

Reguläre Ausdrücke

Kolja Engelmann, 708383

Anlagen:

- Dieses Dokument
- Folien des Vortrages
- 1 Datenträger
- pdf File des Dokuments
- pdf File der Folien

Inhaltsverzeichnis

Reguläre Ausdrücke	1
Inhaltsverzeichnis	2
Reguläre Ausdrücke in Perl	3
Einfache Mustererkennung mittels regulären Ausdrücken	3
Metazeichen	3
Quantifier	4
Allgemeine Quantifier	4
Einfache Quantifier	4
Beispiele für Quantifier	5
Muster Gruppieren	5
Alternativen	5
Fragen	6
Zeichenklassen	6
Abkürzungen in Zeichenklassen	7
Abkürzungen negieren	7
Fragen	7
Muster verankern	8
Wörter verankern	8
Rückwärtsreferenz	8
Speichervariablen	9
Automatische Speichervariablen	9
Präzedenz	10
Fragen	10
Mustervergleiche ohne /.../	10
Flags und Option Modifier	11
Der Bindungsoperator =~	11
Variableninterpolation	11
Ersetzen mittels s///	12
Split Operator	12
Andere Modifier	13
Literaturliste	15

Reguläre Ausdrücke in Perl

Reguläre Ausdrücke sind jedem, der bereits mit einem PC gearbeitet hat schon mindestens einmal begegnet. Reguläre Ausdrücke finden sich beispielsweise innerhalb von Textprogrammen und der „Suchen und Ersetzen“ Funktion wieder, aber auch Suchmaschinen verwenden Reguläre Ausdrücke, um Ihren Datenbestand zu durchsuchen, ähnlich der Möglichkeit unter Unix mittels grep Strings in Dokumenten wiederzufinden.

Ein Regulärer Ausdruck (oft auch als Suchmuster bezeichnet) ist also eine Schablone, die auf einen gegebenen String passt oder nicht.

In Perl geht diese Aussage jedoch noch um einiges weiter, denn hier lassen sich Strings mittels Regulären Ausdrücken schnell und flexibel behandeln, sie stellen regelrecht eine eigene Programmiersprache innerhalb der Programmiersprache dar. Diese ist glücklicherweise leicht zu erlernen, hat sie doch nur eine einzige Aufgabe zu erfüllen: sich einen String anzusehen und zu sagen ob er auf ein bestimmtes Suchmuster passt oder nicht.

Einfache Mustererkennung mittels regulären Ausdrücken

Um das Prinzip eines einfachen Mustervergleichs zu verdeutlichen, führen wir folgendes Beispiel durch:

```
$_ „Gesellschaft“;  
if (/esel/)  
{  
    print „passt“;  
}
```

Ausgabe: Passt

Die Variable `$_`, welche normalerweise Benutzereingaben von der Tastatur enthält, wird mit dem Wert „Gesellschaft“ belegt.

Der Ausdruck `/esel/` sucht nun in `$_` nach dem aus 4 Buchstaben bestehenden String `esel` und liefert entweder `true`, er wurde gefunden, oder `false`, er wurde nicht gefunden an die `if`-Schleife zurück, die dann gegebenenfalls das Wort „passt“ auf der Konsole ausgibt. Ob ein Ausdruck wie in diesem Fall nur einmal passt oder mehrmals vorhanden ist, ist in diesem Fall völlig belanglos.

An diesem Beispiel wird außerdem eine erste Semantische Regel deutlich.

- *Einen regulären Ausdruck schreibt man immer¹ zwischen zwei Schrägstriche (/.../).*

Metazeichen

Oftmals ist es jedoch nicht ausreichend einfache literale Strings auf Ihr Vorhandensein zu überprüfen. Deswegen gibt es eine Reihe von sogenannten Metazeichen, die innerhalb der Regulären Ausdrücke besondere Bedeutungen haben.

Die sogenannte Wildcard ermöglicht den Abgleich eines Strings mit seinem Suchmuster, mit einem variablen Zeichen.

Der Punkt (`.`) passt auf **ein** beliebiges Zeichen außer dem Newline-Zeichen (`\n`), welches ja auch Teil einer Zeichenkette sein kann. Das Suchmuster `/.est/` findet also demnach Strings wie `Pest`, `Test`, `Fest` usw.

¹ Natürlich nicht immer, wenn man den Präfix `m/` verwendet geht dies auch anders, aber darauf kommen wir erst später zu sprechen.

Was passiert jedoch, wenn ich die Zahl Pi 3.14159 innerhalb eines Strings finden möchte?

Die Verwendung des regulären Ausdrucks `/3.14159/` würde im Falle des Suchstrings „3_14159,3*14159“ nicht zum gewünschten Ergebnis führen, denn er würde `true` zurückgeben, es sei denn wir verwenden das sogenannte Escape – Zeichen (`\`).

