemeint? Schauen Sie dazu in der ManPage zu `chmod` nach.
- Was ist der Unterschied zwischen `+` und `-` bei `chmod`?
- Was bedeuten die Buchstaben unmittelbar vor dem `+` bzw. `-` ? Was bedeutet insbesondere der Buchstabe `a`?
- Was bedeuten die Buchstaben hinter dem `+` bzw. `-` ?

WARNUNG: Ändern Sie nicht die Zugriffsrechte Ihrer Home Directory! Falls Sie irgendetwas Geheimes haben, legen Sie dafür in Ihrer Home Directory eine eigene Unterdirectory an, für die Sie allen anderen Usern alle Zugriffsrechte wegnehmen.

13. Aufgabe: Jargon File

Sehen Sie sich `~weihe/unix96/jarg320.txt` an und beantworten Sie sich die folgenden (natürlich todernsten) Fragen:

- Warum ist UNIX ein Virus (suchen Sie nach der Zeichenkette `:UNIX conspiracy`)?
- Wie sehen typische C-Programme aus (suchen Sie nach `-\_-`) ?
- Wie zeichnet man Bilder, wenn man nur Buchstaben hat (suchen Sie nach `ASCII art`) ?
- Wozu ist ein Schlips gut (suchen Sie nach `:suit`) ?
- Was war Biff für eine Rasse?
- Im Jargon File steht auch eine (mehr oder weniger) englische Benutzungsordnung (suchen Sie nach ATTENTION).

Suchen Sie nun nach eigenem Gusto nach weiteren bedeutsamen Informationen. Übrigens: Das Jargon File ist unter dem Namen „New Hacker's Dictionary“ als Buch erschienen und gehört zum Semester-Apparat des G 229.

6 Antworten zum Generalthema: weiteres zur Arbeit mit Files

1. Aufgabe: Output-Redirection

Mit `>`, `>!` und `>>` kann man die Ausgabe eines Kommandos (genauer: Standard Output) auf ein File umlenken. Unterschiede zwischen diesen drei Möglichkeiten zeigen sich, wenn das File schon existiert: `>` verweigert die Ausführung, `>!` führt ohne nachzufragen aus und überschreibt den alten Inhalt, `>>` führt ebenfalls aus, hängt die Ausgabe aber hinten an den alten Inhalt an.

2. Aufgabe: `diff`

Das Kommando `diff` erwartet zwei Files als Argumente und listet die Unterschiede in den Inhalten auf. Diese Auflistung besteht im wesentlichen aus Zeilen der beiden Files, in denen Unterschiede festgestellt wurden. Mit `<` und `>` wird jeweils angezeigt, ob die angezeigte Zeile so im ersten bzw. zweiten File steht. Die begleitenden Nummern sind die Zeilennummern.

In der Man Page zu `diff` ist ein Verweis auf das Kommando `ed`. In dessen Man Page findet sich eine Erklärung unter anderem für `a`=append, `c`=change und `d`=delete. Suchen Sie in der Man Page nach `append`, um die Stelle zu finden. Wiederholung von `Aufgabe4`, 1. Aufgabe: Wie sucht man in einer Man Page nach einer Zeichenkette?

3. Aufgabe: `tr`

Mit `tr` kann man alle Vorkommen eines bestimmten Zeichens durch ein anderes Zeichen ersetzen. Man kann auch mehrere Zeichen in einem Aufruf ersetzen, also z.B.

```
tr "[a-z]" "[A-Z]"
```

anstelle von

```
tr a A | tr b B | ... | tr z Z
```

Die Anführungszeichen bei `"[a-z]"` etc. sind notwendig, damit die Shell die eckigen Klammern nicht selbst auswertet, sondern als Bestandteile des Arguments unausgewertet an das Kommando weiterreicht (vgl. `Aufgabe5`).

Option `d` ersetzt keine Zeichen, sondern löscht sie. Bei Angabe von Option `d` macht es daher keinen Sinn, zwei Zeichenfolgen anzugeben.

4. Aufgabe: Input Redirection

Mit `<` kann man die Eingaben zu einem Kommando (genauer: Standard Input) von einem File lesen anstatt von der Tastatur einzugeben oder durch ein `|` hineinzureichen, wie es in der 3. Aufgabe gemacht worden ist.

5. Aufgabe: `ps` und `sort`

Das Kommando `sort` ist dafür da, die Zeilen eines Files umzusortieren. Man kann das Sortierkriterium durch Optionen sehr flexibel wählen. „By default“ wird lexikographisch sortiert, d.h. eine Zeile X wird vor einer Zeile Y einsortiert, wenn das erste Zeichen von links, das in beiden Zeilen verschieden ist, in Zeile X eine kleinere ASCII-Nummer hat als in Zeile Y. Wie die Aufgabe zeigt, kann man nicht nur Zeilen von Files mit `sort` sortieren, sondern dank `|` auch die Zeilen von

Ausgaben anderer Kommandos. Die Kleinbuchstaben a-z haben aufsteigende, aufeinanderfolgende ASCII-Nummern, ebenso die Großbuchstaben A-Z unter sich sowie die Ziffern 0-9 unter sich. Deshalb werden diese Zeichen also in der richtigen „Telefonbuchordnung“ sortiert.

Mit den Optionen `+` und `-` kann man einzelne Spalten einer Tabelle als Sortierschlüssel verwenden. Option `M` sortiert englische Monatsnamen richtig, Option `r` kehrt die Sortierreihenfolge um.

6. Aufgabe: Sortieren ohne Blanks

Bei Option `b` werden führende Leerzeichen nicht als Sortierschlüssel mitverwendet. Da die Prozeß-Nummern ohne Angabe von Option `b` „richtig“ sortiert sind, muß das Blank eine kleinere ASCII-Nummer haben als die Ziffern 0-9, denn ansonsten würden die längsten Prozeß-Nummern zuerst kommen.

Großbuchstaben haben größere ASCII-Nummern als Ziffern.

7. Aufgabe: numerisch sortieren

Es gibt zwei Ausgabekanäle für Programme: `stdout` (Standardausgabe) und `stderr` (Standard-Error, d.h. Standardfehlerausgabe). Mit `>` wird nur `stdout` umgelenkt, und `stderr` schreibt weiter auf den Bildschirm.

Option `n` sortiert nicht lexikographisch, sondern versucht, die Daten in der Spalte, die zum Sortieren benutzt wird, als Zahlen zu verwenden und sortiert sie dem Betrage nach.

8. Aufgabe: `cat` und `wc`

Das Kommando `cat` ist dazu da, mehrere Files zu lesen und nacheinander auf Standard Output auszugeben. Mit `>` kann man also ein File aus mehreren anderen Files durch Aneinanderhängen erzeugen („konkatenieren“).

9. Aufgabe: `cat` und `sort`

Option `u` von `sort` sorgt dafür, daß keine zwei gleichen Zeilen ausgegeben werden.

10. Aufgabe: `grep` und Regular Expressions

Der Punkt im Argument `.plan` hat für `grep` eine Sonderbedeutung: Er steht als Platzhalter für jedes beliebige Zeichen. Es werden also alle Vorkommen von `plan` ausgegeben, vor denen noch ein anderes Zeichen steht, das heißt alle Vorkommen, die nicht ganz am Anfang der jeweiligen Zeile stehen.

Generell gilt: Die Zeichenkette, nach der mit `grep` gesucht werden soll, muß nicht wörtlich dastehen. Sie kann auch „verklausuliert“ dastehen als ein sogenannter *Regular Expression*. Das hat den Vorteil, daß man nicht eine spezielle Zeichenkette suchen muß, sondern man kann nach einer ganzen Klasse von Zeichenketten suchen, die etwas gemeinsam haben. Der Punkt ist in Regular Expressions Platzhalter für ein einzelnes Zeichen.

Der `*` in einem Regular Expression bedeutet, daß das vorhergehende Zeichen beliebig oft wiederholt vorkommen darf. Zum Beispiel paßt `a0*.tex` auf `a000.tex`, aber nicht auf `abc.tex` oder `aaa.tex`. Um eine beliebige Zeichenkette zu „matchen“, muß man die Kombination `.*` angeben.

Um alle Vorkommen von `plan` zu finden, denen unmittelbar ein Kleinbuchstabe vorangeht. gibt man `"[[:lower:]]plan"` ein. Durch die Double Quotes wird den eckigen Klammern wieder ihre Sonderbedeutung für die Shell genommen.

Generell gilt: Auch in Regular Expressions wird einem Sonderzeichen die Sonderbedeutung durch einen \ genommen (und in vielen anderen formalen Sprachen ebenfalls). Wenn wir allerdings ohne Anführungszeichen nur `\.plan` schreiben, dann wird der \ von der Shell interpretiert, nicht von `grep`! Das heißt, `grep` bekommt wieder als Argument `.plan` und nicht `\.plan`. Also muß dafür gesorgt werden, daß auch der \ von der Shell unangetastet als Teil des Arguments an `grep` weitergereicht wird. Das geht, indem man entweder dem \ selbst durch ein weiteres \ die Sonderbedeutung für die Shell nimmt oder indem man allen Zeichen mit `"..."` die Sonderbedeutung nimmt.

11. Aufgabe: Backslash pur

Beim ersten Kommando wird der \ durch die Shell interpretiert, nämlich um dem nachfolgenden Zeichen (einem Blank) die Sonderbedeutung zu nehmen. Beim ersten Kommando bekommt `grep` also nur ein Argument, nämlich
`~weihe/unix96/Aufgabe5` (beachten Sie das führende Blank!), und die Man Page zu `grep` sagt, daß dieses Argument als zu suchende Zeichenkette genommen wird und die Eingabe, in der gesucht werden soll, von Standard Input genommen wird. Daher wartet `grep` auf Eingaben von der Tastatur, und sie mußten `grep` „mit Gewalt“ abbrechen.

Der Unterschied beim zweiten Kommando ist, daß nun noch ein weiteres Blank zwischen „ “ und `~weihe/unix96/Aufgabe5` steht, dem nicht die Sonderbedeutung für die Shell genommen wurde. Die Sonderbedeutung für „ “ ist, daß einzelne Argumente voneinander getrennt werden. Also bekommt `grep` beim zweiten Aufruf zwei Argumente zu sehen und gibt brav alle Zeilen des Files `~weihe/unix96/Aufgabe5` aus, in dem ein Blank vorkommt (also so ziemlich alle).

Beim dritten Kommando bekommt `grep` als erstes Argument ein einzelnes `\`. Da ein \ für `grep` immer vor einem anderen Zeichen stehen muß, beschwert sich `grep`.

Das letzte Kommando schließlich sucht nach dem \ im File `~weihe/unix96/Aufgabe5`: Die `"..."` nehmen die Sonderbedeutung für die Shell, und `grep` bekommt \\ als erstes Argument. Der erste \ nimmt dem zweiten die Sonderbedeutung für `grep`, und `grep` sucht daher nach \.

12. Aufgabe: Zugriffsrechte und `chmod`

Durch Option `d` von `ls` wird eine Directory nicht aufgelistet, sondern wie ein normales File behandelt. Das heißt: Es wird Information über die Directory als Ganzes ausgegeben.

Das Kommando `chmod` ist dazu da, die Zugriffsrechte von Files zu ändern. Mit `+` werden Zugriffsrechte gewährt, mit `-` verwehrt. Die Buchstaben vor dem `+/-` spezifizieren den Benutzerkreis, dem Rechte eingeräumt bzw. genommen werden. Der Buchstabe `a` steht für alle, das heißt Besitzer, Gruppe und Rest der Welt. Die Buchstaben hinter `+/-` geben die Art der Zugriffsrechte genau an.

7 Generalthema heute: Shells

1. Aufgabe: welche Shell?

Geben Sie folgendes Kommando ein:

```
echo $SHELL
```

Jetzt kommt bei Ihnen entweder `bash` oder `tcsh` mit vollem Directory-Pfad heraus. Wenn bei Ihnen `bash` herauskommt, geben Sie jetzt das Kommando `tcsh` ein, und umgekehrt.

Geben Sie danach ein:

```
echo $SHELL
exit
echo $SHELL
```

Geben Sie nun ein:

`finger` *Benutzerkennung*

wobei Sie Ihre eigene Benutzerkennung eingeben. Auch in dieser Ausgabe sehen Sie, welche Shell Sie normalerweise benutzen.

Frage: Was macht das Kommando `echo` offenbar?

2. Aufgabe: `for` und `foreach`

Geben Sie folgende Zeilen ein, je nachdem, ob bei Ihnen eine `tcsh` oder eine `bash` läuft.

Für `tcsh`:

```
foreach x ( ~weihe/unix96/Aufgabe? )
foreach? echo x
foreach? echo $x
foreach? ls -l $x
foreach? end
```

Für `bash`:

```
for x in ~weihe/unix96/Aufgabe?
> do
> echo x
> echo $x
> ls -l $x
> done
```

Rufen Sie auch einmal die jeweils andere Shell auf und spielen Sie auch die `for/foreach`-Sequenz der anderen Shell durch. Es sollte dabei absolut dieselbe Ausgabe herauskommen.

Wiederholung von Aufgabe4: Was bedeutete noch das Fragezeichen in Filenamen?

Verständnisfragen:

- Wozu ist das `for/foreach`-Konstrukt offenbar gut?

- Was ist die Funktion des Buchstaben `x` ?
- Was macht `$x` ?
- Welche Funktion hat `end/done` ?
- Warum liefern die beiden Aufrufe von `echo` nicht dasselbe Ergebnis?

Suchen Sie in der Man Page „Ihrer“ Shell nach der Beschreibung von `for` bzw. `foreach`.

Hinweis für `bash`: `done` kommt seltener vor als `for`.

Wiederholung von `Aufgabe4`: Wie kann man im `less` nach Zeichenketten suchen?

