Reguläre Ausdrücke

Tutorial Reguläre Ausdrücke

"Reguläre Ausdrücke" sind eine Art Sprache, die beim Programmieren für diverse Problemlösungen verwendet werden kann, insbesondere dann, wenn es darum geht, Zeichenketten (Strings) zu bearbeiten, zu prüfen oder in ihnen etwas zu suchen.

Und weil der Name "Reguläre Ausdrücke" etwas unhandlich ist, heißen die "Regular Expressions" auch oft einfach nur "RegEx(en)".

Du kannst dir dieses Tutorial ausdrucken - es gibt eine Druckversion ohne den schwarzen Hintegrund, wenn du in deinem Browser auf die Drucken-Funktion gehst.

Und für Fehler bitte nicht gleich schlagen, der Text ist zu 100% im Regionalexpress entstanden :-)

Einführung

Hier gibts ein kleines Tutorial zu diesen esoterisch anmutenden aber unglaublich mächtigen Zeichenketten, die beim unbedarften Betrachter Assoziationen eines kleinen Kindes und dessen ersten Versuchen an der Tastatur auslösen.

Was sind reguläre Ausdrücke? Wie gesagt, können mit regulären Ausdrücken Zeichenketten auf eine bestimmte Zusammensetzung geprüft werden, wie dies z.B. bei Anwendungen wichtig ist, die Eingaben eines Benutzers erwarten.

Beispiele:

Schulnote

Postleitzahl

E-Mail-Adresse

Bestellnummern

Konkrete Anwendung

Populäre Stellen, an denen man auf reguläre Ausdrücke stoßen kann und an denen du dieses Wissen anbringen kannst:

Web-Anwendungen (z.B. in PHP, Perl)

Unix-Skripte

Im Folgenden wird davon ausgegangen, dass du eine entsprechende Stelle vorliegen hast und dich bereits informiert hast, wie reguläre Ausdrücke in der Umgebung deiner Wahl verwendet werden. In

PHP geschieht dies z.B. mit den Funktionen preg_match (Prüfen von Strings, finden von Zeichenketten) und preg_replace (Ersetzen von Zeichenketten). Am Besten testest du deinen Ausdruck zuerst "auf dem Trockenen". Dazu kannst du den Regex-Coach benutzen.

Weitere Quellen für Informationen findest du in meinen Links.

Guter Stil

Oft wirst du feststellen, dass es zu einem Problem viele verschiedene Lösungen gibt. Dabei stehen sich dann gegenüber

Eine genaue Lösung und eine allgemeinere. Besteht die Gefahr, dass die genaue Lösung zu "restriktiv" ist (und damit vielleicht ein Benutzer frustriert aufgibt, wenn seine richtige Eingabe nicht akzeptiert wird), nimm lieber die allgemeine. Ist das Problem aber sehr "eindeutig", nimm lieber die genaue.

Eine genaue Lösung und eine schnelle. Lange reguläre Ausdrücke können viel Zeit für die Verarbeitung in Anspruch nehmen. Auch hier musst du abwägen, was dir wichtiger ist. Es könnte Benutzer ärgern, wenn die Prüfung einer Eingabe sehr lange dauert.

Eine einfache Lösung und eine "elegante". Du kannst in 10 Zeichen ausdrücken, was andere nicht in 60 schaffen? Prima, aber denk dran dass du vielleicht später nochmal was ändern willst! Und das kann schwerer werden, als man denkt. Daher nimm lieber die einfachere Variante oder kommentiere komplizierte Ausdrücke.

Probleme

Wenn dein regulärer Ausdruck Probleme macht, du dir aber ziemlich sicher bist dass er richtig ist, schau dir zunächst mal weiter unten die Sonderzeichen an und was man mit ihnen machen muss, um sie zu benutzen. Dabei sind Sonderzeichen schon so "banale" Sachen wie ein Punkt.

