Reguläre Ausdrücke

Eine allgemeine Einführung in die Benutzung regulärer Ausdrücke

Ein Kurs (nicht nur) für Anwender des Mailprogramms TheBat!

1. Inhaltsverzeichnis

2. Einleitung

Häufig, wenn man zu einem interessanten Effekt in einer TB-produzierten Mails etwas in den Listen fragt, hört man als Antwort: „Das macht man mit einem Regex!“ Und das Ergebnis sieht dann manchmal so aus:

```
%QUOTES="%SETPATTREGEXP=""(?is)(-----BEGIN PGP
SIGNED.*?\n(Hash:.*?\n)?\s*)?(.*?)(^(- --|--\n|-----BEGIN PGP SIGNATURE)|
\z)""%REGEXPBLINDMATCH=""%text""%SUBPATT=""3"""
```

So, oder so ähnlich sehen viele der kryptisch anmutenden Bandwürmer aus, die andere immer als Regex oder reguläre Ausdrücke bezeichnen. Ein scheinbar wirres, wahlloses Aneinanderreihen von Zeichen. So -wirr, willkürlich und kryptisch- kamen mir diese Lösungen der Regex-Freaks vor, bis Januk Aggarwal aus der TBTECH (ihm gilt mein besonderer Dank) eine kleine Einführung gab und mir mein Arbeitskollege Alfred Rübartsch den Jeffrey Friedl "Reguläre Ausdrücke" als Lektüre für eine Dienstreise mitgab. Ich bin auch heute kein Experte in Regex, aber zumindest kann ich das ein oder andere erklären und vielleicht, als fortgeschrittener Anfänger anderen Anfängern eine Starthilfe geben.

Dieses Tutorial soll euch diese Regexe etwas näher bringen; lasst uns am Ende mal schauen, ob wir das obige Regex erklären können.

3. Reguläre Ausdrücke

3.1. Was genau heißt „Regulärer Ausdruck“

Regexe werden nicht nur bei TB verwendet. Vielmehr finden sich Regexe in vielen UNIX-Tools, in Programmiersprachen wie Perl (Practical Extraction and Report Language) oder sogar im Editor UltraEdit. Mein Perl-Buch sagt zu diesem Begriff, dass er auf den ersten Blick unsinnig erscheint (bei mir auch auf dem zweiten), da es sich nicht um richtige Ausdrücke handelt und außerdem kaum zu erklären ist, was an ihnen eigentlich „regulär“ ist. Nehmen wir einfach hin, dass der Begriff „reguläre Ausdrücke“ der formalen Algebra entstammt und in der Tat sind Regexe ein Teil der Mathematik.

Am einfachsten umschreibt man reguläre Ausdrücke wohl als Suchmuster für „Pattern Matching“ (Suchmusterübereinstimmung). Jeder von uns, der auf DOS-Ebene oder im Explorer mal nach Dateien gesucht hat, hat solche Suchmuster verwendet:

```
dir *.doc
```

```
copy *.??t c:\temp
```

Hier werden Suchmuster bestehend aus Sternchen und Fragezeichen verwendet, um die Auswahl von Dateien einzugrenzen. Im ersten Beispiel sollten alle Dateien gelistet werden, welche die Endung `.doc` haben. Im zweiten Beispiel sollten nur Dateien kopiert werden, die eine dreibuchstabige Endung haben und als letzten Buchstaben ein `t` aufweisen.

Aber diese "Regexe" sind lediglich reine Platzhalter und trivial. Sie sind in keinster Weise so mächtig wie Regexe in TB, welche – wie wir bald sehen werden- eben nicht nur Platzhalter für Zeichen sind.

4. Einfache Muster

Um im folgenden Beispiele für Regexe geben zu können, müssen wir uns auf eine Darstellungsform einigen. Ich werde die Regexe mit Anführungszeichen begrenzen. Wollt ihr sie ausprobieren, so müsst ihr die Eintragungen zwischen den "-Zeichen verwenden. Testen? Ja, man kann die Wirksamkeit der Muster testen: Dazu geht zum Regex-Coach und installiert ihn. Bedienhinweise entnehmt ihr bitte dem Produkt selbst. Für Benutzer der Editoren Weaverslave oder Ultraedit gibt es Plugins bzw. vorhandene Funktionen zur Nutzung der Regexe im Editor. Achtung, zum Teil haben diese Programme eine eigene Syntax. Bitte vorher deren Hilfe lesen!

4.1. Einfache bekannte Zeichen

Fangen wir mit einfachen Suchmustern an: `"dies oder das"`

Ja, das ist schon ein Regex: es findet die Zeichenfolge 'dies oder das' in einem Text und zwar genau die. Nein, das 'oder' bewirkt nicht, dass entweder 'dies' oder 'das' gefunden wird, sondern genau nur die Zeichenfolge in den Anführungszeichen.

Regexe sind stur: sie suchen genau das, was man ihnen aufträgt. Sie unterscheiden Groß- und Kleinschreibung und sie interessieren sich nicht für Wortgrenzen, wenn das niemand sagt. Im obigen Beispiel wird die Zeichenfolge in 'Das Para**dies oder das** Weib' gefunden.

4.2. Suche nach Metazeichen

Mit einem Regex lässt sich nach allen beliebigen Zeichen – alphanumerische, hexadezimale, binäre usw. – suchen. Eine kleine, aber wichtige Ausnahme bilden Zeichen, die das Regex als besondere Zeichen, den Metazeichen, einsetzt. Diese Metazeichen sind:

`* + ? . ( ) [ ] { } \ / | ^ $`

(Hallo Experten: Ihr habt ja recht. Ich habe da ein bisschen geschummelt. Nicht alle sind tatsächlich Metazeichen. Aber lasst uns mal einfach annehmen, es wäre so. Ich zeige später, warum ich diese Art der Definition von Metazeichen bevorzuge.)

Ihre Bedeutung werden wir im Verlaufe des Tutorials noch kennen lernen, dazu also später. Nur soviel vorweg: wer diese Zeichen in ihrem ursprünglichen Sinn suchen will, also literal, der muss dem Regex dies in irgendeiner Form erkennbar machen. Dem Metazeichen muss ein Escape-Zeichen vorweg gestellt werden: es ist der Backslash \

Wird also nach einem Fragezeichen gesucht, so muss das Regex "\?" lauten. Wird nach dem Schrägstrich gesucht, so muss es "\/" heißen. Na ja, auch wenn es komisch aussieht, aber wer ein Backslash sucht, muss halt zwei solche eingeben: "\\"

4.3. Einfache unbekannte Zeichen

Das erste Metazeichen ist der Punkt "`.`" Er steht für genau ein beliebiges Zeichen, egal was dieses Zeichen darstellt. (Na, schon wieder ein paar Experten, die mehr wissen *gg*? Lasst uns zu Ausnahmen später kommen, ok?) "`M.ier`" findet somit 'Maier', 'Meier' und 'Maiering', aber nicht 'Manieren'. "`H..s`" findet sowohl 'Hans' als auch 'Haus'; es wird aber nicht 'Hase' finden. Das Wort 'Hirse' wird bis auf das 'e' gefunden; das Suchmuster passt genau auf 'Hirs'.

Zu einem späteren Zeitpunkt werden wir sehen, dass man weitere Metazeichen verwenden kann, um mehr als nur ein unbekanntes Zeichen suchen zu können, ohne es mehrfach mit einem "`.`" zu kennzeichnen.

4.4. Zeichengruppen und -klassen

Ein weiteres mächtiges Werkzeug sind die Metazeichen für Zeichengruppen. Hier unterscheiden wir gleich mehrere Möglichkeiten. Beginnen wir mit den einfachen:

"`\d`" steht für eine Ziffer (digit). "`\d\d`" sucht also nach zwei aufeinander folgenden Ziffern.

"`\w`" steht für einen beliebigen Buchstaben, Ziffer oder einen Unterstrich, auch alphanumerische Zeichen genannt.

Auf diese Weise lassen sich schon komplexere Suchmuster aufbauen.

"`Re \[\d\]:`" sucht also einen String nach der Zeichenkette 'Re' gefolgt von einem Leerzeichen, einer öffnenden eckigen Klammer, einer beliebigen Ziffer und einer schließenden eckigen Klammer mit einem Doppelpunkt als Abschluss. Aha! Sieht aus wie eine Subject-Zeile, bei dem in der Antwortvorlage das %SINGLERE vergessen wurde....

Die Regex bieten auch das jeweilige Gegenstück zu den beiden oben genannten: "`\W`" und "`\D`"

Hierbei steht \W für jedes beliebige nicht-alphanumerische Zeichen und \D für jedes Zeichen, das keine Ziffer ist.

Eine weitere elegante Methode Zeichengruppen zu definieren ist die Verwendung von [ ] für Zeichenklassen. Mit dieser eckigen Klammer wird nur genau ein Zeichen gesucht, unabhängig, wie viele Zeichen in der Klammer aufgeführt werden: "`[AEX]`" Diese Kombination sucht Zeichenketten, die aus nur genau einem Zeichen bestehen, welches auch noch nur A, E oder X heißen muss. Will man ganze Bereiche angeben, so muss man nicht alle Elemente einzeln aufführen, vielmehr darf man das erste und das letzte Element mit einem Bindestrich verbinden: "`[e-z]`" heißt alle Buchstaben von e beginnend bis z sollen gefunden werden.

Eine sehr mächtige Methode: "`[0-3][0-9]\.[0-1][0-9]\.`" Hiermit werden nur Datumsangaben im Format TT.MM. gefunden. Andere Zahlenkombinationen, die kein Datum sein können, wie 47.35., werden nicht gefunden (Ja, aufmerksame Leser haben festgestellt, dass mein Regex oben immerhin auch den 39.19. findet, was definitiv kein irdisches Datum ist. Dazu kommen wir nachher, es fehlt uns noch etwas...).

Sehr praktisch ist hieran auch, dass man mit einem Schlag das Suchkriterium negieren kann, also nach dem Motto: "Finde alle Zeichen, sofern sie keine 1, 2, 3 oder 4 sind!" Das Regex lautet: `"[^1-4]"` Die Negation wird mit einem ^ bewirkt. Hoppla, das sollten wir uns merken, denn wir werden nachher sehen, dass dieses ^ eine ganz andere Bedeutung hat, wenn es nicht in eckigen Klammern steht.

4.5. Überblick über dieses Kapitel

In diesem Kapitel haben wir einfache Suchmuster kennen gelernt:

- direkt eingegebene Zeichenketten werden als solche gesucht. "er" sucht nach den aufeinander folgenden Buchstaben e und r . Groß- und Kleinschreibung wird unterschieden
- Regexe verwenden Metazeichen, nach denen man nur dann literal suchen kann, wenn man ein Backslash voranstellt: `* + ? . ( ) [ ] { } \ / | ^ $`
- der Punkt "." dient dazu, nach einem beliebigen unbekannten Zeichen zu suchen. Sucht man nach dem Punkt als Zeichen, so stellt man ein Backslash voran "`\.`"
- Regexe verwenden Zeichengruppen wie

  \d für Ziffern ([0-9])

  \D für nicht-Ziffern ([^0-9])

  \w für alphanumerische Zeichen ([a-zA-Z0-9_])

  \W für nicht-alphanumerische Zeichen ([^a-zA-Z0-9_])
- Zeichenklassen können durch Angabe in eckigen Klammern selbst definiert werden "[A-Z]" Diese Angabe kann durch ein ^ als erstes Zeichen in den eckigen Klammern negiert werden.

Aufgaben

Was sucht die folgenden Regexe?

`"\d\d\.\d\d\.\d\d\d\d"`

`"\w\w\w, \d\d \w\w\w \d\d\d\d"`

`".. \[[0-9]\]:"`

`"[a-zA-Z]"`

Lösung für den ersten Fall: zwei Ziffern. Es folgt der Backslash und dann erst der Punkt, das bedeutet, es soll der Punkt literal, also als Punkt und nicht als beliebiges Zeichen gesucht werden. Erneut sollen zwei Ziffern mit einem Punkt folgen und abschließend nochmals eine vierstellige Zahl. Ein Datum in der Form TT.MM.JJJJ

Der zweite Fall sucht nach drei alphanumerischen Zeichen mit einem Komma, ein Leerzeichen, zwei Ziffern, wieder drei alphanumerischen Zeichen mit ein Leerzeichen und abschließend eine vierstellige Zahl. Auch das sieht nach einem Datum aus, aber in der anglo-amerikanischen Schreibweise: Tue, 19 Feb 2002. Leider ist dieses Regex nicht optimal: es erkennt nur Daten mit zweistelligen Tageszahlen. Wir werden etwas später sehen, wie man das Regex modifiziert, um sowohl einstellige als auch zweistellige Tageszahlen zu finden.

Der dritte Fall sucht nach zwei x-beliebigen Zeichen auf die ein Leerzeichen und eine geöffnete eckige Klammer folgt. Als nächstes wird die eckige Klammer nicht mehr von einem Backslash begleitet, hier beginnt also eine Zeichengruppe. In dieser Zeichengruppe sind alle Ziffern von 0 bis 9 erlaubt. Nach der Ziffer soll eine geschlossene eckige Klammer und ein Doppelpunkt folgen. Also zum Beispiel: 'Re [2]:'

Im letzten Fall soll das Regex nur ein einziges Zeichen finden, das nur aus Klein- oder Großbuchstaben bestehen darf. Warum an dieser Stelle eigentlich kein "\w"? Nun, das würde auch Unterstriche beinhalten, was ggf. ungewünscht ist.

5. Komplexere Suchmuster

So, das war ja einfach. Allerdings auch wenig interessant. Wenn sich die Möglichkeiten von Regexen in den einfachen Suchmustern erschöpften, wäre es wohl kaum wert, sie in einem Tutorial vorzustellen.

Es muss also mehr geben! Fangen wir mal an:

5.1. Zeilengrenzen

Außer einfach den Text irgendwo zu suchen, kann man ein Regex veranlassen, an bestimmten Stellen zu suchen. Dafür gibt es Metazeichen: ^ $ (Ja, liebe Experten, lasst uns für den Anfang die Zeichenkette als eine Zeile betrachten, ok?)

Konkret: "`^dies oder das`". Die Zeichenfolge wird nur gefunden, wenn das Wort "dies" unmittelbar am Anfang einer Zeile steht und von " oder das" gefolgt wird.

Oder: "`Das ist das Ende$`" wird genau nur am Zeilenende gefunden. Es ist egal, was vor der Zeichenfolge erscheint, aber "Das ist das Ende" muss mit der Zeile enden.

Diese beiden Metazeichen sollten unter anderem eingesetzt werden, um das Regex zu beschleunigen. Bei TB ist es zwar nicht so wichtig, da man in aller Regel keine großen Datenmengen bearbeitet, dennoch kann es nicht schaden. Warum wird das Regex schneller? Nehmen wir mal wieder unser Beispiel "`^dies oder das`" und wenden das auf den Text 'Es war einmal' an: die Regex-Maschine prüft zunächst, ob es sich bei dem ersten Zeichen um den Zeilenanfang handelt: das liefert WAHR zurück. Dann prüft die Maschine das nächste Zeichen darauf, ob es ein "d" ist, was natürlich fehlschlägt. Die Suche wird sofort an dieser Stelle mit dem Ergebnis FALSCH abgebrochen. Ohne das ^-Zeichen hätte die Maschine das zweite, dritte, vierte usw. Zeichen geprüft und versucht "dies oder das" im String zu finden, um am Ende festzustellen, dass die Zeichenfolge nicht existiert.

5.2. Wortgrenzen

Neben den Zeilengrenzen sind Wortgrenzen interessant, allerdings selten beachtet. Mit "`\b`" wird das Regex veranlasst, den betreffenden String mit einer Wortgrenze zu suchen: "`\bdies oder das`"

Kennen wir schon, nicht wahr? Das Beispiel vom Anfang, in dem diese beiden Wörter in 'Paradies oder das Weib' gefunden wurde, würde nun nicht mehr ge-

funden. Ich erinnere mich an eine Diskussion in der TB-dt vor längerer Zeit, in der gefragt wurde, was der Sinn des \b sein solle, da man doch auch die Wortgrenze über ein Leerzeichen ermitteln könne. Das ist nur bedingt richtig, denn Wörter enden auch an Interpunktionszeichen: "`er `" findet in der Tat 'der Regen', aber 'wieder.' als Satzabschluss würde nicht gefunden. Dies lässt sich mit "`er\b`" umgehen.

