SelfLinux-0.13.1

Reguläre Ausdrücke

Autor: Dennis Roch (*ysae@users.sourceforge.net*)
Formatierung: Matthias Hagedorn (*matthias.hagedorn@selflinux.org*)

Inhaltsverzeichnis

1 Einleitung

Der Mensch ist ständig auf der Suche. Bei der Suche nach sich selbst, nach Geborgenheit und Wärme oder etwa nach den Autoschlüsseln, kann SelfLinux jedoch leider nicht weiterhelfen ...

Wer aber viel mit Texten arbeitet oder gar programmiert, dem wird dieser Text vermutlich eine nützliche Ergänzung seines Wissens sein.

Wie bereits der Überschrift zu entnehmen, geht es um Reguläre Ausdrücke (auch: Regular Expressions, Regexp, Regex, RE). Sie ermöglichen uns, Kollege Computer zu sagen, was wir eigentlich suchen (und ggf. ersetzen) wollen. Und das mit erstaunlicher Flexibilität und Präzision. Und da es Reguläre Ausdrücke schon zu Urzeiten von Unix gab, existieren auch eine Reihe von Programmen und Programmiersprachen, die sie verstehen. Dazu zählen `grep`, vi, Emacs, sed, more, less oder auch `Perl`.

2 Muster und Reguläre Ausdrücke

Das Prinzip der RE dürfte von den Mustern (engl: Patterns) her bekannt sein. Ihnen ist eigentlich jeder schon mal begegnet.

Einfaches Beispiel:
`echo *`
listet alle Dateien und Verzeichnisse des aktuellen Verzeichnisses auf.

Das Sternchen ist also ein Platzhalter für alle Dateien und Verzeichnisse des aktuellen Verzeichnisses, deren Name eine beliebige Zeichenkette ist (also quasi alle Dateien). Ein **a*** bezöge sich auf alle Dateinamen, die mit einem kleinen **a** beginnen.

Mit nur einem Muster kann man also eine Reihe von Dateinamen (bei Verwendung in Programmen auch Textstellen etc.) abdecken, ohne diese einzeln auflisten zu müssen. Die passenden Dateinamen werden aus allen insgesamt vorhandenen Dateinamen nach einer Regel (z.B. sie beginnen mit **a**) heraus gefiltert.

Reguläre Ausdrücke verfolgen wie gesagt das gleiche Grundprinzip, sind jedoch wesentlich flexibler und mächtiger als Muster. In der Praxis muss man sie jedoch gedanklich vollkommen von den Mustern trennen.

Zum einen müssen reguläre Ausdrücke nicht auf das ganze Wort bzw. die ganze Zeile passen, es reicht wenn eine passende Zeichenkette darin enthalten ist. Wie man dieses Verhalten umgehen kann wird natürlich auch noch erklärt.

Zum anderen weisen reguläre Ausdrücke und Muster einige beachtliche Unterschiede bezüglich der Syntax auf. Sie sind also wie zwei verschiedene Sprachen zu behandeln. Viele Zeichen finden in beiden Sprachsystematiken Verwendung, allerdings mit unterschiedlichen Bedeutungen. Man muss also ein wenig aufpassen, um sie nicht zu verwechseln.

Damit die Shell diese nicht verwechselt, sollte man ihr bei regulären Ausdrücken sagen, dass sie diese nicht (als Muster) interpretieren, sondern unverändert an das Programm übergeben soll (das dann nach Möglichkeit RE beherrscht). Dazu maskiert man sie durch Einschließen in Hochkommata (`SHIFT + #`).