Außerdem solltest du ihn dringend im o.g. RegEx-Coach testen! Dieser fördert manchmal geradezu unglaubliche irreguläre Verhaltensweisen zutage ;-)

Dann darfst du als nächstes Google benutzen ;-) Da findet man sehr viele Informationen, viele Vorlagen um diverse Problemstellungen regulär auszudrücken und auch die Google-Groups sind immer hilfreich.

Konventionen

In diesem Tutorial werden folgende Farben für Texte (Strings) verwendet:

Regulärer Ausdruck, den du theoretisch so verwenden kannst, mit

markierten neuen Teilen

Absichtlich fehlerhafter Code

Öfters verwende ich mal den Programmierern nicht fremden "String" (und nein, ich spar mir jetzt den Unterwäsche-Witz) für ein Textstück. Außerdem gilt: ein Ausdruck "trifft" = "beschreibt einen Text genau" = "erledigt die gewünschte Funktion" = "matcht" = "frisst (etwas)"

Einfache Ausdrücke

Ein-Elementige reguläre Ausdrücke

Fangen wir mal klein an. Wir wollen prüfen, ob eine Eingabe einer Schulnote von 1-6 entspricht.

[123456]

erledigt das für uns. Du siehst: in eckigen Klammern folgt eine Auflistung von Zeichen, die erlaubt sind. Insgesamt steht der gesamte geklammerte Ausdruck aber nur für ein Zeichen: 1 oder 2 oder ... oder 6.

Da unsere Zahlen von 1 bis 6 so schön aufeinanderfolgen, können wir auch schreiben

[1-6]

was die Sache etwas übersichtlicher macht. Genau so könnte man z.b. prüfen, ob eine Eingabe einem Gleis auf einem Bahnhof entspricht:

[1-9]

für einen Bahnhof mit 9 Gleisen. Auf unserem Bahnhof sei heute Gleis 4 gesperrt, also als Eingabe nicht erlaubt:

[1-35-9]

Wir haben den Eingabebereich also aufgeteilt in zwei Bereiche 1-3 und 5-9. Du siehst: Die beiden Bereiche werden einfach hintereinander geschrieben. Das ist am Anfang gewöhnungsbedürftig (intuitiv möchte man vielleicht ein Leerzeichen machen), aber das wird uns noch öfter begegnen.

Mehrelementige reguläre Ausdrücke

Was ist, wenn auf unserem Bahnhof jetzt angebaut wird? Sagen wir, er wird auf 12 Gleise erweitert:

[1-12]

(Gleis 4 sei wieder geöffnet ;-)). Aber Vorsicht, hier ist ein Fehler drin!

Wie oben erwähnt, steht der gesamte Ausdruck in den eckigen Klammern nur für ein Zeichen. "12" enthält aber zwei Zeichen. Der Code oben wird nicht funktionieren: Er bedeutet "1 bis 1" oder "2". Den Umgang mit Bahnhöfen mit mehr als 9 Gleisen demonstrieren wir später.

Für den Moment wollen wir etwas anderes Prüfen: Bahnsteige 1-9 haben noch jeweils Abschnitt "a" und "b". Wir wollen also auf 1a,1b,2a,...,9a,9b prüfen:

[1-9][ab]

Will heißen: Eine Zahl 1-9 gefolgt von einem Buchstaben a oder b.

Mehrere rechteckige Klammern entsprechen also mehreren Zeichen. Soll ein Ausdruck mehrere Zeichen beschreiben, werden diese einfach hintereinander gehangen. Dann wird die Eingabe von links nach rechts mit deinem Ausdruck verglichen.

Selbstverständlich müssen auch nicht immer alle Buchstaben aufgeführt werden:

[1-9][a-d]

trifft jetzt auch z.B. 4d.

Zu beachten ist, dass Groß- und Kleinschreibung getrennt behandelt wird. Eine sinnvolle Erweiterung wäre also z.B.

