D2 Reguläre Ausdrücke

Reguläre Ausdrücke (Abk. ***RegExp*** oder ***Regex***, engl. *regular expression*) dienen der Beschreibung einer Familie von formalen Sprachen, d. h. sie beschreiben (Unter-)Mengen von Zeichenketten. Sie gehören somit zur Theoretischen Informatik. Hier bilden sie die unterste und somit ausdrucksschwächste Stufe der Chomsky-Hierarchie (Typ-3). Es lässt sich zeigen, dass zu jedem regulären Ausdruck ein gleichwertiger endlicher Automat existiert und umgekehrt. Dieser Automat ist einfach bestimmbar. Hieraus folgt die relativ einfache Implementierbarkeit regulärer Ausdrücke.

Der Mathematiker Stephen Kleene benutzte eine Notation, die er *reguläre Mengen* nannte.

Reguläre Ausdrücke in der theoretischen Informatik

Theoretische Grundlagen

Reguläre Ausdrücke unterstützen genau drei Operationen: Alternative, Aneinanderreihung und Wiederholung. Die formelle Definition sieht folgendermaßen aus:

Syntax

1. $\underline{\varnothing}$ (die leere Menge) ist ein regulärer Ausdruck.
2. $\underline{\epsilon}$ (das leere Zeichen) ist ein regulärer Ausdruck.
3. $\forall a_i \in \Sigma$ ist $\underline{a_i}$ (jedes Zeichen aus dem zugrundeliegenden Alphabet) ein regulärer Ausdruck.
4. Sind *x* und *y* reguläre Ausdrücke so auch $(x \cup y)$ (Vereinigung), (*xy*) (Konkatenation) und x^* (Stern-Operator).
5. Es gibt keine weiteren regulären Ausdrücke.

Einleitung

Reguläre Ausdrücke sind eine Art Mini-Programme. Sie wurden entwickelt, um das Arbeiten mit Texten komfortabler und leichter zu gestalten. Reguläre Ausdrücke sind vergleichbar mit "Mustern" oder "Schablonen", die auf einen oder mehrere unterschiedliche Strings passen können. Diese Muster können entweder auf einen String passen oder nicht passen.

Hierbei existieren zwei Kategorien von Regulären Ausdrücken. Einmal eine normale Mustersuche, zum anderen eine "Suche und Ersetze"-Funktion. Die normale Mustersuche wird zum einen darin verwendet, um ganze Eingaben oder Strings zu überprüfen oder einzelne Informationen aus einen String auszulesen. Die Suchen- und-Ersetzen-Funktion hat dabei eine ähnliche Funktion, wie Sie es von grafischen Texteditoren gewohnt sind, nur sind diese in Perl deutlich mächtiger.

Die englischen Begriffe für die Muster- und "Suche & Ersetze"-Funktion sind dabei: "Pattern matching" und "Substitution". Diese Begriffe sollte man kennen, da sie oft benutzt werden. Diese werden hier ebenfalls im weiteren Verlauf Verwendung finden.

Der Aufbau der beiden genannten Typen sieht dabei folgendermaßen aus:

Pattern Matching:

```
m/Regex/Optionen
```

Substitution:

```
s/Regex/String/Optionen
```

Begrenzer

Wie man am Pattern Matching sowie der Substitution erkennen kann, trennen die Slashes "/" die einzelnen Teile eines Regulären Ausdruckes. Allerdings haben Sie bei Perl die Wahl, jedes beliebige Sonderzeichen als Begrenzer zu wählen. Die Vorteile davon werden Sie zu schätzen wissen, wenn Sie praktische Erfahrung mit Regulären Ausdrücken sammeln. Um es kurz vorweg zu sagen: Sie haben damit die Wahl ein Zeichen zu wählen das nicht in Ihrer Regex vorkommt, und Sie können sich somit das Escapen der Zeichen sparen.

