Mitschrift der Vorlesung
Open-Source-Programmierwerkzeuge SS 2012
von
Norman Ulbrich

Eric Schemm, eric.schemm@mni.thm.de
Michael Ringlebe, michael.ringlebe@mni.thm.de

24. Mai 2012

Zusammenfassung

Dies ist eine Ergänzung zur Mitschrift von Fabian Müller aus dem SS09, Version 2.1

1 ant

Für Java Projekte bietet sich die Benutzung von ant anstelle von make an, um auch den Buildprozess portaben zu halten. ant ist in Java geschrieben und benötigt eine Javalaufzeitumgebung. Die Konfigurationsdatei wird in XML geschrieben und heißt gewöhnlich build.xml. Ähnlich zu (CC),(RM) gibt es bei ant vordefinierte Tasks wie "javac", "delete", "mkdir"und "cus". Bei der Konfiguration werden ihre Attribute als XML-Attribute angegeben. Dieser Ansatz dient der Portabilität, weil die Programme zum Durchführen der Operation nicht direkt aufgerufen werden. ant weiß, auf welchem System es läuft, und welche Programme dort zu verwenden sind. Beispiel:

```
make: rm -rf classes
ant: <delete dir="classes" />
```

Man kann eigene Tasks in Java implementieren, die dann auf die gleiche Weise wie die vordefinierten Tasks verwendet werden können. Das macht ant zwar sehr flexibel, aber da es keine andere Möglichkeit gibt, Anweisungen "einfach so"hinzuschreiben, ist es recht umständlich, ein neues Kommando hinzuzufügen. Möglicherweise gibt es das gewünschte Kommando in einer Tasksammlung, die man im Internet herunterladen kann.

Beispiel für eine Konfigurationsdatei (Quelle: `http://en.wikipedia.org/wiki/Apache_Ant`):

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application'
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
```

```
            </manifest>
        </jar>
    </target>
</project>
```

2 Reguläre Ausdrücke

2.1 Was sind reguläre Ausdrücke?

Reguläre Ausdrücke sind eine Sprache zum "Erkennen"von Text. Ein regulärer Ausdruck stellt einen Suchmuster dar, dass mit einer Zeichenkette "verglichen"wird. Je nach Anwendung wird bei einer Übereinstimmung (oder bei keiner Übereinstimmung) die entsprechende Aktion durchgeführt, z.B. die Zeile ausgegeben oder eine Textersetzung durchgeführt. Da es unter Unix schon früh Bibliotheken dafür gab, gibt es viele Programme und Programmiersprachen, die reguläre Ausdrücke mehr oder weniger intensiv nutzen.

Es gibt verschiedene Dialekte für reguläre Ausdrücke, aber wenn das Prinzip klar ist, gewöhnt man sich schnell an die jeweils verwendete Syntax.

2.2 Aufbau von regulären Ausdrücken

Ein regulärer Ausdruck ist eine Zeichenkette, die "normale"und "spezielle"Zeichen enthält. Ein normales Zeichen steht beim Vergleich für sich selbst. Ein spezielles Zeichen verändert den Teilausdruck auf den es angewendet wird. In den meisten Fällen ist dieser Teilausdruck ein "Atom", besteht also entweder aus einem einzelnen Zeichen oder einem geklammerten Ausdruck.