Natürlich ist das auch wieder negierbar: "`\B`" bedeutet, dass diese Zeichenkette nicht an Wortgrenzen gefunden werden soll.

Auch hierzu ein Beispiel: "`Re\B.`" Also, es soll die Zeichenkette 'Re', aber nur, wenn sie nicht die Wortgrenze darstellt, aber von einem beliebigen Zeichen gefolgt wird. Wenden wir das mal auf 'Re: oder Reply:' an. Der Test in TB ergibt 'Rep'. Tauscht mal \B gegen \b aus und es wird 'Re:' gefunden. Alles klar?

5.3. Alternativen

Wir erinnern uns an das erste Regex "`dies oder das`" in diesem Kurs? Dazu schrieb ich den eigentlich überflüssigen Zusatz, dass dies nicht bedeutet, das Regex würde entweder nach "`dies`" oder nach "`das`" suchen. Na ja, so überflüssig war das nicht: ich wollte doch schließlich irgendwas haben, was eine Einleitung für diesen Abschnitt bietet *g*. Genau um diese Alternativen, nämlich dem ODER, dreht es sich nun.

Um nach alternativen Zeichenketten zu suchen, bieten die Regexe ein besonderes Zeichen: "`|`" den Strich, den einige von euch auch als Pipe-Symbol kennen. Konkretes Beispiel: "`dies|das`" Das Regex prüft, ob es die Zeichenkette 'dies' findet. Wenn es diese nicht findet, sucht es nach der zweiten Alternative.

Was passiert aber, wenn beide Alternativen im Text enthalten sind? Nun, man sollte glauben, dass die Alternative gefunden wird, die als erste im Regex steht. Dem ist aber nicht so! Vielmehr wird als gefundener Text das erste Auftreten einer der Alternativen in der Zeichenkette zurückgegeben!

Beispiel:

Gegeben sei das Regex "`das|dies|jenes`" und die Zeichenkette 'Es war jenes Auto, das wir diesseits der Böschung geborgen haben' (Die Beispielzeichenketten sind nicht immer hochliterarisch, aber sie sollten dennoch helfen ;-)) Als gefundener Text wird das 'jenes' zurückgeliefert. Probiert es mal im Regex-Tester aus. Diese Besonderheit wird noch mal für uns interessant, wenn wir zu dem Abschnitt kommen, in welchem wir uns Textstücke mit dem Regex merken wollen.

Solche Alternativen sind auch kombinierbar: "`^aw:|^wg:|^fwd:`" bedeutet, das nach diesen drei Möglichkeiten alternativ gesucht wird. In allen drei Fällen wird nach dem Zeilenanfang gesucht und hinter jeder Zeichenkombination steht ein Doppelpunkt . Ihr habt recht: das müsste man vereinfachen können. Und genau wie in der Mathematik kann ich hier Klammern setzen und das Regex vereinfachen und beschleunigen: "`^(aw|wg|fwd):`"

Derartige Vereinfachungen müssen nicht immer einfacher zu lesen sein: "`d(ie|a)s`" wäre die Vereinfachung zu unserem Eingangsbeispiel in diesem Abschnitt, was zwar richtig wäre, aber nicht gerade den Lesefluss unterstützt ;-)

5.4. Besondere Klassen

Einige der Zeichenklassen haben wir schon oben kennen gelernt. Ich möchte hier noch ein paar von unterschiedlicher Bedeutung nachschieben.

In fast jedem Regex werdet ihr die Zeichenklasse "\s" finden. Sie steht für so genannte Whitespace-Zeichen, also alle Zeichen, die einen weißen Raum auf den Bildschirm zeigen: Leerzeichen, Tabulatoren, die Befehle 'Neue Zeile', 'Wagenrücklauf', 'Blattvorschub'. Es wird genügen, sich zu merken, dass jede leere Stelle **in** einer Zeichenkette hiervon gefunden wird. Natürlich kann man dies auch negieren: "\S" passt auf jedes andere Zeichen, das eben nicht weißen Platz beansprucht.

"\A"wird seltener eingesetzt: es passt auf den Anfang einer Zeichenkette. Dies ist nicht der Beginn einer Zeile, denn das hätten wir mit "^" gesucht. Analog hierzu gibt es "\Z": dieses findet das Ende der Zeichenkette oder anders gesagt: es findet das Zeichen unmittelbar vor dem 'Neue Zeile'-Befehl (den ihr sicher aus TB kennt: \n). Und erneut: dies ist nicht das Ende einer Zeile, denn das hätten wir mit "$" gesucht. Wir werden den Unterschied später sehen, wenn wir zu den Optionen kommen. Sorry, aber ihr müsst euch etwas gedulden *g*.

5.5. Übersicht über dieses Kapitel

Dieser Abschnitt hat uns ein paar neue Möglichkeiten zur Suchmusterdefinition gezeigt:

- Zeilengrenzen haben eigene Suchkriterien: ^ für den Zeilenanfang und $ für das Zeilenende.
- Für Wortgrenzen gibt es "\b" mit dem nach Zeichenketten gesucht werden kann, die den Abschluss eines Wortes bilden. "\B" steht für Zeichenketten, die nicht Wortgrenzen darstellen.
- Man kann Zeichenketten als Alternativen definieren. Dazu trennt man jede Alternative mit dem Strich "|" ab. Gleiche Zeichen an gleichen Stellen innerhalb der Alternativen, lassen sich wie in der Mathematik vor oder hinter eine Klammer ziehen: "^(Re|Aw|Fwd)" Für alle drei Alternativen gilt, dass sie am Anfang einer Zeile stehen müssen.
- Leerzeichen oder Tabulatoren werden als so genannte Whitespace-Character bezeichnet, für die ein spezielles Suchmuster existiert: \s Negiert schreibt man für diese Gruppe \S
- Auch Anfang und Ende einer Zeichenkette sind suchbar: \A und \Z

Aufgaben:

1. Gegeben: "(R.:$|^R.:)" sowie die Zeichenkette 'Ra: oder Re:'. Was wird gefunden?

2. Ich will die Zeichenkette 'Re:' am Zeilenanfang finden; allerdings auch dann, wenn ihr noch der Reply-Zähler folgt, also 'Re[2]:'. Wie muss das Regex nach unserem bisherigen Stand aussehen?

3. Wir versuchen mal, ein Regex zu basteln, das die Zeichenfolge 'Re' am Anfang einer Zeile oder ')' am Zeilenende findet.

4. Was bedeuten die folgenden Suchmuster?
   a) "^"
   b) "^x$"
   c) "^$"

Im ersten Beispiel wird 'Ra:' gefunden. Das hatten wir erwartet: gefunden wird die Alternative, die in der Zeichenkette zuerst erscheint.

Hoppla, das **zweite Beispiel** ist ja schon etwas anspruchsvoller: "^(Re|Re\[\d\]):" Ok, man kann es vereinfachen: "^Re(|\[\d\]):" Um nach *nichts* hinter dem 'Re' und vor dem ':' zu suchen, haben wir auch einfach *nichts* als erste Alternative eingetragen. Wir werden im folgenden Kapitel feststellen, dass man dieses auch anders suchen kann.

Drittes Beispiel:

"(Re|\)$)" wäre die Lösung. Ihr hattet hoffentlich an den Backslash vor der Klammer gedacht? Prima, dann habt ihr aufgepasst *g**. Probiert es mal in TB aus. Ihr werdet erkennen, dass das Regex aus der Zeichenkette 'Re[2]: bladibla (was: noch mehr bla)' nur genau 'Re' findet, denn da war die Bedingung erfüllt. Ändert den Zeilenanfang, indem ihr irgendetwas eintragt und schon wird das 'Re' nicht mehr gefunden. Stattdessen wird die abschließende runde Klammer gefunden.

Viertes Beispiel:

Das erste Muster sucht nach allen Texten, die einen Zeilenanfang haben oder am Zeilenanfang beginnen. Es werden also alle Strings, sogar der leere String gefunden!

Das zweite Muster sucht nach Zeilen, in denen nur der Buchstabe x steht, direkt nach Zeilenbeginn und am Ende der Zeile.

Und das letzte Muster sucht einfach nach allen Texten, die Zeilenanfang und -ende haben, aber nichts dazwischen. Es werden leere Zeilen gesucht.

6. Spezielles - Teil 1

Bislang war es eigentlich recht geruhsam. Dieses Kapitel allerdings hat es in sich. Arbeitet es langsam und mit Geduld durch. Es ist der anspruchsvollere Bereich, der aber auch der interessantere ist, da man nun endlich vernünftige Regexe bauen kann. Testet nach Möglichkeit selber mit dem Regex-Tester die Beispiele durch.

6.1. Quantifizierer

So, bislang konnten wir nach einzelnen Zeichen, Zeichengruppen oder Zeichenklassen und Bereichen suchen. Wir haben es geschafft, alternativ nach dem Einen oder Anderen zu suchen. Aber eines fehlt uns noch: nämlich nach Zeichenwiederholungen zu suchen.

Weiter oben gab es das Beispiel, mit dem nach einem deutschen Datum gesucht wurde:

"\d\d\.\d\d\.\d\d\d\d"

Für jede Ziffer musste man "\d" schreiben. Gibt es da nichts einfacheres? Doch, klar! Natürlich kann man so etwas weniger aufwändig schreiben und mit Zeichenwiederholungen arbeiten. Aber fangen wir mal vorn bei den Quantifizierern an:

+ * ? sind die wichtigsten Quantifizierer.

Das "+"-Zeichen bedeutet, dass das Zeichen vor dem Plus-Zeichen mindestens einmal an der Stelle in der Zeichenkette vorkommen muss! "pa+r" passt somit auf 'paar', aber auch auf 'par' oder 'paaaaar'.

"Re:\s+" heißt zum Beispiel, dass mindestens ein Whitespace-Zeichen auf das 'Re:' folgen muss, um gefunden zu werden. Ich höre da schon wieder Experten rufen, dass die Benutzung von Quantifizierern nicht nur auf Zeichen beschränkt ist. Stimmt, man kann sie auch auf Metazeichen, Zeichenklassen oder anderen Elementen anwenden, die wir aber erst noch lernen müssen.

Mit dem Stern wird festgelegt, dass das Zeichen vor dem Stern beliebig häufig oder gar nicht existieren muss! Ooops, wofür soll das denn gut sein: 'gar nicht existieren muss'?

Am besten schauen wir uns das an einem Beispiel an: "Re:\s*\w+"

Na, sieht ja schon langsam so aus, wie die kryptischen Teile der Profis. Was will das Regex?

Finde ein 'Re' gefolgt von einem Doppelpunkt. Danach können beliebig viele Leerzeichen (oder Whitespaces) folgen oder auch gar keine. Warum das? Da gehört doch normalerweise sowieso ein Leerzeichen hin! Na, wir wollen auch manuell verhunzte Reply-Zähler finden und sollte der Schreiber das Leerzeichen gelöscht haben, so soll das Ganze trotzdem von der Regex-Maschine gefunden werden; es soll uns nicht stören. Danach aber muss mindestens ein alphanumerisches Zeichen folgen.

Achtung: hier wird gern ein Fehler gemacht. Nehmen wir mal die folgende Aufgabe: Es sollen nur Zeilen gefunden werden, in denen beliebig viele Ziffern stehen. Man kann folgendes hin und wieder als Lösung sehen: "^[0-9]*$"

Tatsächlich aber findet das Regex auch leere Zeilen, denn der Stern steht ja auch für 'Häufigkeit 0'. Will man also sicherstellen, dass mindestens eine Ziffer in der Zeile auftritt, so muss das "+"-Zeichen verwendet werden: "^[0-9]+$".

Das Fragezeichen bedeutet dagegen, dass das Zeichen davor höchstens einmal vorkommen darf, aber nicht muss, also mit der 'Häufigkeit 0'. "H..?se" findet 'Hase', aber auch 'Hirse'.

Zeichenwiederholungen lassen sich aber auch anders angeben: "{x,y}" Hiermit lassen sich detailliert Häufigkeiten definieren. "x" steht für die Mindestanzahl und "y" für die Maximalzahl, die ein Zeichen vorkommen darf. "\d{2,4}" bedeutet, dass nur das gefunden wird, was mindestens zwei aber höchstens vier Ziffern hat.

Lässt man das "y" weg, also "{x,}", so muss das Zeichen mindestens x-mal vorkommen, ohne obere Grenze.

Lässt man ",y" weg, also in der Form "{x}", so wird dies als absolute Zahl gewertet: nicht mehr, aber auch nicht weniger häufig darf das Zeichen vorkommen. Nun können wir unser Datumssuchmuster umschreiben: "\d{2}\.\d{2}\.\d{4}"

Die zu Beginn beschriebenen Quantifizierer "?+*" sind eigentlich Spezialschreibweisen der folgenden:

`{0,1} = ?`

`{1,} = +`

`{0,} = *`

Bevor wir zu einem weiteren Problem der Quantifizierer kommen, müssen wir erst die runden Klammern als Gruppierungsmöglichkeit einführen:

6.2. Gruppierung, Subpattern und noch mal Quantifizierer

Gruppierung

Im Abschnitt über Alternativen begegnete uns erstmalig die runde Klammer als Metazeichen. Sie wurde dort wie in der Mathematik verwendet: gleiche Elemente wurden ausgeklammert.

Sie kann aber auch dazu genutzt werden, Suchmuster zu gruppieren, zum Beispiel um darauf einen Quantifizierer anzuwenden: "`foo(bar)?`" findet 'foo' und auch 'foobar'.

Anderes Beispiel:

"`Re\s*(\[\d+\])?:`" Schon wieder unser Reply-Zähler! Aber diesmal schon recht professionell. Wir suchen nach der Zeichenkette 'Re'. Dieser kann, nach beliebigen Whitespace-Zeichen, die eckige Klammer mit dem Zähler folgen, sie muss aber nicht. Die Whitespace-Zeichen haben wir mit dem Stern versehen, damit die Zeichenkette auch dann gefunden wird, wenn der Verfasser einfach das Leerzeichen manuell eingefügt hat, seine Leertaste prellt und mehr als ein Leerzeichen folgt oder er gar nichts gemacht hat und deswegen dort auch kein Leerzeichen steht. Normalerweise dürfte dort nämlich gar kein Leerzeichen stehen. Aber was heißt hier schon 'normalerweise'? Zusätzlich erlauben wir sogar, dass der Zähler astronomisch groß werden darf, denn wir geben mit dem "+"-Zeichen an, dass mindestens eine Ziffer in der Klammer stehen muss, allerdings dürfen es auch beliebig viele sein.

Gefunden wird also:

'Re:'

'Re [1]:'

'Re[123]:'

Nicht gefunden wird dagegen 'Re []:' Aufgabe für zwischendurch: Was müsste am Regex geändert werden, damit auch diese Reply-Zähler gefunden werden?

Lösung: Das "+"-Zeichen müsste nur gegen einen Stern ausgetauscht werden: "`Re\s*(\[\d*\])?:`"

Aus der Zeichenkette 'Re [1]: [3]:' findet das Regex allerdings 'Re [1]:' Der zweite Teil passt nicht auf das gesamte Suchmuster. Wollen wir solche verkorksten Reply-Zähler finden, müssen wir uns noch mal an das Regex machen. Es soll nun zusätzlich beliebig viele oder keine eckige Klammern finden, die auch noch einen Doppelpunkt haben können "`(\[\d+\]:\s*)*`“. Am Schluss soll noch mindestens ein Doppelpunkt folgen ":+“

"Re\s*(\[\d+\]:\s*)*:+"

Damit werden nicht alle kaputten Reply-Zähler gefunden. Da gibt es eine Unmenge an denkbaren Kombinationen. Wer will, kann dies mal selbst erarbeiten. An dieser Stelle sei aber darauf hingewiesen, dass es nicht immer Sinn macht, das perfekte Regex zu suchen. Der Aufwand steigt sehr, nur um die unwahrscheinlichsten Zeichenkombinationen zu finden, die so gut wie nie auftreten. Man erkauft sich den Perfektionismus mit unlesbaren Regex-Code und einer größeren Fehleranfälligkeit, sollte man mal das Regex geringfügig ändern wollen. Wie immer im Leben, gilt auch hier: "Man sollte mal Fünfe gerade sein lassen!"