3 BRE und ERE

Um die Verwirrung vollkommen zu machen, gibt es noch zwei Arten von regulären Ausdrücken, die sich jedoch nur geringfügig voneinander unterscheiden (meist nur ein paar Backslashes mehr oder weniger), vergleichbar mit zwei Dialekten derselben Sprache:

* BRE (Basic Regular Expressions) - Einfache Reguläre Ausdrücke (auch:'ed-style')
* ERE (Extended Regular Expressions) - Erweiterte Reguläre Ausdrücke(auch: 'egrep-style')

Welches Programm welchen Dialekt versteht, wird meist durch einen Blick in die entsprechende Dokumentation ersichtlich. Falls dies nicht weiterhilft, muss man es stattdessen anhand einfacher Beispiele ausprobieren.

In der GNU-Implementierung regulärer Ausdrücke verstehen beide Dialekte jeweils alle Sprachelemente des anderen - sie werden lediglich ein wenig anders geschrieben. Deshalb werden im nächsten Abschnitt auch zunächst keine Unterschiede gemacht. Es wird jeweils die Schreibweise der ERE benutzt, da sie in der Regel etwas übersichtlicher ist.

Zum Ausprobieren eignet sich daher `egrep` ganz gut. Es gibt jeweils die Zeilen der Standardeingabe wieder aus, auf die ein Ausdruck passt. Möchte man wissen, welche Zeilen einer (vorher mit dem Lieblings-Editor erstellten) Datei passen, verwendet man:

```
user@linux / $ egrep 'MeinRegulärerAusdruck' testdatei1
```

Möchte man eine Zeile testen, ist man mit:

```
user@linux / $ echo 'Meine Testzeile' | egrep 'MeinRegulärerAusdruck'
```

wohl schneller.
Bleibt noch die Möglichkeit:

```
user@linux / $ egrep 'MeinRegulärerAusdruck'
```

aufzurufen. Man kann dann einfach Zeilen eingeben und mit `Enter` abschließen. Wenn sie wiederholt ausgegeben wird, so passt der Ausdruck, sonst nicht. Hat man genug rumgespielt, kann `egrep` mit einem beherzten Druck auf `Strg + C` beendet werden.

Soviel also zum Spiel mit ERE. Für einen guten Überblick und damit man auch weiß, wie die jeweiligen Ausdrücke in BRE aussehen, findet sich am Ende dieses Artikels eine Übersichtstabelle. Wer alleine damit klar kommt, kann sich freuen. Für alle anderen folgen ein paar Erläuterungen.

4 Und los geht's

Reiht man ein paar Zeichen aneinander, so stehen diese zunächst jeweils für sich selbst. **abc** findet also alles, was erst ein **a** dann ein **b** und schließlich ein **c** enthält - in genau dieser Reihenfolge und ohne sonstige Zeichen dazwischen. Zugegeben, das bekommt wohl auch jeder bessere Texteditor auf die Reihe.

Will man an einer Stelle mehrere alternative Zeichen finden, kann man das mit Hilfe von eckigen Klammern realisieren. **[aeiou]** findet also Vokale und **H[au]nd** sowohl die **Hand** als auch den besten Freund des Menschen (und Wörter, die diese Wörter enthalten).

Wenn Zeichen eine vorgegebene Reihenfolge haben (wie z.B. Buchstaben durch das Alphabet), so kann man auch Zeichenbereiche benennen. **[3-9]** findet alle Ziffern im Bereich von **3 bis 9**, **[A-Z]** alle Großbuchstaben.

Interessant ist auch die Möglichkeit, Zeichen auszuschließen. Man stellt dazu ein ^ hinter die öffnende eckige Klammer. **A[^r]t** findet also die **Art** (und derartige Zusammensetzungen) nicht, sehr wohl aber das **Amt** oder die **Abtei**. Genauso funktioniert das natürlich auch bei Zeichenbereichen (z.B. **[^0-9]** um Ziffern auszuschließen).

Eine Suche nach einer beliebigen Zeichenkette (in Mustern per *****) lässt sich in regulären Ausdrücken per **.*** realisieren. Dies ist ein zusammengesetzter Ausdruck. Der Punkt steht nämlich für ein einzelnes beliebiges Zeichen. Der Stern dient als Wiederholungsoperator, d.h. durch ihn gilt das vorangehende Zeichen (hier der Punkt) nicht bloß einmal, sondern beliebig oft (oder nie).