Dieses Escape – Zeichen ermöglicht es, jedem Metazeichen innerhalb eines regulären Ausdrucks seine Besonderheit zu nehmen und zu einem gewöhnlichen Zeichen zu machen. Um also Pi 3.14159 zu finden schreibt man das Suchmuster `/3\.14159/`, welches auch wirklich nur auf den gewünschten Teilstring 3.14159 passen würde.

```
$_ = „3.14159“;  
if (/3\.14159/)  
{  
    print „passt“;  
}
```

Ausgabe: Passt

Quantifier

Allgemeine Quantifier

Als Quantifier bezeichnet man das Verfahren innerhalb eines Suchmusters eine bestimmte Anzahl von Vorkommen zu definieren. Möchte man testen, ob innerhalb eines Suchstrings der Buchstabe a mindestens zweimal und höchstens fünfmal gefunden wurde, so schreibt man die untere und die obere Grenze getrennt durch Kommata in geschweifte Klammern (`{...}`) hinter das zu suchende Zeichen.

Der Ausdruck `/a{2,5}/` passt also auf das zwei bis fünfmalige Auftreten des Buchstabens „a“. Kommt der Buchstabe nur einmal vor passt das Muster nicht, bei fünfmaligem Auftreten erhalten wir einen Treffer und auch ein zehnmaliges Auftreten führt `/a{2,5}/` noch zu einem Treffer, da aufgrund der oberen Schranke nur die ersten fünf „a“ gefunden werden.

Allgemeine Quantifier können auch verwendet werden, um das Vorkommen von genau x Zeichen y zu finden. Dazu lässt man bei der Angabe des Suchmusters einfach die obere Grenze und das Komma weg.

Das Suchmuster `/a{3}/` würde also auf „Haaallo“ passen.

Lässt man hingegen nur die obere Schranke wegfällen `/a{3,}/`, so bedeutet dies, das mindestens dreimal a innerhalb des Strings stehen muss, obwohl auch 88x a gefunden werden würde.

- `x{2,4}` findet jeden String der xx, xxx oder xxxx enthält. Der String darf nicht weniger als 1, aber mehr als 4 x enthalten
- `x{2,}` findet jeden String, der mindestens zwei aufeinanderfolgende x enthält.
- `x{2}` findet xx

Einfache Quantifier

Eine Besonderheit der Quantifier sind die einfachen Quantifier.

Da perl ursprünglich als Programmiersprache entwickelt wurde um Systemadministratoren unter Unix das Leben leichter zu machen, existieren unzählige Abkürzungen. Drei dieser Abkürzungen bilden die Gruppe der einfachen Quantifier.

- Das Sternchen (`*`) steht für das Vorkommen von **keinem oder mehr** Zeichen und ist die Abkürzung für `/x{0,}/`.
- Das Pluszeichen (`+`) steht für das Vorkommen von **mindestens einem** Zeichen und ist die Abkürzung für `/x{1,}/`.

- Das Fragezeichen steht für „**einmal oder keinmal**“ auftreten eines Zeichens und ist die Abkürzung für $/x\{0,1\}/$.

Beispiele für Quantifier

Suchmuster	Suchstring	Ergebnis
/hal{2}oe\.e/	halloele	Ja
/ja*\./ gefunden	Jaaa.....gefunden	Ja
/co+l/	cl	Nein
/co?l/	cl	Ja
/(Hallo)+\.{3}/	Hallo...	Ja

Muster Gruppieren

Wie in der Mathematik benutzt man auch in der Mustererkennung runde Klammern zum gruppieren². Denn möchte man überprüfen, ob das Wort „Hallo“ mehrfach innerhalb des Suchstrings vorkommt, erreicht man mittels `/hallo+/` nichts weiter als das auffinden eines Strings, der beispielsweise „hallooooo“ beinhaltet. Wird das zu suchende Teilwort jedoch mittels der runden Klammern gruppiert `/(hallo)+/` so kann der String „hallohallo“ gefunden werden.

Achtung!

`/(Hallo)+/` findet Strings wie beispielsweise „hallohallohallo“, aber was findet dann der reguläre Ausdruck `/(hallo)*/?` Dieser findet Strings wie „Test“, „Katze“ usw.³

Alternativen

Die Alternative innerhalb eines Regulären Ausdrucks bietet die Möglichkeit eine Auswahl von zwei oder mehreren Suchmustern an einer Stelle zu finden.

Dies geschieht mittels des senkrechten Balkens (`|`), der in diesem Zusammenhang auch „oder“ genannt wird. Wird der linke Teil der Alternative nicht gefunden, so wird der rechte Teil überprüft.

Findet `/Hallo Welt|s/` also den String „Hallo Wels“?

