Deutsche Ausgabe der 3. Auflage

Reguläre Ausdrücke

Jeffrey E. F. Friedl

Deutsche Übersetzung von Andreas Karrer

O'REILLY®

Inhalt

KAPITEL 1

Einführung in reguläre Ausdrücke

Hier ein Szenario: Sie bekommen den Auftrag, alle Seiten auf einem Webserver nach verdoppelten Wörtern zu durchsuchen. Solche Verdopplungen (wie »das das«) entstehen häufig bei Texten, die mehrfach überarbeitet werden. Ihre Aufgabe ist es, eine Lösung zu finden, die

- mehrere Dateien überprüft, jede Zeile mit verdoppelten Wörtern herausschreibt, die Verdopplungen (mittels ANSI-Escape-Sequenzen) hervorhebt und bei jeder Zeile den Dateinamen angibt,
- auch dann korrekt arbeitet, wenn das eine Wort am Ende einer Zeile und das verdoppelte am Anfang der nächsten Zeile steht,
- Groß- und Kleinschreibung wie bei ›`Das das ...`‹ ignoriert sowie beliebige Mengen von *Whitespace* (Leerzeichen, Tabulator, Zeilenende usw.) zwischen den Wörtern zulässt und
- auch Wörter findet, die durch HTML-Tags (neben Whitespace) getrennt sind. HTML-Tags werden in Webseiten benutzt, um beispielsweise ein Wort fett gedruckt hervorzuheben: ›`... das ist <B>sehr</B> sehr wichtig ...`‹.

Eine ganz hübsche Liste von Anforderungen! Aber das Problem muss gelöst werden, und es ist ein Problem aus der Praxis. Ich habe ein solches Programm auf eine Version des Manuskripts dieses Buches angewandt und war erstaunt über die Menge der Verdopplungen. Es gibt natürlich viele Programmiersprachen, mit denen sich diese Aufgabe lösen lässt, aber eine, die reguläre Ausdrücke kennt, macht die Lösung entschieden einfacher.

Reguläre Ausdrücke sind ein mächtiges, flexibles und effizientes Mittel, um Texte zu bearbeiten. Reguläre Ausdrücke im engeren Sinne sind eine generelle Notation zur Beschreibung von Textmustern, beinahe wie eine kleine Programmiersprache zum Prüfen und zum Manipulieren von Texten. Mit den zusätzlichen Mitteln eines bestimmten Programmierwerkzeugs können reguläre Ausdrücke benutzt werden, um alle Arten von Text zu erweitern,

zu reduzieren und auf jede erdenkliche Art misshandeln. Es kann sich um einfache Dinge handeln wie um die Suchfunktion eines Texteditors oder um komplexe wie eine ganze Textverarbeitungssprache. Dieses Buch zeigt, wie reguläre Ausdrücke produktivitätssteigernd eingesetzt werden können. Es zeigt, wie man in regulären Ausdrücken *denkt* und so ihr volles Potenzial einsetzen kann.

Ein vollständiges Programm zur Lösung der gestellten Aufgabe kann aus einigen wenigen Zeilen einer modernen Programmiersprache bestehen, die reguläre Ausdrücke kennt. Mit einer einzigen »Suchen und Ersetzen«-Anweisung können verdoppelte Wörter gefunden und hervorgehoben werden. Mit einer weiteren solchen Anweisung werden alle Zeilen weggelassen, die keine Verdopplungen enthalten, so dass nur die interessanten Zeilen übrig bleiben. Mit einer dritten Anweisung wird zu jeder Zeile der Name der Datei ausgegeben, aus der die Zeile stammt. Im nächsten Kapitel finden Sie in Perl und Java ausprogrammierte Beispiele dazu.

Die Programmiersprache (Perl, Java, VB.NET oder was auch immer) stellt das Umfeld bereit, aber die wirkliche Kraft kommt aus den regulären Ausdrücken. Indem Sie mit diesem Potenzial umgehen lernen, werden Sie lernen, reguläre Ausdrücke für Ihre eigenen Zwecke zu verwenden, um Text zu durchsuchen, hervorzuheben und uninteressanten Text wegzulassen. Sie werden Ihre regulären Ausdrücke mit den Konstrukten der Programmiersprache kombinieren und den zu bearbeitenden Text damit manipulieren – ob es sich nun um Escape-Sequenzen oder um das Weglassen oder Ändern von Text handelt.

Probleme aus der Praxis lösen

Wenn Sie mit regulären Ausdrücken umgehen können, stehen Ihnen Lösungsmöglichkeiten offen, die Sie vordem vielleicht gar nicht erahnen konnten. Ich benutze reguläre Ausdrücke jeden Tag mehrfach, um kleine und größere Probleme zu lösen (und oft sind es kleine Probleme, die ohne das Instrument der regulären Ausdrücke zu großen Problemen würden).

Bei einem Beispiel, bei dem die regulären Ausdrücke der Schlüssel zur Lösung eines großen und wichtigen Problems sind, ist der Gewinn offensichtlich. Nicht so offensichtlich ist die Art, wie man mit regulären Ausdrücken viele kleine »uninteressante« Probleme angeht. Mit »uninteressanten« Problemen meine ich solche, deren Lösung sich nicht zum Erzählen von tollen Anekdoten eignet, die aber das Weiterarbeiten unmöglich machen, bis sie gelöst sind. Für mich ist das Einsparen einer Stunde stupider, repetitiver Arbeit schon irgendwie aufregend und damit interessant.

Ein einfaches Beispiel: Bei einer Anzahl von Dateien (genauer: bei den etwa 70 Dateien, aus denen dieses Buch entstand) musste überprüft werden, ob der String ›`SetSize`‹ genauso oft (oder genauso selten) vorkommt wie ›`ResetSize`‹. Die Groß- und Kleinschreibung war dabei irrelevant (also zählt ›`setSIZE`‹ genauso wie ›`SetSize`‹). 32 000 Zeilen von Hand durchzukämmen kommt natürlich nicht in Frage. Auch die normale »Suche Wort«-Funktion eines Text-Editors ist bei der gegebenen Anzahl von Dateien viel zu umständlich.

Hier waren reguläre Ausdrücke die Helfer in der Not! Mit einer *einzigen* kurzen Befehlszeile konnte ich alle Dateien durchforsten und bestätigen, was ich überprüfen wollte. Zeitaufwand: vielleicht 15 Sekunden, um den Befehl zu tippen, und weitere 2 Sekunden für die tatsächliche Rechenzeit. Ha! (Wenn Sie an der Lösung interessiert sind, die ich benutzt habe, blättern Sie vor zur Seite 36.)

Ein anderes Beispiel: Vor einiger Zeit half ich einem Freund bei einem Problem im Zusammenhang mit E-Mail auf einer Maschine, die nur über eine langsame Modemleitung zugänglich war. Ich sollte ihm ein Inhaltsverzeichnis aller Meldungen senden – und das waren sehr viele. Auch wenn es nicht so viele Meldungen gewesen wären, wäre das Zusammenstellen der Überschriften eine mühsame und monotone Arbeit gewesen. Außerdem hätte ich dabei den Text der Meldungen gesehen, der mich nun wirklich nichts angeht.

Wiederum bewährten sich reguläre Ausdrücke als Helfer in der Not! Mit einem simplen Befehl (ich benutzte das Suchwerkzeug *egrep*; vgl. später in diesem Kapitel) konnte ich die `From:`- und `Subject:`-Zeilen aus jeder Meldung extrahieren. Um *egrep* zu instruieren, welche Arten von Zeilen ich sehen wollte (und welche nicht), benutzte ich den regulären Ausdruck `⌜^(From|Subject):␣`.

Als mein Freund diese Liste hatte, fragte er mich nach einer bestimmten (5000 Zeilen langen!) Meldung. Auch hier wäre das Benutzen eines Editors oder eines E-Mail-Programms zum Herausfiltern dieser Meldung eher umständlich gewesen. Ich benutzte ein anderes Werkzeug (dieses Mal *sed*) und wieder einen regulären Ausdruck, der genau die gesuchte Meldung aus der großen Datei herausnahm. Auf diese Art konnte ich die gewünschte Meldung schnell und einfach abschicken.

Uns beiden Zeit und Mühsal zu sparen, war nicht besonders »interessant«, aber sicher interessanter, als mit einem Text-Editor eine Stunde Zeit zu verschwenden. Hätte ich nichts über reguläre Ausdrücke gewusst, wäre mir aber gar keine Alternative eingefallen. Die Geschichte zeigt, wie man mit regulären Ausdrücken arbeiten kann, um Dinge zu tun, die man sich sonst gar nicht ausdenken könnte.

Wenn Sie gelernt haben, mit regulären Ausdrücken umzugehen, werden Sie erkennen, wie wertvoll sie sind, und Sie werden sich fragen, wie Sie vordem ohne sie ausgekommen sind.[1]

Meisterschaft im Umgang mit regulären Ausdrücken ist eine kaum zu überschätzende Fertigkeit. Dieses Buch liefert die Informationen, um diese Fertigkeit zu erlangen, und ich hoffe, dass es auch die Motivation dafür liefert, dies zu tun.

1 Wenn Sie einen digitalen Videorecorder mit Festplatte haben, kennen Sie dieses Gefühl!

Reguläre Ausdrücke als Programmiersprache

Wenn Sie noch nie mit regulären Ausdrücken gearbeitet haben, werden Sie den regulären Ausdruck ⌜^(From|Subject):⌟ aus dem letzten Beispiel nicht verstehen; aber es ist nichts Magisches daran. Zudem: Es ist nichts Magisches hinter der Magie. Ein Magier tut einfach Dinge, die für die Nicht-Eingeweihten im Publikum unmöglich oder widernatürlich *aussehen*. Wenn Sie gelernt haben, wie man eine Spielkarte in der Hand hält, so dass es aussieht, als ob die Hand leer wäre, haben Sie »magische Fähigkeiten« – es braucht nur Übung dazu. Es ist wie mit einer Fremdsprache – wenn man die Grundlagen beherrscht, klingt sie nicht mehr wie völlig unverständliche Lautäußerungen.

Die Analogie zu Dateinamen

Wenn Sie sich für den Kauf dieses Buches entschieden haben, haben Sie wahrscheinlich eine Vorstellung davon, was ein »regulärer Ausdruck« eigentlich ist. Aber auch, wenn Sie keine solche haben, kennen Sie ziemlich sicher das zugrunde liegende Konzept.

Sie wissen, dass *report.txt* einen Namen für eine ganz bestimmte Datei darstellt. Wenn Sie auch nur ein klein wenig Erfahrung mit Unix oder DOS/Windows haben, wissen Sie, dass das Muster »⌜*.txt⌟« dazu benutzt wird, um mehrere Dateien auszuwählen. Bei solchen Mustern (*Dateiglobs* oder *Wildcards* genannt) gibt es einige wenige Zeichen, die eine spezielle Bedeutung haben. Das Sternchen steht für »irgendwelche Zeichen«, und das Fragezeichen steht für »irgendein Zeichen«. Mit dem Dateiglob »⌜*.txt⌟« beginnen wir mit »irgendwas« ⌜*⌟ und enden mit dem Literal ⌜.txt⌟, also haben wir ein Muster, das aussagt: »Wähle die Dateien, deren Namen mit irgendwas beginnen und mit .txt aufhören.«

Die meisten Systeme kennen noch zusätzliche Spezialzeichen, aber im Allgemeinen ist die Ausdruckskraft solcher Dateiglobs doch recht begrenzt. Das ist kaum von Nachteil, weil auch der Umfang des Problems (eine bequeme Art, Gruppen von Dateien zu bezeichnen) doch recht limitiert ist, auf ordinäre Dateinamen eben.

Die Behandlung von beliebigen Texten ist dagegen ein ungleich komplexeres Problem. Prosa, Poesie, Programm-Listings, Memos, HTML, Artikel, Tabellen, Buchmanuskripte (wie dieses), Wörterlisten; die Aufzählung ist unvollständig. Für eine bestimmte Untergruppe, wie eben das Auswählen von Dateien, lässt sich ohne Weiteres ein spezialisiertes Werkzeug entwickeln.

Über die Jahre hat sich jedoch eine generalisierte Sprache entwickelt, die genügend mächtig und ausdrucksstark für eine große Variationsbreite von Verwendungszwecken ist. Jedes Programm implementiert und benutzt diese Muster in leicht unterschiedlicher Weise, aber alle fallen unter den Begriff »reguläre Ausdrücke«.

Die Analogie zu natürlichen Sprachen

Reguläre Ausdrücke sind aus zwei Typen von Zeichen aufgebaut. Die Spezialzeichen (wie das * aus der Dateinamen-Analogie) heißen *Metazeichen*, alle anderen Buchstaben sind *Literale*. Bei regulären Ausdrücken geht die Verwendung von Metazeichen weit über den Bereich von Dateiglobs hinaus. Muster von Dateinamen können mit wenigen Metazeichen behandelt werden, aber die »Programmiersprache« der regulären Ausdrücke erfordert viele und ausdrucksstarke Metazeichen.

Es kann helfen, reguläre Ausdrücke als eigene Sprache zu betrachten, wobei den Literalen die Rolle der Wörter zukommt und den Metazeichen die Rolle der Grammatik. Wörter werden nach den Regeln der Grammatik so kombiniert, dass der resultierende Satz oder Ausdruck einen Sinn ergibt. Im E-Mail-Beispiel habe ich ⌜^(From|Subject):⌟ benutzt, um Zeilen zu finden, die mit ›From:‹ oder ›Subject:‹ beginnen. Die Metazeichen sind hier unterstrichen, und wir kommen bald zur Erklärung der Bedeutung dieser Zeichen.

Wie beim Erlernen jeder neuen Sprache fühlt man sich auch bei regulären Ausdrücken zunächst mal etwas ratlos. Darum sehen sie auch wie Magie aus, wenn man nur wenig davon versteht, und wie völliger Buchstabensalat für jemanden, der gar nichts davon versteht. Ganz ähnlich wie 正規表現は簡単だよ![2] für einen Japanisch-Schüler schnell verständlich ist, wird Ihnen der reguläre Ausdruck

```
s!<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>!<inet>$1</inet>!
```

bald kristallklar sein.

