

Reguläre Ausdrücke: Beschreibung und Anwendung

Version 1.38 — 9.7.2009

© 2003–2009 T. Birnthaler, OSTC GmbH

Die Informationen in diesem Skript wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Der Autor übernimmt keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen.

Alle Rechte vorbehalten einschließlich Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Für Dokumente und Programme unter dem Copyright der OSTC GmbH gilt:

- Dürfen heruntergeladen und im privaten Bereich frei verwendet werden.
- Kommerzielle Nutzung bedarf der vorherigen Zustimmung durch die OSTC GmbH.
- Titelseite und Copyright-Hinweise darin dürfen nicht verändert werden.

Hinweise auf inhaltliche Fehler, Schreibfehler und unklare Formulierungen sowie Ergänzungen, Kommentare, Wünsche und Fragen können Sie gerne an den Autor richten:

OSTC Open Source Training and Consulting GmbH
Thomas Birnthaler
eMail: tb@ostc.de
Web: www.ostc.de

Inhaltsverzeichnis

1	Einleitung	3
1.1	Vier einfache Beispiele	3
1.2	Begriffe	4
1.3	Hinweise	5
1.4	Besonderheiten	5
1.5	Literatur	7
2	Beschreibung	7
2.1	Metazeichen der Shell	7
2.1.1	Beispiele zur Dateisuche	9
2.2	Metazeichen zur Textsuche	9
2.2.1	Unterschiede zur Shell	10
2.2.2	Standard Metazeichen (BRE)	11
2.2.3	Erweiterte Metazeichen (ERE)	11
2.2.4	Ersatzdarstellung	12
2.2.5	Vorrang	12
2.2.6	Hinweise	12
2.2.7	Besonderheiten von <i>grep</i>	13
2.2.8	Escape-Sequenzen von <i>gawk</i> und <i>perl</i>	14
2.2.9	POSIX-Zeichenklassen von <i>gawk</i> und <i>perl</i>	14
2.2.10	Besonderheiten von <i>perl</i> (PCRE)	15
2.3	Beispiele zu Suchmustern	16
2.3.1	Großes Beispiel A: Gleitkommazahlen erkennen (Awk-Syntax)	17
2.3.2	Großes Beispiel B: Datumswerte erkennen (Perl-Syntax)	18
2.4	Metazeichen im Ersetzungsteil	18
2.4.1	Hinweise	18
2.4.2	Beispiele zu Such- und Ersetzungsmustern	19
3	Anwendung	19
3.1	Quotierung	19
3.2	Suchen anwenden	19
3.3	Suchen und Ersetzen anwenden	20
4	Weitere Beispiele zu Suchmustern	21
5	Kurzübersicht	22
6	ASCII-Tabelle	27

1 Einleitung

Reguläre Ausdrücke (**Textmuster** oder **Patterns**) stellen eine **einfache Programmiersprache** dar, die zum Beschreiben von Dateinamen oder Zeichenketten (Textzeilen) dient. Sie werden verwendet in:

- **Kommando-Shells** zur Auswahl von Dateinamen: *sh, t(c)sh, ksh, bash*.
- **Texteditoren** zum Suchen (und Ersetzen) von Zeichenketten: *ed, ex, vi, emacs, ultraedit, word*
- Programmen mit **eingebauter Zeichenkettenverarbeitung** ebenfalls zum Suchen (und Ersetzen) von Zeichenketten: *(e/f)grep, gres, expr, find, more, pg, less, sed, (g/n/m)awk, perl, csplit, (f)lex, tcl/tk, python, ruby, icon, lua, php, mysql*.
- **Server-Konfigurationsdateien** zum Matchen von URLs, Mailadressen, Dateinamen, Fehlermeldungen: *apache, postfix, sendmail, syslog, syslog-ng*.

Mit ihrer Hilfe können auf einfache Weise entweder **Dateinamen** einer bestimmten Form in Verzeichnissen oder **Zeilen** mit einem bestimmten Inhalt in (ASCII-)Textdateien ermittelt werden. Zusätzlich können im zweiten Anwendungsfall die Zeichenketten auch verändert, gelöscht oder durch andere Zeichenketten ersetzt werden.

1.1 Vier einfache Beispiele

In den vier folgenden Kommandobeispielen werden Reguläre Ausdrücke verwendet:

1. Alle Dateinamen mit der Endung `.c` oder `.h` im aktuellen Verzeichnis anzeigen (`-d [directory only]`):

```
ls -d *.*[ch]
echo *.*[ch]
```

2. Alle nicht leeren Zeilen (Leerzeilen sind entweder völlig leer oder enthalten nur Leerzeichen) der Dateien `*.txt` zählen (`-v [vice versa]`, `-c [count]`):

```
grep -v '^ *$' *.txt | wc -l      oder
grep -vc '^ *$' *.txt           oder
grep -c '^[^ ]' *.txt
```

3. Alle Vorkommen von `Unix` oder `unix` in der Datei `kap1.txt` durch `UNIX` ersetzen und das Ergebnis in `kap1.new` abspeichern (`p [print]`, `e [execute]`, `s [substitute]`):

```
sed 's/[Uu]nix/UNIX/g' kap1.txt > kap1.new
awk '{ gsub(/[Uu]nix/, "UNIX"); print }' kap1.txt > kap1.new
perl -pe 's/[Uu]nix/UNIX/' kap1.txt > kap1.new
```

4. Alle Leerzeilen und Leerräume am Zeilenende direkt in der Datei `kap1.txt` entfernen (n [**noprint**], i [**inline**], e [**execute**], s [**substitute**], \s [**space**], \S [**nospace**]):

```
perl -nie 's/\s*$/\n/; print if !/^\\s*$/' kap1.txt
perl -nie 's/\s*$/\n/; print if /\S/' kap1.txt
```

1.2 Begriffe

Folgende Begriffe sind im Zusammenhang mit Regulären Ausdrücken gebräuchlich:

Begriff	Beschreibung
Backreference	Verweis auf vorher gematchten Musterteil später im Muster.
Backtracking	Zurücknahme bereits gemachter Textteile, weil kein Gesamtmatch mehr möglich (evtl. für lange Laufzeiten bei ungeschickt formulierten Regulären Ausdrücken verantwortlich).
BRE	Basic Regular Expressions , überall vorhanden und mit einheitlicher Syntax.
ERE	Extended Regular Expressions , teilweise nicht oder nur eingeschränkt vorhanden und mit unterschiedlicher Syntax.
Escape-Sequenz	Sonderzeichen gebildet durch Backslash + normalen Buchstaben (z.B. \n = Newline, \r = Carriage Return, \t = Tabulator).
File Globbing	Ersetzen eines Regulären Ausdruck durch alle passenden Dateinamen in einem Shell-Kommando (Dateinamen-Expansion).
Greedy	Algorithmus matcht möglichst viele Zeichen („ Gierig “).
Konkatenation	Verkettung von Zeichen. Dies ist die <i>natürliche Operation</i> mit den Zeichen einer Zeichenkette. Für sie gibt es daher keinen expliziten Operator in den Regulären Ausdrücken, sondern sie wird durch bloßes <i>Hintereinanderschreiben</i> ausgedrückt.
Left-most	Ein Algorithmus, der von links nach rechts arbeitet.
Literal	Ein Zeichen, das für sich selbst steht (Zeichen-Konstante).
Matchen	In einer Zeichenkette eine zu einem Regulären Ausdruck passende Teilkette suchen (treffen, passen).
Metazeichen	Stehen nicht für sich selbst, sondern für eine Operation oder eine Menge von Zeichen (Wildcard , Jokerzeichen).
Pattern	Ein konkreter Regulärer Ausdruck bestehend aus Literalen und Metazeichen ((Text)Muster).
Pattern Matching	Vergleichen eines Regulären Ausdrucks mit einer Zeichenkette (Mustererkennung).
PCRE	Perl Compatible Regular Expressions sind eine wesentliche Erweiterung der Möglichkeiten von (Extended Regular Expressions) ERE im Rahmen der Programmiersprache Perl. Wurden in viele andere Programmiersprachen übernommen.
Quantifier	Wiederholungsoperatoren, die Ausdruck <i>n</i> Mal wiederholen.
Quoten	Zeichenkette vor Interpretation durch Shell schützen oder Metazeichen in Literale umwandeln (Quotieren/Zitieren/Schützen).
... matcht...	... passt auf...
Verankern	Zeilenanfang/-ende oder Wortanfang/-ende festlegen (Anchor).
Whitespace	Leerraum , d.h. Leerzeichen, Tabulator (horizontal + vertikal), Zeilenvorschub, Wagenrücklauf.
Wildcard	Metazeichen der Shell, die für eine Menge von Zeichen stehen (Jokerzeichen).

