Reguläre Ausdrücke: Beschreibung und Anwendung

Version 1.42 — 07.11.2016

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

Inhaltsverzeichnis

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, Csh, Tcsh, Ksh, Bash, Zsh*.
- **Texteditoren** zum Suchen (und Ersetzen) von Zeichenketten: *Ed, Ex, Vi/Vim, Emacs, UltraEdit, Word*.
- Kommandozeilentools mit **eingebauter Zeichenkettenverarbeitung** ebenfalls zum Suchen (und Ersetzen) von Zeichenketten: *(e/f)grep, gres, ack, expr, find, more, less, csplit, (f)lex, Sed*.
- Programmiersprachen mit **eingebauter Zeichenkettenverarbeitung** ebenfalls zum Suchen (und Ersetzen) von Zeichenketten: *Awk, Gawk, JavaScript, PHP, Perl, Python, Ruby, Tcl/Tk, Icon, Lua*.
- **Server-Konfigurationsdateien** zum Matchen von URLs, Mailadressen, Dateinamen, Fehlermeldungen, Logmeldungen: *Apache, Postfix, Sendmail, Privoxy, Squidguard, 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]     # Variante A (Shell ersetzt Muster durch passende Namen)
echo *.[ch]      # Variante B (Shell ersetzt Muster durch passende Namen)
```

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

