Title: Andreas Altwein - Basics
Description: Reguläre Ausdrücke

# Reguläre Ausdrücke

Mit regulären Ausdrücken kann man in beliebigen Texten nach einem bestimmten Muster suchen. Nehmen wir an, du hast einen beliebig langen Text, vielleicht sogar ein ganzes Buch, dass als Textdatei vorliegt. Wenn du nun wissen möchtest, wie oft und / oder an welcher Stelle die Zeichenkette “Adler” vorkommt, dann kannst du diverse Werkzeuge verwenden, um danach zu suchen. Jede Textverarbeitung stellt einfache Suchtools zur Verfügung. Einige bessere Editoren oder Kommandozeilentools stellen eine Suche zur Verfügung, die mit regulären Ausdrücken arbeitet - die wesentlich mächtiger sind als eine einfache Textsuche mit Wildcards.

## eGrep

Eines dieser Tools nennt sich *egrep* und ist seit Urbeginn Bestandteil einer jeden Linux-Distribution. eGrep ist ein Werkzeug, das ausschließlich der Suche von Zeichenketten in einer Datei gewidmet ist.

Angenommen, du hast eine Datei “text.txt” mit folgendem Inhalt:

> Weit draußen in den unerforschten Einöden eines total
>
> aus der Mode gekommenen Ausläufers des westlichen
>
> Spiralarms der Galaxis, leuchtet unbeachtet eine kleine
>
> gelbe Sonne.

Nehmen wir weiterhin an, dass du in dieser Datei nach der Zeichenkette “Galaxis” suchen möchtest. Mit eGrep ist das schnell erledigt:

```
egrep 'Galaxis' text.txt
```

Grep übernimmt zwei Parameter, als erstes einen regulären Ausdruck, nach dem gesucht werden soll, und als zweites eine Datei *in* der nach dem Ausdruck gesucht werden soll.

Als Ausgabe erhält man dann alle Zeilen, in der sich der gesuchte Ausdruck befindet, also in obigem Beispiel etwa:

```
Spiralarms der *Galaxis*, leuchtet unbeachtet eine kleine
```

Optional kann als erster Parameter noch eine (oder mehrere) Option angegeben werden. Durch

```
egrep -n 'Galaxis' text.txt
```

erhält man beispielsweise nicht nur die Zeile selbst, sondern auch die Zeilennummer. Möchte man dahingegen nur wissen, wie oft eine Zeichenkette in einem Text vorkommt, kann man das Flag “c” verwenden:

```
egrep -c 'Galaxis' text.txt
```

Ich möchte an dieser Stelle allerdings nicht weiter auf eGrep eingehen, da es sich hierbei nur um *ein* Werkzeug handelt. Wenn man reguläre Ausdrücke erstellen und testen möchte, ist es allerdings das Werkzeug der Wahl.

## String & Number Matching

Die einfachste Form eines regulären Ausdrucks, habe ich bereits in obigem Beispiel gezeigt. Sie ist auch gleichzeitig die Form, die so gut wie Niemandem Probleme bereitet. Wenn man nach einem ganz bestimmten, bekannten String sucht, dann … richtig: schreibt man diesen String einfach als regulären Ausdruck. Möchte ich beispielsweise in diesem Absatz nach dem Wort “String” suchen, dann lautet der entsprechende reguläre Ausdruck hierfür:

```
String
```

Anzumerken ist hier, dass reguläre Ausdrücke grundsätzlich case-sensitive sind. Der Ausdruck “String” auf den obigen Absatz angewendet würde also 3 Treffer landen, wären “string” genau 0 Treffer hat.