1.3 Hinweise

- Die Vergleiche von Regulären Ausdrücken mit Dateinamen oder Texten sind maschinell sehr effizient realisierbar durch **Deterministische Zustandsautomaten** (DFA = Deterministic Finite Automaton) und **Nichtdeterministische Zustandsautomaten** (NFA = Non Deterministic Finite Automaton). Es ist also kaum sinnvoll, derartige Aufgaben von Hand zu programmieren.
- Hat man sich einmal an den Aufbau und die Anwendung von Regulären Ausdrücken gewöhnt, so lassen sie sich sehr **schnell und fehlerfrei** erstellen. Da sie innerhalb des UNIX-Systems an sehr vielen Stellen anwendbar sind, *lohnt sich eine Einarbeitung in sie auf jeden Fall*. Viele tägliche Probleme der Daten- und Textverarbeitung lassen sich mit ihrer Hilfe elegant, effizient und vor allem fehlerfrei durchführen.
- Das **Lesen und Verstehen** komplizierter Regulärer Ausdrücke ohne jeglichen Kommentar kann allerdings schwierig sein. D.h. Reguläre Ausdrücke können überspitzt formuliert auch als **Write-Only-Programmiersprache** bezeichnet werden (sie sind allerdings nicht die einzige Programmiersprache, die man so nennen könnte ; -). Es ist daher sehr wichtig, Reguläre Ausdrücke in Skripten gut zu dokumentieren bzw. zu kommentieren.
- Da sie **historisch/pragmatisch entstanden** sind, gibt es unterschiedliche Formen und Erweiterungen, dies kann gelegentlich zu Verwirrung führen. Solche pragmatischen Dinge/Verhaltensweisen sind in den folgenden Abschnitten jeweils durch **[P]** gekennzeichnet.
- Die **Wildcards** oder **Jokerzeichen** ? und * unter Windows stellen eine *sehr eingeschränkte Form* von Regulären Ausdrücken dar. Das ? steht für ein beliebiges Zeichen, der * für beliebig viele beliebige Zeichen in einem Dateibezeichner. Folgende Einschränkungen sind hier vorhanden:
 - ▷ Der Punkt zwischen Name und Extension kann durch ? und * nicht gematcht werden.
 - ▷ Verschachtelte Verzeichnisse können nicht gematcht werden.
 - ▷ * kann nicht mehrfach im Namen oder der Extension angegeben werden.
- Reguläre Ausdrücke sind typischerweise in Skriptsprachen wie *php*, *perl*, *awk* direkt eingebaut. Aber auch in Compilersprachen wie C, C++ sind sie über **Funktionsbibliotheken** wie *regex* oder *regexp* einsetzbar.

1.4 Besonderheiten

Folgende Besonderheiten sind bei der Anwendung Regulärer Ausdrücke zu beachten:

- In manchen Programmen wird das Zeichen **Schrägstrich** / als Begrenzungszeichen von Regulären Ausdrücken eingesetzt (z.B. *sed*, *awk*, *perl*, *vi*), in anderen nicht (z.B. *grep*, *egrep*). Bei *php* ist es noch etwas komplizierter, da diese Skriptsprache beide

Varianten ERE und PCRE unterstützt und den Regulären Ausdruck immer in Form eines Strings `"..."` oder `'...'` erwartet (bei PCRE zusätzlich noch / außenrum).

```
REGEX                # grep, egrep
/REGEX/              # sed, awk, perl, vi
"REGEX"              # perl (alternativ)
ereg('REGEX', ...)  # php (ERE, Funktionsaufruf)
preg_match('/REGEX/', ...) # php (PCRE, Funktionsaufruf)
```

- **Jedes Zeichen** in einem Regulären Ausdruck — also auch Leerzeichen und Tabulatoren — hat eine Bedeutung und wird beim Vergleichen berücksichtigt (kann in *perl* geändert werden).
- Grundsätzlich wird der *erste* zu einem Regulären Ausdruck passende Teil einer Zeichenkette gefunden. Dieses Verhalten wird als **left-most** (von links nach rechts) bezeichnet (kann in *perl* geändert werden). Beispiel:

```
MMMM passt auf xxMMMMxxxMMMMx
<-->          1. passender Teil, wird gematcht
<-->          2. passender Teil, nicht gematcht
```

- Grundsätzlich wird der *längste* zu einem Regulären Ausdruck passende Teil einer Zeichenkette gefunden, nicht der kürzeste. Dieses Verhalten wird als **greedy** (gierig) bezeichnet (kann in *perl* geändert werden). Beispiel:

```
MMMM* passt auf xxMMMMMMxxx
<->           passender Teil, nicht gematcht
<-->          passender Teil, nicht gematcht
<--->         passender Teil, nicht gematcht
<---->        längster passender Teil, wird gematcht
```

- Ein **Zeilenvorschub** (newline) kann normalerweise nicht gematcht werden (außer durch *sed* und *perl*-Tricks).
- Reguläre Ausdrücke sind **keine vollständige Programmiersprache**, sie haben nur eine beschränkte Ausdruckskraft. Z.B. können mit ihnen *keine rekursiv geklammerten Ausdrücke* erkannt werden („verschachtelte Klammerngebirge“), da kein Zählen von Zeichen möglich ist. Ebenso kann in vielen Fällen die *Negation* eines Regulären Ausdrucks nicht als Regulärer Ausdruck angegeben werden. Die meisten Programme bieten aber hierzu ein **Kommando/Flag** `v/!` oder eine(n) **Schalter/Option** `-v [vice versa]` an, mit dessen Hilfe die Negation ausgewählt werden kann. Grund:

```
ab === a AND b          # bzw. "a gefolgt von b"
NOT ab === (NOT a) OR (NOT b) # bzw. "nicht a gefolgt von nicht b"
```

OR wird aber nicht in allen Arten von Regulären Ausdrücken unterstützt (erst mit dem Operator `|` in der erweiterten Form ERE bzw. PCRE ist dies möglich).

1.5 Literatur

- Daniel Gilly, *UNIX in a Nutshell*, 2. Edition, O'Reilly & Associates.
- Alfred Aho, Peter Weinberger, Brian Kernighan, *The AWK Programming Language*, Addison-Wesley.
- Dale Dougherty, *sed & awk*, 2. Edition, O'Reilly & Associates.
- Larry Wall, *Programming Perl*, 3. Edition, O'Reilly & Associates.
- Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly & Associates.
- Ulrich Cuber, *Linux Skripting*, Franzis Verlag.
- Gottfried Staubach, *UNIX-Werkzeuge zur Textmusterverarbeitung*, Springer.
- Stephan Thesing, *sed&awk GE-PACKT*, mitp.

2 Beschreibung

2.1 Metazeichen der Shell

Die Shell **expandiert** Ausdrücke mit Metazeichen *vor* dem Aufruf des angegebenen Programms oder Kommandos zu allen passenden Dateinamen. Dem Programm wird also nicht mehr das ursprüngliche Muster, sondern bereits die vollständig expandierte Liste übergeben. Soll das Expandieren verhindert werden, müssen Backslashes *vor* oder einfache/doppelte Anführungszeichen *um* die Argumente von Programmen und Kommandos gesetzt werden (**Quoten**):

Metaz.	Beschreibung
<code>\x</code>	Quotiert ein einzelnes Zeichen <i>x</i> mit Sonderbedeutung.
<code>"xyz"</code>	Quotiert alle Zeichen außer <code>\$ ` \</code> (und <code>!</code>).
<code>'xyz'</code>	Quotiert alle Zeichen (außer <code>!</code> und <code>'</code> selbst).

Achtung: Aus **pragmatischen Gründen** sind Reguläre Ausdrücke in der Shell grundsätzlich **verankert**, d.h. Anfang und Ende eines Regulären Ausdrucks müssen mit dem Anfang und Ende der passenden Dateinamen zusammenfallen. Lässt sich durch Verwenden von `*` am Musteranfang/-ende umgehen [**P**].

Die Metazeichen der Shell zur Dateinamenexpansion sind:

Metaz.	Beschreibung
?	1 beliebiges Zeichen
*	0 oder mehr beliebige Zeichen (<i>nicht Wiederholung eines Zeichens</i>) [P]
\x	Metazeichen <i>x</i> <i>quotieren</i> (\ \ steht für \ selbst!)
[abc], [a-z]	1 Zeichen aus Zeichenmenge (Zeichenklasse , [a-z] = Zeichen von a bis z)
[!abc], [!a-z]	1 Zeichen nicht aus Zeichenmenge (<i>sh, ksh, bash</i>) [P]
[^abc], [^a-z]	1 Zeichen nicht aus Zeichenmenge (<i>t(c)sh, bash</i>) [P]
{abc, def, ...}	Liste von Zeichenketten (<i>t(c)sh, ksh, bash</i>) [P]
~	Home-Verzeichnis des aktuellen Users (<i>t(c)sh, ksh, bash</i>)
~USER	Home-Verzeichnis des Users USER (<i>t(c)sh, ksh, bash</i>)

- In der Shell sind die Zeichen Leerzeichen, Tabulator und Zeilenvorschub (**Whitespace**) **Trenner** für die Argumente eines Programmaufrufes (wird durch die Shell-Variable `IFS = internal field separator` festgelegt).
- In allen Shells haben folgende Zeichen eine **Sonderbedeutung** (sie müssen daher durch Quoten geschützt werden, wenn sie Teil eines Dateinamens sein sollen):

```
# & * ? [ ] ( ) = | ^ ; < > ` $ " ' \ Leerzeichen Tabulator Return
```

