Zusammenfassung

Regular Expressions

Datum: 03.01.2005

von Christoph Moder
(© 2005)
http://www.skriptweb.de

Hinweise (z.B. auf Fehler) bitte per eMail an mich: cm@skriptweb.de – Vielen Dank.

Inhaltsverzeichnis

Vorwort

Der Inhalt dieser Zusammenfassung stammt v.a. aus dem Buch *Mastering Regular Expressions* von Jeffrey E. F. Friedl (1. Auflage 1997, O'Reilly-Verlag). Wer sich intensiver mit dem Thema befassen will und sich speziell für Perl interessiert, dem sei dieses Buch wärmstens empfohlen. Außerdem stammen einige Ideen und Beispiele aus dem LRZ-Kurs *Reguläre Sprachen, reguläre Ausdrücke* von Helmut Richter (siehe Literaturliste).

Geschichtliche Entwicklung

Die Grundlagen gehen auf Arbeiten der Neurophysiologen Warren McCulloch und Walter Pitts zurück, die Anfang der vierziger Jahre des 20. Jahrhunderts Modelle über die Funktionsweise des Nervensystems entwickelten. Der Mathematiker Stephen Kleene beschrieb diese Modelle einige Jahre später mit einer Algebra, die er *regular sets* nannte. Um diese zu beschreiben, entwickelte er eine Notation, die er als *regular expressions* bezeichnete. In den fünfziger und sechziger Jahren beschäftigten sich weitere Mathematiker mit diesen Modellen, gegen Mitte der sechziger Jahre tauchten die Regular Expressions zum ersten Mal im Zusammenhang mit Computern auf. Ken Thomson beschrieb sie 1968 in Zusammenhang mit einem Suchalgorithmus; später entwickelte er den Editor *qed*, eine Vorstufe des Unix-Editors *ed*. Damit wurden Regular Expressions (kurz „Regex“) zum ersten Mal auf breiter Front eingesetzt. Der ed-Befehl `g/Regular Expression/p` („Global, Regular Expression, Print“ – suche in allen Zeilen und gibt das Gefundene aus) verselbständigte sich später in ein eigenes Programm – *grep* war geboren.

Heute gibt es eine ganze Menge verschiedener Programme, die Regular Expressions unterstützen. Leider verhalten sie sich alles andere als konsistent. Beispielsweise nahm man früher für die Klammerung bei Backreferences `\(`, weil man dachte, dass Regular Expressions v.a. für Programmcode, der viele Klammern enthält, verwendet würde. Spätere Implementationen verzichteten auf den Backslash bei den Klammern (mit dem Ergebnis, dass man dort Klammern im Suchtext per Backslash schützen muss). Aber viel ärgerlicher als diese systematische Inkonsistenz sind kleine Details und Fehler, die oft schlecht dokumentiert sind. Denn die ersten Programme mit Regex-Unterstützung benutzten dafür keine gemeinsame Library, sondern jedes Programm verwendete eigenen Code. Zudem gibt es die Programme für verschiedene Unix-Versionen, zwischen denen sie oft nicht portiert, sondern einfach nachprogrammiert wurden (die verschiedenen Unix-Implementationen stammen schließlich von verschiedenen Firmen). Die verschiedenen Versionen des selben Programms verwenden also oft ganz verschiedenen Code und verhalten sich deshalb teilweise unterschiedlich und haben unterschiedliche Bugs und Eigenheiten.

Zurück zur Geschichte. Eine große Erweiterungen erfuhren die Regular Expressions durch das von Alfred Aho geschriebene Programm *egrep*, mit dem die sogenannten „extended regular expressions“ (ERE) eingeführt wurden. (Die alte Version nennt man „basic regular expressions“ = BRE.) Einer der auffälligsten Unterschiede zwischen BRE und ERE ist, dass bei ERE Dinge wie Klammern (rund wie geschweift) ohne Backslash geschrieben werden, es gibt jedoch noch diverse sinnvolle Verbesserungen bei den EREs (z.B. Frage- und Pluszeichen als Quantifizierer, Alternation sowie Anwendbarkeit der Quantifizierer auf geklammerte Unterausdrücke). (Die Unterscheidung in BRE und ERE ist ziemlich oberflächlich, weil sich die von den Programmen verwendete Syntax keineswegs in zwei scharf getrennte Gruppen unterteilen lässt. Trotzdem kann man durch die Information, ob das Programm BRE oder ERE verwendet, schon einige Schlüsse auf die Syntax ziehen.) 1986 brachte dann Henry Spencer ein in C geschriebenes Regex-Paket heraus, das jeder für seinen eigenen Code verwenden durfte. Alle darauf aufbauenden Programme (und davon gibt es etliche) bieten entsprechend eine einheitliche Regex-Unterstützung.

Im Laufe der Zeit wurden die Regex-Engines nicht nur um ein paar Features bereichert, sondern insgesamt deutlich komplexer. Während die Regex-Engine von *ed* aus dem Jahr 1979 nur ca. 350 Zeilen Programmcode umfasste, erstreckt sich der Code des Posix-NFA-Pakets (1992) über rund 9700 Zeilen.

Komponenten einer Regular Expression

Regular Expressions beinhalten folgende Elemente:

- **normalen Text** (Regex-Metazeichen müssen durch Backslash geschützt werden)
- **Textanker**, die Positionen im Text markieren (z.B. Wortanfang, Zeilenende usw.)
- **Quantifizierer**, die angeben, wie oft das entsprechende Zeichen bzw. der Textteil vorkommen darf/muss
- **Zeichenklassen**, die eine Menge erlaubter Buchstaben ausdrücken
- **Gruppierungen**, die mehrere Zeichen oder sogar ganze Abschnitte einer Regular Expression klammern
- **Alternations**, die verschiedene erlaubte Sequenzen anbieten (von denen nur eine auf den Text passen muss)

Dazu kommen – je nach Implementation – auch noch Backslash-Sequenzen, mit denen man ASCII-Sonderzeichen ausdrücken kann, z.B. „\n“ für Zeilenumbruch (Line Feed, ASCII Nr. 10). Sie erleichtern die Eingabe dieser Sonderzeichen, haben mit der Regular Expression an sich jedoch nichts zu tun.

Normaler Text

Eine Regex-Suche, die nur normalen Text (Fachbegriff: „terminale Zeichen“) enthält, verhält sich nicht anders als die Textsuche mit einer gewöhnlichen Suchfunktion ohne Unterstützung für Regular Expressions.

Folgende Zeichen sind Regex-Metazeichen und müssen durch einen Backslash geschützt werden (bezogen auf ERE): '.', '*', '+', '?', '(', ')', '[', ']', '\', '^', '$', '|'.

Um nach speziellen Zeichen suchen zu können, die man nicht mit der Tastatur eingeben kann, gibt es die Backslash-Sequenzen. In der Regex-Suche verhalten sich diese Zeichen wie normaler Text (haben also keine Meta-Bedeutung), jedoch gibt es an manchen Stellen Verwechslungen mit Regex-Meta-Elementen – speziell, wenn sie ebenfalls den Backslash verwenden.

Backslash-Escapes

Folgende Backslash-Escapes als Ersatz für Steuerzeichen sind üblich:

- \a: Alert (BEL, 0x07)
- \b: Backspace (BS, 0x08)
- \e: Escape (ESC, 0x1B)
- \f: Form Feed (FF, 0x0C)
- \n: Line Feed (LF, 0x0A)
- \r: Carriage Return (CR, 0x0D)
- \t: Tab (HT, 0x09)
- \v: vertical Tab (VT, 0x0B)

ASCII-Sonderzeichen oktal und hexadezimal

Wenn man ein Zeichen nicht mit der Tastatur eingeben kann, kann man meist in der Regex dessen

ASCII-Wert oktal oder hexadezimal angeben:

- oktal: \0 + drei Ziffern (z.B. `\0101`)
  Das Verhalten bei Werten größer als `\0377` (= 255 im Dezimalsystem) ist uneinheitlich! Manche Programme machen einen „wrap around“ (d.h. nehmen den Rest der Division durch 255), manche funktionieren auch bei größeren Werten anstandslos, und manche interpretieren sogar die im Oktalsystem nicht vorkommende Ziffer "9" als `\011`.
- hex: \x + zwei Ziffern (z.B. `\x41`)
  Das Verhalten der Programme ist ebenso uneinheitlich, z.B. kann bei `\x2Ecom` als Hexadezimalzahl 2E oder 2EC angenommen werden (C ist eine gültige Hexadezimalziffer); in letzterem Fall ist außerdem nicht eindeutig, wie diese Zahl interpretiert wird (ebenso wie bei den Oktalzahlen).

Zu beachten bei Backslashes

- Bei GNU-awk und MKS-awk werden auch durch eine Backslash-Sequenz angegebene Zeichen als Meta-Zeichen interpretiert, d.h. `[+\055*]` wird als gleichwertig zu `[+-*]` (Strich als Bereichsangabe) betrachtet. Die Backslash-Sequenzen werden also in normale Zeichen umgewandelt, bevor die Regex an die Regex-Engine weitergereicht wird.
- Bei vielen Programmen werden die Backslash-Sequenzen zwar nicht in normale Zeichen umgewandelt, jedoch ebenso vor der Regex-Engine interpretiert. Das bedeutet, dass man eine Wortgrenze, die in der Regex als `\b` ankommen soll, als `\\b` schreiben muss, weil `\b` das Backspace-Zeichen meint. Der Backslash muss also `\\` geschrieben werden; entsprechend muss ein Backslash, der auch als solcher in einer Regex landen soll, zweimal kodiert werden, also `\\\\` (Beispiel Suche nach einem Windows-Verzeichnispfad: `C:\\\\Windows\\\\`).
- Emacs ist ein besonderer Härtefall, denn er entfernt auch Backslashes, die als Backslash-Escape-Sequenz keinen Sinn ergeben. Darum muss man z.B. in einem String `\\1` schreiben, um in der Regex dann `\1` zu erhalten.
  Python dagegen leitet alles unverändert weiter, was als Backslash-Escape im String keinen Sinn macht, d.h. `\x41` kann man genau so schreiben, aber `\v12` muss zu `\\v12` gemacht werden, da `\v` das Vertical-Tab-Zeichen meint.

Textanker

Textanker bezeichnen keine Buchstaben bzw. Zeichen, sondern Positionen im Text.

- `^` bezeichnet den Anfang des Texts, `$` das Ende. Je nachdem wie das Programm den Text verarbeitet, ist es unterschiedlich, was als Anfang oder Ende betrachtet wird. Entweder ist es der Anfang und das Ende der Datei, oder bei zeilenweise arbeitenden Programmen der Anfang sowie das Ende der Zeile. Beispiel: `^abc` findet `abc` nur, wenn es am Zeilen-/Textanfang steht.
- In manchem Programmen bezeichnen `\<` den Wortanfang und `\>` das Wortende. Beispielsweise passt die Regex `\<cat\>` dann auf den Text `the cat sits on the tree`, aber nicht auf `he is on vacation`.
- Manche Programme beschreiben statt dessen die Wortgrenzen mit `\b` (egal ob Anfang oder Ende), und mit `\B` das Gegenteil, nämlich jede Position, die keine Wortgrenze ist.

Bei den Wortgrenzen-Anker tritt schnell die Frage auf, was eigentlich ein Wort ist. Grundsätzlich ist es eine Menge von Buchstaben, die von Leerzeichen oder anderen Trennzeichen (z.B. Satzzeichen) umgeben ist; aber der Knackpunkt ist, welche Sonderzeichen man zum Wort zählt und welche als

Trennzeichen gelten. Typischerweise besteht ein Wort aus Buchstaben und Ziffern, manchmal ist auch noch der Unterstrich '_' dabei. Welche Buchstaben als Wortbestandteile akzeptiert werden, hängt oft von den Locale-Einstellungen ab.

Quantifizierer

Ein Quantifizierer bezeichnet die Anzahl der Wiederholungen und bezieht sich immer nur auf das unmittelbar vor ihm stehende Zeichen. Ausnahme: Wenn zuvor eine Gruppierung steht (in runden Klammern), dann bezieht er sich auf die gesamte Gruppierung.

Schreibweise:

- In geschweiften Klammern steht die Anzahl des Auftretens. Beispiel: `a{5}` entspricht `aaaaa`.
- Zwei Zahlen in geschweiften Klammern geben Mindest- und Höchstanzahl an: `a{2,4}` erlaubt `aa`, `aaa` oder `aaaa`. Man kann jede der beiden Angaben weglassen, sie werden durch Null bzw. unendlich ersetzt. `a{,5}` bedeutet dann „höchstens fünfmal“ und `a{5,}` bezeichnet „mindestens fünfmal“.
- Das Sternchen '*' bedeutet „keinmal oder beliebig oft“ (also `a{0,}`); Beispiel: `a*`.
- Das Pluszeichen '+' bedeutet „einmal oder beliebig oft“ (also `a{1,}`); Beispiel: `a+`.
- Das Fragezeichen '?' bedeutet „keinmal oder einmal“ (also `a{0,1}`); Beispiel: `a?`.

Besonderheiten:

- `+` und `?` sind nicht in allen Regex-Implementierungen vorhanden, sondern nur bei ERE.
- In manchen Implementierungen werden die geschweiften Klammern, das Pluszeichen und das Fragezeichen mit Backslash davor geschrieben. Einfach ausprobieren.
- Quantifizierer können bei manchen Programmen nicht nur hinter einzelnen Zeichen bzw. Zeichenklassen und Klammern stehen, sondern sogar hinter Backreferences.
- Die Schreibweise `{n}` für Quantifizierer wird nicht von allen Programmen unterstützt, dort muss man dann `{n,n}` schreiben.
- Der eigentlich unsinnige Quantifizierer `{0,0}` ist bei den meisten Programmen gleich dem '*' (z.B. bei *awk*, *grep*), z.B. bei *sed* jedoch gleich '?'.
  Der Sinn von solchen „unsinnigen“ Konstruktionen: Man könnte damit irgend etwas Spezielles ausdrücken; Larry Wall, der Erfinder von *Perl*, sagt dazu: „*It's like having a zero in your numbering system.*“

Greediness

Quantifizierer, die keine konkrete Anzahl enthalten (z.B. `*`, `+`, `?`), sind *gierig* (engl. *greedy*) – d.h. sie nehmen sich so viele Zeichen wie möglich. Wenn beispielsweise der Suchtext lautet `hier ist "ein Text" mit "vier Anführungszeichen" darin`, dann findet eine Suche mit `".*"` nicht etwa `"ein Text"`, sondern `"ein Text" mit "vier Anführungszeichen"` – denn der Punkt (s.u.) in der Regex passt ebenso auf Anführungszeichen. Dieses Verhalten kann trickreich sein, wenn man nicht weiß, wie man damit umgehen soll. Die Lösung in solchen Fällen ist allerdings auch nahe liegend (wenn auch nicht immer ganz einfach umzusetzen): statt eines Punktes müsste man hier ein Zeichen nehmen, das auf alle Zeichen *außer* dem Anführungszeichen passt.

Nur Perl bietet als Ausweg von jedem Quantifizierer eine zweite Form an, die *non-greedy* ist; hier würde man mit `".*?"` suchen (der Quantifizierer lautet `*?`) und bekäme `"ein Text"`.

Zeichenklassen

Grundlegendes

Eine Zeichenklasse besteht aus eckigen Klammern, zwischen denen die erlaubten Buchstaben stehen. Die Reihenfolge ist dabei egal – `[abcdef]` und `[bdecaf]` sind gleichwertig.

Mit dem Bindestrich '-' kann man Bereiche angeben, falls der Bindestrich zwischen zwei Zeichen steht. `[A-D]` meint beispielsweise die Buchstaben 'A', 'B', 'C' und 'D', und `[0-9]` bezeichnet die Dezimalziffern. Wie die Buchstaben sortiert sind, hängt dabei von der Locale-Einstellung ab (LC_COLLATE) – beispielsweise könnten die Umlaute zu den dazugehörigen Vokalen sortiert werden, statt laut ihres ASCII-Codes ganz an das Ende des Alphabets.
Wenn der Bindestrich an einer anderen Position steht, verliert er seinen Sonderstatus – `[AD-]` meint die Zeichen 'A', 'D' und '-'.

