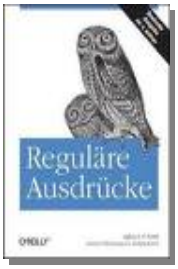
# Info-Blatt zu "Reguläre Ausdrücke"




Quelle: http://openbook.galileocomputing.de/linux/linux_04_regulaere_ausdruecke_005.htm

http://wiki.selfhtml.org/wiki/Perl/Regul%C3%A4re_Ausdr%C3%BCcke

Nun kommen wir zu drei sehr populären und mächtigen Tools: sed, grep und awk. Um mit diesen Tools umgehen zu können, muss man erst einmal sogenannte reguläre Ausdrücke (englisch ***»regular expressions«***) verstehen.

Reguläre Ausdrücke – es gibt übrigens ganze Bücher zu diesem Thema – dienen in der Shell zum Filtern von Zeichenketten (Strings) aus einem Input, etwa einer Textdatei. Am besten lässt sich das an einem Beispiel verdeutlichen.

Das Programm grep filtert aus einem Input (Pipe, Datei) Zeilen heraus, in denen ein bestimmtes Muster vorkommt. Gegeben sei eine Datei mit den Namen von Städten, wobei jede Stadt in einer separaten Zeile steht. grep soll nun all jene Zeilen herausfiltern, in denen ein kleines »a« enthalten ist.

```
$ cat Standorte
Augsburg
Bremen
Friedrichshafen
Aschersleben
Bernburg
Berlin
Halle
Essen
Furtwangen
Kehlen
Krumbach
Osnabrueck
Kempten
// Nun werden alle Orte, die ein 'a' enthalten gefiltert:
$ grep a Standorte
Friedrichshafen
Halle
Furtwangen
Krumbach
Osnabrueck
```

*Text 1: grep filtert alle Zeilen mit einem »a« heraus*


ie Sie sehen, wurden tatsächlich nur die Zeilen ausgegeben, in denen das Zeichen »a« vorkam. Dies können Sie mit jedem Zeichen und sogar mit ganzen Strings durchführen – hier ein paar Beispiele:

```
$ grep b Standorte     // filtert nach 'b'
$ grep B Standorte     // filtert nach 'B'
$ grep hafen Standorte // filtert nach 'hafen'
```

*Text 2: Weitere Beispiele für reguläre Ausdrücke*

Reguläre Ausdrücke sind case-sensitive. Dass bedeutet, es wird zwischen Groß- und Kleinbuchstaben unterschieden. Nun zurück zur eigentlichen Definition regulärer Ausdrücke: Mit ihnen können Sie Muster für solche Filtervorgänge, wie sie gerade gezeigt wurden, angeben. Allerdings können mithilfe dieser regulären Ausdrücke nicht nur explizit angegebene Strings, etwa »hafen«, gefiltert werden. Nein, dies funktioniert auch dynamisch. So können Sie angeben, dass »hafen« am Zeilenende oder -anfang vorkommen kann, dass das zweite Zeichen ein »a«, aber auch ein »x« sein kann, dass das letzte Zeichen entweder klein- oder groß geschrieben werden darf und so weiter.

Sollen beispielsweise alle Zeilen, die auf »n« oder »g« enden, ausgegeben werden, kann der reguläre Ausdruck ***»[ng]$*** verwendet werden:***[Fn***. Keine Angst, dies ist nur ein Beispiel – gleich lernen Sie, wie solche Ausdrücke zu lesen und zu schreiben sind.

```
$ grep "[ng]$" Standorte
Augsburg
Bremen
Friedrichshafen
Aschersleben
Bernburg
Berlin
Essen
Furtwangen
Kehlen
Kempten
```

*Text 3: Ein erstes Beispiel für einen dymanischen regulären Ausdruck*

# Der Aufbau regulärer Ausdrücke, oder - wie man diese schreibt!

Nach dieser kleinen Einleitung werden wir uns nun den regulären Ausdrücken selbst zuwenden. Im Folgenden werden Sie lernen, wie solche Ausdrücke, die übrigens recht oft vorzufinden sind, zu verstehen sind und wie Sie selbst solche Ausdrücke formulieren können. Keine Sorge – so schwer ist das nicht.

### *Erst exakt*

Eben wandten wir den regulären Ausdruck »[ng]$« an. Wie ist dieser zu verstehen? Das Dollarzeichen ($) steht für das Ende einer Zeile. Vor diesem Dollarzeichen sind in eckigen Klammern zwei Zeichen (»ng«) gesetzt. Das bedeutet, dass diese zwei Zeichen (und kein anderes) am Zeilenende stehen können.

Und zwar kann nur genau eines der beiden Zeichen das letzte Zeichen der Zeile ausmachen. Würden Sie in der Klammer also beispielsweise noch ein »h« hinzufügen, so könnte auch dieses »h« das letzte Zeichen in der Zeile sein.

### *Jetzt einfach*

Lesen würde man den Ausdruck so: Das letzte Zeichen der Zeile ($) kann entweder ein »n« oder ein »g« sein ([ng]). Reguläre Ausdrücke können sich aus mehreren solcher Muster zusammensetzen. Sie können beispielsweise das Zeichen, das vor dem letzten Zeichen einer Zeile steht, auch noch festlegen und so weiter. Hier ist eine Auflistung der möglichen Filterausdrücke mit Beispielen:

- ***abc*** der String »abc«
- ***[xyz]*** Eines der Zeichen in der eckigen Klammer muss (an der jeweiligen Stelle) vorkommen.
- ***[aA]bc*** entweder »Abc« oder »abc«
- ***[a-b]*** Mit dem Minus-Operator werden Zeichenbereiche für eine Position festgelegt.
- ***[a-zA-Z0-9bc]*** In diesem Beispiel werden alle kleinen und großen Zeichen des Alphabets und alle Ziffern von 0 bis 9 akzeptiert, worauf die Zeichen »bc« folgen müssen.
- ***[^a-b]*** Das Dach-Zeichen (^) negiert die Angabe. Dies funktioniert sowohl mit dem als auch ohne den Minus-Operator. Dieser Ausdruck ist also dann erfüllt, wenn an der entsprechenden Stelle ein Zeichen steht, das nicht »a« oder »b« ist.
- ***[xyz]*** Der Stern ist das Joker-Zeichen (*) und steht für eine beliebige Anzahl von Vorkommen eines Zeichens.
- ***K[a*tze]*** würde beispielsweise sowohl »Kaaatze«, »Katze« und »Ktze« herausfiltern.
- ***[xyz]\+*** Das Plus-Zeichen (+) steht für eine beliebige Anzahl von Vorkommen eines Zeichens. Im Gegensatz zum Stern muss das Zeichen allerdings mindestens einmal vorkommen.
- ***K[a\+tze]*** würde beispielsweise sowohl »Kaaatze«, »Ktze« und »Katze« herausfiltern.
- ***$*** Dieses Zeichen steht für das Zeilenende.
- hafen***$*** Die letzten Zeichen der Zeile müssen »hafen« sein.
- **^** Dieses Zeichen steht für den Zeilenanfang und ist nicht mit der Negierung (die in eckigen Klammern steht) zu verwechseln.
- **^** Friedrichs Die ersten Zeichen der Zeile müssen »Friedrichs« sein.
- . Der Punkt steht für ein beliebiges Zeichen.
- Friedr.chshafen In der Zeile muss die Zeichenkette »Friedr« enthalten sein. Darauf muss ein beliebiges Zeichen (aber kein Zeilenende) folgen und darauf die Zeichenkette »chshafen«.
- ***\x*** Das Metazeichen »x« wird durch den Backslash nicht als Anweisung im regulären Ausdruck, sondern als bloßes Zeichen interpretiert. Metazeichen sind die folgenden Zeichen: **\\ ^ $ . [ ]**

Auch sind Kombinationen aus solchen Filterausdrücken möglich – hier ein Beispiel: Um festzulegen, dass eine Zeile mit einem kleinen oder großen »H« anfangen soll und dass darauf die Zeichen »alle« und das Zeilenende folgen sollen, wäre dieser Ausdruck der richtige: **^[hH]alle$**

Vergessen Sie nicht, die regulären Ausdrücke in Anführungszeichen oder Hochkommas zu stellen. Wenn Sie diese weglassen, wird die Shell diese Zeichen anders interpretieren und der Ausdruck wird verfälscht. Um einen regulären Ausdruck auch ohne Anführungszeichen verwenden zu können, müssen Sie alle sogenannten Metazeichen »escapen«. Das bedeutet, dass den Zeichen
**\ \^{ $ . [ ]** } ein Backslash (\) vorangestellt werden muss.