Dieses Beispiel kommt aus der Produktion dieses Buches und stammt von meinem Lektor. Ich hatte irrtümlich das Tag <emphasis> benutzt, um Internet-Adressen zu markieren (IP-Adressen bestehen aus Ziffern und Punkten, wie etwa 209.204.146.22). Das Skript benutzt den Substitutionsoperator von Perl mit dem regulären Ausdruck

```
⌜<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>⌟
```

um diese falschen Tags durch das korrekte <inet>-Tag zu ersetzen, und lässt dabei andere <emphasis>-Tags so, wie sie sind. Später werden wir genau diese Anwendung im Detail besprechen, damit Sie die entsprechenden Methoden in Ihrer Programmiersprache anwenden können.

2 »Reguläre Ausdrücke sind einfach!« Ein humoristischer Kommentar dazu: Wie in Kapitel 3 erklärt, kommt der Terminus *regulärer Ausdruck* ursprünglich aus der formalen Algebra. Wenn mich jemand fragt, wovon denn das Buch handle, dann ergibt die Antwort »reguläre Ausdrücke« meist fragende Gesichter bei Leuten, die nicht häufig mit Computern umgehen. Das japanische Wort für regulären Ausdruck, 正規表現, bedeutet auch für den Durchschnittsjapaner ähnlich wenig wie das deutsche Gegenstück, aber von Japanern ernte ich etwas mehr als nur einen verständnislosen Blick. Im Japanischen wird nämlich »regulär« fast genau gleich ausgesprochen wie ein anderes, wesentlich häufigeres Wort, ein medizinischer Ausdruck für »Fortpflanzungsorgane«. Man kann sich ausmalen, was für Vorstellungen dabei geweckt werden …

Das Ziel dieses Buches

Die Wahrscheinlichkeit, dass *Sie* jemals `<emphasis>` durch `<inet>` ersetzen müssen, ist gering, aber es ist sehr wahrscheinlich, dass das Problem »Ersetze *dies* durch *das*« auch bei Ihnen auftritt. Das Ziel dieses Buches ist es nicht, fertige Lösungen für spezifische Probleme zu geben, sondern vielmehr zu zeigen, wie man in regulären Ausdrücken *denkt*, so dass Sie sie anwenden können, wann immer ein so geartetes Problem auftaucht.

Reguläre Ausdrücke als Denkweise

Wie wir bald sehen werden, sind reguläre Ausdrücke aus kleinen Bausteinen aufgebaut. Jeder Baustein ist für sich allein recht einfach, aber weil man diese Bausteine auf unendlich viele Arten zusammensetzen kann, braucht man einige Erfahrung, um zum gewünschten Resultat zu kommen. In diesem Kapitel geht es um einen Überblick über die grundlegenden Begriffe bei regulären Ausdrücken. Nicht mit zu viel Tiefe: Es geht darum, ein Fundament für den Rest des Buches zu legen, auf dem die wichtigen Seitenaspekte aufbauen können, bevor wir uns zu tief in Details verstricken.

Manche Beispiele mögen lächerlich erscheinen (sie *sind* oft lächerlich), aber sie veranschaulichen die Art von Problemen, die man sehr oft vor sich hat – nur erkennen Sie die Probleme vielleicht noch nicht. Machen Sie sich keine Sorgen, wenn Ihnen nicht gleich der hinterste und letzte Punkt einleuchtet. Nehmen Sie den Grundgedanken jeder Lektion mit, und lassen Sie ihn einwirken. Das ist das Ziel dieses Kapitels.

Wenn Sie Erfahrungen mit regulären Ausdrücken haben

Wenn Ihnen reguläre Ausdrücke schon einigermaßen bekannt sind, bietet Ihnen die Übersicht in diesem Kapitel nicht viel Neues. Mindestens überfliegen sollten Sie es trotzdem. Auch wenn Ihnen die Bedeutung der Metazeichen bekannt ist, kann die Herangehensweise an reguläre Ausdrücke neu sein.

Es ist ein Unterschied, ob Sie einem Instrument Töne entlocken oder ob Sie ein Instrument wirklich spielen können. Genauso gibt es einen Unterschied zwischen dem Verstehen von regulären Ausdrücken und dem *wirklichen Verstehen* von regulären Ausdrücken. Manche der Schritte in diesem Kapitel präsentieren, was Sie schon kennen, vielleicht aber in einer neuen Art, und die kann der Weg zum *wirklichen Verständnis* von regulären Ausdrücken sein.

Textdateien durchsuchen: egrep

Textstücke zu suchen, ist die einfachste Anwendung von regulären Ausdrücken – viele Texteditoren und Textverarbeitungssysteme ermöglichen es, ein Dokument nach regulären Ausdrücken zu durchsuchen. *egrep* ist noch einfacher. Man gibt *egrep* einen regulären Ausdruck sowie ein paar Namen von Dateien, die durchsucht werden sollen. *egrep* wendet den regulären Ausdruck auf jede Zeile jeder Datei an und schreibt die Zeilen heraus, für die der

reguläre Ausdruck zutrifft. *egrep* ist ein frei verfügbares Programm für eine Anzahl von Systemen, mindestens für DOS, MacOS, Windows, Unix usw. Auf der Website zu diesem Buch, *http://regex.info*, wird erklärt, wie Sie *egrep* für Ihr Betriebssystem erhalten.

Um zum E-Mail-Beispiel zurückzukommen: Der Befehl, mit dem ich das einfache E-Mail-Inhaltsverzeichnis erzeugt habe, ist in Abbildung 1-1 dargestellt. *egrep* betrachtet das erste Argument als den regulären Ausdruck und alle weiteren Argumente als Datei(en), in denen gesucht werden soll. Beachten Sie, dass die Hochkommas in Abbildung 1-1 *nicht* zum regulären Ausdruck gehören; sie werden von meiner Shell verlangt.[3] Wenn ich *egrep* benutze, kleide ich den regulären Ausdruck fast ausnahmslos in solche Hochkommas ein. Welche Zeichen Spezialzeichen sind, in welchem Zusammenhang und für wen (für den regulären Ausdruck oder für das verwendete Programm), in welcher Reihenfolge sie ausgewertet werden: Diese Detailfragen werden dann wichtig, wenn reguläre Ausdrücke in einer richtigen Programmiersprache verwendet werden; damit beginnen wir im nächsten Kapitel.

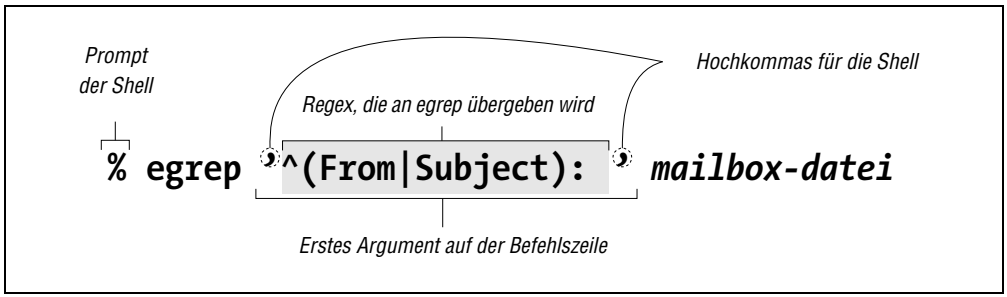


Abbildung 1-1: Aufruf von egrep auf der Kommandozeile

Wir werden die einzelnen Teile dieses regulären Ausdrucks noch genau analysieren, aber es ist wahrscheinlich unmittelbar einleuchtend, dass bestimmte Teile eine besondere Bedeutung haben. In diesem Beispiel sind die Klammern und die Zeichen ⌜^⌟ und ⌜|⌟ Metazeichen, und zusammen mit den anderen Zeichen ergeben sie das gewünschte Resultat.

Sollte Ihr regulärer Ausdruck keines der Dutzend Metazeichen benutzen, die *egrep* versteht, dann wird aus dem Befehl eine simple Suche nach Zeichenketten. Wenn zum Beispiel in einer Datei nach ⌜ding⌟ gesucht wird, gibt *egrep* alle Zeilen aus, die die vier Zeichen d·i·n·g hintereinander enthalten. Das geschieht auch dann, wenn diese Zeichen in einem Wort wie etwa bedingen enthalten sind.

3 Die Shell ist der Teil des Systems, der eingegebene Befehle liest und dadurch erforderliche Programme ausführt. Bei der von mir benutzten Shell dienen Hochkommas dazu, Argumente für ein Programm einzuschließen. Der Shell wird damit gesagt, den Inhalt der Argumente nicht anzurühren. Ohne Hochkommas würde die Shell beispielsweise ein ∗-Zeichen, das als Teil des regulären Ausdrucks gemeint ist, *selbst* interpretieren, nämlich als Dateiglob, als Teil eines Dateinamens. Dagegen soll der Inhalt der Hochkommas unverändert an *egrep* übergeben werden, das ihn als regulären Ausdruck interpretieren wird. Windows- und COMMAND.COM-Benutzer verwenden wahrscheinlich mit Vorteil doppelte Anführungszeichen ("Gänsefüßchen") statt Hochkommas.

Auch wenn die Zeile nicht das *Wort* `ding` enthält, genügt die Sequenz `d·i·n·g`, damit der reguläre Ausdruck zutrifft. Das ist alles, wonach gesucht wird, und wenn es gefunden wird, gibt *egrep* die Zeile aus. Der springende Punkt ist der, dass die Suche nach regulären Ausdrücken nicht auf »Wörtern« basiert – *egrep* kennt wohl Bytes und Zeilen innerhalb von Dateien, weiß aber nicht, was im Deutschen (oder in irgendeiner anderen Sprache) ein Wort, einen Satz, einen Abschnitt oder irgendein höheres Konzept ausmacht.

Metazeichen bei egrep

Wir beginnen mit einigen Metazeichen von *egrep*, um die Möglichkeiten dieses Werkzeugs auszuloten. Ich werde diese nur kurz mit ein paar Beispielen behandeln; die erschöpfende Beschreibung folgt in späteren Kapiteln.

Typografische Konventionen

Vor Beginn stellen Sie bitte sicher, dass Sie die hier verwendeten typografischen Konventionen kennen (Seite XXI).

Zeilenanfang und Zeilenende

Wahrscheinlich am einfachsten zu verstehen sind die Metazeichen ⌜`^`⌟ (*Zirkumflex*) und ⌜`$`⌟ (*Dollar*), die für den Anfang bzw. das Ende der zu prüfenden Zeile stehen. Wie wir gesehen haben, findet der reguläre Ausdruck ⌜`ding`⌟ die Zeichen `d·i·n·g` irgendwo auf der Zeile, aber ⌜`^ding`⌟ findet die Zeichen `d·i·n·g` nur dann, wenn sie am Anfang der Zeile stehen. Analog passt der reguläre Ausdruck ⌜`ding$`⌟ nur am Ende der Zeile, wenn zum Beispiel eine Zeile mit `Unding` endet. Weil Zirkumflex und Dollar den regulären Ausdruck am Anfang oder am Ende der Zeile festmachen, nennt man sie *Zeilenanker* oder einfach *Anker*.

Machen Sie es sich zur Gewohnheit, reguläre Ausdrücke in einem wörtlichen Sinn zu interpretieren. Lesen Sie nicht:

> ⌜`^ding`⌟ passt auf eine Zeile mit `ding` am Anfang

sondern eher:

> ⌜`^ding`⌟ passt auf einen Zeilenanfang, unmittelbar gefolgt von `d`, unmittelbar gefolgt von `i`, unmittelbar gefolgt von `n`, unmittelbar gefolgt von `g`.

Das kommt zwar auf dasselbe heraus, aber die wörtliche Lesart erlaubt es, einen neuen Ausdruck besser zu verstehen. Wie würde *egrep* ⌜`^ding$`⌟, ⌜`^$`⌟ oder ganz einfach ⌜`^`⌟ allein interpretieren? ❖ Für die Auflösung bitte umblättern.

Der Zirkumflex und das Dollarzeichen sind insofern speziell, als sie eine *Position* in der Zeile beschreiben und nicht ein Zeichen selbst. Es gibt natürlich einige Möglichkeiten, um auf wirkliche Zeichen in der Zeile zu prüfen. Neben Literalen, die für das Zeichen selbst stehen, gibt es einige andere Metazeichen, die in den nächsten Abschnitten behandelt werden.

Zeichenklassen

Eines von mehreren Zeichen erkennen

Nehmen wir an, Sie wollen nach dem String »Birma« suchen, aber auch »Burma«[4] zulassen. Die Konstruktion ⌜[···]⌟, meist *Zeichenklasse* genannt, ermöglicht es, eine Liste von Zeichen anzugeben, nach denen gesucht werden soll. Ein ⌜i⌟ passt nur auf ein `i`, ein ⌜u⌟ nur auf ein `u`, ein ⌜iu⌟ aber auf beide. Der reguläre Ausdruck ⌜B[iu]rma⌟ bedeutet also wörtlich: Suche nach `B`, gefolgt von einem `i` oder einem `u`, gefolgt von `r`, einem `m` und danach einem `a`. Ich bin ziemlich schlecht in der Rechtschreibung, daher benutze ich oft reguläre Ausdrücke, um Wörter zu finden, die ich immer wieder falsch schreibe. Einer meiner häufigen Fehler ist ⌜sep[ea]r[ea]te⌟, weil ich mich nie daran erinnern kann, ob das Wort nun »seperate«, »separate«, »separete« oder wie auch immer geschrieben wird.

Beachten Sie, dass zwischen den Zeichen außerhalb der Zeichenklasse (hier ⌜r⌟, ⌜m⌟ und ⌜a⌟ im vollständigen Ausdruck ⌜B[iu]rma⌟) ein Nacheinander oder ein »und dann« impliziert wird – zunächst ein ⌜r⌟ **und dann** ein ⌜m⌟ **und dann** ein ⌜a⌟. Ganz anders innerhalb der Zeichenklasse: Dort spielt die Reihenfolge überhaupt keine Rolle, es geht dort um ein *oder*.