Auf die gleiche Art und Weise funktioniert die Suche nach Ziffern, bzw. Zahlen. Um in dem obigen Absatz nach der Zahl 3 zu suchen, wird diese Ziffer einfach als regulärer Ausdruck verwendet, also

```
3
```

Natürlich kannst du auch nach zusammengesetzten Ziffern, ja sogar nach Dezimalzahlen suchen, z.B. mit

```
3.1416
```

## Platzhalter und Wildcards

Wenn du schon einmal in einer Shell oder einer DOS-Box den Befehl ‘ls’, bzw. ‘dir’ verwendet hast, um dir Dateien des aktuellen Verzeichnisses anzeigen zu lassen, dann weißt du sicherlich, dass es dort Wildscards und Platzhalter gibt.

Diese existieren auch in regulären Ausdrücken, allerdings in einer wesentlich mächtigeren Form.

Der einfachste Platzhalter ist der für ein beliebiges Zeichen (zu denen das Leerzeichen übrigens auch gehört). Es ist der Punkt (.). Wird dieser in einem regulären Ausdruck verwendet, dann kann an seiner Stelle jeder beliebige Buchstabe oder jede beliebige Ziffer, ja auch jedes beliebige andere Zeichen stehen.

Angenommen, du möchstest nach den Worten Haus, Maus und Laus suchen. Diese Wörter unterscheiden sich lediglich im ersten Buchstaben. Hier kannst du den Punkt als Platzhalter setzen.

```
.aus
```

findet also zuverlässig die Begriffe Haus, Maus und Laus, darüber hinaus aber auch solche Begriffe wie Hausmeister, Mausetier und Ähnliches.

Bisher unterscheidet sich dieser Platzhalter in keinster Weise von denen, die wir aus einer Shell

oder DOS-Box kennen. Nun aber die gute Frage: was ist, wenn wir zwar Maus und Haus, aber auf keinen Fall den Begriff Laus finden möchten? Hier beginnen die herkömmlichen Platzhalter zu versagen.

Die Regel, wie wir Haus und Maus finden ist einfach: wir suchen nach einem Wort, das mit einem H oder einem M beginnt und dann mit “aus” weitergeht. Hierfür dienen die eckigen Klammer. Schau dir folgenden regulären Ausdruck an:

```
[HM]aus
```

Die eckigen Klammern, bzw. deren Inhalt stehen für genau ein (1) Zeichen (und damit meine ich tatsächlich genau ein Zeichen). Innerhalb der Klammern stehen die Zeichen, die erlaubt sind. Neben Buchstaben können das natürlich auch Zahlen oder Sonderzeichen sein., d.h. ein Ausdruck wie

```
[123]fach
```

würde die Begriffe 1fach, 2fach und 3fach finden, *nicht* aber den Begriff 123fach!

“Jaaaa”, wirst du wahrscheinlich einwerfen. “Aber wie sieht es aus, wenn ich alle Großbuchstaben außer dem Z haben möchte?” In der Tat sähe der reguläre Ausdruck dann etwas merkwürdig aus. Nehmen wir ein Praxisbeispiel, in dem alle Ausdrücke erkannt werden soll, die einen Großbuchstaben mit Ausnahme des Z gefolgt von einer einzelnen Ziffer enthalten, also zum Beispiel H1, X9, L2, ...

Mit unserem bisherigen Wissen sähe dieser Ausdruck dann wie folgt aus:

```
[ABCDEFGHJIKLMNOPQRSTUVWXY][0123456789]
```

Es geht aber auch wesentlich einfach nämlich so:

```
[A-Y][0-9]
```

Hier ist A-Y einfach ein Kürzel für alle Großbuchstaben von A bis Y und 0-9 ein Kürzel für alle Zahlen von 0 bis 9.

Sollen auch Kleinbuchstaben zugelassen werden, so müssen diese innerhalb der eckigen Klammern noch extra aufgeführt werden. Der Ausruck

```
[A-Za-z][0-9]
```

unterstützt also nicht nur Strings wie H1, X3 und Q8, sondern auch h1, x3 und q8.

Genauso wie wir mit den eckigen Klammern bestimmte Zeichen in die Suche *einschließen* können, ist es auch möglich, sie auszuschließen. Kommen wir noch einmal zurück auf unser Haus, Maus, Laus Beispiel. MIt dem Ausdruck ‘[HM]aus’ konnten wir direkt nach den Wörtern Maus und Haus suchen, während Laus nicht gefunden wurde.