- In der *t(c)sh, ksh, bash* haben zusätzlich folgende Zeichen eine **Sonderbedeutung** (sie müssen daher ebenfalls durch Quoten vor der Shell geschützt werden, wenn sie Teil eines Dateinamens sein sollen):

```
~ { } !
```

- In einer **Zeichenklasse** kann eine beliebige Folge von Einzelzeichen und Zeichenbereichen angegeben werden (ohne Leerraum oder Trennzeichen dazwischen). Ein Zeichenbereich deckt alle Zeichen zwischen den **ASCII-Codes** der beiden angegebenen Zeichen ab, das 1. Zeichen muss einen kleineren ASCII-Code haben als das 2. Zeichen. D.h. [9-0] ist nicht erlaubt und [A-z] und [A-Za-z] sind zwei verschiedene Zeichenbereiche, da die Zeichen z und a nicht direkt aufeinander folgen.
- Sollen die Zeichen -] in einer **Zeichenklasse** enthalten sein, so sind sie direkt *am Anfang* anzugeben, da sie sonst als Bereichsoperator bzw. Klassenende interpretiert werden.
- Das Zeichen / (**Verzeichnistrenner**) wird durch ? * nicht gematcht, es muss explizit angegeben werden [P].
- Es gibt Shells (z.B. *bash*) und Kommandos (z.B. *rsync*), die das Zeichen / (**Verzeichnistrenner**) bei Verwendung von ** statt * matchen.
- Das Zeichen . am Anfang eines Dateinamens (**versteckte Datei**) wird durch ? * nicht gematcht. Es muss explizit angegeben werden, wenn Dateinamen mit führendem Punkt gefunden werden sollen [P]. Alternativ können bei `ls` die Optionen `-a [all]` oder `-A [almost all]` angegeben werden, um alle Dateinamen (bis auf . und ..) aufzulisten.

- Das Muster `.*` zur Suche nach **versteckten Dateinamen** passt auch auf die in jedem Verzeichnis vorhandenen Verzeichnisnamen `.` (aktuelles Verzeichnis) und `..` (Elternverzeichnis). Sollen diese beiden nicht gefunden werden, ist das Muster `[!.]*` bzw. `[^.]*` zu verwenden.
- Argumentlisten sind von der **Gesamtlänge** her beschränkt (Shell-abhängig etwa 100.000 bis 2 Mio Zeichen), d.h. es ist möglich, dass die aus einem Muster generierte Liste von Dateinamen zu lang wird und nicht mehr von der Shell verarbeitet werden kann. Hier bietet das Kommando `xargs` eine Abhilfe (0 = Nul-Byte):

```
find / -type f -name "*.c" -print0 | xargs -0 rm
```

- Wird *kein einziger* passender Dateiname gefunden, so bleibt das Muster stehen. Dies führt in der Regel zu einer **Fehlermeldung** der Shell:

```
CMD: *.sh: No such file or directory
```

2.1.1 Beispiele zur Dateisuche

Folgende Muster listen bei der Angabe nach `ls` oder `echo` die jeweils beschriebenen Dateinamen auf (`_` steht für ein Leerzeichen). Bei Angabe der Option `-d [directory]` wird der Inhalt von Verzeichnissen von `ls` nicht aufgelistet, sondern nur das Verzeichnis selbst:

<code>*.c</code>	Dateinamen, die auf <code>.c</code> enden
<code>*.[ch]</code>	Dateinamen, die auf <code>.c</code> oder <code>.h</code> enden
<code>[!.][!ch]</code>	Dateinamen, die als vorletztes Zeichen nicht <code>.</code> und als letztes nicht <code>c</code> oder <code>h</code> haben (<code>.a/ac/ah</code> passen z.B. nicht)
<code>*.{c,h,sh}</code>	Dateinamen, die auf <code>.c</code> , <code>.h</code> oder <code>.sh</code> enden (<code>t(c)sh</code> , <code>ksh</code> , <code>bash</code>)
<code>[a-z][a-z]</code>	Kleingeschriebene zweibuchstabile Dateinamen (<code>aa...zz</code>)
<code>RCS/*.c,v</code>	Dateinamen mit Endung <code>.c,v</code> im Verzeichnis <code>RCS</code>
<code>~USER/src/*/*a*.c</code>	Dateinamen in den Verzeichnissen <code>src/*</code> des Benutzers <code>USER</code> , die dem Muster <code>a*.c</code> entsprechen
<code>/*_/*/*_/*/*/*</code>	Dateinamen (und Verzeichnisse) im Root-Verzeichnis, dessen Unter-Verzeichnissen und Unter-Unter-Verzeichnissen
<code>[!.]*</code> <code>[^.]*</code>	Versteckte Dateinamen, nicht aber die Verzeichnisse <code>.</code> und <code>..</code> (<code>.*</code> würde auch auf die Verzeichnisse <code>.</code> und <code>..</code> passen)
<code>*abc*</code>	Dateinamen, die im Namen die Zeichenkette <code>abc</code> enthalten (auch die Dateinamen <code>abc</code> , <code>abc...</code> und <code>...abc</code> passen)
<code>*[0-9]*[0-9]*</code>	Dateinamen, die <i>mindestens</i> zwei Ziffern enthalten

Achtung: Dateinamen, die *genau* zwei Ziffern enthalten, können in der Shell prinzipiell nicht gematcht werden, da keine Wiederholung von Zeichen (oder einer Zeichenklasse) angebar ist!

2.2 Metazeichen zur Textsuche

Gibt es schon bei den diversen Shells Unterschiede in der Syntax und den Möglichkeiten der Regulären Ausdrücke, so ist dies bei der Textsuche noch deutlicher ausgeprägt:

- Prinzipiell kann zwischen folgenden **Varianten** von Regulären Ausdrücken unterschieden werden:
 - ▷ Die **Basic Regular Expressions (BRE)** sind überall vorhanden und ihre Syntax ist einheitlich.
 - ▷ Die **Extended Regular Expressions (ERE)** sind teilweise nicht oder nur eingeschränkt vorhanden und ihre Syntax ist unterschiedlich.
 - ▷ Die **Perl Compatible Regular Expressions (PCRE)** waren ursprünglich eine umfangreiche Erweiterung von Regulären Ausdrücken im Rahmen der Programmiersprache *perl*. Diese Erweiterungen wurden in viele andere Programmiersprachen übernommen, ihre Syntax ist einheitlich.

Je nach Programm werden mindestens **BRE** oder **ERE** teilweise oder vollständig oder **PCRE** unterstützt.

- In einigen Programmen werden einige oder alle der „erweiterten“ Metazeichen { } | () durch einen **Backslash** \ davor gekennzeichnet (z.B. in *egrep*). In anderen haben sie direkt die Bedeutung als Metazeichen und werden durch einen Backslash \ davor in ein normales Zeichen umgewandelt (z.B. im *awk*). Grund dafür ist, dass man mit nachträglichen Erweiterungen die bereits für diese Programme im Einsatz befindlichen Reguläre Ausdrücke nicht ungültig machen wollte. **[P]**.
- In einigen Programmen erzeugt ein **Backslash** \ vor einem **normalen Zeichen** ein Metazeichen, das eine spezielle Bedeutung hat (z.B. in *perl* die Metazeichen \s \S \d \D \w \W).
- Dass oft mehrere Programme für fast den gleichen Zweck existieren (z.B. *grep*, *egrep*, *fgrep*) liegt vor allem daran, dass alte (Shell-)Skripte und Anwendungen durch die Einführung neuer Möglichkeiten nicht beeinflusst werden sollten (**Aufwärtskompatibilität**). Aus dem gleichen Grund wurden auch aus heutiger Sicht schlechte Entwurfsentscheidungen (nicht nur im Bereich der Reguläre Ausdrücke ; -) nicht korrigiert **[P]**.
- Je **später** ein Programm entstanden ist, desto mehr Möglichkeiten bietet es üblicherweise (das Extrembeispiel ist hier *perl* mit seinen PCRE), da neue Erkenntnisse, Forschungsergebnisse und die Erweiterungen der Vorläufer natürlich berücksichtigt wurden.
- Für eine genaue Auflistung der Möglichkeiten der einzelnen Kommandos auf dem jeweiligen UNIX-System sollte man immer einen Blick in die `man`-Pages der Kommandos und insbesondere in die `man`-Page `regex(5)` werfen.

2.2.1 Unterschiede zur Shell

In Shells und in Regulären Ausdrücken werden (leider) die gleichen Metazeichen für unterschiedliche Funktionen verwendet. Hier eine Auflistung der Unterschiede zwischen den Metazeichen der Regulären Ausdrücke zu denen der Shell:

- Der `.` steht für **ein beliebiges Zeichen** und ersetzt das `?`.

- Der ***** steht für die **Wiederholung** des *vorherigen Zeichens* oder der *vorherigen Zeichenklasse*, nicht mehr für eine beliebige Zeichenfolge (d.h. * ist durch .* zu ersetzen). Der * ist also nicht für sich alleine verwendbar.
- Die **Negation** ! in [. . .] wird durch ^ ersetzt.
- Das ? wird für die 0- oder 1-malige Wiederholung des Zeichens/der Zeichenklasse davor verwendet (erst in der erweiterten Version der Regulären Ausdrücke verfügbar) und steht nicht mehr für **ein beliebiges Zeichen**.
- Reguläre Ausdrücke sind im Gegensatz zu Mustern in der Shell **nicht automatisch verankert**, dies ist explizit anzugeben (über ^ und \$).
- Eine **Quotierung** kann nur mehr für einzelne Zeichen durch \ erfolgen, " und ' sind ganz normale Zeichen.
- **Leerraum** (Leerzeichen, Tabulator, ...) wird immer als **echtes Zeichen** interpretiert, und nicht mehr als ein **Trennzeichen**, das von der Shell letztlich ignoriert wird (falls es nicht quotiert ist).

