```
Betriebssysteme - Werkzeuge und UNIX-Schnittstelle
==================================================

                Werkzeuge
                =========


          6. Reguläre Ausdrücke
          =====================


Reguläre Ausdrücke - ein Gespenst geht um im Unix.
```



```
Was ist ein regulärer Ausdruck (Wikipedia):


Ein regulärer Ausdruck (englisch regular expression, Abkürzung: Regex)
ist in der Informatik eine Zeichenkette, die der Beschreibung von Mengen
von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.
Reguläre Ausdrücke finden vor allem in der Softwareentwicklung Verwendung.
Neben Implementierungen in vielen Programmiersprachen verfügen auch viele
Texteditoren über reguläre Ausdrücke in der Funktion "Suchen und Ersetzen".

Ein einfacher Anwendungsfall von regulären Ausdrücken sind Wildcards.

Reguläre Ausdrücke können als Filterkriterien in der Textsuche verwendet
werden, indem der Text mit dem Muster des regulären Ausdrucks abgeglichen
wird.  Dieser Vorgang wird auch Pattern Matching genannt.

Zur Implementierung von regulären Ausdrückeng werden deterministische finite
 (endliche) Automaten (DFA) oder nichtdeterministische finite (endliche)
Automaten (NFA) benutzt.
```

Wo werden reguläre Ausdrücke benutzt?

Programm	Autor	Implementierung
awk	Aho, Weinberger, Kernighan	DFA
new awk	Kernighan	DFA
GNU awk	A.Robbins	DFA/NFA
MKS awk	Mortice Kern Systems	Posix-NFA
mawk	M. Brennan	Posix-NFA
egrep	Aho	DFA
MKS egrep	Mortice Kern Systems	Posix-NFA
expr	D.Haight	NFA
grep	K.Thompson	NFA
GNU grep	M.Haertel	DFA/NFA
GNU find	GNU	NFA
more	E.Schienbrood	NFA
perl	L.Wall	NFA
Python	G.v.Rossum	NFA
sed	Lee McMahon	NFA
Tcl	John Ousterhout	NFA
vi	Bill Joy	NFA