3. Aufgabe: der Ursprung aller Shells

In den Man Pages beider Shells – `tcsh` und `bash` – steht im ersten Absatz der DESCRIPTION ein Verweis zu einer anderen, älteren Shell, aus der `tcsh` und `bash` in gewisser Weise weiterentwickelt wurden. In der Man Page zu dieser Shell (wieder in DESCRIPTION, 1. Absatz) steht der Name, unter dem die allererste und grundlegende UNIX-Shell vorkommt.

Blättern Sie weiter bis „SEE ALSO“ und nutzen Sie die Information dort um herauszufinden, unter welchem Stichwort Sie mit dem Kommando `man` Informationen über diese Shell erhalten können.

Schauen Sie in dieser letzten Man Page nun nach, was die Funktion des `$` ist.

4. Aufgabe: History

Geben Sie folgende Kommandos ein:

```
ls
ls -l
!ls
wc *
date
!!
!ls
!?c
!?at
csh
ls
exit
!ls
```

Verständnisfragen:

- Was bedeutet `!`*Zeichenkette* ?
- Was bedeutet im Unterschied dazu `!?`*Zeichenkette* ?
- Was bedeutet `!!` ?
- In einem Satz: Welche Sonderbedeutung hat das `!` für Ihre Shell?
- Warum liefert das letzte Kommando nicht `ls`, sondern `ls -l` ?

Geben Sie nun das Kommando `history 10` ein. Sie sehen nun eine Auflistung der letzten 10 von Ihnen in dieser Shell eingegebenen Kommandos mit fortlaufender Nummer. Vergleichen Sie diese Nummern mit der jeweiligen Nummer in Ihrem Prompt.

Wiederholung von `Aufgabe1`: Was ist der Prompt?

Suchen Sie die Nummer zu einem der Aufrufe von `wc *` heraus und geben Sie das Kommando `!`*Nummer* mit dieser Nummer ein. Was passiert? Geben Sie nun das Kommando `!-1` ein. Was passiert nun? Welche Bedeutung haben also positive bzw. negative Zahlen hinter einem `!` ?

5. Aufgabe: `!$`

Geben Sie folgende Kommandos ein:

```
wc ~weihe/unix96/Aufgabe1
ls -l !$
wc ~weihe/unix96/Aufgabe?
ls -l !$
wc ~weihe/unix96/Aufgabe1 ~weihe/unix96/Aufgabe2
ls -l !$
```

Wofür steht also `!$` ? Suchen Sie das erste Vorkommen von `$` in der Man Page zu `grep`. Versuchen Sie, eine Gemeinsamkeit herzustellen zwischen der Bedeutung von `$` in Regular Expressions und in der History.

6. Aufgabe: `!:`

Gehen Sie in die Directory `~weihe/unix96` und geben Sie folgende Kommandos ein. Achten Sie dabei auch jeweils auf die Zeile unmittelbar nach Ihrer Eingabe, in der Ihre Eingabe nach Verarbeitung durch die Shell erscheint.

Kommandos:

```
grep "^1\. Aufgabe" Aufgabe1 Aufgabe2 Aufgabe3 Aufgabe4 Aufgabe5
grep "^2\. Aufgabe" !:2*
grep "^3\. Aufgabe" !:3*
grep "^4\. Aufgabe" !:4*
```

Was tut also `!:`*Zahl* ?

7. Aufgabe

Gehen Sie in die Directory in Ihrem eigenen Heimbereich, in die Sie `sample.tex` aus `~weihe/others/misc` kopiert haben. Falls Sie nicht mehr wissen, wo das war, Wiederholung: Wie kann man das mit Kommando `find` herausfinden? (Funktioniert natürlich nicht, wenn Sie das File in der Zwischenzeit wieder gelöscht haben.)

Geben Sie nun folgende Kommandos ein:

```
latex sample.dvi
^dvi^tex
```

Was macht also das Kommando `^Zeichenkette1^Zeichenkette2` ?

8. Aufgabe: Quotes und Double Quotes

Geben Sie jetzt folgende Kommandos (jeweils für Ihre Shell) ein.

Für `tcsh`:

```
foreach x ( ~weihe/unix96/Aufgabe1 )
foreach? echo $x
foreach? echo "$x"
foreach? echo '$x'
foreach? end
```

Für `bash`:

```
for x in ~weihe/unix96/Aufgabe1
> do
> echo $x
> echo "$x"
> echo '$x'
> done
```

Sie haben damit einen wichtigen Unterschied zwischen „...“ und '...' gefunden. Welchen?

Versuchen Sie, eine allgemeine Vermutung zu formulieren, worin der grundlegende Unterschied zwischen „...“ und '...' bestehen könnte?

Probieren Sie nun:

```
ls -l
echo !!
ls -l
echo "!!"
ls -l
echo '!!'
```

Wird Ihre Vermutung durch dieses Beispiel eher untermauert oder widerlegt?

9. Aufgabe: Backquotes

Geben Sie ein:

```
echo `ls`
echo `ls -l ~weihe/unix96`
echo `blabla`
wc -l `ls`
ls | wc -l
```

Was ist also die Funktion von `...` ?

Was ist der Unterschied in den Ausgaben der letzten beiden Kommandos, oder konkreter gefragt: Von welchem Text werden jeweils die Zeilen gezählt?

10. Aufgabe: komplexeres Beispiel zu ‘...‘

Geben Sie ein:

```
echo $MANPATH
echo $MANPATH | tr ":" " "
ls -ld ‘echo $MANPATH | tr ":" " "‘
```

Wiederholung von `Aufgabe6`: Was macht `tr`?

Was macht Option `d` bei `ls`?

Geben Sie nun folgendes Kommando **exakt** ein:

```
find ‘echo $MANPATH | tr ":" " " "‘ -follow -name "*grep*" -print
```

Verständnisfragen:

- Was macht Option `follow` bei `find`? (Schauen Sie in der Man Page nach.)
- Ist Option `follow` hier notwendig? (Sehen Sie sich mit `ls -l` die relevanten Directories an.)
- Warum ist `tr` hier notwendig?
- Warum muß man mit ‘...‘ arbeiten und nicht mit `|`, um die Directories zu spezifizieren, in denen `find` suchen soll?

11. Aufgabe: Default-Setzungen für die Shell

Wiederholung von Aufgabe6: Was bedeutet das englische Wort „default“ im Zusammenhang mit Computern?

Die `tcsh`-Leute unter Ihnen sehen sich jetzt bitte das File `~/.cshrc` an, die `bash`-Leute das File `~/.bashrc`. Beide Files tun im Prinzip nichts anderes als zwei weitere Files einzulesen:

— zuerst ein File mit Default-Setzungen für die Shell, die für alle Nutzer dieser Shell gleich sind,

— das zweite mit selbstgemachten Setzungen.

Schauen Sie sich das erste eingelesene File (also das in `/usr/local/defaults`) an.

Verständnisfragen:

- Wie heißt der Newsserver (NNTP-Server), der per Default von News-Programmen wie `xrn` angesprochen wird?
- Wieviele Kommandos werden durch den History-Mechanismus maximal gespeichert?
- In welchen Zeilen wird die Liste aller Directories definiert, die die Shell durchsucht, um ein Kommando aufzurufen? Nehmen Sie dazu `~weihe/unix96/folien/96.11.07.ps` zu Hilfe und seien Sie notfalls großzügig mit Groß- und Kleinschreibung.

  Prüfen Sie das Ergebnis nach, indem Sie `echo $PATH` eingeben. Wie kommt der Punkt an den Anfang der Ausgabe?

- In welcher Zeile wird Ihr MANPATH definiert?
- Wo wird Ihr Shell-Prompt definiert? Durch welche Zeichenkette wird also die fortlaufende Kommandonummer im Prompt erzeugt?
- Wo wird `koala` definiert?

12. Aufgabe: Default-Setzungen interaktiv überschreiben

Machen Sie zwei neue xterms auf und geben Sie darin die folgenden Kommandos in dieser Reihenfolge ein.

`tcsh`-Nutzer:

```
# xterm Nr. 1:
which koala
alias koala 'echo "Nur ein Kuscheltier  ;-)"'
koala
which koala
echo $MANPATH
setenv MANPATH ~
!e
man ls

# xterm Nr. 2:
echo $MANPATH
which koala
man ls

# xterm Nr. 1:
source ~/.cshrc
!wh
!ec
```

`bash`-Nutzer:

```
# xterm Nr. 1:
which koala
alias koala='echo "Nur ein Kuscheltier  ;-)"'
koala
which koala
echo $MANPATH
MANPATH=~
!e
man ls

# xterm Nr. 2:
echo $MANPATH
which koala
man ls

# xterm Nr. 1:
source ~/.bashrc
!wh
!ec
```

Versuchen Sie mit Hilfe der relevanten Man Pages herauszufinden, was Sie soeben eigentlich gemacht haben.

13. Aufgabe: Shell selbst konfigurieren

WARNUNG: Wenn Sie eigenmächtig an Konfigurationsfiles für Shells, X Window System und für ähnlich grundlegende Systemkomponenten herumbasteln, loggen Sie sich nicht sofort aus, sondern loggen Sie sich zuerst an einem anderen Rechner ein, um zu sehen, ob immer noch alles vernünftig funktioniert. Falls nicht, können Sie vom ersten Rechner aus alle Basteleien notfalls wieder rückgängig machen.

Nehmen Sie sich den Inhalt von `/usr/local/defaults/.bashrc.source` bzw. `/usr/local/defaults/.cshrc.source` zum Vorbild, um in Ihrem File `~/.cshrc.$USER` bzw. `~/.bashrc.$USER` ein paar Aliasse zu setzen (`$USER` ist Ihre Benutzerkennung):

`a`	für `apropos`
`c`	für `cd`
`homefind`	Wie `locfind`, nur daß die Suche mit `find` in der eigenen Home-Directory beginnt, egal wo man gerade steht.

Setzen Sie in File `~/.cshrc.$USER` bzw. `~/.bashrc.$USER` außerdem die Maximalzahl der durch dem History-Mechanismus gespeicherten Kommandos auf 300. Außerdem setzen Sie Ihr Prompt-Zeichen um von `%` (`tcsh`) bzw. `$` (`bash`) auf „:-)“.

Rufen Sie nun ein neues xterm auf. Falls Ihnen die eben gemachten Setzungen nicht gefallen, dann löschen Sie sie einfach wieder; sie sollten nur als Beispiele dienen und haben keine weitere Bedeutung.

7 Antworten zum Generalthema Shells

1. Aufgabe: welche Shell?

> Zusammenfassende Wiederholung: Die Shell ist das Programm, das in einem Fenster `xterm` läuft. Dieses Programm ist dafür da, interaktiv Kommandos vom Benutzer entgegenzunehmen und auszuführen. Zuvor bearbeitet die Shell allerdings noch ihre Eingabe (z.B. Ersetzung von *).

Es gibt nicht **die** Shell, sondern es gibt verschiedene Shell-Programme. Sie unterscheiden sich in der Funktionalität, die sie jeweils anbieten, und in der Bequemlichkeit bei der Benutzung. Einige Details, in denen zwei Shells eigentlich das gleiche tun und gleich bequem sind, können sich aber dennoch in syntaktischen Details unterscheiden, siehe 2. Aufgabe unten.

Die Shells `tcsh` und `bash` unterscheiden sich nur gering (nicht jeder Befürworter einer der beiden Shells sieht das so!).

Das Kommando `echo` schreibt alle seine Argumente einfach wieder auf Standard Output.

Wiederholung von `Antworten5`: Was ist Standard Output?

2. Aufgabe: `for` und `foreach`

`for/foreach` ist ein (eher harmloses!) Beispiel dafür, daß die gleiche Funktionalität in verschiedenen Shells durch verschiedene Syntax ausgedrückt wird.

Das Konstrukt `for/foreach` ist dafür da, eine Abfolge von Kommandos auf mehrere verschiedene Argumente anzuwenden. Man nennt so etwas eine „Schleife“, und diese Abfolge von Kommandos bilden den „Schleifenrumpf“.

Der Schleifenrumpf wird Zeile für Zeile nach `for/foreach` eingegeben und mit `end` bzw. `done` abgeschlossen. Die erste Zeichenkette hinter `for/foreach` (hier: `x`) ist eine „Variable“ (Platzhalter), in der das jeweils aktuelle Argument gespeichert ist. Diese Argumente werden bei der `tcsh` in Klammern dahinter spezifiziert, bei der `bash` hinter `in`. Der Inhalt von `x` ist also im ersten Durchlauf der Schleife `~weihe/unix96/Aufgabe1`, im zweiten `~weihe/unix96/Aufgabe2` usw. Auf den Inhalt von `x` greift man zu, indem man ein `$` davorschreibt.

Die Kommandos `echo x` und `echo $x` liefern unterschiedliche Ergebnisse, weil `echo` in beiden Zeilen unterschiedliche Argumente zu sehen bekommt, nämlich beim ersten Mal die Zeichenkette `x`, beim zweiten Mal den Inhalt der Variablen `x`.

3. Aufgabe: der Ursprung aller Shells

In gewisser Weise sind `tcsh` und `bash` Weiterentwicklungen der C-Shell `csh`. Mit `man csh` findet man einen Verweis auf die Bourne Shell. Sieht man sich die Man Pages zu den Kommandos an, die in der Man Page zu `csh` unter „SEE ALSO“ verzeichnet sind, findet man schnell heraus, daß `sh` das Kommando ist, mit dem man die Bourne-Shell aufruft. Dies ist natürlich auch der Name, mit dem man die Man Page zur Bourne-Shell aufruft. In dieser Man Page findet man im Prinzip die Antwort, die oben zur 2. Aufgabe gegeben worden ist: daß man mit dem `$` auf den Inhalt einer Variablen zugreifen kann.

4. Aufgabe: History

Mit `!` *Zeichenkette* kann man sich das allerletzte Kommando (in dieser Shell!) zurückholen, das mit *Zeichenkette* begann.

Im Unterschied dazu wird bei `!?`*Zeichenkette* das allerletzte Kommando geliefert, in dem *Zeichenkette* irgendwo vorkam.