Nein, denn die Präzedenz von (`|`) ist niedriger als die der Zeichenketten.⁴

Vielmehr findet `/Hallo Welt|s/` entweder „Hallo Welt“ oder „s“

Alternativen können hierbei auch verschachtelt werden.

`/Haus(Katze|(n))/?` passt auf Hauskatze, Hauskatzen oder Haus.

Achtung!

Die Alternative unterscheidet sich von den Quantifiern (`.`, `*`, `+`, `?`)! Gilt ein Quantifier nur jeweils für das vor ihm befindliche Zeichen oder Zeichengruppe, so gilt die Alternative (`|`) jeweils für die gesamte Klasse der Wortzeichen vor oder nach dem „oder“ Operator.

Fragen

1. Wie würde ein Muster aussehen müssen, welches eine beliebige Anzahl von Backslashes (`\`) gefolgt von einer beliebigen Anzahl von Sternchen (`*`) finden soll?

`/***/`

² Und zur Speicherung von Suchergebnissen, dazu aber später mehr.

³ Der Quantifier (`*`) bedeutet keinmal oder mehr, dieses Beispiel würde hingegen auf jeden beliebigen, sogar auf einen leeren String passen.

⁴ Präzedenz ähnelt der mathematischen Punkt-vor-Strich-Rechnung. Manche Operatoren werden mit höherer Wertigkeit ausgeführt, dazu jedoch später mehr.

2. Wie würde ein Muster aussehen, welches genau zwei a gefolgt von mindestens zwei b finden soll?
`/aab+/, /a{2}b+/`
3. Welcher Teil des Regulären Ausdrucks `/h|hu|hun|hund/` trifft auf den String `hund` zu? `h`, denn es wird von links nach rechts mit dem Regulären Ausdruck verglichen.
4. Welcher Teil des Regulären Ausdrucks `/hund|hun|hu|h/` trifft auf den String `hund` zu? `hund`, denn es wird von links nach rechts mit dem Regulären Ausdruck verglichen.

Zeichenklassen

Eine Zeichenklasse ist eine Liste von Zeichen die innerhalb eckiger Klammern (`[]`) geschrieben wird. Sie passt auf ein beliebiges Zeichen dieser Liste. Es wird hierbei immer nur ein Zeichen gefunden, aber dies kann ein beliebiges Zeichen der Klasse sein.

So passt die Klasse `[abcdefghi]` auf jedes der 8 Zeichen, zum Abkürzen kann man jedoch den Bindestrich (`-`) verwenden, so dass die Zeichenklasse nun `[a-df-i]` heißt.⁵

Hält sich im vorigen Beispiel die Schreibersparnis noch in Grenzen, würde man bei folgendem Beispiel diese Einsparung doch schwer missen, wäre sie nicht vorhanden. Für ein beliebiges Zeichen innerhalb des Alphabets steht die Klasse `[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]` oder aber die Verkürzte Form `[a-zA-Z]`. Dies funktioniert natürlich auch bei Ziffern, `[0123456789]` entspräche `[0-9]`.

Solche Zeichenklassen sind nur sinnvoll, wenn sie innerhalb eines vollständigen Suchmusters auftreten.

```
$_ „Der Jahr 2000 Bug“;           //zu durchsuchender String
if (/Jahr [0-9]+ Bug| Fehler/)    //Suchmuster passt auch auf Der Jahr 2000 Fehler
{
    print „passt“;
}
Ausgabe: Passt
```

Manchmal ist es aber leichter anzugeben welches Zeichen nicht innerhalb des Suchstrings enthalten sein soll. Hierzu verwendet man das Caret-Zeichen (`^`). Durch ein Caret-Zeichen zu Beginn einer Zeichenklasse wird diese also negiert. Die Klasse `[^abc]` passt also auf alle Zeichen außer auf `a`, `b` und `c`.

Achtung!

Innerhalb einer Zeichenklasse besitzen `^`⁶ und `-` Sonderbedeutungen, und müssen, falls sie als normale Zeichen behandelt werden sollen, durch ein Escape – Zeichen geschützt werden. Außerhalb der Zeichenklasse haben sie hingegen eine andere Bedeutung (`^`) oder werden als ganz normales Zeichen angesehen und bedürfen dabei nicht der Sonderbehandlung durch das Escape – Zeichen.

Abkürzungen in Zeichenklassen

Manche Zeichenklassen werden so häufig verwendet, dass es sich lohnt eigene Abkürzungen für sie zu verwenden. So lässt sich die Zeichenklasse für eine beliebige Ziffer `[0-9]` durch `\d` Abkürzen, weswegen man anstelle des obigen Beispiels auch

```
/Jahr \d+ Bug| Fehler/
```

⁵ Wie gesagt, perl wurde für Administratoren entwickelt um Schreibarbeit gering zu halten.

⁶ Siehe Muster verankern das Caret (`^`) Zeichen

schreiben könnte.