Mit dem Caret-Zeichen '^' negiert man den Inhalt, wenn es als erstes Zeichen in der Menge steht. Beispielsweise meint `[^abc]` alle Zeichen außer 'a', 'b' und 'c'. Wenn das Caret an einer anderen Position steht, verliert es ebenfalls seinen Sonderstatus – `[a^bc]` meint die vier Zeichen 'a', 'b', 'c' und '^'. Eine Negation bedeutet also immer „ein Zeichen ungleich den angegebenen Zeichen“, d.h. es muss ein Zeichen vorhanden sein. Beispielsweise findet die Suche mit `q[^u]` nicht den Begriff „Iraq“, wenn dieser am Ende des Textes steht – denn hinter dem 'q' kommt nicht nur kein 'u', sondern überhaupt kein Zeichen mehr.

Eine Zeichenklasse verhält sich also wie ein einzelner Buchstabe – `abc` entspricht `a[b]c` oder `[a][b][c]`. Umgekehrt ist auch ein Buchstabe eine Zeichenklasse mit genau einem Element. Das bedeutet, dass man Quantifizierer auch mit Zeichenklassen benutzen kann, beispielsweise sucht `[0-9]{3}` nach dreistelligen Zahlen.

Damit keine Verwirrung aufkommt: Innerhalb der eckigen Klammern von Zeichenklassen verlieren die Regex-Metazeichen ihre Funktion (darum muss man Regex-Metazeichen nicht unbedingt mit dem Backslash schützen, man kann sie auch einfach in eckige Klammern einpacken), und andere Zeichen werden zu Metazeichen oder ändern ihre Bedeutung. Regex-Metazeichen haben nichts mit Zeichenklassen-Metazeichen zu tun! Beispielsweise hat der Punkt '.' oder das Sternchen '*' innerhalb einer Zeichenklasse keine spezielle Bedeutung, dafür jedoch der (außerhalb unverfängliche) Bindestrich, und das Caret ändert seine Bedeutung – vom Zeilenanfangs-Anker zum Negierungszeichen. Weil man bei den meisten Programmen innerhalb von Zeichenklassen die Zeichenklassen-Metazeichen nicht durch Backslash schützen kann, muss man beispielsweise, um die Zeichen 'A', 'D' und '-' zu erhalten, `[AD-]` statt `[A\-D]` schreiben. Neben Bindestrich und Caret ist die schließende eckige Klammer das dritte Zeichenklassen-Metazeichen; außerdem kann auch die öffnende eckige Klammer, falls dahinter ein Punkt, Doppelpunkt oder Gleichheitszeichen folgt (siehe unten), ein Metazeichen sein. Wenn diese Zeichen in einer Zeichenklasse vorkommen sollen, empfiehlt sich folgende Regelung:

- Der Bindestrich `-` kommt ans Ende der Zeichenklasse.
- Die öffnende eckige Klammer `[` kommt ebenfalls an das Ende, aber vor den Bindestrich.
- Die schließende eckige Klammer `]` kommt an den Anfang der Zeichenklasse (aber hinter die evtl. vorhandene Verneinung mit `^`).
- Das Caret `^` kommt an eine beliebige Stelle, aber nicht ganz an den Anfang, weil es dort als Verneinung interpretiert würde.

Zusammengefasst würde eine Zeichenklasse, mit der man nach Bindestrich, eckigen Klammern und

Caret suchen kann, so aussehen: `[]^[-]`
Und das Gegenteil davon: `[^]^[-]`

Bei Zeichenklassen erfolgt der Vergleich übrigens Byte-weise, d.h. man kann mit einer Zeichenklasse aus zwei ASCII-Zeichen nach einem entsprechenden Zwei-Byte-Unicode-Buchstaben suchen.

Der Punkt

Eine besondere Zeichenklasse ist der Punkt; er hat die Bedeutung „ein beliebiges Zeichen".
Beispiel: `a..b` findet jedes vierbuchstabige Wort, das mit 'a' beginnt und mit 'b' aufhört – die beiden Buchstaben dazwischen sind beliebig.

Weil der Punkt ein Meta-Zeichen ist, muss man, wenn man im Text nach einen Punkt sucht, diesen per Backslash schützen.
Beispiel: `[0-9]{2}\.[0-9]{2}\.` (um nach zweistelligen Datumsangaben zu suchen)

Achtung: Obwohl der Punkt theoretisch für ein beliebiges Zeichen steht, zählt je nach Programm das Newline-Zeichen und/oder das Null-Byte nicht dazu. Im Gegensatz dazu schließen Zeichenklassen wie `[^e]` das Newline-Zeichen meist ein – d.h. es kann passieren, dass `.*` nur bis zum Ende der Zeile geht, während `[^x]*`, falls kein 'x' im Text vorhanden ist, über den Zeilenumbruch hinaus geht. Will man wirklich *alle* Zeichen einschließen, kann eine Zeichenklasse wie `[\000-\377]` das Richtige sein.

Posix-Zeichenmengen

Es gibt eine Reihe vordefinierter Zeichenmengen, die durch einen Namen, eingeschlossen in eckigen Klammern und Doppelpunkten, bezeichnet werden:

Zeichenmenge	*Bedeutung*	*Äquivalent*
`[:alnum:]`	Buchstaben und Ziffern	`[a-zA-Z0-9]`
`[:alpha:]`	Buchstaben	`[a-zA-Z]`
`[:blank:]`	Leerzeichen und Tabulator	`[ \t]`
`[:cntrl:]`	ASCII-Sonderzeichen	
`[:digit:]`	Ziffern	`[0-9]`
`[:graph:]`	darstellbare Zeichen, d.h. alles außer Sonderzeichen und Abstandszeichen	`[^[:space:] [:cntrl:]]`
`[:lower:]`	Kleinbuchstaben	`[a-z]`
`[:print:]`	druckbare Zeichen: darstellbare Zeichen plus das Leerzeichen	`[[:graph:] ]`
`[:punct:]`	Zeichensetzungszeichen	
`[:space:]`	Abstandszeichen: Leerzeichen, horizontaler und vertikaler Tabulator, Form Feed, Wagenrücklauf, Zeilenvorschub	`[[:blank:]\f\n\r\v]`
`[:upper:]`	Großbuchstaben	`[A-Z]`
`[:xdigit:]`	Hexadezimalziffern	`[0-9a-fA-F]`

Achtung: Die Schreibweise der Posix-Zeichenmengen in eckigen Klammern hat nichts mit den eckigen Klammern der Zeichenklassen zu tun. Um eine Posix-Zeichenmenge in eine Zeichenklasse aufzunehmen, muss man sie mit in deren eckige Klammern einschließen. Wenn man eine

Zeichenklasse mit den Buchstaben 'a' und 'b' macht, lautet sie `[ab]` – analog lautet eine Zeichenklasse aus Großbuchstaben und Zeichensetzungszeichen `[[:upper:][:punct:]]`.

Zu beachten ist, dass die Zeichenklassen `[:alpha:]`, `[:alnum:]`, `[:upper:]` und `[:lower:]` von den Locale-Einstellungen des Rechners abhängig sind, also in den unterschiedlichen Sprachen verschieden sind. Beispielsweise würde in einer deutschen Locale-Umgebung die Menge `[:upper:]` nicht nur A bis Z, sondern auch Ä, Ö und Ü umfassen.

Die Posix-Zeichenmengen werden von allen Posix-Programmen unterstützt. Aber auch manche nicht-Posix-Programme unterstützen diese Klassen, interessanterweise GNU-egrep jedoch nicht.

Andere Zeichenmengen

Die folgenden Zeichenmengen werden nur von bestimmten Programmen unterstützt:

- `\w`: Wortbestandteil, meist `[a-zA-Z0-9]`, manchmal ist auch der Underscore '_' dabei
- `\W`: das Gegenteil von `\w`
- `\d`: Ziffern `[0-9]`, also identisch mit `[:digit:]`
- `\D`: das Gegenteil von `\d`, also `[^0-9]`
- `\s`: Abstandszeichen, identisch mit `[:space:]`
- `\S`: das Gegenteil von `\s`
- Emacs bietet eine Spezial-Syntax: Hinter '\s' wird der gewünschte Zeichentyp geschrieben, z.B. bedeutet `\sw` = ein Zeichen aus der Menge `\w` (d.h. `[a-zA-Z0-9]`), und `\s ` = ein Zeichen aus der Menge der Leerzeichen.

Collating Sequences

Collating Sequences beschreiben zusammengesetzte Buchstaben. Beispielsweise zählt im Spanischen das „ll“ wie ein einzelner Buchstabe und hat einen eigenen Platz im Alphabet (zwischen l und m), ebenso ist im Niederländischen das „ij“ ein einzelner Buchstabe (darum schreibt man „IJsselmeer“ statt „Ijsselmeer“). Das scharfe S im Deutschen ist der umgekehrte Fall: es wird als ein Buchstabe geschrieben, aber im Alphabet sortiert wie „ss“.

In einer Regex kann man diese Buchstaben-Grenzfälle mit collating sequences angeben: `[.span-ll.]` (z.B. `torti[[.span-ll.]a`) oder `[.eszet.]`. Das Verhalten wird außerdem von der Locale-Einstellung beeinflusst; so findet beispielsweise im Spanischen eine Suche nach `[a-z]` (ein Buchstabe!) den Text `ll`.

Equivalence Classes

Um nicht nur einen bestimmten Buchstaben, sondern alle dazu ähnlichen Buchstaben zu bezeichnen, gibt es equivalence classes – ausgedrückt durch eckige Klammern und Gleichheitszeichen. Beispielsweise würde man die Menge der Zeichen, die ähnlich wie das 'a' sind, mit `[=a=]` bezeichnen, was `[aäàáâ]` entsprechen könnte. Das genaue Verhalten ist dabei ebenfalls von der Locale abhängig.

Gruppierungen

Mit runden Klammern gruppiert man einen Ausdruck – entweder, um Quantifizierer auf einen ganzen Regex-Teil statt nur auf ein einzelnes Zeichen anzuwenden (d.h. um mehrere Zeichen zusammenzufassen; z.B. `(abc){3}` bedeutet `abcabcabc`), oder um Alternations auf einen Teil der Regex zu beschränken (d.h. um Zeichenmengen zu trennen), oder damit man später per Rückbezug

auf den Inhalt zugreifen kann (d.h. um den Inhalt zu speichern).

Alternations

Mit dem Pipe-Zeichen '|' (manchmal mit Backslash davor) trennt man verschiedene Alternativen. Beispiel: `Peter|Paul` (erlaubt sowohl „Peter“ als auch „Paul“), oder `(Fahr|Flug)zeug` (erlaubt „Fahrzeug“ oder „Flugzeug“), oder `Tick|Trick|Track` (erlaubt „Tick“ oder „Trick“ oder „Track“; könnte man auch schreiben als `T(i|ri|ra)ck`).

Das Pipe-Zeichen hat eine sehr niedrige Priorität, d.h. die Alternativen reichen bis zum nächsten Pipe-Zeichen, bis zur nächsten Klammer oder bis zum Ende der Regex. Beispiel: `dies und|oder das` entspricht `(dies und)|(oder das)`, nicht `dies (und|oder) das`. Eine niedrigere Priorität als das Pipe-Zeichen haben nur die Anker '^' und '$', beispielsweise ist `^dies|das$` gleichwertig zu `^(dies|das)$` (außer bei *lex*).

Auf den ersten Blick erscheint Alternation als ein ähnliches Konzept wie eine Zeichenklasse. Tatsächlich hat `(a|e)` die gleiche Wirkung wie `[ae]`, aber bereits `(Bob|Robert)` lässt sich nicht mehr durch eine Zeichenklasse darstellen. Konkret: Eine Zeichenklasse bietet für ein einziges Zeichen mehrere Alternativen, während Alternation komplette Ausdrücke unterscheidet, die alle Metazeichen, Zeichenklassen, Quantifizierer und sogar Klammern mit eingeschlossenen Alternations enthalten dürfen.

Die leere Alternative wie z.B. bei `(this|that|)` ist laut Posix erlaubt und sollte gleichwertig zu `(this¦that)?` sein, aber nicht alle Programme erlauben es (z.B. lex und manche awks verbieten es) bzw. ihr Verhalten ist uneinheitlich.

Rückbezüge

Bei einem Rückbezug fügt man den Inhalt einer Klammer an einer anderen Stelle wieder ein. Typischerweise ist das bei einer Ersetzung der Fall:
`/a href="([^"]*)"/URL: \1/`: Hier wird eine URL aus einem HTML-Tag extrahiert; der Inhalt der Klammer wird an der Stelle eingefügt, an der die `\1` steht. Man kann einen Rückbezug auch mehrmals einfügen; das umgekehrte Beispiel, das aus einer URL einen HTML-Link macht (und die URL sowohl in den Link als auch in den Linktext schreibt), würde also lauten:
`/(http:[^[:space:]]*)[[:space:]]/<a href="\1">\1<\/a>/`

Rückbezüge machen aber nicht nur bei Ersetzungen Sinn; beispielsweise kann man mit folgender Konstruktion nach drei gleichen Buchstaben hintereinander suchen – vollkommen egal, welche Buchstaben das sind:
`([a-zA-Z])\1\1`

Man kann nicht nur einen Rückbezug verwenden, sondern mehrere – bei den meisten Programmen gibt es Rückbezüge von \1 bis \9. Dabei bezieht sich \1 auf die erste Klammer, \2 auf die zweite Klammer usw. Weil man Klammern auch verschachteln kann, zählt hier immer die Position der *öffnenden* Klammer von links nach rechts – eine innere Klammer hat also grundsätzlich eine höhere Nummer als die sie umgebende äußere Klammer.

Bei manchen Programmen kann man die Rückbezüge auch außerhalb der Regex verwenden; beispielsweise in Perl stehen sie in Form der Variablen `$1` bis `$9` zur Verfügung, bei Python greift man mit einer Konstruktion wie `MyRegex.group(1)` darauf zu. Manche Programme wie Emacs, Tcl oder Python speichern nicht nur die Rückbezüge selbst für die Benutzung außerhalb, sondern auch noch die Positionen, an denen sie im Text aufgetreten sind.

Rückbezüge sind prinzipiell nur bei NFA-Engines (s.u.) möglich, außerdem gehören sie zum

Standardrepertoire von BREs (basic regular expressions), nicht jedoch von EREs (extended regular expressions). Grund: Bei BREs gibt es keine Alternation; beispielsweise ist bei `(ab|c(a*))d\2` nicht klar, was der Inhalt von \2 ist, wenn die erste der beiden Alternativen genommen wurde, in der es eine innere Klammer gar nicht gibt.

Zusammenfassung der Regex-Bestandteile

Beispiel-Regex

```
(^¦[[:space:]]).{2}\.(foo\\bar){1,3}\b\x41\1.\0101\d+\.*\d?($¦[^a-zA-Z])
```

Farbgebung:

- normaler Text
- Textanker
- Quantifizierer
- Zeichenklassen
- Gruppierung
- Alternation
- Rückbezug

Vorfahrtsregeln

- Die eckigen Klammern binden stärker als die runden.
- Stern, Pluszeichen und Fragezeichen binden stärker als Hintereinanderschreiben, d.h. `ab*` bedeutet `a(b*)` und nicht `(ab)*`.
- Der senkrechte Strich bindet schwächer als Hintereinanderschreiben, d.h. `ab|c` bedeutet `(ab)|c` und nicht `a(b|c)`. Mehrere Sterne, Pluszeichen oder Fragezeichen oder aus diesen Zeichen unmittelbar zusammengesetzte Kombinationen dürfen nicht vorkommen

Endliche Automaten

Siehe *http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/*

Theoretisches Konzept

In der theoretischen Informatik beschreibt die Berechenbarkeitstheorie, in welche Gruppen zu berechnende Probleme eingeteilt werden können und welche Eigenschaften Maschinen haben müssen, um die jeweiligen Problemgruppen bearbeiten zu können. Eines dieser theoretischen Konzepte ist der endliche Automat, der eine begrenzte Anzahl von Zuständen besitzt (daher der Name „endlich“) und nach jedem Eingabesymbol den Zustand wechseln kann. Der dabei zur Verfügung stehende Speicher bestimmt die Fähigkeiten der Maschine. Ist er unendlich groß, handelt es sich um eine Turingmaschine; ist er proportional der Eingabedatenmenge, kann die Maschine kontextsensitive Sprachen bearbeiten; ist er beschränkt, kann die Maschine nur kontextfreie Sprachen bearbeiten; und ist überhaupt kein Speicher vorhanden (im Sinne einer „Zwischenablage“, d.h. um Daten von einem Rechenschritt in den nächsten zu übertragen), kann der endliche Automat nur Sprachen bearbeiten, die durch reguläre Ausdrücke beschrieben werden. Das bedeutet also, dass ein Programm zur Verarbeitung von Regular Expressions (in der Bedeutung von regulären Ausdrücken, die von *grep*, *sed* usw. zur Bearbeitung von Texten verwendet werden) wie ein endlicher Automat funktionieren muss.