Spezielle Zeichen sind (unter Anderem):

- () Gruppierung
- . irgendein Zeichen (außer Newline, abhängig vom Dialekt)
- [] eines der Zeichen, die in Klammern stehen; Bereiche sind möglich, z.B. [o-g]
- [^] ein Zeichen, das nicht in den Klammern steht
- | Oderverknüpfung. Der linke oder der Rechte Teilausdruck
- ? Atom kommt null oder einmal vor
- * Atom kommt null bis n-mal vor
- + Atom kommt ein bis n-mal vor
- n Atom kommt genau n-mal vor

n,	Atom kommt mind. n-mal vor
n,m	Atom kommt n- bis m-mal vor
,m	Atom kommt höchstens m-mal vor
\	das nächste Zeichen steht für sich selbst (bzw. steht nicht für sich selbst, je nach Dialekt, Einstellungen und folgenden Zeichen
^	Zeilenanfang (Länge 0)
\$	Zeilenende (Länge 0)
\<	Wortanfang (Länge 0)
\>	Wortende (Länge 0)
\b	Wortgrenze (Länge 0) Wortanfang oder Wortende

Der reguläre Ausdruck . entspricht ? in der Shell, .* entspricht * (außer bei führenden Punkt), und die Zeichenauswahl mit [ ] bzw. [^] bedeuten beide das Gleiche. Außerdem gibt es in vielen Dialekten Zeichenklassen, die ebenfalls in eckigen Klammern geschrieben werden. Beispiele:

[[:alpha:]]	alphabetische Zeichen
[[:alnum:]]	alphanumerische Zeichen
[[:space]]	Whitespaces (u.A. Space und Tabulator)
[[:upper:]]	Großbuchstaben
[[:lower:]]	Kleinbuchstaben
[[:cntrl:]]	Kontrollzeichen

Je nach Dialekt sind die Zeichenklassen möglicherweise nicht gerade das, was man erwartet. [[:lower:]] enthält zwar bestimmt [a-z], kann bei der Spracheinstellung deutsch aber z.B. die kleinen Umlaute und das ß enthalten. Zum Glück ist das in vielen Fällen kein Problem. Quelltexte enthalten selten derartige Sonderzeichen.

Beispiele für reguläre Ausdrücke:

hallo	⇒	hallo
h[ae]llo	⇒	hallo; hello
abc|def	⇒	abc; def
h(a|e)llo	⇒	hallo; hello
a.b	⇒	a, irgendein Zeichen, b in derselben Zeile
a\.b	⇒	a.b

hell?o	⇒	helo; hello
ab*c	⇒	ac; abc; abbc; abbbc; ...
ab+c	⇒	abc; abbc; abbbc; ...
a(bc)?d	⇒	ad; abcd
a(bc|de)?fg	⇒	afg; abcfg; adefg
[1-9][0-9]*	⇒	ganze Dezimalzahl
[A-Za-z_][A-Za-z_0-9]*	⇒	Bezeichner
^const	⇒	const am Zeilenanfang
{\$	⇒	{ an Zeilenende
[^][()]	⇒	irgendein Zeichen, außer runde und eckige Klammern
[-+*/]	⇒	Plus-, Minus-, Multiplikations-, oder Divisionszeichen

2.3 Gespeicherte Teilausdrücke

Beim Ersetzen von Text ist es oft nötig, einen Teilsaudruck den man mit unspezifizierten Zeichen angegeben hat, unverändert zu lassen.
Zum Ersetzen von Hex Zahlen in C-Notation in eine Notation mit angehängtem h muss die Ziffernfolge eingesetzt werden die dem 0x folgt.

0x12 wird zu 012h, 0xAB12 zu 0ab12h

Als Suchstring eignet sich: \<0x\x+\>(\x steht für eine Hexziffer)
(für Hexzahl: 0x[0-9a-fA-F])

Dieses Beispiel unterstützt kein optionales Suffix (mit dem man in C einen Typ spezifizieren kann). Es geht nun darum, im Ersetzungsstring das einzusetzen, auf das \x+ im Suchstring gepasst hat.

\x = [0-9a-fA-F] Hexziffern

Für solche Operationen kann man Rückwärtsreferenzen auf geklammerte Teilausdrücke verwenden. Was in Klammerpaar 'i' steht, kann später mit \i verwendet werden. Klammerpaare werden gezählt, indem die öffnende Klammer von links nach rechts gezählt werden. Beim Suchstring \<0x(\x+)\>werden die gefundenen Hexzahlen in der Referenz 1 gespeichert. Der Ersetzungsstring erzeugt also eine Hexzahl in der gewünschten Schreibweise.

Bespiele:

Suche nach:	a(bc)d*e	In Text:	abcdde
Ergebnis:	\1: bc\2: gibt es nicht		
Suche nach:	a(b*)c	In Text:	abbbc
Ergebnis:	\1: bbbb		
Suche nach:	a(b*)c	In Text:	ac
Ergebnis:	\1: (leer) ∈		

Suche nach:	a(b*)(c+)d	In Text:	abbcd
Ergebnis:	\1: bb \2: c		
Suche nach:	a(b*)(c+)d	In Text:	ad
Ergebnis:	passt nicht! c muss einmal vorkommen!		