Oder vielleicht wollen Sie die Groß- oder Kleinschreibung eines Wortes zulassen: ⌜[Rr]ot⌟. Das passt noch immer auch auf Zeilen, bei denen der String `rot` oder `Rot` als Teil eines Wortes vorkommt, etwa in `Protest`. Ich reite etwas auf diesem Punkt herum, weil meine Erfahrung zeigt, dass dies bei Neulingen ein Stolperstein ist. Wenn wir ein paar weitere Metazeichen kennen, werde ich erneut darauf zurückkommen.

Die Liste in den eckigen Klammern kann beliebig viele Zeichen enthalten. Zum Beispiel erkennt ⌜[123456]⌟ irgendeine der angegebenen Ziffern. Diese Zeichenklasse kann als Teil von ⌜<H[123456]>⌟ (erkennt `<H1>`, `<H2>`, `<H3>` usw.) nützlich sein, wenn es darum geht, HTML-Tags zu prüfen.

Innerhalb einer Zeichenklasse gibt das *Zeichenklassen-Metazeichen* ›-‹ (*Bindestrich*) einen Bereich von Zeichen an: ⌜<H[1-6]>⌟ bedeutet exakt dasselbe wie das vorherige Beispiel. ⌜[0-9]⌟ und ⌜[a-z]⌟ sind übliche Abkürzungen für Ziffern bzw. Kleinbuchstaben. Mehrfache Bereiche sind zugelassen. ⌜[0123456789abcdefABCDEF]⌟ kann also kürzer so geschrieben werden: ⌜[0-9a-fA-F]⌟ (oder auch ⌜[A-Fa-f0-9]⌟, die Reihenfolge der Bereiche spielt keine Rolle). Diese drei Bereiche werden oft gebraucht, wenn man mit hexadezimalen Zahlen arbeitet. Bereiche können auch mit Literalen kombiniert werden: ⌜[0-9A-Z_!.?]⌟ passt auf Ziffern, Großbuchstaben, den Unterstrich, Ausrufezeichen, Punkt oder Fragezeichen.

Beachten Sie, dass der Bindestrich nur innerhalb einer Zeichenklasse ein Metazeichen ist – außerhalb dieser ist er ein normaler Bindestrich. Sogar innerhalb einer Zeichenklasse ist er nicht immer ein Metazeichen. Falls der Bindestrich das erste Zeichen innerhalb der Klasse ist, kann er ja schlecht einen Bereich angeben; er wird dann als Literal behandelt. Ganz ähnlich sind das Fragezeichen und der Punkt in einem regulären Ausdruck normalerweise

4 Das ist die in der Schweiz und im Englischen übliche Schreibweise.

Wie liest man ⌜^ding$⌟, ⌜^$⌟ und ⌜^⌟?

❖ *Auflösung zum Problem von Seite 8.*

⌜^ding$⌟

Wörtlich: Passt, wenn die Zeile einen Anfang hat (was natürlich alle Zeilen haben), gefolgt von d·i·n·g, und wenn dann unmittelbar das Zeilenende folgt.

Effekt: Zeilen werden erkannt, wenn sie nur ding und nichts anderes enthalten, also keine zusätzlichen Wörter, Wortzwischenräume, Interpunktion usw.

⌜^$⌟

Wörtlich: Passt, wenn die Zeile einen Anfang hat, auf den unmittelbar das Zeilenende folgt.

Effekt: Leerzeilen werden erkannt (nur *wirkliche* Leerzeilen, auch Leerzeichen auf der Zeile sind nicht erlaubt).

⌜^⌟

Wörtlich: Passt, wenn die Zeile einen Anfang hat.

Kein Effekt! Da jede Zeile einen Zeilenbeginn hat, wird dieser reguläre Ausdruck immer passen – auch bei Leerzeilen!

Metazeichen, aber nicht innerhalb einer Zeichenklasse! Um das ganz klarzumachen: Nur die zwei Bindestriche in ⌜[0-9A-Z_!.?]⌟ sind Metazeichen.

> Zeichenklassen haben ihre eigene Miniatur-Sprache. Die Regeln, welche Zeichen als Metazeichen gelten (und was deren Funktion ist), sind inner- und außerhalb von Zeichenklassen verschieden.

Weitere Beispiele dazu folgen in Kürze.

Negierte Zeichenklassen

Wenn man ⌜[^···]⌟ statt ⌜[···]⌟ benutzt, passt die Klasse auf alle Zeichen, die *nicht* unter den angegebenen Zeichen sind. Zum Beispiel erkennt ⌜[^1-6]⌟ alle Zeichen *außer* 1 bis 6. Der Zirkumflex als erstes Zeichen in der Klasse »negiert« also die Liste – statt alle erwünschten Zeichen aufzulisten, gibt man die unerwünschten an.

Sicher ist Ihnen aufgefallen, dass das Zeichen ^ derselbe Zirkumflex ist wie der, der gerade vorhin auf Seite 8 noch »Zeilenanfang« bedeutete. Das Zeichen ist dasselbe, aber die Bedeutung ist völlig verschieden. Das ist ganz ähnlich wie etwa beim deutschen Wort »Hahn«, das je nach Zusammenhang zwei völlig unterschiedliche Dinge bezeichnen kann (den Mann

der Henne oder ein Ventil bei Rohrleitungen). Wir haben schon ein anderes Zeichen gesehen (den Bindestrich), das innerhalb einer Zeichenklasse einen Bereich angibt – und auch da nur, wenn es nicht das erste Zeichen der Klasse ist. ^ fungiert als Anker außerhalb von Zeichenklassen und als Klassen-Metazeichen innerhalb einer Klasse, aber auch nur, wenn es unmittelbar auf die öffnende eckige Klammer folgt (sonst ist es innerhalb von Klassen kein Spezialzeichen). Keine Angst – damit haben wir schon den verzwicktesten Sonderfall behandelt, die anderen Metazeichen sind einfacher.

Hier folgt ein Beispiel mit einer englischen Wörterliste. Wie im Deutschen folgt auch im Englischen auf ein q fast immer ein u. Wenn wir also nach Wörtern suchen, die ein q, danach aber etwas anderes als ein u enthalten, und das als regulären Ausdruck formulieren, dann erhalten wir ⌜q[^u]⌟. Ich habe meine Wörterliste abgesucht und fast keine Ausnahmen gefunden! Und von den wenigen, die ich fand, kannte ich einige nicht:

```
% egrep 'q[^u]' word.list
Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum
%
```

Zwei bekannte Wörter allerdings fehlten: »Qantas«, die australische Fluggesellschaft, und »Iraq« (engl. Schreibweise für Irak). Beide sind zwar in meiner Wörterliste *word.list* vorhanden, aber sie wurden von meinem *egrep*-Befehl nicht ausgegeben. Warum? ❖ Auf der nächsten Doppelseite finden Sie die Antwort.

Eine negierte Zeichenklasse bedeutet also »Gefordert ist ein Zeichen, das nicht in der Liste vorkommt« und nicht etwa »Ein Zeichen aus der Liste darf nicht vorkommen«. Das sieht auf den ersten Blick gleich aus, aber das Beispiel mit Iraq zeigt den subtilen Unterschied. Negierte Klassen sollte man am besten als Abkürzungen für normale Klassen ansehen, die alle Zeichen enthalten *außer* denen, die in den eckigen Klammern angegeben sind.

Auf irgendein Zeichen prüfen: der Punkt

Das Metazeichen ⌜.⌟, *Punkt*, ist eine Abkürzung für die Zeichenklasse, die alle möglichen Zeichen enthält. Es ist häufig praktisch, wenn man ein »irgendwas hier« als Platzhalter in einem regulären Ausdruck braucht. Wenn zum Beispiel nach einem Datum gesucht werden soll, das wie 03/19/76, 03-19-76 oder gar wie 03.19.76[5] aussehen kann, kann man einen expliziten regulären Ausdruck formulieren und als Trennzeichen explizit nur /, -, und . zwischen den Zahlen zulassen: ⌜03[-./]19[-./]76⌟. Man kann aber kürzer auch einfach ⌜03.19.76⌟ verwenden.

5 Man sollte Daten *nie* so angeben, weil dieses Format für die Tage vor dem 13. des Monats in Europa anders interpretiert wird als in den USA. Das Format 1976-05-01 nach ISO 8601 ist eindeutig. (Anm. d. Ü.)

Warum findet ⌜q[^u]⌟ ›Qantas‹ und ›Iraq‹ nicht?

❖ *Antwort zum Problem von Seite 11.*

`Qantas` wird nicht gefunden, weil der reguläre Ausdruck nach einem Kleinbuchstaben `q` verlangt, das `Q` in Qantas ist aber ein Großbuchstabe. Hätten wir ⌜`Q[^u]`⌟ verwendet, wäre `Qantas` wohl gefunden worden, dafür alle anderen Wörter nicht, weil sie kein großes `Q` enthalten. Der reguläre Ausdruck ⌜`[Qq][^u]`⌟ dagegen hätte alle gefunden.

Das Problem mit `Iraq` ist schon fast eine Fangfrage. Der reguläre Ausdruck verlangt nach einem `q`, gefolgt von *einem Zeichen*, das kein `u` sein darf. Nun entfernt aber *egrep* alle Zeilenende-Zeichen, bevor es reguläre Ausdrücke prüft (ein Detail, das ich verschwiegen habe – sorry!), und so gibt es nach dem `q` *überhaupt kein* Zeichen, das irgendwie in den regulären Ausdruck passen könnte. Sicher kommt nach dem `q` kein `u`, aber es kommt auch kein Nicht-`u`!

Grämen Sie sich nicht wegen der Fangfrage.[a]

Wenn nun *egrep* das Zeilenende-Zeichen nicht abschneiden würde (andere Programme tun das nicht) oder wenn auf das Wort `Iraq` ein anderes Wort oder auch nur ein Leerzeichen folgen würde, dann wäre das Wort gefunden worden. Manchmal ist es wichtig, die letzten Feinheiten eines Programms zu kennen, aber bei diesem Beispiel geht es mir nur darum zu zeigen, dass *eine Zeichenklasse, auch wenn sie negiert ist, ein Zeichen braucht, damit sie passt*.

---

a In der vierten Klasse sollte ich einmal in einem mündlichen Buchstabierwettbewerb das Wort »miss« (verfehlen) buchstabieren. Ich antwortete `m·i·s·s`. Aber die Lehrerin, Miss Smith, sagte, dass `M·i·s·s` mit einem großen `M` gemeint wäre und dass ich nach einem Beispielsatz hätte fragen müssen. Ein traumatischer Moment im Leben eines Knaben. Nach diesem Erlebnis mochte ich Miss Smith nicht mehr und stehe seiddem mit der Ortografie auf Krigsfus.

Mehrere Dinge können hier unklar sein. Zunächst sind die Punkte in ⌜`03[-./]19[-./]76`⌟ *keine* Metazeichen, weil sie in einer Zeichenklasse vorkommen (nochmals: die Liste der Metazeichen und deren Bedeutung ist inner- und außerhalb von Zeichenklassen verschieden). Auch die Bindestriche sind hier keine Metazeichen, obwohl sie doch innerhalb von Zeichenklassen meist einen Bereich angeben. Wie erwähnt, verliert ein Bindestrich als erstes Zeichen einer Zeichenklasse seine Spezialfunktion. Falsch wäre in dieser Situation ⌜`[.-/]`⌟, hier wäre der Bindestrich ein Zeichenklassen-Metazeichen.

Dagegen sind die Punkte in ⌜`03.19.76`⌟ *tatsächlich* Metazeichen, die für irgendein Zeichen stehen, also auch für den Schrägstrich, den Bindestrich oder den Punkt, also für die Zeichen,

die wir erwarten. Es ist aber wichtig, sich klarzumachen, dass der Punkt für *jedes* beliebige Zeichen stehen kann; der reguläre Ausdruck kann also auch bei ›Losnummer: 19 203319 7639‹ passen.

⌜03[-./]19[-./]76⌟ ist präziser, aber auch mühsamer zu schreiben und zu lesen. ⌜03.19.76⌟ ist einfacher lesbar, aber wenig spezifisch. Was soll man nun benutzen? Das hängt in hohem Maße von den zu untersuchenden Daten ab oder wenigstens von Ihren Annahmen, wie diese aussehen. Sie müssen die Balance zwischen Ihrem (Vor-)Wissen über den zu untersuchenden Text und der Notwendigkeit finden, den regulären Ausdruck sehr exakt zu formulieren. Wenn Sie wissen, dass in Ihren Daten außer Datumsangaben keine anderen Zahlen (wie Telefonnummern) vorkommen, können Sie getrost ⌜03.19.76⌟ verwenden.

Alternation

Auf einen von mehreren Unterausdrücken prüfen

Ein sehr praktisches Metazeichen ist ⌜|⌟, das »oder« bedeutet. Es erlaubt, mehrere reguläre Ausdrücke zu einem einzigen Ausdruck zu kombinieren, der dann zutrifft, wenn einer der Unterausdrücke passt. Wenn zum Beispiel ⌜Bob⌟ und ⌜Robert⌟ zwei separate reguläre Ausdrücke sind, dann ist ⌜Bob|Robert⌟ ein regulärer Ausdruck, der beide enthält und dann passt, wenn zu einem der Ausdrücke ein Treffer gefunden wird. Wenn reguläre Ausdrücke auf diese Art kombiniert werden, heißen die Unterausdrücke *Alternativen*.

Wenn wir zu unserem ⌜B[iu]rma⌟-Beispiel zurückgehen, können wir dies auch ⌜Birma|Burma⌟ oder sogar ⌜B(i|u)rma⌟ schreiben. Im letzteren Fall dienen die Klammern dazu, die Alternativen örtlich zu beschränken. (Damit haben wir mit den runden Klammern zwei weitere Metazeichen kennengelernt.) Beachten Sie außerdem, dass etwas wie ⌜B[i|u]rma⌟ ganz falsch wäre – innerhalb einer Zeichenklasse ist der senkrechte Strich ein ganz normales Zeichen wie ⌜i⌟ und ⌜u⌟.

