ReguläreAusdrücke-IX.doc

Einer der Gründe für Perls Leistungsfähigkeit sind seine raffinierten "regulären Ausdrücke" mit nummerierten Rückbezügen, positiver und negativer Vorausschau sowie zahlreichen Feinheiten, die die Arbeit erleichtern.

Reguläre Ausdrücke gibt es seit den Urzeiten von Unix, schon der betagte Editor *ed* kannte sie. Ihre Fähigkeiten gehen weit über normales Suchen und Ersetzen hinaus: Mit einigen Sonderzeichen sind einzelne Zeichen und Gruppen von ihnen in fast jedem gewünschten Zusammenhang zu finden. Um die Auswertung ("Matchen") kümmert sich ein "RegEx-Maschine" genannter Mechanismus.

Werkzeuge, die reguläre Ausdrücke verwenden, basieren auf einer von zwei, streng genommen drei Varianten dieser Maschinen: DFA (deterministischer finiter Automat), NFA (nicht-deterministischer finiter Automat) und POSIX-NFA. Der DFA betrachtet Zeichen für Zeichen und schreitet fort, wenn es auf den Ausdruck passt. Er markiert keine Zeichen, um zum Prüfen von Alternativen zurückkehren zu können. Eine DFA-basierte RegEx-Maschine kann nicht schon während des Suchprozesses einen Treffer speichern, um ihn zum Ersetzen zu verwenden. Das Verlängern von. "Hund" zu "Hundekuchen" mittels *s/(Hund)/\1ekuchen/* beherrschen DFAs deshalb nicht.

NFAs basieren auf Backtracking

Reguläre Ausdrücke in Perl basieren auf dem NFA, der anders vorgeht: Er merkt sich die Stellen, an denen mehr als eine Möglichkeit zu kontrollieren ist. Stellt er beim Testen einer Variante fest, dass der Gesamtausdruck nicht mehr zutrifft, geht er zurück zum "Scheideweg" und prüft die Alternative. Erst wenn alle abgehakt sind, entscheidet der NFA, ob der Ausdruck zutrifft oder nicht. Durch dieses "Backtracking" genannte Vorgehen beherrscht Perl nummerierte Rückbezüge wie *s/(Ei) (Henne)/\2\1/g*. Hier sorgen die Klammern dafür, dass Perl sich jedes "Ei" und jede "Henne" merkt. Im zweiten Teil vertauscht dann *\2\1* die beiden miteinander.

Perl versucht, den frühesten Treffer zu finden. So matcht es mit */[Ff]isch/* in "Fischers Fritze fischt frische Fische" Fischers, auch wenn fischt ebenso richtig wäre - der erste Treffer. Kommt aber ein Quantifizierer wie * ins Spiel, will der NFA soviel wie möglich finden - er wird gierig ("greedy"). Dabei ist die "Gierigkeit" stärker als die "Links-Bindung".

Will man in HTML-Code einen bestimmten Tag erwischen, heißt der erste Versuch vermutlich: /<.*>/. Übersetzt: "Suche beliebig viele (auch gar kein) Zeichen, umschlossen von spitzen Klammern." Was würde Perl nun in der Zeile

```
<CENTER>
<H2>Heute frische Fische!</H2>
</CENTER>
```

finden? Alles von der ersten spitzen Klammer bis zur letzen hinter *</CENTER>*. Da ein Quantifizierer dabei ist, gilt nicht mehr "Treffer soweit links wie möglich" (also *<CENTER>*) sondern "Soviel wie möglich" - wegen .* die gesamte Zeile. Dieses Verhalten findet man übrigens genauso im *vi* oder in *grep*. Perls Gierigkeit lässt sich jedoch durch ein hinter + oder * gesetztes Fragezeichen beschränken. Benutzt man im obigen Beispiel <.*?>, wird es *<CENTER>* finden. In solchen Fällen hilft ebenfalls das Motto "think negative": */<[^>]+>/*

erledigt das Gewünschte in allen Werkzeugen. Dieser Ausdruck sucht ein <, dann etwas, was kein > ist, davon mindestens eines, schließlich ein >. Ähnlich geht man zum Beispiel vor, um Worte in Anführungszeichen zu finden: */"[^"]+"/* erledigt diesen Job besser als */".*"/*.

Festessen mit Vergiftung?