Als weiteres Beispiel soll wieder unsere Datumssuche dienen: "\d{2}\.\d{2}\.\d{4}" Wie zu sehen ist, wiederholt sich der Anfang "\d{2}\." Genau das lässt sich mit Klammern verkürzen: "(\d{2}\.){2}\d{4}" Das erste Element des Suchmusters – in den runden Klammern - soll nun mindestens zweimal vorkommen. Dies entspricht genau der Kombination '01.02.'

Auch weiterhin gilt, dass dies nicht die optimale Version ist, da die Tages- und Monatsangaben zweistellig sein müssen und außerdem unsinnige Zahlenkombinationen als Datum erkannt werden könnten. Aber lasst uns noch ein paar weitere Sachen kennen lernen, dann werden wir das Problem in einer der Aufgaben angehen.

Subpattern

Die Klammersetzung hat noch einen ganz anderen Effekt, der in vielen TB-Regexe zum Einsatz kommt. Zeichenketten, die von eingeklammerten Suchmustern gefunden wurden, werden in eine temporäre Variable (Subpattern, SubPatt in TB genannt) für spätere Zwecke zwischengespeichert. Das schaut man sich am besten in einigen Beispielen an:

'erika.mustermann@beispiel.de'

Wir verwenden das folgende Regex "(\w+)\.(\w+)@.*" an. Die erste Klammer passt auf 'erika', die zweite 'mustermann'. Und genau diese beiden Zeichenketten landen in den Subpattern1 bzw. 2.

Oder:

"(\d+\.)(\d+\.)" Dieses Regex wenden wir auf die Zeichenkette '22.02.' an. Im ersten Subpattern wird '22.' gespeichert, im zweiten '02.'.

Wie stellt man eigentlich fest, was das erste Subpattern ist? In obigen Fall ist das ja einfach, was aber, wenn das Regex so aussieht: "Re\s*(\[(\d+)\])*:" Das was von dem ersten runden Klammerpaar umfasst wird, wird in Subpattern 1 gespeichert, das von dem zweiten im Subpattern 2 und so weiter. In unserem Beispiel würde aus dem String 'Re [4]:'

Subpattern 1 = '[4]'

Subpattern 2 = '4'

D.h.: mit jeder neuen geöffneten Klammer wird eine neue Variable erzeugt. Will man das verhindern, also eine Gruppierung vornehmen, ohne dass deren Inhalt in einem

Subpattern landet, so kann dies durch Einfügen von "?:" hinter der öffnenden Klammer erreicht werden: "(?:...)"

Was steht denn in der Subpattern-Variable, wenn eine Klammer mit einem Quantifizierer versehen ist? Nehmen wir mal wieder unser Datums-Regex:
"(\d{2}\.){2}\d{4}"

Wird es nun auf das Datum '19.02.2001' angewendet, so wird das erste Subpattern bei der '19.' gefunden. Die Regex-Maschine versucht nun einen weiteren Teil der Zeichenkette mit dem gleichen Muster (Subpattern) zu finden. Ist sie erfolgreich, dann legt sie diesen Wert erneut in das Subpattern ab. Mit anderen Worten: der zweite Fund überschreibt den ersten. Das Ergebnis ist, dass in unserem Beispiel '02.' im Subpattern enthalten ist.

Wir werden später in den Makros sehen, wie das zu nutzen ist. Der Regex-Coach zeigt euch die Subpattern an.

Und noch mal Quantifizierer

Kommen wir zu einer Besonderheit der Quantifizierer. Einige von ihnen haben eine gewisse 'menschliche' Neigung: sie sind gierig! Was das bedeutet? Schauen wir uns mal den folgenden String an:

"Die Abkürzung 'ISP' heißt 'Internet Service Provider'."

Wir wollen ein Regex, das beliebigen Text in Hochkommata findet und als Subpattern ablegt.

"(.*)'(.*)'.*"

Und was steht im Subpattern 2? "Internet Service Provider" Ooops, ich hätte eigentlich erwartet, dass "ISP" darin steht, denn das erscheint schließlich auch als erstes im Text. Aber hier wird schon deutlich, dass das erste (.*) sehr gierig alles genommen hat, was es finden konnte und nur soviel für die anderen Elemente des Regex ließ, wie unbedingt nötig sind. Damit blieb nur der zweite in Hochkommata befindliche Text für Subpattern 2 übrig. Dies hängt im gewählten Beispiel aber auch am letzten ".*", denn dieser Teil ist auch dann erfüllt, wenn "nichts" für diesen Teil bleibt.

Nehmen wir mal ein weiteres Beispiel:

Wir wollen aus einer beliebigen Mailadresse möglichst viele Elemente extrahieren.

Für den Namen hatten wir schon weiter oben eine Lösung, die aber nicht perfekt ist, da sie nur Wortzeichen akzeptierte. Wir werden das jetzt allgemeiner fassen. Wir nehmen (.*) für den ersten Teil. Der zweite Teil wird durch einen Punkt abgetrennt mit einem weiteren Textzusatz. Dieses kann mehrfach vorkommen bevor das @ erscheint, muss aber nicht. Dieses Regex sollte folgende Beispiele erfassen:

'1234abc@mail.de'

'1234.abc@mail.de'

'12-34.abc.def@mail.de'

Das Regex beginnt mit "(.*)\.?(.*)*@" Danach folgt beliebiger Text, möglicherweise getrennt von mehreren Punkten. Uns soll aber von diesem Rest nur das, was

nach dem letzten Punkt kommt, wichtig sein, um das Beispiel nicht unnötig komplizierter zu gestalten. Das sollte man mit "(.*)\.(.*)" erhalten können.

"(.*)\.(.*)*@(.*)\.(.*)"

Was erwarten wir als Ergebnis, wenn man als Zeichenkette '12-34.abc.def@mail.de' eingeben?

Subpattern 1 = '12-34' ?

Subpattern 2 = '.abc' oder '.def' oder 'abc.def' ?

Subpattern 3 = 'mail' ?

Subpattern 4 = 'de' ?

Lasst uns mal den Regextester fragen.

Subpattern 1 = '12-34.abc'

Subpattern 2 = ' def '

Subpattern 3 = 'mail'

Subpattern 4 = 'de'

Das Subpattern 1 hat fast den gesamten Vorspann erhalten, das Subpattern 2 dagegen nur die letzten drei Zeichen vor dem @! Klar, denn das * ist gierig und hat soviel wie möglich im ersten Subpattern "gefressen".

Vorsicht: nicht nur das Sternchen ist gierig, Pluszeichen sind es auch!

Gebt mal '12-34.abc.def@mail.test.de' ein. Auch hier ist das Suchmuster "(.*)" im dritten Subpattern gierig. Es frisst sofort alles nach dem @-Zeichen bis zum letzten Punkt und speichert 'mail.test' im Subpattern und nicht, wie man vielleicht vermuten könnte, nur 'mail'.

Wie lässt sich das vermeiden? Wir lernen wir eine neue Bedeutung des Fragezeichens kennen (immer mit der Ruhe: es ist erst die dritte Bedeutung, es kommen noch mehr. Aber das sollte erkären, warum es soviele Fragezeichen in Regexe gibt *gg*): durch Anfügen des ? an das gierige Suchmuster wird das Regex veranlasst, weniger gierig zu sein.

Konkret für das erste Subpattern:

"(.*?)\.(.*)@(.*)\.(.*)"

Subpattern 1= '12-34'

Subpattern 2= 'abc.def'

Subpattern 3= 'mail.test'

Subpattern 4='de'

Ein kleiner Exkurs um die Vorgehensweise des Regex zu verstehen: nennen wir das, was die Regexe ausführt mal "Regex-Maschine". Die Regex-Maschine lässt der Gier des (.*) normalerweise freien Lauf. In dem Augenblick, in dem es aber auf die Kombination (.*?) trifft, passiert folgendes: es wird soviel wie möglich von der Regex-Maschine in das Subpattern (.*?) gesteckt und nun Zeichen für Zeichen rückwärts

gehend wieder frei gegeben, bis das gesamte Regex insgesamt gerade wieder passt.

Wie sieht das praktisch aus:

"`(.*?)\.(.*)*`" ist der betreffende Part, den wir auf den String '12-34.abc.def' anwenden. Die Regex-Maschine frisst erst mal '12-34.abc' für das erste Muster. Weil danach ein Punkt und noch Text folgt, ist dies das Maximum, denn damit wäre das Regex (ohne ?) erfüllbar. Es stellt aber fest, dass wir mit dem ? die Gier unterdrücken wollen. Also gibt es nun erst das 'c' frei, was aber noch nicht passt, denn nach dem gefundenen Muster folgt nun kein Punkt, sondern das 'c'. Dann das 'b', was auch nicht reicht. Anschließend noch das 'a'. Nun stellt die Maschine fest, dass dies auch zu einem Treffer führt, da sich vor dem 'a' noch ein Punkt befindet und somit das Minimum für einen Treffer erfüllt. In Realität geht die Regex-Maschine noch weiter zurück, um festzustellen, diese Position die letzte war, die mit einem Minimum an Zeichen einen positiven Match erlaubt.

Wie funktioniert das beim ersten Beispiel mit den Hochkommata?

Das Regex war: "`(.*)'(.*)'.*`" und der Text: "Die Abkürzung 'ISP' heißt 'Internet Service Provider'."

Ändern wir es mal in "`(.*?)'(.*?)'.*`" Hier müssen beide geklammerten Suchmuster mit Fragezeichen versehen werden, damit nicht das zweite den Text " ISP' heißt 'Internet Service Provider" findet! Nur das zweite Muster mit Fragezeichen zu versehen, nützt natürlich auch nichts, da schon das erste gefräßig ist!

6.3. Überblick über dieses Kapitel

Dies war ein sehr schwieriger Teil. Nicht nur für das Verständnis, sondern auch zum Schreiben. Allerdings ist es ein grundlegender Bereich der Regexe und wird häufig benötigt.

Wir haben folgende Elemente kennen gelernt:

- Zeichen, mit denen vorangegangene Suchmusterzeichen wiederholt werden können, auch Quantifizierer genannt:

  \+ das Zeichen davor muss mindestens einmal vorkommen, darf aber häufiger

  ? das Zeichen davor darf höchstens einmal oder aber gar nicht vorkommen

  * das Zeichen davor darf beliebig häufig oder auch gar nicht vorkommen.

- Es gibt Quantifizierer, die Bereiche für Häufigkeiten angeben:

  {x,y} das Zeichen davor muss mindestens x-mal, aber höchstens y-mal vorkommen. Man kann dabei Teile des Bereiches weglassen: {x,} heißt beliebig häufig, aber mindestens x-mal. {x} heißt genau x-mal

- Runde Klammern können zur Gruppierung von Suchmustern verwendet werden, um zum Beispiel Quantifizierer auf sie anzuwenden: "(ab)+" heißt, die Buchstabenkombination 'ab' muss mindestens einmal vorkommen.

- Suchmuster in runden Klammern werden "gemerkt" und in Variablen für spätere Verwendung gespeichert. In TB werden diese Variablen Subpattern genannt. Bei geschachtelten Klammern wird das gesamte Ergebnis eines vom Klammerpaar umgebenen Suchmusters in eine Variable gespeichert; die Ergebnisse der inneren Klammern sind dann Teilmengen der jeweils äußeren. Die erste öffnende Klammer erzeugt die erste Variable, die zweite erzeugt die zweite Variable usw...
- Quantifizierer ohne obere Grenze sind in bestimmten Suchmustern gierig. + und * hinter einem Punkt veranlassen das Regex, alles zu finden, was möglich ist, damit das Suchmuster gerade soeben noch passt. Das Regex (.*)(.*) wird jede Zeichenkette komplett in das erste Subpattern legen.
- Durch Anfügen eines Fragezeichens an gierige Suchmuster (.*?) wird das Regex veranlasst, nicht gierig zu sein. Zwar wird zunächst dennoch alles "gefressen", aber dann Zeichen für Zeichen zurückgegeben, bis das Regex gerade noch passt.

Aufgaben

1: Das Regex für die Datumsangabe ist immer noch nicht perfekt, weil ja einstellige Tages- oder Monatszahlen und ggf. zweistellige Jahreszahlen nicht gefunden werden. Wäre doch mal 'ne Übung wert, oder?

2: So, die Lösung für das obige Problem ist ganz interessant, aber nun wollen wir ein Regex bauen, dass etwas besser nach dem Datum sucht. Es soll nach Möglichkeit nur etwas gefunden werden, was nach einem Datum aussieht. Es muss ja nicht übertrieben werden: wenn es laut Regex auch den 29.2. in einem Nicht-Schaltjahr gibt, ist das nicht so schlimm. Wichtig ist: TT.MM.JJJJ oder T.M.JJ sollen gefunden werden.

3: Über ein Online-Fehlermeldesystem kommen standardisierte Meldungen, die mit einem Regex in seine Bestandteile zerlegt werden soll. Die Meldungen haben die Form:

Absender: vorname.nachname@amt.de

Datum: TT.MM.JJJJ

Fehlernr.: xyz123

Bitte erstellt ein Regex, dass die einzelnen Felder (Vorname, Name, Amt, Datum, Fehlernummer) als Subpattern findet.

4. Erstellt ein Regex, das Uhrzeiten in der Form hh:mm:ss findet. Dabei sollen nach Möglichkeit nur gültige Kombinationen gefunden werden.

Zu 1:

```
"\d{1,2}\.\d{1,2}\.(\d{4}|\d{2})"
```

Ihr habt etwas anderes? Macht nichts, muss nicht falsch sein. Es gibt immer mehrere Wege, es richtig zu machen:

```
"(\d?\d\.){2}(\d{4}|\d{2})"
```

wäre die elegante Lösung. Was ich weniger gelungen fände, wäre "\d{2,4}" für die Jahreszahl, da dann auch dreistellige Jahreszahlen akzeptiert würden.

Zu 2:

Dies ist sicherlich der kompliziertere Fall. Zerlegen wir das mal in Teilprobleme. Die möglichen Tage sind:

a) 01-09, wobei die führende Null fehlen könnte.

b) 10-29, alle Monate haben 29 Tage. Ok, gewollter Fehler: der Februar hat eben nur alle 4 Jahre 29 Tage, aber damit wollen wir jetzt mal leben, sonst wird es nahezu unmöglich.

c) 30, haben alle Monate außer Februar

d) 31, haben nur die Monate Januar, März, Mai, Juli, August, Oktober, Dezember

Die möglichen Monate sind 01-10, wobei die Null fehlen darf und 11, 12.

Wir wollen nur beliebige zweistellige oder vierstellige Jahreszahlen zulassen. Im letzteren Fall sollte die Jahresangabe aber schon mit 19xx oder 20xx beginnen.

Na, dann wollen wir mal:

Fall a) und b) mit den Monaten

```
"(0?[1-9]|[12][0-9])\.(0?[1-9]|1[0-2])\."
```

Fall c) mit den Monaten

```
"30\.((0?[13-9])|(1[0-2]))\."
```

und dann noch Fall d) mit seinen Monaten

```
"31\.(0?[13578]|1[02])\."
```

Zu den Jahren:

```
"(\d{2}|(19|20)\d{2})"
```

Die ersten drei Elemente sind Alternativen, die Jahreszahl ist obligatorisch. Damit nicht aus längeren Ziffernfolgen zufällige Datumsangaben gefunden werden, umgeben wir das Regex noch mit dem \b-Operator. Dann muss das also ungefähr so aussehen:

```
"\b(((0?[1-9]|[12][0-9])\.(0?[1-9]|1[0-2])\.)|(30\.((0?[13-9])|(1[0-
2]))\.)|(31\.(0?[13578]|1[02])\.))(\d{2}|(19|20)\d{2})\b"
```

[Anm.: Das Regex wird aus Layout-Gründen umgebrochen. Tatsächlich muss sich alles in einer Zeile befinden!]

Wahnsinn: ganz schön lang. Ihr habt was anderes? Sogar was besseres? Das würde ich als "normal" bezeichnen. Das obige Regex lässt sich natürlich bei gleichem Ergebnis anders schreiben. Und "besser" geht fast immer ;-) Meine Lösung oben zeigt nur, wie ich das Problem angegangen habe. Ich hoffe, dies ist nachvollziehbar gewesen.

Zu 3.

