Shells: Was passiert mit einem Kommando?

- Shell durchläuft folgenden Zyklus:
  1. Kommando wird aus der Eingabezeile oder einem Script gelesen
  2. Aliases werden expandiert
  3. Variable Substitutions werden vorgenommen
  4. Wildcards (File-Patterns) werden expandiert
  5. Das Kommando wird ausgeführt:
     - Entweder von der Shell selbst (built-in commands)
     - Oder in einem neuen Prozeß
- Im folgenden Annahme tcsh (csh)
  - Sehr häufig ist auch bash, aber IMHO für interaktive CL-Nutzung nicht so komfortabel

1. Aliases

- Ersatz für häufige / längliche Kommandos
- Beispiel:

```
% alias mo less
% alias uni "ssh –l zach –X hera.uni-bonn.de
% mo program.cpp
% uni
```

- **alias** zeigt alle Aliases an

Alias 'uni' wird ersetzt ...
Alias 'mo' wird ersetzt ...
Definiert Alias 'uni'
Definiert Alias 'mo'

2. Variablen

- Variable = Name + Wert, Wert = Zeichenkette
- 2 Arten:
  - Normale Shell-Variablen
  - Environment-Variablen,
    sind auch in Kind-Prozessen (child process) bekannt
- Setzen:

```
% setenv TMP "/tmp"
% set tmpdir = "/home/stud/zach/tmp"
```

- Verwenden:

```
% cp file ${tmpdir}
% echo ${TMP}
```


- Anhängen:

```
% setenv PATH ${PATH}:${HOME}/bin
% setenv PATH ${HOME}/bin:${PATH}
```

Komplexere Variablen-Ersetzung

- Suffix bzw. Präfix löschen:

```
${VAR%suffix}
${VAR#prefix}
```

Löscht Schluß der Variable **VAR**, falls gleich "**suffix**"

Löscht Anfang der Variable **VAR**, falls gleich "**prefix**"

- Beispiel:

```
% set file = "program.cpp"
% mv ${file} ${file%.cpp}.c
% mv ${file} ${file%.p}.c
```

wird ersetzt zu

```
% mv program.cpp program.c
% mv program.cpp program.cpp.c
```


Wichtige Environment-Variablen

Variable	Bedeutung
DISPLAY	Display, auf dem neue Fenster geöffnet werden **(Bsp.: aurikel:0.0)**
HOME	Home-Verzeichnis
PRINTER	Default-Drucker
TMP	Verzeichnis für temporäre Files
PATH	Suchpfad für Kommandos
MANPATH	Suchpfad für Man-Pages
PWD	Aktuelles Verzeichnis, in dem man sich gerade befindet (CWD)

- **printenv** druckt alle Environment-Variablen aus
- **echo $VAR** druckt Wert dieser Environment-Variablen

4. File Patterns

- File-Name mit Wildcards: ***** **?** **[]**

Wildcard	Bedeutung
?	Genau ein beliebiges Zeichen
*	Beliebig viele beliebige Zeichen (auch 0)
{0,1,2}	Genau ein Zeichen aus der Menge {0,1,2}
[0-9]	Genau ein Zeichen aus der Menge {0,...,9}
[a-zA-Z0-9]	Genau ein Zeichen aus der Menge {a,...,z,A,...,Z,0,...,9}
[^0-9]	Genau ein Zeichen *nicht* aus der Menge {0,...,9}

- Beispiele:

```
% ls *.cpp *.h
```

```
% ls [0-9][0-9]*.ppt
```

```
% ls *[^a-zA-Z0-9_.,-]*
```


Kleine Warnung zu **rm**

Task: Shoot Yourself in The Foot

The proliferation of modern programming languages (all of which seem to have stolen countless features from one another) sometimes makes it difficult to remember what language you're currently using. This handy reference is offered as a public service to help programmers who find themselves in such a dilemma.

```
% ls
foot.c foot.h foot.o toe.c toe.o
% rm * .o
rm: .o no such file or directory
% ls
%
```