Deterministische endliche Automaten

Im folgenden Beispiel soll getestet werden, ob eine Zahl durch drei teilbar ist. Dazu konstruiert man einen Automaten mit drei Zuständen, die '0', '1' und '2' heißen. Dann legt man folgende Übergänge fest: Wenn die eingelesene Ziffer durch drei teilbar ist (d.h. der Rest der Division ist 0), bleibt der Automat in seinem Zustand. Ist der Rest der Division gleich 1, so wechselt er von '0' auf '1', von '1' auf '2' oder von '2' auf '0'. Ist der Rest dagegen gleich 2, so wechselt er von '0' auf '2', von '2' auf '1' oder von '1' auf '0'. Wir haben also zwischen den drei Zuständen neun Übergänge (incl. der Übergänge, die in den selben Zustand zurückkehren), weil von jedem Zustand aus drei mögliche Eingaben akzeptiert werden (Rest 0, 1 oder 2). Der Anfangszustand ist '0'. Jetzt kann man eine beliebig große Zahl nehmen und diese ziffernweise von links nach rechts an den Automaten verfüttern.

Beispiel: Die Zahl „651“. Anfangs ist der Automat im Zustand '0', die erste Ziffer ist die Sechs, 6 mod 3 = 0 => bleibt im Zustand '0'. Die zweite Ziffer ist die Fünf, 5 mod 3 = 2 => wechselt in den Zustand '2'. Die dritte Ziffer ist die Eins, 1 mod 3 = 1 => wechselt in den Zustand '0'. Damit ist die Zahl zu Ende, und aus dem Endzustand '0' wissen wir, dass die Zahl durch drei teilbar ist. Wäre der Endzustand '1' bzw. '2', hieße das, dass nach der Division ein Rest von 1 oder 2 bleiben würde.

Hier sieht man sehr schön den Vorteil des Verfahrens: Der Automat braucht keinen zusätzlichen Speicher, es gibt weder dynamische Variablen noch rekursive Aufrufe, sondern der Speicherbedarf ist konstant und steht vorher fest – trotzdem darf die zu testende Zahl beliebig groß sein.

Zweites Beispiel: Addition von Zahlen. Bei einer binären Addition braucht man neben dem Anfangszustand nur einen weiteren Zustand, der einen Übertrag signalisiert. Es gibt dann 8 Übergänge (vier Kombinationen von zwei Binärziffern sind möglich, jeder Übergang von beiden Zuständen aus), bei jedem Übergang wird eine Binärziffer ausgegeben.

Bei diesen beiden Beispielen führten alle Eingaben zu erlaubten Übergängen. Aber es sind auch Automaten möglich, bei denen bestimmte Übergänge nicht erlaubt sind (d.h. in den jeweiligen Zuständen werden nicht alle Eingabezeichen akzeptiert). Beispiel: Ein Automat, der eine Zahl in Gleitkomma-Notation (z.B. `-3,74E+23`) einliest – hier dürfen die Komponenten der Zahl

(Vorzeichen, Dezimalkomma, Exponentenzeichen) nicht in jeder beliebigen Reihenfolge stehen (beispielsweise darf das 'E' nicht mehrfach auftreten oder das Vorzeichen nur am Anfang von Mantisse bzw. Exponent stehen). Um dieses Problem zu beheben, fügt man einen weiteren Zustand hinzu, nämlich einen Fehlerzustand, zu dem immer dann gewechselt wird, wenn kein anderer Übergang erlaubt ist, und von dem aus es keinen Übergang zu einem anderen Zustand hin gibt (sondern nur zu sich selbst). Wenn sich nach dem Einlesen der Eingabe der Automat im Fehlerzustand befindet, weiß man, dass die Eingabe nicht akzeptiert wurde – ansonsten befindet er sich in einem der anderen Endzustände.

Bei den Automaten aus den vorigen Beispielen ist der Weg zwischen den Zuständen eindeutig festgelegt, weil jeder Übergang durch die jeweilige Eingabe (beim ersten Automat: Teilbarkeit der Eingabeziffer; beim zweiten Automat: Summe der Binärziffern) eindeutig vorgegeben wird. Darum bezeichnet man so einen Automat auch als „deterministischen endlichen Automaten“, auf englisch „deterministic finite automaton“ oder kurz DFA. Um einen solchen Automaten in einer Programmiersprache zu implementieren, könnte man die Zustände mit Labels beschreiben und die Übergänge zwischen ihnen durch GOTO-Sprünge. Falls die Programmiersprache das nicht erlaubt, könnte man den Zustand und die erlaubten Übergänge in Variablen speichern (z.B. die Übergangsrelation, die eine Matrix aus den möglichen Eingaben und den vorhandenen Zuständen darstellt, als Array).

Nichtdeterministische endliche Automaten

Man kann aus mehreren DFAs einen großen Automaten konstruieren. Beispielsweise möchte man einen Automaten, der das Datum analysiert – dummerweise kann dieses sowohl in der Schreibweise „Tag/Monat/Jahr“ als auch „Monat/Tag/Jahr“ als auch „Jahr/Monat/Tag“ gegeben sein. Man könnte nun drei DFA-Automaten konstruieren, die auf den Ziel-String losgelassen werden. Wenn dieser beispielsweise „13/05/98“ lautet, wird der zweite und dritte Automat scheitern, weil es weder einen 13. Monat noch einen 98. Tag im Monat gibt. Ob ein Automat scheitern wird, stellt sich aber nicht sofort heraus – der dritte Automat wird sowohl „13“ als Jahreszahl als auch „05“ als Monatszahl akzeptieren, erst bei der Tageszahl geht es schief. Wenn man nun diese drei Automaten zu einem großen Automat kombiniert, der nach den Zahlenbereichen unterscheidet, welcher Übergang möglich und welcher es nicht ist, muss der Ziel-String nur einmal durchlaufen werden (statt von drei Automaten jeweils einzeln). Außerdem sieht man bei der Konstruktion sofort, welche Mehrdeutigkeiten es gibt (z.B. gibt es kein Datum, das als „Monat/Tag/Jahr“ oder „Jahr/Monat/Tag“, gleichzeitig nicht aber als „Tag/Monat/Jahr“ interpretierbar ist), und man kann aus diesem Automaten sehr einfach weitere Automaten ableiten, die nur bestimmte Teilmengen bzw. die Vereinigungs-, Schnitt- oder Differenzmenge der Eingaben akzeptiert.

Dieser große Automat ist im Unterschied zu den DFAs, aus denen er aufgebaut ist, nicht mehr eindeutig – für das selbe Eingabezeichen gibt es mehrere gleichberechtigte Möglichkeiten, und erst wenn der Automat bei einer der Möglichkeiten im Fehlerzustand gelandet ist, weiß er, dass diese Möglichkeit falsch war und dass er eine andere ausprobieren muss. Im Gegensatz zum DFA haben wir also nicht mehr den Fall, dass der Weg durch die Eingabedaten eindeutig festgelegt wird – wir haben also einen nichtdeterministischen Automaten, einen „nondeterministic finite automaton“, kurz NFA.

Vom endlichen Automaten zum regulären Ausdruck

Um einen endlichen Automaten in einem gleichwertigen regulären Ausdruck auszudrücken, geht man folgendermaßen vor:

- Wenn bei einem Zustand nur ein Übergang hineinführt und nur ein Übergang hinausführt, dann

ersetzt man den Zustand mit seinen beiden Übergängen zu einem neuen Übergang, der mit der Konkatenation der Zeichen der beiden ursprünglichen Übergänge beschrieben wird – man hängt die Zeichen einfach hintereinander: aus 'a' und 'b' wird 'ab'.

- Wenn der Zustand zusätzlich einen Übergang zu sich selbst hat, dann führt man ebenfalls eine Konkatenation durch (zwischen dem hineinführenden Übergang und dem zu sich selbst führenden Übergang), und fügt ein Sternchen an – aus 'a' und 'b' wird 'ab*'.
- Zwei parallele Übergänge werden ersetzt durch einen einzigen, der mit der Vereinigung der Bezeichnungen der Übergänge beschrieben wird (mit einem Pipe-Zeichen dazwischen) – aus 'a' und 'b' wird 'a|b'.
- Wenn die Situation komplizierter ist, splittet man die Zustände auf. Aus einem Zustand, in den mehrere Übergänge hinein und hinaus führen, macht man zwei Zustände, in die Übergänge entweder nur hinein oder nur hinaus führen.

Wenn man dieses Verfahren beispielsweise für den endlichen Automaten, der die Teilbarkeit durch drei berechnet, durchzieht, ergibt sich folgender regulärer Ausdruck. Die eckigen Klammern bei z.B. [147] bedeuten, dass für den Übergang „Rest 1“ die Ziffern 1, 4 und 7 erlaubt sind, d.h. der Inhalt einer eckigen Klammer beschreibt einen der ursprünglichen Übergänge, die runden Klammern gruppieren lediglich zusammengehörige Teile (ganz mathematisch), und das Pluszeichen bedeutet „mindestens einmal“:

```
([0369]|[147][0369]*[258]|([258]|[147][0369]*[147])([0369]|[258]
[0369]*[147])*([147]|[258][0369]*[258]))+
```

Eigenschaften von Regex-Engines (DFA, NFA, Posix-NFA)

Beispiele für die verschiedenen Engine-Typen:

- DFA: *grep*, *awk*, *lex*
- NFA: *Perl*, *Tcl*, *Python*, *Emacs*, *sed*, *vi*

Funktionsweise einer DFA-Engine

Ein DFA ist, wie beschrieben, deterministisch, weil die Eingabedaten den Weg durch den Automaten eindeutig festlegen. Der DFA hat also das durch die Regular Expression vorgegebene Suchmuster in Form eines Regelwerks für Zustandsübergänge umgesetzt (in der Art „was passiert, wenn welcher Buchstabe kommt?“) und geht nun den Text Buchstabe für Buchstabe durch, wobei er bei jedem Buchstaben testet, ob er mit der Regex vereinbar ist (d.h. ob ab dort ein Treffer beginnen könnte bzw. ob ein begonnener Treffer fortgesetzt werden kann). Wenn mehrere Alternativen zur Verfügung stehen, schließt er diese der Reihe nach aus, bis nur noch eine Alternative übrig bleibt. Das hat zur Folge, dass erstens die Verarbeitungsgeschwindigkeit praktisch nur von der Textlänge abhängt (weil der Text in jedem Fall genau einmal durchlaufen wird) und dass zweitens verschiedene Regex-Formulierungen, die zum gleichen Ergebnis führen (z.B. `abc` und `[aa-a](b¦b{1})c`), absolut gleichwertig sind und sich von der Suchgeschwindigkeit her nicht unterscheiden. Jeffrey E. F. Friedl bezeichnet den DFA deshalb als *text-driven*.

Funktionsweise einer NFA-Engine

Im Unterschied zu einem DFA geht ein NFA nicht mit dem gesamten Suchausdruck durch den Text, sondern – wenn es mehrere Möglichkeiten gibt – nur mit einem Teil. Dieser wird so lange verfolgt, bis entweder die Suche erfolgreich ist (d.h. diese Möglichkeit im Text gefunden wurde), oder bis ein Fehler auftritt. In diesem Fall kehrt der NFA zurück zum Ausgangspunkt und probiert eine andere Möglichkeit aus. Dieses „ausprobieren und zum Ausgangspunkt zurückkehren“ nennt man Backtracking; umgesetzt wird das mit einem LIFO-Stack. Der NFA geht also u.U. mehrmals durch den Text, dabei bestimmt die Formulierung der Regular Expression, in welcher Weise er das tut – darum bezeichnet Jeffrey E. F. Friedl den NFA als *regex-driven*.

Vorgehensweise: Der NFA durchläuft den Text von links nach rechts (z.B. findet `[0-9]*` bei `ABC 123` zuerst den Leerstring vor dem A), und auch verschiedene Alternativen werden immer von links nach rechts durchprobiert. Das bedeutet, dass Alternation bei einem NFA non-greedy ist, denn wenn eine der Varianten auf den Text passt, probiert er nicht weiter, ob eine andere vielleicht besser passt. Wenn Quantifizierer vorhanden sind, verhält er sich greedy, d.h. bei `ab?c` probiert er zuerst `abc`, dann erst `ac`. Mit dem Stern ist es genauso; `x*` ist nichts anderes als `x?x?x?x?...`. Bei Zeichenklassen könnte man vielleicht meinen, dass sie genauso wie Alternation funktionieren, d.h. bei `[abc]` wird mit Backtracking zuerst a, dann b und dann c probiert. Das stimmt aber nicht; Zeichenklassen funktionieren eher wie ein „Buchstabensieb“, d.h. es wird in einem Schritt verglichen, ob der Buchstabe im Text auf die Zeichenklasse passt.
Beispiel: Wenn die Regex `to(nite¦knight¦night)` lautet, wird, nachdem im Text `to` gefunden wurde, der Reihe nach durchprobiert: wenn `nite` nicht passt, wird es mit `knight` versucht; wenn das nicht passt, kommt `night` an die Reihe. Wenn die Regex umformuliert wird, bedeutet das automatisch, dass sie der NFA auf anderen Wegen durchläuft. Beispiele für Umformulierungen:
`tonite¦toknight¦tonight`, `to(ni(ght¦te)¦knight)`, `to(k?night¦nite)`.

Die Konsequenzen aus diesem Verhalten:

- Die Regex-Kompilierung geht bei einem NFA schneller und braucht weniger Speicher.
- Um festzustellen, dass eine Regex nicht auf den Text passt, muss ein NFA erst alle möglichen Kombinationen durchprobieren, d.h. ein NFA ist immer dann besonders langsam, wenn er die Regex überhaupt nicht oder erst sehr spät findet. Er ist dagegen schnell, wenn der Suchbegriff gefunden wird.
- Der Benutzer kann durch die Formulierung der Regex die Arbeitsweise der Engine und damit die Geschwindigkeit der Suche beeinflussen. Wenn er die Regex-Anfrage ungeschickt stellt, muss die Engine oft herumprobieren, bis sie den richtigen Weg gefunden hat; weil es oft eine sehr große Zahl von Möglichkeiten gibt, kann das durchaus ins Gewicht fallen.

Die Fähigkeit, wieder an einen vorigen Punkt zurückkehren zu können, hat noch weitere Vorteile:

- Rückbezüge werden dadurch möglich. Ein DFA, der nur linear durch den Text geht, weiß hinten nicht mehr, was vorne gewesen ist; ein NFA merkt sich dagegen die Schritte, so dass er zu früheren Zwischenergebnissen zurückkehren kann – wie bei Hänsel und Gretel mit den Brotkrumen. Damit weiß der NFA auch, auf was eine Klammer gepasst hat, und kann diesen Inhalt an späterer Stelle wieder einfügen.
- Ein NFA kann sich außerdem merken, an welcher Stelle im Text die Klammer gepasst hat und dies dem Benutzer sagen. Wenn ein DFA dagegen festgestellt hat, dass die Klammer auf den Text passt, weiß er nicht mehr, an welcher Stelle im Text dieser Bereich begonnen hat.
- Auch Lookahead und Lookbehind (d.h. es wird gesucht in Abhängigkeit davon, was davor oder dahinter steht) ist nur mit Backtracking möglich. Ein DFA muss auf diese Dinge prinzipiell verzichten.

Was ist ein Posix-NFA?

Nach dem theoretischen Modell lässt sich jeder NFA in einen DFA umwandeln. Bestimmte Features, die Programme mit Regular Expressions bieten, lassen sich dagegen nicht in einem DFA ausdrücken – weil ein DFA den Text nur einmal durchläuft, sind Rückbezüge wie z.B. bei der Suche nach doppelten Wörtern (`([a-zA-Z]+) +\1`) mit einem DFA unmöglich, denn er merkt sich nicht, von wo bis wo der Text schon einmal gepasst hat (er weiß nur, dass er gepasst haben muss). Kurz: Es gibt also Gründe, einen NFA statt einem DFA zu implementieren.