Das sieht gar nicht so wild aus. Auch hier wieder in Teilen:

Vorname und Name sowie Amt ergeben sich aus der Mailadresse. Damit sollte

"Absender:\s*(.*?)\.(.*?)@(.*?)\.\w+\s*" ausreichen. Das ? im zweiten Subpattern ist vielleicht gar nicht nötig, weil danach ohnehin das @-Zeichen folgen muss. Aber es schadet auch nix.

Datum: welch Glück, das Format ist fest. Man muss also nicht das mörderische Regex aus Aufgabe 2 verwenden:

"Datum:\s*((\d{1,2}\.){2}\d{4})\s*"

Und nun noch Fehlernummer:

"Fehlernr.:\s*(.*)"

Um sicher zu gehen, dass auch der ganze String abgefragt wird, veranlassen wir mit \A und \Z, dass der ganze Text ausgewertet wird.

```
"\AAbsender:\s*(.*?)\.(.*?)@(.*?)\.\w+\s*Datum:\s*((\d{1,2}\.){2}\d{4})\s*
Fehlernr.:\s*(.*)\Z"
```

[Anm.: Das Regex wird aus Layout-Gründen umgebrochen. Tatsächlich muss sich alles in einer Zeile befinden!]

Die Subpattern 1,2,3,4 und 6 sollten die gewünschten Felder liefern.

Zu 4.

Nun haben wir ja schon gewisse Übung darin, Probleme in Teilprobleme zu zerlegen und das Uhrzeit-Problem ist erneut so eines. Es sollte uns also eigentlich leicht von der Hand gehen. Es macht es geringfügig einfacher, dass das Format feststeht!

Stunden gibt es von 00-19 Uhr und von 20-23 (24 Uhr ist 00 Uhr!):

"([01][0-9]|2[0-3]):"

Minuten und Sekunden verwenden die gleichen Ziffernkombinationen, nämlich 00 bis 59:

"([0-5][0-9]:){2}"

Zusammen, umgeben von Wortbegrenzern:

```
"\b([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]\b"
```

7. Spezielles - Teil 2

Hattet ihr die Befürchtung, dass das Kapitel 7 etwas schwieriger als das sechste wird? Ich will euch beruhigen. Nein, es behandelt ein paar wenige Punkte, die etwas komplexer sind, aber im Ganzen sind viele Elemente recht einfach und zudem nur sehr selten zu verwenden. Wer die vorangegangenen Kapitel verstanden hat, kann schon recht effektive und mächtige Regexe bauen.

Einige Begriffe sind in englisch: leider fehlen mir vernünftige deutsche Übersetzungen. Ich hoffe, dass die Darstellung aber dennoch verständlich geblieben ist.

7.1. Assertion

Was soll eine Assertion sein? Die Übersetzung aus dem Englischen ergibt so etwas wie "Behauptung" oder "Aussage". In der Computersprache wird es mit "Aussage"

noch am besten übersetzt. Sucht man dann diesen Begriff im Friedl, hat man Pech: Fehlanzeige! Dort wird diese Art der Regexe mit "Lookahead" bezeichnet. Na ja, wollen wir uns mal anschauen, was sich dahinter verbirgt.

Mit einer Assertion kann getestet werden, ob eine bestimmte Zeichenfolge der gefundenen Stelle vorangeht oder folgt. Na, das ist ja wenig erquicklich, denn das konnten wir auch schon zuvor. Aber: mit der Assertion wird geprüft, ohne dass die geprüfte Zeichenkette in den gefundenen String übernommen wird; sie "frisst" die Zeichen nicht.

Am besten sieht man sich dazu ein Beispiel an:

Wir wollen nur die Buchstabenfolge 'foo' finden, die von der Zeichenfolge 'bar' gefolgt wird, ohne aber das 'bar' in unserem Suchergebnis wiederzufinden. Normalerweise hätten wir einfach "`foobar`" als Regex angegeben. Dann aber findet das Regex 'foobar'.

Setzt man dagegen eine Assertion "(?=" ein, so sieht das Regex wie folgt aus:

"`foo(?=bar)`". Das Ergebnis lautet nun 'foo'.

Natürlich kann man negative Lookaheads oder Assertions definieren. Dazu verwendet man "(?!".

Wieder auf unser wenig sinnreiches Beispiel angewendet, ergibt dies bei dem Regex "`foo(?!bar)`", dass nur 'foo' gefunden wird, wenn kein 'bar' folgt. Außerdem wird im gefundenen String nur das 'foo' abgelegt. Wird also das Regex auf den Text 'foolish' verwendet, dann lautet das Ergebnis 'foo'.

Aber Achtung, da gibt es eine Falle. Man könnte bei dem Regex "`(?!foo)bar`" meinen, es würde die Zeichenkette 'bar' nur dann finden, wenn dem nicht 'foo' vorangeht. Falsch, sie findet jedes Vorkommen der Buchstaben 'bar', egal was davor steht! Denn, da diese Assertion eine Lookahead, also nach vorn schauende ist, sucht sie ab der Fundstelle des 'bar' nach vorn, ob dort 'foo' steht. Nun, das steht dort nie, denn da steht 'bar'. Logisch, oder?

Was wir brauchen, ist eine Lookbehind-Assertion! Klar gibt es das: "(?<=" für die positive Variante und "(?<!" für die negative Version.

Beispiel:

"`(?<!foo)bar`" findet das 'bar' nun nur dann, wenn kein 'foo' davor steht.

"`(?<=foo)bar`" dagegen findet 'bar' nur, wenn 'foo' davor erscheint.

Selbstverständlich kann man auch Alternativen in den Assertions verwenden. Als Beispiel: "`(?<=Einig|Möglich)keit`" findet die Zeichenkette 'keit', wenn entweder 'Einig' oder 'Möglich' davor steht. ('keit' wird auch gefunden, wenn dort UnMöglich' vor steht. Ich nehme an, dass dies aber allen klar ist, oder?)

Eine Bedingung ist an den Suchbegriffen im Assertion allerdings zu stellen: ihre Länge muss definiert sein, das heißt, es können keine Quantifizierer verwendet werden! Die Assertion "`(?<=\d+,)\d\d`" führt zu einem Laufzeitfehler.

Bei Alternativen, die man im Assertion prüfen kann, dürfen die einzelnen alternativen Elemente sehr wohl unterschiedlich sein, aber sie müssen eine bekannte Länge haben.

Die definierte Länge bezieht sich auf die oberste Ebene der Prüfung im Assertion. Ja, ich weiß: "Was ist die oberste Ebene?" werdet ihr fragen. Ich versuche es erst gar nicht, ich zeige gleich ein Beispiel:

"`(?<=ab(c|de))`" ist verboten, "`(?<=abc|abde)`" dagegen nicht. Durch Öffnen einer weiteren Klammerebene im ersten Assertion verlasse ich die oberste Ebene; damit wird die Assertion für die Regex-Maschine unberechenbar.

Schließlich sollte zumindest bekannt sein, dass man mehrere Assertions hintereinander verwenden darf, aber auch ineinander verschachteln kann.

Beispiel dafür: "`(?<=\d{2},\d{2})(?!00,00)\s+Ausgabe`" findet ' Ausgabe' nur dann, wenn davor jede beliebige Summe steht, die nicht auf '00,00' lautet.

Oder: "`(?<=(?<!un)möglich)keit`" findet das Wort 'keit' nur, wenn davor 'möglich' steht, aber nicht 'unmöglich'.

7.2. Backreference

Wieder mal ein englisches Wort. "Rückbezug" ist ein passender Begriff dafür, denn was wir im Regex damit machen wollen, ist ein solcher Rückbezug. Aber beginnen wir erst einmal vorn!

In einer der letzten Abschnitte hatten wir etwas von Subpatterns gelesen. Das waren gefundene Zeichenketten, die vom Regex in Variablen gespeichert werden. Ich schrieb da "... werden in eine temporäre Variable (Subpattern, SubPatt in TB genannt) für spätere Zwecke zwischengespeichert...". Nun, einen dieser Zwecke haben wir jetzt vor uns.

Betrachten wir einfach gleich das Regex: "`(Rede|Kehrt) und \1wendung)`" Das Regex soll entweder 'Rede' oder 'Kehrt' finden. Dieser Wert wird in das erste Subpattern (wir erinnern uns: die erste öffnende Klammer) gelegt. Danach soll das Wort 'und' folgen. Nun steht im Regex "`\1`". Dies heißt nichts anderes als: verwende den Inhalt des ersten Subpatterns zum Suchen! Da es von 'wendung ' gefolgt wird, soll also entweder 'Redewendung' oder 'Kehrtwendung' an dieser Stelle gefunden werden, je nachdem, welches der beiden Wörter weiter vorn gefunden wurden. Das Regex findet also nur 'Rede und Redewendung' oder 'Kehrt und Kehrtwendung', aber niemals 'Rede und Kehrtwendung'.

Der Rückbezug darf nicht in der Klammer, dem Subpattern stehen, auf den es sich bezieht. Dies führt zu keinem Ergebnis: "`(a\1)`" führt nie zu einem positiven Ergebnis. Allerdings kann diese wieder verwendet werden, wenn sich ein Quantifizierer dahinter befindet: "`(da|de\1)+`" Das findet 'dadadada' oder 'dadeda' oder auch 'dadedadadada'.

7.3. Konditionale Reguläre Ausdrücke

Selten genutzt, aber dennoch interessant sind konditionale Regexe. Sie folgen dem Prinzip "Wenn Muster A gefunden, dann suche nach Muster B; wenn nicht, dann nach Muster C".

Die Syntax ist eigentlich recht einfach: "(?(Bedingung)Ja-Muster|Nein-Muster)"

oder "(?(Bedingung)Ja-Muster)"

Eine Bedingung gilt es aber zu erfüllen: wenn die Bedingung nicht eine Folge von Ziffern ist, dann muss es sich um eine Assertion handeln.

In welchen Situationen lässt sich denn ein solches konditionale s Regex verwenden? Nehmen wir mal an, wir brauchen aus einem Text Tag, Monat und Jahr, wissen aber aus unerfindlichen Gründen nicht, ob das Datum im deutschen TT.MM.JJJJ oder englischen TT, MMM JJJJ Format geschrieben ist. Was wir wissen ist, dass am Zeilenanfang 'Date' oder 'Datum' steht und die Information mit der Zeile endet.

Nun müssen wir also nur das Regex veranlassen, das englische Muster zu suchen, wenn davor 'Date' steht, ansonsten soll es das deutsche Muster suchen.

```
"(?(?=^Date)Date:\s(\d+),\s([A-Za-z]{3})\s(\d{4})$|
Datum:\s(\d{2}\.)(\d{2}\.)(\d{4})$)"
```

[Anm.: Das Regex wird aus Layout-Gründen umgebrochen. Tatsächlich muss sich alles in einer Zeile befinden!]

Ich hätte die Suche nach dem Doppelpunkt und dem folgenden Leerzeichen auch in die Assertion nehmen können, war mir aber nicht sicher, ob das nicht ein Problem bei der Speicherung der Teilinformationen in die Subpattern hervorruft.

Betrachten wir noch ein weiteres Beispiel aus der Zeit, als TB noch kein %IF kannte. Wollte man im Schreiben eine Anrede in der Form 'Sehr geehrte Frau' oder 'Sehr geehrter Herr' benutzen, so konnte man im Grunde keine einheitliches Template benutzen: 'Frau' oder 'Herr' gab es aus einem Feld in dem Adressbuch, aber das geschlechtsspezifische 'r' am Ende von 'geehrte' musste man über einen Würg-Around erarbeiten. Allerdings konnte einem ein Regex dabei helfen. Man fragte einfach ab, ob 'Herr' davor steht und wenn ja, dann sollte das Regex den letzten Buchstaben in ein Subpattern speichern. War dagegen kein 'Herr' in dem Feld, dann sollte das Regex auch nichts suchen.

```
"(?(?=Herr)Her(r))"
```

Keine Alternative für das Nein-Muster und auch keine komplizierten Elemente im Suchmuster. Eigentlich ganz einfach. Wie man dann solche Regexe auf Felder von TB anwendet und wieder in den Text integriert, ist Thema des 7. Abschnitts. Da haben wir noch etwas Zeit.

Eingangs hatte ich schon angedeutet, dass als Bedingung eine Assertion stehen muss oder eine Ziffer. Die Ziffer ist aber in diesem Fall kein Zeichen, das literal gesucht wird. Vielmehr muss es ein Rückbezug sein, ein Backreference auf ein zuvor gefundenes Muster, das in einem Subpattern abgelegt wurde. Wie haben wir das zu verstehen?

Beispiel: in einem Text erscheint in der ersten Zeile ein Name nach dem Eintrag 'Name'. Der Name kann je nach Text variieren. Im weiteren Verlauf des Textes erscheint der Name erneut, aber mit einem Attribut, das wir benötigen, sagen wir einfach mal die Schuhgröße.

"`Name:\s*(.*)?$`" sollte den Namen finden. Danach kommt etwas, was uns nicht stört, aber irgendwo da drin gibt es dann wieder den Namen, gefolgt von einem Doppelpunkt und folgend die Schuhgröße:

```
".*?(?(1):\s*(\d+))"
```

Insgesamt also:

"Name:\s*(.*)?$.*?(?(1):\s*(\d+))"

Versucht es mal mit der folgenden Zeichenkette:

'Name: Lieschen Mueller

Bladibla

Lieschen Mueller: 43'

(Anmerkung: im Regex-Tester müssen die Optionen m und s eingeschaltet sein)

7.4. Optionen, Modifikatoren

Nachdem wir nun eine Vielzahl von "Vokabeln" für das Regexische lernen mussten, kommen wir zu den Modifikatoren. Wie der Name vermuten lässt, modifizieren sie was. Und was? Nun, die Regex-Zeichen werden bei Gebrauch der Modifikatoren "modifiziert". Ich höre euch aufstöhnen: kaum hat man ein paar von den vielen Zeichen gelernt und kann wenigstens rudimentäre Regexe bauen, da wird deren Bedeutung durcheinander gewirbelt. Aber keine Sorge, so schlimm wird es nicht. Wir werden uns auf ein paar wenige beschränken. Mittlerweile sind aber viele hinzugekommen, für die es in TB keine Anwendung gibt. Wer nur TB verwendet, kann sich im Grunde auf die ersten vier beschränken.

Betrachten wir also folgende Auswahl:

i für Caseless

Damit wird die Regex-Maschine veranlasst, Groß- und Kleinschreibung zu ignorieren. Egal, wie es im Regex steht, es wird in jeglicher Schreibweise gefunden.

m für Multiline

Standardmäßig betrachtet die Regex-Maschine die zu untersuchende Zeichenfolge als eine einzige Zeile, gleichgültig, ob im Text Newline-Character stehen (\n). Das ^-Zeichen passt wirklich nur am Anfang und das $-Zeichen am Ende der Zeichenfolge oder aber an einem abschließenden Newline-Character. Probiert es mal im Regex-Tester aus: schaltet die Option Multiline aus. Dann gebt folgenden Text mit Zeilenumbrüchen ein:

'Dies ist ein Test, der

als Test

am Ende steht'

und als Regex "Test$". Es wird nichts gefunden. Schaltet die Option wieder ein und das Wort 'Test' wird gefunden.

Mit eingeschalteter Option werden tatsächlich alle Newline-Character beachtet; die Zeichenfolge wird aus mehreren Zeilen bestehend angesehen. Wichtig, wenn wir mal einen ganzen Mailtext mit einem Regex prüfen.

s für DotAll

Der Punkt steht, wie wir zuvor gesehen haben für alle beliebigen Zeichen, außer dem Zeilenumbruch, also dem Newline. Wird die Option eingeschaltet, so findet der

Punkt auch dieses Zeichen. Allerdings ist das auch nicht die ganze Wahrheit: das Newline wird auch von so genannten negierten Zeichenklassen gefunden. Lautet also ein Regex "[^x]", so werden alle Zeichen, auch Newlines gefunden, die kein 'x' sind!

x für Extended

Wenn diese Option gesetzt ist, werden alle Whitespace-Zeichen ignoriert, es sei denn, sie werden maskiert. Dies geschieht, wie wir im ersten Teil gelernt haben mit einem vorangestellten Backslash "\ ". Oder aber man sucht das Whitespace mit "\s".

