1 Reguläre Ausdrücke

Dieser Workshop ist für Einsteiger in die Problematik regulärer Ausdrücke gedacht. Er ist eine verkürzte und überarbeitete Version eines Tutorials, das ich auf meiner Website für Benutzer eines Mailprogramms TheBat! anbiete, ohne dass dieser dadurch ersetzt wird. Dieses Tutorial ist wirklich dazu geschrieben, ausschließlich die Syntax zu erlernen, unabhängig davon, wo man später die Regexe verwenden will. Ich habe mich bemüht, die Beispiele und Aufgaben allgemeinen Problemen anzupassen. Allerdings wird die Leserin oder der Leser feststellen, dass eine Vielzahl der Beispiel-Regenechsen aus der Welt der Mails stammen. Ich hoffe dennoch, dass der Workshop als Einstieg zum allgemeinen Verständnis der faszinierenden Welt der ›Regenechsen‹ geeignet ist.

1.1 Was genau heißt ›Regulärer Ausdruck‹

Regexe finden sich in vielen UNIX-Tools, in Programmiersprachen wie Perl (Practical Extraction and Report Language), PHP, Javascript oder sogar in verschiedenen Editoren wie UltraEdit oder jEdit. 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.

Vielleicht ist die die richtige Stelle, darauf hinzuweisen, dass es wie bei jeder Sprache auch bei Regexisch Dialekte gibt. Ich konzentriere mich bei der Darstellung auf den bei PERL verwendeten Dialekt PCRE. Andere Tools wie z.B. bei GNU grep verwenden andere Dialekte. Zwar ist die Grundstruktur gleich, deswegen ist dieses Tutorial dennoch ein Einstieg für alle, aber Kleinigkeiten können halt abweichen. Bitte informiert euch zuvor, ob PCRE verwendet wird. Im Kapitel 5.4 erwähne ich diesen Unterschied explizit für die Optionen und Modifikatoren.

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, welche – wie wir bald sehen werden- eben nicht nur Platzhalter für Zeichen sind.

2 Einfache Suchmuster

Um im folgenden Beispiele für Regexe geben zu können, müssen wir uns auf eine Darstellungsform einigen. Ich werde die Regexe mit Anführungszeichen begrenzen und als Code formatiert, also so. Wollt ihr sie ausprobieren, so müsst ihr die Eintragungen zwischen den ›-Zeichen verwenden. Testen? Ja, man kann die Wirksamkeit der Muster in der Hilfe testen: Bitte, ladet euch den Regex-Coach für den Kurs herunter, falls ihr die Regexe testen wollt.

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 die Hilfe lesen!

2.1 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 Paradies oder das Weib‹ gefunden.

2.2 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.

(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 Workshops 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: ›\\‹

2.3 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 *g*? 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.

2.4 Zeichengruppen und -klassen

\d Ziffer
\w beliebiger Buchstaben, alphanumerische Zeichen
\W nicht-alphanumerische Zeichen
\D jedes Zeichen, das keine Ziffer ist
[ ] für Zeichenklassen

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 aufeinanderfolgenden Ziffern.

›\w‹
steht für einen beliebigen Buchstaben oder einen Unterstrich, auch alphanumerische Zeichen genannt.

›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.

Die Regex bieten auch das jeweilige Gegenstück zu den beiden oben genannten:

›\W‹
für jedes beliebige nicht-alphanumerische Zeichen

›\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]‹
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:

›[^1-4]‹
›Finde alle Zeichen, sofern sie keine 1, 2, 3 oder 4 sind!‹ 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.

2.5 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]‹

Lösungen
›\d\d\.\d\d\.\d\d\d\d‹
sucht 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

›\w\w\w, \d\d \w\w\w \d\d\d\d‹
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.

›.. \[[0-9]\]:‹
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]:‹

›[a-zA-Z]‹
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.

3 Komplexe Suchmuster

So, das war ja einfach. Allerdings auchwenig 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:

3.1 Zeilengrenze

^ Zeilenanfang
$ Zeilenende

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. 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.

3.2 Wortgrenzen

\b Wortgrenze
\B nicht Wortgrenze

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 gefunden. Ich erinnere mich an eine Diskussion in einer Mailingliste 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 im Tester ergibt ›Rep‹. Tauscht mal \B gegen \b aus und es wird ›Re:‹ gefunden. Alles klar?

3.3 Alternativen

| entweder oder

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 ;-)

3.4 Besondere Klassen

\s Whitespace-Zeichen
\S nicht Whitespace-Zeichen
\A Anfang einer Zeichenkette
\Z Ende einer Zeichenkette

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. 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*.