Quotation

- Achtung: das Kommando sieht die Wildcards nie!
- Shell expandiert Wildcards
- Quotation verhindert, daß Wildcards (allg. Meta-Zeichen) von der Shell expandiert werden
- Arten:
  - \ verhindert Expansion des folgenden Zeichens

```
% echo \*.ppt \$PATH
```

  - "**...**" verhindert Expansion der Wildcards, erlaubt Variablen

```
% echo "*.ppt $PATH"
```

  - '**...**' verhindert jegliche Expansion (Wildcards, Variablen, ...)

```
% echo '*.ppt $PATH'
```


5. Wie die Shell ein Kommando ausführt

1. Built-in: Shell führt Kommando selbst aus
   - Beispiel: **echo**
2. Sonst: externes Programm
   - Beispiel: **ls dir**
3. **command** in **PATH** (Environment-Variable) suchen
4. Falls nicht gefunden, Fehlermeldung
5. Kind-Prozeß erzeugen
   - Erinnerung: erbt Environment des Vater-Prozesses (Shell)
6. Argumente (Zeichenketten) dem Prozeß bereitstellen
   - Argumente werden durch Space (i.a.) getrennt
7. Warten bis Kind-Prozeß beendet

Suchpfade

- Environment-Variable mit Liste von durch **:** getrennten Verzeichnissen
- Beispiel:

```
% echo $PATH
.:/usr/bin:/bin:/usr/local/bin:/home/II/zach
```

- Kommandos werden in **PATH** gesucht
  - File mit Namen des Kommandos im ersten Verzeichnis und executable? → ausführen
  - Sonst: nächstes Verzeichnis in PATH untersuchen ...
- Analog für Man-Pages und andere

File-"Viewer"

- **more, less**
  - Interaktiv, zeigen *jeden* File-Typ im Terminal-Fenster an
  - Unter Linux/Mac ist **more** = **less**
  - Suchen mit '/', weitersuchen mit 'n', rückwärts 'N', u.v.m.
  - Environment-Variable: **setenv PAGER 'less -I'**
- Nicht-interaktiv:
  - **cat** ("concatenate" = aneinanderhängen)
  - **head** , **tail** = ersten / letzten paar Zeilen anzeigen

Editoren

- Programmierer schreiben ASCII, insbesondere Software
  - Nur für reines ASCII (kein "Markup" irgendwelcher Art)
  - Für kleine Listen
  - Zum Editieren irgendwelcher Text-Files
    - Z.B.: VisualStudio-Project-Files, XML-Files, HTML-Files
    - Vor allem zum "remote" Editieren
- Heiliger Krieg, welches der beste ist
- Ein Programmier-Editor sollte ...
  - Effizientes UI haben (i.A. nicht intuitiv!)
    - Wenige Tasten / Mauskilometer für die häufigen Aktionen
  - Syntax Highlighting
  - Makros
  - Reguläre Ausdrücke zum Suchen und Ersetzen
  - Cross-platform sein

Einige Editoren zur Auswahl

- vim / gvim (Obermenge von vi, welcher immer installiert ist)
  - Effizientester Editor, steilste Lernkurve
  - Die Homepage von vim: www.vim.org
- emacs/xemacs (extrem umfangreich)
  - "Emacs wäre gar kein so schlechtes Betriebssystem, wenn es nur einen brauchbaren Editor hätte" ☺
- nedit (kein non-GUI-Mode → taugt nicht zum remote Editieren, sonst durchaus eine Alternative)
- ...