Weiterhin werden Whitespaces oder Leerzeichen auch dann noch gefunden, wenn sie innerhalb einer Charakterklasse definiert werden: "[ ]" Diese Option ermöglicht das Einfügen von Kommentaren in das Regex, indem die Kommentare hinter nicht maskierte "#"-Zeichen gesetzt werden.

A für Anchored

Wenn dieser Modifikator gesetzt wird, wird das Regex gezwungen, ab dem Anfang der Zeichenkette zu suchen. Dies lässt sich natürlich auch durch das korrekte Verwenden des Metazeichens "\A" erzielen. In Perl gibt es den Modifizierer A nicht, daher ist das Metazeichen "\A" dort die einzige Möglichkeit.

D für Dollar_Endonly

Mit diesem Modifizierer findet das Metazeichen "$" nur noch das Ende der Zeichenkette, unabhängig davon, ob zuvor noch weitere Zeilenumbrüche ("\n") existieren. Ohne diesen Modifizierer matcht "$" unmittelbar vor dem letzten Zeichen, wenn dieses ein Newline ("\n") ist. Er wird allerdings ignoriert, wenn zusätzlich der Modifizierer "m" eingeschaltet wird. In Perl gibt es den Modizierer nicht.

U für Ungreedy

Hiermit wird die Gierigkeit von Quantifizierern umgekehrt: wenn er gesetzt ist, sind alle Quantifizierer im Regex standardmäßig nicht gierig. Sie werden nur durch ein nachfolgendes "?" gierig. Auch hierfür gibt es bei Perl kein Äquivalent.

u für UTF8

Der Modifizierer ist ebenfalls inkompatibel zu Perl: ist er eingeschaltet, dann wird der Suchstring als UTF-8 behandelt.

X für Extra

Noch ein Modifizierer, der in Perl keine Verwendung findet. Jedes Zeichen, das keine besondere Metazeichenbedeutung hat, aber mit einem Backslash maskiert ist, führt zu einem Fehler. Normalerweise wird jedes Nicht-Metazeichen, das einen vorhergehenden Backslash hat, literal gesucht. Ich habe es sogar im Text als unschädlich bezeichnet, ein Backslash zuviel zu setzen, da das Zeichen dennoch gesucht wird. Wird X als Modifizierer gesetzt, führt dies zu einem Fehler.

e für Evaluate

Eigentlich habe ich erfunden, dass dieser Modifizierer für "Evaluate" steht; es gibt kein PCRE-Synonym dafür. Die Modifizierer steht, soweit ich weiß, nur den PHP-Usern zur Verfügung. Er kann nur in der Funktion preg_replace verwendet werden.

Ist er eingeschaltet, so werden Substitutionen und Backreferences ganz normal durchgeführt, allerdings dann als PHP-Code interpretiert und das Ergebnis dann zum Ersetzen verwendet. Schwierig zu verstehen? Ok, hier ein Beispiel: Das Regex soll "`(foo)(.*?)(bar)`". Im Subpattern 2 sind also nicht bekannte Zeichen. Diese wollen wir in Großbuchstaben wandeln und verwenden dazu als Beispielzeichenkette "foosibar":

```
<?php
$ergebnis=preg_replace("/(foo)(.*?)(bar)/e",
"'\\1'.strtoupper('\\2').'\\3'", "foosibar");
echo $ergebnis;
?>
```

Das Ergebnis lautet: "fooSIbar" (Ja, ich weiß schon: kein sehr sinnvolles Beispiel *g* Ich hoffe, man versteht trotzdem, was "e" macht).

S für Speed

Schon wieder eine Erfindung von mir: es gibt kein PCRE_SPEED. Und wieder freuen sich nur die PHP-User, soweit ich das überblicke. Wenn ein Suchpattern mehrfach verwendet werden soll, so macht es für die Regex-Maschine Sinn, es etwas länger zu analysieren und so die Match-Geschwindigkeit (Speed) zu erhöhen. Dieser Modifizierer veranlasst die Extra-Analyse. Sinn macht das natürlich nur, wenn im Suchpattern nicht schon Verankerungen wie Zeilen- oder Zeichenkettenanfang eingetragen sind.

Wer in den Regex-Tester schaut und dort Optionen aufruft, wird noch ein paar weitere Optionen bzw. Modifikatoren finden. Ich werde sie hier nicht weiter erläutern. Es handelt sich um spezielle Einstellungen, deren Auswirkungen auch in geeigneter Literatur nachlesbar ist und zum Grundverständnis von Regexen nur bedingt beitragen. Die vier ersten werden uns in aller Regel reichen.

Wie aber schaltet man diese Optionen ein? Nichts einfacher als das: man trägt den Buchstaben zwischen "(?" und ")" ein. Also: "`(?i)`" schaltet die "Caseless" ein. Man kann die Optionen auch kombinieren: "`(?im)`" heißt 'Caseless, Multiline'. Und noch mehr: man kann die Optionen ein- und ausschalten. "(?im-sx)" schaltet Caseless und Multiline ein, aber DotAll und Extended aus. Erscheint ein Buchstabe sowohl vor dem "-"-Zeichen als auch dahinter, dann wird die Option ausgeschaltet.

Es ist im Grunde auch egal, wo die Option geschaltet wird: es kann am Anfang geschehen, aber auch mitten im Regex.

"`(?i)Test`" ist gleichbedeutend mit "`Te(?i)st`". Wird die gleiche Option mehrfach auf der obersten Suchebene gesetzt, so gewinnt die am weitesten rechts stehende Einstellung. Aber es ist so nicht sehr übersichtlich und daher soll man diese Optionen an den Anfang des Regex stellen.

Ich schrieb "...im Grunde...". Hmmm, das impliziert ja schon wieder eine Ausnahme! Und so ist es auch: erscheint eine Option innerhalb eines Subpatterns, so gilt es nur für das Subpattern. "`(a(?i)b)c`" findet 'abc' aber auch 'aBc'.

Mal was zum Rätseln:

"`(a(?i)b|c)`" sei das Regex. Findet das nun ein 'C' oder nur 'c'? Unstrittig ist sicherlich, dass 'ab' oder 'aB' gefunden werden.

7.5. Besonderes

Kommen wir zu ein paar Besonderheiten, die uns nur selten begegnen werden, wenn wir Regexe in TB verwenden. Ich werde auch nicht im Detail auf alles eingehen; dieses Kapitel soll mehr als Erinnerungshilfe dienen, damit man weiß, wo man nachschauen muss, wenn sich mal ein Regex eigentümlich verhält. ;-)

Meta-Zeichen

Gleich im ersten Kapitel erwähnte ich die Metazeichen. Dabei zählte ich auch die Zeichen] und } auf. Tatsächlich sind dies gar keine Metazeichen. Wird nach ihnen literal gesucht, müsste man sie nicht maskieren. Ich dagegen mache dies aber, um Überblick über mein Regex zu haben. Ich vermeide dabei auch ein Risiko für einen Fehler:

Innerhalb einer Charakterklasse, deren Definition mit "[" beginnt, gibt es eine abweichende Definition für Metazeichen. Nur die nachfolgenden Zeichen sind dort Metazeichen:

\ zum Maskieren

^ zum Negieren der Charakterklasse, aber nur, wenn es das erste Zeichen ist

- zum Kennzeichnen eines Bereichs

] zum Beenden der Charakterklasse

Nun betrachten wir mal folgendes, wenig geistreiches Regex:

"`[Y-]345]`" Dieses Regex soll nach meinem Wunsch eigentlich eine Charakterklasse definieren, bestehend aus allen Zeichen von Y bis zum Zeichen] sowie den Ziffern 3,4 und 5. Was kommt aber heraus? Findet das Regex den String 'Z34' oder Teile davon? Nein! Stattdessen gebt mal 'Y345]' oder '-345]' als zu untersuchenden Text ein. Und siehe da, es wird was gefunden. Aber irgendwie was anderes, als wir eigentlich von dem Regex wollten.

Na gut, versuchen wir das mal zu erklären: die erste schließende eckige Klammer wird unmittelbar als Abschluss einer Charakterklasse verstanden. Das Regex findet also Zeichenketten, die mit 'Y' oder '-' beginnen und danach die Zeichen '345]' haben. Aber, nichts einfacher als das. Der Kursleiter hat ja gesagt, es ist unschädlich, das ']' zu maskieren, damit es literal gesucht wird. Und? Ausprobiert?

Jawohl, "`[Y-\]345]`" ist genau die Lösung.

Eckige Klammern

Bleiben wir mal bei den eckigen Klammern und schauen uns ein paar Sonderfälle an. Nehmen wir an, wir haben die Option Caseless gewählt, dann wird mit "[aeiou]" ein 'A' und auch ein 'a' gefunden. Das Regex "`[^aeiou]`" findet dagegen ein 'A' nur, wenn Caseless nicht eingeschaltet ist.

Ziffern und Zahlen

Außer Dezimalzahlen, die wir bisher einfach mit "\d" gesucht haben, kann man selbstverständlich auch hexadezimale oder oktale Zahlen finden. Ein Regex wie "`\x09`" findet das Zeichen mit dem hexadezimalen Code 09.

Bei den oktalen wird es spannend: die Syntax ist recht einfach "\ddd", wobei d für eine Ziffer steht, sucht nach einem Zeichen mit dem oktalen Wert 'ddd'. Oder, und nun wird es unübersichtlich, nach einem Backreference, einem Rückbezug. Und zwar interpretiert die Regex-Maschine jede Ziffer, die kleiner als 10 ist sofort als Rückbezug, wenn es sich außerhalb einer Charakterklasse befindet. Innerhalb von Charakterklassen oder wenn es nicht genügend öffnende Klammern gibt, wird der Teil als oktales Suchmuster verstanden.

Ein paar Beispiele

\040 ist oktal (Leerzeichen)

\40 ist auch oktal, es sei denn, es gibt genügend öffnende Klammern und somit Subpattern

\6 ist immer Rückbezug

\11 kann Rückbezug sein, ansonsten 'Tabulator'

\011 ist immer 'Tabulator'

\113 ist immer oktal, weil es nie mehr als 99 Rückbezüge geben darf

Was heißt eigentlich "\0113"?

Einschränkungen

Nur zur Information. Im Regelfall werden wir einfache User das gar nicht bemerken, aber ein Regex darf maximal 65535 Bytes lang sein. Alle Werte, die mit einem Quantifizierer gefunden werden, dürfen 65535 Bytes nicht überschreiten. Es darf nicht mehr als 99 Subpatterns geben. Das Maximum aller geklammerten Ausdrücke, also Subpatterns, Optionen, Assertions, konditionalen Mustern darf 200 nicht überschreiten. Tja, und die Textgröße ist ebenso eingeschränkt, aber im Regelfall irrelevant beim Gebrauch mit TB. Sie darf nur so groß werden wie die größte positive Integervariable sein darf. Da aber für Subpatterns und Quantifizierer mit undefinierter Größe rekursiv von der Regex-Maschine vorgegangen wird, könnte der Speicherplatz geringer ausfallen. Aber, das interessiert uns bei TB nun wirklich nicht!

7.6. Überblick über diesen Abschnitt

Dieser Abschnitt hat uns ein Besonderheiten der Regexe nähergebracht. Wir werden damit genügend über Regexe wissen, um im nächsten Abschnitt die TB-spezifischen Makros einsetzen zu können.

Fassen wir mal dieses Kapitel zusammen:

- es gibt so genannte Assertion, mit denen geprüft werden kann, ob eine bestimmte Zeichenfolge vor oder hinter unserem Suchmuster erscheint, ohne dass diese Zeichenfolge selbst als Ergebnis ausgegeben wird. Man unterscheidet

  positive Lookahead-Assertion (?=

  negative Lookahead-Assertion (?!

  positive Lookbehind-Assterion (?<=

negative Lookbehind-Assertion (?<!

Sie dürfen nicht durch Quantifizierer in ihrer Länge unberechenbar werden.

- Zeichenfolgen, die durch Muster in runden Klammern als Subpattern gefunden wurden, können über so genannte Rückbezüge erneut im Regex aufgerufen werden. Rückbezüge oder auch Backreferences werden durch "\d" definiert.
- Mit Assertions oder Rückbezügen lassen sich konditionale Regexe erstellen. Die Syntax lautet "(?(Bedingungsmuster)Ja-Suchmuster|Nein-Suchmuster)"
- Das Regex-Vokabular kann durch Optionen oder Modifikatoren in seiner Interpretation verändert werden. Sie können dem Regex vorangestellt werden: "(?Modifikator)". Wir haben hier nur einen Ausschnitt der möglichen kennen gelernt:

  i für Caseless

  m für Multiline

  s für DotAll

  x für Extended

  A für Anchored

  D für Dollar_Endonly

  U für Ungreedy

  u für UTF8

  X für Extra

  e für Evaluate

  S für Speed

(Anm.: einige dieser Modifikatoren sind nicht im Sprachumfang von TB enthalten. Sie können darüber hinaus ggf. auch in Perl nicht verwendet werden. Zum Teil können sie auch nicht mit (?Modifikator) geschaltet werden. Bitte beachtet die Ausführungen in der Beschreibung der Modifikatoren.)

- Wir lernten ein paar Besonderheiten kennen, die sich auf bestimmte Zeichen beziehen: innerhalb von Charakterklassen können bestimmte Zeichen zu Metazeichen mit anderer Bedeutung werden.
- Mit einem Regex lassen sich auch hexadezimale oder oktale Werte der Zeichen suchen. Dabei kann es zu Kollisionen mit der Bezifferung von Rückbezügen kommen.
- Regexe dürfen nicht beliebig groß werden, auch ihr Ergebnis ist beschränkt. Sogar die zu untersuchende Textlänge ist nach oben eingeschränkt. Aber keine dieser Grenzen wird für TB-Anwendungen relevant sein.

Nun folgen wieder die obligatorischen **Aufgaben:**

1. Erstellt doch bitte ein Regex, das verdoppelte Wörter erkennt (im Stile von 'der der'). Dabei sollte beachtet werden, dass die Wörter nach einem Zeilenumbruch aufeinander folgen dürfen, sie groß oder klein geschrieben sein können und nur als ganze Wörter gefunden werden sollen (also nicht 'Das Dasein').

2. Versuchen wir uns mal an einer ganz einfachen Version eines Subject-Bereinigungsregexes. Das 'Re' kann von allem möglichen und einem Doppelpunkt gefolgt werden. Dann kommt der eigentliche Betreff und nach einem Leerzeichen könnte eine Klammer folgen, mit 'was:' oder auch 'war:' würde darin der ehemalige Betreff eingeleitet. Wir wollen aber nur den eigentlichen Betreff! Aber Achtung: es handelt sich wirklich nur um eine ganz stark abgespeckte Version. Eine bessere Version wird im folgenden Kapitel folgen; dort werden wir näher auf die verschiedenen Möglichkeiten eingehen.

3. Wir erhalten Mails mit Bestellsummen. Aus diesen wollen wir mit einem Regex nur den ganzzahligen Wert auslesen (also alle Ziffern vor dem Dezimaltrennzeichen). Dummerweise kommen die Bestellsummen entweder in $ oder in EUR. Na, das wär ja halb so wild, wenn nicht die $-Beträge mit typisch amerikanischen Dezimal- und Tausendertrennzeichen geliefert würden: #,###.##$ und die EUR-Beträge eben halt im deutschen Format #.###,##EUR geschrieben würden. In jedem Fall soll das Regex entscheiden, mit welchem Muster es welche Summe liest.

Zu 1:

Wichtig war hier der Hinweis auf den Zeilenumbruch und die Groß-/Kleinschreibung. Dies sind Optionen, die wir im Regex erst einmal einschalten sollten: "`(?im)`". Worte zu finden, sollte uns leicht von der Hand gehen: "`[a-z]+`" Wir betrachten wirklich nur Worte ohne Ziffern. Da wir die Option "i" eingeschaltet haben, müssen wir auch in der Zeichenklasse keine Großbuchstaben mehr definieren. Allerdings wollen wir ganze Wörter betrachten: also sollte vor dem Wort eine Wortgrenze sein. Danach lassen wir einfach ein Leerzeichen (oder viele) folgen. Eine Wortgrenze am Ende zu fragen, könnte uns in die Irre leiten. Schließlich könnte ein Satz mit einem Wort enden und mit genau dem gleichen könnte der Folgesatz beginnen. Das liefert uns also bis jetzt: "`(?im)\b[a-z]+\s+`" Das ist natürlich noch falsch, denn das gefunden Wort soll ja noch mal gefunden werden: wir müssen also den Wert in ein Subpattern geben, um es mit einem Rückbezug aufrufen zu können. Also noch mal:

"`(?im)(\b[a-z]+)\s+\1`"

Leider liefert uns das nun genau den Fall 'das Dasein', den wir gar nicht wollten. Nichts leichter als das: auf das verdoppelte Wort darf halt nichts mehr als eine Wortgrenze folgen und damit hätten wir es:

```
"(?im)(\b[a-z]+)\s+\1\b"
```

Zu 2: Noch mal: es ist nur eine sehr vereinfachte Variante eines Regexes, dass wir im Folgekapitel noch etwas professioneller angehen wollen. Als Übung sollte es aber gereicht haben.