Perl weiß anhand des ersten Zeichen, das nach dem "m" respektive "s" folgt, welcher Begrenzer gewählt wurde. Es sind also auch folgende Schreibweisen erlaubt:

```
m!regex!
s#Regex#String#Optionen
s$Regex$String$Optionen
s"Regex"String"Optionen
...
```

Eine spezielle Regel gilt für Zeichen, die ein öffnendes sowie schließendes Zeichen besitzen. Den dort müssen die einzelnen Teile eingeklammert werden. Dies schaut folgendermaßen aus:

```
m(Regex)
s(Regex)(String)Optionen
s{Regex}{String}Optionen
s<Regex><String>Optionen
s[Regex][String]Optionen
```

Wie man sieht werden hier unterschiedliche Anfangs- sowie Endzeichen verwendet. Eine weitere Eigenschaft ist, dass nur das wirkliche Endzeichen die Regex respektive den String einklammert. Im folgenden Praktischen Beispiel ist also auch folgendes möglich:

```
s((H)allo)(Welt)i
```

Wenn Sie etwas Erfahrung mit Regulären Ausdrücken haben, werden Sie sehen, dass diese Regex keinen Sinn ergibt. Allerings dient das lediglich zur Veranschaulichung, dass hier wirklich "(H)allo" als Regex erkannt wird, und nicht "(H" wie man annehmen könnte.

Bindungsoperator

Um einen String mit einer Regex zu verbinden, egal ob nun "Pattern Matching" oder "Substitution", schreibt man einfach folgendes:

```
$text =~ m/Regex/;
$text =~ s/Regex/String/;
```

Dieser Bindungsoperator prüft im ersten Fall, ob in $text das angegebene Muster auf der rechten Seite vorkommt. Zur Substitution kommen wir noch später; allerdings sei hierzu gesagt, dass jedes Vorkommen der angegebenen Regex in $text durch String ersetzt wird.

Weiterhin besitzt dieser Ausdruck einen Rückgabewert. Wenn das Muster, das in Regex angegeben wird, wirklich in $text passt, dann wird "true" zurück gegeben. Das gleiche gilt für die Substitution. Wenn die Regex auf $text gepasst hat, wurde eine Ersetzung durchgeführt und es wird der Wert "true" zurück gegeben.

Sollte die Regex in beiden Fällen nicht auf den String in $text gepasst haben, dann wird "false" als Wert zurück geliefert. Bei der Substitution hat dies noch die Auswirkung, dass eine Ersetzung nicht stattgefunden hat.

Grundlegendes zu Regulären Ausdrücken

Pattern Matching

Elemente, mit denen sich ein regulärer Ausdruck festlegen lässt

Die folgenden Syntaxbeschreibungen beziehen sich auf die Syntax der gängigen Regex-Implementierungen mit Erweiterungen, sie entsprechen also nur teilweise der obigen Definition aus der theoretischen Informatik.

Eine häufige Anwendung regulärer Ausdrücke besteht darin, spezielle Zeichenketten in einer Menge von Zeichenketten zu finden. Die im Folgenden angegebene Beschreibung ist eine (oft benutzte) Konvention, um Konzepte wie *Zeichenklasse*, *Quantifizierung*, *Verknüpfung* und *Zusammenfassen* konkret zu realisieren. Hierbei wird ein regulärer Ausdruck aus den Zeichen des zugrunde liegenden Alphabets in Kombination mit sogenannten *Metazeichen* ([,], (,), {, }, |, ?, +, *, ^, $, \, .) gebildet. Alle übrigen Zeichen des Alphabets stehen für sich selbst.

Zeichenliterale

Diejenigen Zeichen, die direkt (wörtlich, literal) übereinstimmen müssen, werden auch direkt notiert. Je nach System gibt es auch Möglichkeiten, den Oktal- oder Hexadezimalcode anzugeben.

Beliebiges Zeichen

- **.** : Ein Punkt bedeutet, dass an seinem Platz ein (fast) beliebiges Zeichen stehen kann. Abhängig vom verwendeten Programm kann ein Punkt auch ein *Newline* (Zeilenumbruch) enthalten, die meisten Implementierungen sehen jedoch *Newline* nicht als beliebiges Zeichen an.

Ein Zeichen aus einer Auswahl

Mit eckigen Klammern lässt sich eine *Zeichenauswahl* definieren. Der Ausdruck in eckigen Klammern steht dann für *genau ein* Zeichen aus dieser Auswahl (Einzeichenmuster).