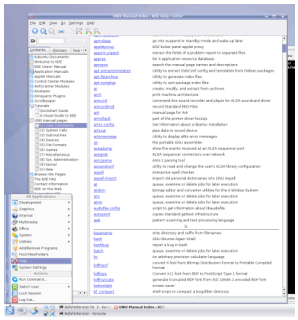
- Meine Empfehlung:
  - Für diejenigen, die sog "power user" werden wollen: suchen Sie sich einen mächtigen, effizienten, und cross-plattform ASCII-Editor aus, und lernen Sie diesen beherschen (dauert eine Weile)
    - **`vim`**, **`emacs`** / **`xemacs`**, **`nedit`**, ...
  - Für alle anderen: nehmen Sie den im Menü angebotenen Editor
    - Unter Linux: **`kedit myfile.txt`** (oder vom "K"-Menü aus)
    - Falls Sie remote editieren möchten / müssen: **`mcedit`**
- Reference-Cards für VIM und Emacs auf der Homepage der Vorlesung

Die wichtigsten Befehle in vi / vim / gvim

- Besonderheit: Vim ist Mode-basiert !!
- Default- ("Home"-) Modus = Kommando-Modus
  - Aus jedem anderen Mode kommt man mit <Esc> dorthin zurück
- Tasten:
  - 'i' → Insert-Mode = Einfügen von Text an der aktuellen Cursor-Pos.
  - 'R' → Replace-Mode = Überschreiben
  - 'x' → aktuelles Zeichen löschen
  - :w → File speichern
  - :q → Vim verlassen
  - '/' → suchen
  - 'n' → nächstes Vorkommen suchen
  - '.' → letztes Kommando wiederholen

More input? Hilfe zur Selbsthilfe

- 4 Arten von Informationsquellen:
  - Man Pages
  - HTML-Seiten
  - Unter KDE: "K"-Menü → Help, dann z.B. "UNIX manual pages"



- Man Pages:
  - **`man cmd`** : zeigt Man-Page zu **`cmd`** an
  - (Programm oder Funktion)
  - **`man -k keyword`** – Alle Man-Pages nach keyword durchsuchen (nur die Titel-Zeile jeder Man-Page)
  - **`man -K keyword`** – Alle Man-Pages nach keyword durchsuchen (komplette Seite)
  - Start-Menü → Help

Format of each man page

Name	Name und 1-zeilige Beschreibung
Syntax	
Description	Ausführliche Beschreibung
Options	
Files	Liste von Files wichtig für diesen Befehl
Return values	
Diagnostics	Mögliche Fehlermeldungen und Ursachen
Bugs	Bekannte Bugs und Unzulänglichkeiten
See also	Verwandte Befehle und Infos

HTML-Seiten

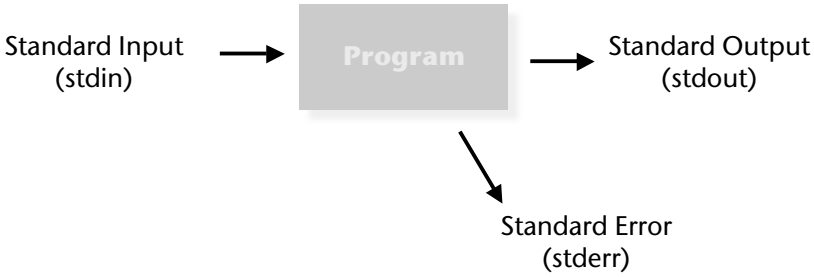
- Hauptproblem: diese zu finden
- Normalerweise in **/usr/share/docs**
- Hilfsmittel: **locate**
- Dann:

Grundregeln unter UNIX

- Don't Panic!
- RTFM! ("read the f*ing manual")
- Probieren geht über studieren ...

Standard-I/O

- Shell etabliert 3 Kanäle zu / vom Prozeß:



- Default-mäßig mit Terminal (Fenster & Keyboard) verbunden

Redirection

- Verbindet Standard-I/O mit Files
- Prozeß (Programm) bemerkt davon nichts!