Der Stern ist aber nur eine Spezialform der Wiederholung. Die allgemeine Form wird mit geschweiften Klammern geschrieben. Sie schließen zwei durch ein Komma getrennte Zahlen ein, die angeben, wie oft das jeweilige Zeichen wiederholt werden darf. **a{2,5}** meint beispielsweise **2** bis **5** **a**'s hintereinander. Man kann die zweite Zahl auch weglassen. **a{2,}** bedeutet dann zwei oder mehr aufeinander folgende **a**'s. Lässt man schließlich noch das Komma weg, handelt es sich um eine genaue Angabe der Wiederholungen. **a{2}** bedeutet also exakt **zwei** **a**'s hintereinander.

Der Stern ist folglich eine Abkürzung für **{0,}**, analog steht das Fragezeichen **?** für **{0,1}** und das Plus **+** für **{1,}**. Noch einmal deutlich: **?** heißt, das Zeichen darf ein Mal stehen muss aber nicht (also höchstens ein Mal). **+** heißt, das Zeichen muss mindestens ein Mal stehen.

All diese Sprachelemente kann man auch miteinander kombinieren. **[0-9]{5}** hilft beispielsweise beim Auffinden von Postleitzahlen. (Wobei natürlich erstmal auch längere Zahlen gefunden werden; wir lernen auch noch, das zu verhindern) Beim Kombinieren muss man ab und an eine Reihenfolge festlegen. Dabei helfen uns runde Klammern.
Ein Beispiel:
bla+se findet 'blase', 'blaase', 'blaaase' usw.
Möchte man hingegen
'blase', 'blablase', 'blablablase' etc. finden, schreibt man **(bla)+se**.

Die Klammern sorgen dafür, dass sich die Wiederholung nicht bloß auf das Zeichen vor dem Plus bezieht (hier das **a**) sondern auf die ganze Silbe. Das funktioniert natürlich mit allen Operatoren.

Alternativen trennt man durch einen senkrechten Strich voneinander (`AltGr + <`). **(Schloss|B[ue]rg)** erkennt also sowohl das **Schloss** als auch die **Burg** und den **Berg**. Im Gegensatz zur eckigen Klammer kann man so nicht bloß alternative Zeichen angeben, sondern auch Auswahlmöglichkeiten zwischen kompletten regulären Ausdrücken. (Übrigens: die runden Klammern sind in diesem Beispiel zunächst nicht wichtig, sie sind jedoch später von Bedeutung um die Alternative zu begrenzen, wenn sie selbst nur ein Teilausdruck ist, wie im Superman-Beispiel am Ende des Artikels.)

Für kommende Experten sind auch die so genannten Backreferences (also Rückbezüge) unverzichtbar. Ein Beispiel: **([a-z])\1\1** findet drei gleiche aufeinander folgende Kleinbuchstaben, wie in **Schifffahrt** oder **Seeelefant**.

Die Eins steht dabei für das im ersten runden Klammerpaar gefundene Zeichen. Im Gegensatz zu den Wiederholungsoperatoren benennt der Rückbezug also das, was konkret gefunden wurde. Backreferences beziehen sich immer auf Ausdrücke in runden Klammern. Man kann bis zu neun von ihnen in einem Ausdruck verwenden. Sie sind von **\1** bis **\9** nummeriert; gezählt wird von links nach rechts. Bei Verschachtelungen sind die öffnenden Klammern ausschlaggebend.

Für den Fall, dass man den Suchbegriff am Anfang oder Ende der Zeile verankern möchte, stehen in regulären Ausdrücken die Zeichen ^ und **$** zur Verfügung. Der Suchbegriff "toll" findet sowohl die Zeilen 'toll' als auch 'tolles Beispiel' sowie 'richtig tolles Beispiel' und 'alle(s) toll'. **^toll** kann sich hingegen nur mit den ersten beiden Möglichkeiten anfreunden, **toll$** nur mit der ersten und der letzten und **^toll$** nur mit der ersten.