[1-9][a-dA-D]

Optionen

Schauen wir uns die Eingabe einer Hausnummer an. Diese können aus einer oder mehreren (bis zu 3) Ziffern bestehen und ein a-z am Ende haben. Müssen aber nicht.

[1-9][0-9]?[0-9]?[a-z]?

Das Fragezeichen hinter einem Element (und [1-9] ist ein Element!) besagt: Das vorhergehende Element kann vorkommen, muss aber nicht.

Der letzte Code heißt also: Eine Ziffer (1-9), optional zwei weitere Ziffern (0-9), optional ein Buchstabe.

Stellt man sich solch eine Konstruktion mit z.B. 10 Ziffern vor, kann man sich leicht ausmalen, dass sie schnell sehr lang wird. Daher gibt es eine andere Schreibweise, wenn man ein Element mehr als ein mal erlauben will:

a{1,3}h

Trifft auf "ah" genau so wie auf "aaah". Die Angabe in Klammern steht also für die {minimale, maximale} Anzahl an Zeichen. Daher können wir unsere Hausnummern auch so formulieren:

[1-9][0-9]{0,2}[a-z]?

und dann auch ohne Probleme für längere Straßen (USA ;-)) umschreiben:

[1-9][0-9]{1,4}[a-z]?

erlaubt jetzt auch Hausnummern im fünfstelligen Bereich, fordert aber mindestens eine zweistellige (beachte die Änderung vor dem Komma).

Eine Konstruktion wie

[0-9]{5}

(also eine Klammer mit nur einer Angabe) verlangt eine genau x-malige Wiederholung des davorstehenden Ausdrucks. In unserem Fall 5 mal: Dieser reguläre Ausdruck wäre geeignet, um (deutsche) Postleitzahlen zu prüfen.

Wir könne auch ein "mindestens" ausdrücken:

[0-9]{3,}

fordert mindestens 3 Ziffern.

Beliebige Wiederholungen

Bisher mussten wir immer wissen, wieviele Zeichen gefunden werden sollen. Es gibt aber durchaus Fälle, in denen sich ein Zeichen beliebig oft wiederholen darf.

Schauen wir uns die Prüfung von Telefonnummern an. Wenn man einen regulären Ausdruck schreibt, empfiehlt es sich oft, zunächst einmal zu notieren, auf was dieser alles treffen soll. Unsere Telefonnummern sollen von diesem Format sein dürfen:

0651/55541-36

0049 160 555678

0180.23.555.63

Es können neben Zahlen also auch Bindestriche, Querstriche, Leerzeichen und Punkte vorkommen. Ein gültiges Element wäre also

[0-9/. -]

(Also: Dieses eine Zeichen besteht aus einer Zahl zwischen 0 und 9 oder einem Slash oder einem Punkt oder einem Leerzeichen oder einem Minus.) Doch Vorsicht! Wie du siehst, kommen in den Klammern zwei Bindestriche vor: ein mal in einer speziellen Funktion, um einen Zahlenbereich anzudeuten und ein mal als "echter" Bindestrich, der so auftauchen darf. Um Verwechslungen auszuschließen müssen wir letzteren maskieren, also zeigen dass er hier keine spezielle Funktion hat. Das geht mit einem vorangestellten Backslash:

[0-9/. \-]

Und dann wollen wir ja noch mitteilen, dass diese Zeichen beliebig oft vorkommen dürfen:

[0-9/. \-]+

Das Plus bedeutet: ein oder mehrmals das davor befindliche Zeichen, also mindestens ein mal. Wollen wir auch eine leere Telefonnummer zulassen (der Mathematiker würde sagen: Die Nullnummer), so gibt es dazu ein anderes Zeichen:

[0-9/. \-]*

trifft jetzt auch auf "" (leere Zeichenkette) und natürlich unsere Telefonnummern.