```
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                    # oder
awk '{ gsub(/[Uu]nix/, "UNIX"); print }' kap1.txt > kap1.new  # oder
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   # oder
perl -nie 's/\s*$/\n/; print if /\S/' kap1.txt       #
```

1.2 Begriffe

Folgende Begriffe sind im Zusammenhang mit Regulären Ausdrücken gebräuchlich:

Begriff	Beschreibung
Anchor	Zeilenanfang/-ende oder Wortanfang/-ende festlegen (**Verankern**).
Backreference	Verweis auf vorher gematchten Musterteil später im Muster.
Backtracking	Zurücknahme bereits gematchter Textteile, weil kein Gesamtmatch mehr möglich ist (evtl. für lange Laufzeiten bei ungeschickt formulierten Regulären Ausdrücken verantwortlich).
BRE	**Basic Regular Expressions**: Überall vorhanden, einheitliche Syntax.
ERE	**Extended RegExp**: Nicht überall vorhanden, unterschiedliche Syntax.
Escape-Sequenz	Sonderzeichen gebildet durch Backslash + normalem Buchstaben (z.B. `\n` = Newline, `\r` = Carriage Return, `\t` = Tabulator).
File Globbing	Shell-Metazeichen (z.B. `* ?`) durch passende Dateinamen ersetzen. (**Dateinamen-Expansion**).
Jokerzeichen	Metazeichen der Shell (z.B. `* ?`), um Dateinamen zu matchen.
Globbing	Shell-Metazeichen (z.B. `* ?`) durch passende Dateinamen ersetzen.
Greedy	Matcht möglichst viele Zeichen (**Gierig**).
Grouping	Klammern eines regulären Ausdrucks zur Wiederholung oder zum Muster speichern (**Gruppierung**).
Konkatenation	**Verkettung**: *Natürliche Operation* mit Zeichen einer Zeichenkette. Daher ohne expliziten Operator in den Regulären Ausdrücken, sondern durch bloßes *Hintereinanderschreiben* ausgedrückt.
Lazy	Matcht möglichst wenige Zeichen (**Non-Greedy**, **Faul**).
Left-most	Arbeitet von links nach rechts.
Literal	Zeichen, das für sich selbst steht (**Zeichen-Konstante**).
Lookahead	Erweiterung des Ankerkonzepts, legt notwendigen Kontext **nach/rechts von** Suchmuster fest, ohne ihn zu matchen.
Lookbehind	Erweiterung des Ankerkonzepts, legt notwendigen Kontext **vor/links von** Suchmuster fest, ohne ihn zu matchen.
Matchen	In einer Zeichenkette eine zu einem Regulären Ausdruck passende Teilkette suchen (treffen, passen).
. . . matcht. . .	. . . passt auf. . .
Metazeichen	Stehen nicht für sich selbst, sondern für eine Operation oder eine Menge von Zeichen (**Wildcard**, **Jokerzeichen**).
Non-Greedy	Matcht möglichst wenige Zeichen (**Lazy**, **Faul**).
Pattern	Konkreter Regulärer Ausdruck (**(Text)Muster**/**Schablone**).
Pattern Matching	Vergleich Regulärer Ausdruck mit Zeichenkette (**Mustererkennung**).
PCRE	**Perl Compatible Regular Expressions**: Wesentliche Erweiterung der Möglichkeiten von BRE/ERE in der Programmiersprache Perl. In viele andere Programmiersprachen übernommen.
Possessive	**Backtracking** verhindern, d.h. einmal gematchte Textstücke nicht mehr freigeben (**Stingy**, **Besitzergreifend**).
POSIX Regex	Definierter UNIX-Standard für Reguläre Ausdrücke (ERE + BRE).
Quantifier	Operatoren zur Wiederholung eines Regulären Ausdrucks.
Quoten	Zeichenkette vor Interpretation durch Shell schützen oder Metazeichen in Literale umwandeln (**Quotieren**/**Zitieren**/**Schützen**).
RA	Regulärer Ausdruck.
RE	Regular Expression.
Stingy	**Backtracking** verhindern, d.h. einmal gematchte Textstücke nicht mehr freigegeben (**Possessive**).
Verankern	Zeilenanfang/-ende oder Wortanfang/-ende festlegen (**Anchor**).
Whitespace	**Leerraum** = Leerzeichen ␣, Tabulator `\t`, vertikaler Tab. `\v`, Zeilenvorschub `\n`, Wagenrücklauf `\r`, Seitenvorschub `\f`.
Wildcard	Metazeichen der Shell (z.B. `* ?`), um Dateinamen zu matchen.

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 `*` steht für beliebig viele beliebige Zeichen in einem Dateibezeichner. Folgende Einschränkungen gelten unter Windows:

  - ▷ Der Punkt zwischen Name und Extension kann durch `?` und `*` nicht gematcht werden.
  - ▷ Pfadnamen mit Verzeichnistrenner `\` können nicht gematcht werden (verschachtelte Verzeichnisse).
  - ▷ `*` kann nicht mehrfach im Namen oder der Extension angegeben werden.

- Reguläre Ausdrücke sind typischerweise in Skriptsprachen wie *Awk, Gawk, JavaScript, PHP, Perl, Python, Ruby, Tcl/Tk, Icon, Lua* direkt eingebaut. Aber auch in Compilersprachen wie *C, C++, C#, Java* 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, Gawk, Perl, Vi/Vim*), in anderen nicht (z.B. *grep, egrep*).
- Bei vielen Programmiersprachen sind reguläre Ausdrücke als **Strings** in `"..."` oder `'...'` anzugeben (z.B. *Python, PHP*). Dies macht die Nutzung etwas kompliziert, da in einem String der **Backslash** `\` und **Escape-Sequenzen** `\.` geschützt werden müssen, um sie unverändert für den Regulären Ausdruck zu belassen.
- Bei *PHP* ist es noch etwas komplizierter, da diese Skriptsprache beide Varianten ERE und PCRE unterstützt (bzw. ab Version 7 nur noch PCRE) und den Regulären Ausdruck immer in Form eines Strings `"..."` oder `'...'` erwartet (bei PCRE zusätzlich noch `/` außenrum).

```
REGEX                          # Grep, Egrep
/REGEX/                        # Sed, Awk, Perl, Vi/Vim
"REGEX"                        # Perl (alternativ)
ereg('REGEX', ...)             # PHP (ERE, Funktionsaufruf)
preg_match('/REGEX/', ...)     # PHP (PCRE, Funktionsaufruf)
re.search(r"REGEX", ...)       # Python (PCRE, Objektmethode, r=Raw String)
```

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

```
MMMM passt auf xxMMMMxxxMMMMx
                <-->        1. passender Teil, 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 (bei *PCRE* änderbar). Beispiel:

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

- Ein **Zeilenvorschub** (newline) kann normalerweise nicht gematcht werden (außer durch *Sed* und *PCRE*-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