Redirection	Bedeutung
> file	stdout wird nach file geschrieben
>> file	stdout wird an file angehängt
>& file	**stdout** und **stderr** umlenken
< file	Programm liest aus file, nicht von Keyboard
...	Shell bietet noch viele weitere Möglichkeiten

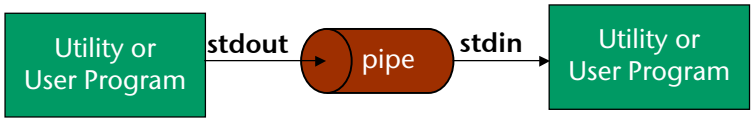
- Beispiel:

```
% ls –l > dir-listing
% sort dir-listing
```

```
% ps -edfyl > procs.log
```

Pipelines

- Effiziente Möglichkeit, komplexere Kommandos aus einfacheren zusammenzubauen!
- Selber Mechanismus wie bei Redirection, um Prozesse miteinander zu verbinden:



- Syntax:

```
% command | command | command | ...
```


Beispiele

- Listings weiterverarbeiten:

```
% ls  | sort | more
% ls -1 *.cpp | wc –l
% ls -1 *.cpp | sort > sorted-dir
```

```
% ls -1 | tr -s "_"
```

Tip:
Bauen Sie eine Pipeline immer eine Stufe nach der anderen auf!!

Unix-Philosophie

- "Small is beautiful"
- Make each tool do *one* thing only
- Make it do it well
- Read from stdin, write to stdout (if sensible)
- Use ASCII-Files

Filter

- Sind hauptsächlich dazu gedacht, Pipelines aufzubauen
- Generelle Verwendung: Filter lesen von stdin, schreiben auf stdout
- Arbeiten oft zeilenweise

Utility	Funktion
cut	Felder oder Zeichenspalten ausschneiden
fmt	Auf 72 Zeichen umformatieren
sort	Zeilenweise sortieren (auch nach Teil-Key)
uniq	Duplikate entfernen
wc	Zeichen, Wörter, und Zeilen zählen
tr	Zeichen ersetzen
grep	Zeichenketten in der Eingabe suchen (s.u.)
rev	Reihenfolge der Zeichen umkehren (zeilenweise)

Beispiel

- Alle Filegrößen in einen File ausgeben:

```
% ls -l | tr -s " " | cut -d' ' -f5 > sizes.txt
```

- Nur die letzten paar Zeilen eines langen Directory-Listings:

```
% ls -l | tail -10
```


Suchen & Finden

- **find**: findet Files aus einem Verzeichnisbaum
  - Kriterien: Name, Datum, Größe, ...
  - Aktionen: Filename ausgeben, Löschen, ...
- **grep**: findet Zeilen in einem File, in denen bestimmte Zeichenketten vorkommen
- **locate**: findet Files aufgrund autom. erzeugter DB

Find

- Syntax:

  find dirs ... ***criteria actions***

- Kriterien:

  -name ***file-pattern***
  -type ***file-type***
  -size ***file-size***
  -date ...

  Aktionen:

  -print
  -exec ***command*** **{} \;**
  -ls
  ...

Beispiele

- Den File foo im Home finden:

```
% find $HOME –name foo
```

- Wenn man den Namen nicht mehr genau kennt:

```
% find $HOME –iname '*foo*'
```

- Alle JPEG's finden und File-Namen in File schreiben:

```
% find $HOME –name '*.jpg' > image-list
```

- Alle JPEG's größer als 100kB finden:

```
% find $HOME –name '*.jpg' –size +100k
```

- Alle core's löschen:

```
% find . –name core –print –exec rm {} \;
```

- Alle Files in einem Verzeichnis-Baum zählen, die auf '.png' enden:

```
% find . -iname '*.png' | wc -l
```


Grep

- Syntax:

  grep 'reg-exp' files ...

- Varianten: fgrep, egrep
- reg-exp = regular expression (eine Art Pattern-Matching)
- Default: Zeilen ausgeben, die matchen
- Einige Optionen:

  -v Invertierung: Zeilen ausgeben, die nicht matchen
  -i case-insensitive
  -n Zeilennummern ausgeben
  -H Filenamen zu den Matches ausgeben
  -e RE weitere reg-exp's (Oder-Verknüpfung)