Wenn es uns aber gar nicht auf die erstgenannten Wörter ankommt, sondern wir generell alle Wörter, die irgendwie mit ‘aus’ weitergehen finden möchten, aber definitiv nicht das Wort ‘Laus’, dann können wir das L auch aus der Suche ausschließen:

```
[^L]aus
```

Steht am Beginn einer Auflistung innerhalb der eckigen Klammern das Potenzzeichen, dann arbeitet dieses als Negierung. In diesem Fall wird nach einem Ausdruck gesucht, der mit irgendeinem Zeichen oder eine Zahl anfängt, mit Ausnahme des L und dann mit ‘aus’ weitergeht. Es würden also die Worte Maus und Haus gefunden, nicht aber Laus.

Auch in diesem Fall sind wir nicht auf ein Zeichen innerhalb der Klammer beschränkt, sondern wir können auch mehrere Zeichen hintereinander schreiben, also etwa:

```
[^LM]aus
[^A-K]aus
```

Im ersten Fall dürfte der Ausdruck nicht mit L oder M anfangen, im zweiten Fall werden alle Wörter unterdrückt, die mit Buchstaben zwischen A und K inklusive beginnen.

## Sonderzeichen

Nach einigen Zeichen kann nicht direkt gesucht werden, da ebendiese Zeichen bereits eine andere Bedeutung in regulären Ausdrücken haben. Hierzu zählen insbesondere folgende Zeichen:

- eckige Klammern
- der Punkt, Anführungszeichen und Hochkommata
- Stern, Plus, Fragezeichen und Dollarzeichen
- das Dachzeichen (^)
- das Pipezeichen (|)
- der Backslash (\)

Um nach diesen Zeichen zu suchen, muss man diese mit einem Backslash markieren. Möchtest du also nach dem String ‘[Haus]’ suchen, so muss den beiden Klammern ein Backslash vorangestellt werden:

```
\[Haus\]
```

## Quantoren

Quantoren in Regulären Ausdrücken sind ein mächtiges Konzept, das uns festzulegen hilft, wie oft ein Zeichen in einem Ausdruck vorkommen soll, muss oder darf.

Gehen wir von dem Fall aus, dass wir eine Textdatei nach Strings durchsuchen möchten, die auf einen Dateinamen mit der Endung “sof” hinweisen. Allein nach “sof” zu suchen ist problematisch, da wir auch solche Zeilen als Ergebnis geliefert bekommen, die “sof” als Teilstring enthalten, also beispielsweise einen String wie “Bring das *sof*ort weg”.

Wir wissen aber, dass eine Datei einen Namen mit beliebig vielen Zeichen enthält, gefolgt von einem Punkt und dann der Endung “sof”. Wir können also schon einmal behaupten, dass der reguläre Ausdruck

```
\.sof
```

kein schlechter Ansatz ist (du erinnerst dich daran, dass der Punkt ein Sonderzeichen ist, der mit dem Backslash maskiert werden muss?).

Wenn wir nun weiterhin wissen, dass der Dateiname aus beliebig vielen (oder gar keinen) Buchstaben besteht, könnten wir nun den Ausdruck zur Erkennung eines Buchstaben davor setzen, also

```
[A-Za-z]\.sof
```

Allerdings nutzt uns auch das nicht viel, denn damit würden nur Datein erkannt, deren Name aus genau einem Buchstaben besteht. Tatsächlich Abhilfe schafft hier das sogenannte “Kleene”, welches aus einem Stern besteht.

```
[A-Za-z]*\.sof
```

Das Kleene sagt aus, dass das vorhergehende Zeichen in dem regulären Ausdruck 0, einmal oder mehrmals vorhanden sein muss. Damit würden also alle Namen gefunden, die aus keinem, einem oder beliebig vielen Buchstaben bestehen.