Perls reguläre Ausdrücke können vorausschauen, ob ein String passen könnte ("lookahead"). Mit

```
/Festessen(?=Champagner)/
```

findet der Interpreter "Festessen" - aber nur, wenn "Champagner" folgt. Schließt sich an "Festessen" jedoch "Lebensmittelvergiftung" an, trifft der Ausdruck nicht mehr zu. Das Ganze darf man auch verneinen: "Finde jedes Festessen, aber nur, wenn dem keine Lebensmittelvergiftung folgt" heißt:

```
/Festessen(?!Lebensmittelvergiftung)/
```

Der Konjunktiv unter den Ausdrücken funktioniert nur, weil der Perl-NFA Backtracking benutzt und illustriert, was es mit dieser Technik auf sich hat. Bei einem *?* zum Beispiel (ein oder kein Zeichen) überprüft Perl zuerst, ob "ein Zeichen" wahr ergeben würde. Wenn nicht, kehrt es zum Scheidepunkt zurück und überprüft, ob "kein Zeichen" den Ausdruck als Ganzes wahr werden ließe. Nach diesem Punkt *ist* der Ausdruck bereits wahr oder falsch. Die Zeichen sind gefressen. Beim Lookahead geht Perl behutsamer vor: Es rennt nicht gleich um den ganzen Häuserblock, stellt dann fest, "hoppla" das passt nicht mehr, also Haus für Haus zurück, sondern lugt erst einmal vorsichtig um die Ecke, ob es sich um das richtige Haus handelt - es verbraucht keine Zeichen. Beim Lookahead wird der gesuchte Ausdruck nicht gesichert, obwohl er durch Klammern begrenzt ist. Seit Perl 5.005 gibt es übrigens auch den Blick zurück, "Look-behind" ist durch (?<) und (?<!) implementiert .

Für nummerierte Rückbezüge kennt Perl zwei Schreibweisen: Die Variablen *$1*, *$2*, *$3* und so weiter enthalten jeweils den Wert des in der ersten, zweiten, dritten … einfangenden Klammer gefundenen Musters. *\1*, *\2* usw. sind Bestandteil der RegEx-Maschine. In ihnen steht ebenfalls der in den ersten, zweiten … einfangenden Klammern gefundene Wert. Die Anzahl der Rückbezüge ist in beiden Fällen unbegrenzt.

Vorsicht ist jedoch geboten: Aufgrund der Art und Weise, wie die RegEx-Maschine den Ausdruck interpoliert und kompiliert, gibt es durchaus einen Unterschied zwischen *$1* und *\1*. Empfohlen wird, innerhalb des Ausdrucks nur den Rückbezug mittels *\1* zu verwenden, etwa

```
/(der|die|das) \1/
```

um doppelte Artikel zu finden. Da eine RegEx-Maschine Variablen wie *$1* zunächst interpoliert und erst dann beginnt, nach Treffern zu suchen und die Klammern zu füllen, enthält *$1* beim ersten Interpretieren eines Ausdrucks noch gar keinen Wert.

```
/(der|die|das) $1/
```

passt also auch auf einzelne Artikel (denn *$1* ist leer) - vorausgesetzt, es gab vorher keine einfangende Klammer, die *$1* mit einem Wert belegt hat. In diesem Fall könnte man den Treffer des vorherigen Ausdrucks wieder finden, wenn dieser sich im gleichen Block befand.

Im Ersetzen-Teil darf *$1* vorkommen, denn dort ist die einfangende Klammer bereits gesucht und bewertet. Für

```
"doppeltes Wort Wort \
finden und löschen"
```

erledigt *s/(\w+) \1)/$1/* das Gewünschte, *s/(\w+)$1/$1/* jedoch in der Regel nicht. *\w+* ist übrigens Perl-ish für "mindestens ein Wortzeichen". Dies sind in der Regel die alphanumerischen Zeichen. Einer Zeichenklasse (etwa *[a-z]*) kann man beliebig Zeichen hinzufügen, die Definition von *\w* steht fest.

Neben *$1, $2* usw. belegt Perl bei jeder RegEx - Auswertung einige Spezialvariablen neu: In *$&* findet sich immer der letzte gültige Treffer, in *$'* alles, was vor ihm und in *$`* das, was nach ihm lag. Wer viel mit Klammerung arbeitet, hat vielleicht noch Verwendung für *$+*: Hier findet sich der Wert der letzten passenden gruppierenden Klammer. Für den Text "der die das" setzt das Muster

```
/(der) (die) (Schüttelreim)?/
```

$+ auf "die". Will man einen Treffer nicht in *$1* oder *$2* speichern, kann man trotzdem Klammern zur Gruppierung verwenden, indem man *(?:Bitte nicht speichern)* verwendet. Rückbezüge sind dann nicht mehr möglich, alle anderen Eigenschaften der Gruppierung bleiben jedoch erhalten. Quantifizierer etwa sind auch mit *(?:)* benutzbar.