Mit `!!` erhält man das wirklich allerletzte Kommando noch einmal.

Der Sinn von `!` ist also, auf (relativ) bequeme Art Kommandos mehr als einmal auszuführen.

Es werden allerdings immer nur Kommandos zurückgeholt, die in dieselbe Shell eingegeben worden sind. Ruft man einen neuen Shell-Prozeß ins Leben (wie in der Aufgabe eine C-Shell durch das Kommando `csh`), dann hat dieser Shell-Prozeß seine eigene Liste von früheren Kommandos.

Alle Kommandos werden fortlaufend durchnumeriert. Die Nummer des Kommandos steht auch jeweils im Prompt. Mit einer positiven Zahl hinter dem Prompt ruft man das Kommando mit dieser Nummer zurück. Mit einer negativen Zahl hingegen wird in der History um so viele Kommandos zurückgegangen.

5. Aufgabe: `!$`

`!$` steht für das letzte Argument des letzten Befehls. Das heißt `!$` ist zum Beispiel sehr nützlich, wenn man mehrere Kommandos hintereinander auf ein sehr langes Argument (etwa mit absoluten Pfadnamen) anwendet.

Auch bei `grep` steht der `$` für das Ende der Zeile.

6. Aufgabe: `!:`

Mit `!:`*Zahl* erhält man die letzten Argumente des vorhergehenden Kommandos, und zwar spezifiziert durch *Zahl*. Genauer gesagt werden alle Argumente bis auf die *Zahl*`-1` ersten damit zurückgeholt.

7. Aufgabe

`^Zeichenkette1^Zeichenkette2` ist eine Möglichkeit, bequem einen Fehler im vorhergehenden Kommando zu korrigieren und das korrigierte Kommando aufzurufen: `Zeichenkette1` wird durch `Zeichenkette2` ersetzt.

Bemerkung: Die 5., 6. und 7. Aufgabe waren inspiriert durch die Kapitel 12.03, 12.04 und 12.05 aus dem Buch „UNIX Power Tools“. Ich selbst halte es da mit Adrian Nye (12.03): „My favorite is `!$`“.

8. Aufgabe: Quotes und Double Quotes

Wiederholung: `"..."` und `'...'` erfüllen im wesentlichen dieselben Zwecke, nämlich

— Sonderzeichen die Sonderbedeutung zu nehmen

— mehrere Zeichenketten zu einer zusammenzufassen (da ja auch dem Leerzeichen die Sonderbedeutung genommen wird).

Bei `"..."` wird allerdings nicht allen Zeichen die Sonderbedeutung genommen, nämlich `$` und `‘`. Das heißt `"..."` ist in gewisser Weise schwächer als `'...'`. Dem `!` wird weder in `"..."` noch in `'...'` die Sonderbedeutung genommen; das geht nur mit `\!`.

9. Aufgabe: Backquotes

Ein Ausdruck in ‘...‘ wird von der Shell als Kommando interpretiert. (Falls es kein Kommando ist, gibt es eine Fehlermeldung.) Dieses Kommando wird ausgeführt, und der Ausdruck in ‘...‘ wird von der Shell durch die Ausgabe des Kommandos ersetzt.

Bei `wc -l ‘ls‘` bekommt `wc` also die Namen aller Files in der Working Directory als Argumente und zählt gemäß Man Page daher die Zeilen in jedem dieser Files.

Bei `ls | wc -l` hingegen bekommt `wc` überhaupt keine Argumente, sondern `wc` bekommt eine Eingabe von Standard Input. (Wiederholung von `Aufgabe5`: Was ist Standard Input?) Laut Man Page bedeutet dies, daß einfach die Zeilen in der Eingabe von Standard Input gezählt werden, also die Zeilen in der Ausgabe von `ls` selbst, nicht die Zeilen in den Files, deren Namen durch `ls` ausgegeben worden sind.

Merkregel: Mit `|` kann man die Ausgabe eines Kommandos (auf Standard Output) zur Eingabe eines anderen Kommandos von Standard Input machen. Mit ‘...‘ kann man die Ausgabe eines Kommandos stattdessen zu Argumenten eines anderen Kommandos machen. Für die meisten Kommandos macht das einen erheblichen Unterschied!

10. Aufgabe: komplexeres Beispiel zu ‘...‘

Option `d` bei `ls` sorgt dafür, daß Argumente, die Namen von Directories sind, so wie andere Files auch behandelt werden. (Der Default für Directories ist ja, daß ihr Inhalt aufgelistet wird.)

Ohne Option `follow` wird nicht in Symbolic Links weitergesucht. Wie die Ausgabe des Kommandos

```
ls -ld ‘echo $MANPATH | tr ":" " "‘
```

gezeigt hat, ist das hier notwendig.

In MANPATH sind die einzelnen Directories durch Doppelpunkt getrennt, aber `find` erwartet alle Directories als einzelne Argumente, das heißt durch Leerzeichen getrennt. Deshalb müssen die Doppelpunkte durch Leerzeichen ersetzt werden, und genau das macht `tr`.

Die einzelnen Directories, in denen `find` suchen soll, werden laut Man Page als Argumente beim Aufruf übergeben. Die Merkregel zur 9. Aufgabe besagt oben, daß dies nur mit ‘...‘ geht, nicht mit `|`.

11. Aufgabe: Default-Setzungen für die Shell

Die Variable NNTPSERVER ist auf den Rechner mit Internet-Adresse `news.uni-konstanz.de` gesetzt - den Newsserver im Rechenzentrum der Uni Konstanz.

Die Variable `history`, in der die Maximalzahl von zu speichernden Kommandos gespeichert ist, ist für die `bash` auf dem Default 500 belassen worden und für die `tcsh` explizit auf 100 gesetzt worden.

Die Directories, in denen die Shell bei Eingabe eines Kommandos nach einem Programm dieses Namens sucht, sind in der Variable `path` bzw. `PATH` gespeichert. Der Punkt wird erst in den Zeilen nach der eigentlichen Setzung dieser Variable hinzuaddiert: durch `PATH=.:~/bin:$PATH` bzw. `set path=( . ~/bin $path )`.

12. Aufgabe: Default-Setzungen interaktiv überschreiben

Man kann einen Shell-Prozeß interaktiv konfigurieren, z.B. Aliasse und Shell-Variablen setzen. Andere Shell-Prozesse merken davon nichts.

Gibt man das Kommando `source`, dann interpretiert `xterm` das Argument als den Namen eines Konfigurationsfiles, liest die darin enthaltenen Setzungen ein und gehorcht diesen Setzungen.

13. Aufgabe: Shell selbst konfigurieren

Hat man den Inhalt des Konfigurationsfiles geändert bzw. den Inhalt von davon eingelesenen Files, dann sind die neuen Setzungen in jedem `xterm` gültig, das Sie von da an neu aufmachen, denn jedes neue `xterm` liest dieses Konfigurationsfile am Anfang.

UNIX/C–Praktikum

Donnerstag, 18.11.1996

Heute:

- Formale Sprachen
- Sprachumformungen
- Viele Beispiele aus bisherigen praktischen Übungen

Aspekte von Sprachen:

- **Syntax:** Wann ist ein sprachlicher Ausdruck korrekt?

  Z.B. Rechtschreibung, Interpunktion.

- **Semantik:** Welchen Sinn hat ein sprachlicher Ausdruck? Hat er einen Sinn?

- **Übersetzung:** Syntax ändern, Semantik gleich lassen.

- **Kontextfreiheit:** Semantik von Ausdruck erschließt sich, ohne daß weiterer Text bekannt ist.

  Beispiel für nicht kontextfrei:

  „Das ist die Höhe.“

Formale Sprachen bei Rechnern:

- Formale Sprachen werden von Programmen oder Hardware automatisiert verarbeitet („*interpretiert*“).
- Überschaubare Zahl von Syntaxregeln.
- Unzweideutige Semantik.
- Normierte schriftliche Formulierung von Syntax und Semantik.

  Extremfall: ANSI/ISO–Standard für Programmiersprachen (z.B. Ada, C, Cobol, Fortran).

- Syntax ist für Programm oder Hardware spezifisch.

  Aber: Ziel ist Angleichung bei Sprachen mit gleichartiger Semantik.

Typen von Sprachen bei Rechnern:

- Imperativ: Sprache ist Folge von Anweisungen, d.h. Sprache schreibt vor, was Programm bzw. Hardware tun soll.
  - Maschinensprache: von Hardware interpretierte Sprache.
  - Imperative Programmiersprache: wird durch *Compiler* in Maschinensprache übersetzt (z.B. Fortran, Pascal, C, C++, Eiffel).
- Deklarativ: Sprache beschreibt oder steuert, was Programm als Ergebnis liefern soll.
  - Textverarbeitung (z.B. LaTeX).
  - Optionen und Argumente für ein Kommando.
  - Verarbeitung von Kommandozeilen durch die Shell.
  - Deklarative Programmiersprache: wird durch *Interpreter* bearbeitet (z.B. Lisp, Prolog, ML).

Beispiel: deklarative Dokumentensprache LaTeX.

```
\documentclass{article}
\usepackage{german}
\begin{document}
  Dies ist ein \underline{sehr} kurzes,
  {\bf einfaches} Beispiel f"ur ein
  \LaTeX-Dokument.  "`In Wirklichkeit"'
  kann man damit {\it viiiel} mehr
  machen.  % z.B. diese Folien
\end{document}
```

Ergebnis:

Dies ist ein sehr kurzes, **einfaches** Beispiel für ein LaTeX–Dokument. „In Wirklichkeit“ kann man damit *viiiel* mehr machen.

Übersetzung in die deklarative Seitenbeschreibungssprache PostScript:

```
...

Ff(Dies)100 b(ist)f(ein)h(sehr)p
1245 4224 344 6 v 100 w(kurzes,)g
Fd(einfaches)h Ff(Beispiel)141
4433 y(f\177)-103 b(ur)57 b(ein)f(L)773
4415 y Fb(A)853 4433 y Ff(T)967 4476
y(E)1069 4433 y(X{Dokument.)
g(In)h(Wirklichk)-6 b(eit)54 b(k)-6
b(ann)141 4666 y(man)68 b(damit)f
Fa(viiiel)h Ff(mehr)f(machen.)p eop

...
```

Kann vom Drucker und vom Programm *ghostview* verarbeitet werden.

Beispiel: imperative Programmiersprache C.

```
#include<stdio.h>
#include<math.h>

int main ()
{
  /* sqrt = square root, d.h.
     berechnet Quadratwurzel */

  printf ( "Die Quadratwurzel von 2 " );
  printf ( "ist: \"%f\".", sqrt(2) );
}
```

Ausgabe:

```
Die Quadratwurzel von 2 ist "1.1414214".
```

Beachte:

- Anführungszeichen klammern auszugebende Zeichenkette ein.
- Durch vorangestelltes Backslash wird Anführungszeichen wörtlich genommen.

1. Merkmal von formalen Sprachen: Sonderzeichen

- Werden nicht wörtlich genommen, sondern steuern die Verarbeitung.
- Andere Sonderzeichen sind nötig, damit Sonderzeichen doch wörtlich genommen werden.

  Konvention in vielen Sprachen:

  – Backslash \ nimmt einem Sonderzeichen die Sonderbedeutung, so daß Zeichen wörtlich genommen wird.

  – Backslash vor anderem Zeichen wird entweder ignoriert (Shell) oder ist syntaktisch nicht korrekt (LaTeX).

2. Merkmal von formalen Sprachen: Klammerungen

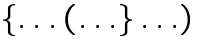
Sonderzeichen, die einzelne Teile vom Rest trennen und zu eigenen Einheiten zusammenfassen.

Typische Klammerungszeichen:

- Klammern: `(...)` `[...]` `{...}`
- Zitierungszeichen: `'...'` `` `...' `` `"..."`

Regel: Klammerungen sind ineinander enthalten oder kommen hintereinander, aber überlappen sich nicht.

Nicht möglich:



3. Merkmal von formalen Sprachen: Kommentare

- werden vom interpretierenden Programm ignoriert.
- unterstützen Verständnis menschlicher Leser.
- sind auch gut zum „Auskommentieren“ von Passagen.

Beispiel: Festlegung des Editors für elm im File ~/.elm/elmrc:

```
# what editor to use
# ("none" means simulate Berkeley Mail)
# editor = vi
editor = /usr/local/bin/emacsclient
```

Typische Syntax für Kommentare:

- Rest der Zeile nach Kommentarzeichen wird ausgeklammert, z.B.

#	viele UNIX–Programme
%	LaTeX und PostScript
//	C++
--	Eiffel
C	Fortran

- Kommentar wird durch Sonderzeichen eingeklammert, z.B.

/* ... */	C und C++
(* ... *)	Pascal

Heini Hacker sagt:

- Kommentare schreiben kostet unnötig Zeit.
- Kommentare sind uncool.
- Nur Dumm–User brauchen Kommentare zum Verständnis.

In der kommerziellen Praxis: Quelltexte

- sind oft riesengroß (bis zu mehreren Millionen Zeilen),
- werden von großen Teams erstellt;
- müssen öfters an neue Situationen angepaßt werden,
- und das oft von Leuten, die den Quelltext nicht selbst geschrieben haben.

⇒ Undokumentierte Quelltexte aller Art kommen teuer zu stehen.

Weiter mit formalen Sprachen

Formale Spezifikation der Syntax: Produktionsregeln

Beispiel für Produktionsregeln: SYNOPSIS bei Man Pages

```
SYNOPSIS
    cal [ [ month ] year ]
```

Produktionsregel:

cal [[*Ausdruck1*] *Ausdruck2*]

⟶ cal oder
cal *Ausdruck2* oder
cal *Ausdruck1* *Ausdruck2*

Weitere Produktionsregeln in SYNOPSIS:

```
SYNOPSIS
    bc [ -lws ] [  file ... ]
```

- bc [*Ausdruck1*] [*Ausdruck2*]

  ⟶ bc oder
  bc *Ausdruck1* oder
  bc *Ausdruck2* oder
  bc *Ausdruck1* *Ausdruck2*

- *File* ...