Nun kann man kritisieren, dass der obige reguläre Ausdruck auch den String “.sof” findet, also eine Datei, völlig ohne Namen. Wir möchten aber nur solche Stellen im Text finden, die einen Namen haben, deren Teilstring vor dem Punkt also aus mindestens einem Zeichen (oder auch mehreren) besteht.

Die geschieht mit dem Plus Zeichen (+). Dieses arbeitet ganz ähnlich wie der Stern, mit dem Unterschied, dass der vorhergehende Ausdruck genau einmal oder mehrmals vorhanden sein muss. Ist er gar nicht vorhanden, führt das bei einem Kleene Plus - im Gegensatz zum Kleene Stern - nicht zum Erfolg.

Um einen Ausdruck zu erhalten, der also alle Dateien mit mindestens einem Zeichen findet, müssen wir unsere obige Definition ein wenig umbauen, nämlich in:

```
[A-Za-z]+\.sof
```

Zu guter letzt gibt es noch einen Quantor, der eine Mischung aus Kleene Stern und Keele Plus ist, nämlich das Fragezeichen. Das Fragzeichen gibt an, dass der vorhergehende Ausdruck entweder gar nicht oder genau einmal (aber nicht öfter) vorhanden sein muss. Mit ihm ist es also möglich, ein optionales Zeichen in unseren Ausdrücken unterzubringen, z.B.

```
Rh?ythmus
```

In diesem Beispiel ist das erste “h” optional, d.h. der Ausdruck erkennt sowohl das Wort “Rhytmus” als auch die (oft verwendete) Falschschreibweise “Rythmus”.

## Der Wiederholungsoperator

Mit den Quantoren stehen uns mächtige Mittel zur Verfügung. Allerdings gibt es Fälle, in denen selbst diese nicht ausreichen. Zum Beispiel dann, wenn eine bestimmte Anzahl von Wiederholungen eines Zeichens gesucht wird.

Nehmen wir an, dass wir nach Euro Beträgen suchen möchten, die 3-stellig und mit Angabe von 2-stelligen Cents im Text verstreut sind, also zum Beispiel “712,77”.

Die Verwendung von Quantoren scheidet hier aus, da wir für die Vorkommastelle genau 3 und für die Nachkommastelle genau 2 Zeichen (bzw. Ziffern) benötigen.

Stattdessen stellt uns das Arsenal der regulären Ausdrücke einen Wiederholungsoperator zur Verfügung, mit dem wir angeben können, wie oft ein Zeichen wiederholt werden soll. Möchten wir beispielsweise das Zeichen “A” genau 7 mal hintereinander erkennen, notieren wir dies, in dem wir hinter dem A eine 7 in geschweifte Klammern setzen:

```
A{7}
```

Kommen wir nun zurück zu unserem Euro-Beispiel. Die korrekte Notation zur Erkennung einer dreistelligen Eurosumme mit 2 Nachkommastellen ist somit

```
[0-9]{3},[0-9]{2}
```

Wem dieser Operator immer noch nicht ausreicht, der kann in den geschweiften Klammern auch eine Ober- und eine Untergrenze für die Anzahl von Wiederholungen schreiben. So führt der Ausdruck

```
[0-9]{3},[0-9]{2,3}
```

dazu, dass nicht nur 3-stellige Eurobeträge mit zwei Nachkommastellen, sonder auch solche mit 3 Nachkommastellen erkannt werden.

## Das logische Oder

Bleiben wir noch kurz bei den Eurobeträgen. In Deutschland trennen wir Euro von Centbeträgen mit einem Komma, in englischsprachigen Ländern wird dies durch einen Punkt gemacht. Da wir alle ja immer internationaler werden, sind beide Schreibweisen bedingt richtig, also

```
150,23
150.23
```