Dieser Ausdruck ist jetzt natürlich ein sehr allgemeiner Ausdruck (siehe die Diskussion oben). Er erlaubt auch Telefonnummern, die offensichtlich nicht korrekt sind, wie

--..//123

Möchte man das weiter einschränken, muss man einen komplizierteren Ausdruck entwickeln.

Also nochmal zusammengefasst: Das + steht für eine mindestens einmalige Wiederholung. * heißt: keinmal oder beliebig oft.

Platzhalter

Gehen wir zu einem anderen Beispiel über: Wir möchten in einer Bibliothek nach einem Autor suchen, wissen aber nicht, ob er evtl. einen zweiten Vornamen hat. Nach dem Vornamen können also beliebige Zeichen kommen und danach dann erst der Nachname. Eine mögliche Lösung sieht so aus:

Marius .*Osterhase

Die im Einzelnen bedeutet: "Marius", danach ein Leerzeichen gefolgt von einem beliebigen Zeichen (dafür steht der Punkt!) beliebig oft (also evtl. auch gar keins) und anschließend der Nachname. Das trifft auf "Marius Osterhase" genau so wie auf "Marius Müller Osterhase".

Genau so kann man dem Punkt natürlich auch ein Plus folgen lassen, um mindestens ein beliebiges Zeichen zu fordern.

Der Punkt "frisst" normalerweise zwar fast alles, aber keine Zeilenumbrüche. Wie du ihn dazu bringst, findest du unter Modifikatoren.

Dem "aufmerksamen Leser" wird nicht entgangen sein, dass wir oben auch schon einen Punkt hatten, und zwar innerhalb der eckigen Klammern. Wie bei vielen, aber nicht allen Sonderzeichen müssen diese zwar außerhalb von eckigen Klammern als solche maskiert werden (Backslash davor), aber nicht innerhalb.

Zeichenklassen negieren

Angenommen wir wüssten den zweiten Vornamen des Autors nicht genau, aber können uns daran erinnern, dass er kein q und kein z enthält. Auch kein Problem:

Marius [^qz]+ Osterhase

verlangt nach einem zweiten Vornamen (deswegen das Plus) und lässt dazu ein beliebiges Zeichen zu, das nicht q oder z ist. Das ^ negiert also eine Zeichenklasse und gilt genau bis zur schließenden Klammer.

Klammern

Klammern können dazu benutzt werden, längere Ausdrücke zu einem Element zusammenzufassen und es damit zu ermöglichen, das oben gelernte auch auf Teilausdrücke anzuwenden:

Marius (Müller)?Osterhase

trifft genau auf "Marius Müller Osterhase" oder "Marius Osterhase" und auf nichts anderes: Das Fragezeichen bezieht sich - dank der Klammern - auf den kompletten zweiten Vornamen und das folgende Leerzeichen.

Ebenso lässt sich schreiben:

Ba(na)*ne

Was jetzt sowohl für dieses gelbe Ding als auch z.B. für "Bananananane" steht. Auch hier lässt sich die Notation mit den geschweiften Klammern anwenden:

Ba(na){2,5}ne

was jetzt natürlich eine entsprechend andere Bedeutung hat.

Alternativen

Mit Klammern kann man aber auch noch andere Sachen machen, z.B. Alternativen zu einem Teilausdruck angeben:

Das Wetter ist (toll|richtig schlecht)

In diesem Beispiel dürfen als letzte Worte "toll" oder "richtig schlecht" vorkommen, aber nicht beide.

Modifikatoren

In allen RegEx-Varianten kannst du sogenannte Modifikatoren setzen und damit das genaue Verhalten des Ausdrucks kontrollieren. In Java kannst du dies z.B. bei der Konstruktion eines matcher-Objektes erledigen, bei PHP hat ein regulärer Ausdruck immer die Syntax

[Begrenzungszeichen][RegEx][Begrenzungszeichen][Modifikator(en)]

also z.B.