- Michael Fitzgerald, *Einstieg in Reguläre Ausdrücke*, O'Reilly.
- Michael Fitzgerald, *Introducing Regular Expressions*, O'Reilly.
- Alfred Aho, Peter Weinberger, Brian Kernighan, *The AWK Programming Language*, Addison-Wesley.
- Jeffrey Friedl, *Mastering Regular Expressions, 3. Edition*, O'Reilly.
- Jeffrey Friedl, *Reguläre Ausdrücke, 3. Ausgabe*, O'Reilly.
- Tony Stubblebine, *Reguläre Ausdrücke kurz & gut*, O'Reilly.
- Christian Wenz, *Reguläre Ausdrücke — schnell + kompakt*, entwickler.press.
- Jan Goyvaerts, Steven Levithan, *Reguläre Ausdrücke Kochbuch*, O'Reilly.
- John Bambenek, Agnieszka Klus, *grep - kurz&gut*, O'Reilly.

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**):

Metazeichen	Beschreibung
$\backslash x$	Quotiert ein einzelnes Zeichen x mit Sonderbedeutung.
"xyz"	Quotiert alle Zeichen außer `$ ` ` \` (und `!`).
'xyz'	Quotiert alle Zeichen (außer `!` und `'` 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:

Metazeichen	Beschreibung
`?`	1 beliebiges Zeichen (*außer Verz.trenner / !*)
`*`	0 oder mehr beliebige Zeichen (*außer Verz.trenner / !*) [**P**]
`**`	0 oder mehr beliebige Zeichen (*inklusive Verz.trenner / !*)
`\`x	Metazeichen x *quotieren* (`\\` 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 (*Sh, Ksh, Bash, Zsh*) [**P**]
`[^`abc`]`, `[^`a`-`z`]`	1 Zeichen **nicht** aus Zeichenmenge (*Csh, Tcsh, Bash, Zsh*) [**P**]
`{`abc`,`def`,...}`	Liste von Zeichenketten (*Csh, Tcsh, Ksh, Bash, Zsh*) [**P**]
`~`	Home-Verzeichnis des aktuellen Users (*Csh, Tcsh, Ksh, Bash, Zsh*)
`~USER`	Home-Verzeichnis des Users `USER` (*Csh, Tcsh, Ksh, Bash, Zsh*)

- 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 22 Zeichen eine **Sonderbedeutung** (sie müssen daher durch Quoten geschützt werden, wenn sie Teil eines Dateinamens sein sollen):

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

- In der *Csh, Tcsh, Ksh, Bash, Zsh* haben zusätzlich folgende 4 Zeichen (insgesamt also 26 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 oder durch `\` zu schützen, 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 können (matcht also beliebig lange Dateipfade).
- 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.
- Die Argumentliste ist 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 (`-print0` bzw. `-0` steht für das Nul-Byte als Stringende):

  ```
  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
  ```

  Hinweis: In der *Bash* deaktiviert die Option `nullglob` dieses Verhalten:

  ```
  set -o nullglob
  ```

  Ein Muster, das auf keinen Dateinamen passt, wird dann aus der Parameterliste des auszuführenden Kommandos entfernt.

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:

Muster	Beschreibung
`~USER/src/*/a*.c`	Dateinamen in den Verzeichnissen `src/*` des Benutzers
`*.c`	Dateinamen, die auf `.c` enden
`*.[ch]`	Dateinamen, die auf `.c` oder `.h` enden
`*[!.][!ch]`	Dateinamen, die als vorletztes Zeichen nicht `.` und als letztes nicht `c` oder `h` haben (`.a/ac/ah` passen z.B. nicht)
`*.{c,h,sh}`	Dateinamen, die auf `.c`, `.h` oder `.sh` enden (*Csh, Tcsh, Ksh, Bash, Zsh*)
`[a-z][a-z]`	Kleingeschriebene zweibuchstabige Dateinamen (`aa...zz`)
`RCS/*.c,v`	Dateinamen mit Endung `.c,v` im Verzeichnis `RCS`
`~USER/src/*/a*.c`	Dateinamen in den Verzeichnissen `src/*` des Benutzers `USER`, die dem Muster `a*.c` entsprechen
`/*␣/*/*␣/*/*/*`	Dateinamen (und Verzeichnisse) im Root-Verzeichnis, dessen Unter-Verzeichnissen und Unter-Unter-Verzeichnissen
`.[!.]*  ...]*`	Versteckte Dateinamen, nicht aber die Verzeichnisse `.` und `..` (`.*` würde auch auf die Verzeichnisse `.` und `..` passen)
`*abc*`	Dateinamen, die im Namen die Zeichenkette `abc` enthalten (auch die Dateinamen `abc`, `abc...` und `...abc` passen)
`*[0-9]*[0-9]*`	Dateinamen, die *mindestens* 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) angebbar 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 werden folgenden **Varianten** von Regulären Ausdrücken unterschieden:
  - ▷ **Basic Regular Expressions** (**BRE**) sind überall vorhanden und ihre Syntax ist einheitlich.
  - ▷ **Extended Regular Expressions** (**ERE**) sind teilweise nicht oder nur eingeschränkt vorhanden und ihre Syntax ist unterschiedlich.
  - ▷ **POSIX Regular Expressions** sind ein Standard, der BRE und ERE für die UNIX-Tools festlegt.
  - ▷ **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** und teilweise bzw. vollständig **ERE** oder **PCRE** unterstützt. Mit Schaltern oder über den Programmnamen ist bei manchen Werkzeugen der Typ ausgewählbar (z.B. beim `grep`: `-G -E -P -F`).
- In manchen 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 *PCRE* 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 `;-)`) belassen [**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 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 `regexp(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 *Zeichens*/der *vorherigen Zeichenklasse* direkt davor, 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 direkt 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 (per `^` 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 mindestens vorhanden und ihre Syntax ist einheitlich.

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

2.2.3 Erweiterte Metazeichen (ERE)

Erweiterte Metazeichen (Extended Regular Expressions, ERE) sind möglicherweise vorhanden und ihre Syntax ist unterschiedlich ($\star$ = vorhanden, g = nur im Gawk).

Metazeichen	grep	egrep	Sed	Awk	PCRE	Vi/Vim	Beschreibung
x`?`		$\star$		$\star$	$\star$		0/1 Wiederholung des Teils x davor (**Option**)
x`+`		$\star$		$\star$	$\star$		1–∞ Wiederholungen des Teils x davor
x`|`y		$\star$		$\star$	$\star$		Entweder x oder y (**Alternative**)
`(...)`		$\star$		$\star$	$\star$		Klammerung mehrerer Zeichen (**Gruppierung**)
x{m`,`n}		$\star$		g	$\star$	$\star$	m–n Wiederholungen des Teils x davor
x{m`,`}		$\star$		g	$\star$	$\star$	m–∞ Wiederholungen des Teils x davor
x{m}		$\star$		g	$\star$	$\star$	m Wiederholungen des Teils x davor (genau)
`\n`			$\star$		$\star$	$\star$	Zeilenvorschub
`(...)`			$\star$	g	$\star$	$\star$	Zeichenkette merken (in `\1`..`\9`)
`\<` `\>`						$\star$	Wortanfang/Wortende
`[[:<:]]` `[[:>:]]`				g	$\star$		Wortanfang/Wortende
`\b` `\B`					$\star$		Wortgrenze/keine Wortgrenze (**break**)

2.2.4 Ersatzdarstellung

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

Regex	Ersatzdarstellung	Beschreibung
$x\star$	x{`0,`}	Abschluss (Closure)
x`?`	x{`0,1`}	Option
x`+`	x{`1,`}	Nichtleerer Abschluss (Closure)
x`+`	$xx\star$	Nichtleerer Abschluss (Closure)
`[`a–z`]`	`(`a`|`b`|`c`|...|`z`)`	Zeichenklasse (Menge von Zeichen)
`\.`	`[.]`	Das Zeichen „Punkt“
␣	`[`␣`]`	Leerzeichen (zur Verdeutlichung!)
x	`[`x`]`	Zeichen x (zur Verdeutlichung!)