Der obige Ausdruck müsste somit folgendermaßen aussehen:

---

# Der Stream-Editor: "sed"

In diesem Unterkapitel soll es nicht um die ehemalige DDR-Partei, sondern um einen mächtigen Editor

gehen, der zum Standardumfang eines jeden Unix-Systems gehört. sed ist kein herkömmlicher Editor, wie man ihn etwa von einer grafischen Oberfläche, ncurses-basiert oder dem vi ähnelnd, kennt. sed verfügt über keinerlei Oberfläche, nicht einmal über die Möglichkeit, während der Laufzeit interaktiv eine Editierung vorzunehmen.

```
$ grep "^[hH]alle$" Standorte
Halle
```
*Text 4: Suche Stadtnamen*

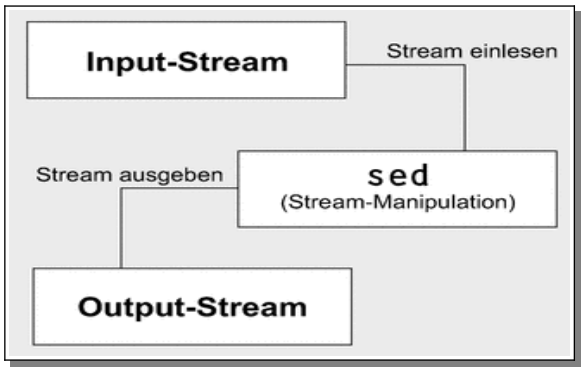
*Abb. 1 Die Arbeitsweise von sed*

Der Nutzen von sed ist auch nicht ganz mit dem eines »normalen« Editors gleichzusetzen. Die Aufgabe von sed ist die auto matische Manipulation von Text-Streams. Ein Text-Stream ist nichts anderes als ein »Strom von Zeichen«. Dieser kann sowohl direkt aus einer Datei als auch aus einer Pipe kommen. Dabei liest sed den Stream zeilenweise ein und manipuliert Zeile für Zeile nach dem Muster, das ihm vom Anwender vorgegebenen wurde.

```
\\^\\[hH\\]alle$
```
*Text 5: Reuglärer Ausdruck mit Escape-Sequenzen*

## Was bringt mir sed?

Der Editor sed ist oftmals dann von hohem Nutzen, wenn es darum geht, ständig wiederkehrende Daten anzupassen beziehungsweise zu manipulieren. Dies ist besonders in Shellskripten und in der Systemadministration oft der Fall. sed nimmt daher neben awk eine dominante Rolle in der Stream-Manipulation ein. Im Gegensatz zu awk bietet sed keine so guten Möglichkeiten in der Programmierung, dafür ist sed oftmals einfacher und schneller einsetzbar, was die bloße Manipulation von Streams anbelangt. Diese Manipulation ist (ähnlich zu awk) besonders durch die Möglichkeit, reguläre Ausdrücke zu verwenden, äußerst einfach und dynamisch zu handhaben.




## Erste Schritte mit sed \ \^{ $ . [ ] }

```
sed [Option] [Skript]
[Eingabedatei]
```
*Text 6: sed-Aufrufen*

Der Aufruf von sed erfolgt durch Angabe einer Manipulationsanweisung (entweder direkt in der Kommandozeile oder durch eine Skriptdatei, die diese Anweisung(en) enthält). Dazu übergibt man entweder eine Datei, die als Eingabequelle dienen soll, oder tut eben dies nicht, woraufhin sed von der Standardeingabe liest.

Sofern sed ohne Eingabedatei betrieben wird, muss das Programm manuell durch die Tastenkombination Strg+D beendet werden. Auf dem Bildschirm erscheint dann die Ausgabe ^D.

Zunächst einmal gibt sed standardmäßig alle Zeilen nach der Bearbeitung aus. Um dies zu veranschaulichen, verwenden wir einen sehr einfachen sed-Aufruf. Dabei wird die Anweisung 'p' verwendet. Diese besagt in diesem Fall nichts weiter, als dass alle Zeilen ausgegeben werden sollen. Doch wie Sie sehen ...

```
user$ sed 'p'
Hallo, Sed!     # dies ist die manuelle Eingabe
Hallo, Sed!     # dies bewirkt 'p'
Hallo, Sed!     # dies ist die standardmäßige Ausgabe
^D              # sed wird durch Strg+D beendet
```
*Text 7: sed 'p'*

```
user$ sed -n 'p'
Hallo, Sed!
Hallo, Sed!
<STGR>+<D> = Ende der
             Verarbeitung
```
*Text 8: sed -n 'p'*

... gibt sed unsere eingegebene Zeile zweimal aus. Die erste Ausgabe kam durch den Befehl 'p' zustande, die zweite durch die standardmäßige Ausgabe jeder manipulierten Zeile. Um dieses, manchmal unerwünschte Feature zu deaktivieren, muss die Option -n verwendet werden:

```
user$ sed -n 'p' /etc/passwd
root:x:0:0::/root:/bin/zsh
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/log:
...
```
*Text 9: sed mit Eingabedatei*

### *sed mit Eingabedatei*

Verwendet man nun eine Datei als Stream-Quelle, muss diese nur zusätzlich beim Aufruf angegeben werden. Da Sie bereits den Befehl 'p' kennen, werden Sie vielleicht schon auf die Idee gekommen sein, sed einmal als cat-Ersatz zu verwenden. cat gibt den Inhalt einer Datei auf dem Bildschirm aus. sed tut dies bei alleiniger Verwendung des 'p'-Befehls ebenfalls.

### *sed-Befehle*

Beschäftigen wir uns nun mit den Befehlen, die uns in sed zur

```
$ sed -n '/[Ff]/p' Standorte
```
*Text 10: Verwendung von Slashes*

Verfügung stehen. Mit dem ersten Befehl 'p' sind Sie bereits in Kontakt gekommen. Dieser Befehl gibt die Zeilen aus.

Doch wie wendet man nun einen regulären Ausdruck auf solch einen Befehl an? Die Antwort ist recht simpel: Man schreibt den regulären Ausdruck in zwei Slashes und den Befehl (je nach Befehl) vor oder hinter die Slashes. Alles zusammen setzt man der Einfachheit halber in Hochkommas, damit die Metazeichen nicht anderweitig von der Shell interpretiert werden – etwa so:

Dieser Ausdruck würde nun alle Zeilen herausfiltern, in denen ein großes oder kleines »f« enthalten ist, und diese anschließend auf der Standardausgabe ausgeben.

```
user$ sed '/F/x' Standorte
Augsburg
Bremen
Aschersleben
Bernburg
Berlin
Halle
Essen
Friedrichshafen
Kehlen
...
```

*Text 11: Austausch von Hold- u. Patternspace*

### *Hold- und Patternspace*

Zur internen Funktionsweise von sed ist noch anzumerken, dass das Programm mit zwei Puffern arbeitet, in denen die Zeilen gespeichert werden. Der erste Puffer ist der Patternspace. In diesen wird eine Zeile geladen, nachdem sie einem Muster entsprach. Der zweite Puffer ist der Holdspace. Nach der Bearbeitung einer Zeile wird diese vom Pattern- in den Holdspace kopiert. Hier ein Beispiel:

Der Befehl x tauscht den Inhalt des Patterns mit dem des Holdspace. Dies geschieht jedes Mal, wenn eine Zeile ein großes »F« enthält, da wir dies als Muster angegeben haben.

Wie Sie sehen, folgt nach »Bremen« eine Leerzeile. Dort war der Holdspace noch leer. Da dieser aber mit dem Patternspace (der »Friedrichshafen« enthielt) vertauscht wurde, wurde eine leere Zeile ausgegeben. Nachdem die Leerzeile ausgegeben wurde, befindet sich nun also »Friedrichshafen« im Holdspace. Später wird »Friedrichshafen« ausgegeben, obwohl »Furtwangen« im Patternspace enthalten war. Dies liegt auch wieder daran, dass durch den x-Befehl der Pattern- und der Holdspace vertauscht wurden. Doch es gibt noch einige weitere sed-Befehle, die folgende Tabelle zeigt die wichtigsten.