/(Mein|Ausdruck)/im

Dabei sind die Slashes die Begrenzungszeichen (andere sind hier denkbar, z.B. ~), und "i" und "m" in diesem Fall Modifikatoren. Gängige Modifikatoren sind unter anderem:

i Case-Insensitivity (die Nichtbeachtung von Groß- und Kleinschreibung) einschalten

s Punkt wird multilinefähig: Der Punkt frisst auch Zeilenumbrüche, dies ist standardmäßig nicht so.

m Zeilenmodus: Die Zeichen ^ und $ matchen auch auf Zeilenanfänge bzw. -enden. Ohne den Modifikator passen sie nur auf Anfang und Ende der gesamten Zeichenkette.

Modifikatoren beziehen sich immer auf den ganzen Ausdruck und sind daher eine leicht übersehene Fehlerquelle.

Kompliziertere Ausdrücke

Verschachtelungen

Selbstverständlich dürfen auch diverse Klammern ineinander geschachtelt werden, wie z.B. in folgendem Ausdruck:

(VW (Golf|Polo)|Fiat (Punto|Panda))

welcher auf "VW Golf", "VW Polo", "Fiat Punto" und "Fiat Panda" trifft. Dies ermöglich zwar recht kurze Ausdrücke für lange Zeichenketten, kann aber auch viel Zeit zur Verarbeitung in Anspruch nehmen.

Genau so lassen sich auch diese Alternativen wiederholen:

(10|01)+

Beschreibt eine Folge aus Nullen und Einsen, in der maximal 2 Nullen oder Einsen aufeinander folgen. (Falls das jetzt nicht direkt klar wird: einfach mal überlegen, was man aus "10" und "01" zusammensetzen kann.)

Gierige Ausdrücke

Ein Beispiel aus der Praxis: Der folgende, etwas längerer Code soll uns aus einer HTML-Datei die Links bzw. deren Zieladressen heraussuchen. Ein Link in einem HTML-Quellcode hat i.d.R. ein Format wie dieses:

<a href="[...Zieladresse...]" [...weitere Angaben...]>

z.B.

<a href="http://www.example.org/" target="_blank">

Ein möglicher Ausdruck dafür ist recht schnell gefunden:

<a href=".*".*>

Sieht gut aus, klappt aber nicht. Warum?

Der Autor hat sich zwar gedacht der Ausdruck würde an der zu dem Link gehörenden schließenden Klammer aufhören, er tut es aber nicht. Im folgenden ein Ausschnitt aus einer HTML-Datei mit markiertem Treffer:

<BODY> Bla Blubb 1 2 3 Linktext Viel, viel weiterer Text Blubb 42

Das geht eindeutig zu weit! Der zweite Punkt ist zu "gierig" und frisst alle Zeichen sogar über mehrere schließende spitze Klammern hinweg. In anderen Situationen wäre es denkbar, dass der erste Punkt ähnlich amokläuft.

Für unseren Fall gibt es zwei Lösungsmöglichkeiten. Oft bleibt jedoch nur eine davon, daher stelle ich beide vor:

Den Punkt ersetzen durch etwas was keine spitzen Klammern mehr frisst:

<a href=".*"[^>]*>

Das ist eine oft anzuwendende Methode (überlege dir, welche Zeichen das Ende markieren und schließe diese von dem zu treffenden aus). Das ganze muss jetzt natürlich noch analog für den ersten Punkt gemacht werden:

<a href="[^"]*"[^>]*>

Den Punkt "umerziehen" (bzw. * genügsam machen) so dass beide zusammen nur noch so viel fressen, wie unbedingt nötig ist. Dies tut ein angehängtes Fragezeichen, was dann natürlich nicht mehr die bisher bekannte Funktion hat:

<a href=".*?".*?>

Natürlich funktioniert diese Sonderbenutzung des Fragezeichens auch hinter einem Pluszeichen.