Das Betreff hätte also laut Vorgaben so aussehen können: 'Re[2]: richtiger Text, der gefunden werden soll ;-) (was: irgendetwas Unwichtiges)'

Den Anfang holen wir uns wie folgt: "`^Re(.*?):`" Also am Zeilenanfang soll das 'Re' stehen. Alles mögliche, was danach kommen kann, soll genommen werden; ggf. aber auch nichts, wenn nämlich gar kein Zähler dort steht. So etwas können wir natürlich mit "`.*`" fangen, aber dummerweise ist das gierig und würde gleich den ganzen anderen Rest des Betreff mitnehmen. Daher also das Fragezeichen.

Nach dem Doppelpunkt soll der Betreff folgen, den wir gerne in ein Subpattern hätten. Ok, nur sicherheitshalber fangen wir auch noch evtl. rumlungernde Leerzeichen davor ab: "^Re(.*?):\s*(.*?)" Im Subpattern 2 sollte sich also der eigentliche Betreff wiederfinden. Wir haben wieder ein Fragezeichen untergebracht, damit nur das gefunden wird, was zwingend nötig ist. Allerdings folgt dahinter noch gar kein Muster, so dass das Regex auch den geklammerten Alt-Betreff mit nimmt.

Das müssen wir auch noch verhindern. Dieser Teil beginnt auf jeden Fall nach einem oder keinem Leerzeichen mit einer Klammer und entweder dem 'was' oder dem 'war'.

Letzteres geht einfach "wa(s|r)" sollte beide Alternativen erledigen. Allerdings könnte dieser Teil auch fehlen! Wir dürfen also im Regex nicht darauf bestehen, dass dieser geklammerte Bereich existiert! Also müsste der Schluss des Regex etwa so aussehen: "\s*(\(wa(s|r):.*\))*$" oder insgesamt:

```
"^Re(.*?):\s*(.*?)\s*(\(wa(s|r):.*\))*$"
```

Das "$"-Zeichen soll sicherstellen, dass auch die gesamte Zeile gelesen wird. Beim Betreff dürfen wir davon ausgehen, dass es in ihm keine Zeilenumbrüche gibt. Wer sich da nicht so sicher ist, sollte dann besser "\Z" als Kennzeichen für das String-Ende verwenden.

Zu 3: Ok, ein sehr konstruiertes Beispiel. Aber manchmal braucht man die halt (ich erinnere mich an Physikaufgaben im Studium, die von "gewichtslosen Lametta-Fäden" ausgingen oder "eindimensionalen Kühen". War auch nicht schlauer ;-)) Sicher könnte man das Einlesen auch anders gestalten, aber ich wollte halt unbedingt ein konditionales Regex haben:

Zuerst brauchen wir eine Assertion, die nach irgendwas gefolgt von zwei Ziffern und einem Dollarzeichen sucht: "(?=.*\d{2}\$)" Wenn das nämlich existiert, dann soll die Dollarvariante gelesen werden: "([\d,]+)\.\d{2}\$" Die Vordezimalstellen habe ich einfach als Zeichenklasse definiert und nur Ziffern bzw. Kommata erlaubt (ok, Fehler möglich: eine Zeichenkette aus Kommata, dann Punkt und zwei Ziffern mit Dollar würde akzeptiert. Na, ihr dürft es gern verbessern ;-))

Ist aber diese Dollarvariante nicht da, dann soll das Regex schleunigst nach einer Euro-Variante suchen: "([\d\.]+),\d{2}EUR)" Auch hier die Zeichenklassendefinition als Vereinfachung.

Zusammen muss das dann so aussehen:

```
"(?(?=.*\d{2}\$)([\d,]+)\.\d{2}\$|([\d\.]+),\d{2}EUR)"
```

Ist euch beim Test aufgefallen, dass das EUR-Ergebnis im zweiten Subpattern steht, das Dollarergebnis aber im ersten?

8. Makros in TB

Nun endlich machen wir uns auf, unsere neu gelernte Sprache in TB anzuwenden. Dazu muss man wissen, welche Werkzeuge von TB überhaupt mit Regexisch arbeiten können. Diese Werkzeuge finden wir in den Makros.

8.1. Makros

Nur einige Makros haben unmittelbar mit Regulären Ausdrücken zu tun; viele andere eigentlich nicht. Auf sie können aber Regexe angewendet werden und das macht TB so mächtig und interessant.

Als erstes Makro sei hier %REGEXPTEXT="regex" genannt. Was macht es? Es sucht den originalen Nachrichtentext nach dem mit "regex" definierten Muster. Beispiel:

```
%REGEXPTEXT="[\d\.]+"
```

Dieses Makro in einer Mail eingesetzt, liefert als Ergebnis Ziffern bzw. Punkte als einfachen Text.

Betrachten wir gleich noch schnell ein ganz ähnliches Makro: %REGEXPQUOTES="regex" Dieses Makro macht nämlich genau das gleiche wie das vorige. Nur wird der gefundene Text nicht als einfacher Text, sondern als zitierter (gequoteter) Text zurückgeliefert.

Soll allerdings ein beliebiger Text wie die Kopfzeilen oder Einträge im Adressbuch, die TB über Makros zur Verfügung stellt, durchsucht werden, so benötigt man eine Kombination aus Makros:

Das erste wird %SETPATTREGEXP genannt. Es wird benutzt, um das Suchmuster zu definieren. Die Syntax ist einfach: %SETPATTREGEXP="regex". Mit "regex" ist das erstellte Regex gemeint.

Das zweite Makro ist %REGEXPMATCH. Auch hier ist die Syntax recht einfach: %REGEXPMATCH="string". "string" steht für jeden beliebigen Text und es ist ein Template. Das bedeutet, dass jedes beliebige Makro aus TB zur Erzeugung des Textes verwendet werden kann.

Die Definition des Regex mit %SETPATTREGEXP gilt übrigens für jedes %REGEXPMATCH bis es wieder durch ein neues %SETPATTREGEXP umdefiniert wird. Man kann also ein und dasselbe Muster auf mehrere Textelemente anwenden.

Bevor ich zu einem Beispiel komme: hatte ich gesagt, dass die Syntax recht einfach ist? Na ja, so allein betrachtet ist das Makro in der Tat recht simpel. Aber schauen wir mal, wie es aussieht, wenn man einen Text durchsuchen lässt.

Wir hatten oben das Makro %REGEXPQUOTES. Dieses kann man aber auch mit den %REGEXPMATCH-Makro erstellen. Vorgabe sei, dass eine Mail den Eintrag "Newsletter: yes" oder "Newsletter: no" hat. Wir wollen eine automatische Antwort generieren, die genau diese Zeile zitiert und etwas Standardtext verwendet.

Das Makro %QUOTES definiert, was als zitierter Text in TB verwendet werden soll. Wir müssen also eigentlich nur diesem Makro mitteilen, was es zitieren soll. Dann packen wir dieses in eine Antwortvorlage und schreiben noch unseren Standardtext darunter.

Zuerst einmal brauchen wir das Regex: "^Newsletter:\s*(yes|no)" Das definieren wir mit

```
%SETPATTREGEXP="^Newsletter:\s*(yes|no)".
```

Dann werden wir das %REGEXPMATCH benötigen, das auf das Makro %TEXT zugreift, in der der Originaltext der Mail steht. Die einzelnen Makros werden nun ineinander verschachtelt.

Ärgerlich ist, dass jedes dieser Makros die "-Zeichen für den Definitionsteil verwendet: bei %SETPATTREGEXP steht das "regex" dort, bei %QUOTES der Text, der Zitat verwendet wird usw. Verschachtelt man diese Makros nun ineinander, dann muss dem ersten Makro mitgeteilt werden, ob das nachfolgende Anführungszeichen das Ende der Definition ist oder der Beginn der Definition des zweiten Makros. Gleiches gilt für das Ende der Definition des zweiten Makros. Das kann man durch Verdoppeln der Anführungszeichen erreichen. Sie werden maskiert, wie wir das bei der literalen Suche von Metazeichen schon kennen.

Vereinfacht ist das %M1="%M2=""Def2""%M3=""Def3""" oder in unserem Beispiel:

```
%QUOTES="%SETPATTREGEXP=""^Newsletter:\s*(yes|no)""%REGEXPMATCH=""%TEXT"""
```

Als Abschluss braucht man dann drei Anführungszeichen: das erste zum Maskieren des Anführungszeichens als Ende der Definitionsteils, das zweite ist Ende des Definitionsteils des letzten Makros und das dritte ist Ende der Definition des ursprünglichen, ersten Makros.

Das obige Beispiel könnte einfacher aussehen:

```
%REGEXPQOUTES="^Newsletter:\s*(yes|no)"
```

aber das geht natürlich nur, wenn wir etwas aus %TEXT brauchen.

Kommen wir zu einem weiteren Makro, das wir benötigen, wenn wir nur bestimmte Teile aus der Mail herausziehen wollen. Bei den Regex kennen wir schon die Möglichkeit durch Klammersetzung Subpattern zu definieren. Nun müssen wir die in TB auch noch ansprechen können.

Um auf Subpattern zurückzugreifen, müssen wir %REGEXPBLINDMATCH="string" verwenden. Allerdings liefert dieses Makro alleine noch gar nichts – und das ist ja auch logisch: ich will ja nur bestimmte Zeichenketten bekommen. Ich muss also noch unbedingt mitteilen, welche Subpattern das sind. Dazu gibt es – na, wer hätte das gedacht - ein Makro: %SUBPATT="n". Das "n" steht für eine Ziffer, welche das n-te Subpattern bezeichnet.

Nun wird der Aufbau schon ein wenig unübersichtlicher. Ich werde das mal an einem Beispiel probieren, stückweise generieren und dann ineinander verschachteln.

Aus dem Originaldatum der Mail soll nur die Jahreszahl, zweistellig, zitiert werden. Das Datum liefert uns %ODATE. Das Regex muss lauten: "\d{2}(\d{2})\b". Wir wollen also aus dem Text nur dann die letzten beiden Ziffern extrahieren, wenn davor noch zwei stehen und eine Wortgrenze folgt. Das Makro lautet:

```
%SETPATTREGEXP="\d{2}(\d{2})\b".
```

Der Text, der durchsucht werden soll, wird mit dem Makro %REGEXPBLINDMATCH="%ODATE" definiert. Daraus wollen wir das erste Subpattern haben: %SUBPATT="1"

So, und nun in eine Zeile und die maskierenden Anführungszeichen nicht vergessen:

```
%QUOTES="%SETPATTREGEXP=""\d{2}(\d{2})\b""%REGEXPBLINDMATCH=""%ODATE""%SUB
PATT=""1"""
```

[Anm.: Das Makro wird aus Layout-Gründen umgebrochen. Tatsächlich muss sich alles in einer Zeile befinden!]

Noch ein Beispiel? Es existiert ein Regex für eine Antwortvorlage, die den Namen des Empfängers modifiziert. Statt 'Gerd Ewald' soll zum Beispiel "Gerd Ewald bei thebat-dt-beginner <thebat-dt-beginner@yahoogroups.de" eingetragen sein. Wir könnten uns das jetzt einfach aus dem Netz saugen, aber lasst es uns doch mal selbst ausprobieren.

Den Namen liefert uns %OFROMNAME.

Die Antwortadresse entnehmen wir dem Makro %OREPLYADDR. Aus dieser soll via Regex der Name der Mailingliste extrahiert werden. In der Regel ist das der Teil vor dem @-Zeichen: %SETPATTREGEXP="(.*?)\@"

Dieses wenden wir also mit %REGEXPBLINDMATCH="%OREPLYADDR" an, wobei wir nur das erste Subpattern brauchen: %SUBPATT="1"

Das Ergebnis soll später im TO-Feld stehen. Bitte beachten, dass dieses Feld zunächst geleert werden muss, bevor man seinen Text einträgt. Das geschieht durch eine initiale Leerzuweisung.

```
%TO=""%TO=''%OFROMNAME bei %-

%SETPATTREGEXP=_(.*?)\@_%-

%REGEXPBLINDMATCH=_%OREPLYADDR_%-

%SUBPATT=_1_" <%OREPLYADDR>'
```

[Anmerkung 1: Das Regex wurde mit dem %- Makro aufgeteilt und kann so eingegeben werden, wie es dort steht! Anmerkung 2: Die Vorlage wurde mit einem Feature neuerer TB-Versionen erstellt: jedes Zeichen kann als Delimiterzeichen verwendet werden. Die Auswahl ist also nicht nur auf das "-Zeichen beschränkt. In diesem Fall wurde der Unterstrich und das einfache Anführungszeichen verwendet. Die doppelte Eingabe von "-Zeichen entfällt dadurch. Anwender älterer Versionen müssen dagegen noch mit dem "-Zeichen arbeiten.]

Die tatsächliche Mailadresse müssen wir am Ende nochmals in "<>"-Zeichen einfügen.

Wie ihr seht, ist der Aufbau stereotyp. Er bleibt im Grunde gleich und ist auch nichts kompliziertes. Kompliziert ist eigentlich nur, herauszufinden, welches Makro mir die gewünschte Information liefert und wie ich die dann mit einem Regex herausbekomme.

Hier noch ein weiteres Beispiel. Ich habe es von der englischen FAQ-Seite bei Marck Pearlstone geholt (www.silverstones.com)

```
%WRAPPED='Historians believe that on %ODATE%-
%SETPATTREGEXP="(?m-s)Date\:\s*?((.*?[\d]{4})\s*?([\d]{0,2}\:%-
[\d]{0,2}\:[\d]{0,2})\s*?(.*))"%-
%REGEXPBLINDMATCH="%HEADERS" , at %SUBPATT="3"[GMT%SUBPATT="4"]%-
(which was %OTIME where I live) you wrote:'%-
```

Vorweg:

Hierbei wird ein Makro verwendet, das der besseren Lesbarkeit dient: %- Es bedeutet nichts, außer dass es am Zeilenende für TB bedeutet, die nachfolgende Zeile so zu werten, als würde sie unmittelbar an die aktuelle Zeile anschließen. Das %WRAPPED bedeutet dagegen, dass der sich aus dem Makro ergebende Text an der eingestellten Zeilengrenze automatisch umbricht.

Was macht nun das Makro?

Der erste Teil "%WRAPPED='Historians believe that on %ODATE%-" ist nur der Vorspann: bei einem Reply soll als Einleitung ein Text erscheinen, der das Datum der Originalmail an den Text 'Historians believe that on ' anhängt.

Der zweite Teil ist interessanter für uns, denn hier wird das Suchmuster, also das Regex mit %SETPATTREGEXP definiert (ich habe das %- entfernt, damit das Regex in einer Zeile erscheint):

```
"(?m-s)Date\:\s*?((.*?[\d]{4})\s*?([\d]{0,2}\:[\d]{0,2}\:[\d]{0,2})\s*?(.*))"
```

Es wird die Option Multiline ein- und die Option DotAll ausgeschaltet: (?m-s)

Dann wird nach der Zeichenfolge 'Date:' gefolgt von beliebig vielen oder keinen Whitespaces gesucht. Wegen der Gierigkeit wird die Suche nach Whitespaces mit dem Fragezeichen begrenzt. Der Autor hat zudem den Doppelpunkt mit dem Backslash maskiert: mir ist nicht klar, warum, aber um ehrlich zu sein: es schadet auch gar nicht.

Nun folgt die erste Klammer, die vermutlich nur aus Gründen der Übersichtlichkeit gesetzt wurde, denn auch sie darf entfallen (aber Achtung: es ändert sich die Zahl der Subpatterns!!), aber es stört nicht, wenn sie existiert!