| Befehl | Auswirkung |
|---|---|
| /ausdruck/= | Gibt die Nummern der gefundenen Zeilen aus. |
| /ausdruck/a\string | Hängt »string« hinter (append) die Zeile an, in der »ausdruck« gefunden wird. |
| b label | Springt zum Punkt »label« im sed-Skript. Falls »label« nicht existiert, wird zum Skriptende gesprungen. |
| /ausdruck/c\string | Ersetzt die gefundenen Zeilen durch »string« (change). |
| /ausdruck/d | Löscht die Zeilen, in denen »ausdruck« gefunden wird (delete). |
| /ausdruck/D | Löscht Patternspace bis zum ersten eingebundenen Zeilenumbruch (\n) in »ausdruck«. Sofern weitere Daten im Patternspace vorhanden sind, werden diese übersprungen. |
| /ausdruck/i\string | Setzt »string« vor der Zeile ein, in der »ausdruck« gefunden wird (insert). |
| /ausdruck/p | Gibt die gefundenen Zeilen aus (print). |
| /ausdruck/q | Beendet sed, nachdem »ausdruck« gefunden wurde (quit). |
| /ausdruck/r datei | Hängt hinter »ausdruck« den Inhalt der Datei »datei« an (read from file). |
| s/ausdruck/string/ | Ersetzt »ausdruck« durch »string«. Ein Beispiel zu diesem Befehl folgt weiter unten. |
| t label | Falls seit dem letzten Lesen einer Zeile oder der Ausführung eines t-Befehls eine Substitution (s-Befehl) stattfand, wird zu »label« gesprungen. Lässt man »label« weg, wird zum Skriptende gesprungen. |

| Befehl | Auswirkung |
|---|---|
| /ausdruck/w datei | Schreibt den Patternspace in die Datei »datei«. Ein Beispiel zu diesem Befehl folgt weiter unten. |
| /ausdruck/x | Tauscht den Inhalt des Haltepuffers mit dem des Eingabepuffers. Eine Beispielanwendung dieses Befehls finden Sie in Listing, das der Tabelle voranging. |
| y/string1/string2/ | Vertauscht alle Zeichen, die in »string1« vorkommen, mit denen, die in »string2« angegeben sind, wobei die Positionen der Zeichen entscheidend sind: Das zweite Zeichen in »string1« wird durch das zweite in »string2« ersetzt usw. Ein Beispiel zu diesem Befehl folgt weiter unten. |

Tabelle T1: Anwendung von SED

Es folgen nun einige Listings zur exemplarischen Anwendung von sed. Zunächst soll der Befehl w angewandt werden, der die gefundenen Ausdrücke in eine Datei schreibt. Um alle anderen Ausdrücke zu unterdrücken, wird die Option -n verwendet. Es sollen dabei alle Zeilen, in denen ein »F« vorkommt, in die Datei out.log geschrieben werden.

```
$ sed -n '/F/w out.log' Standorte
$ cat out.log
Friedrichshafen
Furtwangen
```
*Text 12: w-Befehl*

### *s-Befehl*

Aber auch die Substitution von Ausdrücken ist in sed kein Problem. Mit dem Befehl s kann ohne Weiteres ein Ausdruck durch einen String ersetzt werden. Im Folgenden sollen alle »n«-Zeichen durch den String »123456« ersetzt werden.

(alternativ tr-Befehl)

```
$ sed 's/n/123456/' Standorte
Augsburg
Breme123456
Friedrichshafe123456
Aschersleb123456e
Ber123456burg
Berli123456
Halle
Esse123456
Furtwa123456gen
Kehle123456
Krumbach
Os123456abrueck
Kempte123456
```
*Text 13: s-Befehl*

### *y-Befehl*

Eine ähnliche Funktionalität bietet der y-Befehl. Dieser Befehl ersetzt keine ganzen Strings, sondern einzelne Zeichen. Er sucht zudem nicht nach Ausdrücken, sondern explizit nach den ihm angegebenen Zeichen. Dem Befehl werden zwei Zeichenketten übergeben. In der ersten steht an jeder Stelle ein Zeichen, das im zweiten String an derselben Stelle ein Ersatzzeichen findet. Daher müssen beide Strings gleich lang sein. Im nächsten Listing soll »a« durch ein »i«, »b« durch ein »j« ... und »h« durch ein »p« ersetzt werden.

```
$ sed 'y/abcdefgh/ijklmnop/' Standorte
Auosjuro
Brmmmn
Frimlrikpspinmn
Askpmrslmjmn
Bmrnjuro
Bmrlin
Hillm
Essmn
Furtwinomn
Kmplmn
Krumjikp
Osnijrumkk
Kmmptmn
```
*Text 14: Y-Befehl*

```
$ cat myfile
Zeile1
Zeile2
Zeile3
Zeile4
Zeile5
Zeile6
```
*Text 15: myfile*

## *Nach Zeilen filtern*

Ein weiteres Feature von sed ist die Möglichkeit, nach bestimmten Zeilen zu filtern. Dabei kann entweder explizit eine Zeile oder ein ganzer Zeilenbereich angegeben werden. Gegeben sei die Datei myfile mit dem folgenden Inhalt:

### *Einzelzeilen*

Eine Einzelzeile kann durch Angabe der Zeilennummer in Verbindung mit dem p-Befehl herausgefiltert werden. Durch die Option -e können noch weitere Einzelzeilen in einem

```
$ sed -n '2p' myfile
Zeile2
$ sed -n -e '1p' -e '2p' myfile
Zeile1
Zeile2
$
```
*Text 16: Einzelen-Filter*

einzelnen Aufruf von sed gefiltert werden.

```
$ sed -n '3,$p' myfile
Zeile3
Zeile4
Zeile5
Zeile6
$ sed -n '$p' myfile
Zeile6
```
*Text 17: Das Zeichen $*

### *Zeilenbereiche*

Um nun nach Zeilenbereichen zu filtern, gibt man die beiden Zeilennummern, die diesen Bereich begrenzen, durch ein Komma getrennt an.

Das Dollarzeichen steht dabei symbolisch für das Zeilenende.

### *Wiederholungen in regulären Ausdrücken*

Kommen wir nun zu einem weiteren Feature in Bezug auf die regulären Ausdrücke für sed: das n-fache Vorkommen eines Ausdrucks. Auch für dieses Thema verwenden wir wieder eine Beispieldatei wh (Wiederholung) mit folgendem Inhalt:

```
Ktze
Katze
Kaatze
Katatze
Katatatze
```
*Text 18: Die Datei wh*

```
$ sed -n '/Ka*tze/p' wh
Ktze
Katze
Kaatze
$ sed -n '/Kaa*tze/p' wh
Katze
Kaatze
```
*Text 19: Anwendung des *-Operators*

### *Einzelzeichen*

Das mehrmalige Vorkommen von Einzelzeichen kann durch den Stern-Operator (*), den wir Ihnen bereits vorgestellt haben, festgestellt werden. Er kann in einen regulären Ausdruck eingebaut werden und bezieht sich auf das ihm vorangestellte Zeichen. Dabei kann das Zeichen keinmal, einmal oder beliebig oft vorkommen.

### *Ganze Ausdrücke*

Es ist nicht nur möglich, einzelne Zeichen, sondern auch ganze Ausdrücke beliebig oft vorkommen zu lassen. Dabei wird der jeweilige Ausdruck in Klammern geschrieben (die »escaped« werden müssen).

```
$ sed -n '/\(at\)*/p' wh
Ktze
Katze
Kaatze
Katatze
Katatatze
```
*Text 20: Der Operator ()*

### *{n} Vorkommen*

Möchte man hingegen die Anzahl der Vorkommen eines Zeichens oder eines Ausdrucks festlegen, so muss man diese Anzahl in geschweifte Klammern hinter den jeweiligen Ausdruck schreiben. Dabei ist zu beachten, dass auch die geschweiften Klammern als Escape-Sequenzen geschrieben werden. Im nächsten Listing muss der Ausdruck at zweimal hintereinander vorkommen:

```
$ sed -n '/\(at\)\{2\}/p' wh
Katatze
Katatatze
```
*Text 21: Mehrmalige Vorkommen mit dem {}-Operator angeben*

```
$ grep 'n$' Standorte
Bremen
Friedrichshafen
Aschersleben
Berlin
Essen
Furtwangen
Kehlen
Kempten
```
*Text 22: grep in Aktion*

### *grep*

Kommen wir nun zu einem weiteren Programm namens grep. Mit grep können Sie ähnlich wie mit sed Filterausdrücke aus einem Input-Stream filtern. Jedoch kann grep diese nicht manipulieren. Vielmehr liegen die Stärken von grep in der einfachen Handhabung und in der höheren Geschwindigkeit gegenüber sed. Zum Filtern von Ausdrücken übergibt man grep einfach den gewünschten Ausdruck sowie entweder eine Eingabedatei oder den Input aus einer Pipe bzw. die Tastatur.

### *Filternegierung*

Zudem kann man die Filtervorgabe negieren, womit grep alle Zeilen ausgibt, die nicht dem angegebenen Ausdruck entsprechen. Dies wird mit der -v-Option bewerkstelligt.

```
$ grep -v 'n$'
Standorte
Augsburg
Bernburg
Halle
Krumbach
Osnabrueck
```
*Text 23: grep -v (=invers)*

```
$ egrep -v 'n$|k$'
Standorte
Augsburg
Bernburg
Halle
Krumbach
$ grep -vE 'n$|k$'
Standorte
Augsburg
Bernburg
Halle
Krumbach
```
*Text 24: egrep in Aktion*

### *grep -E und egrep*

Sehr hilfreich ist die Fähigkeit, mehrere Ausdrücke in einem Befehl zu filtern. Dabei verwendet man ein logisches ODER in Form eines Pipe-Zeichens zwischen den Ausdrücken sowie entweder grep mit der Option -E oder das Programm egrep.