Die führende Null soll verhindern, dass ein Bezeichner entsteht, wenn die erste Hexziffer ein Buchstabe ist. Wenn Konstanten wie 001h oder 04h ein Problem darstellen, kann man eine weitere Ersetzung durchführen. Wenn eine Hexzahl mit Null und einer weiteren Dezimalziffer beginnt, wird die Null entfernt. Dazu kann man den Suchstring: \<0(\dx*h)\>und den Ersetzungsstring \1 verwenden. Unter Umständen ist auch \<0+(\d\x*h)\>geeignet, wenn alle führenden Nullen gelöscht werden sollen. Auf den Wert 0h passt der Ausdruck übrigens nicht, wegen \d am Anfang der Klammer.

Manche Dialekte erlauben sogar, auf einen bereits gefundenen Teilstring zuzugreifen, um (h) nochmal im Text zu suchen. Dort kann man mit (a+)b+\1 sogar das Muster an bm an suchen was über die „eigentlichen“regulären Ausdrücke hinausgeht. In manchen Dialekten ist der Ausdruck auch gleichbedeutend mit a+ b+ a+ weil dort eine Referenz im Suchstring schon vor dem Suchen im Text ersetzen wird. Welchen Dialekt man verwendet, muss man nachlesen oder ausprobieren.

2.4 Worauf passt ein RegEx?

Das erste passende Textstück wird verwendet, wenn der Ausdruck weiter hinten im Text nochmal passt, spielt das keine Rolle mehr. Für Programme wie “grep“ist das egal. Wenn das Muster in der Zeile vorkommt, wird Sie ausgegeben, egal wo es dort vorkommt. Beim Ersetzen von Text ist es aber ein Unterschied, wenn das Muster mehrfach vorkommt und genau das erste Vorkommen ersetzt wird.

Die normalen Quantifizierer sind “gierig“(engl.: greedy). Es wird also immer so viel wie möglich von der Zeichenkette verwendet. Der Ausdruck \d+ wird bei der Zeichenkette 123 auf alle Ziffern passen. \d* wird alle Zeichen der Zeile ab der ersten Ziffer erkennen. Diese Gier führt dazu, dass Ausdrücke wie „.*“ nicht geeignet sind, um Text in Ausführungszeichen zu erkennen. Es kann mehr als einmal Text in „“in einer Zeile geben und der Ausdruck passt auf alles vom ersten bis zum letzten „“. Für die üblichen Zeichenketten im C-Stil kann man den Ausdruck „(\\.|[^\\“])*“verwenden. Eine Ausnahme bei der der Ausdruck nicht passt, wird in der Übung behandelt.

Text zu erkennen hat aber eine höhere Priorität als Gier. Teilausdrücke verzichten auf Text, den sie erkennen können, wenn das zu einer Übereinstimmung

führt. Ein Bsp. Für so etwas ist der Ausdruck (.*)a: Wenn er passt, wird \1 alle Zeichen bis vor das letzte a der Zeile enthalten.

Welcher Text wird von dem geklammerten Ausdruck erkannt und in \1 gespeichert, wenn:

ˆ.*([0-9]+) auf den Text “Copyright 2003.“Ausgeführt wird?

-3: ˆ.* nimmt alles ab dem Zeilenanfang und gibt so lange Zeichen zurück, bis auch der Rest des Ausdrucks passt.

Es gibt auch die Möglichkeit zu sagen, dass ich möglichst wenige Zeichen möchte:

,\{-}.?* Gibt es in allen Dialekten.

2.5 Kontextabhängige Muster:

Manche Dialekte unterstützen lookaround, mit dem ein Muster nur passt, wenn auch passt, was davor oder danach steht. Der Kontext wird zwar erkannt, ist aber nicht Teil des passenden Textes. Das Prinzip kann man recht gut mit den leichteren VIM-Mustern \ZS für start-of-match und \ZE für end-of-match verdeutlichen; dazwischen steht der eigentliche Suchstring, der nur erkannt wird, wenn auch der Kontext passt.

Will man „La“vor „Tex“finden, so ist La\ZE Tex ein RegEx im VIM-Notation dafür. Für das „T“in LaTeX ist La\ZS T\ZE e* geeignet. Da hier keine Wortgrenzen angegeben sind, passt das Muster aber auch im Text wie „abclaTexdef“.