Die zweite Klammer sucht nach allem, was von einer vierstelligen Zahl gefolgt wird. Wir wissen ja schon, dass das Regex auf die Kopfzeilen der Mail (%HEADERS) angewendet wird und dort nach 'Date' sucht. Also vermutlich will der Autor die Jahreszahl finden. Der dürfen weitere Whitespaces folgen.

Nun beginnt die dritte Klammer und auf die will der Autor zurückgreifen: er sucht nach drei null- bis zweistelligen Ziffern, die durch einen Doppelpunkt getrennt sind: die Uhrzeit! Danach erneut ein paar Whitespaces und im vierten Subpattern dann den ganzen Rest der Date-Zeile. Dies ist dann die Angabe, welche Zeitzone die Originalmail hat (GMT-Angabe).

Die %REGEXPBLINDMATCH-Angabe zeigt uns, dass dieses Regex auf die Kopfzeilen eingesetzt wird. Es soll aber nur das Subpattern 3, die Uhrzeit und das Subpattern 4, die Zeitzone ausgegeben werden.

Das Ergebnis könnte zum Beispiel so aussehen:

'Historians believe that on Sonntag, 7. April 2002 , at 11:22:59[GMT

+0200](which was 11:22 where I live) you wrote:'

Wie wir sehen, würde ein wenig Layoutarbeit an der Schnellvorlage nicht schaden ;-) aber sie funktioniert.

8.2. Andere Verwendungen in TB

Außer reguläre Ausdrücke in Makros zu verwenden, bietet TB noch weitere Einsatzgebiete.

Da wäre die Textsuche im Maileditor. Gerade bei langen Mails kann es sinnvoll sein, nicht einfach nur normal zu suchen, sondern ganz spezifische Suchkriterien vorgeben zu können.

Abbildung 1: Textsuche

Dieses Textsuchfenster lässt sich mit Strg-F oder über das Menü 'Bearbeiten' im Maileditor öffnen. In die Textzeile kann dann das Regex eingetragen werden. Im Bereich 'Optionen' muss dann aber die Option 'Reguläre Ausdrücke' gewählt werden.

So wie ich in einer Mail suchen kann, kann ich auch nach Mails suchen: F7 in der Ordneransicht liefert mir ein Suchfenster, in welchem ich ebenfalls reguläre Ausdrücke verwenden kann, um die Mails zu selektieren. Auf der Karteikarte Optionen trägt man in der Zeile 'Suche nach' das Regex ein.

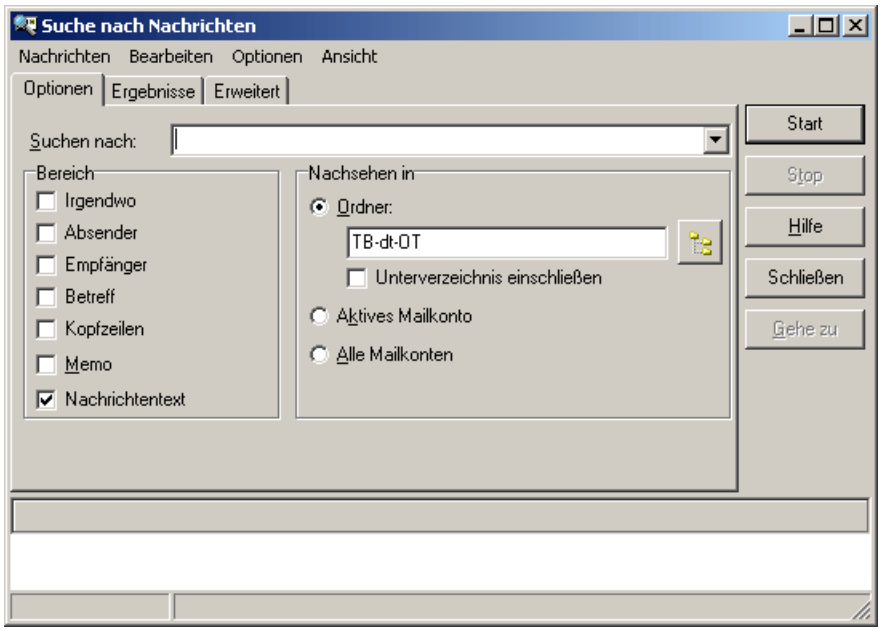

Abbildung 2: Suche nach Nachrichten - Optionen

Man muss dann auf den Karteireiter 'Erweitert' wechseln, um da dann in der zweiten Spalte die Option 'Reguläre Ausdrücke' anzuwählen.

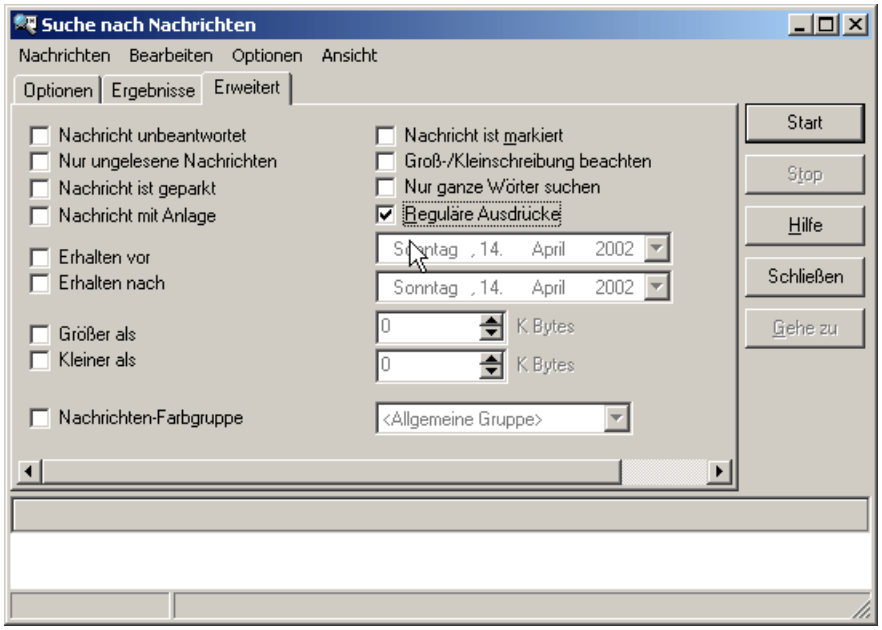

Abbildung 3: Suche nach Nachrichten - Erweitert

Ein Einsatzgebiet, in welchem die regulären Ausdrücke effektiv wie in Makros eingesetzt werden können, sind Filter. Auch hier lassen sich die Filterbedingungen zum Teil mit Regulären Ausdrücken definieren. Dazu muss im Filterassistent auf den Karteireiter 'Optionen' gewechselt und die Option 'Reguläre Ausdrücke' gewählt werden.

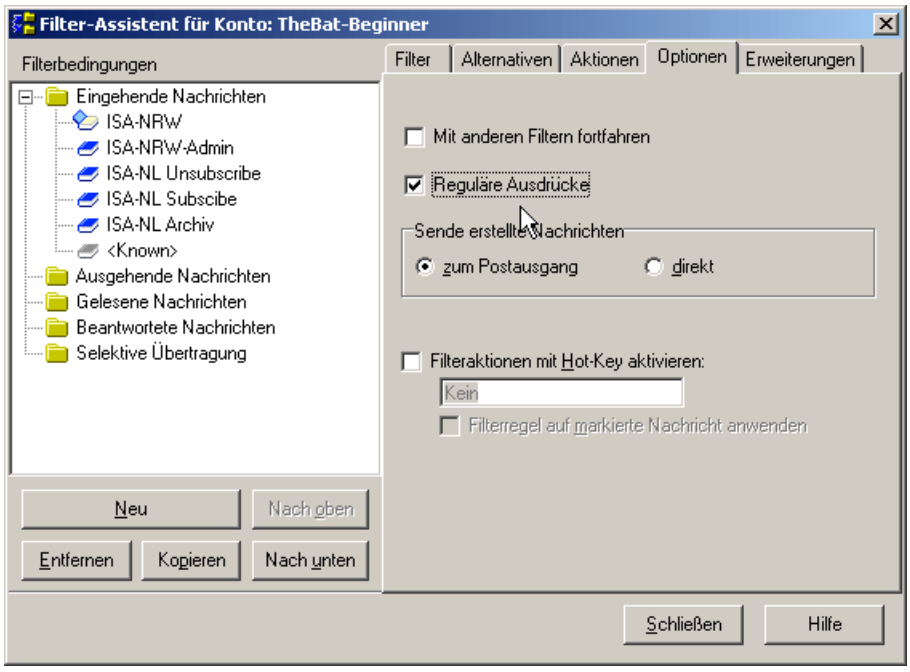

Abbildung 4: Filter - Optionen

8.3. Übersicht über dieses Kapitel

Was hat uns das letzte Kapitel gebracht?

Wir haben erfahren, wie in TB reguläre Ausdrücke an den verschiedenen Stellen eingesetzt werden können:

- TB bietet Makros, die mit regulären Ausdrücken zur Textmanipulation verwendet werden:
  - %REGEXPTEXT="regex" liefert die gefundene Zeichenkette in der Nachricht als Text zurück
  - %REGEXPQUOTES="regex" liefert die gefundene Zeichenkette in der Nachricht als zitierten Text zurück
  - %REGEXPMATCH="string" definiert den generischen Text, in dem das Regex etwas finden soll und liefert den Text zurück. Dabei kann jedes Makro als Textlieferant (string) verwendet werden.
  - %REGEXPBLINDMATCH="string" definiert den generischen Text, in dem das Regex etwas finden soll, liefert aber keinen Text zurück, sondern erwartet %SUBPATT zur Angabe, welcher Textteil zurückgeliefert werden soll. Dabei kann jedes Makro als Textlieferant (string) verwendet werden.
  - %SETPATTREGEXP="regex" definiert das Regex für die Makros %REGEXPMATCH und %REGEXPBLINDMATCH. Es bleibt solange gültig, bis es durch ein weiteres %SETPATTREGEX neu definiert wird.
  - %SUBPATT="n" liefert in Verbindung mit %SETPATTREGEXP und %REGEXPBLINDMATCH das n-te Subpattern zurück.
- Reguläre Ausdrücke können in TB zum Suchen nach und in Mails sowie zum Filtern der eingehenden Mails verwendet werden

Aufgabe 1:

In einem früheren Kapitel haben wir ein einfaches Regex erstellt, das die Betreffzeile säubert:

```
"^Re(.*?):\s*(.*?)\s*(\(wa(s|r):.*\))*$"
```

Vielleicht versucht ihr euch mal an einer Erweiterung: statt des Zusatzes '(war: xyz)' steht dort für Benutzer von PGP nach dem Entschlüsseln einer Mail '(PGP Decrypted)'. Das Regex soll auch dieses finden. Zudem soll dann dieses Regex in einer Antwortvorlage verwendbar sein.

Aufgabe 2:

Weiter oben haben wir ein Regex, dass die To-Adresse für Gruppen modifiziert:

```
%TO=""%TO="""%OFROMNAME bei
%SETPATTREGEXP="(.*?)\@""%REGEXPBLINDMATCH=""%OREPLYADDR""%SUBPATT=""1""
"" <%OREPLYADDR>"
```

Bitte ändert dieses doch so ab, dass nicht mit %REGEXPBLINDMATCH und %SUBPATT sondern nur %REGEXPMATCH erforderlich ist. Dazu müsst ihr das Regex modifizieren. Hinweis nötig? Ok: das Subpattern musste nur deswegen gebildet werden, weil sonst auch das @-Zeichen zur gefundenen Zeichenkette gehört hätte. Also muss man ein Regex finden, welches das @-Zeichen nicht frisst.

Lösung zu 1:

Nun, eigentlich ist das nicht schwierig. Man wandelt einfach das letzte Subpattern in eine Alternativsuche:

```
"^Re(.*?):\s*(.*?)\s*(\(wa(s|r):.*\)|\(PGP Decrypted\))*$"
```

Nun aber zur Vorlage: wir wollen ein neues Subject bauen, also müssen wir auch das Makro anwenden: %SUBJECT. Da wir es bei Antworten verwenden und wir mit einem sauberen 'Re' beginnen wollen, muss der Anfang also so aussehen:

```
%SUBJECT="Re
```

Nun folgt das Regex:

```
%SUBJECT="Re: %SETPATTREGEXP=""^Re(.*?):\s*(.*?)\s*(\(wa(s|r):.*\)|\(PGP
Decrypted\))*$""
```

Wir wenden es auf das vollständige Originalsubject %OFULLSUBJ an und verwenden für die weitere Bearbeitung nur das zweite Subpattern %SUBPATT="2"

```
%SUBJECT="Re: %SETPATTREGEXP=""^Re(.*?):\s*(.*?)\s*(\(wa(s|r):.*\)|\(PGP
Decrypted\))*$""%REGEXPBLINDMATCH=""%OFULLSUBJ""%SUBPATT=""2"""
```

[Anm.: Das Makro wird aus Layout-Gründen umgebrochen. Tatsächlich muss sich alles in einer Zeile befinden!]

So, das müsste klappen. Zusatzaufgabe: was aber, wenn der Betreff gar kein 'Re' am Anfang aufweist, sondern 'AW:', 'Betrifft:' oder 'FWD' bzw. andere Alternativen? Versucht es mal selbst, diese Alternativen am Anfang in das Regex einzubauen.

Lösung zu 2:

Eine positive Lookahead-Assertion ".*?(?=\@)" Das @-Zeichen wird nun durch die Assertion gesucht und wenn es gefunden wird, gehört es nicht zur zurückgelieferten Zeichenkette. Die Schnellvorlage ist daher etwas kürzer:

```
%TO=""%TO="""%OFROMNAME bei
%SETPATTREGEXP="".*?(?=\@)""%REGEXPMATCH=""%OREPLYADDR"" "" <%OREPLYADDR>"
```

9. Schluss

So, nun wollen wir mal schauen, ob wir unser Eingangsbeispiel wenigstens ein bisschen erklären können:

```
%QUOTES="%SETPATTREGEXP =""(?is)(-----BEGIN PGP
SIGNED.*?\n(Hash:.*?\n)?\s*)?(.*?)(^(- --|--\n|-----BEGIN PGP SIGNATURE)|
\z)""%REGEXPBLINDMATCH=""%text""%SUBPATT=""3"""
```

Es beginnt mit %QUOTES=. Offensichtlich soll der mit dem Regex gefundene Text als zitierter Text weiterverwendet werden.

"%SETPATTREGEXP="" definiert das Regex:

```
(?is)(-----BEGIN PGP SIGNED.*?\n(Hash:.*?\n)?\s*)?(.*?)(^(- --|--\n|-----
BEGIN PGP SIGNATURE)|\z)
```

[Anm.: Das Regex wird aus Layout-Gründen umgebrochen. Tatsächlich muss sich alles in einer Zeile befinden!]

Die doppelten Anführungszeichen sind uns mittlerweile klar? Wir müssen die Zeichen maskieren, damit sie nicht fälschlicherweise zum anderen Makro gezählt werden.

"(?is)" setzt die Optionen: Ignoriere Groß- und Kleinschrift, betrachte den ganzen Text als eine Zeile und lasse den Punkt auch auf das Zeichen "Newline" wirken

```
"(-----BEGIN PGP SIGNED.*?\n(Hash:.*?\n)?\s*)?"
```

Gleich zu Beginn wird mit "Klammer auf" das erste Subpattern definiert. Anweisung an die Regexp-Maschine lautet: Finde 5 Bindestriche gefolgt von der Buchstabenkombination BEGIN PGP SIGNED. Danach dürfen beliebige Zeichen beliebig häufig oder auch gar nicht folgen (.*?). Wegen der Gierigkeit wird das mit einem "?" eingeschränkt. Es soll dann eine neue Zeile (\n) folgen.

In dieser neuen Zeile soll 'Hash:' gefolgt beliebigen Zeichen und einem Zeilenumbruch stehen. Dieses ist das zweite Subpattern, dass höchstens einmal dort stehen darf; es darf aber auch fehlen. Dieses bewirkt das Fragezeichen direkt hinter der zweiten Gruppierung. Dieser zweiten Klammer dürfen beliebig viele oder auch gar keine Whitespace-Zeichen folgen. Dann wird das erste Subpattern mit ")" geschlossen. Auch hier folgt nun ein Fragezeichen: dieses erste Subpattern darf nur einmal auftauchen oder aber gar nicht, um das Regex insgesamt zu erfüllen.