Ein Blick in die Manpage verrät uns das Geheimnis: egrep ist mit einem Aufruf von grep -E gleichzusetzen. Zudem findet man im Dateisystem, zumindest unter Slackware-Linux, den symbolischen Link /bin/egrep auf /bin/grep. Dies bedeutet, dass das Programm grep intern abfragt, ob der Programmname egrep oder nur grep lautet, und sein Verhalten der Option -E im Falle von egrep automatisch anpasst.

## Geschwindigkeitsvergleich

Da wir einen Test auf einen regulären Ausdruck sowohl mit sed als auch mit grep durchführen können, interessiert uns natürlich, welches Programm das schnellere ist. Besonders Shellskripts, die große String-Mengen durchsehen müssen, können dadurch eventuell sinnvoll optimiert werden.
Zum Test erzeugen wir eine 188 MB große Datei mit dem Namen »TESTFILEB«, in der unterschiedlich lange Textstrings enthalten sind. Das Testsystem läuft unter Slackware-Linux 9.1 mit Kernel 2.4.22, einem AMD Athlon XP 2400+-Prozessor und einer UDMA133-Platte. Hier nun die Testaufrufe sowie deren Ergebnisse:

- sed -n '/n$/p' TESTFILEB >/dev/null
  benötigte im Schnitt 9,358 Sekunden, um diese Datenmenge zu bewältigen.
- grep 'n$' TESTFILEB >/dev/null
  benötigte durchschnittlich nur 7,075 Sekunden.
- Ein von uns speziell für diesen einen Filtervorgang entwickeltes, geschwindigkeitsoptimiertes, vom GNU-Compiler gcc-3.2.3 optimiertes, gestriptes C-Programm, in dem die darin verwendeten Standard-Libc-Funktionen strlen() und bzero() durch schnellere ersetzt wurden, benötigte übrigens nur noch 5,940 Sekunden.[Fn. Man könnte den Test noch schneller absolvieren, indem man beispielsweise auf Assembler zurückgreift, die Testdatei in eine virtuelle Partition im Hauptspeicher auslagert (oder komplett in den RAM einliest), eventuell eine andere Kernel-Version verwendet oder schlicht auf bessere Hardware zurückgreift.]

## Exkurs: PDF-Files mit grep durchsuchen

Es ist möglich, mit Hilfe der ***»poppler-utils«*** (auch ***»poppler-tools«*** genannt), den Inhalt von PDF-Dateien in Textform auszugeben. Diese Textform kann dann wiederum mit Programmen wir grep durchsucht werden. Die poppler-utils stellen dazu das Programm pdftotext bereit. Übergeben wird dem Programm dabei zunächst ein Dateiname und als zweiter Parameter die Ausgabedatei oder ein »-« um zu signalisieren, dass der Inhalt der Datei auf die Standardausgabe geschrieben werden soll.

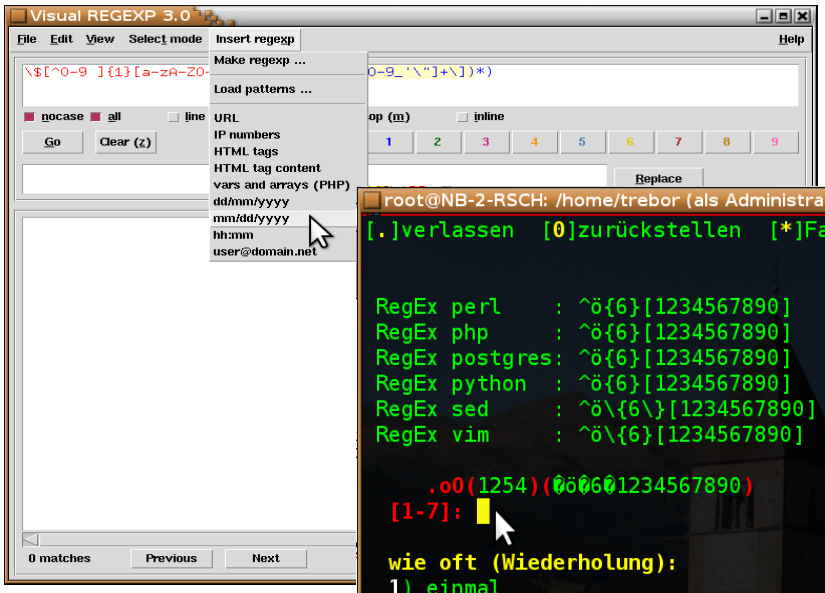
```
$ pdftotext CovertChannels.pdf - | grep portknocker
keywords : covert, channels, ... portknocker
It seems obvious that t... portknocker or ...
…
```
*Text 25: Eine PDF-Datei durchsuchen (Ausgabe gekürzt)*

```
$ pdftohtml essay.pdf essay.html
Page-1
Page-2   …
```
*Text 26: Eine PDF-Datei in HTML konvertieren*

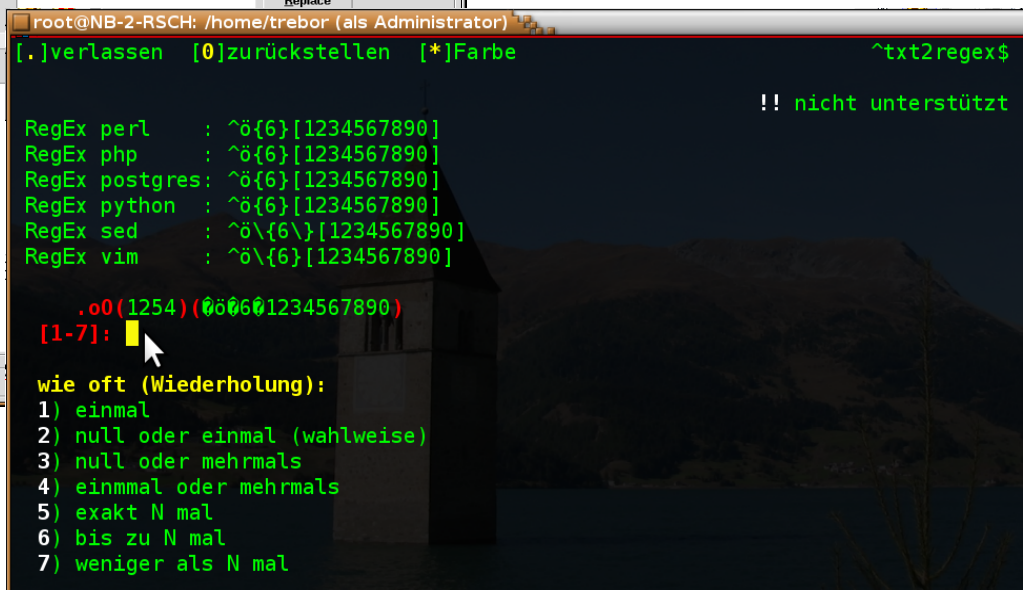
Die poppler-utils beinhalten übrigens auch einige weitere Programme wie etwa ***pdftohtml***, mit dem der Inhalt von PDF-Dateien in HTML umgewandelt werden kann. Mit ***pdftops*** können die Dateien hingegen zum Postscript-Format konvertiert werden..

# Die Anwendung von regexxer, visual-regexp oder txt2regex:

echo 1. | egrep "^(([0]{0,1}[1-9])|([12][0-9])|([3][0-1]))[.]"