Nach der Theorie müsste sich ein DFA und ein NFA stets gleich verhalten. Das ist aber auch nicht unbedingt gegeben. Beispiel: Die Suche von `(tour¦to¦tournaments)` auf den String `three tournaments`. Ein DFA geht den Suchstring von vorne nach hinten durch und entdeckt dabei, dass die Suche ab dem Beginn des Wortes „tournaments“ erfolgreich ist. Während er das restliche Wort durchläuft, fliegt von den gebotenen Alternativen zuerst „to“ raus (weil es nicht mehr auf „tou“ passt, während die beiden anderen schon passen), dann fliegt „tour“ raus (weil es nicht mehr auf „tourn“ passt, aber immer noch ein Kandidat im Rennen ist), und schließlich stellt der DFA fest, dass „tournaments“ auch passt, es aber am längsten durchgehalten hat. Der DFA findet also immer das längste Suchergebnis, weil er den kompletten Text durchläuft. (Wenn es mehrere gleichlange Suchergebnisse gibt, meldet er das erste von ihnen, d.h. das, welches am weitesten links im Text steht.)

Ein NFA dagegen probiert die Möglichkeiten der Reihe nach durch, d.h. sucht erst einmal, ob „tour“ im Suchstring gefunden wird. Das ist der Fall, also ist der NFA glücklich und meldet eine erfolgreiche Suche – er hat jedoch nicht das längste Ergebnis gefunden. (Das würde er, wenn man die Suche umformulieren würde zu z.B. `(to(ur(nament)?)?)?` – bei einem DFA ist dagegen, wie gesagt, die Formulierung der Regex egal.)

Bei der Standardisierung der Regex-Engines durch das Posix-Projekt (wo insgesamt über 70 Programme standardisiert wurden), wurden die DFA-Engines unverändert übernommen und gefordert, dass ein Posix-konformer NFA die gleichen Suchergebnisse wie ein DFA liefert. Das bedeutet: Im Gegensatz zu einem „klassischen" NFA muss ein Posix-NFA immer alle Möglichkeiten durchprobieren, damit er, genauso wie der DFA, das längstmögliche Suchergebnis findet. Man kann also sagen: Bei DFA und Posix-NFA ist die Alternation greedy (da dort immer soviel mitgenommen wird wie möglich, vgl. `.*`), nicht aber beim klassischen NFA.

Folgen aus dem unterschiedlichen Verhalten von DFA und NFA

Beispiele:

- Suche mit `(to¦top)(o¦polo)(gical¦o?logical)` auf das Wort `topological`: Ein klassischer NFA findet, dass „to" passt, dann dass „polo" dahinter passt, und schließlich „gical" – sein Ergebnis lautet also „to-polo-gical". Ein DFA und ein Posix-NFA dagegen schaut sich bei der ersten Klammer beide Möglichkeiten an und nimmt die längere der beiden – letztendlich findet er „top-o-logical".
- Suche mit `one(self)?(selfsufficient)?` auf `oneselfsufficient`: Der klassische NFA springt auf das erste Fragezeichen an (greedy) und findet „oneself", ein Posix-NFA und ein DFA findet dagegen „oneselfsufficient".
- Suche von Datumswerten wie `Jan 16`: Ein einfacher Ansatz wie `Jan [0123][0-9]` erlaubt falsche Werte wie `Jan 00` oder `Jan 34` und verbietet gleichzeitig richtige Werte wie `Jan 7`. Halbwegs funktionieren tut `Jan (0?[1-9]¦[12][0-9]¦3[01])` – der NFA macht jedoch wegen seiner non-greedy Alternation Probleme und findet bei `Jan 31` nur „Jan 3". Die Lösung ist hier ein simples Umsortieren, d.h. `Jan (3[01]¦[12][0-9]¦0?[1-9])`. Andere funktionierende Lösungen: `Jan (31¦[123]0¦[012]?[1-9])` oder `Jan (0[1-9]¦[12][0-9?]¦3[01]¦[4-9])` (bei dieser Lösung ist Reihenfolge unwichtig).

Interessant wird dieser Unterschied zwischen DFA und NFA, wenn man einen Text sucht, der über mehrere Zeilen geht – beispielsweise in einem Quellcode eine Zuweisung, die über mehrere Zeilen geht, die per Backslash verbunden sind. Ein erster Versuch könnte `^\w=(.*)` lauten, jedoch schließt bei den meisten Regex-Engines der Punkt keinen Zeilenumbruch ein. Die Formulierung `^\w=(.*)(\\\n.*)*` würde bei einem DFA tatsächlich funktionieren, während bei einem klassischer NFA der Punkt gleich noch auf den Backslash passen würde, so dass die zweite Klammer nicht gefunden wird (nur der Stern ist greedy, nicht die Alternation). Die Lösung für den klassischen NFA lautet dann z.B. `^\w=([^\n\\]*)(\\\n[^\n\\]*)*` (wobei dann keine Backslashes innerhalb der Zeilen auftreten dürfen) oder `^\w=(\\\n¦.)*`; wenn jedoch ein doppelter Backslash am Zeilenende nicht als Zeilenverbinder betrachtet werden soll (sondern als „per Backslash geschützter Backslash"), muss man den Punkt durch `[^\n\\]` ersetzen. Dann hat man jedoch noch das Problem, dass innerhalb der Zeilen nur '\n'-Sequenzen, aber keine anderen Backslash-Sequenzen vorkommen dürfen. Soll diese Beschränkung auch noch wegfallen, lautet die Regex `\w=(\\[\000-\377]¦[^\n\\])*`.

Zusammengefasst kann man sagen, dass man beim klassischen NFA mit mehr Sorgfalt arbeiten muss, allerdings ist seine „non-greedy Alternation" mächtiger, weil man die Reihenfolge selber angeben kann!

Performance-Betrachtungen

Ein DFA geht den Text immer einmal von Anfang bis Ende durch und ist daher unter Performance-Gesichtspunkten uninteressant, weil er immer gleich schnell ist – verschiedene Formulierungen der Regular Expression ändern nichts an der Performance, weil sie absolut gleichwertig sind.

Ein klassischer NFA dagegen probiert die verschiedenen Möglichkeiten, die die Regex bietet, nacheinander aus, und hört auf, sobald er eine Möglichkeit gefunden hat, die passt. Das bedeutet erstens, dass ein klassischer NFA nur dann schnell ist, wenn er ein Ergebnis findet; ansonsten probiert er alle Möglichkeiten durch, bis er aufgibt, und das kann dauern (z.B. würde der Ausdruck `(x*y*)*` theoretisch unendlich lang laufen, weil das Innere der Klammer leer sein kann; Emacs und moderne Perl-Versionen können damit umgehen, Python lässt solche Ausdrücke immer scheitern, andere Engines verbieten eine derartige Syntax). Zweitens kann man bei einem NFA durch die Sortierung der Möglichkeiten in der Regex die Geschwindigkeit beeinflussen, denn er ist schneller, wenn die Möglichkeit, die häufig erfolgreich ist, vorne steht.

Für einen Posix-NFA gilt im Prinzip das Gleiche wie für einen klassischen NFA; der einzige Unterschied ist, dass der Posix-NFA immer alle Möglichkeiten durchprobiert und somit immer langsam ist. Das heißt aber keinesfalls, dass bei ihm jegliche Optimierung sinnlos ist – im Gegenteil, man kann die Anzahl Möglichkeiten, die er durchprobieren muss, möglichst gering halten.

Grundprinzipien einer Regex-Engine

Um funktionierende Regular Expressions konstruieren zu können, muss man wissen, nach welchem Schema die Engine vorgeht. Die Grundprinzipien lauten:

- **Die Suche muss erfolgreich sein.** Die folgenden Grundprinzipien werden nur angewandt, wenn damit der Erfolg der Suche nicht verhindert wird. Klingt trivial, kann aber bereits bei einer Ersetzung zu unerwarteten Effekten führen, weil die Regex-Engine lieber auf Greediness verzichtet, um die Ersetzung durchführen zu können (siehe bei den Beispielen).
- **Quantifizierer sind gierig.** Sie holen sich so viele Zeichen wie möglich, unter der Bedingung, dass die Suche immer noch erfolgreich ist. Dazu holen sie sich erst einmal alle verfügbaren Buchstaben und geben dann einen nach dem anderen ab, bis die Suche erfolgreich ist. Beispiel: Eine Suche mit `/\w+s/` auf `Messias` findet den gesamten Text, statt nur `Mes`. Das `\w+` holt sich zuerst das gesamte Wort – damit wäre aber kein Platz mehr für das letzte 's'. Also gibt es einen Buchstaben ab (d.h. beansprucht nur noch `Messia`), dann funktioniert die Suche.
- **Das erste Suchergebnis im Text gewinnt.** Beispielsweise findet eine Suche mit `/Stadt|gehe/` auf den Text `Ich gehe heute in die Stadt` zuerst das Wort „gehe“, weil es im Text zuerst vorkommt. Dabei ist wichtig, welches Wort zuerst anfängt – nicht, welches zuerst aufhört.
- **Die erste Alternative gewinnt** (beim klass. NFA) bzw. **die längste Alternative gewinnt** (DFA und Posix-NFA): Bei einer Suche mit `(to|top)p?ological` auf `topological` gewinnt beim klassischen NFA die erste Alternative, nämlich `to`, während beim DFA und beim Posix-NFA die längste Alternative gewinnt, nämlich `top`.
- **Der erste Quantifizierer gewinnt.** Bei zwei gierigen Quantifizierern hintereinander kommt der erste zuerst zum Zug, so dass für den zweiten Quantifizierer nur noch das Minimum, das er benötigt, übrig bleibt. Beispiele:
  `Subject: (Re: )?(.*)`: Hier ist das Fragezeichen zuerst dran und schnappt das `Re:` weg, falls vorhanden, d.h. es landet niemals in der zweiten Klammer.
  `^.*([0-9]+)`: Diese Suche sollte aus dem Text `die Zahl lautet 1234` die Zahl `1234` extrahieren – aber das funktioniert nicht, weil der Stern vorher gierig wird und für das Plus nur ein einziges Zeichen übrig lässt (welches er mindestens braucht). Die Suche liefert in der Klammer also nur `4`.
  `(.*)foo(.*)bar`: Dies ist eine prinzipiell sinnlose Konstruktion, weil der erste Stern zuerst gierig wird und weil die minimale Zeichenanzahl bei einem Stern gleich Null ist – d.h. für die zweite Klammer wird niemals etwas übrig bleiben.

Anwendungsbeispiele

Vorgehensweise: Zuerst sollte man die Suchbedingung in Worten ausformulieren, d.h. beschreiben, was man finden will und was ausdrücklich nicht („say what you mean“). Ist dieser Schritt erledigt, ist die Übersetzung in eine Regular Expression oft nur noch eine Formsache. Wenn man sie schrittweise aufbaut und optimiert, ist das Schreiben einer Regular Expression einfacher als das Lesen.

Bei einer guten Regular Expression sind folgende Dinge erfüllt:

- sie findet was man sucht
- sie findet das, was man von der Suche ausschließen will, nicht
- sie ist trotzdem nicht zu komplex, sondern noch so verständlich wie möglich
- sie ist effizient (falls man es mit einem NFA zu tun hat; bei einem DFA kann man in Sachen Performance nichts falsch machen)

Wenn man eine Regex schreibt, sollte man also bedenken:

- Was genau will ich finden?
- Welche ähnlichen Fälle will ich nicht finden? Auf welche Art und Weise können sie vorkommen?
- Was passiert eigentlich, wenn nichts gefunden wurde? Wie sind die Auswirkungen auf die Performance?

Man kann fast jede Regex unendlich perfektionieren, um ihre Performance zu steigern oder die Suche allgemeingültiger zu machen. Manchmal ist es sinnvoll und lohnenswert (z.B. wenn eine einfachere Regex bei entsprechend tückischen Suchtexten gelegentlich fehlerhafte Ergebnisse liefert, oder wenn eine an sich einfache Suche eine merkliche Rechenleistung frisst) – dann ist man glücklich, wenn man weiß, wie man vorgehen muss. Oft lohnt sich eine Optimierung aber auch nicht – es hängt vom Einzelfall ab.

Diverses

- Suche nach doppelten Wörtern (mit Backreferences bei der Suche):
  `([a-zA-Z]+) +\1`
- Suche nach erlaubten Variablennamen in Programmiersprachen:
  `[a-zA-Z_][a-zA-Z_0-9]{0,31}`
- Suche nach Uhrzeit im 12h-Format:
  `(1[012]|[1-9]):[0-5][0-9] (am|pm)`
- Suche nach Uhrzeit im 24h-Format:
  `0?[0-9]|1[0-9]|2[0-3]` bzw. `[01]?[0-9]|2[0-3]`
- Suche nach Leerzeichen und Tabulatoren:
  `[\t ]*` hat gegenüber `( *|\t*)` den Vorteil, dass Leerzeichen und Tabs gemischt vorkommen können
- Etwas an den Zeilenanfang setzen (hier '>'), indem man den Zeilenanfang ersetzt:
  `s/^/> /`
- E-Mail-Adresse im Mail-Header (z.B.: `Hans Meier <hans.meier@aol.com>`, Name wahlweise in Anführungszeichen, allerdings ohne Überprüfung, ob in den spitzen Klammern eine gültige E-Mail-Adresse steht):
  `([^"<>]|"[^"]*")*<([^"<>]|"[^"]*")*>([^"<>]|"[^"]*")*`

- Um alle Zeichen incl. dem Zeilenumbruch zu suchen, falls der Punkt '.' den Zeilenumbruch nicht beinhaltet:
  `(.¦\n)` oder `[\000-\377]`
- Pfad aus Dateinamen entfernen: `s/.*\///`
  Pfad und Dateiname trennen: `(.*)/(.*)`
- mehrere Bindestriche in das HTML-<HR>-Tag umwandeln: `s/-*/<HR>/` funktioniert nicht, weil der Stern auch 0-mal erlaubt => durch Pluszeichen ersetzen.
- Zahlen (mit Vorzeichen und Dezimalpunkt) suchen:
  `-?[0-9]*\.?[0-9]*`
  ist suboptimal, statt dessen
  `-?([0-9]+(\.[0-9]*)?¦\.[0-9]+)`
  (es wird jedoch auch ein Datum wie z.B. `20.9.1978` gefunden)
- Dezimalstellen reduzieren: Bei einer Zahl sollen nur drei Dezimalstellen übrig bleiben bzw. nur zwei, wenn die dritte gleich Null ist.
  `/(\.\d\d[1-9]?)\d*/$1/`: Funktioniert bei den Zahlen `1.23456`, `1.23045` und `1.23` (hier keine Ersetzung). Bei `1.234` funktioniert die Regex auch, aber hier ist unschön, dass eine Ersetzung durchgeführt wird, bei der sich nichts ändert (also sinnloser Performance-Verlust). Man käme hier vielleicht auf die Idee, `\d+` statt `\d*` zu schreiben, damit die Regex nur auf Zahlen mit mehr als drei Dezimalstellen passt und somit nur bei diesen eine Ersetzung durchführt. Damit würde jedoch die Regex bei der Zahl `1.234` nicht mehr funktionieren. Der Grund ist: In diesem Fall wird das Fragezeichen non-greedy, d.h. es wird eine Ziffer aus der Klammer abgetreten, um das `\d+` dahinter erfüllen zu können. Es ist für die Engine wichtiger, dass die Ersetzung überhaupt durchgeführt werden kann, als dass die Quantifizierer ihre Greedyness ausspielen können.
  Eine mögliche Lösung für dieses Problem, die aber noch in keiner Software verwirklicht wurde, wären „possessive quantifiers“, die einen lokalen Sucherfolg niemals aufgeben (und nur durch eine umgebende Klammer begrenzt werden können), auch wenn dadurch ein globaler verhindert wird.
  Obiges Beispiel könnte man schreiben als `(\.\d\d¦\.\d\d[1-9])\d*` – aber nur, wenn die Alternation greedy ist (also nicht bei klassischen NFAs). Bei non-greedy Alternation gibt es wieder Problem, dass 3. Dezimale außerhalb der Klammer landet. Indem man eine Alternation macht, in der im zweiten Teil das Gleiche wie im ersten Teil plus etwas Zusätzliches steht, kann man also quasi einen Fragezeichen-Quantifizierer realisieren, der non-greedy ist!