Diese Zeilen werden von PGP beim Clear-Signing produziert. Sie sind standardisiert und lassen sich daher gut mit Regex finden. Da der Autor dieses Makro aber auch auf Text anwenden will, der nie PGP gesehen hat und daher diese Zeilen auch gar nicht aufweist, setzt er die Möglichkeit, dass dieses nicht vorhanden sein muss.

```
"(.*?)"
```

Dies ist der wichtige Subpattern drei: er enthält die eigentliche Mail!! Das davor war nötig, um diesen Subpattern lokalisieren und isolieren zu können. Die Syntax sagt "Finde alles beliebig häufig oder gar nicht". Um die Gierigkeit einzuschränken, wurde wieder das Fragezeichen angefügt.

Nun folgt eine Alternativsuche:

```
"(^(- --|--\n|-----BEGIN PGP SIGNATURE)|\z)"
```

Zunächst wird Subpattern 4 begonnen und nach dem zwingenden Zeilenanfang gesucht. Alles, was in der nachfolgenden Alternative gesucht wird, muss auf jeden Fall am Zeilenanfang stehen "(^". Dann das Subpattern 5:

```
"(- --|--\n|-----BEGIN PGP SIGNATURE)"
```

Subpattern 5 besteht selber aus drei Alternativen:

```
"- --" bzw. "--\n" oder "-----BEGIN PGP SIGNATURE"
```

Das erste ist vielen bekannt, die PGP-clear-signed Nachrichten gesehen haben: es ist der verkrüppelte Sig-Trenner. PGP markiert sich Zeilen, die mit einem Bindestrich beginnen, damit es die von seinen eigenen Zeilen unterscheiden kann. Nicht sehr glücklich, aber nicht an dieser Stelle diskussionsfähig. Wir nehmen das mal so hin.

Das zweite ist unser bekannter Sigtrenner, gefolgt von einem Zeilenumbruch. Aha, falls also die Mail gar nicht von PGP berührt wurde, soll eben diese Alternative greifen.

Und die dritte Alternative sucht nach den Zeilen, die PGP erstellt, um dahinter den sogenannten Hash abzulegen (ok, ok, nur Teile des Hashes, aber dies ist ein Reg-ex- und kein PGP-Kurs. Dazu schaut auf den geeigneten Seiten nach ;-)). Damit wird das Subpattern 5 abgeschlossen.

Die zweite Alternative des Subpattern 4 "\z)" sucht statt der Zeilenanfangsversionen des Subpattern 5 nur noch das Ende der Zeichenkette. Es muss also weder ein kaputter noch ein funktionierender Sigtrenner oder gar ein PGP-Hashwert dort stehen. Die Mail kann auch unvermittelt nach dem Text enden.

Tatsächlich wird dieses eigentümliche Ende nur aus einem Grund gesucht: man will den eigentlichen Text der Mail vom Rest, also im wesentlichen von den PGP-Zusätzen und –Veränderungen, trennen. Ein weitergehendes Interesse hat der Autor an diesen Textstücken nicht, es dient nur der Abgrenzung.

Nun folgt das nächste Makro:

```
%REGEXPBLINDMATCH=""%text""
```

welches sagt: wende den festgelegten Suchpattern auf den Text der Mail an.

Das

```
%SUBPATT=""3""""
```

liefert uns den Inhalt von Subpattern 3 zurück an die Variable %QUOTES Dies war, wie wir oben festgestellt haben, der eigentliche Mailtext.

So, das war es! Ich hoffe, ihr habt etwas mitnehmen können.

Ein Kurs, der ohne direktes Feedback gehalten wird, ist etwas ungewohnt: man sieht nicht am Gesicht der Kursteilnehmer, wenn man zu akademisch und kompliziert wird. Ich habe mich bemüht, dies gerade nicht zu werden und nur die Punkte von Regulären Ausdrücken zu zeigen, die man am häufigsten gebrauchen kann. Ich hoffe also, dass es mir gelungen ist.

Der Kurs ist keine vollständige Darstellung, dazu hätte ich einfach nur den Friedl und die deutsche bzw. englische Hilfe zu TB kopieren müssen ;-) Nein, der Kurs sollte euch den Anreiz geben, selber Regexe zu schreiben und zu probieren. Wie bei einer richtigen Sprache lernt man auch in diesem Fall nur durch Anwenden des Vokabulars und der Grammatik. Wenn mir das gelungen ist, dann reicht mir das schon :-)

10. Liste der Elemente von Regulären Ausdrücken und Makros

Zeichen	Bedeutung
.	Beliebiges Zeichen außer Newline. Im Modus DotAll (?s) wird auch Newline erkannt
^	Erkennt den Zeilenanfang
$	Erkennt das Zeilenende
...|...|...	Stellt Alternativen für das Suchmuster. Die erste auftretende Alternative im String wird gefunden.
(...)	Dient der Gruppierung von Suchmustern. Das gefundene Muster wird in ein Subpattern für spätere Verwendung gespeichert.
[...]	Zeichenklasse: die in den eckigen Klammern stehenden Zeichen werden als Alternative verwendet. Es können Bereiche angegeben werden: [a-p]. Einige Metazeichen haben hier andere Bedeutung. [^...] negiert die Klasse.
\	Backslash: hebt die besondere Bedeutung von Metazeichen auf: +?.*()^$[{|\ um diese literal suchen zu können
*	Erkennt vorhergehendes Element nicht oder mehrmals
+	Erkennt vorhergehendes Element ein- oder mehrmals
?	Erkennt vorhergehendes Element nicht oder einmal ODER Hebt die Gierigkeit der anderen Quantifizierer auf (.*?), minimal nötige Menge wird gefunden ODER Leitet einen Modifikator ein (?i) ODER Leitet konditionale Regexe (?(....)....|....) sowie Assertions (?=...) ein
{x,y}	Erkennt vorhergehendes Element x- bis höchstens y-mal. ',y' ist optional: {x} erkennt das Element x-mal. {x,} erkennt das Element mindestens x-mal, aber auch beliebig häufiger.
\d	Erkennt Ziffern

Zeichen	Bedeutung
\D	Erkennt Nicht-Ziffern
\w	Erkennt alphanumerische Zeichen und den Unterstrich
\W	Erkennt Nicht-\w-Zeichen
\s	Erkennt Whitespaces (Tab, Leerzeichen usw.)
\S	Erkennt Nicht-Whitespaces
\b	Erkennt Wortgrenzen
\B	Erkennt Nicht-Wortgrenzen
\Z	Erkennt das Ende des Strings bzw. das Zeichen unmittelbar vor dem Newline am Ende
\z	Erkennt das tatsächliche, physikalische Ende des Strings
\A	Erkennt den Anfang des Strings
\1...\9	Rückbezüge. Sie verweisen auf ein zuvor in (...) gefundenes Subpattern. Bei entsprechend vielen Subpattern sind auch Zahlen größer 9 möglich
\nnn	Erkennt Oktalzahl nnn. Besonderheiten sind zu beachten
\xnn	Erkennt Hexadezimalzahl nn
(?:...)	Gruppierung wird nicht als Subpattern zwischengespeichert.
(?(Bedingungsmuster)Ja-Muster|Nein-Muster)	Konditionales Regex: erkennt das Ja-Muster, wenn das Bedingungsmuster in der Zeichenkette enthalten ist, ansonsten erkennt es das Nein-Muster. Das Nein-Muster ist optional.
…(?=…)	Positive Lookahead-Assertion. Das Muster vor der Klammer wird nur erkannt, wenn das Muster in der Klammer danach gefunden wird. Das Muster in der Klammer ist im Suchergebnis nicht enthalten.
…(?!…)	Negative Lookahead-Assertion. Das Muster vor der Klammer wird nur erkannt, wenn das Muster in der Klammer danach nicht gefunden wird. Das Muster in der Klammer ist im Suchergebnis nicht enthalten.
(?<=…)…	Positive Lookbehind-Assertion. Das Muster nach der Klammer wird nur erkannt, wenn das Muster in der Klammer davor gefunden wird. Das Muster in der Klammer ist im Suchergebnis nicht enthalten.
(?<!…)…	Negative Lookbehind-Assertion. Das

Zeichen	Bedeutung
	Muster nach der Klammer wird nur erkannt, wenn das Muster in der Klammer davor nicht gefunden wird. Das Muster in der Klammer ist im Suchergebnis nicht enthalten.
(?Modifikator)	Schalter zum Setzen oder Ausschalten von Modifikationen zur Mustererkennung. Durch Aufführen eines Modifikators wird dieser eingeschaltet. Ein Minus-Zeichen davor schaltet ihn aus. Es dürfen mehrere Modifikatoren gleichzeitig eingetragen sein (?i-s). i ignoriert Groß-/Kleinschreibung m betrachtet den String als mehrzeilig, ^ und $ erkennen jedes Newline-Zeichen in der Zeichenkette s betrachtet den String als einzeilig, der Punkt erkennt auch im String befindliche Newline-Zeichen x ermöglicht Kommentare und zusätzliche Whitespaces im Muster, die nicht erkannt werden. Leerzeichen müssen ggf. mit Backslash maskiert werden A fixiert das Suchpattern an den Anfang der Zeichenkette. D findet „$“ nur noch am Ende der Zeichenkette, wenn das letzte Zeichen dort ein Zeilenumbruch ist. Andere, vorhergehende Umbrüche werden ignoriert. U kehrt die Standardeinstellung für Gierigkeit der Quantifizierer um: Quantifizierer sind nie gierig, es sei denn sie werden von einem Fragezeichen gefolgt. u Suchstring wird als UTF-8 interpretiert. X Zeichen, die keine Metazeichenbedeutung haben und mkit einem Backslash maskiert werden, führen zu einem Fehler. e Evaluierung von Backreferences oder Substitutionen als PHP-Code (nur in preg_replace verwendbar) S zusätzliche Analyse des Suchpatterns zur Beschleunigung bei häufiger Verwendung des Regex **Achtung: einige Modifikatoren sind weder in TB noch in Perl verwendbar!**

11. Beispiele für verschiedene reguläre Ausdrücke

Im abschließenden Kapitel möchte ich verschiedene Beispiele für reguläre Ausdrücke geben, die zum einen im täglichen Gebrauch hilfreich sein können und zum anderen verschiedene in diesem Tutorial dargestellte Syntaxregeln verdeutlichen.

Die unten aufgeführten Regexe wurden verschiedenen Quellen entnommen.

Prüfung auf korrekte E-Mailadressen-Syntax

```
[\w-]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}
```

Dieses Regex trifft auf 99% aller gängigen Mailadressen zu. Eine perfekte Prüfung auf Mailadressen ist nahezu unmöglich: der Aufwand und die Fehleranfälligkeit des Regexes steigt überproportional mit der Komplexität des Regexes.

Prüfung auf erlaubte IP-Adresse

```
(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|
2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-
9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-
1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[0-9])
```

Die Form dieses Regexes sollte aus einer der Aufgaben bekannt sein: Prüfung auf ein (mehr oder weniger) gültiges Datum. Auch dort wurde das Regex in Teilprobleme zerlegt. Im Falle der IP-Adressen sind bestimmte Nummern ausgeschlossen (0.0.0.0 oder jeder Wert größer 255) und die Schreibweise ist individuell (192.0.0.1 oder 192.000.000.001): beides wird berücksichtigt.

Prüfung auf mehrfach auftretende gleichen Empfänger in E-Maildresse

```
((.*?)@.*?,\s*)(\2@.*?(,\s*)?){2,}
```

Diese Prüfung geht von einer Auflistung von Emailadressen der Art

"person@web.de, person@gmx.de, person@weissnicht.de"

aus. Die Zeichen vor dem @-Zeichen werden gespeichert und im Subpattern 2 abgelegt `((.*?)@.*?,\s*)`. Der zweite Teil `(\2@.*?(,\s*)?)` verwendet das Ergebnis des Matches und prüft die nachfolgenden Adressen über einen Rückbezug auf das zweite Pattern `\2`. In Fällen, in denen die Adressen aber wie folgt geschrieben werden

```
"Person 1 <person@web.de>, Person 1 <person@gmx.de>,   Person
1 <person@weissnicht.de>"
```

reicht dieses Regex nicht aus. Stattdessen sollte dann das universellere

```
(((.*?)\s*<)?(.*?)@.*?>?,\s*)(((.*?)\s*<)?\4@.*?>?(,\s*)?){2,}
```

verwendet werden. Achtung: in diesem Fall musste der Rückbezug geändert werden. Der Empfänger in der Mailadresse ist nunmehr im vierten Klammerpaar und wird daher mit `\4` angesprochen. In beiden Fällen gibt die Ziffer in der geschweiften

Klammer am Ende des Ausdrucks an, wie häufig der gleiche Empfänger zusätzlich aufgeführt werden muss.

Prüfung auf mehrfach auftretende gleiche Domain in E-Mailadresse

```
(.*?(@.*?),\s*)(.*?\2(,\s*)?){3,}
```

Viel häufiger als die Mehrfachnennung des Empfängernamens kommt die Mehrfachnennung der Domain bei Spam vor:

Albert.a@web.de, berta.b@web.de, charlie.c@web.de, dora.d@web.de

Diese werden durch das obige Regex gefunden. Auch hier wird wieder ein Backreference verwendet.

Bedingte Prüfung auf eine Vielzahl von Ziffern im Betreff

Die Überschrift an sich ist schon kryptisch ;-) Gemeint ist, dass nur Betreffs gefunden werden, in denen eine Vielzahl von Ziffern vorkommen. Klassischer Fall, der in den selektiven Downloadfiltern verwendet wird. Allerdings birgt sowas eine Gefahr, denn zum Beispiel erhalten eBay-Käufer oder auch -Verkäufer Mails mit der 10-stelligen Artikelnummer und diese würden dann auch im Spam-Filter landen. Also muss ein Regex her, dass eine Bedingung in sich trägt, nämlich "finde nur Ziffern, wenn davor nicht 'Artikelnummer' steht".

```
.*?(?<!Artikelnummer:) \d{5,}
```

Hierbei wird eine negative Lookbehind-Assertion verwendet, die jede mindestens 5-stellige Ziffernfolge findet, die das Wort "Artikelnummer:" nicht davor hat. Sobald "Artikelnummer:" den 5 Ziffern vorausgeht, findet das Regex diese: in einem selektiven Downloadfilter verwendet bliebe diese Mail am Server hängen.

Prüfung auf mehrere aufeinander folgende Konsonanten

Verschiedene Spammer verwenden wechselnde Mailadressen von verschiedensten Domänen. Merkmal dieser Mailadressen ist, dass sehr viele Konsonaten aufeinander folgen: fndghklxrrstU@beispiel.com Wenn man nicht generell die Domäne im selektiven Download sperren will, so muss man ein anderes Vorgehen wählen. Eine Methode ist es, eine Zeichengruppe zu definieren, die alles erlaubt, nur keine Vokale:

```
(?i)[^aeiou]{5,}
```

Nimmt man mal an, dass in einem mehr oder weniger sinnvoller Name nicht mehr als 5 Konsonanten direkt aufeinander folgen, so kann das obige Regex schon zum Filtern verwendet werden.

12. Stichwortverzeichnis

13. Danksagung

Mein Dank gilt an dieser Stelle all denen, die den Inhalt geprüft, korrigiert und in die Online-Tauglichkeit überführt haben, sowie denen, die zum Teil in Telefonaten und Gesprächen, wissentlich oder unwissentlich, Anregungen gegeben haben. Dies sind (in alphabetischer Folge):

Januk Aggarwal
Bert Bohla
Dirk Heiser
Thomas Martin
Hanja Nowicka
Peter Palmreuther
Stefan Peukert
Alfred Rübartsch
Andreas Rumpenhorst
Ingrid Spitzer
Carsten Thönges
Karin Uhlig
Arnd Wichmann
Thomas Woelk

Mein Dank geht ferner an das OpenOffice-Team bei www.openoffice.org, mit deren Produkt, welches beim Suchen und Ersetzen ebenfalls Regexe verwenden kann, dieses Tutorial erstellt wurde.

Die PDF-Datei wurde mit PDF995 und PDFedit von www.pdf995.com erstellt.