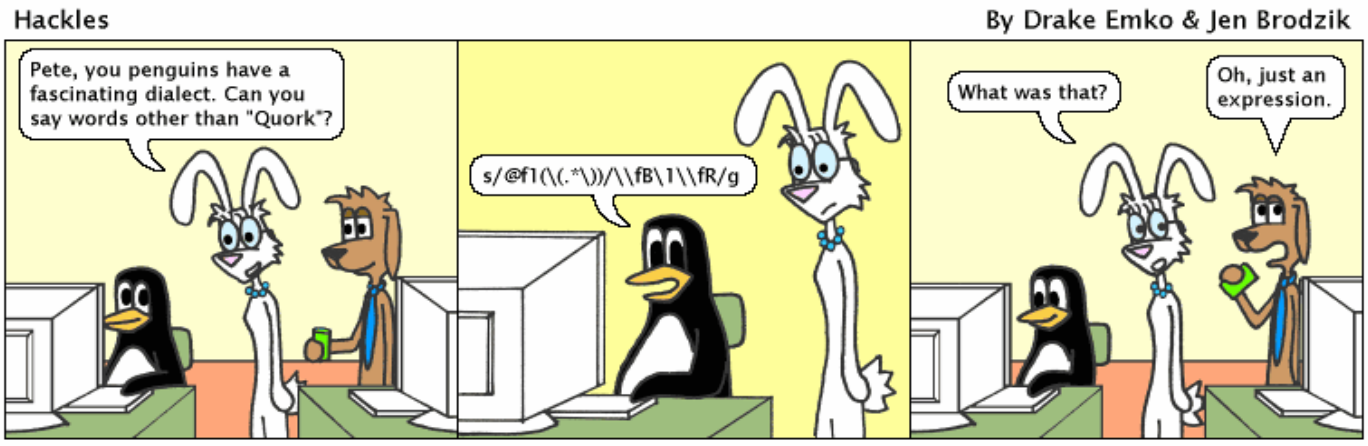



***visual-regexp*** in Aktion

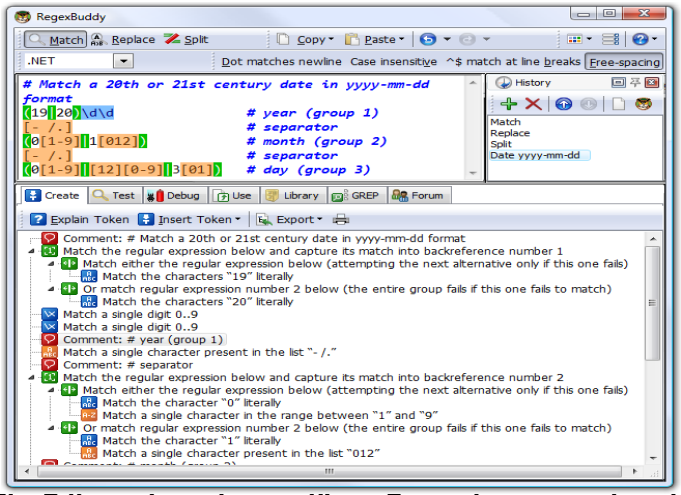



***txt2regex*** in Aktion


***regexxer*** durchsucht Dateien nach regulären Ausdrücken und ersetzt diese durch neuen Text.

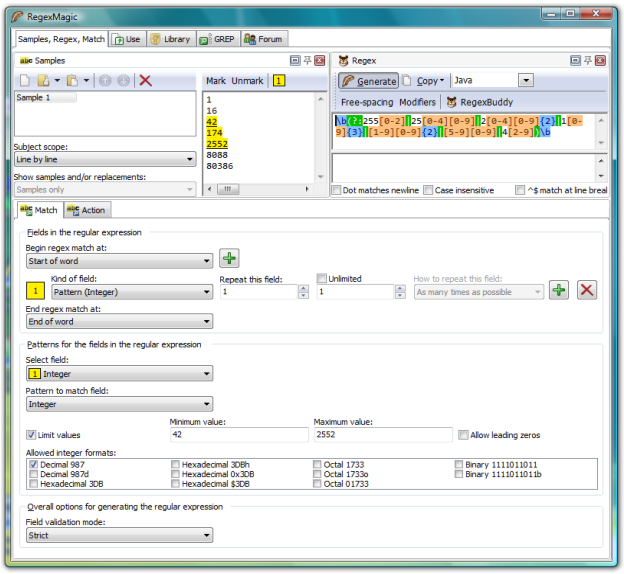


***RegexBuddy:***

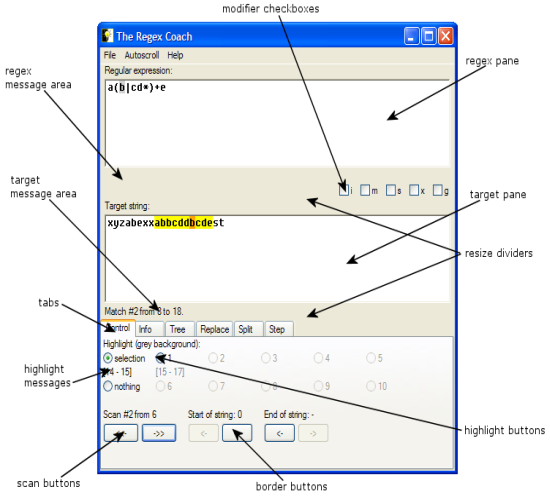



http://www.regular-expressions.info/regexbuddy.html

***regexmagic:***
http://www.regular-expressions.info/regexmagic.html

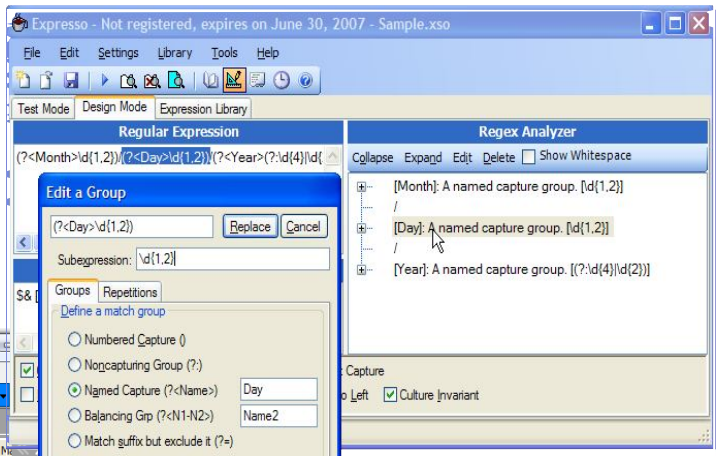


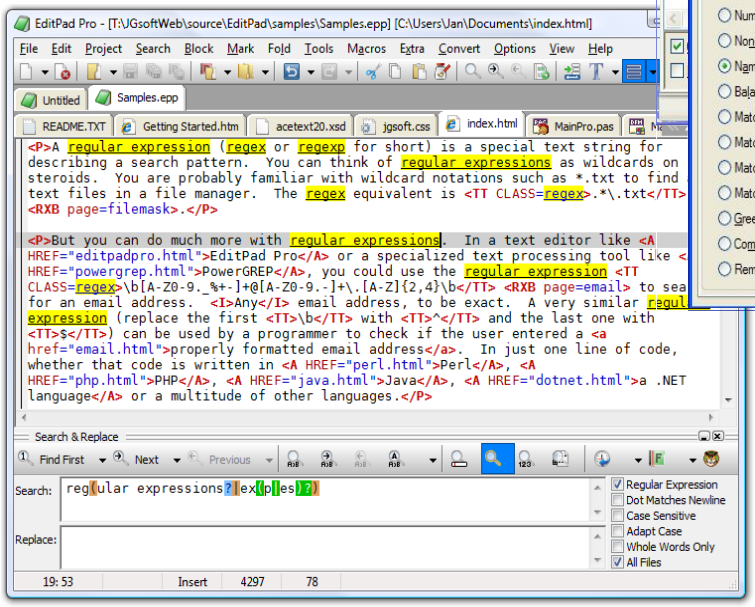

**Ein Editor, der mit regulären Expresions umgehen kann:**
http://www.regular-expressions.info/editpadpro.html




http://cutesoft.net/Downloads/

***EditPad Pro***




***"The Tegex Coach"***

***zu finden bei:***
http://www.ultrapico.com/Expresso.htm

http://www.heise.de/software/download/the_regex_coach/14569 oder http://weitz.de/regex-coach/

*Ein Link-Sammlung:*

1. http://de.linwiki.org/wiki/Linuxfibel_-_Unix-Werkzeuge_-_Regul%C3%A4re_Ausdr%C3%BCcke
2. http://www.linux-user.de/ausgabe/2001/02/049-pcorner/pcorner3.html
3. http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/
4. http://openbook.galileocomputing.de/linux/linux_04_regulaere_ausdruecke_002.htm
5. http://forum.de.selfhtml.org/archiv/2005/4/t106768/
6. http://regexp-evaluator.de/tutorial/
7. http://www.regular-expressions.info/
8. http://www.codeproject.com/KB/dotnet/regextutorial.aspx
9. http://regexlib.com/
10. http://msdn.microsoft.co m/en-us/library/az24scfc.aspx
11. http://regexlib.com/Resources.aspx
12. http://www.sellsbrothers.com/tools/#regexd

*Welche Skript- und Programmiersprachen, verwenden die "regular expressions":*

http://www.regular-expressions.info/tools.html
http://www.powergrep.com/

*Für unicode gibt es eine besondere Syntax:*

http://www.regular-expressions.info/refunicode.html

*Der Streaming-Editor, der innerhalb von Bash-Skripten Textstellen verändern kann:*

http://de.linwiki.org/wiki/Linuxfibel_-_Unix-Werkzeuge_-_Sed

*Regular-Expressions-Online-Tester:*

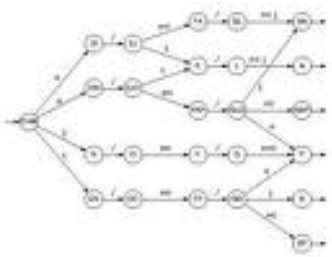
http://myregexp.com/ http://www.regextester.com/ http://www.myregextester.com/
http://www.fileformat.info/tool/regex.htm http://regexpal.com/

*Tracer and hook-Funktion an der Com-Schnittstelle*

***http://www.sellsbrothers.com/tools/#tracehook***

## *Hinweis: Weiteren 10 Seiten zu: grep, egrep, sed, reguläre Ausdrücke, PCRE (perl-compatible-eegular-expressions) findet man unter:* **"....Vorlage/fsi/fsi/BS-Schuster/grep-Befehle.odt".**

# Übungsaufgaben zu "(e)grep", "PCRE" und "sed"



1. Aufgabe
   Wodurch unterscheiden sich die Befehle "grep", "grep -e" und "egrep"?
   (Siehe man-page!)

2. Aufgabe
   Geben Sie mittels eines Befehls auf der bash-Konsole alle Zeilen der Datei Text.txt aus, die jeweils mit "Otto" am Zeilenende enden!