Was für Zeilen funktioniert, geht natürlich auch auf Wortebene. "car" könnte beispielsweise 'car', 'cartoon', 'oscar' und 'scary' finden. **\<car** gibt sich jedoch nur mit 'car' und 'cartoon' zufrieden. **car\>** mag nur 'car' und 'oscar'. **\<car\>**" verlangt schließlich ausdrücklich 'car'. Zu erwähnen wäre noch **\b**, dass sowohl auf Wortanfang als auch -ende passt und **\B**, dass überall da passt, wo **\b** das nicht tut.

4.1 Übersicht

Nun folgt wie versprochen eine Übersicht über die Sprachmittel von ERE und BRE. Die Möglichkeiten, die sich in der GNU-Implementierung ergeben, aber so nicht im entsprechenden POSIX-Standard gefordert werden, sind durch **-1-** gekennzeichnet.

BRE	ERE	Bedeutung
xy	xy	Ein 'x' gefolgt von einem 'y'
.	.	Ein beliebiges Zeichen
[xyz]	[xyz]	Ein 'x' oder ein 'y' oder ein 'z'
[a-z]	[a-z]	Ein beliebiges Zeichen, das in der Sortierreihenfolge (in diesem Fall das Alphabet) zwischen 'a' und 'z' liegt; 'a' und 'z' gelten ebenfalls
[^xyz]	[^xyz]	Ein beliebiges Zeichen, außer 'x', 'y' und 'z'
a\{2,5\}	a{2,5}	zwei bis fünf mal 'a' (hintereinander)
a\{2,\}	a{2,}	zwei mal 'a' oder öfter
a\{2\}	a{2}	genau zwei mal 'a'
a*	a*	beliebig oft 'a' (also auch kein mal)
a\+ -1-	a+	mindestens ein mal 'a'
a\? -1-	a?	höchstens ein mal 'a'
\(...\) -1-	(...)	Klammern legen die Reihenfolge der Operationen fest (und die Zählung für Rückbezüge)
a|b -1-	a|b	entweder 'a' oder 'b' (a und b können auch zusammengesetzte Ausdrücke sein)
\1 .. \9	\1 .. \9 -1-	Rückbezüge
^	^	Zeilenanfang (am Anfang des Ausdrucks)
$	$	Zeilenende (Am Ende des Ausdrucks)
\< -1-	\< -1-	Wortanfang
\> -1-	\> -1-	Wortende
\b -1-	\b -1-	Wortanfang oder -ende
\B -1-	\B -1-	Weder Wortanfang noch -ende
\. * \[\] \+ \?	\. * \[\] \+ \?	Jeweils das Zeichen '.', '*', '[', ']', '+' bzw. '?'

( ) | { }	\(\) | \{ \}	Jeweils das Zeichen '(', ')', '|', '{' bzw. '}'

5 Ein paar Beispiele

Zum Abschluss noch ein paar Beispiele zum besseren Verständnis:

5.1 Ab geht die Post

\<[0-9]{5}\>
Dieser Ausdruck sucht nach fünfstelligen Zahlen, wie zum Beispiel Postleitzahlen. **[0-9]** findet eine Ziffer zwischen 0 und 9. Durch die **{5}** dahinter muss etwas Derartiges fünfmal hintereinander gefunden werden. \< und \> sorgen schließlich dafür, dass um diese 5 Ziffern nichts herum stehen darf.