Reguläre Ausdrücke

- Besteht aus normalen Zeichen und Meta-Zeichen:
  - Meta-Zeichen: . **?** [ ] * + $ ^ \ ( )
  - Normale Zeichen: alle übrigen
- Regulärer Ausdruck = Zeichenkette aus normalen Zeichen und Meta-Zeichen
- Matching:
  - Vergleicht gegebene Zeichenkette und RE von links nach rechts
  - Arbeitet Zeichen ab, falls sie, gemäß Regeln, "übereinstimmen"
  - Arbeitet "greedy"
- Extended RE's

- Definition einer *regular expression:*

Zeichen	Bedeutung / Match
a	matcht das Zeichen selbst
. (Punkt)	matcht ein beliebiges Zeichen
[abc-f]	matcht ein Zeichen aus {a,b,c,d,e,f}
[^abc]	matcht ein Zeichen *nicht* aus {a,b,c}
^ $	stehen für den Anfang/Ende der Zeile
a?	a ist optional ("schluckt" a, falls vorhanden)
a+	a muß einmal oder öfter vorkommen
a*	a darf belieb oft, auch keinmal, vorkommen
(*RE*)	Gruppierung
RE1*|*RE2	matcht a oder b
\	hebt Bedeutung des nachfolgenden Meta-Zeichens auf
...	

Beispiele

- **grep 'abc' file:**
  - Zeigt alle Zeilen, die "abc" enthalten
- **grep 'a.c' file:**
  - alle Zeilen, die "axc" enthalten, wobei x beliebiges Zeichen ist
- **grep –n 'my_function *(' my_code.c :**
  - alle Zeilen, wo **my_function** aufgerufen wird (oder deklariert wird)
- **grep 'a\[[^]]*\]=' file:**
  - alle Vorkommen der Form "a[...]=", wobei ... eine beliebig lange Zeichenkette ist, die kein] enthält
- **grep 'a\[[^]]*\] *=' file:**
  - wie vorher, mit beliebig vielen Spaces zwischen "]="

- Bestimmte Strings entfernen
- Beispiel: ein File enthält eine große Tabelle, die so aufgebaut ist:

```
<table>
[...]
<tr height="12">
<td height="12" align="right">324557</td>
<td class="xl24" align="right">5.0</td>
</tr>
<tr height="12">
<td height="12" align="right">329356</td>
<td class="xl24" align="right">3.3</td>
</tr>
[...]
</table>
```

- Aufgabe: alle vorkommen von height=".." entfernen
- Lösung: ein Editor, der reguläre Ausdrücke beherrscht

- Der reguläre Ausdruck, der die gesuchten Vorkommen matcht:

```
/ height=".."/
```

oder

```
/ height="[0-9][0-9]"/
```

oder

```
/ height="[0-9]*"/
```

oder

```
/ height="[^"]"/
```


- Welche Prozesse laufen gerade von mir:

```
ps -auxw | grep mylogin
```

Linux (und andere von AT&T abgeleitete Unixes)

```
ps -edfl | grep mylogin
```

BSD-Unices (z.B. Mac OS X)

- Das Ganze etwas eleganter als Alias:

```
alias myps 'ps auxw | grep mylogin'
```

- Und noch eleganter, damit es mit jedem Login automatisch funktioniert:

```
alias myps 'ps auxw | grep `id –un` '
```

- Eine Art Spell-Checker für Web-Seiten:

```
curl "http://zach.in.tu-clausthal.de" |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - /usr/dict/words
```

1. **curl** liefert die angefragte Webseite auf stdout
2. **sed** löscht alle Zeichen, die keine Spaces oder Buchstaben sind
3. **tr** ersetzt alle Großbuchstaben durch kleine Buchstaben; außerdem ersetzt es Spaces durch Newline; jetzt sind alle Wörter auf einer eigenen Zeile
4. **grep** löscht alle Zeilen, die leer sind (nur Whitespace enthalten)
5. **sort** sortiert die Liste der Wörter und löscht doppelte Wörter
6. **comm** findet Wörter (Zeilen), die nicht im Wörterbuch enthalten sind (hier **/usr/dict/words**).

xargs

- Oft soll am Ende einer Pipeline ein Kommando stehen, das die Eingabe von **stdin** als Parameter (typ. Filename) verwendet
- Beispiel:

- Lösung: **xargs**

```
% ... | xargs -L 1 command
```

  - **xargs** nimmt jede Zeile von **stdin** und formt damit ein Unix-Kommando, indem diese Zeile nach ***command*** angefügt wird und als Befehl ausgeführt wird

Beispiel

- Aufgabe: alle Files in '**duplicates.txt**' löschen, die in Unterverzeichnis **astronomie** sind, und doppelt vorhanden sind. Alle anderen nicht berühren.

duplicates.txt

```
artchive/Art/ARTCHIVE ON CDROM/artchive/ftptoc/altdorfer.gif
artchive/Art/ARTCHIVE ON CDROM/artchive/A/altdorfer.gif .
artchive/Art/ARTCHIVE ON CDROM/artchive/ancient.html
artchive/Art/ARTCHIVE ON CDROM/artchive/old/ancient.html .
artchive/Art/Images/m/marc/animals.jpg
unm.edu~randy/Others/animals.jpg .
astronomie/astronomy_pic_of_the_day/0007/anticenter_glastsim.jpg
astronomie/astronomy_pic_of_the_day/9811/anticenter_glastsim.jpg .
astronomie/astronomy_pic_of_the_day/9811/anticenter_glastsim_big.gif
astronomie/astronomy_pic_of_the_day/0007/anticenter_glastsim_big.gif .
astronomie/astronomy_pic_of_the_day/0006/apollo17_night.jpg
astronomie/astronomy_pic_of_the_day/9705/apollo17_night.jpg .
```

```
% grep astronomie duplicates.txt | grep '\.$' |
  cut -d' ' -f1 | xargs -L 1 rm
```


Weitere Tools / Utilities

Utility	Funktion
more	File anzeigen (**more** ist ein sog. Pager)
less	Noch besser als **more**
head / tail	Anfang / Ende des Files ausgeben
cat file	File ausgeben (keine Funktionalität)
echo string(s)	String(s) auf stdout (typ. Terminalfenster) ausgeben
diff file1 file2	Unterschiede zwischen 2 Files anzeigen
du –sk dirs ...	Speicherbedarf der Verzeichnisse in kB anzeigen
df –h dir	Größe und freien Platz auf einer Platte anzeigen
df -hl	Größe und freien Platz aller lokalen Platten anzeigen
quota -v	Freie Quota anzeigen
lpr [–P*drucker*] file.ps	Postscript-File ausdrucken
lpq [–P*drucker*]	Printer-Queue anzeigen
a2ps [–P*drucker*] file	ASCII-File (z.B. Listing) ausdrucken

Utility	Funktion
id	Eigene IDs ausgeben
who / w	Wer ist eingeloggt?
date	Datum anzeigen
cal	Jahreskalender anzeigen
locate *file*	Ort von File anzeigen (auch Teilstrings)
where *command*	Ort(e) von Command anzeigen
tar czf archive.tgz dirs ...	Komplettes Verzeichnis (inkl. Unterverzeichnisse) zusammenpacken und komprimieren
tar xzf archive.tgz	Archiv wieder auspacken
gzip/gunzip	File komprimieren / dekomprimieren
mount /mnt/floppy	Floppy mounten / unmounten

Shell Scripts

- Die Shell hat 2 Aufgaben:
  - Als Kommandozeilen-Interpreter, d.h., als direkte Schnittstelle zum Kernel
  - Als Programmiersprache
- Shell-Programme heißen *Script*
- Sprachumfang:
  - Alle Kommandos / Syntax bisher
    - D.h., alles was man interaktiv schreiben kann, kann man auch als Script schreiben
  - Die üblichen Konstrukte (Schleifen, Verzweigungen, etc.)
- Genereller Aufbau:
  - Ganz normaler ASCII-File
  - Spezielle erste Zeile
  - Executable-Bit setzen: **`chmod u+x script`**
  - Ausführen mit **`./script`**, oder in ein Verzeichnis im **PATH** moven

```
#!/bin/bash -p

# Kommmentar
# ...

commands ...
```