A(b(c*))?d	abcd /1: bc	/2: c
	ad /1: (leer) ∈	
	/2: ∈	
	abd /1: b	/2: ∈

2.6 Verwendung regulärer Ausdrücke:

Reguläre Ausdrücke werden alltäglich für Textersetzung mit einem Editor benutzt. Sie werden auch beim parsen verwendet, u.A. beim Compilieren, beim Syntax Highlighting und zum Prüfen von Eingaben auf formale Korrektheit z.B.: auf die Gültigkeit von Email-Adressen oder URIs. Manche Programmiersprachen haben eingebaute Unterstützung für reguläre Ausdrücke, zu ihnen gehören AWK, SED, Perl. Für viele andere Sprachen gibt es Bibliotheken mit denen man aus RegEx Objekte erzeugen kann, die mit Text verglichen werden und Ersetzungen darin durchführen können. Beispiele für solche Sprachen sind: C/C++, C#, Java und PHP.

3 Einige Unix Programme im Überblick

3.1 grep

Dient zum Suchen von Text in Dateien. Es verwendet einen regulären Ausdruck als Suchmuster und gibt gewöhnlich die Zeilen aus, in denen das Muster passt. Das ist bei verschiedenen Anwendungsfällen nützlich. Zwar ist das Suchen innerhalb einer Datei im Editor meist einfacher, manchmal weiß man aber gar nicht, in welcher Datei die gesuchte Stelle vorkommt, oder man will die entsprechenden Zeilen weiterverarbeiten. Dann kann man mit grep in z.B. allen Quelltext Dateien suchen und ggf. die Dateinamen (mit) ausgeben lassen. Der Name grep kommt vom UI Kommando „g“(global) mit der Anweisung „print“(p), zusammen: g re p. re steht dabei für den regulären Ausdruck.

Der allgemeine Aufruf lautet:

grep [option] ... pattern [FILE] ...

grep unterstützt ettliche Optionen. Die folgenden werden häufig verwendet:

-e Als nächstes folgt das Suchmuster. Diese Option ist besonders dann hilfreich, wenn das Muster mit einem Strich anfängt, damit grep es nicht als Option interpretiert

-n Die Nummern der ausgegebenen Zeilen (und ein Doppelpunkt) wird den Zeilen vorangestellt

-v Es werden die Zeilen ausgegeben, in denen das Muster nicht vorkommt

-i ignore case

-H/-h Dateinamen werden bzw. werden nicht ausgegeben. Bei mehreren Dateien als Argumente wird der Dateiname normalerweise dem Zeileninhalt (und der optionalen Zeilennummer) vorangestellt; wenn keine oder nur eine Datei als Argument angegeben ist, wird kein Name ausgegeben. Dieses Verhalten lässt sich ändern, um Zeilen ohne Dateinamen zu erhalten, oder um bei der Benutzung mit xargs bei nur einer Datei auch den Namen zu erhalten

-l Für jede durchsuchte Datei, in der das Muster vorkommt, wird nur der Dateiname ausgegeben und die Suche in dieser Datei beendet (weil bereits feststeht, dass das Muster vorkommt)

-q (quiet) Es wird nichts ausgegeben, sondern nur der exitcode verwendet: 0 heißt Muster gefunden, 1 heißt nicht gefunden und 2 heißt Fehler

-c Statt der gefundenen Zeilen wird ausgegeben, wie viele Zeilen der einzelnen Dateien das Muster enthalten.

-B/-A/-C Oft will man nicht nur die gefundenen Zeilen, sondern auch die Zeilen danach sehen (wie bei svn diff). Das lässt sich mit den Optionen -B (before), -A (after), -C (Context, davor und danach) einstellen. Wenn Zeilen zwischen den Ausgaben fehlen, wird – als Trennzeichen eingesetzt

-r sucht rekursiv in Verzeichnissen

- -color Diese Option kann mit „=“noch folgen, wann Farben verwendet werden sollen. never, always und auto. Im letzten Fall entscheidet der Typ des Ausgabedeskriptors, ob Farben verwendet werden sollen. Wenn die Ausgabe auf ein Terminal kommt, werden Farben benutzt; bei Umleitungen in eine Datei oder Pipe nicht. Oft ist grep ein Alias für grep –color=auto

Beim Aufruf von grep von der Shell aus ist es ratsam, den regulären Ausdruck in einfache Anführungszeichen einzuschließen, damit die Shell Zeichen wie * und |nicht selbst interpretiert.
Beispielausgabe für ein „grep main main.c“

* main.c -SPL compiler
int main(int argc, char *argv[ ]){

3.2 cmp

Cmp vergleicht Dateien byteweise und gibt Offset und Zeilennummer des ersten Unterschieds aus, wenn es einen gibt.