Beispiele:

[egh]	eines der Zeichen „e“, „g“ oder „h“
[0-6]	eine Ziffer von „0“ bis „6“ (Bindestriche sind Indikator für einen Bereich)
[A-Za-z0-9]	ein beliebiger lateinischer Buchstabe oder eine beliebige Ziffer
[^a]	ein beliebiges Zeichen außer „a“ („^“ am Anfang einer Zeichenklasse negiert selbige)

In vielen neueren Implementationen können innerhalb der eckigen Klammern auch Klassen angegeben werden, die selbst wiederum eckige Klammern enthalten. Sie lauten beispielsweise:

[:alnum:]	Alphanumerische Zeichen: [:alpha:] und [:digit:].
[:alpha:]	Buchstaben: [:lower:] und [:upper:].
[:blank:]	Leerzeichen und Tabulator.
[:cntrl:]	Steuerzeichen. Im ASCII-Kode sind das die Zeichen 00 bis 1F, und 7F (DEL).
[:digit:]	Ziffern: 0, 1, 2,... bis 9.
[:graph:]	Graphische Zeichen: [:alnum:] und [:punct:].
[:lower:]	Kleine Buchstaben: a bis z.
[:print:]	Druckbare Zeichen: [:alnum:], [:punct:] und Leerzeichen.
[:punct:]	Zeichen wie: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~ .
[:space:]	Whitespace: tab, newline, vertical tab, form feed, carriage return, and space.
[:upper:]	Großbuchstaben: A bis Z.
[:xdigit:]	Hexadezimale Ziffern: 0 bis 9, A bis F, a bis f.

[Bearbeiten]

Vordefinierte Zeichenklassen

Es gibt vordefinierte Zeichenklassen, die allerdings nicht von allen Implementationen unterstützt werden, da sie lediglich Kurzformen sind und auch durch eine *Zeichenauswahl* beschrieben werden können. Wichtige Zeichenklassen sind:

- **\d** : eine Ziffer [0-9]
- **\D** : keine Ziffer [^0-9]
- **\w** : ein Buchstabe, eine Ziffer oder der Unterstrich [a-zA-Z_0-9]
- **\W** : kein Buchstabe, keine Zahl und kein Unterstrich [^\w]
- **\s** : Whitespace, meistens [\f\n\r\t\v]
- **\S** : alle Zeichen außer Whitespace [^\s]

Quantoren (Angabe der Anzahl Wiederholungen)

Quantoren (auch *Quantifizierer* oder *Wiederholungsfaktoren*) erlauben es, den vorherigen Ausdruck in verschiedener Vielfachheit in der Zeichenkette zuzulassen:

- **?** : Der voranstehende Ausdruck ist optional, er kann einmal vorkommen, muss es aber nicht, d. h. der Ausdruck kommt null- oder einmal vor.

- `+` : Der voranstehende Ausdruck muss mindestens einmal vorkommen, darf aber auch mehrfach vorkommen.
- ***** : Der voranstehende Ausdruck darf beliebig oft (auch keinmal) vorkommen.
- **{*n*}** : Der voranstehende Ausdruck muss exakt *n*-mal vorkommen.
- **{*min*,}** : Der voranstehende Ausdruck muss mindestens *min*-mal vorkommen.
- **{*min*,*max*}** : Der voranstehende Ausdruck muss mindestens *min*-mal und darf maximal *max*-mal vorkommen.

Beispiele:

- `a+` erlaubt ein "a" oder ein "aa" oder auch "aaaa" etc.
- `[ab]+` dagegen erlaubt ein "a", "b", "aa", "baab" etc.
- Ein `[0-9]{2,5}` findet "13", "28333", "123", aber nicht "0", "123123223" etc.

Gieriges Verhalten

Normalerweise wird von einem regulären Ausdruck mit Quantor die größtmögliche passende Zeichenkette gefunden (*gematcht*, von englisch "to match"), weshalb dieses Verhalten als „gierig“ (engl.: "greedy") bezeichnet wird. Da dieses Verhalten jedoch nicht immer so gewollt ist, lassen sich bei manchen neueren RegEx-Implementierungen Quantoren als "non-greedy" (also "nicht gierig") deklarieren. Hierfür wird dem Quantor ein Fragezeichen **?** nachgestellt. Der entstehende Ausdruck führt bei herkömmlichen RegEx-Implementierung zu einer Fehlermeldung.