IP-Adressen

Versuchen wir einmal, möglichst exakt nach IP-Adressen (z.B. `192.168.17.4`) zu suchen.

- Erster Entwurf: `[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*`
- Der erste Entwurf ist insofern ungeschickt, weil es sogar auf `...` passt, d.h. es müssen gar keine Ziffern zwischen den Punkten vorkommen. Außerdem würden auch Zahlenkolonnen wie `72123.3.21.993` gefunden werden, die definitiv keine IP-Adressen sind. Der erste Schritt lautet also, den Ausdruck mit '^' und '$' zu umgeben und die Sternchen durch etwas anderes zu ersetzen (`\d{1,3}` oder `\d\d?\d?` oder `\d(\d\d?)?`). Der zweite Entwurf lautet also:
  `^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$`
- Jetzt sollen die Grenzen etwas flexibler gestaltet werden. Statt '^' und '$' kann man nicht '\w' nehmen, weil der Punkt kein Wortbestandteil ist, d.h. die IP-Adresse würde nicht als Wort betrachtet werden. Mit einem Ausdruck wie `(^¦[[:space:]])` am Anfang bekommt man jedoch das, was man will. Der dritte Entwurf lautet also:

```
(^¦[[:space:]])\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}($¦[[:space:]])
```

- Als nächstes muss der Wertebereich der vier Zahlen auf 0-255 beschränkt werden. Statt `\d{1,3}` muss man also eine Konstruktion in der Art wie `\d¦\d\d¦[01]\d\d¦2[0-4]\d¦25[0-5]` oder `[01]?\d\d?¦2[0-4]\d¦25[0-5]` schreiben (wobei die zweite Version beim NFA etwas schneller ist, weil sie weniger Alternativen bietet und damit weniger Backtracking). Die finale Version sieht also ungefähr so aus:
  ```
  (^¦[[:space:]])[01]?\d\d?¦2[0-4]\d¦25[0-5]\.[01]?\d\d?¦2[0-4]\d¦25[0-5]\.
  [01]?\d\d?¦2[0-4]\d¦25[0-5]\.[01]?\d\d?¦2[0-4]\d¦25[0-5]($¦[[:space:]])
  ```
- Jetzt gibt es nur noch das Problem, dass auch die Adresse `0.0.0.0` gefunden wird, die jedoch keine gültige IP-Adresse ist. Mit Regular Expressions kann man das nicht sinnvoll implementieren, sondern prüft lieber mit anderen Methoden auf diesen Sonderfall. Einzig *Perl* bietet mit dem negativen Lookahead eine Regex-Möglichkeit: `(?!0+\.0+\.0+\.0+$)`

Datum

Gesucht werden soll eine Datumsangabe nach ISO 8601, d.h. Jahr-Monat-Tag, z.B. „1978-09-20“.

V seien die durch vier teilbaren zweistelligen Zahlen außer „00“:
V = `0[48]¦[2468][048]¦[13579][26]`

Damit definieren wir *S*, die vierstelligen Jahreszahlen der Schaltjahre, sowie *J*, alle Jahreszahlen. Dabei fangen wir mit 1000 an, weil es vorher noch keinen Gregorianischen Kalender gab; dessen genaues Einführungsdatum, den 1582-10-15, berücksichtigen wir nicht. Das soll aber die einzige Vereinfachung bleiben.
S = `[1-9][0-9]`*V*`¦`*V*`00`
J = `[1-9][0-9][0-9][0-9]`

M bezeichne alle Monate, *N* die außer Februar und *L* die mit 31 Tagen:
M = `0[1-9]¦1[0-2]`
N = `0[13-9]¦1[0-2]`
L = `0[13578]¦1[02]`

T seien die Tageszahlen, die es in jedem Monat gibt:
T = `[01][1-9]¦10¦2[0-8]`

Damit definieren wir *D*, die Daten, die es in jedem Jahr gibt. Das Minuszeichen, das hier auftritt, ist im Gegensatz zu den vorherigen terminal, d.h. es bedeutet nur, dass dort ein Minuszeichen zu stehen hat. Vorher war nämlich jedes Minuszeichen innerhalb einer eckigen Klammer, wo es eine Bedeutung als Metazeichen hatte.
D = *M*`-`*T*`¦`*N*`-(29¦30)¦`*L*`-31`

Schließlich erhalten wir alle erlaubten Daten *G*:
G = *J*`-`*D*`¦`*S*`-02-29`

Jetzt setzen wir für die Variablen wieder die zugehörigen regulären Ausdrücke ein; erster Schritt:
G = `[1-9][0-9][0-9][0-9]-(`*M*`-`*T*`¦`*N*`-(29¦30)¦`*L*`-31)¦([1-9][0-9]`*V*`¦`*V*`00)-02-29`

Endgültige Version:
G = `[1-9][0-9][0-9][0-9]-((0[1-9]¦1[0-2])-([01][1-9]¦10¦2[0-8])¦(0[13-9]¦1[0-2])-(29¦30)¦(0[13578]¦1[02])-31)¦([1-9][0-9](0[48]¦[2468][048]¦[13579][26])¦(0[48]¦[2468][048]¦[13579][26])00)-02-29`

Verbotene Zeichenketten

(Beispiel von Helmut Richter)

Um nur die Wörter „Fisch“ und „Fleisch“ zu erlauben, reicht der simple Ausdruck `Fisch|Fleisch` (oder `F(le)?isch`). Um aber das Gegenteil zu erreichen, d.h. dass alles außer den Wörtern „Fisch“ und „Fleisch“ erlaubt ist, muss man viel größere Anstrengungen unternehmen. Das Grundprinzip lautet (am Beispiel „Fisch“): Erlaubt ist, was mit einem anderen Buchstaben als 'F' anfängt, oder was mit 'F' anfängt und nicht mit 'i' weitergeht, oder was mit 'Fi' anfängt und nicht mit 's' weitergeht usw. (wobei „weitergehen“ auch heißen kann: „mit einem Leerstring weitergehen“, d.h. 'Fis' alleine ist erlaubt). Für „Fisch“ ergibt sich dann:

```
F((i(([^s]|s[^c]|sc[^h]|sch.).*|(sc?)?))?)|([^F].*)?
```

Endergebnis:

```
F((l([^e]|e[^i]).*|(le)?i(([^s]|s[^c]|sc[^h]|sch.).*|(sc?)?)|le?)|([^le].*)?)|
([^F].*)?
```

Und das ist der Grund, warum manche Regex-Programme eine Option besitzen, mit der man die Regex invertieren kann...

Begrenzer

Text zwischen Begrenzern zu suchen ist eine häufige Aufgabe. Ein Beispiel dafür wäre HTML-Code, CSV-Tabellen, Text innerhalb von Anführungszeichen oder auch C-Code (z.B. Kommentare). Nach steigender Schwierigkeit werden auch die Regular Expressions immer komplizierter – manchmal ist es nicht sinnvoll möglich, die Suche mit einer einzigen Regex durchzuführen. Beispiel HTML: Hier bestehen sowohl Anfang als auch Ende eines von zwei Tags geklammerten Bereichs aus mehreren Zeichen, außerdem treten sie verschachtelt auf, darum arbeitet man hier lieber in mehreren Schritten und filtert die Suchergebnisse mehrmals hintereinander.

Schwierigkeitsstufe 1: einfacher Begrenzer, tritt nur als Begrenzer auf

Dieser Fall ist einfach. Um beispielsweise den Text innerhalb von Anführungszeichen extrahieren, wird `"(.+)"` zwar scheitern (weil das greedy '+' alles, inklusive Anführungszeichen, aufnimmt), aber man kann ungefähr so etwas schreiben:

```
"([^"]+)"
```

Auf deutsch: „Alle Nicht-Anführungszeichen zwischen Anführungszeichen“, oder allgemeiner gesagt: Zuerst der öffnende Begrenzer, dann das Innere (welches die Begrenzer nicht enthält), dann der schließende Begrenzer.

Wenn die Begrenzer nur in der Mitte stehen, aber nicht am Zeilenanfang oder Zeilenende, sollte man diese Fälle auch berücksichtigen. Beispiel CSV-Tabelle: hier werden die Felder durch Kommata oder Zeilenanfang/-ende begrenzt, d.h. man muss nach `(^¦,)` als Anfangsbegrenzer und `($¦,)` als Schlussbegrenzer suchen.

Schwierigkeitsstufe 2: einfacher Begrenzer, tritt in verschiedenen Bedeutungen auf

Jetzt nehmen wir das Problem aus Schwierigkeitsstufe 1, aber erlauben innerhalb der Anführungszeichen weitere durch Backslash geschützte Anführungszeichen, z.B. `"1\" bedeutet ausgeschrieben ein Zoll"`. Ein erster Versuch könnte lauten `"([^"]¦\\")+"` – aber das funktioniert nicht, weil ein Backslash auch in `[^"]` passt und ein klassischer NFA ihn deshalb gleich in die erste Alternative einsortieren würde (remember: ein DFA sucht dagegen immer das längste Ergebnis). Die Abhilfe ist einfach – die Alternativen umdrehen: `"(\\"|[^"])+"`. Hat nur den kleinen Nachteil, dass bei allen anderen Buchstaben außer Backslash (der recht selten auftritt) immer beide

Alternativen ausprobiert werden müssen, was die Sache etwas langsam macht. Nächstes Problem: Was passiert, wenn das schließende Anführungszeichen fehlt? Dann wird ein mit Backslash geschütztes Anführungszeichen als schließendes Anführungszeichen betrachtet, und der Backslash fällt wieder unter die Menge `[^"]`. Lösung: `"([^"\\]|\\")+"` – damit kann man sogar die ursprüngliche Reihenfolge der Alternativen wiederherstellen, d.h. auf einem NFA ist dieser kompliziertere Ausdruck sogar schneller.

Übrigens: In manchen Skriptsprachen ist der Backslash bereits von sich aus ein Metazeichen, außerdem in Regular Expressions – was zur Folge hat, dass, wenn man in einer Regex tatsächlich einen Backslash als Zeichen meint, ihn doppelt escapen muss, d.h. die Suche nach einem Backslash lautet dann `/\\\\/`.

Schwierigkeitsstufe 3: einfacher Begrenzer, verschachtelt

Beispiel: Man will in mehrfach verschachtelten Klammern etwas suchen, wie z.B. `val = foo(bar(this,3.7)) + 2 * (that - 1)`. Folgende Regex-Bausteine haben wir im Angebot:

- `\(.*\)` sucht den Inhalt eines Klammernpaares (von der ersten öffnenden bis zur letzten schließenden Klammer), findet im obigen Beispiel `bar(this,3.7) + 2 * (that - 1`.
- `\([^)]\)` sucht von der ersten öffnenden bis zur ersten schließenden Klammer, findet im obigen Beispiel `bar(this,3.7`.
- `\([^()]\)` sucht ein Klammernpaar, das im Inneren gar keine Klammern hat, d.h. die innerste Klammernebene, findet im obigen Beispiel `this,3.7`.

Um den Inhalt der ersten verschachtelten Klammernebene zu extrahieren, muss man dann etwas wie `\([^()]*\(([^()]*)\)\)` oder auch `\(([^()]*|\(([^()]*)\))*\)` schreiben. Man kann sich vorstellen, wie kompliziert die Ausdrücke werden, wenn noch tiefere Klammernebenen extrahiert werden sollen. Mit Regular Expressions kann man prinzipiell nicht in beliebig tief verschachtelten Klammerebenen suchen, sondern man muss die Regex auf die gewünschte Ebenenanzahl anpassen. Bei *Perl* geht das etwas einfacher; um bis zur Ebene Nummer $depth zu suchen, setzt man die Regex folgendermaßen zusammen (ohne Erklärung):
`('\(' . '([^()]*|\(' x $depth . '[^()]*' . '\))*' x $depth . '\)'`.

Schwierigkeitsstufe 4: Begrenzer aus mehreren Zeichen

Bisher war der Trick, dass man zwischen den Begrenzern die Begrenzerzeichen nicht erlaubt, was mit Zeichenklassen kein Problem ist. Wenn der Begrenzer aber aus mehreren Zeichen besteht (wie z.B. bei HTML-Tags), fallen Zeichenklassen weg, weil es auf die Reihenfolge ankommt. Und jetzt wird es wirklich hässlich. Beispiel: Um einen Text zu erkennen, der zwischen den Wörtern „Anfang“ und „Ende“ steht, muss man folgendes Konstrukt basteln (vgl. das „Fisch und Fleisch“-Beispiel oben):
`Anfang([^E]|E[^n]|En[^d]|End[^e])*(E(nd?)?)?Ende`

Beispiel: C-Kommentare

Ein C-Kommentar (d.h. er beginnt mit „/*“ und endet mit „*/“) ist ein gutes Praxisbeispiel für einen Mehrzeichenbegrenzer mit allen Schikanen, da beide Zeichen auch im Inneren vorkommen können und außerdem Regex-Metazeichen (ok, der Schrägstrich nicht im engeren Sinne...) sind (d.h. man muss sie mit Backslash schützen, was der Optik nicht gerade zugute kommt).

- `\/\*[^*]*([^*]|\*[^/])*\*\/` oder `\/\*([^/]|[^*]\/)\*\/` (wörtlich „Begrenzer, Nicht-Stern, dann mehrere Nicht-Sterne oder Stern mit Nicht-Backslash, dann Begrenzer“): scheitert an

den gültigen Kommentaren `/** blabla **/` und `/*/ blabla /*/` und nimmt bei `x = 2; /* Kommentar 1 */ y = 3; /* Kommentar 2 */` alles vom Anfang des ersten bis zum Ende des zweiten Kommentars.

- `\/\*[^*]*([^*]¦\*+[^/*])*\*\/`: Ein kleines Plus und der Stern zu den verbotenen Zeichen der dritten Zeichenklasse löst das erste Problem der mehrfachen Sterne hinter dem Schrägstrich.
- `\/\*[^*]*([^*]¦\*+[^/*])*\*+\/`: Damit sind jetzt alle Probleme gelöst. Unter *Perl* ginge das mit dem Lookahead-Mechanismus etwas einfacher.
- `\/\*[^*]*\*+([^*/][^*]*\*+)*\/`: Das Gleiche mit „Loop Unrolling“ für mehr Performance bei NFAs.
- `"(\\.¦[^\\"])*"¦\/\*[^*]*\*+([^*/][^*]*\*+)*\/¦\/\/[^\n]`: Hinzugefügt wurden zwei Alternativen, die erstens C++-Kommentare und zweitens Zeichenketten (in Doublequotes) erkennen; in beiden machen C-Kommentare keinen Sinn.
- Jetzt wird es langsam aber sicher unübersichtlich. Um sowohl die Fälle C-Kommentar, C++-Kommentar, Zeichenkette in Doublequotes und Zeichenkette in Singlequotes zu erkennen, muss die Regex beispielsweise nach folgendem Schema angeordnet sein (hier: zum Entfernen von Kommentaren):
  `/(Doublequotes¦Singlequotes)¦C-Kommentar¦C++-Kommentar/$1/`
- Weil in diesem Schema viele Alternativen vorhanden sind, ist es langsam – ein NFA muss für jeden Buchstaben alle Möglichkeiten durchprobieren. Darum fügen wir in die Klammer eine weitere Alternative ein, die sich erstmal die „normalen Fälle“ kümmert (Zeichenklasse `[^"'/]`, für den klassischen NFA mit Quantifizierer dahinter), dies bringt bereits einen Performance-Gewinn um mehr als den Faktor 10.
  `/([^"'/]+¦Doublequotes[^"'/]*¦Singlequotes[^"'/]*)¦C-Kommentar¦C++-Kommentar/$1/`
- Jetzt kann man noch bei den Doublequotes und Singlequotes etwas Loop Unrolling betreiben (d.h. `"[^\\"]*(\\.[^\\"]*)*"` bzw. `'[^\\']*(\\.[^\\']*)*'` einfügen), was weitere 25% Performance-Gewinn liefert. (Konkret: Laufzeit roh 36 Sekunden, mit der Zusatz-Zeichenklasse als erste Alternative 3,2 Sekunden, mit der Zusatz-Zeichenklasse auch bei den Doublequotes und Singlequotes 2,9 Sekunden, und mit Loop Unrolling 2,3 Sekunden. Die Performance hängt natürlich davon ab, wie der C-Code beschaffen ist – wenn viele Kommentare enthalten sind, dauert die Suche länger, weil auf die Kommentare erst in den hinteren Alternativen geprüft wird.) Dafür ist die Regex jetzt wirklich kompliziert.