2.2.5 Vorrang

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

Operator	Beschreibung
`\`x	Quotierung nächstes Zeichen
`[...]`	Zeichenklasse
`(...)`	Klammerung (**Gruppierung**)
`* + ? {...}`	Quantifizierer (Multiplikator)
—	Konkatenation/Verkettung (kein Operator)
`^ $`	Anker
`|`	Alternative (ERE)

2.2.6 Hinweise

- Redundante Klammern können bei passendem Vorrang weggelassen werden.
- Zeichen in einer Zeichenmenge `[...]` sind **automatisch quotiert**.
  - ▷ Soll das Zeichen `]` in einer Zeichenmenge enthalten sein, ist es als **1. Zeichen** zu schreiben `[]...]` oder zu quotieren `[...\]...]`.
  - ▷ Soll das Zeichen `-` in einer Zeichenmenge enthalten sein, ist es als **1. Zeichen** `[-...]` oder als **letztes Zeichen** `[...-]` zu schreiben oder zu quotieren `[...\-...]`).
- Das Zeichen `/` muss im *Sed/Awk/Perl/Vi/Vim* mit `\` quotiert werden, wenn es im Such- oder Ersetzungsmuster vorkommt, da Reguläre Ausdrücke dort normalerweise durch `/` **begrenzt** werden. In *Sed/Perl/Vi/Vim* (nicht im *Awk*) kann auch ein beliebiges anderes Begrenzungszeichen verwendet werden — dann ist dafür dieses im Such- oder Ersetzungsmuster mit `\` 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. Das **Wortinnere** `\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 Anzahl passender 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 *nicht* passenden Zeilen suchen [**vice versa**]
`-r`	Verz.baum rekursiv durchsuchen (symb. Links ignorieren) [**recursive**]
`-R`	Analog, symb. Links nicht ignorieren [**Recursive**]

2.2.8 Escape-Sequenzen von *Gawk* und *Perl*

Folgende **Escape-Sequenzen** sind in *Gawk* und *Perl* vorhanden:

Metazeichen	Beschreibung
`\a`	Akustisches Signal [**alert**]
`\f`	Seitenvorschub [**form feed**]
`\n`	Zeilenvorschub [**newline**]
`\r`	Wagenrücklauf [**carriage return**]
`\t`	Tabulator
`\v`	Vertikaler Tabulator (nicht in Perl!)
`\`ddd	Zeichen mit oktalem Wert ddd (Zahlen zwischen `000` und `377`) [**octal**]
`\x`dd	Zeichen mit hexadezimalem Wert dd (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`	`\d`	*	Dezimalziffern
`graph`	*	*	Alle druckbaren und sichtbaren Zeichen
`lower`	*	*	Kleine Buchstaben
`print`	*	*	Druckbare Zeichen (keine Kontroll-Zeichen)
`punct`	*	*	Satzzeichen
`space`	`\s`	*	Whitespace (Leerzeichen, Tabulator, Zeilenvorschub, ...)
`upper`	*	*	Große Buchstaben
`word`	`\w`		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*, z.B. `[[: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):

Metazeichen	Beschreibung
`\A \Z \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 `[ \t\r\n\f]` (Leerzeichen, horiz. Tabulator, Zeilenvorschub, Wagenrücklauf, Seitenvorschub) / kein Leerraum `[^ \t\r\n\f]` [**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
`\C`x	Steuerzeichen x [**control**]
`\N`$name$	Unicode-Zeichen $name$ [**name**]
`\l`x	Nächstes Zeichen x in Kleinschreibung umwandeln [**lowercase**]
`\u`x	Nächstes Zeichen x 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
`/.../e`	**execute**	Anweisung im Ersetzungsteil ausführen
`/.../x`	**extended**	Leerraum, Zeilenvorschub und Kommentar erlaubt
`/.../g`	**global**	Alle Treffer ersetzen oder Matchposition merken
`/.../i`	**ignorecase**	Gross/Kleinschreibung ignorieren
`/.../m`	**multiple**	Metazeichen `^` und `$` matchen auch `\n` im Inneren von Strings
`/.../o`	**once**	Nur 1× übersetzen (Optimierung)
`/.../p`	**preserve**	Text vor/in/nach Treffer (statt generell merken in `$`‵, `$&`, `$'`) merken in `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}`
`/.../s`	**single**	Metazeichen `.` matcht auch `\n`

Insbesondere das **Extended-Format** verwandelt die Write-Only-Programmiersprache Reguläre Ausdrücke wieder in eine lesbare Sprache, indem es Formatierung mit Leerraum und Kommentare darin erlaubt (dafür sind `#` und das Leerzeichen ␣ im RA folgendermaßen zu schreiben: `\#` und `[␣]` oder `\␣` oder `\s` (allgemeiner Leerraum).

```
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
    )                               #
    $2                              # Gleiches Trennzeichen erneut
    (                               #
        \d\d(\d\d)?                 # Jahr 00-99,0000-9999 merken
    )                               #
    $                               # Zeilenende
}x);                                # x=Extended Format
```

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

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**, **stingy** = „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 (**lookahead/lookbehind** = erweiterte Anker) sowie weitere Tricks durchzuführen.