Der allgemeine Aufruf lautet:

cmp [option] ... File 1 [File2[Skip1[Skip2]]]

Nützliche Optionen sind:

-b Die Bytewerte des ersten Unterschieds werden mit ausgegeben

-i Die folgende Zahl gibt an, wie viele Bytes der ersten Datei beim Vergleich übersprungen werden sollen; eine mit „:“abgetrennte Zahl gibt Offset der zweiten Datei an (wie bei den Skip Argumenten). Ein Suffix gibt entsprechende Faktoren an, z.B. kB für 1000, K für 1024, MB für 1.000.000, M für 1.048.576 usw. für G, T, P, E, Z, Y

-l Es wird nicht nur der Bytewert des ersten, sondern von jedem Unterschied angezeigt. Bei verschieden großen Dateien wird nur bis zum Ende der kleineren Datei verglichen

-n Die der Option folgende Zahl gibt an, wie viele Bytes verglichen werden sollen

-s Es gibt keine Ausgabe. Der Exitcode zeigt die Gleichheit an. 0 heißt gleich, 1 verschieden, größer oder gleich 2 Fehler

Beispielausgabe (erzeugt durch Vergleich von zwei Versionen der Datei Main.c die Erste von der Standardeingabe; Aufruf: cmp -main.c):

- - main.c differ: char 1504, line 59

3.3 diff

diff vergleicht zwei Dateien Zeilenweise und gibt die unterschiedlichen Zeilen aus. Mittels der Präfixe <und >kann man zwischen den Zeilen der linken und rechten Datei unterscheiden.

Der allgemeine Aufruf lautet:

diff [option] ... Files

Häufig verwendete Optionen sind:

- -i Groß-/und Kleinschreibung wird beim Vergleich ignoriert
- -w Whitespaces werden beim Vergleich ignoriert
- -c Bei Unterschieden wird die angegebene Anzahl von Kontextzeilen mit ausgegeben
- -u Bei Unterschieden wird die angegebene Anzahl von Kontextzeilen im unified Format mit ausgegeben
- -p diff gibt die C-Funktion an, in der sich ein Unterschied befindet
- -e Erzeugt ein ed-Skript, mit dem ed die linke in die rechte Datei umwandeln kann
- -y Gibt die Dateien in zwei Spalten aus, so dass man leichter erkennen kann, wie die beiden Dateien aussehen
- -r Vergleicht zwei Verzeichnisse rekursiv (also jeweils gleichnamige Dateien und Verzeichnisse) und gibt die Unterschiede aus

Die Ausgabe des diff-Kommandos einer Versionsverwaltung wird möglicherweise durch das Programm diff erstellt.
Beispielausgabe bei zwei ähnlichen Quelltext Dateien (erzeugt durch Vergleich von zwei Versionen der Datei main.c, die erste von der Standardeingabe, Aufruf: diff - - main.c):

58 u 59[1]
>boolean optionNoStackClech

[1] Diese Angaben ermöglichen es aus der einen Datei eine andere zu machen (patchen)

72 u 74
>optionNoStackClech = false

100a 103, 105
>if(strcmp(argv[i], "- -nostackclech)==0){
>optionNoStackClech = TRUE;
>}else

152, 157
<gencode(progTree, globalTable, contFile, optionNoSpilling, optionCodeComments);

- - -
>
gencode(...)

3.4 patch

Mit patch kann man die per diff ermittelten Unterschiede zwischen zwei Dateien verwenden, um aus der erstem Datei die zweite zu machen.
Der übliche Anwendungsfall ist:
Man hat eine Datei (z.B. Quelltext) und nimmt Änderungen daran vor. Durch ein diff auf die alte und die neue Fassung (in dieser Reihenfolge) ermittelt man die Unterschieden zwischen den Versionen. Diese Unterschiede kann man jemandem zur Verfügung stellen, der ebenfalls die alte Version der Datei besitzt. Mit patch kann er daraus die neue machen.

Der allgemeine Aufruf lautet.