Die Abkürzung `\w` steht für ein Wort-Zeichen. Dies ist eine Abkürzung für die Zeichenklasse `[a-zA-Z0-9_]`⁷ `\w` findet natürlich kein ganzes Wort, es steht nur für ein einzelnes Zeichen dieser Klasse. Um ein vollständiges Wort nach Perl Definition zu finden benötigt man den + Quantifier `\w+` findet beispielsweise ein beliebiges Wort bestehen aus Zeichen des Alphabetes, Ziffern und dem Unterstrich (`_`).

Die Abkürzung `\s` steht für ein beliebiges Whitespace - Zeichen. Whitespace - Zeichen sind Elemente der Zeichenklasse `[\f\t\n\r]`.

- `\f` steht für den Formularvorschub
- `\b` steht für den Tabulator
- `\n` steht für Neue Zeile
- `\r` steht für Wagenrücklauf

Diese Zeichen verändern nur die Ausgabeposition und sehen für den menschlichen Betrachter alle gleich aus. Deswegen werden sie zu einer Klasse zusammengefasst und meistens mit `\s+` oder `\s*` abgefragt. `\s/` ohne Quantifier kommt dabei eher selten vor.⁸

Abkürzungen negieren

Möchte man das exakte Gegenteil einer der Zeichenklassen `\w`, `\d` oder `\s` verwenden, zum Beispiel um einen String zu finden, der zwischen zwei Wörtern alles außer einem Whitespace-Charakter aufweist, also beispielsweise „Hallo-Hallo“, so kann man die Zeichenklasse `\s` entweder wie folgt negieren,

```
$_ „Hallo-Hallo“;           //zu durchsuchender String
if (/w+[^s]w+/)           //Suchmuster passt auf den gewünschten String
    {print „passt“;}
Ausgabe: Passt
```

oder aber man schreibt anstelle des `[^s]` die Abkürzung `\S`.
Für die Klassen `\d` und `\w` gilt entsprechendes.

1. `\d` steht für eine Ziffer
2. `\D` steht für ein Zeichen außer Ziffern
3. `\w` steht für ein Wort-Zeichen
4. `\W` steht für ein Zeichen außer Wort-Zeichen
5. `\s` steht für ein Whitespace-Zeichen
6. `\S` steht für Zeichen außer Whitespace-Zeichen

Fragen

1. Was kann man mittels der Zeichenklasse `/[\dA-Fa-f]+/` finden?
Hexadezimalzahlen
2. Worauf passt die Klasse `[\d\D]`?
Auf eine Ziffer oder eine Nichtziffer einschließlich des Newline Zeichens(`\n`)!
3. Wenn die Zeichenklasse `[\d\D]` auf eine Ziffer oder eine Nichtziffer passt, worauf passt dann `[^\d\D]`?
Passt auf alles das keine Zahl oder keine Nichtzahl ist – auf nichts.
4. Passt `(19|20)\d\d\s?` auf den String „20“ und wenn ja warum?

⁷ Die „Wort“ Definition sollte hier nicht allzu ernst genommen werden.

⁸ Information übernommen aus dem „Kamelbuch“ Programmieren mit Perl

Ja, als Alternative wird „“ ausgewählt und das Whitespace Zeichen kommt keinmal vor!

Muster verankern

Passt ein Muster nicht sofort zu Beginn eines Strings so handelt es sich von links nach rechts durch den String und versucht an einer anderen Stelle den gewünschten String zu finden. Dies muss nicht unbedingt erwünscht sein. Ist es wichtig ein Suchmuster genau am Anfang oder am Ende eines Strings zu identifizieren, oder soll ein bestimmtes Wort erkannt werden, so benutzt man Ankerzeichen.

Das Caret (^) Ankerzeichen markiert hierbei den Anfang eines Strings, das Dollarzeichen(\$) das Ende. Das Muster /^hallo/ findet „hallo“ also nur, wenn es direkt am Anfang des Strings steht, „hallihallo“ würde es ignorieren. Das Muster /hallo\$/ hingegen würde „hallo“ nur dann finden, wenn es am Ende eines Strings stünde, „hallodrio“ würde es ignorieren.

Die beiden Ankerzeichen (^) und (\$) werden häufig gemeinsam benutzt um festzustellen, ob ein Muster auf einen gesamten String passt. Am häufigsten wird hierbei das Muster /^\s\$/ verwendet, um zu überprüfen ob ein String leer ist. Ohne diese Anker würde das Suchmuster auch auf andere Strings passen.