Der reguläre Ausdruck ohne Klammern, ⌜Bi|urma⌟, würde nur ⌜Bi⌟ oder ⌜urma⌟ bedeuten, was ebenfalls nicht das Gewünschte wäre. Die Alternativen reichen weit, aber nicht weiter als die einfassenden Klammern. Ein anderes Beispiel ist ⌜(First|1st)␣[Ss]treet⌟.[6] Da sowohl ⌜First⌟ als auch ⌜1st⌟ mit einem ⌜st⌟ enden, kann man dies zu ⌜(Fir|1)st␣[Ss]treet⌟ verkürzen, aber das lässt sich nicht so gut lesen. Machen Sie sich die Mühe zu verstehen, dass beide Möglichkeiten dasselbe bedeuten.

Hier ein Beispiel mit verschiedenen Schreibweisen meines Vornamens. Vergleichen Sie die drei folgenden regulären Ausdrücke, die alle das Gleiche bewirken:

```
⌜Jeffrey|Jeffery⌟
⌜Jeff(rey|ery)⌟
⌜Jeff(re|er)y⌟
```

6 Sie erinnern sich an die typografischen Konventionen von Seite XXII: »␣« ist ein sichtbar gemachtes Leerzeichen.

Wenn außerdem auch die britischen Varianten berücksichtigt werden sollen, sehen die regulären Ausdrücke wie folgt aus:

```
⌜(Geoff|Jeff)(rey|ery)⌟
⌜(Geo|Je)ff(rey|ery)⌟
⌜(Geo|Je)ff(re|er)y⌟
```

Beachten Sie außerdem, dass diese letzten drei Ausdrücke dasselbe bewirken wie der längere (aber einfacher verständliche) ⌜`Jeffrey|Geoffery|Jeffery|Geoffrey`⌟. Das sind alles verschiedene Arten, dasselbe auszusagen.

Obwohl die Beispiele ⌜`B[iu]rma`⌟ und ⌜`B(i|u)rma`⌟ das nahelegen könnten, sind Zeichenklassen und Alternationen ganz und gar nicht dasselbe. Eine Zeichenklasse repräsentiert *genau ein Zeichen* im zu untersuchenden Text. Bei einer Alternation kann dagegen jede Alternative ein ausgewachsener regulärer Ausdruck sein. Zeichenklassen sind so etwas wie Mini-Sprachen innerhalb von regulären Ausdrücken (mit ihrer eigenen Auffassung darüber, was Metazeichen sind), während die Alternation Teil der »eigentlichen« Sprache der regulären Ausdrücke ist. Beide haben ihren Platz, und beide sind sehr nützlich.

Etwas Vorsicht ist bei der Verwendung der Zeilenanker `^` und `$` geboten. Man vergleiche ⌜`^From|Subject|Date:·`⌟ und ⌜`^(From|Subject|Date):·`⌟. Beide erinnern an unser E-Mail-Beispiel, aber sie verhalten sich ganz unterschiedlich. Der erste Ausdruck besteht aus drei Alternativen und wird dann passen, wenn »⌜`^From`⌟ oder ⌜`Subject`⌟ oder ⌜`Date:·`⌟« gefunden wird – nicht besonders nützlich. Der Zirkumflex am Anfang und das ⌜`:·`⌟ am Ende soll bei allen drei Alternativen vorkommen. Mit den Klammern können wir die Alternation »beschränken«:

```
⌜^(From|Subject|Date):·⌟
```

Damit wird gefunden:

1) ein Zeilenanfang, gefolgt von `F·r·o·m`, gefolgt von ›`:·`‹

oder 2) ein Zeilenanfang, gefolgt von `S·u·b·j·e·c·t`, gefolgt von ›`:·`‹

oder 3) ein Zeilenanfang, gefolgt von `D·a·t·e`, gefolgt von ›`:·`‹

Etwas weniger umständlich ausgedrückt: Es werden die Zeilen gefunden, die mit ›`From:·`‹, ›`Subject:·`‹ oder ›`Date:·`‹ beginnen; und das ist es, was wir wollen, wenn wir relevante Informationen aus einer E-Mail-Datei extrahieren möchten. Zum Beispiel:

```
% egrep  '^(From|Subject|Date): '  mailbox
From: elvis@tabloid.org (The King)
Subject: be seein' ya around
Date: Mon, 11 Jun 2007 11:04:13
From: The Prez <president@whitehouse.gov>
Date: Wed, 13 Jun 2007  8:36:24
Subject: now, about your vote...
...
```

Groß- und Kleinschreibung ignorieren

Diese E-Mail-Kopfzeilen sind ein guter Anlass, um das Thema der Groß- und Kleinschreibung einzuführen. Normalerweise werden die Schlüsselwörter in den Header-Zeilen mit einem Großbuchstaben am Anfang geschrieben, also beispielsweise »Subject« und »From«, aber der E-Mail-Standard erlaubt auch die durchgehende Großschreibung wie »DATE« und jede beliebige Mischung von Groß- und Kleinbuchstaben. Mit den regulären Ausdrücken aus dem vorherigen Abschnitt erwischen wir diese leider nicht.

Wir könnten statt ⌜`From`⌟ so etwas wie ⌜`[Ff][Rr][Oo][Mm]`⌟ schreiben und würden so jede mögliche Schreibweise von »From« abdecken, aber das wäre schon sehr mühsam. Deshalb kennt *egrep* eine spezielle Option, mit der man das Programm instruiert, beim Vergleich von Zeichen die Groß- und Kleinschreibung nicht zu beachten. Diese Option gehört nicht direkt zu den regulären Ausdrücken, aber fast jedes Programm, das reguläre Ausdrücke verwenden kann, hat eine entsprechende Funktion. Bei *egrep* heißt die Option »`-i`«:

```
% egrep -i  '^(From|Subject|Date): '  mailbox
```

Damit erhalten wir die gleichen Zeilen wie vorhin, aber zusätzlich auch noch Zeilen wie:

```
SUBJECT: MAKE MONEY FAST
```

Ich verwende die Option `-i` sehr häufig (vielleicht hängt das ja mit der Fußnote im Kasten auf Seite 12 zusammen!) und empfehle deshalb, sich den Gebrauch zu merken. Wir werden in späteren Kapiteln noch auf andere Optionen dieser Art treffen.

Wortgrenzen

Häufig entsteht das Problem, dass ein regulärer Ausdruck ein gewünschtes Wort als Teil eines anderen, nicht gewünschten Wortes findet. Wir haben das bei `ding` innerhalb von `bedingen` und bei `rot` innerhalb von `Protest` gesehen. Obwohl ich gesagt habe, dass *egrep* im Allgemeinen nichts von übergeordneten Konzepten wie Wörtern versteht, gibt es doch Versionen von *egrep*, die ein rudimentäres Verständnis dafür haben: Sie können Wortgrenzen erkennen (wo ein Wort beginnt und wo es aufhört).

Dafür benutzen Sie die etwas merkwürdig aussehenden *Metasequenzen* ⌜`\<`⌟ und ⌜`\>`⌟, falls Ihr *egrep* diese unterstützt (nicht alle Versionen tun das). Das sind Anker wie ⌜`^`⌟ und ⌜`$`⌟, nur dass sie sich auf Wortanfänge und Wortenden beziehen statt auf Zeilenanfänge und -enden. Genau wie die Anker Zirkumflex und Dollar »verbrauchen« sie keine Zeichen. Der Ausdruck ⌜`\<ding\>`⌟ besagt wörtlich: »Finde eine Position an einem Wortanfang, unmittelbar gefolgt von den Zeichen `d`·`i`·`n`·`g`, gefolgt von einer Position am Ende eines Wortes.« Verständlicher ausgedrückt: »Finde das Wort `ding`.« Mit ⌜`\<ding`⌟ bzw. ⌜`ding\>`⌟ könnte man nach Wörtern suchen, die mit `ding` beginnen oder enden.

Beachten Sie, dass ⌜<⌟ und ⌜>⌟ für sich keine Metazeichen sind. Erst wenn davor ein Backslash steht, wird die *Sequenz* zu etwas Besonderem. Darum nennt man sie »Metasequenzen«. Die besondere Interpretation dieser Sequenzen oder Zeichen ist wichtig, nicht die Tatsache, dass manche aus einem Zeichen und andere aus mehreren bestehen; daher verwende ich die zwei Meta-Begriffe meist als Synonyme.

Wie erwähnt, beherrschen nicht alle *egrep*-Programme die Wortgrenzen-Metazeichen, und auch diejenigen, die es tun, haben nicht plötzlich ein magisches Verständnis für die deutsche (oder englische) Sprache. Ein »Wortanfang« im Sinne von *egrep* ist einfach eine Position, an der eine Serie von alphanumerischen Zeichen beginnt. Ein »Wortende« wird an der Position gefunden, wo eine solche Serie endet. Abbildung 1-2 markiert solche Positionen an einer Beispielzeile.

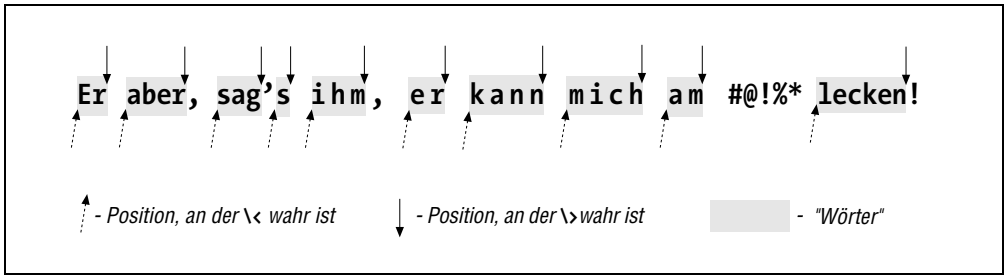


Abbildung 1-2: Anfang und Ende von »Wort«-Positionen im Götz-Zitat

Die Wortanfänge (im Sinne von *egrep*) sind mit nach oben gerichteten Pfeilen markiert; die Wortenden mit Pfeilen nach unten. Wie man sieht, sollte »Wortanfang und -ende« präziser als »Anfang und Ende einer alphanumerischen Zeichensequenz« bezeichnet werden, aber das ist dann vielleicht doch etwas zu viel des Guten.

Kurze Rekapitulation

Tabelle 1-1: Zusammenfassung der bisher aufgetretenen Metazeichen

Metazeichen	Name	Bedeutung
`.`	*Punkt*	irgendein Zeichen
`[…]`	*Zeichenklasse*	irgendein Zeichen aus der Liste
`[^…]`	*negierte Zeichenklasse*	irgendein Zeichen nicht aus der Liste
`^`	*Zirkumflex, »Hütchen«*	die Position am Zeilenanfang
`$`	*Dollar*	die Position am Zeilenende
`\<`	*Backslash Kleiner-als*	die Position am Wortanfang (nicht von allen *egrep*-Versionen unterstützt)
`\>`	*Backslash Größer-als*	die Position am Wortende (nicht von allen *egrep*-Versionen unterstützt)
`|`	*»oder«, »Pipe«*	trennt Alternativen in Alternationen
`(…)`	*(runde) Klammern*	beschränken die Reichweite von ⌜|⌟ (weitere Bedeutungen später)

Zusätzlich zur Tabelle sollten Sie sich Folgendes merken:

- Die Regeln, was ein Metazeichen ausmacht (und was es exakt bedeutet), sind inner- und außerhalb von Zeichenklassen unterschiedlich. Zum Beispiel ist ein Punkt außerhalb einer Zeichenklasse ein Metazeichen, aber nicht innerhalb. Umgekehrt ist der Bindestrich innerhalb einer Klasse ein Metazeichen, aber nicht außerhalb. Der Zirkumflex hat eine spezielle Bedeutung außerhalb und eine andere, ebenfalls spezielle Bedeutung als erstes Zeichen innerhalb einer Zeichenklasse.
- Alternationen und Zeichenklassen sind nicht dasselbe. Die Klasse ⌜`[abc]`⌟ und die Alternation ⌜`(a|b|c)`⌟ bedeuten im Endeffekt dasselbe, aber diese Ähnlichkeit gilt nicht generell. Eine Zeichenklasse erkennt exakt ein Zeichen, egal wie lang oder wie kurz die Liste der möglichen Zeichen ist. Eine Alternation dagegen kann beliebig lange Alternativen besitzen, und die Alternativen brauchen keinerlei Verwandtschaft untereinander zu haben: ⌜`\<(1·000·000|eine·Million|tausend·Tausender)\>`⌟. Alternationen können nicht wie Zeichenklassen negiert werden.
- Eine negierte Zeichenklasse ist einfach eine Notation für eine normale Klasse, die alle außer den aufgelisteten Zeichen enthält. ⌜`[^x]`⌟ bedeutet nicht »passt, falls kein `x` vorkommt«, sondern eher »passt auf ein Zeichen, das kein `x` ist«. Der Unterschied scheint minimal zu sein, ist aber wichtig. Die erste – falsche – Interpretation würde eine Leerzeile erkennen, die in Wirklichkeit nicht auf den regulären Ausdruck ⌜`[^x]`⌟ passt.
- Mit der nützlichen Option `-i` kann man *egrep* anweisen, die Groß- und Kleinschreibung zu ignorieren (☞ 15).[7]

Damit kann man schon ganz hübsche Sachen machen, aber so richtig geht es erst mit den *optionalen* und *zählenden* Elementen zur Sache.

Optionale Elemente

Wenn wir fast identische Wörter wie `Marlen` und `Maren` vergleichen, sehen wir, dass sie sich nur in dem einmal fehlenden `l` unterscheiden. Der reguläre Ausdruck ⌜`Mar`**`l?`**`en`⌟ passt auf beide. Das Metazeichen ⌜`?`⌟ (*Fragezeichen*) bedeutet *optional*. Das Fragezeichen wird hinter das zu untersuchende Zeichen gesetzt. Wenn das Zeichen im zu prüfenden Text vorkommt, ist das okay, wenn es fehlt, passt der reguläre Ausdruck auch.