To quote or not to quote

Reguläre Ausdrücke leben davon, dass man auseinander halten kann, wann welches Zeichen eine Sonderbedeutung hat und wann nicht. Enthält ein Ausdruck viele *$*, */* oder *()*, die nicht für Zeilenende, Trenner oder Gruppierungen stehen, sondern für sich selbst, müssen sie geschützt (quotiert) werden.

In Einzelfällen genügt dafür ein vorangestellter \. Bei langen Ausdrücken oder vielen Sonderzeichen erlaubt Perl mehrere Varianten des Quotens, beispielsweise die Funktion *quotemeta()*:

```
$string = quotemeta("Schützen Sie \
den \ oder den /,".
"indem Sie einen \ vor den / oder \
den \ stellen. ".
"Benutzen Sie $ für $skalar.")
```

Sonst müsste man vor *$*, */*, \ und den . einen Backslash als Schutz setzen. Innerhalb eines Ausdrucks erreicht man das gleiche mit

```
\Q(<-Beginn des zu quotenden \
Strings - Ende des zu quotenden \
Strings->)\E.
```

Die *()* und die <> verlieren ihre Sonderbedeutung, und Perl behandelt sie als normale Zeichen.

Einige Zeichen ändern ihre Bedeutung allerdings je nach Zusammenhang, zum Beispiel das Leerzeichen. Normalerweise ist es nur es selbst - es sei denn, man verwendet den

Modifizierer *x* am Ende des Ausdrucks. Dadurch sind Leerzeichen und Kommentare erlaubt. Das verbessert vor allem bei komplizierten Ausdrücken die Lesbarkeit:

```
/"      # öffnendes Anführungszeichen
[^"]+   # gefolgt von irgendwas != "
"\x     # gefolgt von schließendem "
```

Wer hier ein Leerzeichen benötigt, muss es mit einem Backslash schützen. Perls *\s* bietet sich nur dann als Ersatz an, wenn es auf die Art des Leerraums nicht ankommt, denn diese Abkürzung steht für jede Art von Leerraum - auch Tabulatoren und Newlines.

Beliebige Begrenzer statt /

Nicht nur Leerzeichen verändern ihre Bedeutung, sondern auch die Begrenzer des Ausdrucks. Oft kombiniert man den Modifizierer *x* mit *m*, um den */* als normales Zeichen gebrauchen zu können. Der Perl-Befehl *m* schaltet explizit die Wahl eines neuen Begrenzers für den Ausdruck ein, er gilt nur für die Suche. Die übliche Schreibweise */Ausdruck/* ist eine Abkürzung für *m/Ausdruck/*.

```
$path =~ m#([^/]*)$#;
$filename = $1;
```

erledigt dasselbe wie

```
$filename = (split("/", $path))[-1];
```

Beides extrahiert den Dateinamen ("alles nach dem letzten / ") aus einer Zeichenkette. *m#* sorgt dafür, dass man */* ohne besondere Vorsichtsmaßnahmen im regulären Ausdruck benutzen kann. Beim Suchen und Ersetzen mit *s* kann ebenfalls ein anderes Zeichen benutzt werden.

Vorsicht: Die Operatoren *s* und *m*, die vor einem Ausdruck stehen (*s///* oder *m//*) sind nicht dasselbe wie die Modifizierer *s* beziehungsweise *m*, die ihm folgen.

Perl unterstützt mit *///s* und *///m* zwei Varianten, wie ein . mit einem *\n* (Newline) umgehen soll und wie sich ^ (Zeilen- oder Stringanfang) und *$* (Zeilen- oder Stringende) verhalten. Diese beiden Modifizierer schalten den Single- beziehungsweise Multiline-Mode ein. Ohne sie trifft der . kein Newline, und *^/$* erreichen den Anfang respektive das Ende der Zeile, letzteres erkennbar durch "\n". Schaltet man nun den normalerweise auf "\n" stehenden Zeilentrenner *$/* zum Beispiel auf "
" um, passt ^ auf den Zeilenanfang und auf *$* "
". Dadurch hat sich nur die Definition von "Zeile" verändert. In Perl kann man eine Datei zeilenweise in ein Array *@datei* einlesen:

```
@datei = <datei.txt>;
```

Hier enthält jedes Element des Array eine durch *$/* definierte Zeile. Alternativ lässt sich eine Datei als ein langer String in einem Skalar speichern:

```
undef $/;
$datei = <datei.txt>
```

Da hier *$/* nicht definiert ist, gibt es keine "Zeile", und Perl liest die ganze Datei in einem Rutsch. In diesem Fall matchen ^ und *$* Stringanfang beziehungsweise -ende.