3.5 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 ›Foo:‹ am Zeilenanfang finden; allerdings auch dann, wenn ihr noch der etwas folgt, also zum Beispiel ›Foobar:‹. Wie muss das Regex nach unserem bisherigen Stand aussehen?

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

4
Was bedeuten die folgenden Suchmuster?
›^‹
›^x$‹
›^$‹

Zu 1

›(R.:$|^R.:)‹

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

Zu 2

Hoppla, das zweite Beispiel ist ja schon etwas anspruchsvoller:

›^(Foo|Foobar):‹

Ok, man kann es vereinfachen:

›^Foo(|bar):‹

Um nach nichts hinter dem ›Foo‹ 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.

Zu 3

›(^Der|\)$)‹

wäre die Lösung. Ihr hattet hoffentlich an den Backslash vor der Klammer gedacht? Prima, dann habt ihr aufgepasst *g*. Probiert es mal im Tester aus. Ihr werdet erkennen, dass das Regex aus der Zeichenkette ›Der Regen fällt (was aber keinen interessiert)‹ nur genau ›Der‹ findet, denn da war die Bedingung erfüllt. Ändert den Zeilenanfang, indem ihr irgendetwas eintragt und schon wird das ›Der‹ nicht mehr gefunden. Stattdessen wird die abschließende runde Klammer gefunden.

Zu 4

›^‹

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!

›^x$‹

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.

4 Quantifizierer, Subpattern, Gruppierung

Bislang, in den ersten beiden Teilen, 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.

4.1 Quantifizierer

+ Zeichen davor muss mindestens 1 x in der Zeichenkette vorkommen
***** Zeichen davor muss beliebig häufig oder gar nicht existieren
? Zeichen davor höchstens einmal vorkommen, aber nicht muss
x Zeichenwiederholungen, mind.
y Zeichenwiederholungen, max.

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.

▶ 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.

▶ Das ›*‹-Stern

legt fest, 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+‹

Dies ist ein Regex, mit dem wir den Anfang von Betreffzeilen aus Mails erkennen wollen (wir werden später erkennen, wofür wir uns eigentlich so häufig um die Betreffzeile kümmern und was wir mit Regexen daraus machen wollen. Ihr müsst euch leider gedulden) und dabei können einige fehlerhafte Formatierungen vorliegen, die wir natürlich mit dem Regex erkennen müssen. 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 ›**?**‹-Zeichen
bedeutet, dass das Zeichen davor höchstens einmal vorkommen darf, aber nicht muss, also mit der ›Häufigkeit 0‹. ›H..?se‹ findet ›Hase‹, aber auch ›Hirse‹.

▸ ›**x,y**‹ 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:

4.2 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‹.

Beispiel

›Re\s*(\[\d+\])?:‹

Schon wieder unsere Betreffzeile! 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

›Re\s*(\[\d*\])?:‹

Das ›+‹-Zeichen müsste nur gegen einen Stern ausgetauscht werden:

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 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. Aber immerhin haben wir jetzt den Vorspann eines Betreffs schon ganz gut im Griff und könnten uns auf das konzentrieren, was nach diesem Vorspann kommt: der eigentliche Betreff!

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!‹

Beispiel
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 Regexe zum Einsatz kommt. Zeichenketten, die von eingeklammerten Suchmustern gefunden wurden, werden in eine temporäre Variable (Subpattern) für spätere Zwecke zwischengespeichert. Das schaut man sich am besten in einigen Beispielen an:
›erika.mustermann@beispiel.de‹
›(\w+)\.(\w+)@.*‹
Die erste Klammer passt auf ›erika‹, die zweite ›mustermann‹. Und genau diese beiden Zeichenketten landen in den Subpattern_1 bzw. Subpattern_2.

Oder
›22.02.‹
›(\d+\.)(\d+\.)‹
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.

Der Regex-Tester zeigt euch die Subpattern an. Der kleine Karteireiter unten mit der ›0‹ zeigt den gesamten ›Fund‹ an, der mit der ›1‹ den ersten Subpattern usw. Tatsächlich werden bei den Programmiersprachen die gefundenen Elemente in ein Array abgelegt. Das Element mit dem Index 0 beinhaltet in der Tat den gesamten Match; die Subpattern sind in den Elementen mit der korrespondierenden Index-Ziffer.