  ⟶ *File1* oder
  File1 *File2* oder
  File1 *File2* *File3* oder
  File1 *File2* *File3* *File4* oder

  usw.

- lws

  ⟶ *Option* ...

  wobei jede Ausprägung von *Option* entweder *l*, *w* oder *s* ist.

Spezifikation der Semantik:

Als Abbildung: Input ⟶ Output

Beispiel: cal

```
month   The month is a number between 1 and 12.
year    The year can be between 1 and 9999.
```

- Input: Zahl n zwischen 1 und 9999

  ⟶ Output: Kalender des ganzen Jahres n.

- Input: Zahl m zwischen 1 und 12
  Zahl n zwischen 1 und 9999

  ⟶ Output: Kalender für m–ten Monat des Jahres n.

Reicht noch nicht, denn:

```
An unusual calendar is printed for September 1752.
That is  the  month  11 days were skipped to make
up for lack of leap year adjustments.

The command cal 83 refers to the year 83, not 1983.

The year is always considered  to start in January
even though this is historically naive.
```

Beispiel: Modus bei *chmod* (vereinfacht):

```
chmod [ Options ] Mode File...
```

Modus besteht aus

1. betroffenem Personenkreis: u=user, g=group, o=others und a=all (auch mehrere).
2. einem Operator:

   \+ Zugriffsrecht einfügen.

   − Zugriffsrecht wegnehmen.

   = Zugriffsrecht exakt so setzen wie angegeben.

3. Zugriffsrechten: r=read, w=write und x=execute (auch mehrere).

Beispiel: Mathematische Ausdrücke für Programm bc (stark vereinfacht).

Rekursive Produktionsregeln:

Ausdruck ::=
  Zahl oder
  (*Ausdruck*) oder
  Ausdruck BinOp Ausdruck

Zahl ::=
  GanzeZahl oder
  GanzeZahl.GanzeZahl

GanzeZahl ::=
  Ziffer oder
  Ziffer#GanzeZahl

Ziffer ::= 0 1 2 3 4 5 6 7 8 9

BinOp ::= + − * /

Beachte: # ist hier Sonderzeichen und bedeutet „konkatenieren“ (zusammenfügen) ohne Blank.

Weiteres Beispiel: Regular Expressions

grep *RegExp Files*

Syntax für *RegExp*: Zeichenkette korrekt, wenn alle Klammerungen korrekt sind.

Semantik für *RegExp*: Jeder RegExp steht stellvertretend für eine Menge von Zeichenketten, die durch Platzhalter spezifiziert ist.

Beispiele:

- **bla.blu** ⟶ jede Zeichenkette mit sieben Zeichen, die mit „*bla*“ beginnt und mit „*blu*“ endet.
- **bla?blu** ⟶ „*blablu*“ und „*blaablu*“.
- **bla*blu** ⟶ „*blablu*“, „*blaablu*“, „*blaaablu*“, „*blaaaablu*“ usw.
- **bla[e–k]blu** ⟶ „*blaeblu*“, „*blafblu*“, „*blagblu*“ ... „*blakblu*“.
- **^blablu** ⟶ „*blablu*“ am Anfang einer Zeile.
- **blablu$** ⟶ „*blablu*“ am Ende einer Zeile.

Generelles zu RegExps:

Jeder Typ von Platzhalter definiert eine elementare Produktionsregel.

Zusätzlich rekursive Produktionsregel: Zwei RegExps konkateniert bilden wieder einen RegExp.

„*bla.bli*“ und „*[a-z]blu*“ sind RegExps

⟶ „*bla.bli[a-z]blu*“ ist wieder RegExp.

⇒ beliebig lange RegExps mit beliebig „verstreuten“ Platzhaltern möglich.

Nützliches Beispiel: „*bla.*blu*“ ⟶ jede Zeichenkette, die mit „*bla*“ beginnt und mit „*blu*“ endet, denn:

„.*“ ⟶ „.“, „..“, „...“, „....“, usw.

Nächstes Beispiel: Verarbeitung von Kommandozeilen durch die Shell (tcsh und bash)

- Blanks, Tabs u.ä. trennen Zeichenketten voneinander. Shell reicht jede Zeichenkette einzeln an Kommando weiter (auch den Kommandonamen selbst).
- Mit | werden zwei Kommandos so verknüpft, daß Ausgabe des ersten Eingabe des zweiten ist.
- Mit > wird Ausgabe eines Kommandos auf ein File „umgelenkt“.
- Der * ist in Filenamen Platzhalter für beliebige, beliebig lange Zeichenketten (⟷ RegExps).
- Das ? ist Platzhalter für genau ein Zeichen (⟷ RegExps).

Fortsetzung tcsh und bash:

- Das $ wertet eine Shell–Variable aus.
- Backslash nimmt Sonderzeichen die Sonderbedeutung.
- Mit "..." und '...' wird allen so eingeklammerten Sonderzeichen Sonderbedeutung genommen.

  Unterschied: Bei "..." behalten einige Sonderzeichen ihre Sonderbedeutung (z.B. $).
- Inhalt von `...` wird als Kommandoaufruf interpretiert und durch Ausgabe des Kommandoaufrufs ersetzt.
- Klammerung { ... , ... , ... } in Filenamen drückt Alternativen aus (durch Kommas getrennt).

Achtung: Aufpassen, wenn ein Zeichen sowohl für Shell als auch für aufgerufenes Kommando Sonderbedeutung hat.

Beispiel: Zeichenkette „*.plan*“ mit grep in Man Pages suchen.

1. Versuch: `grep .plan /usr/man/man1/*`

⇒ **Alle** Vorkommen von „*plan*“ aufgelistet, da Punkt in RegExps für beliebiges Zeichen steht

⇒ viel zu viel Output.

2. Versuch: `grep \.plan /usr/man/man1/*`

⇒ Derselbe Output, da Backslash von Shell interpretiert wird (d.h. ignoriert, da „.“ keine Sonderbedeutung für die Shell hat).

3. Versuch: `grep "\.plan" /usr/man/man1*`

⇒ Gewünschter Output, da Backslash nun von Shell an grep weitergereicht wird.

Das Sonderzeichen ! für tcsh und bash

```
weihe@hoyren ~ % date
  Sat Nov 11 17:16:55 MET 1995

weihe@hoyren ~ % whoami
  weihe

weihe@hoyren ~ % !!
  whoami
  weihe

weihe@hoyren ~ % !d
  date
  Sat Nov 11 17:17:07 MET 1995

weihe@hoyren ~ % !who
  whoami
  weihe

weihe@hoyren ~ % !-2
  date
  Sat Nov 11 17:17:23 MET 1995

weihe@hoyren ~ % history 7
   206  17:16   date
   207  17:16   whoami
   208  17:17   whoami
   209  17:17   date
   210  17:17   whoami
   211  17:17   date
   212  17:17   history 7
```

History bei tcsh und bash:

- Shell speichert Liste aller einmal aufgerufenen Kommandozeilen mit laufender Nummer und Uhrzeit bis zu einer Obergrenze.
- Obergrenze ist in Shell–Variable gespeichert ($history bei tcsh, $HISTSIZE bei bash).
- Wenn Obergrenze erreicht, wird bei jedem neuen Kommando ältestes gespeichertes Kommando aus Liste gelöscht.

Beispiele für Zugriff auf History:

!! Letztes Kommando noch einmal ausführen.

!-3 Drittletztes Kommando noch einmal ausführen.

!212 Kommando Nr. 212 in History–Liste ausführen.

!da Letztes Kommando, das mit „*da*“ begann, noch einmal ausführen.

!?da Letztes Kommando, wo „*da*“ auf der Kommandozeile vorkam, noch einmal ausführen.

Wie nutzen Gurus die History?

Jerry Peek, Tim O'Reilly, Mike Loukides:
Autoren von „*UNIX Power Tools*“

- Tim O'Reilly: „*My favorite is !$.*“

```
weihe@hoyren ~ % whatis cal
  cal   cal (1)  - display a calendar
weihe@hoyren ~ % whereis !$
  whereis cal
  cal: /usr/bin/cal /usr/man/man1/cal.1
```

- Mike Loukides: „*My favorite is ^^.*“

```
weihe@hoyren ~ % whatsi cal
  whatsi: Command not found.
weihe@hoyren ~ % ^si^is
  whatis cal
  cal   cal (1)  - display a calendar
```

- Jerry Peek: „*My favorite is !:n*.*“

```
weihe@hoyren ~ % grep hallo Aufgabe1 Aufgabe2 Aufgabe3
weihe@hoyren ~ % grep holla !:2*
  grep holla Aufgabe1 Aufgabe2 Aufgabe3
weihe@hoyren ~ grep holladrio !:3*
  grep holladrio Aufgabe2 Aufgabe3
```

8 Generalthema heute: Shell-Programmierung

1. Aufgabe: mein erstes Shell-Skript

Erstellen Sie ein File `~/ldg` mit folgendem Inhalt:

```
latex $1.tex
dvips $1.dvi -o $1.ps
ghostview $1.ps &
```

Falls `sample.tex` nicht bei Ihnen in `~` steht: Kopieren Sie das File aus `~weihe/others/misc` nach `~`.

Geben Sie nun folgende Kommandos ein:

```
ldg sample
ls -l ldg
chmod u+x !$
!l
!ld
```

Sie haben jetzt Ihr erstes eigenes Shell-Skript erstellt und ausgeführt.

Wiederholungen:

- Warum resultierte der erste Aufruf von ldg in einer Fehlermeldung?
- Was macht `chmod u+x` ?
- Was machen `!$`, `!l` und `!ld` hier?

Frage: Wofür steht offenbar `$1` in `ldg`?

2. Aufgabe: `sh`

Fügen Sie am Anfang von `ldg` eine Zeile ein:

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

Es muß aber wirklich die allererste Zeile sein und ganz am Anfang der Zeile beginnen.

Wiederholung von `Aufgabe7`: Was ist `sh`?

Führen Sie `ldg sample` nun noch einmal aus. Es sollte dasselbe wie in der 1. Aufgabe passieren. Schauen Sie in den Man Pages nach, was `#!` tut.

Hinweis: Am besten (oder am wenigsten schlecht!?) ist `#!` in der Man Page zur `csh` beschrieben.

Wiederholung von `Aufgabe7`: Wie steht die C-Shell `csh` in Beziehung zu `tcsh` und `bash`?

Erweitern Sie den Aufruf von sh in der ersten Zeile um die Option `-x` und führen Sie `ldg` wieder mit Argument sample aus. Vergleichen Sie das Ergebnis mit den Erklärungen von `-x` in der Man Page zu `sh`.

Bemerkung: Im weiteren lassen wir alle Shells-Skripte durch `sh` ausführen, damit es keine Probleme mit verschiedener Syntax der einzelnen Shells gibt. Die `sh` ist sinnvoll für Shell-Programmierung, da es sie auf jedem UNIX- System gibt, so daß Portierung von Shell-Skripten kein Problem ist (zumindest ist das Vorhandensein der „richtigen“ Shell kein Problem).

3. Aufgabe: `dirname` und `basename`

Fügen Sie zwischen die erste und zweite Zeile in `ldg` noch folgende Zeile ein:

```
cd ‘dirname $1‘
```

Außerdem ersetzen Sie in allen weiteren Zeilen `$1` jeweils durch

```
‘basename $1 .tex‘
```

Geben Sie nun folgende Kommandos ein:

```
cd ~weihe/unix96
ldg ~/sample
~/ldg ~/sample
~/ldg ~/sample.tex
pwd
```

Schauen Sie in den Man Pages zu `dirname` und `basename` nach, was diese beiden Kommandos machen.

Verständisfragen:

- Warum hat der Aufruf von `ldg` ohne `~/` vorneweg nicht geklappt?
- Was würde anders sein, wenn überall nur ‘`basename $1`‘ anstelle von ‘basename `$1 .tex`‘ stünde?
- Hat das `cd` in der zweiten Zeile von `ldg` einen Effekt außerhalb der Abarbeitung von `ldg`?

4. Aufgabe: eigene Variablen

Fügen Sie zwischen der zweiten und dritten Zeile von `ldg` noch eine weitere Zeile ein:

```
FILENAME=‘basename $1 .tex‘
```

Außerdem ersetzen Sie jedes Vorkommen von ‘`basename $1 .tex`‘ in den folgenden Zeilen einfach durch

```
$FILENAME
```

Führen Sie jetzt `ldg` haargenau so wie in der 3. Aufgabe aus. Es sollte sich kein Unterschied zu den Ausgaben in der 3. Ausgabe zeigen.

Aufgabe: Bevor Sie weiterarbeiten, überlegen Sie sich mindestens drei Gründe dafür, solche Ausdrücke wie ‘`basename $1 .tex`‘ durch selbstdefinierte Variable wie FILENAME zu ersetzen.

5. Aufgabe: `/bin`

Bewegen Sie `ldg` von `~` nach `~/bin` und geben Sie folgende Kommandos ein:

Für `tcsh`

```
cd ~weihe/unix96
ldg ~/sample
rehash
```

Für `bash`

```
cd ~weihe/unix96
ldg ~/sample
hash
```

Frage: Warum kann man einfach `ldg` anstelle von `~/bin/ldg` schreiben (im Gegensatz zur dritten Aufgabe) ? Sehen Sie sich dazu die Ausgabe von `"echo $PATH"` an.

6. Aufgabe: Schleifen

Erstellen Sie ein Shell-Skript `ls1`:

```
#!/usr/bin/sh
for x in 1 2 3 4 5 6 7 8 9
do
  ls -l /home/inf/weihe/unix96/Aufgabe$x
done
```

und ein Shell-Skript `ls2`:

```
#!/usr/bin/sh
for x in /home/inf/weihe/unix96/Aufgabe?
do
  ls -l $x
done
```

und lassen Sie sie beide ohne Argumente laufen. Beide Läufe sollten dieselben Ausgaben liefern. Vergleichen Sie mit der 2. Aufgabe in Aufgabe7.