patch [option] ... [origfile[patchfile]]

Nützliche Optionen sind:

-c Das Patchfile enthält Kontextinformationen

-e Das Patchfile ist ein ed-Skript

-n Das Patchfile liegt im unified-Format vor

3.5 sort

Sort sortiert Dateien zeilenweise und schiebt das Ergebnis auf stdOut. Die übliche Sortierung erfolgt nach dem Bytewert der Zeichen; dass ist aber abhängig von der Spracheinstellung mit LC_ALL=C bekommt man diese "normale Verhalten".

Der allgemeine Aufruf lautet:

sort [option] ... [file] ...

Häufig verwendete Optionen sind :

-r Die Sortierung erfolgt absteigend

-s Es wird ein stabiler Sortieralgorithmus verwendet

-b Führende Leerzeichen werde ignoriert

-i Nicht druckbare Zeichen werde ignoriert

-f Groß-/Kleinschreibung wird beim Vergleich ignoriert

-c Es wird nur geprüft ob die Dateien sortiert sind

-C Es wird nur geprüft ob die Dateien sortiert sind aber es wird nichts ausgegeben

M Es wird nach Monatsnamen sortiert (Schreibweise abhängig von LC_TIME)

n Es wird nach Zahlenwert sortiert

k Die angegebene Zahl gibt die erste Position des angegebenen Sortierschlüssels an; eine mit Komma abgetrennte zweite Zahl gibt die letzte Spalte des Schlüssels an.

m Die Eingabedateien werden Zeilenweise einsortiert, es findet keine neue Sortierung statt. Die Daten sollten bereits sortiert sein. Doppelte Zeilen werden entfernt. Mit -c auf strenge Sortierung prüfen.

Der Exitcode 0 bedeutet Erfolg, 1 bei -c oder -C, dass die Eingabe nicht sortiert war, 2 Fehler.

3.6 uniq

Uniq findet doppelte Zeilen in der Eingabe und gibt sie aus oder entfernt sie (wobei "doppelt"hier immer bedeutet, dass die gleiche Zeile mehr als einmal direkt hintereinander kommt; es wird nie sortiert).
sort — uniq

Der allgemeine Aufruf lautet:

uniq [option] ... [input[option]]

Häufig verwendete Optionen sind:

-c Gibt vor jeder Zeile aus wie häufig sie vorkommt

-d Es werden nur Zeichen ausgegeben die mehr als einmal vorkommen

-f Die angegebene Anzahl Felder wird beim Vergleich übersprungen

-i Groß-/Kleinschreibung wird beim Vergleich ignoriert

-s Die angegebene Anzahl Zeichen wird beim Vergleich ignoriert

-n Es werden nur Zeilen ausgegeben, die nur einmal vorkommen

Der Exitcode 0 heißt Erfolg; alles andere bedeutet Fehler.

3.7 xargs

Xargs liest Zeichenketten von der Standardeingabe und gibt sie als Argumente an das angegebene Programm weiter. Als Trennzeichen zwischen den Zeichenketten dienen Leerzeichen und Zeilenumbrüche (wenn sie nicht durch Anführungszeichen und Backslash markiert sind). In vielen Fällen besteht die Eingabe aus Namen von Dateien, die von einem weiteren Programm verarbeitet werden sollen.

Der allgemeine Aufruf lautet:

xargs [option] ... [command[[initial_arguments]...]

Nützliche Optionen sind:

-a Die Eingabe wird aus der angegebenen Datei gelesen und die Standardeingabe für das erste Programm wird nicht nach /dev/null umgeleitet.

-0 Die Eingabestrings werden von binären Nullen statt von whitespace getrennt; alle Zeichen stehen für sich selbst.

-n Es wird höchstens die angegebene Anzahl von Argumenten pro Programmaufruf übergeben.

-p xargs zeigt jeden Programmaufruf und fragt ob es ihn durchführen soll

-r Es wird höchstens die angegebene Anzahl Zeichen pro Aufruf verwendet (inkl. Programmnamen, übergebene Argumente und trennende binäre Nullen)

-p Es wird maximal die angegebene Anzahl an Programmen gleichzeitig gestartet; default ist 1, 0 bedeutet auf Probleme hin; weitere Infos siehe Manpage.

Beispiel für die Verwendung:

grep -l MUSTER1 * |xargs -r grep -l MUSTER2 → Findet Dateien im aktuellen Verzeichnis, die sowohl MUSTER1 als auch MUSTER2 enthalten.

cut -d: -f 1,2 |</etc/passwd |sort |xargs echo → Gibt die Benutzer aus /etc/passwd sortiert in einer Zeile aus.