Das Fragezeichen bezieht sich – im Unterschied zu den bisher behandelten Metazeichen – nur auf das unmittelbar vorausgehende Element. ⌜`Marl?en`⌟ wird daher so interpretiert: »Erst ⌜`M`⌟, dann ein ⌜`a`⌟, dann ein ⌜`r`⌟, dann ein ⌜`l?`⌟, dann ein ⌜`e`⌟ und schließlich ein ⌜`n`⌟.«

Das Fragment ⌜`l?`⌟ wird immer passen: Manchmal wird es ein tatsächliches `l` im Text erkennen und manchmal nicht. Der Punkt bei optionalen Elementen ist der, dass der reguläre Ausdruck »funktioniert«, ob das Element nun passt oder nicht. Das bedeutet überhaupt nicht, dass ein regulärer Ausdruck mit einem `?` immer passen würde. So würde beim Text

7 Ich erinnere Sie an die Konventionen von Seite XXII: »☞ 15« ist einfach ein Seitenverweis, der etwas Platz spart.

›Marlon‹ sowohl ⌜Mar⌟ als auch ⌜l?⌟ gefunden, aber das o danach verhindert, dass der ganze reguläre Ausdruck den Text erkennt.

Ein anderes Beispiel: Wie findet man die verschiedenen Schreibweisen für den 4. Juli, den amerikanischen Nationalfeiertag? Geschrieben wird das »July 4th«, »July fourth« oder auch nur »Jul 4«. Wir könnten natürlich alle Kombinationen explizit aufzählen und erhielten ⌜(July|Jul)•(fourth|4th|4)⌟, aber lassen Sie mich andere Möglichkeiten darlegen, die dasselbe aussagen.

Zunächst können wir ⌜(July|Jul)⌟ zu ⌜(Jul**y?**)⌟ vereinfachen. Ist Ihnen klar, warum das aufs Gleiche hinausläuft? Das Entfernen des ⌜|⌟ bedeutet auch, dass die Klammern nicht mehr gebraucht werden. Sie zu belassen würde zwar nichts schaden, aber ⌜July?⌟ ist ohne Klammern schon deutlich lesbarer. Also haben wir jetzt ⌜July?•(fourth|4th|4)⌟.

Auch im zweiten Teil lässt sich ⌜4th|4⌟ zu ⌜4**(th)**?⌟ vereinfachen. Wie man sieht, kann sich das ⌜?⌟ auch auf einen geklammerten Ausdruck beziehen. Innerhalb der Klammern kann es so kompliziert zugehen wie nur denkbar, aber von außen gesehen ist die ganze Klammer eine Einheit. Dies ist einer der häufigeren Verwendungszwecke für runde Klammern: Sie bilden Gruppen für das ⌜?⌟-Metazeichen (und für ähnliche Zeichen, die ich nächstens vorstelle).

Unser Ausdruck sieht nun so aus: ⌜July?•**(**fourth|4(th)?**)**⌟. Obwohl hier ziemlich viele Metazeichen und sogar verschachtelte Klammern auftreten, ist das Ganze noch gut lesbar und nicht schwer zu dechiffrieren. Diese Besprechung von zwei eigentlich simplen Beispielen ist recht lang geworden, aber nebenbei haben wir angrenzende Gebiete gestreift, die viel zum Verständnis von regulären Ausdrücken beitragen – vielleicht auch nur unbewusst. Wir haben dabei Erfahrungen gesammelt, wie man das gleiche Problem auf verschiedene Arten anpacken kann. In den späteren Kapiteln dieses Buches werden Sie noch viele Stellen vorfinden, an denen zur optimalen Problemlösung Kreativität gefragt ist. Das Schreiben eines regulären Ausdrucks hat oft mehr mit Kunst als mit trockener Wissenschaft zu tun.

Andere Quantoren: Repetition

Eine ähnliche Funktion wie das Fragezeichen haben ⌜+⌟ (*Plus*) und ⌜*⌟ (Sternchen oder der Kürze halber einfach *Stern*). Das Metazeichen ⌜+⌟ bedeutet »ein oder mehr des unmittelbar Vorausgehenden«, und ⌜*⌟ bedeutet »eine beliebige Anzahl, inklusive null, des unmittelbar Vorausgehenden«. Mit anderen Worten bedeutet ⌜…*⌟: »Versuche das Vorausgehende so häufig wie möglich zu erkennen, aber es ist auch gut, wenn es gar nicht gefunden wird.« Die Variante mit dem Pluszeichen, ⌜…+⌟, ist insofern ähnlich, als auch hier versucht wird, so viele Zeichen wie möglich zu erkennen. Allerdings muss hier das vorausgehende Zeichen mindestens einmal vorkommen. Die drei Metazeichen Fragezeichen, Pluszeichen und Stern nennen wir *Quantoren* (engl. *quantifier*), weil sie sich auf die Anzahl, das Quantum, der vorausgehenden Zeichen beziehen.

Wie bei ⌜…?⌟ wird das Element ⌜…*⌟ immer passen, die Frage ist nur, wie viele Zeichen des zu prüfenden Textes damit erkannt werden. Im Gegensatz dazu muss bei ⌜…+⌟ das vorausgehende Zeichen im Text mindestens einmal tatsächlich vorkommen.

Zum Beispiel erlaubt ⌜␣?⌟ ein Leerzeichen oder auch keines, und ⌜␣*⌟ erlaubt *jede beliebige Anzahl* von Leerzeichen. Wir können das benutzen, um das <H[1-6]>-Beispiel von Seite 9 flexibler zu formulieren. Die HTML-Spezifikation[8] besagt, dass Leerzeichen unmittelbar vor dem schließenden > erlaubt sind, wie in <H3␣> und <H4␣␣␣>. Wenn wir in unseren regulären Ausdruck ⌜␣*⌟ da einsetzen, wo Leerzeichen erlaubt (aber nicht vorgeschrieben) sind, erhalten wir ⌜<H[1-6]␣*>⌟. Der Ausdruck erkennt noch immer <H1>, weil die Leerzeichen nicht erzwungen werden, er ist aber flexibler, weil er außerdem auch die anderen Varianten erkennt.

Als weiteres HTML-Tag betrachten wir <HR␣SIZE=14>, das einen 14 Pixel dicken horizontalen Balken (»Horizontal Rule«) spezifiziert. Wie beim <H3>-Beispiel vorhin sind auch hier Leerzeichen vor der schließenden spitzen Klammer zulässig. Außerdem sind Leerzeichen links und rechts vom Gleichheitszeichen erlaubt. Mindestens ein Leerzeichen muss zwischen HR und SIZE stehen, mehrere sind erlaubt. Für diese letzte Forderung könnten wir ⌜␣␣*⌟ schreiben, aber wir werden ⌜␣+⌟ benutzen. Das Pluszeichen fordert mindestens ein Leerzeichen, erlaubt aber mehrere. Ist Ihnen klar, warum das dasselbe erkennt wie ⌜␣␣*⌟? Das ergibt den regulären Ausdruck ⌜<HR␣+SIZE␣*=␣*14␣*>⌟.

Damit ist unser regulärer Ausdruck flexibler in Bezug auf Leerzeichen, aber noch immer etwas stur bezüglich der Dicke des Balkens. Statt nur Tags mit einer ganz bestimmten Dicke (14) möchten wir Balken jeder Dicke finden. Dazu ersetzen wir ⌜14⌟ durch einen Ausdruck, der eine beliebige ganze Zahl beschreibt. Nun, eine Zahl besteht aus einer oder mehreren Ziffern, und eine Ziffer lässt sich mit ⌜[0-9]⌟ beschreiben. »Eine oder mehrere« können wir mit dem Pluszeichen ausdrücken. Also ersetzen wir ⌜14⌟ durch ⌜[0-9]+⌟. Wie man sieht, ist eine Zeichenklasse eine »Einheit«, der direkt ein Plus, ein Fragezeichen usw. nachgestellt werden kann, ohne dass man sie in runde Klammern einfassen muss.

Damit erhalten wir ⌜<HR␣+SIZE␣*=␣*[0-9]+␣*>⌟ – schon ganz schön kompliziert, auch wenn ich hier mit Halbfettdruck, mit kleinen Abständen zwischen den Zeichengruppen und dem »sichtbaren Leerzeichen« ›␣‹ sehr nachgeholfen habe! (Immerhin kennt *egrep* die Option -i, ☞ 15, mit der die Groß- und Kleinschreibung ignoriert wird, sonst hätte ich ⌜[Hh][Rr]⌟ statt ⌜HR⌟ schreiben müssen.) Der nackte reguläre Ausdruck ⌜<HR +SIZE *= *[0-9]+ *>⌟ ist zweifellos noch verwirrender. Dieser reguläre Ausdruck sieht vor allem deshalb ungewohnt aus, weil sich die meisten Sterne und Pluszeichen auf das Leerzeichen davor beziehen, und unsere Augen sind darauf konditioniert, Leerraum in Texten gesondert zu behandeln. Da muss man sich beim Lesen von regulären Ausdrücken etwas umstellen, weil bei regulären Ausdrücken das Leerzeichen keine besondere Bedeutung hat. Es ist konzeptionell nicht verschieden von, sagen wir mal, j oder 4. (In späteren Kapiteln werden wir sehen, dass man manche Programme so konfigurieren kann, dass sie Leerzeichen in regulären Ausdrücken nicht beachten, aber *egrep* gehört nicht zu diesen.)

8 Falls Sie mit HTML nicht vertraut sind, macht das fast gar nichts. Ich benutze HTML als typisches Beispiel aus der Praxis, und ich erläutere dazu so viel, wie zum Verständnis der regulären Ausdrücke nötig ist. HTML-Experten werden mehr Subtilitäten in den Beispielen entdecken, aber um solche Details geht es in dieser Einführung noch nicht.

Das Beispiel ist aber noch nicht ausgereizt. Die Dickenangabe, das Attribut SIZE, ist beim HR-Tag optional, man kann auch ganz einfach <HR> schreiben und bekommt so einen Balken mit vorbestimmter Dicke (zusätzliche Leerzeichen vor dem > sind wie immer erlaubt). Wie sieht ein regulärer Ausdruck aus, der beide Formen zulässt? Tipp: Es ist wichtig zu verstehen, dass der Teil mit der Größenangabe *optional* ist. ❖ Auf der nächsten Doppelseite finden Sie die Antwort.

Die Antwort zur vorhergehenden Aufgabe verdient einige Beachtung. Sie zeigt, wie die Quantoren Fragezeichen, Stern und Plus zusammenspielen und wie sie in der Praxis benutzt werden. Tabelle 1-2 fasst ihre Bedeutungen zusammen.

Jeder Quantor fordert eine minimale Anzahl an Treffern, damit der reguläre Ausdruck passt, und eine maximale Anzahl an Versuchen. Bei manchen ist das Minimum null, bei manchen ist das Maximum unendlich.

Tabelle 1-2: Zusammenfassung der Quantoren »Repetitions-Metazeichen«

	Minimum	Maximum	Bedeutung
?	null	1	Einmal erlaubt, keinmal möglich (›*kein- oder einmal*‹)
*	null	kein Limit	Jede Anzahl (auch null) erlaubt (›*keinmal, einmal oder mehrmals*‹)
+	1	kein Limit	Einmal erlaubt, mehrfach möglich (›*einmal oder mehrmals*‹)

Explizites Minimum und Maximum: Intervalle

Manche *egrep*-Versionen unterstützen eine Metasequenz, mit der man Zahlen für das erlaubte Minimum und Maximum angeben kann: ⌜…{*min*, *max*}⌟. Dies ist der *Intervall*-Quantor. Zum Beispiel findet ⌜…{3,12}⌟ bis zu zwölf Dinge der vorher angegebenen Art, aber es müssen mindestens drei vorhanden sein. Der Ausdruck ⌜[a-zA-Z]{1,5}⌟ würde auf alle Aktientitel an der Börse von New York passen (1 bis 5 Buchstaben). Mit dieser Notation bedeutet {0,1} nichts anderes als das Fragezeichen.

Nur wenige Versionen von *egrep* unterstützen diese Notation, aber viele der gewichtigeren Programme kennen sie. Ich werde sie daher erst in Kapitel 3 genauer behandeln, wenn wir das breite Spektrum von Metazeichen betrachten, das heutige Werkzeuge anbieten.

Klammern und Rückwärtsreferenzen

Bis jetzt haben wir Klammern auf zwei Arten verwendet: um den Geltungsbereich einer Alternation einzuschränken und um Gruppen zu bilden, auf die wir Quantoren wie das Fragezeichen oder den Stern anwenden können. Ich möchte hier eine dritte Verwendung anführen, die zwar bei *egrep* häufig nicht verwendet werden kann (immerhin unterstützt die weitverbreitete GNU-Version von *egrep* diese), die aber bei anderen Werkzeugen sehr oft gebraucht wird.

Bei vielen Implementationen von regulären Ausdrücken können sich Klammern an den Text »erinnern«, der auf den Unterausdruck gepasst hat, den sie einschließen. Wir werden diese Eigenschaft für eine erste Lösung des Problems mit den verdoppelten Wörtern vom Anfang dieses Kapitels benutzen. Wenn uns das das zu suchende verdoppelte Wort schon bekannt ist (wie »das« weiter vorne in diesem Satz – haben Sie's gemerkt?), dann können wir explizit danach suchen: ⌜das␣das⌟. In diesem Fall würden wir (vorausgesetzt, wir benutzen die -i-Option) auch das␣Dasein finden. Das ließe sich mit einem *egrep* vermeiden, das die Metasequenzen für Wortgrenzen (⌜\<···\>⌟) kennt: ⌜\<das␣das\>⌟. Wir könnten auch ⌜␣+⌟ statt des Leerzeichens benutzen.

Auf diese Art nach allen möglichen verdoppelten Wörtern zu suchen wäre allerdings unmöglich. Nachdem wir ein Wort gefunden haben, wäre es schön, wenn wir irgendwie ausdrücken könnten: »Und jetzt suche dasselbe Wort nochmals!«. Falls Ihr *egrep Rückwärtsreferenzen* unterstützt, können Sie das. Rückwärtsreferenzen sind eine Möglichkeit, nochmals nach dem gleichen Text zu suchen, der weiter vorne im regulären Ausdruck schon einmal gefunden wurde – ohne dass der Text explizit bekannt sein muss.