2.2.2 Standard Metazeichen (BRE)

Standard-Metazeichen (Basic Regular Expressions, BRE) sind überall vorhanden und ihre Syntax ist einheitlich.

Metaz.	Beschreibung
.	1 beliebiges Zeichen
x^*	0–∞ Wiederholungen von Zeichen x davor
^	Zeilenanfang
\$	Zeilenende
\ x	Metazeichen x <i>quotieren</i>
[abc], [$a-z$]	1 Zeichen aus Zeichenmenge ([$a-z$] = Zeichenbereich)
[abc], [^a-z]	1 Zeichen nicht aus Zeichenmenge (alle außer diesen)

2.2.3 Erweiterte Metazeichen (ERE)

Erweiterte Metazeichen (Extended Regular Expressions, ERE) sind teilweise gar nicht oder nur eingeschränkt vorhanden und ihre Syntax ist unterschiedlich (* = vorhanden, = nicht vorhanden, + = ohne \ davor in Perl, g = nur im Gawk und ebenfalls ohne \ davor).

Metaz.	grep egrep sed awk perl vi				Beschreibung
$x?$	*	*	*	*	0/1 Wiederholung des Teils x davor (Option)
$x+$	*	*	*	*	1– ∞ Wiederholungen des Teils x davor
$x y$	*	*	*	*	Entweder x oder y (Alternative)
(...)	*	*	*	*	Klammerung mehrerer Zeichen (Grouping)
$x\{m,n\}$	*	g	+	*	m – n Wiederholungen des Teils x davor
$x\{m,\}$	*	g	+	*	m – ∞ Wiederholungen des Teils x davor
$x\{m\}$	*	g	+	*	m Wiederholungen des Teils x davor (genau)
$\backslash n$		*	*		Zeilenvorschub
$\backslash(\dots\backslash)$		*	g	+	Zeichenkette merken (in $\backslash 1..9$)
$\backslash <$ $\backslash >$				*	Wortanfang, Wortende
$\backslash b$ $\backslash B$				*	Wortgrenze, keine Wortgrenze (break)

2.2.4 Ersatzdarstellung

Folgende **Ersatzdarstellungen** (bzw. Definitionen) von Metazeichen gelten.

Regex	Ersatzdarstellung	Beschreibung
x^*	$x\{0,\}$	Abschluss (Closure)
$x?$	$x\{0,1\}$	Option
x^+	$x\{1,\}$	Nichtleerer Abschluss (Closure)
xx^+	xx^*	Nichtleerer Abschluss (Closure)
$[a-z]$	$\backslash(a b c \dots z\backslash)$	Zeichenklasse (Menge von Zeichen)
$\backslash .$	$[\]$	Das Zeichen „Punkt“
$\backslash _$	$[_]$	Leerzeichen (zur Verdeutlichung!)
x	$[x]$	Zeichen x (zur Verdeutlichung!)

2.2.5 Vorrang

Der **Vorrang** der Metazeichen-Operatoren in absteigender Reihenfolge.

Operator	Beschreibung
(...)	Klammerung (Grouping)
* + ? {...}	Quantifizierer (Multiplikatoren)
Rest	Anker, Konkatenation (Verkettung)
	Alternative

2.2.6 Hinweise

- Redundante Klammern können bei passendem Vorrang weggelassen werden.
- Die Zeichen in einer Zeichenmenge $[\dots]$ sind **automatisch quotiert**. Soll das Zeichen $]$ in einer Zeichenmenge enthalten sein, ist es als **1. Zeichen** zu schreiben ($[] \dots]$). Soll das Zeichen $-$ in einer Zeichenmenge enthalten sein, muss es als **1. oder letztes Zeichen** in der Menge stehen ($[- \dots]$ oder $[\dots -]$).

- Das Zeichen / muss im *sed/awk/perl/vi* mit \ quotiert werden, wenn es im Such- oder Ersetzungsmuster vorkommt, da Reguläre Ausdrücke dort normalerweise durch / begrenzt werden. In *sed/perl/vi* (nicht im *awk*) kann auch ein beliebiges anderes Begrenzungszeichen verwendet werden (dann ist dafür dieses im Such- oder Ersetzungsmuster zu quotieren, falls es dort vorkommt). Soll z.B. /bin durch /usr/local/bin ersetzt werden, so gibt es folgende Möglichkeiten, dies auszudrücken (s **[substitute]**):

```
s\/bin\/usr\/local\/bin\/    # Standard (/ ist zu quotieren!)
s#/bin#/usr/local/bin/#      # Begrenzungszeichen #
s@/bin@/usr/local/bin/@      # Begrenzungszeichen @
...
```

- Zeilenanfang ^ und Zeilenende \$ stehen nicht für ein Zeichen, sondern für den **Zwischenraum** zwischen Zeilenanfang und dem ersten Zeichen bzw. dem letzten Zeichen und dem Zeilenende.
- Wortanfang **<** und Wortende **>** stehen nicht für ein Zeichen, sondern für einen **Zeichenübergang** Non-Letter ↔ Letter oder umgekehrt. Was als **Letter** betrachtet wird, hängt vom verwendeten Tool ab, normalerweise die Gross/Kleinbuchstaben, der Unterstrich und die Ziffern (d.h. [A-Za-z_0-9], eventuell auch die Umlaute).
- Die Wortgrenze **b** steht nicht für ein Zeichen, sondern für einen **Zeichenübergang** Non-Letter ↔ Letter oder umgekehrt. Keine Wortgrenze **B** steht nicht für ein Zeichen, sondern für einen **Zeichenübergang** Letter ↔ Letter oder Non-Letter ↔ Non-Letter. Was als **Letter** betrachtet wird, hängt vom verwendeten Tool ab, normalerweise die Gross/Kleinbuchstaben, der Unterstrich und die Ziffern (d.h. [A-Za-z_0-9], eventuell auch die Umlaute).
- Im *gawk* sind die Operatoren { } nur bei Angabe der Option `--re-interval` funktionsfähig, sie werden ohne den Backslash davor geschrieben.
- Ein Zeilenanfang oder -ende gilt auch als Wortanfang oder -ende.
- **Control-Zeichen** (Steuerzeichen) können durch Voranstellen von `Ctrl-V` (**verbose**) eingegeben und in Regulären Ausdrücken verwendet werden (z.B. `Ctrl-M` durch `Ctrl-V Ctrl-M`). Sie werden auf dem Bildschirm in der Form `^x` dargestellt. Für einige gibt es auch Ersatzdarstellungen in Form von sogenannten **Escape-Sequenzen**, die durch \ eingeleitet werden (z.B. `\n`, `\r`, `\t`, ...).

2.2.7 Besonderheiten von *grep*

Folgende Schalter beeinflussen das Verhalten von *grep*:

Option	Beschreibung
-c	Nur die Anzahl der passenden Zeilen ausgeben [count]
-h	Dateinamen nicht ausgeben (bei mehr als einer Datei) [hide/head]
-i	Gross/Kleinschreibung ignorieren [ignore case]
-l	Nur Dateinamen ausgeben (Muster passt auf mind. eine Zeile) [list]
-n	Zeilennummer den passenden Zeilen voranstellen [number]
-v	Nach <i>nicht</i> passenden Zeilen suchen [vice versa]
-R	Alle Dateien in Verzeichnisbaum rekursiv durchsuchen [recursive]

2.2.8 Escape-Sequenzen von *gawk* und *perl*

Folgende **Escape-Sequenzen** sind in *gawk* und *perl* vorhanden:

Metaz.	Beschreibung
<code>\a</code>	Akustisches Signal [alert]
<code>\f</code>	Seitenvorschub [form feed]
<code>\n</code>	Zeilenvorschub [newline]
<code>\r</code>	Wagenrücklauf [carriage return]
<code>\t</code>	Tabulator
<code>\v</code>	Vertikaler Tabulator (nicht in <i>Perl</i> !)
<code>\ddd</code>	Zeichen mit oktalem Wert <i>ddd</i> (Zahlen zwischen 000 und 377) [octal]
<code>\xdd</code>	Zeichen mit hexadezimalen Wert <i>dd</i> (Zahlen zwischen 00 und ff/FF) [hexadecimal]

2.2.9 POSIX-Zeichenklassen von *gawk* und *perl*

Im *gawk* und in *perl* sind innerhalb von Zeichenlisten [...] in Regulären Ausdrücken **POSIX-Zeichenklassen** der Form `[:class:]` erlaubt. Sie dienen zur Angabe von Zeichen unabhängig von der verwendeten Zeichencodierung (ASCII, EBCDIC, ...), aber z.B. abhängig von der verwendeten Landessprache. Folgende POSIX-Zeichenklassen *class* gibt es:

Klasse	perl	gawk	Bedeutung
alnum		*	Alphanumerische Zeichen (Buchstaben + Ziffern)
alpha		*	Buchstaben
ascii			ASCII-Bereich 1-127
blank		*	Leerzeichen oder Tabulator
cntrl		*	Control-Zeichen
digit	<code>\d</code>	*	Dezimalziffern
graph		*	Alle druckbaren und sichtbaren Zeichen
lower		*	Kleine Buchstaben
print		*	Druckbare Zeichen (keine Kontroll-Zeichen)
punct		*	Satzzeichen
space	<code>\s</code>	*	Whitespace (Leerzeichen, Tabulator, Zeilenvorschub, ...)
upper		*	Große Buchstaben
word	<code>\w</code>		Wortzeichen
xdigit		*	Hexadezimalziffern

In *perl* gibt es für die Zeichenklassen `[:digit:]`, `[:space:]` und `[:word:]` die alternative Darstellung `\d`, `\s` und `\w`. Im *gawk* gibt es die beiden Zeichenklassen `[:ascii:]` und `[:word:]` nicht. In *perl* kann eine Zeichenklasse durch `^` nach dem ersten Doppelpunkt negiert werden: `[:^digit:]`.