Parsing von CSV-Tabellendokumenten mit Perl

Zuerst einmal ein kleines Perl-Beispielprogramm:

```
(@fields = ();
(push(@fields, $+) while $text =~ m{
( "([^"\\]*(?:\\.[^"\\]*)*)",? # quoted Text mit evtl. Komma dahinter
( ¦ ([^,]+),? # sonstige Fälle
( ¦ , # einsames Komma
(}gx;
(push(@fields,undef) if substr($text,-1,1) eq ',';
```

Optimierungen bei Leerzeichen davor/dahinter („leading/trailing whitespace“):

- `s/^\s+//` und `s/\s+$//`: einfach und effektiv, nur bei sehr langen Strings langsamer als die schnellste kombinierte Lösung
- `s/^\s*(.*?)\s*$/$1/`: sehr langsam, braucht etwa die dreifache Zeit, da viel Backtracking

- `s/^\s*¦\s*$//g`: ebenfalls langsam wegen Alternation
- `s/^\s*(.*\S)?\s*$/$1/`: schnell
- `$_ = $1 if m/^\s*(.*\S)?/`: noch ca. 10% schneller

Optimierungen beim Tausenderkomma in Zahlen:

- `while s/^(-?\d+)(\d{3})/$1,$2/`: Durchläuft die Zahl iterativ, nimmt immer die letzten drei Stellen weg und schreibt sie, durch ein Komma getrennt, dahinter. Man braucht kein `/g`, weil die Zahl rückwärts durchlaufen wird (für die Wiederholung sorgt die Schleife).
- `s/(\d{1,3})(?=(?:\d\d\d)+(?!\d))/$1,/g`: Durchläuft die Zahl nicht rückwärts, sondern packt alles in eine einzige Regex und ist gegenüber dem obigen Ausdruck um 30% schneller.
  Damit dieses Beispiel mit dem ersten wirklich identisch ist, muss man die erste Klammer durch `\G((?:^-)?\d{1,3})` ersetzen, oder umgekehrt im obigen Beispiel `-?` wegnehmen.

Performance-Optimierung

Für einen Computer ist der Vergleich von Zeichenketten eine leichte Sache, entsprechend geht eine Textsuche meist recht schnell, der Flaschenhals ist oft eher das Einlesen der Daten als die Rechenleistung für die eigentliche Suche. Aber man kann auch problemlos Regular Expressions konstruieren, die z.B. zu sehr viel Backtracking führen, wodurch die benötigte Rechenleistung ziemlich hoch wird – nicht weil der Ausdruck so kompliziert ist, sondern weil so wahnsinnig viele Möglichkeiten durchprobiert werden müssen. Darum ist es in manchen Fällen durchaus lohnend, sich über die Performance Gedanken zu machen – und das kann man nur, wenn man versteht, wie die Regex-Engine arbeitet.

Generelle Regeln

- *Wenig Alternation*: Backtracking und damit Alternation ist nach Möglichkeit zu vermeiden. Wenn das nicht möglich ist, sollte man bei Alternation die Alternativen nach ihrer Häufigkeit im Text anordnen, so dass der klassische NFA weniger oft herumprobieren muss.
  - Beispiel: `[uvwxyz]` braucht bei einem String mit nur einem 'u' an der 34. Position entsprechend genau 34 Tests, weil für jeden Buchstaben im Text geprüft werden muss, ob er zu dieser Zeichenklasse gehört. Die Suche nach `(u¦v¦w¦x¦y¦z)` braucht dagegen 204 Tests, weil von jeder Startposition aus 6 Backtracks durchgeführt werden müssen.
  - Beispiel: Die Suche nach einem Datum in der Art `(Jan¦Feb¦...¦Dec)? (31¦[123]0¦[012]?[1-9])` ist ebenfalls desaströs, weil es pro Buchstabe viele Möglichkeiten gibt (alleine 12 Monate, die für jeden Buchstaben im Text durchprobiert werden müssen).
  - String in einfachen oder doppelten Anführungszeichen: `('[^']*'¦"[^"]*")` ist ebenfalls schlecht für die Performance, da ein Backtrack bei jedem einzelnen Zeichen des Textes nötig ist. Manche Regex-Engines können zwar hier optimieren, indem sie mit der Suche nur dort ansetzen, wo ein `"` oder ein `'` gefunden wurde (*fixed string check*); besser ist jedoch z.B. die Verwendung von Perl-Lookahead `(?=['"])`, was etwa 20-30% Performance bringt. Beim obigen Monatsbeispiel bringt ein Lookahead `(?=[ADFJMNOS])` sogar ca. 60% mehr Performance. Bei der Suche nach Anführungszeichen `('.*'¦".*")` könnte man zwar `(['"]).*\1` schreiben, aber das entspricht nicht obigem Beispiel, weil man \1 auch in der Zeichenklasse `[^"]` bzw. `[^']` dranbringen müsste – das geht aber nicht.
- *Häufigere Variante zuerst*: Bei einer Suche nach Text mit Backslash innerhalb von Anführungszeichen ist `(\\.¦[^"\\])*` suboptimal, weil die erste Variante den Backslash plus das folgende Zeichen enthält, was seltener vorkommt als ein beliebiges anderes Zeichen, und die Varianten von links nach rechts durchprobiert werden. Bei einem DFA und einem Posix-NFA kann man nichts ändern, bei einer Suche mit einem klassischen NFA kann man durch die richtige Reihenfolge jedoch etliche Backtracking-Schritte einsparen. Es sind nämlich viele! Für die Suche nach `([^"\\]+¦\\.)` müssen bei einem Posix-NFA 2^{n+1} Backtracks und $2^{n+1}+2^n$ Tests (n: Textlänge) durchgeführt werden!
- *Konstanten Text verwenden*: Die Suche nach festem Text geht sehr schnell, weil nur Zeichen für Zeichen verglichen werden muss – es ist kein Backtracking nötig; entweder es passt oder es passt nicht, und das ist sofort klar. Je mehr fester Text in einer Regex vorhanden ist, desto weniger Möglichkeiten bleiben für die flexiblen Bestandteile (Alternation, Quantifizierer usw.) übrig und entsprechend weniger muss dafür herumprobiert werden. Wenn der konstante Text am Anfang der Regex steht, muss die zeitraubende Suche nach dem Rest nur dort durchgeführt werden, wo der konstante Anfang bereits passt.

- *Textanker verwenden*: Die Suche nach Textankern geht ähnlich schnell wie bei einzelnen Zeichen oder Zeichenklassen, da nur immer zwei benachbarte Zeichen verglichen werden müssen. Somit bieten Textanker die selben Vorteile wie konstanter Text.
  Beispiel: Die Suche mit `.*§` ist desaströs, wenn das gesuchte Zeichen '§' nicht im String ist; dann wird zuerst von `.*` der gesamte String beansprucht, dann nach und nach ein Zeichen abgegeben, bis keiner übrig ist; anschließend geht es weiter zum zweiten Buchstaben des Strings, wo wieder `.*` den gesamten Rest beansprucht usw. Wenn die Suche dagegen als `^.*§` formuliert ist, erfolgt der ganze Durchlauf nur ein einziges Mal.
  Außerdem: Die Position des Textankers muss geschickt gewählt werden. Wenn der Suchbegriff ziemlich weit hinten im String steht, ist es oft suboptimal, als Textanker `^` zu verwenden, weil es dann dazwischen viele Möglichkeiten gibt; statt dessen einen Textanker, der näher am Suchmuster steht.
- *Klammern reduzieren*: Jede Klammer (solange sie nicht „non-capturing“ ist) bedeutet, dass bei jedem Vergleichsschritt, bei dem der Klammerinhalt auf den Suchtext passt, dieser Suchtext in einen Speicher kopiert wird – und das kostet Zeit, wenig bei jedem Schritt, aber es summiert sich. Wenn man auf eine Klammer nicht verzichten kann, dann sollte sie den Quantifizierer mit einschließen (d.h. `(.*)` statt `(.)*`), damit der Kopiervorgang nicht für jeden einzelnen Buchstaben durchgeführt werden muss. Es gibt einen Unterschied zwischen beiden Varianten; wenn der Quantifizierer in der Klammer ist, enthält diese am Ende eine ganze Menge von Buchstaben, während im zweiten Fall (Quantifizierer außerhalb) nur ein Buchstabe, nämlich der letzte, auf den die Suche passt, gespeichert wird. Wenn man nur diesen Buchstaben sucht, nimmt man trotzdem `(.*)` und extrahiert daraus den letzten Buchstaben z.B. mit `strpos()`.
- *Konstanten Text raus aus den Klammern*: `"(.*)"` speichert den Text zwischen zwei Doublequotes und tut dies, sobald bei der Suche ein Quote-Zeichen gefunden wurde. Eine Suche nach `("*")` würde dagegen nicht erst nach einem gefundenen Quote-Zeichen beginnen, den Inhalt bei jedem Suchschritt zu kopieren, sondern grundsätzlich bei jedem Zeichen (=> viel Overhead).
  Übrigens ist das Geschwindigkeitsverhältnis zwischen beiden Varianten bei der Verwendung des `*?`-Quantifizierers bei Perl umgekehrt, weil der Quantifizierer anders herum arbeitet (er nimmt nicht zuerst die maximale Buchstabenanzahl und gibt nach und nach einen Buchstaben ab, sondern nimmt immer mehr).
- *Weniger ist mehr*: Um Abkürzungen wie `he'll` und `don't` zu suchen, kann man sowohl `\w'(ll¦nt)` als auch `\<\w+'(ll¦nt)` verwenden – wobei die erste Version allerdings zehnmal schneller ist, weil sie sich nur für das interessiert, was für die Entscheidung wichtig ist (nämlich ein Buchstabe, dahinter Apostroph, dahinter die Endung), und auf den Wortanfang verzichtet.
- *Passende Werkzeuge*: Bei einfachen Suchproblemen lieber einfache Funktionen wie `strtok()` verwenden, die viel effizienter (aber auch entsprechend unflexibler) als Regular Expressions sind.
- Perl: `\G(this¦that)` statt `\Gthis¦\Gthat` verwenden (da Letzteres nicht optimiert wird); bei Rückbezügen `$1` statt `$&` verwenden.
- Tcl: Hier kann man sich zu einer Regex-Suche angeben lassen, an welcher Stelle des Suchtextes die Suche das letzte Mal erfolgreich war, und kann diese Information nutzen, statt noch einmal genauso von vorne zu suchen.

Anders herum betrachtet – folgende Dinge gehen schnell und können deshalb problemlos verwendet werden:

- der Vergleich von Zeichen

- der Vergleich von Zeichenklassen
- Textanker (dazu müssen nur jeweils zwei benachbarte Zeichen verglichen werden)
- Klammern für Rückbezüge bringen zwar einen gewissen Overhead zum Speichern des Inhaltes, aber sie erzeugen – im Gegensatz zu Klammern, die Alternation enthalten – keine zusätzlichen Backtracking-Schritte.

Loop Unrolling

Hier wird das schon vorher erwähnte Verfahren des Loop Unrolling anhand der Suche von Text in Anführungszeichen (incl. durch Backslash geschützte Anführungszeichen innerhalb) erklärt. Hintergedanke: Alternation führt (bei einem NFA) zu Backtracking, d.h. die Regex-Engine muss mehrere Möglichkeiten durchprobieren (was einer Wiederholungsschleife entspricht, daher „Loop"); wenn man dagegen die Regex so umbaut, dass die Suche in einem Durchgang erledigt werden kann, wird sie dadurch merklich beschleunigt – wie als wenn man eine Schleife nimmt und „ausrollt", so dass die Befehle hintereinander stehen und ohne Wiederholung in einem Rutsch ausgeführt werden können.

Beim erwähnten Beispiel mit den Anführungszeichen schreibt man also statt:
`"(\\.¦[^"\\]+)*"`
lieber:
`"[^"\\]*(\\.[^"\\]*)*"`
Allgemein formuliert („öffnend" und „schließend" meint die jeweiligen Begrenzer):
`öffnend normale_Zeichen*(spezielle_Zeichen normale_Zeichen*)* schließend`
Loop Unrolling macht die Regex also länger (weil die Menge der normalen Zeichen zweimal vorkommt) und unübersichtlicher, aber eben auch schneller.

Herleitung aus einem konkreten Beispiel:
`"Er sagte \"Servus!\" und ging."`
Darauf passt folgende Regex:
`"[^"\\]+\\.[^"\\]+\\.[^"\\]+"`
Man sieht, dass der hintere Teil doppelt vorkommt, d.h. man kann ihn einmal nehmen, klammern und dahinter einen Quantifizierer schreiben. Weil die mit Backslash geschützten Anführungszeichen beliebig oft vorkommen können (auch 0-mal) und außerdem ohne „normalen Text" dazwischen (z.B. `"blabla\"\"blabla"`) beliebig oft direkt hintereinander, kann man als Quantifizierer den Stern nehmen. Das war's.

Zu beachten ist:

- Die Mengen der normalen und der speziellen Zeichen dürfen sich nicht überlappen, ansonsten ergibt sich eine „endlose" Suche (d.h. ein Posix-NFA probiert dann alle Möglichkeiten durch, was wirklich lange dauert).
- Es kann Probleme geben, wenn die Menge der speziellen Zeichen auch „nichts" finden kann (d.h. wenn als Quantifizierer ein Stern oder Fragezeichen verwendet wird). Genau überdenken, was in so einem Fall passiert!
- Was durch eine Instanz der speziellen Zeichen gefunden wird, darf nicht gleichzeitig durch mehrere Instanzen gefunden werden. Beispiel Pascal-Kommentare (stehen in geschweiften Klammern, normale Sequenz = innerhalb von geschweiften Klammern, spezielle Sequenz = ein oder mehrere Leerzeichen):
  `(\{[^}]*\})*( +(\{[^}]*\})*)*`
  Hier wird das Leerzeichen sowohl vom Plus direkt dahinter als auch vom Stern hinter der Klammer vervielfacht. Lieber entfernt man das Plus, und weil Leerzeichen häufiger auftreten

dürften als Kommentare, vertauschen wir noch Normal- und Spezialfall:
`*(\{[^}]*\} *)*`
Verschachtelte Quantifizierer sind immer verdächtig und können zu Problemen führen, müssen sie aber nicht. Beispiel:
`^Subject: (Re: *)*`
Dies ist richtig, weil das „Re:“ dafür sorgt, dass der zweite Stern nicht ebenfalls ein mehrfaches Leerzeichen akzeptiert.