Newline ist nicht immer Zeilenende

Diese Verarbeitung einer Datei an einem Stück hilft beispielsweise beim Erfassen von Zeilenumbrüchen innerhalb von C-Kommentaren. Für diesen Fall steht der Singleline-Mode bereit, der den zu bearbeitenden Text wie einen langen String behandelt. Die Newlines sind noch vorhanden - aber nun erkennt der . sie. Im Singleline-Mode kann also

```
/\ /\ *.+\*\//s
```

C-Kommentare mit Zeilenumbrüchen finden. In einem langen String (*$/ = undef;*) hätte . ohne *s* die Newlines nicht gefunden.

Gedichte an einem Stück verarbeiten

Im Multiline-Mode ändert sich wiederum das Verhalten von ^ und *$* bezogen auf Zeilenanfänge beziehungsweise -enden; der . findet hier kein Newline. Der Unterschied zwischen Multiline- und normalem Mode offenbart sich, wenn man *$/* verändert. Setzt man es wie oben auf "
", passen ^ und *$* nun auf zweierlei: das durch *$/* neu definierte Zeilenende - aber ebenso logisches Zeilenende und logischen Zeilenanfang. Das *\n* steckt noch in der Datei, deshalb ist der logische Zeilenanfang die Stelle hinter dem *\n* und das logische Zeilenende das Zeichen vor dem Newline.

Newlines verschwinden nicht durch eine Änderung von *$/*, sondern markieren nur nicht mehr das Zeilenende. Wurde die gesamte Datei in einen String eingelesen, finden ^ und *$* im Multiline-Mode auch die logischen "Zeilenenden \n Zeilenanfänge", die sich mitten im Text befinden - und nicht nur den Anfang der Datei beziehungsweise das Ende wie im normalen Modus.

```
Dies sei an dem Beispieltext
Dies ist eine Gedichtzeile\n
und hier ist noch eine Zeile\n
und hier noch eine\n
und alles ohne Satzzeichen\n
```

erläutert (*\n* ist hier nur der Deutlichkeit halber gezeigt), der als langer String in einem Skalar enthalten ist. Benutzt man nun den Multiline-Ausdruck

```
s/\w+$/m
```

findet Perl "Gedichtzeile", denn *$* erkennt auch logische Zeilenenden. Sucht man stattdessen

```
s/\w+$/s
```

heißt der Treffer "Satzzeichen", denn *$* bezieht sich nun auf das Ende der gesamten "Datei".

Funktionen in Ausdrücken

Perl erlaubt in seinen regulären Ausdrücken "Formulierungen", die einen korrekten String ergeben. Liefert beispielsweise eine Funktion einen String zurück, darf man sie im Ersetzungsteil eines Ausdrucks verwenden, wenn man den Modifizierer *e* benutzt. Er sorgt dafür, dass die RegEx-Maschine alle Variablen interpoliert, den Ausdruck übersetzt, *\1* usw.

belegt und schließlich den Ersetzungsteil evaluiert. Man verwendet den *e*-Modifizierer zum Beispiel so:

```
$system = "Ich benutze Windows.";
$system =~ s/[Ww]indows/&os/eg;
print "$system\n";
sub os {
$string = `uname -a`;
return $1 if ($string =~ /(^\w+)/)
}
```

s/[Ww]indows/&os/eg ersetzt jedes "Windows" und "windows" durch den Rückgabewert von *os()*. Diese Routine wiederum liefert das erste von *`uname -a`* Zurückgelieferte Wort, so dass auf der Standardausgabe der Name des benutzten Betriebssystems erscheint - vorausgesetzt, *uname* ist vorhanden.

Literatur

[1] Jeffrey Friedl; Reguläre Ausdrücke; O'Reilly GmbH, Köln 1997, ISBN: 3-930673-62-2

[2] Kurzer Überblick von Tom Christiansen