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 $\$n$ (**reine Gruppierung**)
$(?=regex)$	**Positiver Lookahead** (muss danach/rechts vorkommen, nicht gematcht)
$(?!regex)$	**Negativer Lookahead** (darf danach/rechts nicht vorkommen, nicht gematcht)
$(?<=regex)$	**Positiver Lookbehind** (muss davor/links vorkommen, nicht gematcht)
$(?<!regex)$	**Negativer Lookbehind** (darf davor/links nicht vorkommen, nicht gematcht)

Siehe Literatur in Abschnitt 1.5 auf Seite 8 oder die Perl Online-Dokumentation `perlre`, `perlrebackslash`, `perlrecharclass`, `perlrequick`, `perlreref` und `perlretut`.

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):

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

In vielen Skriptsprachen kann ein Regulärer Ausdruck auch **schrittweise** als Zeichenkette in Variablen aufgebaut werden.

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

```
BEGIN {                                  # Verketten durch Hintereinanderschreiben
   SIGN  = "[-+]"                        # Vorzeichen, nicht "[+-]"!
   DIGIT = "[0-9]"                       # Ziffern
   DEC   = DIGIT "+" "[.]?" DIGIT "*"    # Dezimalzahl
   FRAC  = "[.]" DIGIT "+"               # Nachkommazahl
   EXP   = "[Ee]" SIGN "?" DIGIT "+"     # Exponent
   REAL  = "^" SIGN "?" "(" DEC "|" FRAC ")" "(" EXP ")?" "$"
#        = /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)([Ee][+-]?[0-9]+)?$/
}
$0 ~ REAL { print "Fließkommazahl", $0 }
```

`REAL` passt dann auf alle Dezimalzahlen mit Vorzeichen, Vorkomma- und/oder Nachkommstellen 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)

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

while (<>) {
   chomp;
   if (/$date/) {
      print "DATUM: $_\n";
   }
}
```

`$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   3/1/1792   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 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):

Metazeichen	Sed	Awk	Perl	Vi/Vim	Beschreibung
$\backslash n$	*	g	(*)	*	n-te per `(...)` gemerkte Zeichenkette (n=1..9, alt)
$n			*		n-te per `(...)` gemerkte Zeichenkette (n=1..9, neu)
&	*	*	*	*	Vollständiges Suchmuster einsetzen
~				*	Vorheriges Suchmuster verwenden
\u \l			*	*	Nächstes Zeichen in Gross/Kleinschrift umwandeln
\U \L			*	*	Folgende Zeichen in Gross/Kleinschrift umwandeln
\E			*	*	Durch `\U` oder `\L` 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 (bzw. `$`n in Perl) auf den mit einem Klammernpaar gemerkten Musterteil zu ermitteln, sind die öffnenden Klammern `(` *von links nach rechts* durchzunumerieren.

- `\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 ")"    # Awk-Syntax
$date = "($day)([./])($month)$2($year)"               # Perl-Syntax
```

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):

Metazeichen	Beschreibung
`/^[␣TAB]*$/d`	Alle Leerzeilen löschen (, ... *Sed*, `d` [**delete**])
`s/␣␣␣*/␣/`	Mehr als 1 Leerzeichen hintereinander in 1 umwandeln
`s/␣␣+/␣/`	analog mit `+` statt `*` (`s` [**substitute]**)
`s/.*/cp␣&␣~\/src/`	Zeile umwandeln in `cp␣ZEILE␣~/src`
`s/.*/\U&/`	Zeile in Grossschrift umwandeln
`s/(\<[Uu]nix\>)/\U\1/`	Wörter der Form `unix/Unix` in `UNIX` umwandeln
`s/^(...)(..)/\2\1/`	Die Spalten 1-3 mit den Spalten 4-5 vertauschen
`s/^(...)../\1/`	Die Spalten 4-5 löschen
`s/"([^"]*|\\")"//g`	Alle C-Strings (`"..."`) löschen (`g=global`)
`s/([^␣][^␣]*)(␣*)`	Die ersten beiden durch Leerzeichen getrennten Texte
`  ([^␣][^␣]*)/\3\2\1/`	jeder Zeile vertauschen (Leerzeichen bleiben erhalten)
`s/([^␣]+)(␣*)([^␣]+)/\3\2\1/`	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