Noch ein Anwendungsbeispiel für Loop Unrolling: Suche nach Domain-Namen; hier gibt es keine Begrenzer, das spezielle Zeichen ist der Punkt, und bei der ersten Menge der normalen Zeichen brauchen wir als Quantifizierer ein Plus, weil ein Domain-Name nicht mit Punkt beginnen darf:
`[^.]+(\.[^.]*)*`

Automatische Optimierungen durch die Software

- *Fixed String Check*: Falls die Regex mit einem oder mehreren konstanten Zeichen (statt Zeichenklassen) beginnt, sucht die Engine diesen String mit Hilfe konventioneller Algorithmen (z.B. Algorithmus von Boyer-Moore), und wendet die aufwändigere Regex-Suche erst dann an, wenn dieser Anfang gefunden wurde. Man muss also, falls möglich, dafür sorgen, dass am Regex-Anfang ein oder mehrere konstante Zeichen stehen, z.B. `th(is¦at¦em)` statt `(this¦that¦them)` (nur manche Programme wie z.B. Perl können erkennen, dass der fixed string check auch beim zweiten Beispiel möglich ist).
- *First Character Discrimination*: Erkennt „Rausschmeißer“ vor einer Klammer, damit diese nicht jedes Mal den Inhalt speichert, sondern nur, nachdem der erste Buchstabe passt. Bei einem DFA ist es unnötig (da Geschwindigkeit konstant ist), manche NFA beherrschen dieses Verfahren (z.B. *Emacs*, sehr gut), manche jedoch eher halbherzig (*Perl*, *Tcl*).
  Beispiele: Z.B. bei `<(.*)>` muss der Inhalt der Klammer nur dann gespeichert werden, wenn ein '<' davor steht. Ebenfalls sinnvoll bei Alternativen: `[JFMASOND]` vor einer Monatsliste vereinfacht die Arbeit, weil nur nach bei diesen Buchstaben das volle Prüfprogramm (d.h. alle Monatsnamen durchprobieren) gefahren wird.
  Zu beachten: Weil die First Character Discrimination nicht unbedingt gut implementiert ist, kann es sinnvoller sein, statt in einer großen Alternation alle Ausdrücke auf einmal durchzutesten, sie einzeln nacheinander zu prüfen – bringt beispielsweise einen Geschwindigkeitsvorteil um den Faktor 6. Emacs hat jedoch eine sehr gute First Character Discrimination, dort ist die große Alternation z.B. um 40% schneller, wenn der interne Regex-Cache bei den vielen einzelnen Regexen überläuft, sogar um 280%. Außerdem ist zu bedenken: Was sucht man? Die Alternation findet immer das erste passende Wort im Text, während bei einer Suche mit mehreren einzelnen Regexes natürlich die erste Suche (d.h. der erste Suchbegriff) das erste Ergebnis liefert.
- *Optimierung spezieller Ausdrücke*: Einfache Ausdrücke wie z.B `x*` oder `[a-f]` werden oft intern optimiert. Darum ist auch `.*` oft deutlich schneller als `(.)*`. (Bei Perls `*?` findet keine derartige Optimierung statt, d.h. die letzten beiden Beispiele sind dort gleich schnell.)
  Ebenso: `xxx` ist normalerweise schneller als `x{3}`.
- *Längenanalyse*: Bestimmung der Mindestlänge der Regex, d.h. so viele Buchstaben vor String-Ende kann abgebrochen werden; DFAs sind hier besser, weil sie dort mit allem auf einmal aufhören.
- *Posix-NFA*: Wenn ein Treffer bis zum String-Ende reicht, muss nicht weitergesucht werden, weil eh kein längerer Treffer möglich ist; wegen Greediness passiert das meist bereits ziemlich am Anfang => große Einsparung.

- *Vereinfachte Suche*: Wenn nur getestet werden soll, ob der Suchbegriff überhaupt gefunden wird (und nicht das längstmögliche Suchergebnis interessiert), kann beim DFA und Posix-NFA nach dem ersten Fund die Suche abgebrochen werden.
- *Zeilenanker*: Wenn die Regex ein '^' enthält, muss nur am Anfang des Strings gesucht werden, statt den kompletten String durchzugehen. Manche Programme erkennen sogar Ausdrücke, die die gesamte Zeile umfassen (z.B. wenn am Regex-Anfang `.*` steht), und hängen intern ein '^' davor – denn wenn die Suche nicht gleich am Anfang der Zeile erfolgreich ist, wird sie es später auch nicht sein.
- *Rare Character*: Perl weiß über die Häufigkeiten von Buchstaben (ermittelt aus C-Code und englischen Texten) bescheid (selten = "Q", häufig = "e" oder Leerzeichen); wenn das seltenste Zeichen, das im Suchbegriff vorkommt, nicht im momentan durchsuchten Text vorhanden ist, wird die Regex gar nicht weiter angewandt, sondern der nächste Textabschnitt angeschaut.
- *Caching*: Die Regular Expression muss vor der eigentlichen Suche kompiliert werden, d.h. sie wird u.a. umgesetzt in eine Tabelle erlaubter und nicht erlaubter Zeichen. Die Suche wird also beschleunigt, wenn nach der selben Regex bereits einmal gesucht wird und die kompilierte Version noch im Cache vorhanden ist. Beispiele:
  - *grep*: Kompiliert pro Aufruf nur einmal, d.h. wenn man den Suchtext in mehreren Dateien sucht, sollte man bei grep alle diese Dateien angeben, statt in jeder Datei einzeln mit einem grep-Aufruf suchen.
  - *Perl*: Wenn die Regular Expression eine Konstante ist, wird bei der nächsten Suche die kompilierte Version verwendet. Wenn die Regex in einer Variable steckt, wird zuerst noch ein Stringvergleich durchgeführt, der schaut, ob der Variableninhalt gleich der Regex, die kompiliert wurde, geblieben ist – wenn ja, kann die kompilierte Version verwendet werden.
  - *Tcl*: Hier ist die Regex-Suche eine Funktion (Perl: ein Operator), die nichts über ihre Parameter weiß (z.B. ob sie Konstanten oder Variablen sind). Es gibt jedoch einen Cache über die letzten fünf verwendeten Regex, d.h. wenn man nicht mehr als fünf Regular Expressions abwechselnd verwendet, können diese immer aus dem Cache geholt werden.
  - *Python*: Hier gibt es ein Regex-Objekt, das mit `regex.compile` kompiliert und ganz normal gespeichert wird (wie eben eine Variable).
  - *DFA*: Manche Engines verlagern Teile der Compile-Arbeit bis zu dem Zeitpunkt, wenn ein Ergebnis gefunden wurde – ansonsten wurde lange an der Regex kompiliert, ohne dass etwas gefunden wurde.

GNU-egrep besitzt intern zwei Engines, nämlich einen DFA und einen NFA. Für die gewöhnliche Suche wird der DFA benutzt, aber wenn Dinge vorkommen, die einen NFA erfordern (z.B. Backreferences), wird dieser verwendet. Ähnlich funktioniert gawk, zur Suche wird der DFA verwendet, aber wenn die genaue Position des Suchergebnisses bekannt sein muss oder wenn Klammerninhalt gespeichert werden muss, wird der NFA benutzt.

Regex-Benchmarks

Bei Performance-Tuning muss man sich immer die Frage stellen, auf welchen Apekt das Tuning abziehlt. Während ein DFA immer gleich schnell ist, kann eine Verbesserung nur einem klassischen NFA zugute kommen, nur einem Posix-NFA, oder auch beiden. Außerdem kann es sein,

Um eine Vorstellung zu bekommen, wie schnell eine Regex-Engine ist, hilft nur Benchmarking. Außerdem kann es sein, dass man nicht weiß, welcher Engine-Typ von der Software verwendet wird

– auch das lässt sich mit Benchmarking herausfinden. Wir wissen nämlich, dass ein DFA immer relativ schnell ist (unabhängig davon, ob die Suche gelingt oder scheitert), während ein klassischer NFA nur bei einer erfolgreichen Suche schnell ist (weil er dann das Backtracking abbricht) und ein Posix-NFA immer langsam ist (weil er das Backtracking immer bis zum Ende durchzieht).

Erster Test: Sorgt dafür, dass die Suche möglichst oft scheitert, d.h. ein DFA ist bei diesem Test schnell, ein NFA ist langsam.
Regex: `X(.+)+X`
Suchtext: `=XX==========================================================`

Zweiter Test: Jetzt soll die Suche gleich ziemlich am Anfang erfolgreich sein, d.h. ein klassischer NFA findet die Übereinstimmung ziemlich schnell und hört dann auf, während ein Posix-NFA alle weiteren Möglichkeiten durchprobiert und deshalb deutlich länger braucht.
Suchtext: `=XX==========================================================X`

Dritter Test: Die erste (und einzige) Übereinstimmung erstreckt sich bis zum letzten Zeichen. Eine intelligente Posix-NFA-Engine könnte daraus schließen, dass sie nicht weitersuchen muss, weil keine längere Übereinstimmung möglich ist, wodurch sie genauso schnell wie ein klassischer NFA wäre. Also verlängert man den Suchtext, so dass er länger als die Übereinstimmung ist – dann muss auch eine intelligente Posix-NFA-Engine alle Möglichkeiten durchprobieren.
Suchtext: `=XX==========================================================X===`

Außerdem muss man noch bedenken, dass sich z.B. mawk wie ein DFA verhält, wenn man lediglich einen Text *sucht* – erst wenn man Text *ersetzt*, kommt ihr Posix-NFA-Charakter zum Vorschein.

Perl

Perl wird im Buch von Jeffrey Friedl so ausführlich behandelt, dass es ein eigenes Kapitel spendiert bekommt. Hier ist der Inhalt grob stichpunktartig und weitgehend unsortiert aufgeführt:

- `+?`, `*?`, `??`, `{}?` sind die non-greedy-Varianten der entsprechenden normalen Regex-Metazeichen; Beispiel: Suche nach HTML-Tags `/<[a-zA-Z]*?>/`
- Manchmal werden diese non-greedy-Quantifizierer als Ersatz für Zeichenklassen missbraucht (d.h. `<(.+?)>` statt `<([^>]+)>`). Zeichenklassen sind jedoch grundsätzlich effizienter, außerdem findet der erste Suchausdruck ohne /s-Modifier kein Newline-Zeichen. Außerdem: Im Suchtext `eins <zwei> drei <vier> fünf` findet der erste Suchausdruck `<zwei>`, der zweite jedoch `<zwei> drei <vier>`.
- `(?` öffnet „non-capturing parentheses“, d.h. Klammern, deren Inhalt nicht gespeichert wird – z.B. wenn man Klammern nur zum Gruppieren verwendet, wird die Regex dadurch etwas schneller (da kein Overhead zum Speichern des Inhalts nötig ist), man kann mit dieser Klammer optimieren (z.B.: `(?:return-to¦reply-to)` zu `re(?:turn-to¦ply-to)`), und man kann damit problemlos der Regex neue Klammern hinzuzufügen, ohne die Nummern der Backreferences anpassen zu müssen.
- `(?#` (z.B. `(?#blabla)`) ist ein Kommentar; wird bereits beim ersten Parsing entfernt, d.h. die schließende Klammer ist das einzige Perl-Regex-Element, das nicht escaped werden kann: die erste Klammer nach `(?#` beendet den Kommentar.
- Suche:
  - Syntax: `$variable =~ m/regex/i`; prüft, ob die Regex auf die Variable zutrifft.
  - Die Vergleichsoperatoren in Perl lauten '==' für den exakten Vergleich, '=' für die Zuweisung, '=~' für Regex-Vergleich und 'eq' für String-Vergleich.
  - `$is_match = m/regex/` ist gleich wie `$is_match = ($_ =~ m/regex/)`
  - Regex-Vergleichsoperator `=~`: Das Ergebnis wird verworfen, weil nur die Seiteneffekte wie $1 interessieren. Außerdem wird eine neue Default-Regex gesetzt (`m//`), bei einer Suche mit /g wird die Position aktualisiert.
  - Je nach Kontext liefert `=~` im skalaren Fall Boolean (gefunden bzw. nicht gefunden), oder ein Array `($tag, $monat, $jahr) =~ m/^(\d+)\.(\d+)\.(\d+)$/` oder eine Liste `%liste =~ m/^([a-zA-Z]+): (.+)/mg` (Liste von EMail-Headern + Werten).
- Ersetzen: `$variable =~ s/regex/replace/` (Rückgabewert: true/false bzw. Anzahl der Ersetzungen)
- Wenn keine Variable gegeben ist, wirken diese Operatoren auf $_.
- split:
  - Gegenteil zum Match-Operator, d.h. `m/:/g` auf `a:b:c:d` liefert Array mit drei ':', während `split(/:/,"a:b:c:d")` ein 4er-Array mit 'a', 'b', 'c', 'd' liefert.
  - split ist ein Operator, obwohl es wie eine Funktion aussieht (die Klammern kann man ab Perl5 weglassen).
  - bei split wird /g implizit gemacht
  - Wenn das zweites Argument fehlt, wird $_ genommen.

- Drittes Argument: maximale Anzahl, z.B. wenn man nur in den ersten beiden Feldern interessiert ist, trägt man dort eine drei ein (drittes Feld für Rest). Es werden, falls der Parameter positiv ist, auch leere Felder am Ende zurückgeliefert, wenn die gewünschte Anzahl größer als die tatsächliche Feldanzahl ist (Parameter -1 = „unendlich groß“).
- Auch zwischen mehrfach vorhandenen Trennzeichen wie „::“ werden leere Felder zurückgegeben.
- Leerer Match möglich: split mit `\W*` liefert bei `abc def` die Buchstaben einzeln (ohne Leerzeichen, Trennzeichen usw.), aber kein leeres Feld am Anfang; bei einem nichtleerem Match-Operator auf `:a:b` würde dagegen ein leeres erstes Element entstehen.
- `split(//,text)` bedeutet, den Suchtext an jedem Zeichen zu trennen (also ganz anders als `m//`).
- Suche nach `??` hat keine besondere Wirkung
  $&, $1 usw. funktionieren nicht
  `/ /` hat spezielle Bedeutung: wie `/\s+/`, aber whitespace am Anfang wird ignoriert (d.h. wie bei awk's default input separator).
- Defaultwerte: "split" = `split(/ /, $_, 0)`
- Klammern im Ausdruck: Klammerwert wird mit eingefügt; z.B. `split(/(<[^>]*>)/)` auf `Text <b>fetter Text</b>` liefert `Text`, `<b>`, `fetter Text`, `</b>` usw.). Es ist sinnvoll, dafür non-capturing-Klammern `(?:` zu verwenden.

- Man kann bei `m/foo/` andere Zeichen als Begrenzer nehmen (empfohlen: `%¦#!,`), auch Klammern (paarweise!) funktionieren: `()[]<>{}` (Vorteil: können verschachtelt werden). Bei einer Ersetzung müssen Klammern entsprechend „innen doppelt“ auftreten: `s<a><b>`.
- Singlequotes als Begrenzer (z.B. `m'foo'`): keine Doublequotes-Interpretation (d.h. z.B. '$' wird nicht als Variablenanfang betrachtet)
- Fragezeichen als Begrenzer (z.B. `m?foo?`): nur der erste Treffer wird gemeldet, alle folgenden nicht erkannt (z.B. Email-Header: erster Match reicht; bei mehreren Versionen soll die erste geliefert werden)
- Bei "/" oder "?" als Begrenzer ist das "m" optional; man kann den Begrenzer bei "=~" auch komplett weglassen (und m) => wird wie ein Perl-Ausdruck (außer wenn in Doublequotes), dann wie ein String betrachtet, dann an die Engine verfüttert;
- `m//` entspricht der letzten Regex-Suche und verwendet dabei die noch im Cache befindliche Regex ohne Rekompilierung, inklusive vorhandener Modifizierer (darum macht `m//gi` keinen Sinn).
- Regex-Zeichenklassen in Perl:
  `\s` (Whitespace, `[ \f\t\r\n]`)
  `\S` (Gegenteil von Whitespace)
  `\w` (Wortbestandteil, `[a-zA-Z0-9]`)
  `\W` (Gegenteil davon)
  `\d` (Ziffern [0-9])
  `\D` (Gegenteil davon)
  `\c` (Kontrollzeichen)
- Backslash-Escapes:

- `\000` = oktal
- `\x00` = hexadezimal
- `\055` ist der Bindestrich, wird in der Regex jedoch nicht als Metazeichen interpretiert (d.h. `[a\055z]` ist nicht gleich `[a-z]`), in einem String jedoch schon, d.h. `$string="a\055z";` `[$string]` ist identisch zu `[a-z]`
- Bei `\d` in einem String wird der Backslash entfernt, d.h. für eine anschließende Regex-Suche muss der Backslash doppelt sein.