5.2 Mail me!

[^\b]\@[^\b]
Dies ist ein simpler Ausdruck zum Aufspüren von E-Mail-Adressen. **\b** bedeutet Wortanfang oder -ende. Durch das ^ zu Beginn der eckigen Klammerung wird es verneint.
So wird von obigem Ausdruck alles gefunden, was aus einem **@** besteht, das links und rechts von mindestens einem Wortzeichen umrahmt wird. (der \ vor dem **@** ist übrigens nicht Pflicht, aber in vielen Programmen hat das **@** eine Sonderbedeutung, und quoten schadet nicht)

5.3 IP-Adressen

([0-9]{3}\.){3}[0-9]{3}
Hierbei handelt es sich um einen einfachen IP-Adressen-Finder. Er sucht dreistellige Zahlen, die durch drei Punkte getrennt sind. Zur Verbesserung wäre noch eine Begrenzung auf ein Wort per \< und \> möglich.

5.4 Das IPerium schlägt zurück

\<(([01]?[0-9]{1,2}|2[0-4][0-9]|25[0-5])\.){3}([01]?[0-9]{1,2}|2[0-4][0-9]|25[0- 5])\>
Dieser doch schon recht komplexe IP-Finder ist eine ausgebaute Version des vorhergehenden Ausdrucks. Für eine schnelle Suche in Texten ist er natürlich viel zu lang. Er könnte vielleicht eher in einem Skript eingesetzt werden, um Benutzereingaben zu überprüfen.

Der Ausdruck lässt nämlich nur gültige IPv4-Adressen zu.
Ausnahme: 0.0.0.0 ist keine gültige IP-Adresse, wird aber trotzdem anerkannt.
Auch Sonderadressen (z.B. für Broadcasts) werden zunächst erkannt. Um dies zu verhindern kann der Ausdruck entsprechend angepasst werden.

Zum besseren Verständnis: Grundsätzlich ist der Ausdruck aufgebaut wie unser erster IP-Finder. Es wurden \< und \> ergänzt, damit beispielsweise **ag1234.122.33.4.012sd9** nicht gefunden wird. Anschließend wurde das **[0-9]{3}** durch eine präzisere Variante ersetzt. Sie besteht aus drei Alternativen:

[01]?[0-9]{1,2} findet alle ein- und zweistelligen Zahlen und dreistellige, die mit 1 beginnen. Jeweils eingeschlossen sind auch die Entsprechungen mit führenden Nullen (auffüllend bis drei Stellen).

Es sind jetzt also alle Zahlen von 0 bis 199 abgedeckt. **2[0-4][0-9}** findet alle Zahlen von 200 bis 249. **25[0-5]** alle von 250 bis 255. Damit ist die Suche komplett.

5.5 Emanzipierte Superfrauen

```
user@linux / $ sed -e 's/\(Super\|Spider\|Bat\)man/\1woman/g' testdatei
```

Dieser Befehl ersetzt in testdatei alle Wörter wie Spiderman, Batman und Superman durch ihre weiblichen Pendants.

Hier wurde, da wir `sed` benutzten, auf BRE zurückgegriffen. Zur Erläuterung: Die Option `-e` dient dazu, `sed` ein Kommando zu übergeben. In diesem Falle handelt es sich um das Ersetzen-Kommando (gekennzeichnet durch das `s` zu Beginn).

Die / dienen dazu die einzelnen Angaben voneinander zu trennen. Auf den ersten trennenden / folgt der zu suchende Ausdruck **\(Super\|Spider\|Bat\)man**.

Hinter dem nächsten / ein Ausdruck, der klar macht, was an den Fundstellen einzusetzen ist: **\1woman**. **\1** ist einen Rückbezug auf den Suchausdruck. Wenn beispielsweise 'Superman' ersetzt wird, wurde im Klammerpaar die Alternative 'Super' gefunden. Diese wird nun an die Stelle von **\1** gesetzt, gefolgt von 'woman'. Also wird aus 'Superman' kurzerhand 'Superwoman'.

Das `g` am Ende sorgt übrigens dafür, dass die Ersetzungen global, also in der gesamten Datei, vorgenommen werden.