Gruppen

So gut wie alle RegEx-Dialekte erlauben das Bilden von Gruppen und deren Speicherung für spätere Verwendung. Auch dazu können Klammern dienen.

Spätere Verwendung kann z.B. heißen, dass ein Ausdruck zwar auf eine längere Zeichenkette treffen soll, aber nur ein Teil davon wirklich verwendet werden soll.

Soetwas wäre in Verbindung mit dem obigen Ausdruck sehr nützlich, um alle Zieladressen aus einer HTML-Seite zu filtern.

Wie genau du an die Inhalte der Gruppen kommst, steht in der Anleitung der von dir verwendeten Sprache. Bei PHP findest du es z.B. bei preg_match und Java soweit ich mich erinnern kann irgendwo in der Nähe von regex.matcher. Google sollte da mehr zu wissen.

<a href="(.*?)".*?>

Der erste Punkt und seine "Vervielfacher" sind eingeklammert. In der ersten Gruppe befindet sich jetzt also die URL.

Beachte, dass bei der Nummerierung der Gruppen die Reihenfolge der öffnenden Klammer zählt. Das ist wichtig zu beachten bei verschachtelten Klammern. Außerdem zählen normalerweise alle Klammern, auch wenn sie nur zur Ausweisung von Alternativen (s.o.) verwendet werden.

Referenzen

Es gibt noch einen anderen, sehr nützlichen Verwendungszweck für Gruppen. Stell dir vor, du willst in einer Folge von Zahlen (sagen wir sie seien mit einem Leerzeichen voneinander getrennt) all die Zahlen finden, die mit der selben Ziffer beginnen und enden.

Wir haben also

129 337873 78324 43938 9388 824998 349734

Dazu können wir solch einen Ausdruck verwenden:

([0-9])[0-9]*\1

wobei das \1 auf den Inhalt der ersten Klammer verweist (referenziert) und daher an der Stelle von \1 das selbe stehen muss wie in dieser Klammer. Das erste und letzte Zeichen unseres Ausdrucks ist jeweils ein Leerzeichen, um damit das Ende und den Anfang einer Zahl zu kennzeichnen.

Schauen wir uns an, auf was er trifft:

129 337873 78324 43938 9388 824998 349734

Wie du siehst, zählt das erste und letzte Leerzeichen auch immer noch zu dem jeweiligen Treffer dazu. Das ist auch kein Wunder, schließlich steht vor und hinter unserem Ausdruck auch jeweils ein Leerzeichen. Das ist nicht wirklich gut, wenn uns dieses Leerzeichen eigentlich gar nicht interessiert. Daher stellt die RegEx-Sprache auch hier ein Mittel zur Verfügung, welches auch funktioniert, wenn wir keine Gruppen benutzen möchten oder können:

Spezielle Zeichen: Wortgrenzen

Es gibt Zeichen, die stehen zwar im regulären Ausdruck, aber nicht in dem Text, der nachher gematcht wird. Das sagt dir jetzt nichts? Naja, schauen wir uns als ein Beispiel an, wie man obigen Ausdruck ohne die Leerzeichen schreibt:

\b([0-9])[0-9]*\1\b

Dieses "\b" ist ein zusammengehöriges Element und kennzeichnet ein Wortanfang oder ein Wortende (also eine Wortgrenze). Kennst du die in Textverarbeitungsprogrammen und Editoren oft verwendete Möglichkeit "Nur ganzes Wort suchen" in der Suchfunktion? Wählst du diese aus, werden z.B. bei einer Suche nach "Kai" statt diesen Treffern

Kai fährt nach

Kaiserslautern

nur noch dieser Treffer gefunden:

Kai fährt nach

Kaiserslautern

Letzteres ließe sich in regulären Ausdrücken so umsetzen bzw. ausdrücken:

\bKai\b

Also "Kai" nur, wenn es von Wortgrenzen umgeben ist. Was genau ist überhaupt eine Wortgrenze? Eine Wortgrenze tritt zwischen einem Wort-Zeichen und einem Nichtwortzeichen auf. Hä? (Erklärung folgt!)