Beispiel:

- Angenommen es wird auf den String "ABCDEB" der reguläre Ausdruck `A.*B` angewendet, so würde er den kompletten String "ABCDEB" matchen. Mit Hilfe des "non-greedy"-Quantors "*?" matcht der Ausdruck `A.*?B` die Zeichenkette "AB", bricht also die Suche nach dem ersten gefundenen "B" ab.

Die Implementierung von genügsamen ("non-greedy") Quantoren ist vergleichsweise aufwändig, weshalb nicht alle RegEx-Parser diese unterstützen.

Gruppierung mit runden Klammern

Ausdrücke lassen sich mit runden Klammern **(** und **)** *zusammenfassen*: Etwa erlaubt "(abc)+" ein "abc" oder ein "abcabc" etc.

Einige Programme speichern die Gruppierung ab und ermöglichen deren Wiederverwendung im Regulären Ausdruck oder bei der Textersetzung: Ein Suchen und Ersetzen mit

```
AA(.*?)BB
```

als Regulären Suchausdruck und

```
\1
```

als Ersetzung ersetzt alle Zeichenketten, die von **AA** und **BB** eingeschlossen sind, durch den zwischen **AA** und **BB** enthaltenen Text. D. h. **AA** und **BB** und das dazwischen wird ersetzt durch das dazwischen, also fehlen **AA** und **BB** im Ergebnis. \1, \2 usw. nennt man

Rückwärtsreferenzen (engl. "Backreferences"). \1 bezieht sich auf das erste Klammerpaar, \2 auf das zweite usw.; dabei zählt man die öffnenden Klammern.

Interpreten von regulären Ausdrücken, die Rückwärtsreferenzen zulassen, entsprechen nicht mehr dem Typ 3 der Chomsky-Hierarchie. Mit dem Pumping-Lemma lässt sich einfach zeigen, dass folgender regulärer Ausdruck, der feststellt, ob in einem String vor und nach der 1 die gleiche Anzahl von 0 steht, keine reguläre Sprache ist.

```
/^(0*)1\1$/
```

Alternativen

Man kann alternative Ausdrücke mit dem "|"-Symbol zulassen:

- "(ABC|abc)" bedeutet "ABC" oder "abc", aber z. B. nicht "Abc".

Weitere Zeichen

Um die oft auf Zeichenketten bezogenen Anwendungen auf dem Computer zu unterstützen, werden in der Regel zusätzlich zu den bereits genannten die folgenden Zeichen definiert:

- ^ steht für den Zeilenanfang. (nicht zu Verwechseln mit ^ bei der Zeichenauswahl mittels **[** und **]**)
- **$** kann je nach Kontext für das Zeilen- oder Stringende stehen.
- \ hebt ggf. die Metabedeutung des nächsten Zeichens auf, beispielsweise lässt der Ausdruck "(A*)+" die Zeichenketten "A*", "A*A*" etc. zu.
- **\b** steht für die leere Zeichenkette am Wortanfang oder am Wortende.
- **\B** steht für die leere Zeichenkette, die *nicht* den Anfang oder das Ende eines Wortes bildet.
- \< steht für die leere Zeichenkette am Wortanfang.
- \> steht für die leere Zeichenkette am Wortende.

Greedy (Gieriges) Verhalten beim Pattern Matching

Bei einem Pattern wie z.B. [0-9]* können 0 bis beliebig viele Pattern gematcht werden. Es wird jeweils die Anzahl gematcht, mit der die restlichen Pattern erfüllt werden können. Im Normalfall (greedy, gierig) werden zunächst möglichst viele gematcht. Im folgenden Beispile wird durch `[2-7]*` auch die 5 gematcht, da mit dem Rest durch 67a die restlichen Pattern gematcht werden können.

```
$s="1234567abc";
$s=~ s/1([2-7]*)(567|67)ab/xyz/;
print $s,"\n",$1,"\n",$2,"\n1------------\n";
```

```
xyzc
2345
67
1--------------
```

Bei einem Pattern, das nicht gierig ist (nongreedy) wird mit einem möglicht kurzen Teilstring das Pattern zu matchen, jedoch auch hier nur so, dass die restlichen Pattern gematch werden

können. Im Beispiel matcht `[2-7]*?` nur 234, da dann mit 567 der Rest gematcht werden kann,

```
$s="1234567abc";
$s=~ s/1([2-7]*?)(567|67)ab/xyz/;
print $s,"\n",$1,"\n",$2,"\n2------------\n";
```

```
xyzc
234
567
2----------
```