Kommandos zum Suchen verwenden oft Reguläre Ausdrücke:

- 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                   # oder
echo PATTERN                    # oder
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...                  # oder
sed -n '/REGEXP/p' FILE...             # oder
sed '/REGEXP/!d' FILE...               # oder
awk '/REGEXP/' FILE...                 # oder
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/Vim*:

```
/REGEXP     # Abschließender "/" darf fehlen, mit Return Suche starten
/REGEXP/    # Mit Return Suche starten
```

3.3 Suchen und Ersetzen anwenden

Kommandos zum Suchen + Ersetzen verwenden oft Reguläre Ausdrücke:

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

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

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

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

- Ersetzen des zu `REGEX` passenden Teils einer Zeichenkette durch `SUBST` im *Vi/Vim*. `%` 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 ...

Muster	Beschreibung
`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	
Metazeichen	**Beschreibung**
`?`	1 beliebiges Zeichen
`*`	0 oder mehr beliebige Zeichen (*außer Verz.trenner / !*) [**P**]
`**`	0 oder mehr beliebige Zeichen (*inklusive Verz.trenner / !*)
`\`*x*	Metazeichen *x* *quotieren* (`\\` 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 (*Sh, Ksh, Bash, Zsh*) [**P**]
`[^`*abc*`]`, `[^`*a*`-`*z*`]`	1 Zeichen **nicht** aus Zeichenmenge (*Csh, Tcsh*, Bash, Zsh) [**P**]
`{`*abc*`,`*def*`,...}`	Liste von Zeichenketten (*Csh, Tcsh, Ksh, Bash, Zsh*) [**P**]
`~`	Home-Verzeichnis des aktuellen Users (*Csh, Tcsh, Ksh, Bash, Zsh*)
`~USER`	Home-Verzeichnis des Users `USER` (*Csh, Tcsh, Ksh, Bash, Zsh*)

Shell-Quotierung	
Metazeichen	**Beschreibung**
`\`*x*	Quotiert ein einzelnes Zeichen *x* mit Sonderbedeutung.
`"`*xyz*`"`	Quotiert alle Zeichen außer `$ ` ` \` (und `!`).
`'`*xyz*`'`	Quotiert alle Zeichen (außer `!` und `'` selbst).

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

Erweiterte Metazeichen						
Metazeichen	**grep**	**egrep**	**Sed**	**Awk**	**Perl**	**Vi/Vim**	**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 (**Gruppierung**)
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)
`\n`			*		*		Zeilenvorschub
`(...)`			*	g	*	*	Zeichenkette merken (in `\1`..`\9`)
`\<` `\>`						*	Wortanfang/Wortende
`[[:<:]]` `[[:>:]]`				g	*		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]	(a|b|c|...|z)	Zeichenklasse (Menge von Zeichen)
\.	[.]	Das Zeichen „Punkt“
␣	[␣]	Leerzeichen (zur Verdeutlichung!)
x	[x]	Zeichen x (zur Verdeutlichung!)

Vorrang	
Operator	**Beschreibung**
\x	Quotierung nächstes Zeichen
[...]	Zeichenklasse
(...)	Klammerung (**Gruppierung**)
* + ? {...}	Quantifizierer (Multiplikator)
—	Konkatenation/Verkettung (kein Operator)
^ $	Anker
|	Alternative (ERE)

Metazeichen im Ersetzungsteil					
Metazeichen	**Sed**	**Awk**	**Perl**	**Vi/Vim**	**Beschreibung**
\n	*	g	*	*	n-te per (...) gemerkte Zeichenkette (n=1..9
&	*	*	*	*	Vollständiges Suchmuster einsetzen
~				*	Vorheriges Suchmuster verwenden
\u \l			*	*	Nächstes Zeichen in Gross/Kleinschrift umwandeln
\U \L			*	*	Folgende Zeichen in Gross/Kleinschrift umwandeln
\E			*	*	Durch \U oder \L begonnene Umwandlung beenden