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 zwischen den Hochkommata. 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. Also:
›(.*)\.(.*)*@(.*)\.(.*)‹

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 eine neue Bedeutung des Fragezeichens kennen (immer mit der Ruhe: es ist erst die zweite Bedeutung, es kommen noch mehr. Aber das sollte erklären, 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, als wäre das Fragezeichen gar nicht da 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 ausreicht, 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. Ihr könnt euch denken, dass dies die Regex-Maschine erheblich beansprucht und es die Performance drückt, aber manchmal ist es halt nötig.

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!

4.3 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. Sie werden *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. Stellt euch vor, dass die Informationen in einem einzigen String stehen und die Zeilenumbrüche gar nicht existieren.

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.

e

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:

a und **b** mit den Monaten

›(0?[1-9]|[12][0-9])\.(0?[1-9]|1[0-2])\.‹

c mit den Monaten

›30\.((0?[13-9])|(1[0-2]))\.‹

d mit seinen Monaten

›31\.(0?[13578]|1[02])\.‹

e die Jahre

›(\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‹

Anmerkung
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‹

Anmerkung
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‹

5 Optionen, konditionales Regex, Assertion

Hattet ihr die Befürchtung, dass das folgende Kapitel etwas schwieriger als der vorangegangene 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.

5.1 Assertion

?= Lookahead *nach vorne schauend*
?! negative Lookahead
?<= Lookbehind *zurück schauend*
?<! negative Lookbehind

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

›(?<=\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‹.

5.2 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 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‹. Ganz schön verwirrend, oder?

5.3 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 konditionales 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‹ gefolgt vom englischen Format oder ›Datum‹ gefolgt vom deutschen Format dort 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})$)‹

Anmerkung
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.

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‹

5.4 Optionen, Modifikatoren

i Caseless
m Multiline
s DotAll
x Extended
A Anchored
D Dollar_Endonly
U Ungreedy
X Extra
e Evaluate
S Speed

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. Betrachten wir also folgende Auswahl:

i 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 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 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 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 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 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 sei ›(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** 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 hier aufgeführten 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.

Das Setzen der Modifikatoren kann bei den Programmiersprachen unterschiedlich erfolgen. Lest bitte die entsprechenden Infos dort.

5.5 Besonderes

Kommen wir zu ein paar Besonderheiten, die uns nur selten begegnen werden. 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

**** Maskieren
^ Negieren der Charakterklasse, wenn es das erste Zeichen
- Kennzeichnen eines Bereichs
] Beenden der Charakterklasse

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.

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. Alles Klarheiten beseitigt? Ok, dann doch besser mit ein paar Beispielen:

- \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. 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.

5.6 Überblick über diesen Abschnitt

Dieser Abschnitt hat uns ein Besonderheiten der Regexe nähergebracht. Wir werden damit genügend über Regexe wissen, um sie in PHP, Perl oder sonstwo einsetzen zu können.

Zusammenfassung

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. Diese dürfen nicht durch Quantifizierer in ihrer Länge unberechenbar werden. Man unterscheidet
positive Lookahead-Assertion (?=
negative Lookahead-Assertion (?!
positive Lookbehind-Assterion (?<=
negative Lookbehind-Assertion (?<!
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 Caseless
m Multiline
s DotAll
x Extended

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 im Regelfall für uns relevant sein.

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, also einem Regex, das den eigentlichen Betreff aus der Betreffzeile herausoperiert. Beispiel für ein solches Betreff ist zum Beispiel: ›Re [2]: Regenechsen in freier Wildbahn (was: Irgendein anderer Thread)‹. Das ›Re‹ kann von allem möglichen und einem Doppelpunkt gefolgt werden. Dann kommt der eigentliche Betreff ›Regenechsen in freier Wildbahn‹ 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.

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. 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, wenn noch irgendwo ein Doppelpunkt dort stünde. 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?

6 Schlussbemerkung

Ein sehr langer und umfangreicher Kurs. Dabei erklärt er noch nicht mal den Gebrauch von Regexen in den einzelnen Sprachen, werdet ihr denken. Stimmt! Das war auch gar nicht meine Absicht! Vielmehr sollte dieser Workshop eine Schritt-für-Schritt-Anleitung für den Neuling in Regexisch sein. Hat man erst mal verstanden, wie so ein Regex funktioniert, dann ist die spätere Verwendung in PHP, Perl, Javascript oder wo auch immer nur noch eine Frage der sprachspezifischen Syntax. Und dafür gibt es schon Workshops und wird es bestimmt auch weitere geben. Ich hoffe, ich habe euch die Regenechsen näher gebracht. Es ist eine faszinierende Sprache! So faszinierend, dass sie von Programmiersprachen gern als Werkzeug verwendet wird.

Autor: Gerd Ewald
www.regenechsen.de/phpwcms/index.php?regex_allg