Beispiel (erst ein Buchstabe, dann beliebig viele Buchstabe, Ziffern oder Unterstriche):

```
/[[[:alpha:]]][[:alnum:]]_+/?
```

Achtung: Die eckigen Klammern *müssen doppelt geschrieben werden* (`[[:alpha:]]`), die Form `[:alpha:]` ist nicht korrekt.

2.2.10 Besonderheiten von *perl* (PCRE)

Die Programmiersprache *perl* bietet mit den **Perl Compatible Regular Expressions** (PCRE) wesentlich erweiterte Möglichkeiten in den Regulären Ausdrücken. Es gibt viele weitere Metazeichen gegenüber BRE und ERE, wie z.B. (keine vollständige Liste).

Metaz.	Beschreibung
\A \Z	Absoluter Zeilenanfang / -ende (beim Matchen von Text mit mehreren \n darin)
\b \B	Wortgrenze / keine Wortgrenze [break/no break]
\G	An letzter Trefferposition von m/.../g beginnen [Global]
\s \S	Leerraum (Leerzeichen, horizontaler + vertikaler Tabulator, Zeilenvorschub Wagenrücklauf, Seitenvorschub) / kein Leerraum [space/no space]
\d \D	Ziffer = [0-9] / keine Ziffer = [^0-9] [digit/no digit]
\w \W	Buchstabe = [a-zA-Z_0-9] / kein Buchstabe = [^a-zA-Z_0-9] [word/no word]
\e	Escape-Zeichen
\Cx	Steuerzeichen <i>x</i> [control]
\Nname	Unicode-Zeichen <i>name</i> [name]
\lx	Nächstes Zeichen <i>x</i> in Kleinschreibung umwandeln [lowercase]
\ux	Nächstes Zeichen <i>x</i> in Grossschreibung umwandeln [uppercase]
\l... \E	Alle Zeichen in Kleinschreibung umwandeln bis \E [Lowercase]
\u... \E	Alle Zeichen in Grossschreibung umwandeln bis \E [Uppercase]
\Q... \E	Alle Metazeichen quotieren bis \E [Quote]

In *perl* ist es möglich, durch Anhängen von **Optionen** am Ende des Regulären Ausdrucks seine Verhaltensweise insgesamt zu modifizieren.

Option	Name	Beschreibung
/.../i	ignorecase	Gross/Kleinschreibung ignorieren
/.../g	global	Alle Treffer ersetzen oder Matchposition merken
/.../s	single	. matcht auch \n
/.../m	multiple	^ und \$ matchen auch \n im Stringinneren
/.../p	preserve	Text vor/in/nach Treffer statt generell merken in \$`, \$&, \$'
/.../x	extended	Leerraum, Zeilenvorschub und Kommentar erlaubt

Insbesondere das **Extended-Format** verwandelt die Write-Only-Programmiersprache Reguläre Ausdrücke wieder in eine lesbar Sprache:

```
my ($day, $month, $year) = ($date =~ m/
    ^                               # Zeilenanfang
    (
        [012]?[1-9]|[123]0|31      # Tag 1-31 merken
    )
    [.\-/]                          # Trennzeichen
    (
        0?[1-9]|1[012]            # Monat 1-12 merken
    )
    [.\-/]                          # Trennzeichen
    (
        \d\d(\d\d)?                # Jahr 00-99,0000-9999 merken
    )
)
```

```

$ # Zeilenende
/(x);

```

In *perl* ist es möglich, durch Anhängen eines `?` an den Wiederholungsoperator nur das **kleinstmögliche** passende Stück Text zu ersetzen (**non-greedy**).

Regex	Beschreibung
$x^*?$	0–∞ Wiederholungen des Teils x davor
$x^+?$	1–∞ Wiederholungen des Teils x davor
$x??$	0/1 Wiederholung des Teils x davor (Option)
$x\{m, n\}?$	m – n Wiederholungen des Teils x davor
$x\{m, \}$?	m –∞ Wiederholungen des Teils x davor
$x\{m\}$?	m Wiederholungen des Teils x davor (genau)

In *perl* ist es möglich, durch Anhängen eines `+` an den Wiederholungsoperator das **Backtracking** zu verhindern. D.h. einmal gematchte Textstücke werden nicht mehr freigegeben (possessive = „besitzergreifende“ Quantifizierer).

Regex	Beschreibung
x^*+	0–∞ Wiederholungen des Teils x davor
x^++	1–∞ Wiederholungen des Teils x davor
$x?+$	0/1 Wiederholung des Teils x davor (Option)
$x\{m, n\}+$	m – n Wiederholungen des Teils x davor
$x\{m, \}+$	m –∞ Wiederholungen des Teils x davor
$x\{m\}+$	m Wiederholungen des Teils x davor (genau)

Weiterhin ist es in *perl* möglich, über **Erweiterte Muster** der Form `(?...)` Ersetzungen abhängig von der Umgebung der Ersetzungsstelle (**lookafter**, **lookbehind**) sowie weitere Tricks durchzuführen

Regex	Beschreibung
<code>(?#text)</code>	Kommentar <i>text</i>
<code>(?pimsx)</code>	Optionen <i>pimsx</i> für folgendes Teilmuster setzen
<code>(?-imsx)</code>	Optionen <i>imsx</i> für folgendes Teilmuster zurücksetzen
<code>(?:regex)</code>	Klammerung ohne speichern in $\backslash n$ oder $\$n$
<code>(?=regex)</code>	Positiver Look-Ahead (muss vorkommen, wird nicht gematcht)
<code>(?!regex)</code>	Negativer Look-Ahead (darf nicht vorkommen, wird nicht gematcht)
<code>(?<=regex)</code>	Positiver Look-Behind (muss vorkommen, wird nicht gematcht)
<code>(?<!regex)</code>	Negativer Look-Behind (darf nicht vorkommen, wird nicht gematcht)

Siehe Literatur oder die Perl Online-Dokumentation `perlre`, `perlref`, `perlrequick`, `perl tut`, `perlrebackslash` und `perlrecharclass`

2.3 Beispiele zu Suchmustern

Folgende Suchmuster passen auf die jeweils beschriebenen Zeichenketten, Zeilen oder Inhalte einer String-Variablen (TAB steht für das Tabulatorzeichen, `_` für das Leerzeichen):

<code>abc</code>	Zeichenkette <code>abc</code> irgendwo in Zeile
<code>^abc</code>	Zeichenkette <code>abc</code> am Zeilenanfang
<code>abc\$</code>	Zeichenkette <code>abc</code> am Zeilenende
<code>^abc\$</code>	Zeile, die genau die Zeichenkette <code>abc</code> enthält
<code>[Aa][Bb][Cc]</code>	<code>abc</code> in beliebiger Gross/Kleinschreibung
<code>[0-9][0-9]*</code>	Ganzzahl (mind. 1 Ziffer)
<code>[0-9]+</code>	analog mit <code>+</code> statt <code>*</code>
<code>[0-9][0-9]*\.[0-9][0-9]*</code>	Gleitkommazahl (mind. 1 Ziffer vor+nach Dez.punkt)
<code>[0-9]+\.[0-9]+</code>	analog mit <code>+</code> statt <code>*</code>
<code>[A-Za-z_][A-Za-z_0-9]*</code>	C-Bezeichner (mind. 1 Zeichen, keine Ziffer am Anfang)
<code>80([1-4]?86 1?88)</code>	8086, 80186, 80286, 80386, 80486, 8088, 80188
<code>\<abc\></code>	Das Wort <code>abc</code> (kann nicht Teil eines Wortes sein)
<code>^[^#][^#]*#[_TAB]*include</code>	<code>#include</code> -Anweisung, die nicht am Zeilenanfang steht
<code>^[^#]+#[_TAB]*include</code>	analog mit <code>+</code> statt <code>*</code>
<code>^.....L..._</code>	Zeilen mit genau 10 Zeichen, an Position 6 muss das Zeichen <code>L</code> , am Zeilenende muss ein Leerz. stehen
<code>^\{5\}L.\{3\}_</code>	
<code>\(\<[_]*\>\)_*\<1\></code>	Zwei gleiche Wörter hintereinander (<i>vi/ved</i>)
<code>(\b[_]*\b)_*\b1\b</code>	Zwei gleiche Wörter hintereinander (<i>perl</i>)
<code>[_~]</code>	ASCII-Zeichen außer 32-127 (d.h. Ctrl- + Sonderzeichen)
<code>^[^0-9]*[0-9][^0-9]*[0-9][^0-9]*\$</code>	Genau 2 Ziffern in einer Zeile

In *awk* (und *perl*) kann ein Regulärer Ausdruck **schrittweise** durch Zuweisung an Variablen und Verketteten der Variableninhalte aufgebaut werden.