Spezielle Zeichen: Weitere Zeichen

Es gibt - Überraschung! - noch mehr Sonderzeichen, die du verwenden kannst und die eine RegEx schön abkürzen können. Diese bestehen immer aus einem Backslash gefolgt von einem weiteren Zeichen (Buchstaben). Übrigens steht ein großer Buchstabe dabei immer für das Gegenteil eines kleinen.

\w \W

Ein Wort-Zeichen (kleines w) steht genau für [a-zA-Z0-9] und ein Nichtwortzeichen (großes W) steht genau für alles andere, also [^a-zA-Z0-9].

\d \D

Ein Digit, also eine Ziffer von 0-9. Entspricht damit [0-9] und in der Großschreibung [^0-9].

\b \B

Die "kleine" Variante hast du ja oben schon kennengelernt. Ein großes B steht dementsprechend für alle Stellen, an denen keine Wortgrenze auftritt.

\s \S

Die Kleine Variante steht für alle Whitespaces: Das sind so gut wie alle Zeichen, die man nicht sieht. Also Return (bzw. Enter), Leertaste (Space), Tab(ulator).

\\

Wenn man mit Backslash, wie du gesehen hast, Sonderzeichen anfängt, muss man ja auch den Backslash selbst irgendwie fabrizieren können, wenn man genau diesen meint. Dies macht man einfach durch einen doppelten solchen. Man sagt: der erste "maskiert" den zweiten.

\. \+ * \(\) \[\] \- \$ \|

Der Punkt und viele andere Zeichen haben, wie du oben gesehen hast, eine Sonderfunktion. Daher werden sie mittels eines Backslashs von dieser Sonderfunktion abgehalten ("maskiert"). Dies gilt für das Minuszeichen nur innerhalb von Zeichenklassen, dafür darfst du in selbigen bei vielen anderen Zeichen die Backslashs weglassen.

Anfangs- und Endezeichen

Stell dir vor, du willst ein Datum prüfen, sagen wir mit

1\.3\.2004

Und diesen Ausdruck lässt du auf

11.3.2004

los. Was kommt raus? Dein Ausdruck wird dir melden, dass er trifft. Ist ja auch irgendwo klar, schließlich steht, wenn man eine Eins weglässt, dein Datum da. Du willst aber, dass das gesamte Datum wie deins aussieht? Gibt\'s da nicht was von Ratiodingsbums? Doch gibts:

^1\.3\.2004$

Dieses Dach am Anfang sagt: Treffe nur, wenn hier der zu durchsuchende String anfängt. Und das Dollarzeichen steht für: Treffe nur, wenn genau hier der zu durchsuchende String endet.

Selbstverständlich kann man beide auch einzeln einsetzen; Als Beispiel schauen wir uns zwei reguläre Ausdrücke an: Der erste trifft auf einen String, der mit einer Zahl endet. Der zweite trifft auf einen String, der mit einer öffnenden Klammer beginnt. Dabei benutzen wir die oben angegebene Maskierung für die Klammer.

\d$

^\(

Positive Lookaheads und Lookbehinds

Du hast jetzt Möglichkeiten kennengelernt, wie du Wortgrenzen feststellen kannst, ohne sie selbst wirklich mitzufressen. Es gibt auch eine universelle Möglichkeit.