DFA - Deterministischer Finiter (endlicher) Automat
NFA - Nichtdeterministischer Finiter (endlicher) Automat


```
DFA
---
"Textgesteuert" - geht vom Text aus und prüft, ob der Ausdruck passt
Vorteile:  Schnell, nicht von der Programmierung des Ausdrucks abhängig
           POSIX-Konform
Nachteile: Keine Klammern, Keine Rückwärtsreferenzen

NFA
---
"Ausdrucksgesteuert" - geht vom Ausdruck aus und prüft ob der Text passt
Vorteil:  der Programmierer kann durch die Gestaltung des Ausdrucks die
          Arbeitsweise des Suchens bestimmen,
          leistungsstärker, da Klammern unterstützt werden
Nachteil: nicht immer POSIX-Konform, verschiedene Implementierung liefern
          unterschiedliche Ergebnisse, eventuell langsamer, da Backtracking

POSIX
-----
Verlangt die Lieferung des "längsten frühesten Treffers", Klammern optional

Regulärer Ausdruck:
  Allgemeine Notation zur Beschreibung von Textmustern
  bestehend aus Zeichen des Codes und Metazeichen
    Muster:
      'Müller'                  -  die Zeichenkette 'Müller'
      '^Bell '                  -  'Bell' am Zeilenanfang
      ' Bell$'                  -  'Bell' am Zeilenende
      '^[       ]*$'           -  <leere Zeilen>
      '\<([a-zA-Z]+)\> +\<\1\>'  -  doppelte Wörter in einem Text
      '^(From|Subject): '       -  'From:' oder 'Subject:' am Zeilenanfang
```

```
Metazeichen
-----------

Metazeichen sind Zeichen, die eine spezielle Funktion haben.
Sollen sie diese Funktion nicht haben, muessen sie mit '\' maskiert werden.

Einzelkämpfer:
     ^  - Kennzeichnet den Zeilenanfang (Zeilenanker)
           /^Bell / beschreibt ein Muster, bei dem das Wort "Bell" am
                    Zeilenanfang steht und von einem Leerzeichen gefolgt
                    wird.

          Beispiel:                                                     b1
            echo "Bell hat gruene Haare" | grep "Bell"
            echo "Der Bell hat gruene Haare" | grep "Bell"
            echo "Bell hat gruene Haare" | grep "^Bell"
            echo "Der Bell hat gruene Haare" | grep "^Bell"

     $  - Kennzeichnet das Zeilenende (Anker, Begrenzer)
           / Bell$/ beschreibt ein Muster, bei dem das Word "Bell" am
                    Zeilenende stehr und vor dem ein Leerzeichen positioniert
                    ist.

          Beispiel:                                                     b2
            echo "Wer hat gruene Haare? Bell" | grep "Bell"
            echo "Wer hat gruene Haare? alle Bells" | grep "Bell"
            echo "Wer hat gruene Haare? Bell" | grep "Bell$"
            echo "Wer hat gruene Haare? alle Bells" | grep "Bell$"
            echo 'Bell$ hat gruene Haare' | grep '^Bell$'
            echo 'Bell$ hat gruene Haare' | grep '^Bell\$'
```



```
Ein universeller Einzelkämpfer:
     .  -  Der Punkt kennzeichnet ein beliebiges Zeichen
           /Schmid./ beschreibt ein Muster, bei dem das Wort "Schmid" von
                     einem beliebigen Zeichen gefolgt werden kann, also
                    "Schmidt", "Schmid ", "Schmidx"

           Beispiel:                                                    b3

             echo "Schmidt hat gruene Haare" | grep "Schmid."
             echo "Schmid hat gruene Haare" | grep "Schmid. "
             echo "Schmidx hat gruene Haare" | grep "Schmid. "

Pärchen:
     \< - Kennzeichnet den Wortanfang (Wortanker)
           /\<Bell/ beschreibt ein Muster, bei dem die Zeichekette "Bell" am
                   Wortanfang erkannt wird. Zeilenanfang ist auch ein
                   Wortanfang.

          Beispiel:                                                     b4

            echo "Bell hat gruene Haare" | grep "Bell"
            echo "DerBell hat gruene Haare" | grep "Bell"
            echo "Der Bell hat gruene Haare" | grep "Bell"
            echo "Bell hat gruene Haare" | grep "\<Bell"
            echo "DerBell hat gruene Haare" | grep "\<Bell"
            echo "Der Bell hat gruene Haare" | grep "\<Bell"
```

```
\> - Kennzeichnet das Wortende (Anker, Begrenzer)
     /Das\>/ beischreibt ein Muster, bei dem die Zeichenkette "Das" am
             Wortende erkannt wird. Zeilenende ist auch ein
             Wortende.

    Beispiel:                                                     b5

      echo "Wer hat gruene Haare? Bell" | grep "Bell"
      echo "Wer hat gruene Haare? alle Bells" | grep "Bell"
      echo "Wer hat gruene Haare? Bell" | grep "Bell\>"
      echo "Wer hat gruene Haare? alle Bells" | grep "Bell\>"
```

Paarkämpfer - Mengen von Zeichen

```
[...] -  verlangt ein Zeichen aus der beschriebenen Menge,
         'b[ea]ll' kennzeichnet die Worte 'bell' und 'ball'

         Beispiel:                                                b6

           echo "Bell hat gruene Haare" | grep "Bell"
           echo "Der Ball ist rund" | grep "Bell"
           echo "Bell hat gruene Haare" | grep "Ball"
           echo "Der Ball ist rund" | grep "Ball"
           echo "Bell hat gruene Haare" | grep "B[ae]ll"
           echo "Der Ball ist rund" | grep "B[ae]ll"
```



```
[^...] - verlangt Zeichen, die nicht aus der beschriebenen Menge stammen,
         'b[ea]ll' kennzeichnet die Worte 'bbll', 'bcll', ..
         aber nicht 'bell' und 'ball'

         Beispiel:                                               b7

              echo "Bell hat gruene Haare" | grep "Bell"
              echo "Der Ball ist rund" | grep "Ball"
              echo "Bolle ist auch ok" | grep "Bell"
              echo "Bolle ist auch ok" | grep "Ball"
              echo "Bell hat gruene Haare" | grep "B[^ae]ll"
              echo "Der Ball ist rund" | grep "B[^ae]ll"
              echo "Bolle ist auch ok" | grep "B[^ae]ll"
```