Der reguläre Ausdruck, denn wir oben entworfen haben, erkennt aber nur das Komma. Um ein Komma *oder* einen Dezimalpunkt zu erkennen, benötigen wir den Pipe-Operator. Dessen Verwendung führt dazu, dass entweder der Ausdruck, der vor dem Operator, *oder* der Ausdruck, der nach dem Operator steht, gesucht wird - einer von beiden muss vorhanden sein, aber nicht beide auf einmal. Das sieht dann so aus:

```
   [0-9]{3}(.|,)[0-9]{2}
```

Beachte an dieser Stelle die Klammer um den Ausdruck “.|,”. Sie bilden eine Gruppe, so dass tatsächlich nur nach einem Punkt oder einem Komma gesucht wird. Wären die Klammern nicht da, würde nach einer dreistelligen Zahl mit einem Punkt *oder* nach einem Komma mit einer zweistelligen Zahl gesucht.

## Gruppen

Gruppen können aber noch viel mehr als nur ein oder mehrere Zeichen zu gruppieren. Stellen wir uns einmal vor, wir möchten einen Text danach durchsuchen, ob er irgendwo den Ausdruck

```
f(x)=x^2
```

enthält. Mit dem Wissen, dass wir runde Klammern und das Dachzeichen mit einem Backslash markieren müssen, sollte es uns nicht schwer fallen, auf die Lösung zu kommen, die da lautet:

```
f\(x\)=x\^2
```

Was ist aber, wenn der Text nicht “x” als Variable enthält, sondern zum Beispiel “t”? Klar, wir können den Range-Operator verwenden:

```
f\([a-z]\)=[a-z]\^2
```

Tatsächlich funktioniert das auch, und selbst ein Ausdruck wie f(t)=t$^2$ wird gefunden. Darüber hinaus aber auch Ausdrücke, die unerwünscht sind, wie beispielsweise f(t)=x$_2$.

Abhilfe schaffen hier - du kannst es dir sicherlich schon denken - die Gruppen. Wenn wir den ersten Range-Operator in runde Klammern setzen, und damit eine Gruppe bilden, und wenn darüber hinaus ein Ausdruck gefunden wird, der diesem entspricht, dann ist das Ergebnis im weiteren Verlauf des regulären Ausdruckes über den Operator \n wiederverwendbar, wobei n hier der 1-basierte Index der Gruppe ist, dessen Ergebnis wir verwenden möchten.

Sprich: wenn wir den ersten Range-Operator in einer Gruppe verwenden, können wir auf den zweiten Range-Operator nach dem Gleichheitsszeichen verzichten und stattdessen einfach \1 verwenden. Wenn die Gruppe beispielsweise den Buchstaben “s” gefunden hat, dann wird das \1 im späteren Verlauf unserer Ausdruckes ebenfalls (intern) durch ein “s” ersetzt.

Nun bewegen wir uns langsam auch in den Bereich der etwas anspruchsvolleren regulären Ausdrücke:

```
f\(([a-z])\)=\1\^2
```

## Start- und Endoperator

Manchmal ist es sinnvoll, eine Möglichkeit an der Hand zu haben, direkt am Anfang oder am Ende einer Zeile zu suchen. Das Suchen am Beginn einer Zeile erledigt das Dachzeichen, z.B. so:

```
^Es ist
```

Dieser reguläre Ausdruck findet nicht alle Zeichenketten “Es ist” in einem Text, sondern nur solche Zeichenketten “Es ist”, die genau am Anfang einer Zeile stehen.

Ewas verwirrend ist das Dachzeichen, dass hierfür verwendet wird, denn weiter oben habe ich geschrieben, dass es für eine Negierung innerhalb der eckigen Klammern verwendet wird.
Das ist auch weiterhin richtig, allerdings bedeutet das Dachzeichen *nur* als erstes Zeichen hinter einer sich öffnenden eckigen Klammern eine Negierung - in allen anderen Fällen verweist es auf den Anfang einer Zeile.

Ähnlich verwendet wird das Dollarzeichen, um am Ende einer Zeile zu suchen:

```
   hat\.$
```

Dieser Ausdruck findet alle Zeichenketten “hat.”, die sich am Ende einer Zeile befinden.