- `\A` und `\Z` sind Stringanfang bzw. Ende; Ausnahme: sowohl `$` als auch `\Z` matchen vor einem Newline am String-Ende, d.h. man muss für das absolute Ende eines Strings schreiben `(?!\n)$` (Perl5).
- Perl hat kein Symbol für „start-of-word“ und „end-of-word“, sondern die Wortgrenze `\b` (Achtung, in anderem Kontext bedeutet diese Sequenz den Backspace). Als Wort zählen Buchstaben und der Underscore; dies kann emuliert werden durch `\b(?=\w)` und `\b(?!\w)`
  Problem: z.B. `\b\$3.75\b` suchen ($ ist kein Wortbestandteil, d.h. davor kann keine Wortgrenze sein); Lösung: Wenn `m/^\w/` matcht, ein `\b` davor hängen; wenn `m/\w$/` matcht, ein `\b` danach anhängen; oder: `(?:\W¦^)Suchbegriff(?!\w)`
- `\G` ist ein Anker für das Ende des letzten gefundenen Suchbegriffs (bei einer Suche mit /g); gab es vorher noch keine Fundstelle, ist die Position identisch mit dem Anfang des Strings. Beispiel: Suche mehrerer Postleitzahlen hintereinander, die alle mit 85 beginnen: `\G(85\d{3})`
- Perl merkt sich Position des letzten Matches:
  `$data =~ m/<taganfang>/g`
  `@num = $data =~ m/\d+/g`
  `pos($data)` liefert diese Stelle
- Kommandozeilen-Optionen:
  `perl -e 'regex'`: führt den Befehl gleich auf der Kommandozeile aus
  `perl -e -i -p 'regex' Datei`: führt die Änderungen in der Datei aus und schreibt sie gleich zurück
- Wenn Perl mit Debugging Info übersetzt ist, dann man mit `perl -Dr` Regexes debuggen.
- `$/ = ".\n"`: Die spezielle Variable '$/' ändert das Zeilenende-Zeichen für Regex-Matches; hier: Punkt-Zeilenumbruch wie bei E-Mails (SMTP!).
- Modifizierer: Ein Modifizierer bestimmt Interpretation der Regex (/o, /x), des Texts (/i, /m, /s) oder der Anwendung (/g) des Suchbegriffs.
  - 'i': case insensitive
  - 'g': ersetzt alle Vorkommen statt nur eines
  - 'o': Regex einmal kompilieren und im Folgenden ungeprüft kompiliert weiterverwenden
  - 'x': Damit werden Leerzeichen ignoriert, man kann also komplizierte Regex-Ausdrücke übersichtlich geschrieben und eingerückt über mehrere Zeilen verteilen und außerdem Kommentare (mit #) hinzufügen. Leer- und Anführungszeichen haben dabei „umgedrehte“ Wirkung, d.h. Leerzeichen sind keine normalen Zeichen mehr => echte Leerzeichen muss man mit Backslash escapen, und Anführungszeichen sind normale Zeichen (d.h. nicht mehr String-Begrenzer, müssen nicht escaped werden)
    Beispiel: `/"Ein\ Satz"/x`

- 'm', 's': Multi-Line oder Single-Line (d.h. String ist eine einzige Zeile)
  z.B. *undef $/* sorgt dafür, dass es kein Zeilenende mehr gibt => Regex wirkt auf den gesamten String statt auf eine Zeile
  - default mode: ^$ matchen Stringanfang bzw. -ende; Punkt matcht keine Newline
  - single line: ^$ matchen Stringanfang bzw. -ende; Punkt matcht alle Zeichen
  - multi line: ^$ matchen Zeilenanfang bzw. -ende; Punkt matcht keine Newline
  - clean multi line: ^$ matchen Zeilenanfang bzw. -ende; Punkt matcht alle Zeichen
- 'e': man kann eigene Funktionen einbauen, z.B. *s/regex/&meineFunktion/e*, wobei *meineFunktion* einen beliebigen Text zurückliefert. Hier kann es Sinn machen, diesen Modifizierer mehrfach anzugeben, so dass er mehrfach aufgerufen wird und die Ersetzung nochmals auf das vorige Ergebnis anwendet.

Modifizierer dürfen wiederholt auftreten und ihre Reihenfolge ist egal, d.h. ein Ausdruck wie `learn/by/osmosis` ist erlaubt.

- Beachte: Der Operator m oder s (am Anfang einer Regex) ist nicht das Gleiche wie der Modifizierer m oder s (am Ende der Regex)!
- Wenn man einen Modifizierer nicht für die gesamte Regex haben möchte, sondern nur für einen Teil, kann man das mit der Syntax `(?modifier)` erreichen: `/(?iblabla)/` ist identisch zu `/blabla/i`; man kann auch mehrere Modifizierer kombinieren, z.B. `(?ig)`.
- $* sollte man nicht verwenden; wenn nötig (bei Scripts mit Perl4-Libs):
  `{local ($^W)=0; eval '$*' = 1;}`
- Lookahead: Testet, was danach kommt; Beispiel für positives Lookahead:
  `/Festessen(?=Champagner)/`
  (findet „Festessen“ nur dann, wenn dahinter „Champagner“ kommt);
  Beispiel für negatives Lookahead:
  `/Festessen(?!Lebensmittelvergiftung)/`
  (findet „Festessen“ nur dann, wenn dahinter nicht „Lebensmittelvergiftung“ kommt)
- Lookbehind: Analog zu Lookahead, die Metazeichen lauten `?<` bzw. `?<!`.
- Beispiele für Lookahead/-behind:
  - `\d+(?=[^.])` und `\d+(?!\.)`: Ersteres findet bei `OH44272` nur `4427` (weil danach noch ein Nicht-Punkt benötigt wird), Letzteres die ganze Zahl. Besser: `\d+(?=[^.\d])` und `\d+(?![.\d])`.
  - `^(?![A-Z]*)[a-zA-Z]*$`: findet Buchstaben, aber nicht dann, wenn es nur Großbuchstaben sind
  - `(?!000)\d\d\d`: findet drei Ziffern, aber nicht `000`
- Lookahead empfiehlt sich generell am Anfang der Regex, um Sonderfälle auszuschließen bzw. die Suche zu beschleunigen, oder am Ende der Regex, um einen Match zu verhindern, wenn etwas Unerlaubtes danach kommt
- Vorsicht bei negativem Lookahead und Wörtern: `(?!cat)\w+` findet bei `cattle` immer noch `attle` => korrekt würde man so suchen: `\b(?!cat)\w+`
- Mit dem Operator 'm' kann man den Regex-Begrenzer festlegen, d.h. statt `m/regex/` kann man auch `m#regex#` (oder ein beliebiges Zeichen statt der Raute) schreiben. `m/regex/` lässt sich

abkürzen als `/regex/`.

- In den Variablen $1, $2, $3 landen nach der Regex-Suche die entsprechenden Inhalte der Klammern. Achtung: $0 ist Name des Scripts und hat nichts mit Regex zu tun!
- $1, $2, $3 auch außerhalb der Regex sichtbar, innerhalb sollte man \1, \2, \3 nehmen;
- Für Rückbezüge sollte man lieber \1, \2, \3 nehmen als $1, $2, $3. Sie sind nicht identisch; $1 steht als normale Perl-Variable erst nach der Auswertung der Regex zur Verfügung, während \1 eine interne Regex-Variable ist, die nur innerhalb der Regex gilt, dort jedoch schon während der Regex-Auswertung zur Verfügung steht. Für eine Suche mit /g bedeutet das, dass $1 erst ganz am Ende der Suche gesetzt wird; bei einer Ersetzung mit /g wird $1 dagegen nach jedem Durchlauf ausgewertet – was aber immer noch heißt, dass sich sein Inhalt während eines Durchlaufs immer auf das vorige Suchergebnis bezieht.
- Prinzipiell gibt es die Backreferences \1 bis \9, wenn aber entsprechend viele Klammern vorhanden sind, sind auch \10, \11 usw. definiert.
- $&: aktueller Klammer-Match
  $+: letzter Klammer-Match
  $`: Text vor dem aktuellen Klammer-Match; wenn man nicht den Text ab Stringanfang haben will, sondern – bei einer Suche mit /g – seit dem letzten Match, kann man schreiben:
  `\G([\x00-\xFF]*?)`
  $': Text nach dem aktuellen Klammer-Match
- Quoting: Bei langen Strings kann die Funktion *quotemeta()* interessant sein, die den übergebenen String quotet. Innerhalb eines RegEx erreicht man das Selbe mit
  `\Qfoo\E`: foo wird Regex-escaped, d.h. `./test *` wird zu `\.\/test \*`
- `\U` und `\L` liefern „uppercase“ und „lowercase“ des Suchbegriffs
- `{3}?` ist identisch zu `{3}`
  `*?` ist sinnlos (und fehlerhaft Perl4)
- Bei der Suche nach `(text1(text2?)+")` ist in Perl5 $2 undefiniert, wenn beim letzten Durchgang der Suche (Plus hinter Klammer) `text2` nicht vorhanden ist; in Perl4 hat dagegen $2, wenn `text2` vorher einmal gefunden wurde, immer noch diesen Wert gespeichert.
- Strings in Doublequotes sind nicht konstant, sondern werden ausgewertet (z.B. eingebettete Variablen, auch Funktionsaufrufe). Echte konstante Strings sind in Singlequotes eingeschlossen. Es gibt noch eine alternative Quoting-Möglichkeit, falls man Doublequotes bzw. Singlequotes vermeiden will: dd/bla/ (bzw. dd_bla_) statt Doublequotes, d/bla/ statt Singlequotes.
- Ein Regex-Ausdruck verhält sich ähnlich wie ein String in Doublequotes (d.h. Variablen sind möglich usw.), aber z.B. `\b` ist Word Boundary statt Backspace, oder `\3` ist Referenz statt Oktalzeichen, `$var[2-7]` ist Variable+Zeichenklasse statt Array (Abhilfe: als `${var[2-7]}` schreiben).
- Regex in einer Stringvariablen:
  $str = "regex # Kommentar\n";
  Hier ist der Zeilenumbruch am Ende nötig, weil sonst der Kommentar in der Regex (mit /x-Modifizierer möglich) nicht beendet wird.
- Zeichenklasse „sichtbare Zeichen“: `[!-~]` bzw. `[\x21-\x7e]`
- Array sortieren, wobei 'ü' bei 'u' steht usw. statt nach ASCII-Wert: Kopie des Arrays anlegen, 'ü' usw. durch 'u' ersetzen, sortieren, Sortierreihenfolge auf das Original-Array übertragen.

- Perl4: Die Suche mit `\d*` auf `123` liefert 2 Ergebnisse, nämlich einmal die Zahl und einmal den Leerstring am Zeilenende. Perl5 liefert korrekterweise ein Ergebnis, da dort eine erfolgreiche Suche auch dann, wenn das Ergebnis keinen Buchstaben umfasst, die Position im Text um ein Zeichen weiterrückt.
- Die Funktion *study* erzeugt zu einem String einen Suchindex (d.h. eine Liste mit Positionen der Buchstaben). Das beschleunigt die Suche, aber vergrößert den Speicherplatz (braucht ca. die vierfache Stringgröße) => sinnvoll bei kurzen (nicht zu kurzen) Strings mit häufigen Regex-Checks (Regex mit konkreten Buchstaben statt Platzhaltern) ohne Modifikation des Suchtexts zwischendurch.
- Die Funktion *reset* setzt eine Regex-Suche zurück.
- Wenn die Regex zur Laufzeit noch nicht bekannt ist, kann es Sinn machen, den entsprechenden Programmcode in einen String zu packen und dann bei Bedarf an eval() zu vefüttern – dadurch wird der Code erst ausgewertet, wenn die Regex bekannt ist. Dies ist oft sinnvoll mit mehreren Regular Expressions, vielen Suchtexten und /o. Der Aufruf von eval kostet jedesmal Zeit, dafür wird bei jeder Anwendung die Regex-Kompilierung gespart.
- Effizienzbetrachtungen:
  - Kein $&, $`, $' verwenden (oft nicht möglich, da viele Libraries Carp.pm nutzen) – wenn eine dieser Variablen einmal auftaucht, wird alles langsamer, da sie sich auf den Originalstring beziehen => wenn der String verändert wird (z.B bei bei s///), müssen Kopien (nach dem erfolgreichen Match) gemacht werden.
    Ersatz: $` durch `(.*?)` am Anfang, $& durch Klammern außenrum, $' durch `(?=(.*))` am Ende der Regex ersetzen.
  - capturing parentheses bewirken ebenfalls eine Kopie des Inhalts, d.h. nach Möglichkeit darauf verzichten.
  - /i bewirkt auch eine Kopie des Texts, die in Kleinbuchstaben umgewandelt wird (und zwar vorher, d.h. zusätzlich zu $& und den Klammern). Außerdem wird eine Kopie der Regex angefertigt, die ebenfalls zu Kleinbuchstaben konvertiert ist.
  - /ig macht bei jedem Durchgang eine Kopie des ganzen Zielstrings, die umso kürzer wird, je weiter nach hinten man im String kommt.
  - Optimierung von "s/\s+$//": Wenn die Ersetzung vorher bekannt ist (d.h. keine Variable enthält) und kürzer oder gleichlang als das Suchergebnis ist, wird der String verkürzt statt umkopiert. Dies wird aber nur durchgeführt, wenn keine anderen Kopieraktionen stattfinden (wie bei /i oder $&).
- Beispiele:
  - Suche nach Postleitzahlen, die mit „44...“ beginnen:
    `m/(?:\d\d\d\d\d)*?(44\d\d\d)*(?:(?!44)\d\d\d\d\d)*/g`
    alternativ (weniger elegant):
    `(44\d\d\d)¦\d\d\d\d\d` oder in zwei Schritten `\d\d\d\d\d` + `^44`
  - `(?!44)\d\d\d\d\d`: 5-stellige Zahl, die nicht mit 44 beginnt
    `([^4]\d\d\d\d¦\d[^4]\d\d\d)*`: Gegenteil, d.h. Zahl, die mit 44 beginnt; `([^4][^4]\d\d\d)*` geht nicht, da `43156` durchgelassen würde

Andere Tools

grep/egrep

Suche ohne Unterscheidung zwischen Groß- und Kleinschreibung:
`egrep -i 'regexp' Datei`

awk

Awk war die erste „eierlegende Wollmilchsau“ unter den Unix-Programmen und deutlich mächtiger als z.B. *sed*. Das Prinzip ist jedoch ähnlich: Eine Datei wird zeilenweise eingelesen, bearbeitet, und entsprechend ausgegeben. Der Name besteht aus den Anfangsbuchstaben der Namen der drei Autoren, von denen jeder spezielle Kenntnisse einbringen konnte:
Alfred Aho: schrieb *egrep* und war am Regex-Teil von *lex* beteiligt
Peter Weinberger: konnte Datenbank-Kenntnisse beisteuern
Brian Kernighan: von ihm stammt ein programmierbarer Editor

Es wird oft behauptet (sowohl in der Original-Dokumentation als in manchen O’Reilly-Büchern), dass *awk* kompatibel zu *egrep* sei – das stimmt aber nicht. Noch unübersichtlicher wird die Lage dadurch, dass es unzählige verschiedene Versionen von *awk* gibt, die sich in vielen Details unterscheiden und auch unterschiedliche Bugs haben.

Tcl

Tcl benutzt Henry Spencers NFA regex package. Eine Besonderheit: Die Regex-Engine unterstützt keine Backslash-Escapes (wie z.B. „\n“); will man solche Zeichensequenzen nutzen, muss man sie zuerst in einen String einbauen, wo sie in die entsprechenden Sonderzeichen umgewandelt werden, und diesen dann an die Regex-Engine verfüttern.

Emacs

Emacs ist ein GNU-Programm; Spötter sagen, es sei kein Editor, sondern eine Lisp-Programmierumgebung mit angeflanschter Editierfunktion. Tatsächlich bietet diese Monstersoftware tausende von Funktionen bis hin zum eingebauten News-Agent und Browser. Die Regex-Engine ist dagegen recht wenig optimiert. Behandelt werden nur first character discrimination, simple repetition und length cognizance (eher halbherzig).

Python

Python ist eine objektorientierte Programmiersprache, entsprechend sind Regular Expressions auch Objekte. Man erstellt dort zuerst ein derartiges Objekt und wendet es dann auf einen String an. Interessant dabei ist, dass das Kompilieren der Regex von ihrer Anwendung getrennt ist, d.h. man erstellt einmal ein Regex-Objekt und wendet es dann auf beliebig viele Texte an.

Weitere Besonderheiten sind, dass Python nicht nur neun Rückbezüge (von `\1` bis `\9`), sondern im Prinzip beliebig viele kennt (werden mit `\v10`, `\v11` usw. bezeichnet) und dass man, ähnlich wie bei Perl, auf die in Klammern gespeicherten Werte auch von außen zugreifen kann (mit der Syntax z.B. `MyRegex.group(1)`). Ein ebenfalls interessantes Feature ist, dass man eigene Zeichenklassen definieren kann (z.B. könnte man die Währungssymbole Euro-, Pfund-, Yen- und Dollarzeichen in eine gemeinsame Klasse packen).

Literatur

- man regex(5)
- man locale(5)
- Einführung mit schöner Erklärung finiter Automaten:
  http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/
- Sammlung von Regular Expressions:
  http://regxlib.com/
- Regex-Linksammlung von Jeffrey Friedl:
  http://public.yahoo.com/~jfriedl/regex/links.html
- Tutorial über endliche Automaten:
  http://rw4.cs.uni-sb.de/~ganimal/GANIFA/
- *http://www.regenechsen.de/regex_de/regex_1_de.html*