Mengen von Zeichen lassen sich auch verkürzt schreiben:

```
    [a-zA-z] beschreibt alle Klein- und alle Großbuchstaben.
        z.B:  [0-9a-z], [0-9_!.?]

    Beispiel:                                                    b8

        echo "Bell hat gruene Haare!" | grep "B[a-z]ll"
        echo "Boll hat gruene Haare!" | grep "B[a-z]ll"
        echo "Bill hat gruene Haare!" | grep "B[a-z]ll"
        echo "BOll hat gruene Haare!" | grep "B[a-z]ll"
        echo "Bell hat gruene Haare," | grep "[!,]$"
        echo "Bell hat gruene Haare!" | grep "[!,]$"
        echo "Bell hat gruene Haare" | grep "[!,]$"
```

Häufigkeiten

? – **Kennzeichnet, das das vorangestellte Zeichen oder ein Zeichen aus der vorangestellten Zeichenklasse ein- oder keinmal in der Zeichenkette auftreten kann.**

Beispiel: **b9, b9a**

```
echo "Bell kann man mit zwei l schreiben" | grep "Bell\>"
echo "Bel kann man auch mit einem l schreiben" | grep "Bell\>"
echo "Belll kann man auch mit 3 l schreiben." | egrep "Bell?\>"
echo "Bell kann man mit zwei l schreiben" | grep "Bell?\>"
echo "Bel kann man auch mit einem l schreiben" | grep "Bell\>"
echo "Belll kann man auch mit 3 l schreiben." | egrep "Bell?\>"
```

+ – **Kennzeichnet, das das vorangestellte Zeichen oder ein Zeichen aus der vorangestellten Zeichenklasse ein- oder mehrmals in der Zeichkette auftreten kann.**

Beispiel: **b10**

```
echo "Bell kann man mit zwei l schreiben" | grep "Bell\>"
echo "Bel kann man auch mit einem l schreiben" | grep "Bell\>"
echo "Belll kann man auch mit 3 l schreiben." | egrep "Bell\>"
echo "Bell kann man mit zwei l schreiben" | grep "Bel+\>"
echo "Bel kann man auch mit einem l schreiben" | grep "Bel+>"
echo "Belll kann man auch mit 3 l schreiben." | egrep "Bel+\>"
```


***** – **Kennzeichnet, das das vorangestellte Zeichen oder ein Zeichen aus der vorangestellten Zeichklasse 0 oder n–mal in der Zeichkette autreten kann.**

Beispiel: **b11**

```
echo "Bell kann man mit zwei l schreiben" | grep "Bell\>"
echo "Bel kann man mit zwei l schreiben" | grep "Bell\>"
echo "Belll kann man auch mit 3 l schreiben." | egrep "Bell\>"
echo "Be kann man auch ohne l schreiben." | egrep "Bell\>"
echo "Bell kann man mit zwei l schreiben" | grep "Bel+\>"
echo "Bel kann man auch mit einem l schreiben" | grep "Bel*>"
echo "Belll kann man auch mit 3 l schreiben." | egrep "Bel*\>"
echo "Be kann man auch ohne l schreiben." | egrep "Bel*\>"
echo "Be kann man auch ohne l schreiben." | egrep "Be[xl]*\>"
echo "Bexx kann man auch mit xx schreiben." | egrep "Be[xl]*\>"
echo "Bex kann man auch mit x schreiben." | egrep "Be[xl]*\>"
```

Gruppierungen

(...) – durch eine Gruppierung kann man mehrer Zeichen zu einer Gruppe zusammenfassen und für diese Gruppe dann weitere Eigenschaften (z.B. Häufigkeiten) festlegen.

Beispiel: b12

```
echo "xxab soll gefunden werden" | egrep "xxab"
echo "xx soll auch gefunden werden" | egrep "xx"
echo "xxab soll gefunden werden" | egrep "xx(ab)?"
echo "xx soll gefunden werden" | egrep "xx(ab)?"
```

Alternativen

| – trennt Suchmuster, bei dem jedes für sich allein die Suchbedingung erfüllt.