Bemerkung: Die `sh` versteht die Tilde nicht; das ist ein Konstrukt, das erst später zu UNIX hinzugekommen ist. Statt dessen muß man immer den ganzen Pfadnamen `/home/inf/` angeben (oder einen relativen Pfadnamen).

7. Aufgabe: `for`-Schleife über Argumente

Erstellen Sie jetzt ein Skript `~/bin/ls3` mit folgendem Inhalt:

```
#!/usr/bin/sh
echo "-->" $#
for x in $@
do
  ls -l $x
done
```

Lassen Sie das Skript `ls3` zweimal laufen: einmal explizit mit den drei Argumenten

```
~weihe/unix96/Aufgabe1 ~weihe/unix96/Aufgabe2
~weihe/unix96/Aufgabe3
```

und einmal mit den (inzwischen mindestens) acht impliziten Argumenten

```
~weihe/unix96/Aufgabe?
```

Was ist also der Inhalt von `$#` und `$@` ?

8. Aufgabe: `if` und `test`

Ändern Sie `~/bin/ls3` so ab, daß der Inhalt hinterher wie folgt aussieht:

```
#!/usr/bin/sh
echo "-->" $#
for x in $@
do
  if test -d $x
  then
    echo It\'s a directory : $x
  elif test -h $x
  then
    echo It\'s a symbolic link : $x
  elif test -f $x
  then
    echo It\'s a regular file : $x
  else
    echo Neither a directory nor a symbolic link
    echo nor a regular file : $x
  fi
done
```

Rufen Sie `ls3` nun mit dem Argument `~weihe/unix96/*` auf. Suchen Sie in der Man Page zu `sh` die Passage, in der das Konstrukt `if-then-elif-then-else-fi` erklärt wird (am schnellsten suchen Sie nach `elif`). Sehen Sie sich ebenfalls die Man Page zu `test` an.

Verständnisfragen:

- Wiederholung: Warum ist der \ vor dem ' jeweils notwendig?
- Wann liefert der Test `test -d` bzw. `test -f` jeweils `true`? (Suchen Sie nicht nach `-h`, sondern warten Sie dazu die nächste Aufgabe ab!)

9. Aufgabe: Knieschuß mit `test` und `Man Page`

In der Man Page zu `test` finden Sie Option `-e`. Probieren Sie die einmal aus. Im Gegensatz zu dem, was in der Man Page steht, sollte es nicht funktionieren, sondern eine Fehlermeldung ergeben. Außerdem ist Option `-h` nicht beschrieben. Offenbar stimmt die Man Page zu `test` nicht mit der hier verwendeten Version von `test` überein.

Lassen Sie sich alle Man Pages zu `test` auflisten (Wiederholung: 1. Aufgabe in `Aufgabe5`). Finden Sie darin eine, die besser paßt?

10. Aufgabe: einzelne Argumente ansprechen

Schreiben Sie ein Shell-Skript `findgrep` folgenden Inhalts:

```
#!/usr/bin/sh
if test $# -ne 2
then
  echo Wrong number of arguments \(should be 2\)
elif test ! -d $2
```

```
then
  echo Argument \#2 is not a directory
else
  find $2 -type f -exec grep -l $1 {} \;
fi
```

Geben Sie sich wie üblich Ausführrechte auf `findgrep` und geben Sie dann folgende Kommandos ein:

```
findgrep
findgrep a
findgrep a b c
findgrep a b
findgrep convention ~/weihe/unix96/news-start
```

Fragen:

- Wiederholung der 7. Aufgabe: Was bedeutet `$#` ?
- Was passiert, wenn man die `\` vor `(` und `)` wegläßt?
  (Probieren Sie es aus.)
- Was passiert, wenn man das `\` vor `#` wegläßt? Eine Erklärung dieses Verhaltens steht (nicht ganz offensichtlich) in `~weihe/unix96/folien/96.11.18.ps`.
- Was bedeutet das `!` gemäß Man Page zu test?
- Was bedeuten offenbar `$1` und `$2` ? Was war daher die **genaue** Bedeutung von `$1` in der 1. Aufgabe oben?
- Wiederholung von `Aufgabe5`: Was bedeuten `-type` und `-exec`?
  Was sollen `{}` und `\;` ?
- Was macht die Option `l` bei `grep`?
- Welche Files werden also durch das letzte `findgrep` aufgelistet?

11. Aufgabe: `getopts` und `case`

Erstellen Sie ein Skript `~/bin/opt` mit folgendem Inhalt:

```
#!/usr/bin/sh
while getopts abcd x
do
  case $x in
    a)  echo Option a;;
    b)  echo Option b;;
    c|d)  echo Option c or d;;
  esac
done
```

Geben Sie nun folgende Kommandos ein:

```
opt -abcd
opt abcd
opt -cdba
opt -abab
opt -blabla
opt -ab hallo
opt -ab -ab hallo
opt -ab hallo -ab
```

Was ist also der Sinn von `getopts`?

Was ist die Idee hinter der Wortschöpfung `esac`? Was ist offenbar die Idee hinter der Wortschöpfung `fi` in der 8. Aufgabe?

12. Aufgabe: Signale abfangen

Schreiben Sie ein Skript `~/bin/schlaf`, das eine ganze Zahl als Argument bekommt. Inhalt von `schlaf`:

```
#!/usr/bin/sh

SIGHUP=1
SIGINT=2
SIGQUIT=3

while :
do
  sleep $1
  echo Kurz aufgewacht - schlafe sofort weiter.
  trap 'echo "Aaaaarghhh  :-("; exit 0' $SIGHUP $SIGINT $SIGQUIT
done
```

Lassen Sie `schlaf` jetzt viermal laufen und brechen Sie es jeweils nach ein paar Sekunden ab, und zwar jeweils mit einem anderen Signal: `HUP`, `INT`, `QUIT` und `KILL`. Was passiert jeweils?

Verständnisfragen:

- Was macht der `:` hinter `while`? Suchen Sie dazu in der Rubrik „Special commands“ in der Man Page zu `sh` nach der Erklärung.
- Wiederholung von Aufgabe5: Wie kommt man an die Man Page, in der die einzelnen Signale mit ihren Nummern aufgeführt sind?
- Was tut das Kommando `exit`?
- Was tut das Kommando `sleep`?
- Was tut das Kommando `trap`?

Schauen Sie in der Man Page, in der die Signale mit ihren Nummern aufgeführt sind, nach, welche Nummer Signal KILL hat, und fügen Sie SIGKILL völlig analog zu den anderen drei Signalen ein. Lassen Sie `schlaf` wieder laufen und schicken Sie dem Prozeß das Signal KILL. Was passiert und warum?

13. Aufgabe: Exit Status und Skript in Skript aufrufen

Schreiben Sie ein Skript `~/bin/testfile`, das genau ein Argument bekommt und in einem großen `if-...-fi` Konstrukt und mit Hilfe des Kommandos `test` mehrere Fälle abtestet. Das Skript soll im Gegensatz zu bisher das Ergebnis nicht mit `echo` mitteilen, sondern anstelle eines `echo`-Kommandos soll ein `exit`-Kommando stehen. Das Kommando `exit` erwartet ein einzelnes Argument, das eine ganze Zahl sein soll. In den einzelnen abzutestenden Fällen soll das Argument von `exit` jeweils verschieden sein.

Konkret soll `testfile` folgendes abtesten:

- Falsche Anzahl von Argumenten $\Longrightarrow$ exit 0
- Reguläres File $\Longrightarrow$ exit 1
- Symbolic link $\Longrightarrow$ exit 2
- Directory $\Longrightarrow$ exit 3

Schreiben Sie nun ein Skript `testmanyfiles`, das beliebig viele Argumente bekommen kann und auf jedes dieser Argumente jeweils `testfile` anwendet, und zwar in der Form

```
for x in $@
do
  testfile $x
  case $? in
  ...
  esac
done
```

Vergleiche die 11. Aufgabe zur Syntax von `case-esac`.

Die einzelnen Zeilen in `case-esac` sollen jeweils die Bedeutung ausgeben, also z.B. eine Meldung „It's a regular file“ bei Exit Status 1.

Testen Sie `testmanyfiles` mit Argumentliste `~weihe/unix96/*`.

Schlagen Sie in der Man Page zu sh nach, was `$?` bedeutet. (Suchen Sie nach Zeichenkette `automatically`.)

8 Antworten zum Generalthema Shell-Programmierung

1. Aufgabe: mein erstes Shell-Skript

`$1` steht hier für das einzige Argument, also `sample`.

2. Aufgabe: `sh`

Mit `#!` am Anfang eines Shell-Skripts wird festgelegt, welche Shell dieses Skript ausführen soll.

Die Option `-x` protokolliert alle Aktionen des Skripts mit.

3. Aufgabe: `dirname` und `basename`

Mit `dirname` und `basename` wird der Name eines Files in den eigentlichen (Directory-)pfad und den eigentlichen Filenamen zerlegt. Gibt man `basename` noch ein zweites Argument, dann prüft `basename`, ob das zweite Argument die Endung des ersten Arguments ist. In diesem Fall wird auch diese Endung entfernt. Mit `basename $1` anstelle von `basename $1 .tex` würde also z.B. `dvips` nicht auf `sample.dvi`, sondern auf `sample.tex.dvi` angewendet, und dieses File gibt es nicht.

Wenn man nicht in `~` steht, muß man `~/ldg` anstelle von `ldg` spezifizieren, weil `~` nicht im Suchpfad enthalten ist und die Shell daher nicht in `~` sucht. Steht man in `~`, dann reicht `ldg`, weil die Working Directory im Suchpfad enthalten ist.

Offenbar wirkt das `cd` in `ldg` nur lokal im Shell-Skript selbst. (Alles andere wäre auch nicht sinnvoll.)

4. Aufgabe: eigene Variablen

Gute Gründe für Variable:

- Man kann lange Ausdrücke abkürzen.
- Mit einem gut gewählten Namen für die Variable, der die Bedeutung gut widerspiegelt, wird das Skript verständlicher, weil man nicht erst den Ausdruck verstehen muß.
- Wenn zufällig derselbe Ausdruck zwei verschiedene logische Bedeutungen hat, kann man diese Bedeutungen voneinander durch zwei verschiedene Variablennamen trennen. Das passiert z.B. häufig, wenn der Ausdruck einfach eine Zahlenkonstante ist.
- Wenn man einen Ausdruck ändern muß, kann man das einfach in der Zeile tun, in der die Variable definiert wird. Man muß nicht das ganze Skript dafür durchgehen und an vielen Stellen abändern, was bei größeren, ernsthaften Skripten aufwendig und fehleranfällig wäre.

5. Aufgabe: `/bin`

Die Directory `~/bin` ist im Suchpfad enthalten, dessen Inhalt man mit `echo $PATH` bekommt. Daher wird `~/bin/ldg` beim Aufruf von `ldg` ohne Pfadangabe gefunden.

7. Aufgabe: `for`-Schleife über Argumente

In `$#` steht die Anzahl der Argumente, `$@` ist eine Liste aller Argumente, die man z.B. mit `for` durchlaufen kann.

Bemerkung: Besonders im Englischen (aber nicht nur dort) ist `#` häufig die Abkürzung für Nummer oder Anzahl, z.B. `#3 = Nummer 3`.

8. Aufgabe: `if` und `test`

Mit `if-...-fi` kann man in einem Shell-Skript Fallunterscheidungen treffen. Vor `then`, `elif` (= `else if`), `else` und `fi` muß entweder ein Zeilenumbruch oder ein Semikolon stehen. Es gibt mehrere konkrete Möglichkeiten, das Konstrukt zu verwenden, zum Beispiel:

- Kommandos nur ausführen, wenn eine bestimmte Bedingung erfüllt ist:

  `if` *Bedingung*; `then` *Kommandos*; `fi`

- Unterscheidung von genau zwei Fällen:

  `if` *Fall 1*; `then` *Kommandos für Fall 1*;
  `else` *Kommandos für Fall 2*; `fi`

- Unterscheidung von mehr als zwei Fällen:

  `if` *Fall 1*; `then` *Kommandos für Fall 1*;
  `elif` *Fall 2*; `then` *Kommandos für Fall 2*;
  `elif` *Fall 3*; `then` *Kommandos für Fall 3*;
  `...`
  `else` *Kommandos für letzten Fall*;

Der \ vor dem ' ist notwendig, um dem ' die Sonderbedeutung als Zeichen für wörtliche Zitierung zu nehmen.

Die Tests `test -d` und `test -f` erwarten jeweils einen Filenamen als Argument. Der erste Test liefert „wahr“, wenn das File eine Directory ist, der zweite Test liefert „wahr“, wenn es ein „normales“, „reguläres“ File ist.

9. Aufgabe: Knieschuß mit `test` und Man Page

Wiederholung von Aufgabe5: Durch `man -l test` werden alle Man Pages zu `test` aufgelistet. Die Man Page in `/usr/local/man` wird als erste aufgelistet und daher automatisch genommen, sofern man nicht mit `-M` eine andere Directory spezifiziert. Mit `man -M /usr/man test` findet man eine Man Page, die besser paßt.

Das Problem rührt daher, daß test ein Built-In Kommando der Shell ist (vergleiche `~weihe/unix96/folien/96.11.07.ps`). Die Inhalte von PATH und MANPATH sind aufeinander abgestimmt, aber `test` wird eben nicht durch PATH gefunden, sondern gehört zur `sh` dazu. Die `sh` ist eine sehr alte Shell, und die Man Pages in `/usr/local/man` sind jünger. Die Man Page von `test`, auf die sich `man sh` bezieht, sind die „alten“ in `/usr/man`.

Merkregel: Wenn Sie merken, daß eine Man Page nicht stimmt, kann es einfach die „falsche“ Man Page sein.

10. Aufgabe: einzelne Argumente ansprechen

Die \ vor (und) sind wieder notwendig, um den Klammern ihre Sonderbedeutung zu nehmen.