Wörter verankern

Es gibt nicht nur Anker, die einen String am Anfang oder am Ende eines Strings verankern, sondern auch einen Anker, der ein Wort begrenzt. \b steht für eine beliebige Wortgrenze und /\bTest\b/ würde somit „Dies ist ein Test für reguläre Ausdrücke“ finden, „Dies ist ein Testprogramm“ jedoch ignorieren. Es ist hierbei nicht zwingen notwendig immer ein ganzes Wort mit \b zu verankern, man kann durchaus die verschiedenen Ankerzeichen kombinieren. /^hallo\b/ findet demnach einen String „hallo“, der zu Beginn eines Strings steht und am Ende durch ein anderes Zeichen außer einem Wortzeichen begrenzt ist.

Um das genaue Gegenteil der Wortbegrenzung \b auszudrücken, verwendet man den \B Anker. Dieser greift nur dann, wenn auf den zu suchenden String ein Wortzeichen folgt. /\bhall\B/ findet also „hallo“, jedoch nicht „hall“.

- & steht nicht nur für das String - Ende Zeichen, sondern auch für das Newline-Zeichen, somit ist „hallo“ für das Suchmuster /hallo\$/ das gleiche wie „hallo\n“.
- Wortbegrenzungszeichen lassen sich kombinieren
- \b steht für die Wortbegrenzung → das Wort endet an dieser Stelle
- \B steht für Wortfortführung → das Wort geht hier noch weiter

Rückwärtsreferenz

Die runden Klammern, wie wir sie bisher nur zum gruppieren kennen gelernt haben, haben noch eine zweite Funktion. Sie werden dazu verwendet die regexp Maschine anzuweisen, sich den Teil des durch sie gefundenen Strings zu merken. Dies geschieht automatisch, perl merkt sich immer den in Klammern gefundenen Suchstring, selbst wenn man diese Klammern nur zum Gruppieren verwenden wollte.

Eine Rückwärtsreferenz bezieht sich also auf einen Speicher, der zu einem früheren Zeitpunkt der Mustererkennung gefüllt wurde. /(.)\1/ Dies ist eine solche Rückwärtsreferenz. Das Muster lautet ausgeschrieben „Finde ein beliebiges Zeichen, lege es im Speicher eins ab [(.)] und finde exakt das gleiche Zeichen danach noch einmal (\1)“. Hieran wird deutlich wie man auf die angelegten Speicher zugreifen kann, nämlich durch ein Escape - Zeichen gefolgt von der Zahl des Speichers in der der Wert

gespeichert wurde. Wie aber findet man die korrekte Speichernummer? Die folgenden Fälle sollen Aufschluss bringen.

```
/((Hallo|Ciao) (Welt)) \1/ findet „Hallo Welt Hallo Welt“ oder aber „Ciao Welt Ciao Welt“  
/((Hallo|Ciao) (Welt)) \2/ findet „Hallo Welt Hallo“ oder „Ciao Welt Ciao“  
/((Hallo|Ciao) (Welt)) \3/ findet „Hallo Welt Welt“ oder „Ciao Welt Welt“
```

Man sieht, die Nummerierung der Rückwärtsreferent bezieht sich immer unmittelbar auf die x. öffnende Klammer. Man kann sich leicht merken, das die erste öffnende Klammer den Speicher \1, die zweite \2 usw. Speicher zugewiesen bekommt.⁹

Speichervariablen

Die Rückwärtsreferenzen habe noch eine weitere Bedeutung, denn sie werden zusätzlich in den Variablen \$1-\$n für perl auch außerhalb der Mustererkennung zugänglich gemacht und stehen nach dem Mustererkennung zur Weiterverarbeitung zur Verfügung.¹⁰

Achtung!

\$n bedeutet den n-ten Speicher einer bereits beendeten Mustererkennung, während \n auf den n-ten Speicher der gegenwärtigen Mustererkennung deutet.

```
my $test= „hallo du“;  
if ($test =~ /\s(\w+)/)11  
{print „Das gefundene Wort lautet $1“;}  
Ausgabe: du
```

Automatische Speichervariablen

Zusätzlich zu den explizit erzeugten Variablen (\1 - \n) erstellt perl automatisch zu jedem Mustervergleich drei weitere Variablen, die im weiteren Programmverlauf verwendet werden können.¹² Dies sind die Variablen \$`, \$& und \$´.

Die Variable \$` (Backtick, nicht Hochkomma) speichert den Teil des ursprünglich zu durchsuchenden Strings, der **vor** dem durch das Muster gefundenen Teilstring steht.

Die Variable \$& enthält den Teilstring, der durch das Suchmuster identifiziert wurde oder, falls dieses nicht gefunden wurde, den Wert undef.

Die Variable \$´ (Backtick, nicht Hochkomma) speichert den Teil des ursprünglich zu durchsuchenden Strings, der **hinter** dem durch das Muster gefundenen Teilstring steht.