Beispiel: b13

```
echo "abxx soll gefunden werden" | egrep "abxx"
echo "abyy soll gefunden werden" | egrep "abyy"
echo "abzz soll gefunden werden" | egrep "abzz"
echo "abxx soll gefunden werden" | egrep "ab(xx|yy|zz)"
echo "abyy soll gefunden werden" | egrep "ab(xx|yy|zz)"
echo "abzz soll gefunden werden" | egrep "ab(xx|yy|zz)"
```


Weitere Funktion von Klammern:

Speichern von "gematchten" Texten

Ein durch runde Klammern eingeschlossener Text kann später im Suchmuster noch- einmal benutzt werden. Diese Rückwärtsreferenzen werden durch '\1', '\2', u.s.w. spezifiziert. Die Klammerpärchen werden durchnummerriert. Maximal 9 Rückwärtsreferenzen möglich.

Beispiel: `\<([a-zA-Z]+)\> +\<\1\>` b14

```
echo "das das ist nicht schön" | egrep '\<([a-zA-Z]+)\> +\<\1\>'
echo "das ist nicht schön" | egrep '\<([a-zA-Z]+)\> +\<\1\>'
echo "ist das alles alles nicht schön" | egrep '\<([a-zA-Z]+)\> +\<\1\>'
echo "ist das alles nicht schön" | egrep '\<([a-zA-Z]+)\> +\<\1\>'
echo "und jetzt etwas weiter etwas" | \
         egrep '\<([a-zA-Z]+)\> +\<([a-zA-Z]+)\> +\<\1\>'
echo 'und jetzt etwas weiter etwas' | \
      egrep '\<([a-zA-Z]+)\> +\<([a-zA-Z]+)\> +\<\2\>'
echo 'und jetzt etwas weiter weiter' | \
      egrep '\<([a-zA-Z]+)\> +\<([a-zA-Z]+)\> +\<\2\>'
echo 'und und jetzt etwas weiter' | \
      egrep '\<([a-zA-Z]+)\> +\<([a-zA-Z]+)\> +\<\2\>'
```

```
1.Zusammenfassung
-----------------
Metazeichen     Bezeichnung                Bedeutung

  Anker
     ^          Zirkumflex, Dach           Zeilenanfang
     $          Dollarzeichen              Zeilenende
     \<         Backslash Kleinerzeichen   Position am Wortanfang
     \>         Backslash Größerzeichen    Position am Wortende

  Zeichenklassen
   .         Punkt                    irgendein Zeichen
   [...]     Zeichenklasse            irgendein Zeichen aus der Menge
   [^...]    verneinte Zeichenklasse  irgendein Zeichen nicht aus der Menge

  Gruppierungen
    |        Pipe                    Alternation
    (...)    runde Klammern          Beschränkung der Reichweite von  "|"

  Häufigkeiten
    ?        Fragezeichen        vorangestellte Zeichen vorhanden oder nicht
    +        Plus                vorangestellte Zeichen ein- oder mehrmals
    *        Stern               0-n Mal das vorangestellte Zeichen

  Wiederholungen
    \1, \2   Rückwärtsreferenzen   vorheriges Auftreten eines () Ausdrucks
```


Wo gibt es was?

Feature	grep	egrep	awk	vi	perl
*,^,$,[...]	x	x	x	x	x
?,+,|	\? \+ \\\|	? + |	? + |	\? \+	? + |
\(...\)	x	(...)	(...)	x	(...)
\<,\>	–	x	–	–	x
Rückwärtsreferenz	x	–	–	x	x

Einige einfache Aufgaben:

```
   1. Entfernen aller Leerzeilen aus einem Text.

         1. Versuch
                sed  '/^$/d'   file
                   Problem: Leerzeichen
         2. Versuch
                sed  '/^ $/d'  file                          m1,m1a,m1b,m1c
                   Problem: mehrere Leerzeichen, kein Leerzeichen
         3. Versuch
                sed  '/^ *$/d' file
                   Problem: Tabulatoren
         4. Versuch
                sed '/^[       ]*$/d' file
                     <Leerzeichen><Tabulator>
                                                             m2,m2a,m2b,m2c
```