Wir beginnen mit ⌜\<das␣+das\>⌟ und ersetzen das erste ⌜das⌟ durch einen regulären Ausdruck, der auf ein beliebiges Wort passt, sagen wir mal ⌜[A-Za-z]+⌟. Diesen Ausdruck klammern wir ein – aus Gründen, die bald klar werden. Außerdem ersetzen wir das zweite ›das‹ durch die neue Metasequenz ⌜\1⌟. Das ergibt ⌜\<**(**[A-Za-z]+**)**␣+**\1**\>⌟.

Werkzeuge, die Rückwärtsreferenzen unterstützen, »erinnern« sich an den Text, der auf einen Unterausdruck innerhalb eines Klammerpaares gepasst hat, und die Metasequenz ⌜\1⌟ repräsentiert genau diesen Text weiter hinten im regulären Ausdruck.

Das geht natürlich auch mit mehreren Klammerpaaren. Mit ⌜\1⌟, ⌜\2⌟, ⌜\3⌟ usw. kann man auf den Text aus dem ersten, zweiten, dritten usw. Klammerpaar zurückgreifen. Beim Nummerieren der Klammerpaare zählt man die öffnenden Klammern von links nach rechts, bei ⌜([a-z])([0-9])\1\2⌟ bezieht sich das ⌜\1⌟ auf den von ⌜[a-z]⌟ erkannten Text, und ⌜\2⌟ bezieht sich auf den von ⌜[0-9]⌟.

Bei unserem ›das␣das‹-Beispiel passt der Unterausdruck ⌜[A-Za-z]+⌟ auf das erste das. Der Unterausdruck ist eingeklammert, damit wird dieses ›das‹ durch das Metazeichen ⌜\1⌟ zugänglich. Falls das darauffolgende ⌜␣+⌟ gefunden wird, wird nicht nach irgendeinem Wort gesucht, sondern nach dem Wort, das in ⌜\1⌟ aufbewahrt ist. Wenn dieses zweite, gleiche Wort gefunden wird, wird mit ⌜\>⌟ noch auf ein Wortende getestet (damit nicht Wortteile wie bei das␣Dasein gefunden werden). Nur wenn auch dieser Test wahr ist, passt der ganze reguläre Ausdruck. Einen regulären Ausdruck, der der Problemstellung immer gerecht wird, können wir sowieso nicht finden, weil es ja durchaus Sätze gibt, in denen verdoppelte Wörter (wie das »der« in diesem Satz) korrekt sind. Wenn die verdächtigen Zeilen ausgegeben werden, sieht man sehr schnell, wo etwas nicht stimmt.

Einen Unterausdruck optional machen

❖ *Antwort zum Problem von Seite 20.*

In diesem Fall ist mit »optional« Folgendes gemeint: einmal erlaubt, aber nicht Bedingung. Daher werden wir ⌜?⌟ benutzen. Weil das optionale Element mehr als ein einzelnes Zeichen umfasst, werden wir es in Klammern setzen müssen: ⌜(···)?⌟. In unseren regulären Ausdruck eingesetzt, ergibt das:

⌜<HR(+SIZE *= *[0-9]+)? *>⌟

Beachten Sie, dass das letzte ⌜ *⌟ außerhalb von ⌜(···)?⌟ steht. Damit erreichen wir, dass auch Texte wie <HR > erkannt werden. Stünde es innerhalb der Klammern, würden Leerzeichen vor dem > nur dann erkannt, wenn eine explizite Dickenangabe vorhanden ist.

Beachten Sie außerdem, dass im Gegensatz dazu das ⌜ +⌟ vor dem SIZE in den Klammern enthalten ist. Wäre das nicht so, dann müsste mindestens ein Leerzeichen nach dem HR vorkommen, auch wenn kein SIZE-Attribut im Tag vorkommt. Dann würde <HR> nicht erkannt.

Als ich dieses Beispiel entworfen habe, habe ich es natürlich ausprobiert, und zwar an dem Text dieses Buches, den ich bis dahin geschrieben hatte. (Ich verwendete eine *egrep*-Version, die sowohl ⌜\<···\>⌟ als auch Rückwärtsreferenzen beherrscht.) Um den Test noch etwas nützlicher zu machen, damit ›Das das‹ auch gefunden würde, habe ich die erwähnte `-i`-Option angegeben, um die Groß- und Kleinschreibung zu ignorieren:[9]

```
% egrep -i '\<([a-z]+) +\1\>' datei ...
```

Es ist mir ein bisschen peinlich, aber ich habe im ganzen Buch vierzehn falsche Paare von ›verdoppelten verdoppelten‹ Wörtern gefunden!

So nützlich dieser reguläre Ausdruck ist, so wichtig ist es auch, seine Grenzen zu kennen. Weil *egrep* jede Zeile für sich allein betrachtet, ist es nicht möglich, verdoppelte Wörter zu finden, die sich auf zwei Zeilen verteilen (eines am Ende der ersten, das verdoppelte am Anfang der nächsten Zeile). Gerade diese sind aber besonders unauffällig. Um auch diesen Fall zu behandeln, braucht es mächtigere Instrumente als *egrep*; Sie werden im nächsten Kapitel welche kennenlernen.

9 Bei bestimmten Versionen von *egrep*, auch GNU-*egrep*, muss allerdings der von der Rückwärtsreferenz erkannte Text auch bezüglich der Groß- und Kleinschreibung gleich dem Text aus dem entsprechenden Klammerpaar sein, auch dann, wenn die `-i`-Option benutzt wird. Mit der *egrep*-Zeile wird also sowohl »das das« als auch »Das Das« gefunden, nicht aber »Das das«, wie das zum Beispiel Perl tun würde.

Ausbrecher!

Einen wichtigen Punkt habe ich noch nicht angesprochen: wie man nach einem Zeichen sucht, das normalerweise in einem regulären Ausdruck als Metazeichen interpretiert würde. Wenn ich zum Beispiel nach einem Internet-Hostnamen wie `ega.att.com` mit ⌜`ega.att.com`⌟ suchte, fände der reguläre Ausdruck auch Dinge wie `megawatt computing`. Wie wir wissen, ist ⌜`.`⌟ ein Metazeichen, das auf jedes Zeichen passt.

Die Metasequenz, um nach einem wirklichen Punkt zu suchen, ist ein Punkt mit einem Backslash davor: ⌜`ega\.att\.com`⌟. Der Backslash wird in diesem Zusammenhang auch *Escape* (Ausbruch) genannt, weil aus der üblichen Bedeutung ausgebrochen wird. Dies funktioniert mit allen Metazeichen außerhalb von Zeichenklassen.[10] Wenn vor einem Metazeichen ein Backslash steht, verliert es seine besondere Bedeutung und wird zu einem Literal. Wenn Sie wollen, können Sie die Sequenz »Backslash-Metazeichen« als weitere Metasequenz ansehen, die das ursprüngliche Literal erzeugt. Es funktioniert so oder so.

Ein anderes Beispiel: ⌜`\([a-zA-Z]+\)`⌟ kann benutzt werden, um nach einem eingeklammerten Wort zu suchen, etwa ›`(sehr)`‹. Die Backslashes in den Sequenzen ⌜`\(`⌟ und ⌜`\)`⌟ nehmen den Klammern ihre spezielle Bedeutung, und der reguläre Ausdruck sucht im Text nach literalen runden Klammern.

Vor einem Nicht-Metazeichen kann ein Backslash Verschiedenes bewirken; das hängt vom verwendeten Werkzeug oder sogar von dessen Version ab. Wie wir gesehen haben, behandeln manche Versionen von *egrep* die Sequenzen ⌜`\<`⌟, ⌜`\>`⌟, ⌜`\1`⌟ usw. als Metasequenzen, andere nicht. Wir werden in späteren Kapiteln mehr davon sehen.

Erweiterung der Fundamente

Ich hoffe, die bisherigen Beispiele und Erläuterungen haben eine solide Basis geschaffen für das Verständnis von regulären Ausdrücken. Aber es muss gesagt sein, dass wir noch nirgends wirklich in die Tiefe der Materie vorgedrungen sind. Es gibt noch viel zu lernen!

Linguistisches Divertissement

Ich habe eine Anzahl Features erwähnt, die fast jedes *egrep* kennt. Aber es gibt mehr davon, und nicht jede Version unterstützt alle diese Features.

Unglücklicherweise unterscheidet sich die Sprache der regulären Ausdrücke dabei in keiner Weise von natürlichen Sprachen: Es gibt die verschiedensten Dialekte, von Akzenten und Unterschieden in der Betonung gar nicht zu reden. Es macht fast den Eindruck, als ob jedes neue Programm, das reguläre Ausdrücke unterstützt, seine eigenen »Verbesserungen« anbringt. Natürlich ist das Gebiet im Fluss, aber über die Jahre hat sich eine breite Palette von Dialekten ausgebildet. Wir werden etliche davon kennenlernen.

10 Bei vielen Programmiersprachen und Software-Werkzeugen sind Escapes auch innerhalb von Zeichenklassen zulässig, aber nicht bei *egrep*. Hier würde der Backslash als normales Zeichen in die Liste der von der Zeichenklasse erlaubten Zeichen aufgenommen.

Das Ziel eines regulären Ausdrucks

Global gesehen passt ein regulärer Ausdruck entweder auf ein Stück Text (bei *egrep*: eine Zeile), oder er passt eben nicht. Wenn Sie einen regulären Ausdruck zusammenstellen, müssen Sie sich dieses Seilziehen zwischen »passen« und »nicht passen« öfter vergegenwärtigen.

egrep kümmert sich nicht darum, *wo* auf einer Zeile ein Muster gefunden wird: Wenn es irgendwo auf der Zeile vorkommt, wird die Zeile ausgegeben. Bei anderen Programmen kann das sehr wohl eine Rolle spielen. Falls der zu prüfende Text etwa so aussieht

```
...Postleitzahl 50670. Bitte 4,50 Euro in Briefmarken für Rückporto...
```

und Sie nur nach Zeilen mit Ziffern suchen (⌜`[0-9]+`⌟), dann kann es Ihnen egal sein, welche der Zahlen einen Treffer erzeugt. Wenn Sie dagegen die gefundenen Zahlen *weiterverarbeiten* wollen (sie in eine Datei schreiben, zu der Zahl etwas addieren usw.), dann ist es ziemlich wichtig, *welche* der Zahlen einen Treffer bewirkt. Schließlich spielt es schon eine Rolle, ob Sie Briefmarken für 50 670 Euro oder für 4,50 Euro kaufen.

Weitere Beispiele

Wie bei jeder Sprache ist Übung *sehr wichtig*. Daher gebe ich hier ein paar Beispiele, die Suchmuster für häufige Konstrukte behandeln.

Beim Konstruieren von regulären Ausdrücken ist das Finden von Mustern die halbe Miete. Die andere Hälfte ist natürlich das Vermeiden von unerwünschten Treffern. Beides ist wichtig, aber für den Moment will ich das Augenmerk auf die erste Hälfte lenken. Wir werden die Beispiele nicht bis ins letzte Detail ausloten, weil es vor allem darum geht, Übung im Umgang mit regulären Ausdrücken zu erlangen.

Variablennamen

Viele Programmiersprachen haben *Bezeichner* (für Variablen und Ähnliches), die aus alphanumerischen Zeichen und dem Unterstrich zusammengesetzt sind, die aber nicht mit einer Ziffer beginnen dürfen, also ⌜`[a-zA-Z_][a-zA-Z_0-9]*`⌟. Die erste Zeichenklasse ergibt ein Muster für den ersten Buchstaben, die zweite (mit dem zugehörigen Stern) eines für die restlichen Zeichen des Bezeichners. Falls ein Limit für die Länge der Bezeichner besteht (oft 32) und falls unser Werkzeug die ⌜`{`*`min`*`,`*`max`*`}`⌟-Notation versteht, kann der Stern durch ⌜`{0,31}`⌟ ersetzt werden (dieses Konstrukt, das Intervall, wurde kurz auf Seite 20 erwähnt).

Ein String in Anführungszeichen

Eine einfache Lösung könnte sein: ⌜`"[^"]*"`⌟.

Die Anführungszeichen zu Anfang und Ende des Ausdrucks müssen so im Text erscheinen. Dazwischen darf alles vorkommen … außer ein weiteres Anführungszeichen! Also benutzen wir die negierte Zeichenklasse ⌜`[^"]`⌟ und einen Stern, der besagt, dass es von diesen Nicht-Anführungszeichen beliebig viele geben darf.

Eine weiter gehende (und kompliziertere) Definition eines Strings in Anführungszeichen erlaubt solche Gänsefüßchen auch im Innern des Strings, allerdings müssen diese dann durch einen vorangestellten Backslash geschützt sein; etwa `"^•oder•\"Hütchen\"•oder•Zirkumflex"`. Wir werden in späteren Kapiteln auf diese schwierigere Interpretation zurückkommen, wenn wir untersuchen, wie die Mustersuche genau vor sich geht.

Heller und Pfennig

Hier folgt ein Ansatz für einen regulären Ausdruck für gebrochene Geldbeträge (wir nehmen hier Dollar und Cent): ⌜`\$[0-9]+(\.[0-9][0-9])?`⌟.

Von außen nach innen vorgehend, ist dies zunächst einmal ein regulärer Ausdruck aus drei Teilen: ⌜`\$`⌟ und ⌜`···+`⌟ und ⌜`(···)?`⌟. Salopp übersetzt bedeutet er etwa: »Ein Dollarzeichen, ein Dingsda oder ein paar davon und am Schluss vielleicht ein anderes Dingsda«. In diesem Fall ist das erste »Dingsda« eine Ziffer (mehrere Ziffern ergeben eine Zahl), und das »andere Dingsda« ist die Kombination von einem Dezimalpunkt und zwei Ziffern.

Dieser Ansatz ist aus mehreren Gründen etwas naiv. Zunächst werden Geldbeträge manchmal in Dreiergruppen aufgeteilt (in den USA mit Kommas, in Deutschland mit Punkten, in der Schweiz mit Hochkommas): `$1,000`. Mit *egrep* interessiert uns im Allgemeinen nur, ob ein Dollar-Betrag auf der Zeile vorkommt; ob er nun Cent-Beträge enthält, ist irrelevant: Die Zeile wird auf jeden Fall ausgegeben. Bei *egrep* spielt es keine Rolle, *wie viel* vom Text auf den regulären Ausdruck passt, es ist nur wichtig, ob der Ausdruck *überhaupt* passt oder nicht.