Das Verhalten erkennt man auch, wenn`[2-7]*` zweimal vorkommt. Bei greedy matcht das erste Vorkommen bereits alle Ziffern von 2-7, bei Nongreedy den leeren String.

```
$s="1234567abc";
$s=~ s/1([2-7]*)([2-7]*)ab/xyz/;
print $s,"\n",'$1=',$1,"\n",'$2=',$2,"\n3---------------\n";
xyzc
$1=234567
$2=
3---------------

$s="1234567abc";
$s=~ s/1([2-7]*?)([2-7]*?)ab/xyz/;
print $s,"\n",'$1=',$1,"\n",'$2=',$2,"\n4---------------\n";
xyzc
$1=
$2=234567
4---------------
```

Ausdrücke mit RegExp (Patternsuche, Substitution)

Die regulären Ausdrücke werden in folgenden Ausdrücken benutzt, deren Verhalten mit vielen nachgestellten Optionen verändert werden kann.

- Der m-Befehl steht für *match*, was man hier mit *Suche* übersetzen kann. Das *m* kann auch weggelassen werden. Der folgende Ausdruck durchsucht den Inhalt der Variable *$var* und liefert einen Array von Zeichenketten, auf die der Suchausdruck passt. Mit aktivierter g-Option liefert die Suche im Hashkontext alle Funde, deaktiviert alle erkannten Subausdrücke. Im Skalarkontext liefert der Ausdruck einen positiven Wert wenn der Suchausdruck gefunden wurde, mit *g*-Option die Anzahl der Funde. *i* lässt Groß/Kleinschreinung ignorieren, *o* Variablen nur einmal interpolieren, *m* den String als mehrzeilig und *s* als einzeilig betrachten. Die *x*-Option erlaubt den Suchausdruck über mehrere Zeilen zu verteilen und diesen auszukommentieren.

```
$var =~ [m]/<Suchausdruck>/[g[c]][i][m][o][s][x];
```

- Der s-Befehl steht für *substitute*, was ersetzen bedeutet. Er ersetzt den Teil des gegebenen Textes, auf den der Suchausdruck passt mit dem Ersatzausdruck.

```
$var =~ s/<Suchausdruck>/<Ersatzausdruck>/[e][g][i][m][o][s][x];
```

Dabei gilt:

- Ein String wird von Links nach rechts durchgearbeitet.
- Es wird immer der frühestmöglichste Treffer links gefunden.

Um jetzt jedes Vorkommen vom "Kamel" durch "Lama" zu ersetzen, könnte man folgendes schreiben:

```
1: $string = "Wir mögen Kamele, auch wenn Kamele übel riechen";
2: while ($string =~ s/Kamel/Lama/) {}
3: print $string;
```

Wenn also "Kamel" gefunden und erfolgreich ersetzt wurde, wird als Rückgabewert "true" geliefert. Dadurch wird unser Schleifeninhalt durchgeführt, der allerdings Leer ist. Danach wird erneuert unsere Bedingung durchgeführt. Zu beachten ist, dass die Substitution wieder ganz von vorne beginnt, also an der ersten Stelle im String anfängt, und nicht an der Stelle, an der zuletzt etwas verändert wurde. Manchmal ist dieses Verhalten gewünscht, aber in den seltesten Fällen ist dies der Fall, und es kann hierbei zu Problemen kommen, in folgendem while Konstrukt:

```
while ( $string =~ s/e/ee/ ) {}
```

Hiermit hat man eine Endlossschleife gebaut. Das erste vorkommen eines "e" wird durch ein "ee" ersetzt. Der Rückgabewert ist true und unsere Substitution beginnt wieder von vorne. Jetzt sind zwei "e" an der Stelle vorhanden, und dort ersetzen wir wieder das erste "e" durch "ee". Es existieren also schon 3 "e" an dieser Stelle. Dieses wird jetzt unendlich oft durchgeführt.

