Find — in aller Kürze

- Das Kommando zum suchen nach Files unter bestimmten Kriterien
- Häufigste Verwendung:

```
% find directory -name 'file-pattern'
```

- Liefert (fast) immer eine Liste von Files, die auf die Kriterien passen

Beispiele

- Den File foo im Home suchen (rekursiv!):

```
% 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 Backup-Kopien löschen:

```
% find . –name '*.bak' –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 (regulärer Ausdruck)
- Default: alle Zeilen ausgeben, von denen ein Teil den regulären Ausdruck matcht
- Einige Optionen:

  -v Invertierung: Zeilen ausgeben, die nicht matchen

  -i case-insensitive

  -n Zeilennummern mit ausgeben

  -H Filenamen zu den Matches ausgeben

  -e *RE* weitere reguläre Ausdrücke (Oder-Verknüpfung)

Reguläre Ausdrücke

- Regulärer Ausdruck ist ein String bestehend aus normalen Zeichen und Meta-Zeichen:
  - Meta-Zeichen = **. ? [ ] * + $ ^ \ ( )**
  - Normale Zeichen = alle übrigen Zeichen
- Bedeutung: Regulärer Ausdruck *RE* beschreibt eine Menge(!) $\mathcal{L}_{RE}$ von Strings (= Zeichenketten)
- Sprechweise:
  Ein String *S* matcht den regulären Ausdruck *RE* $\Leftrightarrow S \in \mathcal{L}_{RE}$
- Matching:
  - Vergleicht gegebene Zeichenkette und RE von links nach rechts
  - Arbeitet Zeichen ab, falls sie, gemäß Regeln, "übereinstimmen"
  - Arbeitet "greedy"

Bedeutung der Meta-Zeichen (und damit Definition der *regular expressions* bzw. Definition von $\mathcal{L}_{RE}$)

Zeichen	Bedeutung / Match
a	matcht das Zeichen selbst (heißt Literal)
. (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}
^ $	matchen 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
Achtung: bei **?**, + und * kann **a** eine beliebige RE sein!	
(*RE*)	Gruppierung
RE1*|*RE2	matcht a oder b
\	hebt Bedeutung des nachfolgenden Meta-Zeichens auf (Quotation)

Beispiele für reguläre Ausdrücke

Regulärer Ausdruck	Strings, die damit matchen (= $\mathcal{L}_{RE}$)
[01]+0[01]	Alle Strings bestehend aus 0 und 1, deren vorletztes Zeichen eine 0 ist
[a-zA-Z]+	Wörter nur bestehend aus Buchstaben (ohne Umlaute)
(ab)+	ab, abab, ababab, ...
(aa|bb)+	aa, bb, aabb, bbaa, aabbaa, bbaabb, ...
[0-9]+\.[0-9]+	0.0, 0.1, 7.2, 00099.999000, ...

Unterschied zwischen File Patterns und Reg-Exp's

- Achtung: die Meta-Zeichen * und ? bedeuten in File-Patterns etwas anderes als in regulären Ausdrücken!
  - Das File-Pattern

    ab*c

    matcht auf die Strings (= File-Namen)

    abc, abxc, ab123c, ab@!#$c, ...
  - Der reguläre Ausdruck

    ab*c

    matcht auf die Strings

    ac, abc, abbc, abbbc, ...

Beispiele mit grep

- **grep 'abc' file:**
  - Zeigt alle Zeilen, die "abc" enthalten
- **grep 'a.c' file:**
  - alle Zeilen, die "axc" enthalten, wobei x ein 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 (also Zuweisungen an ein Element des Arrays a)
- **grep 'a\[[^]]*\] *=' file:**
  - wie vorher, mit beliebig vielen Spaces zwischen "]="

- **grep '(Mr\.|Mrs\.|Ms\.|Dr\.) [A-Za-z]+' sample.txt:**
  - Zeigt alle Zeilen mit einer Anrede an
- **grep –i 'href=' web_page.html:**
  - Zeigt alle Zeilen mit einem Link an (-i = ignore case)

- Email-Harvesting:
  **grep –i '[a-z0-9_\.-]+@([a-z0-9_-]\.)+[a-z0-9][a-z0-9]+' **
  web_page.html:
  - Zeigt alle Zeilen mit einer Email-Adresse an
- Erläuterung:
  - **[a-z0-9_\.-]+** = xxx, xxx.xxx , xx-xx.xx , xx_xx. , etc.
  - **@** = @-Zeichen
  - **[a-z0-9_-]\.** = xxx., xxx-xxx. , xx_xx. , etc.
  - **([a-z0-9_-]\.)+** = Folge von xxx. (mindestens 1)
  - **[a-z0-9][a-z0-9]+** = Wort mit mindestens 2 Zeichen, ohne ./- (die "top-level domain")
- Bemerkung: diese Regex matcht auch ungültige Email-Adressen!