Optionen von grep	
Option	**Beschreibung**
-c	Nur Anzahl passender 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 *nicht* passenden Zeilen suchen [**vice versa**]
-r	Verz.baum rekursiv durchsuchen (symb. Links ignorieren) [**recursive**]
-R	Analog, symb. Links nicht ignorieren [**Recursive**]

Escape-Sequenzen von Gawk und Perl	
Metazeichen	**Beschreibung**
\a	Akustisches Signal [**alert**]
\f	Seitenvorschub [**form feed**]
\n	Zeilenvorschub [**newline**]
\r	Wagenrücklauf [**carriage return**]
\t	Tabulator
\v	Vertikaler Tabulator
\ddd	Zeichen mit oktalem Wert ddd (Zahlen zwischen 000 und 377) [**octal**]
\x$dd$	Zeichen mit hexadezimalem Wert dd (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	\d	*	Dezimalziffern
graph	*	*	Alle druckbaren und sichtbaren Zeichen
lower	*	*	Kleine Buchstaben
print	*	*	Druckbare Zeichen (keine Kontroll-Zeichen)
punct	*	*	Satzzeichen
space	\s	*	Whitespace (Leerzeichen, Tabulator, Zeilenvorschub, ...)
upper	*	*	Große Buchstaben
word	\w		Wortzeichen
xdigit	*	*	Hexadezimalziffern

Metazeichen von Perl	
Metazeichen	**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 [\t\r\n\f] (Leerzeichen, horiz. Tabulator, Zeilenvorschub, Wagenrücklauf, Seitenvorschub) / kein Leerraum [^ \t\r\n\f] [**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
\C$x$	Steuerzeichen x [**control**]
\N$name$	Unicode-Zeichen $name$ [**name**]
\l$x$	Nächstes Zeichen x in Kleinschreibung umwandeln [**lowercase**]
\u$x$	Nächstes Zeichen x 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**]

Regex-Optionen von Perl		
Option	**Name**	**Beschreibung**
`/.../e`	**execute**	Anweisung im Ersetzungsteil ausführen
`/.../x`	**extended**	Leerraum, Zeilenvorschub und Kommentar erlaubt
`/.../g`	**global**	Alle Treffer ersetzen oder Matchposition merken
`/.../i`	**ignorecase**	Gross/Kleinschreibung ignorieren
`/.../m`	**multiple**	Metazeichen `^` und `$` matchen auch `\n` im Inneren von Strings
`/.../o`	**once**	Nur 1× übersetzen (Optimierung)
`/.../p`	**preserve**	Text vor/in/nach Treffer statt generell merken in `$`` `, `$&`, `$'` merken in `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}`
`/.../s`	**single**	Metazeichen `.` matcht auch `\n`

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 $\$n$ (reine Gruppierung)
$(?=regex)$	Positiver Lookahead (muss danach/rechts vorkommen, nicht gematcht)
$(?!regex)$	Negativer Lookahead (darf danach/rechts nicht vorkommen, nicht gematcht)
$(?<=regex)$	Positiver Lookbehind (muss davor/links vorkommen, nicht gematcht)
$(?<!regex)$	Negativer Lookbehind (darf davor/links nicht vorkommen, nicht gematcht)

Metazeichen im Ersetzungsteil					
Metazeichen	**Sed**	**Awk**	**Perl**	**Vi/Vim**	**Beschreibung**
`\`n	*	g	(*)	*	n-te per `(...)` gemerkte Zeichenkette (n=1..9, alt)
`$`n			*		n-te per `(...)` gemerkte Zeichenkette (n=1..9, neu)
`&`	*	*	*	*	Vollständiges Suchmuster einsetzen
`~`				*	Vorheriges Suchmuster verwenden
`\u \l`			*	*	Nächstes Zeichen in Gross/Kleinschrift umwandeln
`\U \L`			*	*	Folgende Zeichen in Gross/Kleinschrift umwandeln
`\E`			*	*	Durch `\U` oder `\L` 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.