Optionen

Optionen dienen dazu das Verhalten der Regex zu verändern. Mit ihnen sind Sachen möglich, die so ohne weiteres nicht möglich wären. Jede Option kann dabei Regex grundlegend verändern. Die Optionen werden hierbei hinter den Regulären Ausdruck angehängt. Es ist auch möglich mehrere Optionen zu benutzen, die Reihenfolge der Optionen spielt hierbei keine Rolle. Hier aber nochmals ein paar praktische Beispiele:

```
m/kamel/i
m/k a m e l/xi
s/kamel/lama/ig
s/k a m e l/igx
```

Einige wichtige Optionen sollen hierbei bereits erläutert werden, andere wenn sie wichtig werden:

Option: /i

Mit der Option "i" *ignore-case* können wir Perl dazu veranlassen, dass zwischen Groß- und Kleinschreibung nicht mehr unterschieden wird. Folgendes Muster:

```
m/kamel/i
```

Würde also auch auf "KAMEL", "KAmel", "KaMeL", ... passen.

Option: /g

Im Kapitel "Substitution" wollten wir, dass jedes Vorkommen von "Kamel" durch "Lama" ersetzt wird, bei der folgenden Lösung mit der while-Schleife funktionierte das zwar, jedoch könnte dieses unter Umständen zu neuen Problemen führen. Das "g" steht hier für *global* und es möchte damit ausdrücken, dass wir den ganzen String durcharbeiten. Das Verhalten von "/g" ist folgendermaßen zu erklären, dass die Substitution nach einer Ersetzung nicht beendet wird, sondern an der letzten Position weiter macht, und nach weiteren Treffern sucht. Der Rückgabewert ist hierbei die Anzahl von Substitutionen, die innerhalb des Strings durchgeführt wurden. Wenn also mindestens eine Substitution durchgeführt wurde, ist der Rückgabewert automatisch "> 0" und somit ein "wahrer" Wert.

Hierbei ist zu beachten, dass die Option nur Auswirkung auf eine Substitution hat. Wir können z.B. nicht mit:

```
$string =~ m/e/g;
```

Die Anzahl der "e" Buchstaben innerhalb eines Strings zählen.

Dadurch, dass unsere Substitution jedoch nicht immer wieder von vorne anfängt, können wir das letzte Problem im Kapitel "Substitution" lösen.

```
1: $string = "Wir mögen Kamele, auch wenn Kamele übel riechen";
2: $string =~ s/e/ee/g;
3: print $string;
```

Dies würde nun nicht mehr in eine Endlosschleife enden, sondern folgenden String zurück geben:

```
Wir mögeen Kameelee, auch weenn Kameelee übeel rieecheen
```

Option: /x

Normalerweise werden innerhalb einer Regex Whitespace Zeichen als zu dem Muster gehörend angesehen.

```
m/a b/
```

Diese Regex würde auf ein "a" gefolgt von einem Leerzeichen, gefolgt von einem "b" passen. Mit dieser Option verliert das Leerzeichen seine bedeutung, und wir würden ein "a" gefolgt von einem "b" suchen.

Unter Whitespace Zeichen versteht man alle Zeichen, die nicht direkt etwas auf dem Bildschirm Zeichnen. Dies sind z.B.: Leerzeichen, Tabulatoren, Newlines, Formfeeds ...

Wahrscheinlich werden Sie den Sinn dahinter noch nicht nachvollziehen können. Für die sehr kleinen Regexe die wir bisher geschrieben haben, ist dieses auch unbrauchar. Allerdings werden wir später auf sehr viel komplexere Regexe stoßen. Damit diese besser lesbar sind, wurde diese Option implementiert. Damit kann man eine Regex über mehrere Zeilen verteilen, und ihnen sogar Kommentare geben.

Zusammenfassung

- "i", *case-insensitive*: Groß- und Kleinschreibung wird ignoriert
- "g", *global*: Es werden alle Stellen gesucht, die passen. Die Funktion ist eher für das Ersetzen mit regulären Ausdrücken interessant.
- "m", *multi-line*: Verändert die Bedeutung des $-Zeichen (siehe Sonderzeichen), damit es an der Position vor einem "\n" passt.
- "s", *single-line*: Verändert die Bedeutung des .-Zeichen (siehe Sonderzeichen), damit es auch auf einem "\n" passt.
- "x", *extended*: Alle Whitespace-Zeichen innerhalb des Regulären Ausdrucks verlieren ihre Bedeutung. Dadurch kann man Reguläre Ausdrücke optisch besser aufbereiten und über mehrere Zeilen schreiben.