Reguläre Ausdrücke in Text-Editoren

- Beispiel: ein HTML-File enthält folgende große Tabelle

```
<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

- Das Ganze etwas eleganter als Alias:

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

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

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

Ein einfacher 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 | uniq |
comm -23 – words.txt
```

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 | uniq** sortiert die Liste der Wörter und löscht doppelte
6. **comm** findet Wörter (Zeilen), die nicht im Wörterbuch enthalten sind (hier **/usr/dict/words**).

Tools, die reguläre Ausdrücke verwenden

- grep
- vi / vim / emacs — Text-Editoren (vim ist der beste ☺)
- awk / gawk / nawk — Pattern scanner / processor
- sed — Stream editor
- Perl — Skriptsprache mit einer extrem mächtigen Regular-Expression-Engine für Text-Processing
- Google's Codesearch: http://www.google.com/codesearch
- PHP, Javascript (Programmiersprachen für Web-Seiten)

Beispiel mit **sed**

- Aufgabe: einen File dieses Formats

```
1,23; 4,0; 3,1415926;
100;1000,0; 10E3;0,0001
...
```

in einen File dieses Formats umwandeln

```
1.23 4.0 3.1415926
100 1000.0 10E3 0.0001
...
```

- Lösung:

```
sed -e 's/;/ /g' -e 's/,/./g' tabelle1.txt > tabelle2.txt
```

"global" flag = alle Vorkommen in einer Zeile ersetzen

substitute command

Weiterführende Links

- Formale Definition:
  http://en.wikipedia.org/wiki/Regular_expression
  - Definiert eine wesntlich umfangreichere Syntax, zur Beschreibung von sog. "extended regular expressions"!
- Video-Tutorial (Kopie ist auf der VL-Homepage):
  http://www.multiurl.com/ga/regexdummies
  - Geeignet für Anfänger
  - Ist allerdings zugeschnittenauf PHP und Javascript, d.h., man muß in der Lage sein, von den PHP-spezifischen Teilen zu abstrahieren
  - Auch dieses Tutorial präsentiert extended regular expressions

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

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

Editoren

- Programmierer schreiben ASCII ...
  - 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" ☺
- kate / kwrite oder 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
  - :q! → Schließen ohne zu speichern

Programmierung von Shell Scripts

- Die Shell hat 2 Aufgaben:
  - Als Kommandozeilen-Interpreter, d.h., als direkte Schnittstelle zum Kernel
  - Als Programmiersprache
- Ein Shell-Programm heißt *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 ...
```

Parameter von der Kommandozeile

- Ein Shell-Skript ist ein ganz normales Kommando, wie jedes andere auch
- Fazit 1: Shell-Skripte werden im PATH gesucht
  - Wer "langlebige" eigene Shell-Skripte schreibt, sollte folgendes machen:
  - Verzeichnis **~/bin** anlegen
  - **setenv PATH ${PATH}:${HOME}/bin**
- Fazit 2: Shell-Skripte können Parameter erhalten
  - Z.B. bei einem Aufruf der Art

```
% my_script hallo
```

  - Diese Parameter kann man innerhalb des Skriptes durch die speziellen Variablen **$1 $2** etc. abfragen (reine Zeichenketten):

```
#!/bin/bash -p
echo $1
```


Beispiel: ein reversibles **rm**-Kommando

- Erinnerung:

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

- Eine Idee: ersetze **rm x** durch **mv x ~/.Trash**
- Lösung: ein Shell-Skript

```
#!/bin/bash -p

mv $1 ~/.Trash
```

  - Speichere das Shell-Script in **~/bin** (z.B.)
  - Trage **~/bin** in **PATH** ein
  - Evtl. noch:

```
% alias rm rmm
```

For-Schleifen in Shell-Skripts

- Syntax:

```
#!/bin/bash -p

for x in word-list
do
    commands
done
```

  - Weist x nacheinander die Wörter in *word-list* zu, und arbeitet die Kommandos ab. Innerhalb dieser kann x wie jede andere Variable verwendet werden.

- Beispiel:

```
#!/bin/bash -p

for f in *.png
do
    ls -l $f
done
```


Weitere Beispiele

- Häufiges Problem: Liste von Files umbenennen

```
#!/bin/bash -p

for f in *.cxx
do
    mv $f ${f%cxx}cpp
done
```

Wird von der Shell expandiert, bevor die Schleife gestartet wird!

- Etwas kompliziertere Umbenennung:
  - Gegeben: ein Verzeichnis mit 100 Files, von denen manche eine Nummer als Präfix haben, manche nicht

```
Private SNAFU - Home Front.mp4
01 - Private SNAFU - Snafuperman.mp4
Aesop's Fables - Silvery Moon (1933).mpg
02 - Aesop's Fables - 1927.mpg
A is for Atom.mp4
```

  - Aufgabe: die Nummer am Anfang des File-Namens löschen
  - Lösung:

```
#!/bin/bash -p

for f in [0-9][0-9]*.mp4
do
   mv $f ${f#.. - };
done
```


- Eine Menge von Bildern croppen (z.B., um danach einen Film daraus zu machen)
  - Annahmen: PWD enthält die Bilder (JPEG); Verzeichnis **../cropped** soll die ge-crop-ten Bilder (selbe Filenamen) enthalten
  - Script:

```
#!/bin/bash -p

for f in *.jpg
do
  echo $f
  convert -crop 1100x900+250+0 -quality 95 $f ../cropped/$f
done
```

- Nach dem Kopieren des Bildarchives einer Web-Site entstand folgendes Unterverzeichnis:

```
hostname: bildarchiv/pics% ls
1000_1137871168348b.jpg  1644_11518627567274.jpg  435_113416441328e5.jpg
1000_1137871168dc0d.jpg  1644_11518627567371.jpg  435_11341644136720.jpg
1001_1137871169470c0.jpg  1644_1151862756dc9b.jpg  435_11341644139fb5.jpg
1002_1137871170dc0d.jpg  1644_1151862756f176.jpg  435_1134164413a1db.jpg
1003_1137871170470c0.jpg  1645_11518627561778.jpg  435_1134164413d226.jpg
1004_1137871171fa36.jpg  1645_11518627566227.jpg  435_1134164413d42c.jpg
1005_11378711721408.jpg  1645_1151862756a590.jpg  436_11341648314031.jpg
```

- Problem: alle Bilder der Form 123_abcdef.jpg waren exakte Kopieen voneinander
- Aufgabe: von jeder solchen Menge von Kopien alle bis auf eine löschen
- Lösung:

```
$ for f in *.jpg; do mv $f ${f%_*}.jpg; done
```


Variablenzuweisungen im Shell-Skript

- Syntax der Zuweisung:

```
#!/bin/bash -p
var=xxxxx
```

  - ACHTUNG: KEIN SPACE LINKS ODER RECHTS VOM =-ZEICHEN!!
- Verwendung von Variablen:

```
#!/bin/bash -p
var=xxxxx
echo $var
```

- Beispiel:

```
#!/bin/bash -p
trash=~/.Trash
mv $1 $trash
```

  - Vorteil: falls man das Trash-Verzeichnis einmal ändern möchte, braucht man im Skript nur eine Zeile zu ändern

Bedingungen in Shell-Scripts

- Syntax:

```
#!/bin/bash -p
if [[ bedingung ]]
then
    ...
fi
```

- Beispiele:

```
if [[ -f filename ]]  # Test, ob File filename existiert
if [[ -d filename ]]  # Test, ob filename existiert und
                        ein Verzeichnis ist
if [[ ! bedingung ]]  # Test, ob Bedingung nicht erfüllt ist
```

  - Siehe die Man-Page der bash für viele weitere Tests

- Anwendung auf unser **rm**-Beispiel:
  - Was, wenn das Verzeichnis **~/.Trash** noch nicht existiert?
- Lösung:

```
#!/bin/bash -p

trash=~/.Trash

if [[ ! -d $trash ]]
then
    echo Creating directory $trash
    mkdir $trash
fi

mv $1 $trash
```

Minimalwissen

- Folgende Befehle sollten Sie gut beherrschen:
  - ls, cd, mkdir, rmdir, cp, mv, rm, ln, chmod,
  - less, cat, sort, wc
  - grep, find
  - ps, jobs, kill
- Lesen Sie deren Man-Pages (immer wieder)

Empfehlung für Unix-Guru-Aspiranten

- Die Kommandos
  - **`sed`** - "stream editor"
  - **`awk`** - pattern scanning and processing language
  - **`tr, cut, sort, uniq`**, ...

Initialisierungs-Files

- Für tcsh: **~/.tcshrc** (und **~/.cshrc)**
- Für bash: **~/.bashrc**
- Inhalt: beliebige Kommandos (shell-built-in & extern)
- Ausgeführt von jeder Shell des Users
- Typischerweise:
  - Environment-Variablen setzen (**PATH**, **MANPATH**, ...)
  - Aliases definieren
  - Completion-Regeln (für Tab an verschiedenen Positionen)
- Achtung: neue Shell aufmachen, falls Änderung

- Empfehlung: Dot-Files von der Vorlesungs-Home-Page installieren
  - Downloaden Sie http://zach.in.tu-clausthal.de/teaching/werkzeuge/downloads/dotfiles.tar.gz
  - Ins Home verschieben: **mv ..../dotfiles.tar.gz ~**
  - Sicherheitskopie das alten .cshrc: **mv .cshrc .cshrc.old**
  - Auspacken: **tar tzvf dotfiles.tar.gz**
  - Neue Shell aufmachen (zum Test)