3. Aufgabe
   Suchen Sie aus der Passwortdatei /etc/passwd alle Login-Namen heraus!
   Jede Zeile der Datei hat die Struktur:
   ***user:x:id:guid:Beschreibung:Home-Verzeichnis:Kommando-Shell***

4. Aufgabe
   Nenne drei grafische Windows-Anwendungen, die nach der Perl-Syntax Zeichenketten filtern können!

5. Aufgabe
   Was bedeutet nachfolgender Befehl und was wird als Ergebnis ausgegeben?
   Befehl: ***echo "Hallo Ostern ist nicht mehr weit!" | sed s/ern/preußen/g | s/" weit" //g***

6. Aufgabe
Welches Ergebnis liefert: ***echo "D-91154 Roth"*** **|** ***sed*** **s/[a-zA-Z\-]/ /g**

_____

7. Aufgabe
```
#! /bin/bash
a="http://192.168.3.4/index.php"
${a#http}
echo $a
```

_____

_____

_____

8. Aufgabe
```
who | grep -v "root"
```

_____

_____

_____

9. Aufgabe
```
 T1="www.bs-roth.de/xyz/fg1.php"
text="http://192.168.0.3/index.php"
um=${text:0:7}${T1:0:14}${text:18}
 echo $um
```

_____

_____

_____

10. Aufgabe
Die nachfolgende Datei: datei.sh wird wie folgt aufgerufen:
```
#!/bin/bash
 while read i
      do
         #echo $i
         echo $i | sed s/aber//g
  done
```

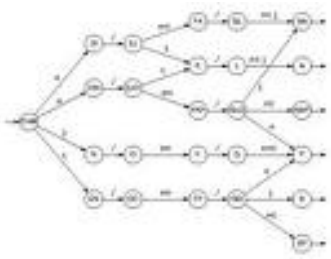
Aufruf: echo "asdf aber der Hans der kann es aber" | datei.sh

_____

_____

_____

# Übungsaufgaben zu "(e)grep", "PCRE" und "sed"



1. Aufgabe
   Wodurch unterscheiden sich die Befehle "grep", "grep -e" und "egrep"?
   (Siehe man-page!)

grep filtert nur Zeichenfolgen, während egrep mit „Mustererkennung“ arbeitet.
Der egrep Befehl und der Befehl grep -E sind identische Befehle!

2. Aufgabe
   Geben Sie mittels eines Befehls auf der bash-Konsole alle Zeilen der Datei Text.txt aus, die jeweils mit "Otto" am Zeilenende enden!

   Der Befehl lautet: egrep „Otto$“ /Pfad/Text.txt

3. Aufgabe
   Suchen Sie aus der Passwortdatei /etc/passwd alle Login-Namen heraus!
   Jede Zeile der Datei hat die Struktur:
   ***user:x:id:guid:Beschreibung:Home-Verzeichnis:Kommando-Shell***

   cat /etc/passwd | sed "s/:[[:alnum:][:space:]/]*//g"

4. Aufgabe
   Nenne drei grafische Windows-Anwendungen, die nach der Perl-Syntax Zeichenketten filtern können!
   http://www.rexegg.com/pcregrep-pcretest.html
   https://lists.exim.org/lurker/message/20120801.082712.76bccbad.en.html
   http://www.ultrapico.com/Expresso.htm
   oder ***RegexBuddy , RegexMagic***

5. Aufgabe
   Was bedeutet nachfolgender Befehl und was wird als Ergebnis ausgegeben?
   Befehl: **echo "Hallo Ostern ist nicht mehr weit" | sed "s/ern/preußen/g" | sed "s/ weit//g"**
   Die Echo-Ausgabe wird mittels Pipe „|“ als Stream an den sed-Befehl weitergeleitet.
   Dieser nimmt den Stream auf und ersetzt jedes Vorkommen von „ern“ durch den Text-String „preußen“ und „weit“ durch einen Leerstring. Die Lösung lautet: „Hallo Ostpreußen ist nicht mehr“.

6. Aufgabe
   Welches Ergebnis liefert: **echo "D-91154 Roth" | sed "s/[a-zA-Z\-]//g"**
   91154

7. Aufgabe

```
#! /bin/bash                          Verwende /bin/bash als Interpreter
a="http://192.168.3.4/index.php"      Setze den Inhalt der Variable „$a“
b=${a#http}                           Setze die Variable „$b“ auf „://192.168.3.4/index.php“
echo $b                               Gebe die Variable „$b“ aus.
```

8. Aufgabe
   who | grep -v "root"
   Gibt alle gegenwärtig am System angemeldeten Benutzerlogins aus, nur keine „root“-Logins!

9. Aufgabe

```
T1="www.bs-roth.de/xyz/fg1.php"
text="http://192.168.0.3/index.php"
um=${text:0:7}${T1:0:14}${text:18}
echo $um
```

   Die URL setzt sich wie folgt zusammen:
   http://www.bs-roth.de/index.php = http://www.bs-roth.de/index.php

10. Aufgabe
Die nachfolgende Datei: datei.sh wird wie folgt aufgerufen:

```
#!/bin/bash
 while read i
      do
         #echo $i
         echo $i | sed „s/aber//g“
  done
```

Aufruf: echo "asdf aber der Hans der kann es aber" | datei.sh

**Lösung: Die Ausgabe lautet:** asdf der Hans der kann es

# *Die Zeichenkettenprüfung in Perl - String matching*

Quellen: http://www.tutorialspoint.com/perl/perl_regular_expression.htm

http://www.fedorawiki.de/index.php/Bash_Benutzerhandbuch_3.2_Start

http://docs.python.org/library/string.html

http://www.troubleshooters.com/codecorn/littperl/perlreg.htm

http://affy.blogspot.com/p5be/ch10.htm

http://ezinearticles.com/?A-Quick-Guide-to-Using-the-Regular-Expression-Substitution-Operator&id=1406619

http://www.hurricanesoftwares.com/substitution-translation-of-regular-expressions-in-perl/

http://www.comp.leeds.ac.uk/Perl/matching.html

Eines der meist genutzten, sinnvollen und mächtigsten Perl-Eigenschaften ist die Zeichenkettenverarbeitung (string manipulation facilities). Als Kern dieser Technik gilt die RE-Technik (*regular expression* - technic), die von vielen UNIX-System-Tools angewandt wird.

## Perl-Compatible-Regular-Expression = PCRE

Ein regulärer Ausdruck besteht aus Schrägstrichen "Slashes", und Zeichenketten-Treffern mit dem **=~** Operator. Der nachfolgende Ausdruck ist wahr, wenn die Zeichenkette: "*the*" in der Variable: $sentence enthalten ist.

```
$sentence =~ /the/
```

Die RE berücksichtigt die Groß- und Kleinschreibung, so dass

```
$sentence = "The quick brown fox";
```

keinen "Treffer" darstellt, also false zurückliefert. Der negierte Vergleichsoperator: **!~** zeigt an, dass "the" nicht im Satz enthalten ist. Der Vergleich liefert als true zurück.

```
$sentence !~ /the/
```

# Die $_ Spezial-Variable

Man eine nun eine Abfrage wie folgt formulieren:

```
if ($sentence =~ /under/)
{
        print "We're talking about rugby\n";
}
```

which would print out a message if we had either of the following

```
$sentence = "Up and under";
$sentence = "Best winkles in Sunderland";
```

But it's often much easier if we assign the sentence to the special variable **$_** which is of course a scalar. If we do this then we can avoid using the match and non-match operators and the above can be written simply as

```
if (/under/)
{
        print "We're talking about rugby\n";
}
```

The **$_** variable is the default for many Perl operations and tends to be used very heavily.