Wozu diese Variablen verwendet werden können soll das folgende kleine Programm verdeutlichen.

```
1)#!/usr/bin/perl -w  
2)while (<>)  
3)    {  
4)        chomp;  
5)        if (/Testsuchmuster/)  
6)            {  
7)                print „Treffer: |$`<$&>$´|\n“;  
8)            }  
9)        else
```

⁹ Rückwärtsreferenzen ab \10 gelten als geschützte Oktalzahlen. Um eine Verwechslung zu vermeiden, schreibt man deswegen bei Oktalzahlen eine führende null (\010) Die Menge der Rückwärtsreferenzen ist demnach nur durch den Hauptspeicher begrenzt.

¹⁰ Dazu aber erst später mehr

¹¹ Was bedeutet =~? Darauf kommen wir gleich zu sprechen (s.Bindungsoperator)

¹² Zugegebener Maßen aber unmögliche Bezeichnungen besitzen ☺

```
10)          {  
11)          print „Kein Treffer\n“;  
12)          }  
13)      }13
```

Dieses Programm dient dem Testen von Mustern. es liest die Eingaben des Benutzers von der Konsole ein und führt einen Mustervergleich mit dem in Zeile 5 angegebenen regulären Ausdruck. Interessant ist hierbei nur Zeile 7. In dieser Zeile wird das gefundene Suchmuster in der Form „StringDavor<Suchmuster>StringDanach“ angezeigt.

Präzedenz

Bei der Menge an Metazeichen und Regeln fällt es schwer zu entscheiden welche Regel Vorrang vor einer anderen besitzt. Aus diesem Grund wurden ähnlich wie in der Mathematik Regeln für den Zusammenhalt einzelner Metazeichen, Quantifier, Ankerzeichen usw. erstellt.

1. Runde Klammern `()`, die zum gruppieren und speichern von Teilen eines Musters verwendet werden, halten am stärksten zusammen.
2. Quantifier `*`, `+`, `?`, `{x,4}` hängen immer mit dem Element zusammen auf das sie folgen
3. Die Ankerzeichen `^`, `$`, `\b`, `\B` und Zeichenfolgen¹⁴. Die Buchstaben halten genauso eng zusammen wie die Ankerzeichen mit denen sie in Verbindung stehen.
4. Auf der niedrigsten Stufe steht die Alternative `|`. Dies ist wichtig um Suchstrings der Form `/Hallo|Tach/` auch wirklich auf die String „Hallo“ oder „Tach“ zutreffen zu lassen. Wäre die Präzedenz der Alternative höher ließen sich mit dem obigen Beispiel die unsinnigen Strings „Halloach“ oder „HallTach“ finden.

Fragen

1. Wie findet man einen kompletten String, der entweder das Wort „Hallo“ oder das Wort „Ciao“ enthält?
`/^(Hallo|Ciao)$/`
2. Was findet der Ausdruck `/^\$[a-zA-Z_]+/`
Passt auf einen String der den Namen einer skalaren variablen enthält (nicht deren Wert)

Mustervergleiche ohne `/.../`

Bisher wurden Suchmuster jeweils mittels der normalen Schrägstriche `/.../` abgegrenzt. Dies kann bei gegebenen Fällen nicht zum gewünschten Erfolg führen. Ein Beispiel soll hier Klarheit verschaffen. Ein Suchmuster welches den String „http://“ findet, um beispielsweise URL's aus einem String herauszufiltern würde bisher so aussehen: `/http:///`. Man erkennt, dass dies nicht möglich ist, der Suchstring würde bereits mit dem ersten Schrägstrich beendet, der gewünschte String nicht gefunden werden. Abhilfe schafft hierbei der `m`-„Operator“ vor dem Suchstring. Dieser weist die RegExp Maschine an, dass auf das `m` folgende Zeichen als Begrenzung des gesamten regulären Ausdrucks anzusehen ist. In unserem Falle wäre beispielsweise eine Raute (`#`) sinnvoller `m#http://#`, aber jedes beliebige Zeichen wäre hier denkbar.

- *Möglich für die Verwendung des `m` – „Operators“ sind jegliche Zeichen, die keinen Unterscheid zwischen der linken und der rechten Entsprechungen aufweisen (`m!...!`, `m%...%` usw.) genauso wie jegliche Art von Klammern (`{...}`, `[...]`, `(...)`)*

¹³ Originalprogramm entnommen aus dem „Lamabuch“ Einführung in Perl

¹⁴ Eine Zeichenfolge ist eigentlich auch ein Operator, hierfür wird jedoch kein Zeichen verwendet.

Flags und Option Modifier

Die bisherigen Möglichkeiten zum Ignorieren von Groß- und Kleinschreibung waren mehr als dürftig. Um beispielsweise eine nicht case-sensitive Überprüfung auf Vorkommen des Strings „Yes“ durchzuführen war ein Muster des Typs `/[Yy][Ee][Ss]/`. Es leuchtet ein, dass dies bei längeren Strings nicht nur aufwändig, sondern auch langsam ist. Abhilfe schafft hier der `/i` Option – Modifier.