Pattern Matching

Bei der Alternation innerhalb des Pattern Matching gibt es wenig zu beachten. Stellen Sie sich vor, Sie schreiben ein Programm und möchten, dass sich das Programm nach diversen Eingaben des Benutzers beendet. Sie könnten folgendes Schreiben

```
1: $input = <STDIN>;
2: if ( $input =~ m/q|quit|exit/i ) { exit; }
3: print "Hallo, Welt!\n";
```

Dieses Programm wartet auf eine Eingabe des Benutzers. Würden wir "q", "quit" oder "exit" eingeben, dann würde sich unser Programm beenden. Bei allem anderen würden wir die Meldunug "Hallo, Welt!" auf dem Bildschirm sehen.

Substitution

Innerhalb einer Substitution gibt es einige Punkte die wir beachten müssen. Stellen Sie sich vor, Sie möchten jedes vorkommen von "q" oder "quit" durch "exit" ersetzen. Vielleicht würden Sie so etwas schreiben:

```
s/q|quit/exit/ig;
```

Dies funktioniert aber nicht richtig. Sollte wirklich "quit" im String vorkommen auf dem Sie diese Regex anwenden, dann würde folgendes dabei raus kommen:

```
exituit
```

Um die genaue Vorgehensweise zu verstehen, müssen Sie wissen wie Perl eine Alternation behandelt. Es wird wieder Zeichen für Zeichen überprüft. Dabei wird aber auch die Reihenfolge der Alternation beachtet. Es wird zuerst der Ausdruck ganz links überprüft und erst wenn dieser nicht passt, wird der nächste Ausdruck überprüft. Sollte ein Ausdruck passen, wird sofort die Substitution ausgeführt. Bei "quit" passt das "q" sofort und es würde hier eine Substitution mit "exit" statt finden. Dadurch würde ein "quit" nie im Text ersetzt werden, da wir schon vorher das "q" durch "exit" ersetzt haben. Wir müssen also die Reihenfolge vertauschen:

```
s/quit|q/exit/ig;
```

Merke:

- Bei der Alternation ist die Reihenfolge wichtig

Dieser Punkt gilt auch für das Pattern Matching, allerdings hängt es von der Verwendung ab, ob wir die Reihefolge anpassen müssen. Im obrigen Beispiel ist es egal wodurch unser Programm beendet wird. Würden wir aber Informationen auslesen, kann es sehr wohl von Bedeutung werden, welche Reihenfolge wir in der Alternation gewählt haben.

Sonderzeichen

Für die Suchmuster stehen diverse Sonderzeichen zur Verfügung. Diese können dann beispielsweise für beliebige Zeichen oder für das Zeilenende stehen.

- . steht für *ein* beliebiges Zeichen außer dem "\n".
- ? steht für das ein- oder nullmalige Vorkommen des vorangegangenen Zeichens oder Gruppierung. Folgt das Fragezeichen auf einen Quantor dann wird dieser zu einem nicht gierigen Quantor.
- + steht für das ein- oder mehrmalige Vorkommen des vorangegangenen Zeichen oder Gruppierung.
- * steht für das null- oder mehrmalige Vorkommen des vorangegangenen Zeichen oder Gruppierung.
- ^ steht für den Beginn einer Zeile.
- $ steht für das Ende eines Strings. Wenn die /m Option benutzt wird, wird die Bedeutung so verändert das es für ein "\n" Zeichen steht.
- \A steht für den Beginn eines Strings.
- \Z steht immer für das Ende eines Strings, unabhängig ob die Option /m verwendet wurde.
- \ - das nächste Zeichen wird ohne Funktion interpretiert. Um einen Punkt zu suchen muss "\." benutzt werden, sonst wird ein beliebiges Zeichen gefunden.
- [] wird Zeichenklasse genannt und steht für ein angegebenes Zeichen. [dhb]ot würde "dot", "hot" oder "bot" finden.
- () 1. Funktion: Gruppiert Ausdrücke. 2.Funktion: Speichert den tatsächlichen Wert, der an dieser Stelle gefunden wurde, in einer Variable.
- | steht für das logische ODER. "ist|war" gibt sowohl true, wenn "ist" im Ausdruck vorkommt, als auch wenn "war" im Ausdruck enthalten ist.