Falls wir aber an Zeilen interessiert sind, die nichts außer diesem Dollar-Betrag enthalten, würden wir den regulären Ausdruck in ⌜`^···$`⌟ einkleiden. In diesem Fall sind die optionalen Cent-Beträge sehr wohl wichtig, weil sie zwischen dem ganzen Dollar-Betrag und dem Zeilenende vorkommen können – oder auch nicht.

Eine Art von Dollar-Beträgen erkennt unser regulärer Ausdruck nicht: ›`$.49`‹. Um dieses Problem zu lösen, ist man versucht, das Plus durch einen Stern zu ersetzen – aber das geht schief. Warum? Das lasse ich erst einmal offen, bis wir uns in Kapitel 5 mit einem ganz ähnlichen Problem befassen (☞ 200).

Eine HTTP/HTML-URL

Das allgemeine Format für eine URL ist erstaunlich kompliziert, und deshalb ist es auch der entsprechende reguläre Ausdruck. Wenn wir unsere Ansprüche etwas herunterschrauben und nur die häufigsten Typen von URLs betrachten, wird es deutlich einfacher. Eine Anwendung dafür wäre etwa die Suche im eigenen E-Mail-Archiv nach einer URL, an die man sich nur noch undeutlich erinnert; dabei ist formale Korrektheit nicht wichtig, es geht nur darum, viele URLs auszugeben. Die gesuchten URLs findet man beim schnellen Durchlesen der Ausgabe von *egrep*.

»Normale« HTTP/HTML-URLs haben etwa die folgende generelle Form:

```
http://servername/pfad.html
```

Auch die Endung `.htm` kommt häufig vor.

Welchen genauen Regeln ein Hostname (hier der *servername*, zum Beispiel `www.yahoo.com`) genügen muss – auch das ist nicht ganz so einfach. Im Moment genügt uns ⌜`[-a-z0-9_.]+`⌟ für das, was nach dem ›`http://`‹ folgt. Der *pfad* kann noch vielgestaltiger sein, wir benutzen hier ⌜`[-a-z0-9_:@&?=+,.!/~*%$]*`⌟. Beachten Sie, dass beide Zeichenklassen mit dem Bindestrich als Literal beginnen – sonst würde er als Zeichen für einen Bereich interpretiert (☞ 9).

Zusammengesetzt sieht unser erster Ansatz folgendermaßen aus:

```
% egrep -i '\<http://[-a-z0-9_.]+/[-a-z0-9_:@&?=+,.!/~*%$]*\.html?\>' dateien ...
```

Wieder sind wir ziemlich großzügig und lassen viel mehr zu, als eigentlich erlaubt ist. Der Ausdruck würde auch auf »`http://..../foo.html`« passen, was sicher keine zulässige URL ist. Ist das ein Problem? Das kommt ganz darauf an, was Sie eigentlich erreichen wollen. Beim Durchsuchen von E-Mails spielt es sicher keine Rolle, wenn wir ein paar falsche Treffer bekommen. Wir kämen auch mit etwas Einfacherem wie

```
% egrep -i '\<http://[^ ]*\.html?\>' dateien ...
```

gut zurecht. Es ist wichtig zu wissen, wie die erwarteten Daten aussehen. Wenn wir lernen, wie man einen regulären Ausdruck aufbaut, wird es immer wichtiger, die Balance zwischen Komplexität und Vollständigkeit zu halten. Wir werden das Beispiel im nächsten Kapitel noch gründlicher studieren.

Ein HTML-Tag

Mit einem Werkzeug wie *egrep* ist es nicht besonders sinnvoll, nach Zeilen mit HTML-Tags zu suchen. Dennoch wird ein regulärer Ausdruck, der HTML-Tags erkennt, sehr oft benötigt, gerade in Programmiersprachen, die in den späteren Kapiteln behandelt werden.

Wenn wir einfache Fälle wie ›`<TITLE>`‹ oder ›`<HR>`‹ betrachten, drängt sich ein einfacher Ausdruck wie ⌜`<.*>`⌟ auf. Fast jeder kommt auf diese Idee, aber sie ist falsch. Der Ausdruck ⌜`<.*>`⌟ lautet in Worten: »Suche nach einem ›<‹, gefolgt von irgendetwas, gefolgt von ›>‹.« So gesehen wird klar, dass der Ausdruck auch auf mehrere Tags auf einmal passen kann, wie beim unterstrichenen Teil in ›`ein <I>kleines</I> Beispiel`‹.

Das mag erstaunen, aber wir sind noch immer im ersten Kapitel und verstehen reguläre Ausdrücke erst ganz oberflächlich. Das Beispiel soll einerseits zeigen, dass reguläre Ausdrücke nicht besonders schwierig sind, dass sie aber dann schwierig werden, wenn man nicht versteht, was eigentlich vor sich geht. In den nächsten Kapiteln betrachten wir alle Einzelheiten von regulären Ausdrücken, um dieses Problem zu verstehen und zu lösen.

Amerikanische Uhrzeiten wie »9:17 am« oder »12:30 pm«

Das Problem »amerikanische Uhrzeiten erkennen«[11] kann man von salopp bis sehr strikt behandeln. Ein Ausdruck wie

```
⌜[0-9]?[0-9]:[0-9][0-9]•(am|pm)⌟
```

erkennt sowohl `9:17•am` als auch `12:30•pm`, aber er erlaubt auch Unsinniges wie `99:99•pm`.

Wenn wir die Stundenzahl anschauen, stellen wir fest, dass die erste Zahl eine 1 sein muss, wenn die Stundenzahl zweistellig ist. Aber `⌜1?[0-9]⌟` erlaubt noch immer eine Stundenzahl von 19 (und auch 0), also ist es vielleicht gescheiter, den Stunden-Teil in zwei Alternativen aufzuteilen: `⌜1[012]⌟` für zweistellige Stundenzahlen und `⌜[1-9]⌟` für einstellige. Daraus resultiert `⌜(1[012]|[1-9])⌟`.

Der Minuten-Teil ist einfacher. Die erste Ziffer muss auf `⌜[0-5]⌟` passen, die zweite Ziffer können wir bei `⌜[0-9]⌟` belassen. Wenn wir alle einzelnen Ausdrücke kombinieren, ergibt sich `⌜(1[012]|[1-9]):[0-5][0-9]•(am|pm)⌟`.

Wie erweitert man das auf das anderswo übliche 24-Stunden-Schema, mit Stundenzahlen von 0 bis 23? Als zusätzliche Erschwernis sollen hier mit null beginnende Stundenzahlen wie `09:59` erlaubt sein. ❖ Versuchen Sie, Ihre eigene Lösung zu finden, bevor Sie auf der nächsten Doppelseite meine anschauen.

Reguläre Ausdrücke: Terminologie

Regex

Wie Ihnen zweifellos aufgefallen ist, ist die dauernde Wiederholung der vollen Bezeichnung »regulärer Ausdruck« auf die Dauer ermüdend. Ich benutze dafür normalerweise »Regex«, die Kurzform für das englische »*regular expression*«. Manchmal spreche ich davon, einen Gedanken ins »Regexische« zu übersetzen, oder benutze gar »regexifizieren«.[12] Die Bezeichnung »Regex-Maschine« oder einfach »Maschine« benutze ich für das Programm, das die eigentliche Arbeit des Pattern-Matchings verrichtet.

Pattern-Matching

Wenn ich bei der Mustersuche oder dem *Pattern-Matching* von einem *Treffer* rede, dann meine ich, dass die Regex-Maschine einen passenden String in dem zu untersuchenden Text

11 US-Amerikaner haben große Mühe damit, dass ein Tag 24 Stunden haben könnte. Sie bevorzugen meistens die Zwölfstunden-Notation und setzen ein »am« (manchmal »a.m.« oder nur »a«, *ante meridiem*, vormittags) oder ein »pm« (*post meridiem*, nachmittags) dahinter. Fast völlig unfassbar sind dem Amerikaner Null-Uhrzeiten wie `0.13` oder `12.59`, er wird dafür »`12:13` pm« und »`12:59` am« schreiben, obwohl `12.59` entschieden nicht mehr zum Vormittag gehört. (Anm. d. Ü.)

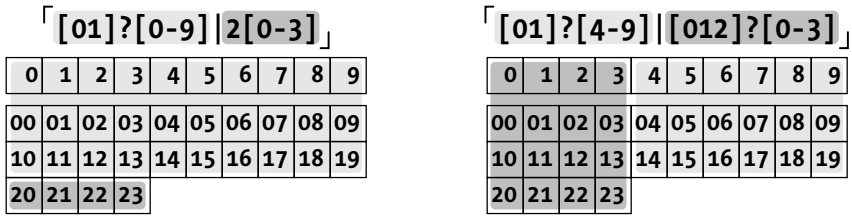
12 Sie werden anderswo auch das unschöne »Regexp« antreffen. Ich kann das kaum aussprechen; Lispelnde haben damit wahrscheinlich keine Mühe.

Die Uhrzeit-Regex für das 24-Stunden-Schema erweitern

❖ *Antwort zum Problem von Seite 27.*

Es gibt natürlich viele korrekte Antworten; bei dieser hier wird die gleiche Logik wie vorhin verfolgt. Diesmal teile ich die Aufgabe in drei Teile auf: einen für die Morgenstunden von `00` bis `09` (die führenden Nullen sind dabei optional), einen für die Tagesstunden von `10` bis `19` und einen für die Nachtstunden von `20` bis `23`. Schritt für Schritt in die Sprache der regulären Ausdrücke übersetzt: ⌜`0?[0-9]|1[0-9]|2[0-3]`⌟.

Die ersten zwei Alternativen können wir zu einer einzigen zusammenfassen. Dadurch erhalten wir das kürzere ⌜`[01]?[0-9]|2[0-3]`⌟. Warum das so ist, ist vielleicht nicht auf den ersten Blick klar. Die Abbildung mag helfen, und sie zeigt auch eine weitere Möglichkeit, das Problem anzugehen. Die schattierten Gruppen umfassen jeweils Zahlen, die sich mit einem Teil einer Alternation (einer Alternative) beschreiben lassen.



gefunden hat. Genauer gesagt, passt die Regex ⌜`i`⌟ nicht auf den String `ding`, sie passt nur auf das `i` im Text `ding`. Dies mehr der Vollständigkeit halber; es bestehen diesbezüglich kaum Verständnisschwierigkeiten.

Metazeichen

Ob ein Zeichen ein Metazeichen (oder eine Metasequenz, ich benutze beide Begriffe unterschiedslos) ist oder nicht, hängt davon ab, wo genau in der Regex das Zeichen vorkommt. Beispielsweise ist ⌜`*`⌟ ein Metazeichen, aber nur außerhalb von Zeichenklassen und nur, wenn es nicht durch einen vorangestellten Backslash geschützt ist. Ein »Escape« ist ein durch einen vorangestellten Backslash geschütztes Zeichen – meistens. In ⌜`\*`⌟ bilden Stern und Backslash ein Escape, aber nicht in ⌜`\\*`⌟ (der erste Backslash schützt hier den zweiten vor seiner üblichen Interpretation), obwohl der Stern beide Male ein vorangestelltes Escape-Zeichen besitzt.

Aufgrund der verschiedenen Regex-Dialekte kann ein Zeichen einmal als Metazeichen auftreten und ein andermal nicht. Kapitel 3 zeigt das im Detail.

Dialekte

Wie schon angedeutet, benutzen die verschiedenen Werkzeuge reguläre Ausdrücke für sehr unterschiedliche Dinge, und die unterstützten Metazeichen können sehr verschieden sein. Als Beispiel dienen hier wiederum die Wortgrenzen: Manche Versionen von *egrep* unterstützen dafür die \<···\>-Notation. Andere unterscheiden nicht zwischen Wortanfang und Wortende, sondern kennen nur ein Metazeichen, ⌜\b⌟, für »Wortgrenze«. Noch andere kennen alle drei, und natürlich gibt es auch solche, die keines dieser Metazeichen kennen.

Ich benutze den Begriff »Dialekt«, um die Summe dieser kleineren Implementationsunterschiede zu bezeichnen, dies selbstverständlich in Analogie zu natürlichen Sprachen. Oberflächlich betrachtet, geht es nur darum, ob bestimmte Metazeichen unterstützt werden oder nicht, aber es steckt mehr dahinter. Auch wenn zwei Programme die ⌜\<···\>⌟-Metazeichen kennen, können sie noch immer verschiedene Vorstellungen darüber haben, was denn nun ein Wort ausmacht. Das wird wichtig, wenn ein Programm wirklich *benutzt* und nicht nur evaluiert wird.

»Dialekt« bedeutet aber nicht das Gleiche wie »Werkzeug« oder »Programm«. So wie zwei Personen die gleiche Mundart sprechen können, gibt es völlig unterschiedliche Programme, die den gleichen Dialekt von regulären Ausdrücken verwenden. Umgekehrt gibt es verschiedene Programme gleichen Namens (und mit dem gleichen Zweck), die subtil unterschiedlichen Mikrodialekten angehören (manchmal sind die Unterschiede auch gar nicht so subtil). Die vielen *egrep*-Implementationen unterstützen fast ebenso viele Dialekte von regulären Ausdrücken.

In den späten Neunzigern wurde die Programmiersprache Perl mit ihrem sehr reichen Dialekt von regulären Ausdrücken sehr bekannt. Bald haben andere Programmiersprachen von Perl abgekupfert. Manche geben das ganz offen zu und nennen ihre Implementation »Perl-kompatibel«. Die Nachahmer sind PHP, Python, viele Regex-Packages für Java, .NET von Microsoft, Tcl und eine ganze Reihe von Bibliotheksroutinen für C. Dennoch unterscheiden sich viele in kleinen, aber wichtigen Aspekten. Außerdem werden auch die regulären Ausdrücke von Perl weiterentwickelt – manchmal werden neuerdings auch Ideen der Nachahmer aufgenommen. Die Landschaft der regulären Ausdrücke verändert sich, wird weiter und damit nicht immer einfacher.