Als Beispiel soll folgende Zahlenfolge dienen:

000566403580000345050052301078906040092800100001007680

Aus dieser Zahlenfolge sollen alle jene Folgen herausgefischt werden, die drei Ziffern ungleich Null enthalten und von einer Null an jeder Seite begrenzt werden. In unserem Fall also "358", "345", "523", "789", "928" und "768". Dazu kannst du folgenden Code benutzen:

(?<=0)\d{3}(?=0)

Keine Panik! Zur Erklärung, und wir fangen in der Mitte an: das \d{3} sollte klar sein (3 Ziffern). Davor findest du die Klammer (?<=0). Die Zeichen ?<= musst du "en bloc" betrachten und sie kennzeichnen eine Sonderfunktion für die Klammer. Diese steht damit für "vornedran eine Null". Und weil "vornedran" in der normalen Leserichtung der westlichen Welt bedeutet, dass die Klammer "rückwärts" schaut, heißt diese Funktion "positive lookbehind".

Genauso mit den Zeichen "?=", diese weisen der Klammer die Sonderfunktion "hintendran muss ... kommen" zu, in unserem Fall also "hintendran eine Null". Das ganze nennt sich "positive lookahead", weil die Klammer "nach vorne" schaut.

Unser "\b"-Beispiel von oben könnten wir also auch folgendermaßen ausdrücken, wenn wir unser Wissen über "\W" benutzen:

(?<=\W)(\d)\d*\1(?=\W)

Zur Erklärung fangen wir wieder in der Mitte an: Da steht der von oben bekannte Code (Eine Ziffer, beliebig viele weitere Ziffern und eine weitere Ziffer, die die gleiche wie die erste ist). Die allererste Klammer fordert vor der ersten Ziffer ein Nichtwortzeichen und die allerletzte Klammer ein Nichtwortzeichen hinter der letzten Ziffer.

Selbstverständlich können lookaheads und lookbehinds auch einzeln in einem regulären Ausdruck vorkommen und sogar mehrfach. Das einzige, was nicht geht, ist ein lookbehind mit einer unbekannten Anzahl von Zeichen. (also so Dingern wie *, + und {0,4})

Alles klar? Wenn nicht, dann zurück zum Anfang dieses Abschnitts, gehe nicht über "Los" und ziehe keine 200 RegExen ein.

Negative Lookaheads und Lookbehinds

Du hast das kompliziert gefunden und siehst so langsam ein, wie voodoo reguläre Ausdrücke sind? Es kommt noch besser.

Dazu noch ein Beispiel: Du willst eine E-Mail-Adresse prüfen, die aber nicht aus Frankreich stammen soll. Zunächst schauen wir uns eine einfache E-Mail-Adressen-Prüfung ohne die Einschränkung an:

[^@]+@.+\.[^.]+

(Ein Zeichen ungleich dem @-Zeichen, gefolgt von einem @-Zeichen, beliebigen Zeichen (aber mindestens eins), einem Punkt und dahinter beliebige Zeichen, aber kein Punkt mehr. Trifft auf toll@example.com genau so wie auf viele_.Zeichen@noch.mehr.Punkte.example.com.) Wichtig: das letzte Paar von eckigen Klammern frisst immer den letzten Teil (die Top-Level-Domain, im Beispiel "com") der E-Mail-Adresse.

Jetzt fügen wir eine Beschränkung ein und wollen "fr" an dieser Stelle ausschließen. Auf den Punkt darf also nicht "fr" folgen.

[^@]+@.+\.(?!fr)[^.]+

erledigt das für uns. ?! kennzeichnet eine Klammer also als "hier darf nicht folgen".

Damit könnte man auch eine besondere Suche nach einem Wort gestalten.

\bF(?!eta\b).*\b

trifft auf alle Wörter, die mit "F" anfangen, aber ungleich "Feta" sind. Im folgenden Text sind alle Strings markiert, die getroffen werden:

Fett Feta Falsch Feucht Fuffziger

Das war der "negative lookahead".

Und weils so schön war, leg ich noch einen drauf, den "negative lookbehind":

.*(?<!Müll)eimer

trifft auf alle Eimer, in die kein Müll gehört, meint also: "treffe an dieser Stelle, wenn davor kein "Müll" steht".