# More on REs

In an RE there are plenty of special characters, and it is these that both give them their power and make them appear very complicated. It's best to build up your use of REs slowly; their creation can be something of an art form.

Here are some special RE characters and their meaning

```
.       # Any single character except a newline
^       # The beginning of the line or string
$       # The end of the line or string
*       # Zero or more of the last character
+       # One or more of the last character
?       # Zero or one of the last character
```

and here are some example matches. Remember that should be enclosed in **/.../** slashes to be used.

```
t.e     # t followed by anthing followed by e
```

```
        # This will match the
        #                 tre
        #                 tle
        # but not te
        #         tale
^f      # f at the beginning of a line
^ftp    # ftp at the beginning of a line
e$      # e at the end of a line
tle$    # tle at the end of a line
und*    # un followed by zero or more d characters
        # This will match un
        #                 und
        #                 undd
        #                 unddd (etc)
.*      # Any string without a newline. This is because
        # the . matches anything except a newline and
        # the * means zero or more of these.
^$      # A line with nothing in it.
```

There are even more options. Square brackets are used to match any one of the characters inside them. Inside square brackets a **-** indicates "between" and a ^ at the beginning means "not":

```
[qjk]           # Either q or j or k
[^qjk]          # Neither q nor j nor k
[a-z]           # Anything from a to z inclusive
[^a-z]          # No lower case letters
[a-zA-Z]        # Any letter
[a-z]+          # Any non-zero sequence of lower case letters
```

At this point you can probably skip to the end and do at least most of the exercise. The rest is mostly just for reference.

A vertical bar **|** represents an "or" and parentheses **(...)** can be used to group things together:

```
jelly|cream     # Either jelly or cream
(eg|le)gs       # Either eggs or legs
(da)+           # Either da or dada or dadada or...
```

Here are some more special characters:

```
\n              # A newline
\t              # A tab
\w              # Any alphanumeric (word) character.
                # The same as [a-zA-Z0-9_]
\W              # Any non-word character.
                # The same as [^a-zA-Z0-9_]
\d              # Any digit. The same as [0-9]
\D              # Any non-digit. The same as [^0-9]
\s              # Any whitespace character: space,
                # tab, newline, etc
\S              # Any non-whitespace character
\b              # A word boundary, outside [] only
\B              # No word boundary
```

Clearly characters like **$**, **|**, **[**, **)**, ****, **/** and so on are peculiar cases in regular expressions. If you want to match for one of those then you have to preceed it by a backslash. So:

```
\|              # Vertical bar
\[              # An open square bracket
\)              # A closing parenthesis
\*              # An asterisk
\^              # A carat symbol
\/              # A slash
\\              # A backslash
```

and so on.

## Some example REs

As was mentioned earlier, it's probably best to build up your use of regular expressions slowly. Here are a few examples. Remember that to use them for matching they should be put in **/.../** slashes

```
[01]            # Either "0" or "1"
\/0             # A division by zero: "/0"
\/ 0            # A division by zero with a space: "/ 0"
\/\s0           # A division by zero with a whitespace:
                # "/ 0" where the space may be a tab etc.
\/ *0           # A division by zero with possibly some
                # spaces: "/0" or "/ 0" or "/  0" etc.
\/\s*0          # A division by zero with possibly some
                # whitespace.
\/\s*0\.0*      # As the previous one, but with decimal
                # point and maybe some 0s after it. Accepts
                # "/0." and "/0.0" and "/0.00" etc and
                # "/ 0." and "/  0.0" and "/   0.00" etc.
```

## Exercise

Previously your program counted non-empty lines. Alter it so that instead of counting non-empty lines it counts only lines with

- the letter *x*
- the string *the*
- the string *the* which may or may not have a capital *t*
- the word *the* with or without a capital. Use **\b** to detect word boundaries.

In each case the program should print out every line, but it should only number those specified. Try to use the **$_** variable to avoid using the **=~** match operator explicitly.

# Substitution and translation

As well as identifying regular expressions Perl can make substitutions based on those matches. The way to do this is to use the **s** function which is designed to mimic the way substitution is done in the vi text editor. Once again the match operator is used, and once again if it is omitted then the substitution is assumed to take place with the **$_** variable.

To replace an occurrence of *london* by *London* in the string $sentence we use the expression

```
$sentence =~ s/london/London/
```

and to do the same thing with the **$_** variable just

```
s/london/London/
```

Notice that the two regular expressions (*london* and *London*) are surrounded by a total of three slashes. The result of this expression is the number of substitutions made, so it is either 0 (false) or 1 (true) in this case.

## Options

This example only replaces the first occurrence of the string, and it may be that there will be more than one such string we want to replace. To make a global substitution the last slash is followed by a **g** as follows:

```
s/london/London/g
```

which of course works on the **$_** variable. Again the expression returns the number of substitutions made, which is 0 (false) or something greater than 0 (true).

If we want to also replace occurrences of *lOndon*, *lonDON*, *LoNDoN* and so on then we could use

```
s/[Ll][Oo][Nn][Dd][Oo][Nn]/London/g
```

but an easier way is to use the **i** option (for "ignore case"). The expression

```
s/london/London/gi
```

will make a global substitution ignoring case. The **i** option is also used in the basic **/.../** regular expression match.

# Remembering patterns

It's often useful to remember patterns that have been matched so that they can be used again. It just so happens that anything matched in parentheses gets remembered in the variables **$1**,...,**$9**. These strings can also be used in the same regular expression (or substitution) by using the special RE codes **\1**,...,**\9**. For example

```
$_ = "Lord Whopper of Fibbing";
s/([A-Z])/:\1:/g;
print "$_\n";
```

will replace each upper case letter by that letter surrounded by colons. It will print *:L:ord :W:hopper of :F:ibbing*. The variables **$1**,...,**$9** are read-only variables; you cannot alter them yourself.

As another example, the test

```
if (/(\b.+\b) \1/)
{
        print "Found $1 repeated\n";
}
```

will identify any words repeated. Each **\b** represents a word boundary and the **.+** matches any non-empty string, so **\b.+\b** matches anything between two word boundaries. This is then remembered by the parentheses and stored as **\1** for regular expressions and as **$1** for the rest of the program.

The following swaps the first and last characters of a line in the **$_** variable:

```
s/^(.)(.*)(.)$/\3\2\1/
```

The ^ and **$** match the beginning and end of the line. The **\1** code stores the first character; the **\2** code stores everything else up the last character which is stored in the **\3** code. Then that whole line is replaced with **\1** and **\3** swapped round.

After a match, you can use the special read-only variables **$`** and **$&** and **$'** to find what was matched before, during and after the seach. So after

```
$_ = "Lord Whopper of Fibbing";
/pp/;
```

all of the following are true. (Remember that eq is the string-equality test.)

```
$` eq "Lord Wo";
$& eq "pp";
$' eq "er of Fibbing";
```

Finally on the subject of remembering patterns it's worth knowing that inside of the slashes of a match or a substitution variables are interpolated. So

```
$search = "the";
s/$search/xxx/g;
```

will replace every occurrence of *the* with *xxx*. If you want to replace every occurence of *there* then you cannot do **s/$searchre/xxx/** because this will be interpolated as the variable $searchre. Instead you should put the variable name in curly braces so that the code becomes

```
$search = "the";
s/${search}re/xxx/;
```

---

# Translation

The **tr** function allows character-by-character translation. The following expression replaces each *a* with *e*, each *b* with *d*, and each *c* with *f* in the variable $sentence. The expression returns the number of substitutions made.

```
$sentence =~ tr/abc/edf/
```

Most of the special RE codes do not apply in the tr function. For example, the statement here counts the number of asterisks in the $sentence variable and stores that in the $count variable.

```
$count = ($sentence =~ tr/*/*/);
```

However, the dash is still used to mean "between". This statement converts **$_** to upper case.

```
tr/a-z/A-Z/;
```

---

# Exercise

Your current program should count lines of a file which contain a certain string. Modify it so that it counts lines with double letters (or any other double character). Modify it again so that these double letters appear also in parentheses. For example your program would produce a line like this among others:

```
023 Amp, James Wa(tt), Bob Transformer, etc. These pion(ee)rs conducted many
```

Try to get it so that all pairs of letters are in parentheses, not just the first pair on each line.

For a slightly more interesting program you might like to try the following. Suppose your program is called countlines. Then you would call it with

```
./countlines
```

However, if you call it with several arguments, as in

```
./countlines first second etc
```

then those arguments are stored in the array **@ARGV**. In the above example we have **$ARGV[0]** is *first* and **$ARGV[1]** is *second* and **$ARGV[2]** is *etc*. Modify your program so that it accepts one argument and counts only those lines with that string. It should also put occurrences of this string in paretheses. So

```
./countlines the
```

will output something like this line among others:

```
019 But (the) greatest Electrical Pioneer of (the)m all was Thomas Edison, who
```

# Split

A very useful function in Perl is **split**, which splits up a string and places it into an array. The function uses a regular expression and as usual works on the **$_** variable unless otherwise specified.