Das # hat in Shell-Skripten eine Sonderbedeutung, nämlich einen Kommentar einzuleiten, der bis zum Ende der Zeile geht.

Das `!` als Argument für `test` bedeutet Negierung: `test ! -d` ist genau dann wahr, wenn `test -d` falsch ist. Die Konvention, `!` als Zeichen für logische Negierung zu verwenden, findet sich auch in anderen Sprachen, z.B. C und C++.

Für die Ziffern n=0..9 bedeutet `$`*n* das n-te Argument. Das 0-te „Argument“ ist dabei der Kommandoname selbst. In der 1. Aufgabe bedeutete `$1` also genau genommen nicht das einzige Argument, sondern das erste Argument.

Option `l` sorgt bei `grep` dafür daß keine Zeilen ausgegeben werden, in denen die Zeichenkette gefunden wurde, sondern nur die Namen der Files, in denen die Zeichenkette überhaupt irgendwo gefunden wurde.

Durch `findgrep convention ~/weihe/unix96/news-start` werden also alle Files in `~/weihe/unix96/news-start` aufgelistet, in denen die Zeichenkette `convention` vorkommt.

11. Aufgabe: `getopts` und `case`

Mit `getopts` kann man sehr bequem Optionen verarbeiten, die in der UNIX-üblichen Form angegeben werden.

`esac` ist `case` umgedreht, so wie `fi` gleich `if` umgekehrt ist. Diese Idee, ein Konstrukt durch die Umkehrung des einleitenden Wortes zu beenden, taucht in vielen Computersprachen wieder auf.

12. Aufgabe: Signale abfangen

Der `:` ist ein „leeres“ Kommando, das nichts tut und immer Wert 0 (=„wahr“) zurückliefert. Die `while`-Schleife in `~/bin/schlaf` ist also eine sogenannte „Endlosschleife“, das heißt die Bedingung zum Abbruch wird nie erreicht.

Mit `man -s 5 signal` bekommt man eine Auflistung aller Signale, die man Prozessen schicken kann, nebst ihrer Zahlenwerte.

Mit `exit` kann man die Abarbeitung eines Shell-Skriptes sofort beenden. Das Kommando `exit` erwartet eine ganze Zahl als Argument und liefert diese Zahl als Exit Status zurück. Unter UNIX ist es üblich, daß ein Programm einen Exit Status zurückliefert (vergleiche z.B. DIAGNOSTICS in der Man Page zu `grep`).

Mit `sleep` *Zahl* kann man einen Prozeß *Zahl* Sekunden lang schlafenlegen.

Mit `trap` kann man Signale abfangen, das heißt die Default-Reaktion durch eine selbstdefinierte Reaktion überschreiben (vergleiche `~weihe/unix96/folien/96.11.07.ps`).

Das Signal KILL läßt sich nicht abfangen (siehe wieder `~weihe/unix96/folien/96.11.07.ps`), auch nicht durch `trap`.

13. Aufgabe: Exit Status und Skript in Skript aufrufen

`$?` liefert den Exit Status des allerletzten ausgeführten Kommandos, also von `testfile $x`.

UNIX/C++–Praktikum

Donnerstag, 28.11.1996

Heute: Shell–Programmierung

Einführendes Beispiel:

```
weihe@hoyren ~ % latex sample.tex
   ...
weihe@hoyren ~ % dvips sample.dvi
   ...
weihe@hoyren ~ % ghostview sample.ps &
```

Idee: Kommandos in File schreiben, das von Shell interpretiert wird (*Shell–Skript*).

Inhalt des Files ldg:

```
latex $1.tex
dvips $1.dvi
ghostview $1.ps &
```

Fertigmachen zur eigenen Nutzung:

```
weihe@hoyren ~ % chmod u+x ldg
```

Aufruf:

```
weihe@hoyren ~ % ldg sample
```

Kommandoaufruf durch Shell:

1. Shell bearbeitet Kommandozeile, z.B.:
   - Aliasse ersetzen.
   - Variablennamen nach $ durch Inhalt ersetzen.
2. Falls erstes Wort ein Shell Built–In, läßt Shell entsprechendes Unterprogramm ablaufen.
3. Sonst: Shell sucht in Directories in $PATH nach File dieses Namens.
4. Falls gefunden: Shell testet, ob File Maschinenprogramm oder Shell–Skript ist.
5. Falls Maschinenprogramm, wird Prozeß mit diesem Programm gestartet.

   ↖ bisher der Normalfall
6. Sonst: Shell interpretiert Skript.

Verfeinerung:

- LaTeX legt sample.dvi in „.“ ab,
- sample.dvi und sample.ps sollen jetzt aber unter allen Umständen in derselben Directory stehen wie sample.tex.

Lösung: *dirname* und *basename*

```
weihe@hoyren ~ % dirname /home/inf/weihe/sample.tex
  /home/inf/weihe
weihe@hoyren ~ % dirname `dirname /home/inf/weihe/sample.tex`
  /home/inf
weihe@hoyren ~ % basename /home/inf/weihe/sample.tex
  sample.tex
weihe@hoyren ~ % basename /home/inf/weihe/sample.tex .tex
  sample
weihe@hoyren ~ % basename sample.tex .tex
  sample
```

Wiederholung: Kommandoaufruf in `...` wird ersetzt durch Ausgabe des Kommandos.

Neue Variante von ldg:

```
#  1. Schritt: in die Directory von file.tex gehen
cd ‘dirname $1‘

#  2. Schritt: file.tex -> file.dvi
latex ‘basename $1 .tex‘.tex

#  3. Schritt: file.dvi -> file.ps
dvips ‘basename $1 .tex‘.dvi\
   -o ‘basename $1 .tex‘.ps

#  4. Schritt: Ergebnis anzeigen lassen
ghostview ‘basename $1 .tex‘.tex &
```

Erläuterungen:

- $1 steht für das erste (und hier einzige) Kommandozeilenargument (sample oder sample.tex).
- # leitet Kommentar ein, der bis Zeilenende geht.
- \ vor Zeilenende (auch kein Blank danach!) heißt: Zeilenumbruchszeichen (ASCII 10) wird Sonderbeutung genommem

  ⇒ Diese und nächste Zeile bilden zusammen **eine** Kommandozeile.

Verbesserung durch selbstdefinierte Variable:

```
#! /bin/sh
#  ^^^^^^^ Auswertende Shell explizit festgesetzt.

cd ‘dirname $1‘

file=‘basename $1 .tex‘
latex $file
dvips $file.dvi -o $file.ps
ghostview $file.ps &
```

Erläuterungen zu #! /bin/sh:

- Die Shell sh (*Bourne–Shell*) interpretiert Skript, nicht die aktuell laufende Shell.

  ⇒ Shell–Skript kann unter verschiedenen Shells aufgerufen werden ohne Probleme mit unterschiedlicher Syntax.
- Konkretes Beispiel oben: Syntax der Definition der Variable *file*.
- Bourne–Shell gibt es auf jedem UNIX–System.

  ⇒ Shell–Skript läuft überall.

Shell–Skript mit beliebig vielen Kommandozeilenargumenten

Beispiel: Shell–Skript wcsum

Inhalt: `cat $@ | wc`

Erläuterung:

- $@ steht für **alle** Kommandozeilenargumente.
- Wdh.: cat gibt alle Eingabefiles wieder aus — konkateniert zu einem File.

Also:

```
weihe@hoyren ~ % wcsum ~weihek/Aufgabe?
  2453   11070   77343
```

Zum Vergleich:

```
weihe@hoyren ~ % wc ~weihek/Aufgabe?
   394    1753   12027 /home/student/weihek/Aufgabe1
   443    2210   15710 /home/student/weihek/Aufgabe2
   365    1717   12077 /home/student/weihek/Aufgabe3
   449    2142   14379 /home/student/weihek/Aufgabe4
   388    1528   10962 /home/student/weihek/Aufgabe5
   414    1720   12188 /home/student/weihek/Aufgabe6
  2453   11070   77343 total
```

Kommandozeilenargumente einzeln behandeln

Beispiel: Namensextension bei mehreren Files abschneiden; basename geht nicht, denn:

```
SYNOPSIS
   basename file [suffix]
```

Also Shell–Skript schreiben:

```
NAME
  exsuf - extrahiere suffix von filenamen
SYNOPSIS
  exsuf suffix file...
```

Inhalt:

```
#! /bin/sh
suffix=$1
shift
for x in $@
do
  basename  $x $suffix
done
```

Erläuterung: shift macht $2 zu $1, $3 zu $2, $4 zu $3 etc.; altes $1 wird aus $@ hinausgeworfen.

Generell nützlich: das Kommando test

Beispiel: Erst abtesten, ob $1 ein „normales“ File und lesbar ist, bevor $1 durchge„grep“pt wird.

```
#! /bin/sh

if test -f $1           # $1 existiert und ist
                        # "normales" File?
then
  if test -r $1         # File $1 ist lesbar?
  then
     grep blabla "$1"   # >>> EIGENTLICHE AKTION <<<
  else
     echo  "Keine Leseberechtigung fuer File $1."
  fi                    # fi = if spiegelbildlich
else
  echo  "File $1 existiert nicht oder"
  echo  "ist kein normales File."
fi
```

Man Page von test (Auszüge)

Achtung: „test“ ist ein Built–in Kommando von sh.

Beschreibung in „man sh“: Verweis auf „test(1)“.

Gemeint: 1. Sektion in /usr/man.

Also: man -M/usr/man test

```
NAME
     test - check file types and compare values

SYNOPSIS
   test [expr]
   ...

DESCRIPTION
   ...

   -r filename   True if filename exists and is readable.
   -w filename   True if filename exists and is writable.
   -x filename   True if filename exists and is executable.
   -f filename   True if filename exists and is a regular
                 file. ...
   -d filename   True if filename exists and is a directory.
   -h filename   True if filename exists and is a  symbolic
                 link.

   ...
```

Bequeme Variante im Gebrauch von test:

```
if [ -r $1 ]
then
  ...
else
  echo  Keine Leseberechtigung fuer File $1.
```

Erläuterung:

Es gibt Programm namens „[“ mit folgenden Eigenschaften:

- [hat die gleiche Synopsis wie test, nur daß] als zusätzliches Kommandozeilenargument verlangt wird.
- [ruft test auf und reicht alle Kommandozeilenargumente außer dem abschließenden] an test weiter.

```
weihe@hoyren ~ % which [
  /local/bin/[
```

Weitere Möglichkeiten von „test“

- Filevergleich, z.B. [File1 –ot File2] $\rightarrow$ true $\Longleftrightarrow$ File1 hat ältere Modification Time als File2 (–ot=older than).
- Zeichenketten vergleichen, z.B.

  –z S $\rightarrow$ true $\Longleftrightarrow$ S hat Länge 0 (–z=zero).

  S1=S2 $\rightarrow$ true $\Longleftrightarrow$ S1 und S2 sind gleich.
- Logische Kombination von Ausdrücken:

  –a=and: [A1 –a A2] $\rightarrow$ true $\Longleftrightarrow$ [A1] $\rightarrow$ true und [A2] $\rightarrow$ true.

  –o=or: [A1 –o A2] $\rightarrow$ true $\Longleftrightarrow$ [A1] $\rightarrow$ true und/oder [A2] $\rightarrow$ true.

  !=not: [!A] $\rightarrow$ true $\Longleftrightarrow$ [A] $\rightarrow$ false.

Noch eine Möglichkeit von „test“: ganze Zahlen vergleichen

```
#! /bin/sh
if [ $# -eq 0 ]      #  $# ist Anzahl der Kom-
                     #  mandozeilenargumente
then
  echo  "Keine Argumente angegeben!"
else
  if [ $# -eq 1 ]
  then
    echo  "Nur ein Argument angegeben!"
  else
    zeichenkette=$1
    shift
    grep "$zeichenkette" $@
  fi
fi
```

In man test:

```
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 inte-
  gers,  or the special  expression  -l string,
  which evaluates to the length of string.
```

Wie funktioniert „if test“?

In man test:

```
DESCRIPTION
  test returns a status of 0 (true)  or 1 (false) depen-
  ding on the  evaluation of the conditional expression.
```

In man sh (vereinfacht):

```
if command
then
  list
else
  list
fi


The command following if is executed and,  if it returns
a zero exit status, the list following then is executed.
Otherwise,  the list following  else is executed.
```

Exit Status von Kommandos

Beispiel: grep

```
DIAGNOSTICS
  Normally, exit status is 0 if matches were found, and
  1 if no matches were found.  ... Exit status is 2  if
  there were syntax errors in the pattern, inaccessible
  input files, or other system errors.
```

Exit Status ausnutzen: Shell–Skript grepfail gibt alle Filenamen aus, in denen „zeichenkette“ nicht vorkommt.

```
SYNOPSIS
  grepfail zeichenkette file...
```

Inhalt von grepfail:

```
#! /bin/sh
zeichenkette=$1
shift
for x in $@
do
  grep "$zeichenkette" "$x"
  exitstatus=$?
  if [ $exitstatus -eq 1 ]
  then
    echo $x
  fi
done
```

Noch einmal grepfail:

```
#! /bin/sh
zeichenkette=$1
shift
for x in $@
do
  grep "$zeichenkette" "$x"
  exitstatus=$?
  if [ $exitstatus -eq 1 ]
  then
    echo $x
  fi
done
```

Erläuterungen:

- Programme liefern ganze Zahl als Exit Status zurück (wie test).
- Konvention: 0 heißt „alles ging glatt“; verschiedene Zahlen $\neq 0$ geben verschiedene Fehlerfälle an.
- Exit Status des allerletzten Kommandos zugreifbar mit $?.
- **Achtung:** Inhalt von $? zur Sicherheit in Variable zwischenspeichern, weil $? mglw. vor Auswertung überschrieben wird durch Auswertung anderer Teilausdrücke.

Problem: wenn nur Exit Status wichtig ist und Output unterdrückt werden soll.