Modifier im Allgemeinen werden hinter das schließende Zeichen zur Begrenzung des Suchmusters geschrieben (`/.../i`) und können kombiniert werden, in dem sie hintereinander an das Suchmuster ohne Trennzeichen angehängt werden.

Ein weiterer Modifier ist der `/s` Option – Modifier, der es erlaubt dem Metazeichen Punkt (`.`) die Einschränkung auf ein beliebiges Zeichen ohne das Newline Zeichen zu nehmen. Wird der `/s` Option - Modifier also gesetzt verhält sich das Metazeichen Punkt (`.`) genau wie die Zeichenklasse `[\d\D]` und findet somit auch Newline – Zeichen (`\n`).

Der Bindungsoperator `=~`

Der Bindungsoperator (`=~`) weist perl an, einen Mustervergleich mit dem String rechts des Operators durchzuführen.¹⁵

```
my $test = ‚Ein kleiner Test‘;
if ($test =~ /einer/)
{print „passt“;}
```

Ein besonders häufig verwendetes Beispiel für den Zuweisungsoperator (`=~`) findet sich hier.

```
my $wert = „Abfragewert“;
$abfrage = (<STDIN> =~ /\b$wert\b/is);
```

Der Variablen `$abfrage` wird hierbei ein Boolescher Wert zugewiesen, die eigentliche Benutzereingabe wieder verworfen. Auf diese kommt es in diesem Falle auch gar nicht an, es geht darum, ob `/\b$wert\b/is` Teil der Benutzereingabe `<STDIN>` ist.¹⁶

Variableninterpolation

Im obigen Beispiel wurde absichtlich nach dem Suchmuster `/\b$wert\b/is` gesucht. Hieran wird deutlich, dass perl Variableninterpolation unterstützt. Natürlich sucht die RegEx Maschine nun nicht nach dem String „\$wert“, sondern nach dem Inhalt der skalaren Variablen `$wert`, in diesem Fall „Abfragewert“. Reguläre Ausdrücke funktionieren in diesem Punkt genau wie Strings in doppelten Anführungszeichen.¹⁷ Dies bietet zum Beispiel den Vorteil Daten von der Kommandozeile als Argument einzulesen und weiterzuverwenden.

¹⁵ Bedeutet im Grunde genommen nur: Führe die Mustererkennung rechts von mir mit dem String links von mir aus. Dies führt dazu, dass die Variable `$_` nicht mehr verwendet werden muss

¹⁶ Wieder einmal eine Abkürzung für einen weitaus längeren Ausdruck, perl - Programmierer wollen ja Buchstaben sparen.

¹⁷ Strings in einfachen Quotas führen keine Variableninterpolation durch, Strings in doppelten Quotas hingegen interpolieren Variablen

Ersetzen mittels s///

Die RegExp Maschine kann auch zur Verarbeitung von Strings mittels einer Funktion, ähnlich dem Suchen und Ersetzen handelsüblicher Textprogramme verwendet werden. Hierzu dient der Ersetzungsoperator s///.¹⁸ Dieser ersetzt den durch das Suchmuster gefundenen Teil einer Variablen durch einen Ersetzungstring. Die Pflicht an erster Stelle des Ersetzungsoperators eine Variable anstelle eines normalen Strings zu verwenden rührt daher, dass s/// Daten verändert und diese somit in „etwas“ enthalten sein müssen. Üblicherweise ist dies eine Variable.¹⁹

```
my $test = „Hallo du“;
s/du/sie/;
print $test;
Ausgabe: Hallo sie
```

Wie auch bei den einfachen Mustervergleichen (/.../) gibt es auch beim Ersetzungsoperator Modifier. Ohne Modifier verwendet, ersetzt s/// das erste Vorkommen des Suchstrings durch den Ersetzungstring und bricht dann ab. Ist jedoch eine gesamte Ersetzung ähnlich der Funktion „Alle Ersetzen“ erwünscht, so verwendet man den /g Modifier. Die Modifier zum Ignorieren von Groß- und Kleinschreibungen (/i) sowie die Erweiterung des Metazeichens Punkt (.) (/s) sind natürlich ebenfalls verfügbar.

Oftmals kommt es vor, dass ein ganzes Wort komplett durch Großbuchstaben ersetzt werden soll, oder entsprechend nur in Kleinbuchstaben dargestellt werden soll. Dies kann durch die Verwendung des Uppercase (\U) und Lowercase (\L) Symbols bewerkstelligt werden, welche standardmäßig immer für alle nachfolgenden Zeichen gelten. Sollen (\U) und (\L) nicht bis zum Ende des jeweiligen Strings gelten, so kann man deren Wirkung mit Einem Endzeichen (\E) wieder aufheben.