The **split** function is used like this:

```
$info = "Caine:Michael:Actor:14, Leafy Drive";
@personal = split(/:/, $info);
```

which has the same overall effect as

```
@personal = ("Caine", "Michael", "Actor", "14, Leafy Drive");
```

If we have the information stored in the **$_** variable then we can just use this instead

```
@personal = split(/:/);
```

If the fields are divided by any number of colons then we can use the RE codes to get round this. The code

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:+/);
```

is the same as

```
@personal = ("Capes", "Geoff",
             "Shot putter", "Big Avenue");
```

But this:

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:/);
```

would be like

```
@personal = ("Capes", "Geoff", "",
             "Shot putter", "", "", "Big Avenue");
```

A word can be split into characters, a sentence split into words and a paragraph split into sentences:

```
@chars = split(//, $word);
@words = split(/ /, $sentence);
@sentences = split(/\./, $paragraph);
```

In the first case the null string is matched between each character, and that is why the @chars array is an array of characters - ie an array of strings of length 1.

# Exercise

A useful tool in natural language processing is concordance. This allows a specific string to be displayed in its immediate context whereever it appears in a text. For example, a concordance program identifying the target string *the* might produce some of the following output. Notice how the occurrences of the target string line up vertically.

```
discovered (this is the truth) that when he
t kinds of metal to the leg of a frog, an e
rrent developed and the frog's leg kicked,
 longer attached to the frog, which was dea
normous advances in the field of amphibian
ch it hop back into the pond -- almost.  Bu
ond -- almost.  But the greatest Electrical
ectrical Pioneer of them all was Thomas Edi
```

This exercise is to write such a program. Here are some tips:

- Read the entire file into array (this obviously isn't useful in general because the file may be

extremely large, but we won't worry about that here). Each item in the array will be a line of the file.

- When the chop function is used on an array it chops off the last character of every item in the array.
- Recall that you can join the whole array together with a statement like **$text = "@lines";**
- Use the target string as delimiter for splitting the text. (Ie, use the target string in place of the colon in our previous examples.) You should then have an array of all the strings between the target strings.
- For each array element in turn, print it out, print the target string, and then print the next array element.
- Recall that the last element of an array @food has index $#food.

As it stands this would be a pretty good program, but the target strings won't line up vertically. To tidy up the strings you'll need the **substr** function. Here are three examples of its use.

```
substr("Once upon a time", 3, 4);       # returns "e up"
substr("Once upon a time", 7);          # returns "on a time"
substr("Once upon a time", -6, 5);      # returns "a tim"
```

The first example returns a substring of length 4 starting at position 3. Remember that the first character of a string has index 0. The second example shows that missing out the length gives the substring right to the end of the string The third example shows that you can also index from the end using a negative index. It returns the substring that starts at the 6th character from the end and has length 5.

If you use a negative index that extends beyond the beginning of the string then Perl will return nothing or give a warning. To avoid this happening you can pad out the string by using the **x** operator mentioned earlier. The expression (" "x30) produces 30 spaces, for example.

# Associative arrays

Ordinary list arrays allow us to access their element by number. The first element of array @food is $food[0]. The second element is $food[1], and so on. But Perl also allows us to create arrays which are accessed by string. These are called *associative arrays*.

To define an associative array we use the usual parenthesis notation, but the array itself is prefixed by a % sign. Suppose we want to create an array of people and their ages. It would look like this:

```
%ages = ("Michael Caine", 39,
         "Dirty Den", 34,
         "Angie", 27,
         "Willy", "21 in dog years",
         "The Queen Mother", 108);
```

Now we can find the age of people with the following expressions

```
$ages{"Michael Caine"};         # Returns 39
$ages{"Dirty Den"};             # Returns 34
$ages{"Angie"};                 # Returns 27
$ages{"Willy"};                 # Returns "21 in dog years"
$ages{"The Queen Mother"};      # Returns 108
```

Notice that like list arrays each **%** sign has changed to a **$** to access an individual element because that element is a scalar. Unlike list arrays the index (in this case the person's name) is enclosed in curly braces, the idea being that associative arrays are fancier than list arrays.

An associative array can be converted back into a list array just by assigning it to a list array variable. A list array can be converted into an associative array by assigning it to an associative array variable. Ideally the list array will have an even number of elements:

```
@info = %ages;          # @info is a list array. It
                        # now has 10 elements
$info[5];               # Returns the value 27 from
                        # the list array @info
%moreages = @info;      # %moreages is an associative
                        # array. It is the same as %ages
```

# Operators

Associative arrays do not have any order to their elements (they are just like hash tables) but is it possible to access all the elements in turn using the **keys** function and the **values** function:

```
foreach $person (keys %ages)
{
        print "I know the age of $person\n";
```

```
}
foreach $age (values %ages)
{
        print "Somebody is $age\n";
}
```
When **keys** is called it returns a list of the keys (indices) of the associative array. When **values** is called it returns a list of the values of the array. These functions return their lists in the same order, but this order has nothing to do with the order in which the elements have been entered.

When **keys** and **values** are called in a scalar context they return the number of key/value pairs in the associative array.

There is also a function **each** which returns a two element list of a key and its value. Every time **each** is called it returns another key/value pair:

```
while (($person, $age) = each(%ages))
{
        print "$person is $age\n";
}
```

---

# Environment variables

When you run a perl program, or any script in UNIX, there will be certain environment variables set. These will be things like USER which contains your username and DISPLAY which specifies which screen your graphics will go to. When you run a perl CGI script on the World Wide Web there are environment variables which hold other useful information. All these variables and their values are stored in the associative **%ENV** array in which the keys are the variable names. Try the following in a perl program:

```
print "You are called $ENV{'USER'} and you are ";
print "using display $ENV{'DISPLAY'}\n";
```

# Subroutines

---

Like any good programming langauge Perl allows the user to define their own functions, called *subroutines*. They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end. A subroutine has the form

```
sub mysubroutine
{
        print "Not a very interesting routine\n";
        print "This does the same thing every time\n";
}
```

regardless of any parameters that we may want to pass to it. All of the following will work to call this subroutine. Notice that a subroutine is called with an **&** character in front of the name:

```
&mysubroutine;          # Call the subroutine
&mysubroutine($_);      # Call it with a parameter
&mysubroutine(1+2, $_); # Call it with two parameters
```

---

# Parameters

In the above case the parameters are acceptable but ignored. When the subroutine is called any parameters are passed as a list in the special @_ list array variable. This variable has absolutely nothing to do with the **$**_ scalar variable. The following subroutine merely prints out the list that it was called with. It is followed by a couple of examples of its use.

```
sub printargs
{
        print "@_\n";
}


&printargs("perly", "king");    # Example prints "perly king"
&printargs("frog", "and", "toad"); # Prints "frog and toad"
```

Just like any other list array the individual elements of @_ can be accessed with the square bracket notation:

```
sub printfirsttwo
{
```

```
        print "Your first argument was $_[0]\n";
        print "and $_[1] was your second\n";
}
```
Again it should be stressed that the indexed scalars **$_[0]** and **$_[1]** and so on have nothing to with the scalar **$_** which can also be used without fear of a clash.

---

# Returning values

Result of a subroutine is always the last thing evaluated. This subroutine returns the maximum of two input parameters. An example of its use follows.

```
sub maximum
{
        if ($_[0] > $_[1])
        {
                $_[0];
        }
        else
        {
                $_[1];
        }
}


$biggest = &maximum(37, 24);    # Now $biggest is 37
```
The &printfirsttwo subroutine above also returns a value, in this case 1. This is because the last thing that subroutine did was a **print** statement and the result of a successful **print** statement is always 1.

---

# Local variables

The @_ variable is local to the current subroutine, and so of course are **$_[0]**, **$_[1]**, **$_[2]**, and so on. Other variables can be made local too, and this is useful if we want to start altering the input parameters. The following subroutine tests to see if one string is inside another, spaces not withstanding. An example follows.

```
sub inside
{
        local($a, $b);                  # Make local variables
        ($a, $b) = ($_[0], $_[1]);      # Assign values
        $a =~ s/ //g;                   # Strip spaces from
        $b =~ s/ //g;                   #   local variables
        ($a =~ /$b/ || $b =~ /$a/);     # Is $b inside $a
                                        #   or $a inside $b?
}


&inside("lemon", "dole money");         # true
```
In fact, it can even be tidied up by replacing the first two lines with

```
local($a, $b) = ($_[0], $_[1]);
```