1. Lösung: „Quiet“–Option

```
DESCRIPTION
     ...
     -q    Quiet; suppress normal output.
```

Problem: Manche Kommandos bieten keine „Quiet“–Option an.

Beispiel: cp

Generelle Lösung: sämtliche Ausgaben umlenken mit „>“

```
cp file1 file2 1> /dev/null 2>&1
```

Zu `cp file1 file2 1> /dev/null 2>&1`

- Programme sind mit mindestens drei „*Filedeskriptoren*“ verbunden.
  - 0: stdin (=standard input=Standardeingabe) für Eingaben von Tastatur.
  - 1: stdout (=standard output=Standardausgabe) für Ausgaben auf Bildschirm.
  - 2: stderr (=standard error=Standardfehlerausgabe) speziell für Fehlermeldungen auf Bildschirm.
- „>“ ohne Nummer: stdout wird umgelenkt.
- „*Nummer*>“ heißt: Filedeskriptor „Nummer“ wird umgelenkt auf File.
- „*Nummer1*>&*Nummer2*“ heißt: „Nummer1“ wird auf selbes File umgelenkt wie „Nummer2“.
- /dev/null ist File mit Spezialbedeutung:
  - Von /dev/null lesen → sofort EOF–Zeichen (EOF=end of file).
  - Auf /dev/null schreiben → Ausgabe verschwindet „im Mülleimer“.

Shell–Skript mit Exit Status

Beispiel: Shell–Skript askrm

```
NAME
  askrm  --  File loeschen, aber erst fragen
SYNOPSIS
  askrm file
DIAGNOSTICS
  Exit Status ist
    0,  wenn Benutzer durch "j" (+optionale weitere
        Zeichen, z.B. "a") Loeschen erlaubt.
    1,  wenn Benutzer durch "n" (+optionale weitere
        Zeichen, z.B. "ein") Loeschen verbietet.
    2,  wenn Benutzereingabe fehlerhaft.
    3,  wenn File nicht existiert.
```

Inhalt von askrm:

```
#! /bin/sh

if [ -f $1 ]        # File $1 existiert?
then
  echo  "File $1 existiert!  Loeschen (j,n) ?"
  read antwort
  case $antwort in
  j*) rm $1         # Antwort beginnt mit "j".
      exit 0
      ;;
  n*) exit 1        # Antwort beginnt mit "n".
      ;;
  *)  exit 2        # Weder mit "j" noch mit "n".
      ;;
  esac              # esac=case spiegelbildlich
else
  exit 3
fi
```

Noch einmal Inhalt von askrm:

```
#! /bin/sh

if [ -f $1 ]       # File $1 existiert?
then
  echo  "File $1 existiert!  Loeschen (j,n) ?"
  read antwort
  case $antwort in
  "j*") rm $1      # Antwort beginnt mit "j".
        exit 0
        ;;
  "n*") exit 1     # Antwort beginnt mit "n".
        ;;
   "*") exit 2     # Weder mit "j" noch mit "n".
        ;;
  esac             # esac=case spiegelbildlich
else
  exit 3
fi
```

Erläuterungen:

- Kommando „read“ liest Zeile von stdin und schreibt sie in Variable antwort.
- „case antwort ... esac“ testet Alternativen ab; erste Alternative, auf die $antwort paßt, wird genommen.
- Kommando „exit“ setzt Exit Status und bricht Abarbeitung von Shell–Skript ab.

Shell–Skript in anderem Shell–Skript aufrufen

Beispiel (Fragment eines Shell–Skripts):

```
for x in $@
do
  askrm $x
  exitstatus=$?
  case $exitstatus in
  0) ...
     ;;
  1) ...
     ;;
  2) ...
     ;;
  3) ...
     ;;
  #  Alternative "*") unnoetig, da askrm nur
  #  0, 1, 2 oder 3 zurueckliefert.
  esac
done
```

Vorteile:

- Man muß einzelne, immer wieder benötigte Bestandteile nicht jedesmal neu schreiben.
- Shell–Skript wird lesbarer (falls aufgerufene Shell–Skripts sinnvolle Namen haben).

Signal–Behandlung:

Beispiel: Signale SIGHUP (Hangup) und SIGINT (Interrupt) abfangen, um temporäres Hilfsfile vor Terminierung wegzuräumen.

Fragment eines Shell–Skripts:

```
SIGHUP=1
SIGINT=2
tempfile=...
trap 'rm $tempfile; exit 1' $SIGHUP $SIGINT
```

Erläuterungen:

- Signale sind intern nur als Nummern bekannt. Zur besseren Lesbarkeit wird statt dessen Variable mit selbsterklärendem Namen verwendet.
- Signalnummern: „man -s5 signal“.
- Typischer Fall bei trap: Erstes Argument ist Kommandoliste (in '...' !), alle weiteren sind Signalnummern.
- Wenn Prozeß, der Shell–Skript abarbeitet, eines dieser Signale bekommt, wird erstes Argument von trap ausgeführt.

Optionen behandeln

Einfaches Test–Beispiel: Shell–Skript testopts

```
SYNOPSIS
  testopts [ -abc ] file...
```

Inhalt von testopts:

```
#! /bin/sh

while getopts abc opt
do
  case $opt in
    a) echo  Option -a
       ;;
    b) echo  Option -b
       ;;
    c) echo  Option -c
  esac
done
```

Ergebnis:

```
weihe@hoyren ~ % testopts -abac -ada ~weihek/Aufgabe?
  Option -a
  Option -b
  Option -a
  Option -c
  Option -a
  testopts: illegal option -- d
  Option -a
```

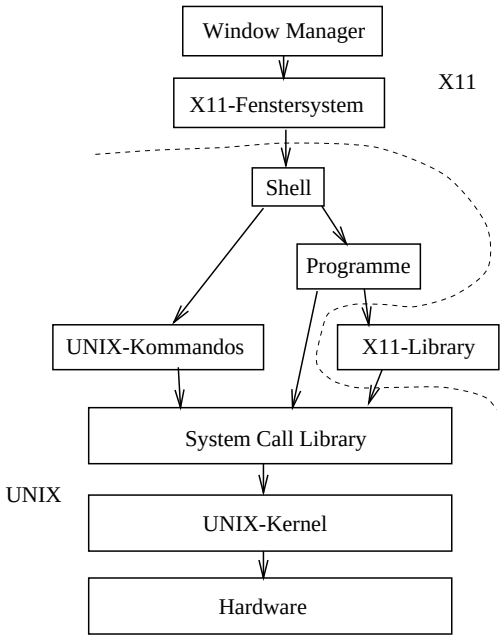
UNIX/C–Praktikum

Montag, 2.12.1996

Heute:

- X11 Windows
- Lokales Netzwerk Informatik, Uni Konstanz
- Internet
- Evaluierungsbogen

Aufbau der Systemsoftware:



X11: Display und X–Server

Display: Tastatur, Bildschirm und Pointing Device (z.B. Maus).

Ein oder mehrere Displays pro Computer.

X–Server: spezieller Daemon (permanenter Hintergrundprozeß).

Genau ein X–Server für jedes Display, auf dem X11 läuft.

Aufgaben von X–Server:

- *Events* (Signale) von Tastatur und Pointing Device an das jeweils „richtige“ Programm weitergeben.
- Ausgaben von Programmen an der „richtigen“ Stelle auf dem Bildschirm auf die „richtige“ Art anzeigen.
- Zugriff anderer X–Server auf Display kontrollieren.
- Ein oder mehrere Screens verwalten

  Beispiel für *zwei* Screens: Farbrechner mit Farb- und Schwarz–Weiß–Modus.

Prozeßmodell unter X11

- X–Server verwaltet Windows auf seinem Display.
- In jedem Window läuft ein Prozeß (*Client*).
- X–Server bestimmt aus Koordinaten des Mauszeigers, welches Fenster aktiv ist, und leitet alle Events an Client in aktivem Fenster weiter.
- Prozeß muß nicht auf demselben Rechner laufen, sondern auf irgendeinem Rechner im Netzwerk.
- Sogar prinzipiell überall auf der Welt via Internet.
- X11 organisiert Kommunikation mit Prozeß über Netzwerke hinweg.
- X–Terminal: Computer, auf dem (im wesentlichen) **nur** ein X–Server läuft (alle anderen Prozesse auf anderen Rechnern).

Windows

Wichtige **Attribute** von Windows:

- Geometrie: Höhe, Breite, Lage auf Bildschirm, gemessen in Anzahl Pixelpunkten.
- Tiefe: Anzahl Bits reserviert pro Pixel.
- Farbklasse: schwarz–weiß, Graustufen, farbig,...
  $\Rightarrow$ Jede Farbklasse verlangt Mindesttiefe.
- Client: Programm, das in Window läuft.

Zusammenhang zwischen einzelnen Windows:

- Alle Windows bilden eine Hierarchie.
- Jede Screen hat genau ein *Root Window* (Hintergrundfenster), die Spitze der Hierarchie.
- „Normale“ Windows mit Rahmen und Menü stehen direkt unter Root Window in der Hierarchie (*Top–Level Windows*).
- Buttons, Textfelder u.ä. in Window sind wieder Windows und stehen hierarchisch unter diesem Window.

Window xterm

- Simuliert rein zeichenorientierten Bildschirm.
- Zeichenorientierter Bildschirm:
  - Bildschirm ist Matrix aus Zeichen.
  - Keine Maus.
  - Keine Graphik.
  - Kein Window System.
- Genauer: Default ist DEC VT102 (zeichenorientiert), möglich ist mindestens auch Tektronix 4014 (Graphik).

```
weihe@hoyren ~ % stty
  speed 9600 baud; evenp hupcl
  rows = 51; columns = 125; ypixels = 769; xpixels = 1144;
  ...
```

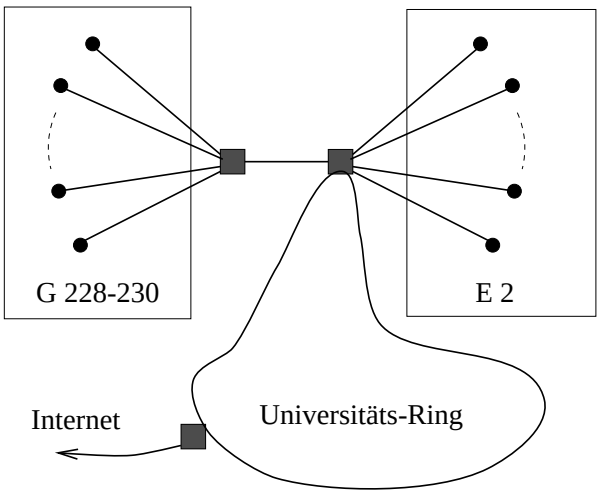
Window Manager (WM)

- Spezieller Client, der in engerer Verbindung zum X–Server steht.
- Maximal ein WM pro Display.
- WM nicht unbedingt notwendig, aber **sehr** hilfreich.
- Typische Aufgaben eines WM:
  - Menüs im Root Window.
  - Rahmen um jedes Top–Level Window mit weiteren Menüs u.ä.

Informationsfluß X–Server $\longrightarrow$ WM:

- X–Server schickt an WM alle Events im Root Window und in Windowrahmen.
- X–Server benachrichtigt WM von jeder Anforderung eines Prozesses, ein Window aufzumachen oder Gestalt von Window zu ändern.

Lokales Netzwerk



Internet: Domain Name System (DNS)

Internet ist hierarchisch in Domains unterteilt.

Beispiel:

„konstanz.pool.informatik.uni–konstanz.de“

- Top–Level Domain: *de*=Deutschland
- *uni–konstanz* ist Subdomain von *de*, *informatik* Subdomain von *uni–konstanz*, *pool* Subdomain von *informatik*
- *konstanz* ist Rechnername in Subdomain *informatik*.

Administratoren einer Domain sind jeweils zuständig für Einteilung in Subdomains und Namensgebung.

Zuständig für Top–Level Domains: Internet Society (ISOC).

Top–Level Domains:

- Eine Top–Level Domain für jedes Land, z.B.

  de: Deutschland.
  at: Österreich (Austria).
  ch: Schweiz (Confederatio Helvetiorum).
  fr: Frankreich.
  uk: Großbritannien (United Kingdom).
  us: USA (ungebräuchlich).

- Ausnahme: weitere Top–Level Domains für USA

  com: Firmen (commercial).
  edu: Schulen, Universitäten (education).
  gov: Verwaltung (government).
  mil: Militär.
  net: Zentrale Netzdienste.
  org: Organisationen, Gesellschaften.

Email–Adressen

Allgemeine Form:
„Benutzername“@„Domainname“

Voraussetzung: Domain muß zentralen Mail–Server haben.

Reale Beispiele:

- weihe@informatik.uni–konstanz.de
  weihek@informatik.uni–konstanz.de
  karsten.weihe@uni–konstanz.de
  weihe@cantor.mathe.uni–konstanz.de
  weihe@math.tu–berlin.de
  karsten@combi.math.tu–berlin.de
- goofy@disney.com
- santa@north.pole.com
  rudolf@north.pole.com
- president@whitehouse.gov
  vice–president@whitehouse.gov

Datentransfer im Internet

- Jeder Rechner, der direkt am Internet hängt (nicht Modem o.ä.), hat eindeutige *IP–Adresse*.
- Möglicherweise auch mehrere IP–Adressen, wenn er an verschiedenen Knotenpunkten im Netz hängt.
- IP–Adresse: 32 Bit, Adresse des lokalen Netzes + Identifikation innerhalb des lokalen Netzes.
- **1. Schritt:** DNS–Name $\longrightarrow$ IP–Adresse:
  - Jede Domain auf jeder Ebene des Domain Name Systems hat einen oder mehrere *Name Server*.
  - Lokaler Rechner fragt sich Schritt für Schritt „in Richtung“ Zieladresse durch, bis ein Name Server genaue IP-Adresse weiß.