root: ... root hg10597
hg10597 ... hg10597 nobody
nobody ... nobody root
Ausgabe: (echo) hg10597 nobody root

3.8 find

find sucht Dateien mit bestimmten Eigenschaften und führt Operationen darauf aus; normalerweise wird der Dateiname ausgegeben. Die Suchkriterien sind vielseitig und kombinierbar und die Möglichkeit, Programme mit den gefundenen Dateien aufzurufen oder die Liste der Dateien weiterzuleiten, machen find sehr mächtig.
Die Anweisungen für find beginnen mit einem einzelne Strich, auch wenn sie aus mehreren Buchstaben bestehen.

Der allgemeine Aufruf lautet:

find [-L] [path...] [expression]
Normalerweise wird symbolischen Links nicht gefolgt; mit der Option -L lässt sich das ändern.

Die Ausdrücke sind Kombinationen von Operatoren, Optionen, Tests und Aktionen. Operatoren sind () für Prioritäten, ! für Negation, Verkettung und -a für Und-Verknüpfung, -o für Oder-Verknüpfung und ”,”für bedingungslose sequenzielle Ausführung.

Beispiele für benutzte Befehle sind:
-name, -iname:
Der Name entspricht dem angegebenen Shellmuster

-regex, iregex:
Der komplette relative Pfad entspricht den angegebenen regex.

-print, print0, printf, fprintf, ls, -fls:
Der Dateiname oder andere Informationen zur Datei und ein Zeilenumbruch bzw. eine binäre Null werden auf die Standardeingabe bzw. in die angegebene Datei geschrieben.

-usr, -group:
Die Datei gehört den angegebenen Benutzer bzw. der angegebenen Gruppe.

-innm, -samefile:
Die Datei hat die angegebene Inodenummer bzw. ist die selbe Datei wie die angebene (mit -L auch bei symbolischen Links).

-links:
Die Datei hat die angegebene Anzahl von Hardlinks.

-mtime, -mmin, -atime, -amin, -ctime, -cmin:
Die Datei bzw. ihr Inode wurde vor n Tagen gelesen bzw. geändert.

-daystart:
Die Prüfung auf eine Zeit bezieht sich auf das Datum, nicht auf n*24 Stunden.

-newer, -anewer, cnewer:
Die Datei ist später als die angegebene Datei geändert, gelesen bzw. ihr Status geändert worden. Es gibt keine Old-Option , aber Invertierung mit ! ist mgl.

-perm:
Die Zugriffsrechte entsprechen den angegebenen. Symbolische Darstellung ist mgl. Rechte mit - sind mindestens diese, mit / irgendeins von diesen.

-readable, -writeable, -executable:
Die Datei ist für die suchenden Benutzer les-, schreib- bzw. ausführbar. Dieser Test berücksichtig Access-Control-Lists, kann aber Probleme beim UID-Mapping von NFS-Servern haben.

-size:
Die Datei ist n Einheiten groß. Die Einheiten b (512 Byte Blöcke; default), c(Bytes), w(zwei Byte Worte), k, M, G (Kilo, Mega und Gigabyte) sind erlaubt.

-type, -xtype:
Die Datei hat den angegebenen Typ, erlaubt sind b (block special), c (character special), d (Verzeichnis), p (named pipe, fifo), f (normale Datei), l (symbolischer Link) und s (socket).

-maxdepth:
find wird nicht über die angegebene Suchtiefe hinaus im Verzeichnis absteigen, 0 bedeutet, es werden nur die Argumente geprüft.

-exec,-execdir,-ok,-okdir:
Für jede gefundene Datei wird das angegebene Programm gestartet (das möglicherweise selbst eine Prüfung durchführt) Dabei steht {} für den Dateiname, den man dem Programm meist übergeben wird. Das Kommando wird mit einem Semikolon abgeschlossen, welches für die Shell maskiert werden muss. Bei den dir-Varianten wird das Programm in dem Verzeichnis ausgeführt, in dem Die Datei gefunden wurde. Bei den ok-Varianten wird vor jedem Aufruf gefragt, ob er durchgeführt werden soll.

-depth:
Der Inhalt von Verzeichnissen wird vor dem Verzeichniseintrag behandelt

-prune:
Verzeichnisse werden nicht rekursiv behandelt. Ein anderer Exitcode als 0 weißt auf Fehler hin.