Achtung!

Werden die Zeichen für Groß- und Kleinschreibung (\U, \L) klein geschrieben (\u, \l), so beziehen sie sich nur noch auf das unmittelbar folgende Zeichen.

Natürlich kann man die Zeichen auch kombinieren

```
my $test = „hALlO Welt“;
$test = s/(Hallo|Ciao)/\u\L$1/ig;
Ausgabe: Hallo Welt
```

Dies ersetzt innerhalb des strings „hALlO Welt „das Wort Welt und schreibt den ersten bchstaben groß, und den Rest klein. Das hierbei eine Rückwärtsreferenz verwendet wurde, um dies zu realisieren sei nur am Rande erwähnt.

Split Operator

Der Split-Operator benutzt reguläre Ausdrücke, um einen String anhand eines vorher gewählten Trennzeichens in ein Array aufzuspalten. Die Syntax für diesen Operator lautet

```
@feld = split /Trennzeichen/ $string
```

Beispiel:

```
@feld = split /:/, „abc:def:ghi:jkl“;
ergibt: ("abc","def","ghi","jkl")
```

Achtung!

¹⁸ Natürlich können, wie beim einfachen Mustervergleich, auch andere Begrenzungszeichen verwendet werden (#,% usw.)

¹⁹ Obwohl dies natürlich alles sein kann was links vor einem Zuweisungsoperator steht.

Folgen zwei der Trennsymbole direkt nacheinander so wird ein leerer Feldeintrag erzeugt.

Andere Modifier

Perl kennt außer den schon bekannten /s, /g, /i usw. Modifiern noch viele andere, erwähnt sei hierbei jedoch nun der x/ Modifier, der für Quellcodeleser wohl wichtigste Modifier²⁰.

Der /x Modifier ignoriert innerhalb eines regulären Ausdrucks jegliche Leerzeichen, Zeilenumbrüche und Kommentare. Was im ersten Moment unnötig erscheint soll im folgenden Beispiel verdeutlicht werden:

Aufgabe: Die Kommentare sollen aus einer C Datei entfernt werden.

Lösung:

```
$/ = undef;
$_ = <>;
s#/\[^[^]*\*\+([\^/*][^]*\*\+)*|("(\.|\.[^\"]\)*"|'(\.|\.[^']\)*'|.[^/'"\]*)#$2#gs
print;
```

Ohne das Wissen um was es sich hier handelt wäre es viel Arbeit dies zu entschlüsseln, der /x Modifier lässt den Quelltext nun wie folgt darstellen:

```
s{
    /\*          ## Start of /* ... */ comment
    [^]*\*\+    ## Non-* followed by 1-or-more '*'s
    (
        [^/*][^]*\*\+
    )*          ## 0-or-more things which don't start with /
                ## but do end with '*'
    /          ## End of /* ... */ comment

    |          ## OR various things which aren't comments:

    (
        "        ## Start of " ... " string
        (
            \\.    ## Escaped char
            |      ## OR
            [^\"]  ## Non "\"
        )*
        "        ## End of " ... " string

        |          ## OR

        '        ## Start of ' ... ' string
        (
            \\.    ## Escaped char
            |      ## OR
            [^']  ## Non "'"
        )*
        '        ## End of ' ... ' string

        |          ## OR

        .        ## Anything other char
        [^/'"\]* ## Chars which doesn't start a comment, string or escape
    )
}{$2}gxs;
    [^\"]      ## Non "\"
    )*
    "          ## End of " ... " string

    |          ## OR
```

²⁰ Es gibt Wettbewerbe in denen der unverständlichste perl Code prämiert wird, der x/ Modifier wirkt dem entgegen, zumindest bei regulären Ausdrücken

```
'      ## Start of ' ... ' string
(
  \\.      ## Escaped char
|      ## OR
  [^\\" ]  ## Non '\
)*
'      ## End of ' ... ' string

|      ## OR

.      ## Anything other char
[^\\" ]*  ## Chars which doesn't start a comment, string or escape
)
}{{2}}gxs;21
```

²¹ Code entnommen aus dem Perl Faq Seite 6 <http://www.perl.com>

Literaturliste

Randal L. Schwarz & Tom Phoenix, Einführung in Perl,
O'Reilly Köln 3. Auflage 2002

Thorsten Roßner, Perl,
BHV, 2.

Stefan Münz, Selfhtml,
Selfhtml8.0

Tom Christiansen and Nathan Torkington, Perl F.A.Q.
<http://www.perl.com> 1997-1999

Tekromancer (?), Einführung in Perl,
<http://www.tekromancer.com> 2001

Literaturtipp:

Jeffrey E.F. Friedl, Reguläre Ausdrücke,
O'Reilly Köln, 1. Auflage Oktober 1997