2. Ein etwas exotischer regulärer Ausdruck zum Suchen
   von zwei aufeinanderfolgenden Zeichen (XX):

```
/X(.+)*X/
```

```
#!/bin/sh
echo "Eingabe: =XX=================================="
echo "    Test fuer egrep"
echo  egrep "X(.+)+X"
echo "=XX==================================" | egrep "X(.+)+X"
echo  egrep "X(.+)*X"
echo "=XX==================================" | egrep "X(.+)*X"
echo "     Test fuer awk "
echo  awk "/X(.+)+X/{ print }"
echo "=XX==================================" | awk "/X(.+)+X/{ print }
echo  awk "/X(.+)*X/{ print }"
echo "=XX==================================" | awk "/X(.+)*X/{ print }
echo  "    Test fuer sed"
echo  sed "s/X(.+)+X//"
echo "=XX==================================" | sed "s/X(.+)+X//"
echo  sed "s/X(.+)*X//"
echo "=XX==================================" | sed "s/X(.+)*X//"
```

Wir sehen: Reguläre Ausdrücke sind nicht gleich reguläre Ausdrücke!!!!!
machine


```
Weitere Metazeichen
-------------------

  Anker
     \w      - Wortbestandteil: Klasse [a-zA-Z0-9] oder [a-zA-Z0-9_]
     \W      - Nicht-Wortbestandteil: Klasse [^a-zA-Z0-9] oder [^a-zA-Z0-9_]


Abkürzungsmetazeichen
---------------------

     \x<hex>  -   Hexadezimal-Escapes: \x00..\xff
     \<okt>   -   Oktal-Escapes:  \000..\377
        \a    -   Alarm: 007
        \b    -   Backspace: 010   oder Metazeichen für Wortgrenze
        \d    -   Ziffer: [0-9]
        \D    -   nicht Ziffer: [^0-9]
        \e    -   Escape: 010
        \f    -   FormFeed - Seitenvorschub:  014
        \n    -   NewLine - Zeilenvorschug:   012
        \r    -   Wagenrücklauf: 015
        \t    -   Horizontaltabulator:   011
        \v    -   Vertikaltabulator: 013
        \s    -   Whitespace-Zeichen: [ \f\b\r\t\v]
        \S    -   kein Whitespace-Zeichen: [^ \f\b\r\t\v]
```

Zeichenklassen

```
   [:alnum:]   -  Alphanumerische Zeichen oder Ziffern
   [:alpha:]   -  Alphanumerische Zeichen
   [:blank:]   -  Leerzeichen oder Tabulator
   [:cntrl:]   -  Control-Zeichen
   [:digital:] -  Ziffern
   [:graph:]   -  Graphische Zeichen
   [:lower:]   -  Kleinbuchstaben
   [:print:]   -  druckbare Zeichen
   [:punct:]   -  Satzzeichen
   [:space:]   -  Whitespace, Tabulatoren, Leerzeichen
   [:upper:]   -  Großbuchstaben
   [:xdigit:]  -  Hexadezimalziffern
```

bei einigen Programmen auch als: `[[: ....  :]]` kodiert!!!

Neue Häufigkeiten: Intervalle

```
{min,max} oder \{min,max\}
{genau} oder \{genau \}
```

m3

Probleme:

```
 - Gierigkeit von: .+ und .*

   HTML-Text:
     <B>fetter Text</B>nicht fetter Text <B> fetter Text </B>
    Ausdruck bilden der <B>fetter Text</B> "matched"??
          '<B>.*</B>' tut es nicht!!! nicht trivial,
     .* ist gierig und geht bis zum letzten Zeichen und dann rückwärts
        (Backtracking).

 - Reichweite von Klammern:

       ".*"    -  nichts
      (".*")   -  Strings einschließlich "
      "(.*)"   -  Text zwischen den Anführungszeichen
      "(.)*"   -  Letztes Zeichen des Textes
```

m4,m4a,m4b