Noch: Datentransfer im Internet

2. Schritt: eigentlicher Datentransfer.

- Einzelne Datenpakete (*Datagramme*) werden in „Briefumschläge“ gesteckt (*Internet Protocol, IP*) und einfach an lokalen Router geschickt.
- Internet Protocol: besteht aus Zusatzinfo, z.B. Sender und Empfänger.
- *Router*: Jedes Teilnetzwerk im Internet hat einen oder mehrere Router.
- Router speichert Tabelle, mit der für jede IP–Adresse ein benachbarter Router bestimmt werden kann, der „näher“ an Zieladresse ist.

  $\Rightarrow$ Datentransfer von Router zu Router ohne zentrale Steuerung (=anarchisch).
- Innerhalb von Teilnetzwerk: netzwerkeigenes technisches Protokoll als zusätzlicher „Briefumschlag“ (bei uns: Ethernet).

Noch: Datentransfer im Internet

Internet Protocol: keine Rechnerverbindung.

Datentransfer mit Verbindung: durch Zusatzprotokoll.

- Daten werden in Datagramme zerlegt und einzeln gemäß Internet Protocol losgeschickt.
- In IP–„Briefumschlag“ steckt Umschlag von Zusatzprotokoll mit Zusatzinfos.
- Beispiel Zusatzinfo: fortlaufende Numerierung aller Datagramme (weil Reihenfolge bei Transfer durcheinanderkommen kann).
- Es werden zusätzliche Datagramme mit Kontrollinformationen zwischen Startrechner und Zielrechner ausgetauscht („handshake“).
- Wichtige Kontrollinformationen:
  - Vorab Anfrage von Startrechner, ob Zielrechner bereit für Kommunikation.
  - Empfangsbestätigung von Zielrechner an Startrechner (sonst wird Datagramm erneut losgeschickt).

Beispiel für Zusatzprotokoll: *TCP = Transmission Control Protocol*, benutzt z.B. für email, telnet, ftp...

Wichtige Dienste im Internet

- Email: elektronische Post.
- Talk: schriftliche Rede und Gegenrede.
- Internet Relay Chat (IRC): schriftliche Rede und Gegenrede in beliebig großen, weltweit offenen Gruppen.
- News: Informations- und Diskussionsforen für unendlich viele verschiedene Themengebiete.
- World Wide Web (WWW, W3).
- Adreßverzeichnisse von Personen („*Yellow Pages*“) und Firmen/Institutionen („*White Pages*“).
- Einloggen auf anderem Rechnern.
- File Transfer und Filearchive.
- Zugriff auf Datenbanken.

Wissenswertes zu Email

- Adressen der Form

  Vorname.*Nachname*`@uni-konstanz.de`

  sind Aliasse. Mail–Server des Rechenzentrums verwaltet Tabelle mit „echter“ Email–Adresse für jeden Alias.
- File `~/.signature` für automatische Unterschrift (in `~/.elm/elmrc` eintragen).

  Bsp.: Inhalt von `~weihe/.signature`:

```
Univ. Konstanz    | Phone: ++49-7531-88-4375
Informatik        | Fax:   ++49-7531-88-3577
78434 Konstanz    | Email: karsten.weihe@uni-kon...
Germany           | WWW:   http://www.informatik...
```

- Achtung: Zeichen im erweiterten ASCII–Zeichensatz (z.B. ä, ö, ü, ß) werden nicht von jeder Zwischenstation korrekt weitergeleitet!

File ~/.forward

Beispiel: Emails, die *weihek* bekommt, ungelesen nach *weihe* weiterschicken.

```
weihek@konstanz ~ % cat .forward
  weihe
```

Lokale Kopie in Mailbox von *weihek* aufbewahren:

```
weihek@konstanz ~ % cat .forward
  \weihek, weihe
```

Auch möglich: Emails an Programme weiterreichen.

Standardfall:

```
weihek@konstanz ~ % cat .forward
  \weihe, |"IFS=' ';exec /usr/bin/vacation weihe"
```

Programm „vacation“: Schickt Inhalt von File `~/.vacation.msg` per Email an Absender zurück (urspr. gedacht für Nachricht „I'm out of town“).

WARNUNG: Ohne Setzung von IFS auf ' ' ist so ein `.forward` potentielle Sicherheitslücke!

UNIX–Aufgaben für den Praktikumsschein im UNIX/C++–Praktikum, Wintersemester 1996/97

Alle Aufgaben auf diesen drei Seiten müssen erfolgreich bearbeitet werden, um einen Praktikumsschein zum UNIX–Teil zu erwerben.

Abnahme der Aufgaben: Die erste Rücksprache über die Bearbeitung der Aufgaben findet bis **spätestens** zum Anfang der Vorlesungszeit im Sommersemester 1997 statt. Ich empfehle **dringend**, sich die Aufgaben frühzeitig vorzunehmen und einen Termin für die erste Rücksprache zu vereinbaren, der deutlich vor dem Anfang des Sommersemesters 1997 liegt.

Bei dieser ersten Rücksprache muß für **jede** Aufgabe schon ein funktionsfähiges (wenn auch nicht unbedingt perfektes) Ergebnis vorliegen. Falls das Ergebnis noch nicht in allen Details der Aufgabenstellung entspricht, können eine Nachbearbeitung und eine weitere Rücksprache vereinbart werden (in begründeten Fällen auch mehrmals).

Allgemeine Hinweise:

- Bei Schwierigkeiten aller Art mit den Aufgaben wenden Sie sich bitte an mich. Ich stehe Ihnen gerne beratend zur Seite. Machen Sie davon Gebrauch!
- Falls bestimmte Fragen zu den Aufgaben öfters gestellt werden, werde ich Emails an die Gruppe `unix96` schreiben, in denen diese Fragen noch einmal gesammelt und für alle beantwortet werden. Bevor Sie sich an mich wenden, sollten Sie also zunächst in Ihrer Mailbox nachschauen, ob Ihre Fragen nicht schon beantwortet sind.
- Die schriftlichen Unterlagen zum Praktikum und die Übungsaufgaben und Antworten in `~weihe/unix96/` sind an vielen Stellen hilfreich. An dieser Stelle sei noch einmal daran erinnert, daß die schriftlichen Unterlagen im Sekretariat E 212 kurzfristig zum Kopieren ausgeliehen werden können.
- Konsultieren Sie ausgiebig die jeweils relevanten Man Pages. Wenden Sie auch ausgiebig „`apropos`“ auf Fachbegriffe an, die in den Aufgaben vorkommen.
- Wenn Sie dabei sind, Ihre Shell–Skripte zu debuggen, schreiben Sie „`#! /bin/sh -x`“ als erste Zeile.

Achtung: Zur Erledigung der Aufgaben auf diesem Blatt ist es **nicht** zulässig, Programme in einer Programmiersprache wie C, Pascal oder Fortran zu schreiben. Bei einigen Aufgaben sind Skripte für Kommandointerpreter zu erstellen, was man auch als Programmierung auffassen kann. Diese Ausnahmen sind explizit in der jeweiligen Aufgabenstellung vermerkt.

Aufgabe 1

Legen Sie eine Directory `~/unixc` an und nehmen sie allen Benutzern und Benutzerinnen außer Ihnen selbst sämtliche Rechte für diese Directory. (Benutzen Sie es dennoch nicht zur Ablage privater Daten.) Sich selbst geben Sie volle Rechte auf diese Directory und auf alle Files, die Sie bei der Bearbeitung der folgenden Aufgaben in dieser Directory ablegen.

Aufgabe 2

Erstellen Sie in der Sprache html eine eigene, persönliche Home–Page `~/unixc/ich.html` für das World Wide Web. Diese Page soll mindestens Ihren Namen enthalten sowie ein paar Links auf Pages, die Sie interessant finden (Ihre „*Hot List*“). Außerdem soll sie mindestens ein Bild enthalten. Ihre Page soll durch Kapitelüberschriften und Auflistungen strukturiert sein. Nehmen Sie sich für Formatierung und Strukturierung die Pages des Bereichs Informatik der Universität Konstanz zum Vorbild.

Aufgabe 3

Schreiben Sie in der Sprache LaTeX einen Brieftext Ihrer Wahl und legen Sie ihn in einem File namens `~/unixc/brief.tex` ab. Dieser Text soll mit dem Dokumentenstil (bzw. Dokumentenklasse) „letter“ erstellt sein und mindestens eine Tabelle (tabular–Umgebung), eine Tabulatorliste (tabbing–Umgebung), eine Beschreibungsliste (description–Umgebung) und mindestens ein Bild enthalten, das mit xfig erzeugt wurde. Legen Sie außerdem ein daraus erzeugtes dvi–File `~/unixc/brief.dvi` und PostScript–File `~/unixc/brief.ps` an.

Aufgabe 4

Erstellen Sie ein Skript `~/unixc/aapr` für die Bourne–Shell, das eine beliebig lange Folge $A_1, A_2, A_3, \ldots$ von Zeichenketten als Kommandozeilenargumente bekommt. Ausgabe dieses Skriptes auf stdout ist die Schnittmenge der Ausgaben aller Kommandos

„apropos A_1“, apropos A_2“, „apropos A_3“, ...

das heißt nur die NAME–Zeilen von Man Pages, die bei **jedem** dieser einzelnen Aufrufe von apropos ausgegeben werden. Falls keine Argumente auf der Kommandozeile angegeben werden, gibt das Skript statt dessen eine informative Fehlermeldung aus.

Hinweis: Legen Sie Hilfsfiles in /tmp für Zwischenergebnisse an. Der Name eines solchen Hilfsfiles sollte Ihre BenutzerInnenkennung und die Zeichenkette aapr enthalten (um zufällige Namenskollisionen in /tmp zu vermeiden). Das Skript muß diese Hilfsfiles aber am Ende löschen!

Aufgabe 5

Erstellen Sie ein Skript `~/unixc/mangrep` für die Bourne–Shell, das ein oder zwei Argumente bekommt: einen regulären Ausdruck A wie in der Man Page zu grep definiert und optional eine Directory D. Falls mehr als zwei Argumente angegeben werden oder das zweite Argument keine Directory ist, soll das Skript ohne weitere Aktion mit einer informativen Fehlermeldung terminieren.

Falls eine Directory D auf der Kommandozeile angegeben ist, soll das Skript alle Files bearbeiten, die in D selbst oder in einer direkten oder indirekten Unterdirectory von D enthalten sind, und deren Namen auf das Muster X`.[1-9]` paßt, wobei X eine beliebige Zeichenkette ist. Jedes dieser Files soll wie folgt bearbeitet werden: Wenn das File eine Zeichenkette enthält, auf die der Regular Expression A paßt, soll die Zeichenkette X ausgegeben werden, ansonsten soll nichts für dieses File passieren.

Abgesehen von diesen Ausgaben soll das Skript keine Ausgaben auf stdout oder stderr produ-

zieren.

Falls keine Directory D angegeben wird, soll das Skript auf diese Art statt dessen alle Directories bearbeiten, die in der Environment–Variable MANPATH zusammengefaßt sind.

Hinweis: Fassen Sie alle Operationen, die Sie auf ein einzelnes File anwenden, in einem eigenen Skript zusammen, und wenden Sie dieses Skript auf jedes File an, dessen Namen und Directory wie oben beschrieben heißen. Verwenden Sie „`grep -v`“ und Umlenkung auf `/dev/null`, um unerwünschte Ausgaben auszufiltern.

Aufgabe 6

Legen Sie eine Directory `~/.trash` an und nehmen Sie allen Benutzerinnen und Benutzern außer Ihnen selbst sämtliche Zugriffsrechte für diese Directory.

Erstellen Sie ein Skript `~/unixc/srm` für die Bourne–Shell (srm=safe remove) mit folgender Synopsis:

```
srm [ -r ] file...
```

Ein File wird „*normal*“ genannt, wenn das erste Zeichen in der Anzeige von „`ls -l`“ ein Minus ist. Ist Option `-r` nicht angegeben, darf die Fileliste „`file...`“ nur normale Files enthalten, mit Option `-r` auch Directories. Für jedes Argument in der Fileliste, das diese Bedingung nicht erfüllt, soll eine informative Fehlermeldung ausgegeben werden.

Das Skript `srm` behandelt alle normalen Files in der Fileliste „`file...`“. Ist Option `-r` angegeben, geht das Skript außerdem alle Directories in der Liste „`file...`“ rekursiv durch und behandelt alle normalen Files in diesen Directories und allen ihren direkten und indirekten Unterdirectories.

Ein File F „behandeln“ heißt hier: Falls die Directory, in der F steht, dem Prozeß, der das Shell–Skript ausführt, kein Schreibrecht gewährt, soll eine entsprechende Fehlermeldung ausgegeben werden; ansonsten bewegt der Prozeß das File in die Zieldirectory `~/.trash`. Auch bei Option `-r` werden alle diese Files direkt in die Zieldirectory hineinbewegt, das heißt es werden keine Unterdirectories von `~/.trash` angelegt.

Der Name eines neuen Files in der Zieldirectory soll wie folgt gebildet werden. Sei F der ursprüngliche Name des Files (Ausgabe von `basename`). Existiert noch kein File `~/.trash/`F, dann ist F auch der Name in `~/.trash`. Ansonsten wird solange ein `x` an den Namen F angehängt, bis es kein File `~/.trash/`F`x`...`x` mehr gibt. Das wird dann der neue Name des Files. Die Modification Time des Files `~/.trash/`F`x`...`x` soll gleich dem Zeitpunkt der Bewegung nach D sein (oder kurz danach; es kommt nicht auf die Sekunde an).

Nachdem diese Aktion vollzogen ist, testet `~/unixc/srm`, wieviel Speicherplatz durch die „normalen“ Files in `~/.trash` insgesamt verbraucht wird. Überschreitet die Summe einhundert Kilobytes, so werden soviele Files wieder gelöscht, bis die Gesamtsumme unter einhundert Kilobytes liegt, und zwar auch **nur** so viele, wie dazu notwendig sind. Dabei werden die Files mit der ältesten Modification Time als erstes gelöscht.