Beispiel:
find . -iname "*.mp3"
⇒ Listet Dateien mit der Erweiterung .mp3 unterhalb des aktuellen Verzeichnisses auf.

find /tmp -name core -type f -print |xargs /bin/rm -f
⇒ löscht coredumps im Tempverzeichnis

find /tmp -name core -type f -print0 |xargs -0 /bin/rm -f

find . -type f -exec f -exec file '' \;
⇒ Ruft das Programm file für jede normale Datei unterhalb des aktuellen Verzeichnisses auf.

find / \(-perm -4000 -fprintf /root/suid.txt "%#m %n %p \n"\) -o \(-size +100M -fpintf /root/big.txt "%-105 %p\n"\)
⇒ Schreibt Informationen über Dateien mit gesetztem suid-Bit bzw. größer als 100MB in die entsprechenden Dateien. Die Informationen sind oktale Zugriffsrechte, Eigentümer und relativer Dateiname bzw. Dateigröße und relativer Dateiname.

find $HOME -mtime 0
⇒ Sucht Dateien im Homedir, die in den letzten 24 Stunden geändert wurden.

find /sbin /usr/sbin -executable \! -readable -print
⇒ Listet Dateien auf, die ausführbar nicht lesbar sind.

find . -perm 664
⇒ Sucht Dateien mit der (oktalen) Zugriffsmaske 664

find . -perm -664
⇒ Sucht Dateien mit mindestens den Zugriffsrechten 664

find . -perm /222
⇒ Sucht Dateien die für irgend jemanden schreibbar sind.

cd /source-dir; find . -name .snapshot -prune -o \(\! -name "* print0 \) |cpio -pmd 0 /dest-dir
⇒ Kopiert den Inhalt von source-dir nach /dest-dir, lässt dabei aber Dateien und Verzeichnisse, die .snapshot heißen und Dateien (und Verzeichnisse, aber nicht deren Inhalt), die auf Tilde enden, aus. Die Klammern dienen nur der

Klarheit, sind aber nicht nötig.

find /repo -exec test -d /.svn -o -d /.git -o -d /cvs \; -print -prune
⇒ Listet Verzeichnisse unterhalb von repo/ auf, die Versionsverwaltungsinformationen enthalten; die gefunden Verzeichnisse werden nicht weiter Untersucht.

find /foo -maxdepth 1 -atime +366 -print0 |xargs -r0 sh -c "mv$@"/archive' move
⇒ Findet Einträge im Verzeichnis /foo, die länger als 1 Jahr nicht gelesen wurden und verschiebt sie ins Verzeichnis /archive. "move"wird zu Argument 0 des Aufrufs; ohne es würde die erste Datei nicht verschoben.

find /usr/include -name "*.h"|xargs grep -wl mode_t |xargs -r sh -c 'exec emacs "$@"</dev/tty' Emacs
⇒ Findet Headerdateien, die den Bezeichner mode_t enthalten und öffnet sie mit emacs. Ohne Eingabeumleitung würde die Eingabe aus /dev/null gelesen.

3.9 awk

awk ist eine Programmiersprache, deren Stärke Texterkennung und spaltenweise Verarbeitung sind. Inzwischen wurde awk weitgehend von Perl verdrängt; für viele Anwendungen eignet es sich aber noch sehr gut. Der Name awk besteht aus den Anfangsbuchstaben der Nachnamen der Entwickler: Aho, Weinberger

Kurzbeispiele für die Verwendung:
ls -l |awk ' sum += $5 END print sum
⇒ gibt die Gesamtgröße aller nicht (nicht versteckten) Dateien im Verzeichnis aus.

awk 'BEGIN {FS=":"}{print $1 |ßort"}' c /etc/passwd /* FS = Field separator */
⇒ gibt eine sortierte Liste der Loginnamen aller Benutzer aus.

awk 'nlines ++ ENDprint nlines' file
⇒ Zählt die Zeilen der Datei file

awk '/^ a/ nlines++ END print nlines'
⇒ Zählt die Zeilen der Eingabe, die mit a anfangen

awk '/^ a/ alines++ /^ b/ blines++ END print alines blines'
⇒ Zählt die Zeilen der Eingabe, die mit a und b anfangen.

awk 'print FNR, $0' file
⇒ Gibt vor jeder Zeile ihre Zeilennummer aus.