2.3.1 Großes Beispiel A: Gleitkommazahlen erkennen (Awk-Syntax)

```

SIGN = "[+-]"           # Vorzeichen, nicht "[+-]!"
DIGIT = "[0-9]"         # Ziffern
DEC = DIGIT "+" "[.]?" DIGIT "*" # Dezimalzahl
FRAC = "[.]" DIGIT "+"  # Nachkommazahl
EXP = "[Ee]" SIGN "?" DIGIT "+" # Exponent
REAL = SIGN "?" "(" DEC "|" FRAC ")" "(" EXP ")"?
      = "[+-]?([0-9]+[.]?[0-9]*|.[0-9]+)([Ee][+-]?[0-9]+)?"

```

`REAL` passt dann auf alle Dezimalzahlen mit Vorzeichen, Vorkomma- und/oder Nachkommastellen und (vorzeichenbehaftetem) Exponenten, also z.B. auf:

```
123   -3.141592   +.0123   1E2   100e-200   ...
```

Durch die Kombination von `DEC` und `FRAC` wird erzwungen, dass entweder mindestens eine Vorkommastelle oder ein Dezimalpunkt und mindestens eine Nachkommastelle in der Zahl vorhanden sein muss. Dies ist durch ein Muster der Form

```
DIGIT "*" "[.]?" DIGIT "*"
```

nicht erreichbar, da dieses Muster z.B. auch auf eine *leere* Zeichenkette oder einen einzelnen Dezimalpunkt passen würde.

2.3.2 Großes Beispiel B: Datumswerte erkennen (Perl-Syntax)

```
$day   = "[012]?[1-9]|[[123]0|31]"      # Tag [0]1-31
$month = "0?[1-9]|1[012]"              # Monat [0]1-12
$year  = "\d\d(\d\d)?"                  # Jahr 00-99,0000-9999
$date  = "($day)[.\/]($month)[.\/]($year)"
```

`$date` passt dann auf alle Datumswerte der Form `TT.MM.JJJJ` bzw. `TT/MM/JJ`, also z.B. auf:

```
1.1.1111   31.12.1999   01.03.1792   29/2/89   ...
```

Es handelt sich dabei um eine *rein syntaktische Prüfung*, der 31. in Monaten ohne 31. Tag bzw. der 30./29. im Monat Februar ohne 30. Tag und ohne 29. Tag außerhalb eines Schaltjahres werden nicht abgewiesen. Ebenso wird die amerikanische Schreibweise `MM/DD/YY` nicht erkannt und es ist eine Mischung der beiden Trennzeichen `.` und `/` erlaubt.

Achtung: Die Klammern um die einzelnen Komponenten sind notwendig, da die Oder-Verknüpfung `|` den niedrigsten Vorrang hat.

2.4 Metazeichen im Ersetzungsteil

Folgende Metazeichen sind im Ersetzungsteil verfügbar (g = nur im `Gawk`):

Metaz.	sed awk perl vi	Beschreibung
<code>\n</code>	* g (*) *	<i>n</i> -te per <code>\(...\)</code> gemerkte Zeichenkette (<i>n</i> =1..9, alt)
<code>\$n</code>	* *	<i>n</i> -te per <code>\(...\)</code> gemerkte Zeichenkette (<i>n</i> =1..9, neu)
<code>&</code>	* * * *	Vollständiges Suchmuster einsetzen
<code>~</code>	* *	Vorheriges Suchmuster verwenden
<code>\u \l</code>	* *	Nächstes Zeichen in Gross/Kleinschrift umwandeln
<code>\U \L</code>	* *	Folgende Zeichen in Gross/Kleinschrift umwandeln
<code>\E</code>	* *	Durch <code>\U</code> oder <code>\L</code> begonnene Umwandlung beenden

2.4.1 Hinweise

- Das Zeichen `&` muss quotiert werden (`\&`), wenn es im Ersetzungstext verwendet werden soll (sonst wird an seiner Stelle das Suchmuster eingesetzt).
- Das Zeichen `~` muss quotiert werden (`\~`), wenn es im Suchmuster verwendet werden soll (sonst wird an seiner Stelle das letzte Suchmuster eingesetzt).
- Das Begrenzungszeichen `/` muss quotiert werden (`\/`).
- Durch `\(...\)` (bzw. `(...)` in `gawk` und `perl`) können auch *mehrere* Musterteile gemerkt und im Ersetzungsteil wiederverwendet werden, diese Klammern können sogar verschachtelt werden. Um die Nummer für die Referenz `\n` auf den mit einem Klammernpaar gemerkten Musterteil zu ermitteln, sind die öffnenden Klammern `\(` von links nach rechts durchzunummerieren.

- `\1.. \9` oder `$1.. $9` können auch im Suchteil des Suchmusters verwendet werden und passen auf den vorher im Suchteil per `\(. . \)` gemerkten Text mit der entsprechenden Nummer (**Backreference**). Hierdurch kann z.B. die Verwendung des gleichen Trennzeichens in Datumswerten erkannt werden:

```
$date = "($day)([./])($month)\2($year)"
```

2.4.2 Beispiele zu Such- und Ersetzungsmustern

Folgende Such- und Ersetzungsmuster führen die jeweils beschriebene Änderung durch (TAB steht für das Tabulatorzeichen, `_` für das Leerzeichen):

<code>/^[_TAB]*\$/d</code>	Alle Leerzeilen löschen (<i>sed</i> , <code>d</code> [delete])
<code>s/_*_/_/</code>	Mehr als ein Leerzeichen hintereinander in eines umwandeln
<code>s/_+/_/</code>	analog mit + statt * (<i>s</i> [substitute])
<code>s/.*cp_&_~/src/</code>	Zeile umwandeln in <code>cp_ZEILE_~/src</code>
<code>s/.*/\U&/</code>	Zeile in Grossschrift umwandeln
<code>s/\(\<[Uu]nix\>\)/\U\1/</code>	Wörter der Form <code>unix/Unix</code> in <code>UNIX</code> umwandeln
<code>s/^\(...\)\(..\)\2\1/</code>	Die Spalten 1-3 mit den Spalten 4-5 vertauschen
<code>s/^\(...\)\.\/\1/</code>	Die Spalten 4-5 löschen
<code>s/"([^\"]* \\")"//g</code>	Alle C-Strings (" <code>...</code> ") löschen (<i>g=global</i>)
<code>s/\([^_][^_]*\)\(.*\)\(^[^_][^_]*\)/\3\2\1/</code>	Die ersten beiden durch Leerzeichen getrennten Texte jeder Zeile vertauschen (Leerzeichen bleiben erhalten)
<code>s/\([^_]+\)\(.*\)\(^[^_]+\)/\3\2\1/</code>	Analog mit + statt *

3 Anwendung

3.1 Quotierung

- Bei *ls* und *echo* müssen die Anführungszeichen um `PATTERN` weggelassen werden, da die Shell den Regulären Ausdruck sonst *nicht* sieht. Sie allein ist nämlich für die Expansion der Muster in die Dateinamen zuständig.
- Die Anführungszeichen um `PATTERN` bzw. `REGEXP` bzw. das Skript beim Aufruf von *grep/expr/find/sed/awk/perl* sind *notwendig*, da sonst bereits die Shell die Metazeichen interpretieren würde und nicht erst das Kommando selbst.

3.2 Suchen anwenden

Anwenden kann man Reguläre Ausdrücke zum Suchen mit folgenden Kommandos:

- Auflisten der zu `PATTERN` passenden Dateinamen (im aktuellen Verzeichnis bzw. in allen Unterverzeichnissen). Durch die Angabe der Option `-d` [**directory only**] wird der Inhalt von passenden Verzeichnissen nicht aufgelistet:

```
ls -d PATTERN
echo PATTERN
find . -name 'PATTERN' -print
```

- Auflisten der zu REGEXP passenden Zeilen in den Textdateien FILE... (-n [noprint], -p [print], -e [execute], p [print], d [delete], ! [not]):

```
grep 'REGEXP' FILE...
sed -n '/REGEXP/p' FILE...           oder
sed '/REGEXP/d' FILE...
awk '/REGEXP/' FILE...
perl -ne '/REGEXP/ && print' FILE...
```

- Ausgabe des zu REGEXP passenden Teils der Zeichenkette TEXT bzw. seiner Länge:

```
expr TEXT : 'REGEXP'
```

Wird ein Teil von REGEXP in \(\...\) geklammert, so gibt *expr* diesen Teil aus, sonst wird die *Anzahl* der passenden Zeichen ausgegeben. Der Mustervergleich wird bei *expr* immer beim *ersten* Zeichen von TEXT begonnen (**automatische Verankerung**).