Weitere Beispiele:

In jedem 2-dimensionalen Array soll Zeile und Spalte vertauscht werden. Array mit anderer Dimension sollen unverändert bleiben. Das ganze soll auch rekursiv funktionieren.

```
sub matrix{
my $s=@_[0];
print "$s\n";
while($s=~s/\[([^,\[\]]+),([^\[\],]+)]/Ä$2Ö$1Ü/){
 print "Nach substitute $s\n";
};
print "Nach Ende substitute $s\n";
$s=~ tr/ÄÖÜ/[,]/;

return $s;
}
#$e=<STDIN>;
$e="b[3,4]+a[d[1,2],c[3,4]+e[7,8]]";
print "Ergebnis\n";
print &matrix($e);
print "\n";
```

Das Programm liefert:

```
Ergebnis
b[3,4]+a[d[1,2],c[3,4]+e[7,8]]
Nach substitute bÄ4Ö3Ü+a[d[1,2],c[3,4]+e[7,8]]
Nach substitute bÄ4Ö3Ü+a[dÄ2Ö1Ü,c[3,4]+e[7,8]]
Nach substitute bÄ4Ö3Ü+a[dÄ2Ö1Ü,cÄ4Ö3Ü+e[7,8]]
Nach substitute bÄ4Ö3Ü+a[dÄ2Ö1Ü,cÄ4Ö3Ü+eÄ8Ö7Ü]
Nach substitute bÄ4Ö3Ü+aÄcÄ4Ö3Ü+eÄ8Ö7ÜÖdÄ2Ö1ÜÜ
Nach Ende substitute bÄ4Ö3Ü+aÄcÄ4Ö3Ü+eÄ8Ö7ÜÖdÄ2Ö1ÜÜ
b[4,3]+a[c[4,3]+e[8,7],d[2,1]]


ae oe ue ss ue oe ae
```

Es sollen alle Vokale verdoppelt werden:

```
$s="Hello World";
$s =~ s/([aeiou])/$1$1/g;
print $s, "\n";
```

Heelloo Woorld

Die Umlaute und ß sollen ersetzt werden:

```
%a =('ä','ae','ö','oe','ü','ue','ß','ss');
$s ="ä ö ü ß ü ö ä";
$s=~ s/([äöüß])/$a{$1}/ge;
print $s, "\n";
```

ae oe ue ss ue oe ae

Das gleiche kann auch ohne /g durch Wiederholung erreicht werden:

```
$s ="ä ö ü ß ü ö ä";
while ($s=~ s/([äöüß])/$a{$1}/e){};
print $s, "\n";
```

Die Zeichen sollen durch ihren ASCII Wert ersetzt werden:

```
$s='ABCDE';
$s=~ s/(.)/'{'.ord($1).'}'/ge;
print $s,"\n";
```

{65}{66}{67}{68}{69}

Die Zeichen sollen durch das nachfolgende Zeichen ersetzt werden:

```
$s='ABCDE';
$s=~ s/(.)/chr(ord($1)+1)/ge;
print $s,"\n";
```

BCDEF

Die Großbuchstaben sollen durch den entsprechenden kleinen Buchstaben ersetzt werden:

```
$s='1ABx-CDEy';
$s=~ s/([A-Z])/chr(ord($1)+32)/ge;
print $s,"\n";
```

1abx-cdey

Zeichen ersetzen *ohne* Reguläre Ausdrücke

Geht es nur darum, Zeichen durch andere zu ersetzen, sind Reguläre Ausdrücke häufig 'überqualifiziert', da es mit `tr//` eine deutlich einfachere und performantere Alternative gibt.

```
 $string =~ tr/SUCHEN/ERSETZEN/Optionen
```

Einige nützliche Beispiele:

```
 $string =~ tr/A-Z/a-z/;  # ersetzt alle Großbuchstaben durch
Kleinbuchstaben
 $string =~ tr/+/ /;      # ersetzt das + durch ein Leerzeichen
```

Umsetzen von DOS nach ANSI Code

```
$a ='äöüÄÖÜß';
$d =chr(132).chr(148).chr(129).chr(142).chr(153).chr(154).chr(225);
print $a,"\n";
print $d,"\n";
while ($_=<STDIN>) {
 eval("tr/$d/$a/");
 print STDOUT $_;
}
```