- Suchen der zu REGEXP passenden Zeilen in einer Textdatei im *vi*:

```
/REGEXP/
```

3.3 Suchen und Ersetzen anwenden

Anwenden kann man Reguläre Ausdrücke zum Suchen und Ersetzen durch folgende Kommandos:

- Ersetzen des *ersten* zu REGEX passenden Teils einer Zeichenkette durch SUBST (s [substitute], p [print], e [execute]):

```
sed 's/REGEX/SUBST/'
awk '{ sub(/REGEX/, "SUBST"); print }'
perl -pe 's/REGEX/SUBST/'
```

- Ersetzen *aller* zu REGEX passenden Teile einer Zeichenkette durch SUBST (g [global]):

```
sed 's/REGEX/SUBST/g'
awk '{ gsub(/REGEX/, "SUBST"); print }'
perl -pe 's/REGEX/SUBST/g'
```

- Ersetzen des zu REGEX passenden Teils einer Zeichenkette durch SUBST im *vi*. % steht für *alle Zeilen*, g [global] steht für *alle Vorkommen in der Zeile*. Beide Angaben können auch weggelassen werden, dann wird in der aktuellen Zeile das 1. Vorkommen ersetzt. Soll das *N*-te Vorkommen ersetzt werden, ist nach dem abschließenden / die Zahl *N* anzugeben:

```

:s/REGEX/SUBST/          # 1. Vorkommen in aktueller Zeile
:s/REGEX/SUBST/g        # ALLE Vorkommen in aktueller Zeile
:s/REGEX/SUBST/N        # N. Vorkommen in aktueller Zeile
:%s/REGEX/SUBST/       # Erstes Vorkommen in ALLEN Zeilen
:%s/REGEX/SUBST/g      # ALLE Vorkommen in ALLEN Zeilen
:%s/REGEX/SUBST/N      # N. Vorkommen in ALLEN Zeilen

```

4 Weitere Beispiele zu Suchmustern

Die folgenden Muster passen nur auf Zeilen ...

```

b*           ... die leer sind oder b oder bb oder ... enthalten
ab*c        ... die ac oder abc oder abbc oder ... enthalten
abb*c       ... die abc oder abbc oder abbbc oder ... enthalten
ab+c        ... die abc oder abbc oder abbbc oder ... enthalten
ab?c        ... die ac oder abc enthalten
[A-Z]+      ... die einen oder mehrere Grossbuchstaben enthalten
(ab)+c      ... die abc oder ababc oder abababc oder ... enthalten
^abc        ... die abc am Anfang enthalten
abc$        ... die abc am Ende enthalten
^abc$       ... die genau abc enthalten
^.$         ... die genau ein beliebiges Zeichen enthalten
^...$       ... die genau drei beliebige Zeichen enthalten
.           ... die mindestens ein beliebiges Zeichen enthalten
...         ... die mindestens drei beliebige Zeichen enthalten
\. $        ... die am Ende einen Punkt enthalten
^[abc]      ... die a, b oder c am Anfang enthalten
^[^abc]     ... die nicht a, b oder c am Anfang enthalten
[^abc]      ... die mindestens ein Zeichen ungleich a, b oder c enthalten
^[^A-Z]$    ... die genau ein Zeichen enthalten, aber keinen Grossbuchstaben

```

5 Kurzübersicht

Shell-Metazeichen	
Metaz.	Beschreibung
?	1 beliebiges Zeichen
*	0 oder mehr beliebige Zeichen (<i>nicht</i> Wiederholung eines Zeichens) [P]
\ <i>x</i>	Metazeichen <i>x</i> <i>quotieren</i> (\ \ steht für \ selbst!)
[<i>abc</i>], [<i>a-z</i>]	1 Zeichen aus Zeichenmenge (Zeichenklasse , [<i>a-z</i>] = Zeichen von a bis z)
[! <i>abc</i>], [! <i>a-z</i>]	1 Zeichen nicht aus Zeichenmenge (<i>sh</i> , <i>ksh</i> , <i>bash</i>) [P]
[^ <i>abc</i>], [^ <i>a-z</i>]	1 Zeichen nicht aus Zeichenmenge (<i>t(c)sh</i> , <i>bash</i>) [P]
{ <i>abc, def, ...</i> }	Liste von Zeichenketten (<i>t(c)sh</i> , <i>ksh</i> , <i>bash</i>) [P]
~	Home-Verzeichnis des aktuellen Users (<i>t(c)sh</i> , <i>ksh</i> , <i>bash</i>)
~ <i>USER</i>	Home-Verzeichnis des Users <i>USER</i> (<i>t(c)sh</i> , <i>ksh</i> , <i>bash</i>)

Shell-Quotierung	
Metaz.	Beschreibung
\ <i>x</i>	Quotiert ein einzelnes Zeichen <i>x</i> mit Sonderbedeutung.
" <i>xyz</i> "	Quotiert alle Zeichen außer \$ ` \ (und !).
' <i>xyz</i> '	Quotiert alle Zeichen (außer ! und ' selbst).

Standard-Metazeichen	
Metaz.	Beschreibung
.	1 beliebiges Zeichen
<i>x</i> *	0–∞ Wiederholungen von Zeichen <i>x</i> davor
^	Zeilenanfang
\$	Zeilenende
\ <i>x</i>	Metazeichen <i>x</i> <i>quotieren</i>
[<i>abc</i>], [<i>a-z</i>]	1 Zeichen aus Zeichenmenge ([<i>a-z</i>] = Zeichenbereich)
[^ <i>abc</i>], [^ <i>a-z</i>]	1 Zeichen nicht aus Zeichenmenge (alle außer diesen)

Erweiterte Metazeichen					
Metaz.	grep	egrep	sed	awk perl vi	Beschreibung
<i>x</i> ?	*	*	*	*	0/1 Wiederholung des Teils <i>x</i> davor (Option)
<i>x</i> +	*	*	*	*	1–∞ Wiederholungen des Teils <i>x</i> davor
<i>x</i> <i>y</i>	*	*	*	*	Entweder <i>x</i> oder <i>y</i> (Alternative)
(...)	*	*	*	*	Klammerung mehrerer Zeichen (Grouping)
<i>x</i> \{ <i>m, n</i> \}	*		g	+ *	<i>m–n</i> Wiederholungen des Teils <i>x</i> davor
<i>x</i> \{ <i>m, \</i> \}	*		g	+ *	<i>m–∞</i> Wiederholungen des Teils <i>x</i> davor
<i>x</i> \{ <i>m</i> \}	*		g	+ *	<i>m</i> Wiederholungen des Teils <i>x</i> davor (genau)
\ <i>n</i>		*		*	Zeilenvorschub
\(<i>... \</i>)		*	g	* *	Zeichenkette merken (in \1.. \9)
\< \>				*	Wortanfang, Wortende
\b \B				*	Wortgrenze, keine Wortgrenze (break)

Ersatzdarstellungen		
Regex	Ersatzdarstellung	Beschreibung
x^*	$x\{0,\}$	Abschluss (Closure)
$x?$	$x\{0,1\}$	Option
x^+	$x\{1,\}$	Nichtleerer Abschluss (Closure)
x^+	xx^*	Nichtleerer Abschluss (Closure)
$[a-z]$	$\backslash(a b c \dots z\backslash)$	Zeichenklasse (Menge von Zeichen)
$\.$	$[.]$	Das Zeichen „Punkt“
$_$	$[_]$	Leerzeichen (zur Verdeutlichung!)
x	$[x]$	Zeichen x (zur Verdeutlichung!)

Vorrang	
Operator	Beschreibung
(\dots)	Klammerung (Grouping)
$* + ? \{ \dots \}$	Quantifizierer (Multiplikatoren)
<i>Rest</i>	Anker, Verkettung
$ $	Alternative

Metazeichen im Ersetzungsteil				
Metaz.	sed	awk	perl vi	Beschreibung
$\backslash n$	*	g	* *	n -te per $\backslash(\dots\backslash)$ gemerkte Zeichenkette ($n=1..9$)
$\&$	*	*	* *	Vollständiges Suchmuster einsetzen
\sim			*	Vorheriges Suchmuster verwenden
$\backslash u \backslash l$			* *	Nächstes Zeichen in Gross/Kleinschrift umwandeln
$\backslash U \backslash L$			* *	Folgende Zeichen in Gross/Kleinschrift umwandeln
$\backslash E$			* *	Durch $\backslash U$ oder $\backslash L$ begonnene Umwandlung beenden

Optionen von grep	
Option	Beschreibung
-c	Nur die Anzahl der passenden Zeilen ausgeben [count]
-h	Dateinamen nicht ausgeben (bei mehr als einer Datei) [hide/head]
-i	Gross/Kleinschreibung ignorieren [ignore case]
-l	Nur Dateinamen ausgeben (Muster passt auf mind. eine Zeile) [list]
-n	Zeilennummer den passenden Zeilen voranstellen [number]
-v	Nach <i>nicht</i> passenden Zeilen suchen [vice versa]
-R	Alle Dateien in Verzeichnisbaum rekursiv durchsuchen [recursive]

Escape-Sequenzen von gawk und perl	
Metaz.	Beschreibung
<code>\a</code>	Akustisches Signal [alert]
<code>\f</code>	Seitenvorschub [form feed]
<code>\n</code>	Zeilenvorschub [newline]
<code>\r</code>	Wagenrücklauf [carriage return]
<code>\t</code>	Tabulator
<code>\v</code>	Vertikaler Tabulator
<code>\ddd</code>	Zeichen mit oktalem Wert <i>ddd</i> (Zahlen zwischen 000 und 377) [octal]
<code>\xdd</code>	Zeichen mit hexadezimalen Wert <i>dd</i> (Zahlen zwischen 00 und ff/FF) [hexadecimal]

POSIX-Zeichenklassen von gawk und perl			
Klasse	perl	gawk	Bedeutung
alnum		*	Alphanumerische Zeichen (Buchstaben + Ziffern)
alpha		*	Buchstaben
ascii		*	ASCII-Bereich 1-127
blank		*	Leerzeichen oder Tabulator
cntrl		*	Control-Zeichen
digit	<code>\d</code>	*	Dezimalziffern
graph		*	Alle druckbaren und sichtbaren Zeichen
lower		*	Kleine Buchstaben
print		*	Druckbare Zeichen (keine Kontroll-Zeichen)
punct		*	Satzzeichen
space	<code>\s</code>	*	Whitespace (Leerzeichen, Tabulator, Zeilenvorschub, ...)
upper		*	Große Buchstaben
word	<code>\w</code>	*	Wortzeichen
xdigit		*	Hexadezimalziffern

Metazeichen von perl	
Metaz.	Beschreibung
<code>\A \Z</code>	Absoluter Zeilenanfang / -ende (beim Matchen von Text mit mehreren <code>\n</code> darin)
<code>\b \B</code>	Wortgrenze / keine Wortgrenze [break/no break]
<code>\G</code>	An letzter Trefferposition von <code>m/.../g</code> beginnen [Global]
<code>\s \S</code>	Leerraum (Leerzeichen, horizontaler + vertikaler Tabulator, Zeilenvorschub Wagenrücklauf, Seitenvorschub) / kein Leerraum [space/no space]
<code>\d \D</code>	Ziffer = <code>[0-9]</code> / keine Ziffer = <code>^[^0-9]</code> [digit/no digit]
<code>\w \W</code>	Buchstabe = <code>[a-zA-Z_0-9]</code> / kein Buchstabe = <code>^[^a-zA-Z_0-9]</code> [word/no word]
<code>\e</code>	Escape-Zeichen
<code>\Cx</code>	Steuerzeichen <i>x</i> [control]
<code>\Nname</code>	Unicode-Zeichen <i>name</i> [name]
<code>\lx</code>	Nächstes Zeichen <i>x</i> in Kleinschreibung umwandeln [lowercase]
<code>\ux</code>	Nächstes Zeichen <i>x</i> in Grossschreibung umwandeln [uppercase]
<code>\l... \E</code>	Alle Zeichen in Kleinschreibung umwandeln bis <code>\E</code> [Lowercase]
<code>\u... \E</code>	Alle Zeichen in Grossschreibung umwandeln bis <code>\E</code> [Uppercase]
<code>\Q... \E</code>	Alle Metazeichen quotieren bis <code>\E</code> [Quote]

Regex-Optionen von perl		
Option	Name	Beschreibung
/.../i	ignorecase	Gross/Kleinschreibung ignorieren
/.../g	global	Alle Treffer ersetzen oder Matchposition merken
/.../s	single	. matcht auch \n
/.../m	multiple	^ und \$ matchen auch \n im Stringinneren
/.../p	preserve	Text vor/in/nach Treffer statt generell merken in \$`, \$&, \$'
/.../x	extended	Leerraum, Zeilenvorschub und Kommentar erlaubt

Non-Greedy Operatoren von perl	
Regex	Beschreibung
$x^*?$	0–∞ Wiederholungen des Teils x davor
$x^+?$	1–∞ Wiederholungen des Teils x davor
$x??$	0/1 Wiederholung des Teils x davor (Option)
$x\{m, n\}?$	m – n Wiederholungen des Teils x davor
$x\{m, \}\?$	m –∞ Wiederholungen des Teils x davor
$x\{m\}?$	m Wiederholungen des Teils x davor (genau)

Backtracking verhindern in perl	
Regex	Beschreibung
x^{*+}	0–∞ Wiederholungen des Teils x davor
x^{++}	1–∞ Wiederholungen des Teils x davor
$x^{?+}$	0/1 Wiederholung des Teils x davor (Option)
$x\{m, n\}^+$	m – n Wiederholungen des Teils x davor
$x\{m, \}^+$	m –∞ Wiederholungen des Teils x davor
$x\{m\}^+$	m Wiederholungen des Teils x davor (genau)

Erweiterte Muster in perl	
Regex	Beschreibung
$(?#text)$	Kommentar $text$
$(?pimsx)$	Optionen $pimsx$ für folgendes Teilmuster setzen
$(?-imsx)$	Optionen $imsx$ für folgendes Teilmuster zurücksetzen
$(?:regex)$	Klammerung ohne speichern in $\backslash n$ oder $\$n$
$(?=regex)$	Positiver Look-Ahead (muss vorkommen, wird nicht gematcht)
$(?!regex)$	Negativer Look-Ahead (darf nicht vorkommen, wird nicht gematcht)
$(?<=regex)$	Positiver Look-Behind (muss vorkommen, wird nicht gematcht)
$(?<!regex)$	Negativer Look-Behind (darf nicht vorkommen, wird nicht gematcht)

Metazeichen im Ersetzungsteil				
Metazeichen	sed	awk	perl vi	Beschreibung
<code>\n</code>	*	g	(*) *	<i>n</i> -te per <code>\(. . . \)</code> gemerkte Zeichenkette (<i>n</i> =1..9, alt)
<code>\$n</code>			*	<i>n</i> -te per <code>\(. . . \)</code> gemerkte Zeichenkette (<i>n</i> =1..9, neu)
<code>&</code>	*	*	* *	Vollständiges Suchmuster einsetzen
<code>~</code>			*	Vorheriges Suchmuster verwenden
<code>\u \l</code>			* *	Nächstes Zeichen in Gross/Kleinschrift umwandeln
<code>\U \L</code>			* *	Folgende Zeichen in Gross/Kleinschrift umwandeln
<code>\E</code>			* *	Durch <code>\U</code> oder <code>\L</code> begonnene Umwandlung beenden

6 ASCII-Tabelle

Der ASCII-Zeichencode definiert die **Standardbelegung** der Codes 0–127 mit Zeichen (kennt keine landesspezifischen Sonderzeichen wie z.B. Umlaute). Die Codes 128–255 werden je nach Zeichensatz unterschiedlich belegt (mit Sonderzeichen wie z.B. Umlauten) und sind hier nicht dargestellt. Die wichtigsten ASCII-Zeichen und ihre Reihenfolge sind:

- **Steuer-Zeichen** (Control) (0–31, *zusammenhängend*)
- **Leerzeichen** (32)
- **Ziffern** 0–9 (48–57, *zusammenhängend*)
- **Großbuchstaben** A–Z (65–90, *zusammenhängend*)
- **Kleinbuchstaben** a–z (97–122, *zusammenhängend*)
- **Tilde** ~ (126)
- **Druckbare Zeichen** SPACE–~ (32–127, *zusammenhängend*)

d.h. es gelten folgende **Beziehungen**: SPACE < 0–9 < A–Z < a–z < ~

Hex		00	10	20	30	40	50	60	70
Hex	Dez	00	16	32	48	64	80	96	112
0	0	^@ [NUL]	^P	[SPACE]	0	@	P	`	p
1	1	^A	^Q	!	1	A	Q	a	q
2	2	^B	^R	"	2	B	R	b	r
3	3	^C	^S	#	3	C	S	c	s
4	4	^D	^T	\$	4	D	T	d	t
5	5	^E	^U	%	5	E	U	e	u
6	6	^F	^V	&	6	F	V	f	v
7	7	^G [BEL]	^W	'	7	G	W	g	w
8	8	^H [BS]	^X	(8	H	X	h	x
9	9	^I [TAB]	^Y)	9	I	Y	i	y
A	10	^J [LF]	^Z	*	:	J	Z	j	z
B	11	^K [VTAB]	^[[ESC]	+	;	K	[k	{
C	12	^L [FF]	^\	,	<	L	\	l	
D	13	^M [CR]	^]	-	=	M]	m	}
E	14	^N	^^	.	>	N	^	n	~
F	15	^O	^_	/	?	O	_	o	[DEL]

Hinweise:

- ^x steht für `Ctrl-x` (Control) oder `Strg-x` (Steuerung) und beschreibt die Terminal-Steuerzeichen.
- **Zeichennamen**: BEL = Glocke, BS = Backspace, CR = Carriage Return, DEL = Delete, ESC = Escape, FF = Formfeed, LF = Linefeed, SPACE = Leerzeichen, TAB = Tabulator, VTAB = Vertikaler Tabulator.