Unterausdruck

Die Bezeichnung *Unterausdruck* bedeutet zunächst einfach einen Teil eines größeren regulären Ausdrucks, meistens aber etwas spezifischer einen geklammerten Teil oder eine Alternative innerhalb einer Alternation. Zum Beispiel ist ⌜Subject|Date⌟ ein Unterausdruck von ⌜^(Subject|Date):␣⌟. Innerhalb dieses Ausdrucks sind die Alternativen ⌜Subject⌟ und ⌜Date⌟ ebenfalls Unterausdrücke.

Dagegen ist etwas wie ⌜1-6⌟ kein Unterausdruck von ⌜H[1-6]␣*⌟, weil das ⌜1-6⌟ Teil einer »Einheit«, einer Zeichenklasse ist, die man nicht weiter zerlegen kann. Umgekehrt sind ⌜H⌟, ⌜[1-6]⌟ und ⌜␣*⌟ sehr wohl Unterausdrücke des Originals.

Die Quantoren (Stern, Plus, Fragezeichen) beziehen sich immer auf den unmittelbar vorausgehenden Unterausdruck. Deswegen kontrolliert das + in ⌜vertip+t⌟ nur das ⌜p⌟ und nicht etwa ⌜vertip⌟ oder ⌜ip⌟. Wenn dem Quantor unmittelbar ein geklammerter Unterausdruck vorangeht, dann bezieht er sich natürlich auf diesen ganzen Unterausdruck, ungeachtet dessen, wie komplex er aufgebaut ist.

Zeichen

In der Informatik kann *Zeichen* je nach Zusammenhang ganz verschiedene Dinge bedeuten. Was für ein Zeichen ein bestimmtes Byte darstellt, ist weitgehend Interpretationssache. Ein Byte mit einem bestimmten Wert hat dagegen immer diesen Wert, ganz egal in welchem Kontext. Welches *Zeichen* damit dargestellt werden soll, hängt vom verwendeten Zeichensatz ab. Zwei Bytes mit den (dezimalen) Werten 64 und 53 repräsentieren die Zeichen »@« und »5«, wenn sie als ASCII-Zeichen aufgefasst werden, aber etwas völlig anderes, wenn als Zeichensatz EBCDIC verwendet wird (dann sind es das Leerzeichen und ein Kontrollzeichen).

Wenn die zwei Bytes aber in einem der für das Japanische gebräuchlichen Zeichensätze aufgefasst werden, bilden sie nur ein einziges Zeichen, 正. Um im EUC-JP-Zeichensatz das Zeichen 正 darzustellen, braucht es wiederum zwei ganz andere Bytes. Diese zwei Bytes, im *Latin-1*-Zeichensatz (ISO-8859-1) dargestellt, ergeben die zwei Zeichen »Àµ« oder das eine koreanische Zeichen 솹 im Unicode-Zeichensatz.[13] Entscheidend ist, dass es vom Standpunkt abhängt, wie viele Bytes ein Zeichen ausmachen (und so ein *Encoding* bilden). Beim Vergleichen von Zeichen muss sichergestellt sein, dass Ihr Standpunkt mit dem des verwendeten Werkzeugs oder der Programmiersprache übereinstimmt.

Bis vor Kurzem haben fast alle Regex-Werkzeuge ihre Daten als Ansammlung von 8-Bit-Bytes angesehen und sich nicht oder kaum um das Encoding gekümmert. Immer häufiger verwenden die Programme aber intern die eine oder andere Abart von Unicode (in Kapitel 3 finden Sie eine Einführung zu Unicode, ☞ 110). Auf solchen Systemen braucht sich der Benutzer normalerweise nicht um diese Belange zu kümmern, wenn der Regex-Teil richtig implementiert ist. Wenn das Wörtchen »wenn« nicht wär … gerade deshalb befasst sich Kapitel 3 genauer mit dieser Thematik.

Den Status quo verbessern

Wenn man sich ernsthaft mit ihnen befasst, sind reguläre Ausdrücke nicht besonders schwierig. Aber wenn man mit Benutzern von Regex-Programmen spricht, wird man oft Leute finden, die »ein bisschen was davon« verstehen, sich aber nicht sicher genug im Umgang damit fühlen, um reguläre Ausdrücke für komplexe Aufgaben einzusetzen oder sie in nicht so häufig benutzten Programmen zu gebrauchen.

13 Das Standardwerk über Multibyte-Zeichensätze ist *CJKV Information Processing* von Ken Lunde, auch bei O'Reilly erschienen. Das CJKV steht für *Chinesisch*, *Japanisch*, *Koreanisch* und *Vietnamesisch*, alles Sprachen, die Multibyte-Zeichensätze erfordern. Von Ken und von der Firma Adobe habe ich die speziellen Zeichensätze für dieses Buch erhalten.

Die traditionelle Dokumentation zu regulären Ausdrücken besteht oft nur aus einer kurzen Beschreibung von ein, zwei Metazeichen, gefolgt von einer Tabelle der restlichen. Man sieht Beispiele wie ⌜a*((ab)*|b*)⌟ und sinnlose Beispieldaten wie ›a•xxx•ce•xxxxxx•ci•xxx•d‹ ohne Bezug zur Praxis. Meist bleiben die subtilen, aber wichtigen Punkte unerwähnt; oder es wird behauptet, der Dialekt sei derselbe wie in einem anderen, bekannten Programm, ohne die Unterschiede anzugeben, die fast zwangsläufig vorhanden sind. Diesem Zustand muss abgeholfen werden.

Nun bilde ich mir nicht ein, dass dieses einführende Kapitel diese Lücke füllt. Es setzt nur das Fundament, auf dem der Rest des Buches aufbaut. Es mag ambitiös klingen, aber ich hoffe, dass das ganze Buch die Lücke füllt. Ich habe zur ersten Auflage viele erfreuliche Reaktionen erhalten und habe mir Mühe gegeben, diese zweite Auflage quantitativ und qualitativ zu verbessern.

Vielleicht gerade wegen der traditionell schlechten Dokumentation fühle ich mich verpflichtet, die Dinge bis ins Detail zu beschreiben. Ich möchte, dass Sie das volle Potenzial der regulären Ausdrücke ausschöpfen können und dass Sie reguläre Ausdrücke wirklich, *wirklich* verstehen.

Das ist gleichzeitig gut und schlecht.

Gut ist es, weil Sie dabei lernen, in regulären Ausdrücken zu *denken*. Sie werden sehen, auf welche Besonderheiten und Unterschiede man bei Programmen verschiedener Dialekte gefasst sein muss. Sie werden sich auch mit regulären Ausdrücken in einem faden, unvollständigen Dialekt ausdrücken können. Sie werden verstehen, warum ein bestimmter regulärer Ausdruck effizienter ist als ein anderer; Sie können Effizienz gegenüber Komplexität abwägen und Resultate vergleichen. Wenn Sie mit einem komplexen Ausdruck konfrontiert werden, können Sie nachvollziehen, wie dieser Ausdruck vom Programm verarbeitet wird, und Sie können den Ausdruck auf die gleiche Art analysieren. Kurz: Es wird Ihnen ein Leichtes sein, die volle Kraft der regulären Ausdrücke auszunutzen.

Leider ist der Lernaufwand mit dieser Methode nicht gerade gering, denn Sie müssen sich mit drei verschiedenen Bereichen befassen:

Wie reguläre Ausdrücke benutzt werden
: Die meisten Programme benutzen reguläre Ausdrücke in weitergehender Art, als wir das bei *egrep* gesehen haben. Bevor wir uns um die Details kümmern, müssen wir sehen, wie und wo reguläre Ausdrücke eingesetzt werden. Damit befasst sich das nächste Kapitel.

Features von regulären Ausdrücken
: Die Auswahl des richtigen Programmierwerkzeugs für ein bestimmtes Problem ist oft sehr wichtig und spart Zeit. Ich will mich daher in diesem Buch nicht auf ein einziges Programm beschränken. Nun haben aber verschiedene Programme (und oft sogar verschiedene Versionen desselben Programms) unterschiedliche Features und Metazeichen. Wir müssen uns daher einen Überblick verschaffen, bevor wir diese Features verwenden. Das passiert in Kapitel 3.

Wie reguläre Ausdrücke wirklich arbeiten
: Um aus nützlichen (und oft komplizierten) regulären Ausdrücken zu lernen, muss man verstehen, wie reguläre Ausdrücke von Programmen abgearbeitet werden. Wir werden sehen, dass es eine Rolle spielt, in welcher Reihenfolge bestimmte Metazeichen geprüft werden. Reguläre Ausdrücke können auf verschiedene Arten implementiert werden, und diese Arten tun manchmal verschiedene Dinge mit dem gleichen Ausdruck. Wir werden uns diesen dicken Brocken in den Kapiteln 4 und 5 vornehmen.

Dieser letzte Teil ist der wichtigste, aber auch der am schwierigsten zu behandelnde. Die Beschreibung ist leider manchmal etwas trocken, und man muss sich schon etwas in das Thema verbeißen, bevor wir zum spannenden Teil kommen – zum Lösen von Problemen aus der Praxis. Aber das Verstehen der Arbeitsweise der Regex-Maschine ist der Schlüssel zum *wirklichen Verstehen* von regulären Ausdrücken.

Sie werden einwenden, dass man nicht zu verstehen braucht, wie ein Auto funktioniert, um damit fahren zu können. Das mag stimmen, aber Autofahren ist eine schlechte Analogie zu regulären Ausdrücken. Mein Ziel ist es zu zeigen, wie man Probleme mit regulären Ausdrücken löst. Die bessere Analogie ist die zum Konstruieren eines Autos. Und dafür sollte man vielleicht schon wissen, wie ein Auto funktioniert.

Kapitel 2 entspricht etwa ein paar weiteren Fahrstunden. Kapitel 3 plaudert etwas aus der Geschichte des Autos und befasst sich mit Karosserie, Lack und Pflege von regulären Ausdrücken; und in Kapitel 4 schauen wir den Motor von regulären Ausdrücken an. Kapitel 5 zeigt einige weitere Beispiele, Kapitel 6 befasst sich mit der Leistungssteigerung von Motoren; die Kapitel danach gehen genauer auf bestimmte Marken und Modelle ein. Vor allem in den Kapiteln 4, 5 und 6 werden wir einige Zeit »unter dem Auto liegen«, da ist es vielleicht ganz angebracht, Arbeitskleidung zu tragen und Putzfäden zur Hand zu haben.

Zusammenfassung

Tabelle 1-3 auf der nächsten Seite fasst die *egrep*-Metazeichen zusammen, die wir in diesem Kapitel betrachtet haben.

Außerdem sind folgende Punkte wichtig:

- Nicht alle Programme namens *egrep* sind gleich. Die unterstützten Metazeichen und deren Bedeutung können verschieden sein – vgl. die lokale Dokumentation (☞ 23).
- Drei Gründe für das Setzen von (runden) Klammern sind: Limitieren des Bereichs von Alternationen (☞ 13), Gruppieren (☞ 14) und das »Sich-Merken« von Sub-Matches (☞ 21).
- Zeichenklassen sind speziell: Die Regeln, was ein Metazeichen ausmacht und was nicht, sind darin völlig anders (☞ 10).
- Alternationen und Zeichenklassen sind fundamental verschieden; nur in pathologisch einfachen Situationen kann deren *Wirkung* dieselbe sein (☞ 14).

Tabelle 1-3: Zusammenfassung: Metazeichen bei egrep

Metazeichen	Beschreibung	Passt auf
Dinge, die auf einzelne Zeichen passen		
.	*Punkt*	Irgendein Zeichen
[…]	*Zeichenklasse*	Eines der Zeichen aus der Liste
[^…]	*Negierte Zeichenklasse*	Irgendein Zeichen *nicht* aus der Liste
**** *Zeichen*	*Escape*	Wenn *Zeichen* ein Metazeichen ist oder wenn die Kombination mit Backslash kein neues Metazeichen ergibt: das Literal *Zeichen*
Dinge, die »zählen«; Quantoren		
?	*Fragezeichen*	Einmal erlaubt, aber optional
*****	*Stern*	Jede Anzahl erlaubt, auch null
+	*Plus*	Mindestens eins, mehr erlaubt
{*min, max***}**	*Expliziter Bereich* [a]	*min* gefordert, bis und mit *max* erlaubt
Dinge, die auf Positionen passen		
^	*Zirkumflex*	Passt auf den Zeilenanfang
$	*Dollar*	Passt auf das Zeilenende
\<	*Wortgrenze* [a]	Passt auf die Position am Wortanfang
\>	*Wortgrenze* [a]	Passt auf die Position am Wortende
Anderes		
|	*Alternation*	Passt auf mindestens eine der Alternativen
(…)	*Klammern*	Beschränkt den Geltungsbereich von Alternationen; gruppiert Dinge für nachfolgende Quantoren; »merkt sich Text« für Rückwärtsreferenzen
\1, **\2**, …	*Rückwärtsreferenzen* [a]	Passt auf Text, der im ersten, zweiten usw. Paar von Klammern vorkam

a wird nicht von allen *egrep*-Versionen unterstützt

- Eine negierte Zeichenklasse braucht noch immer ein Zeichen im Suchraum, damit sie passt; sie ist noch immer eine »positive Behauptung«. Die Aufzählung wird negiert, daher muss das passende Zeichen eines sein, das *nicht* in der Liste vorkommt (☞ 12).
- Die `-i`-Option bewirkt, dass Groß- und Kleinbuchstaben als identisch angesehen werden (☞ 15).
- Es gibt drei Arten von Escapes (Zeichen mit einem Backslash davor):
  1. Die Kombination von ⌜\⌟ mit einem Metazeichen ist eine Metasequenz, die das ursprüngliche Literal ergibt (zum Beispiel ergibt ⌜*⌟ einen literalen Stern).
  2. Die Kombination von ⌜\⌟ mit bestimmten Nicht-Metazeichen erzeugt ein Metazeichen, dessen Bedeutung vom verwendeten Programm abhängt (zum Beispiel bedeutet ⌜\<⌟ oft »Wortanfang«).

3. Die Kombination von ⌜\⌟ mit einem anderen Zeichen ergibt einfach das Zeichen selbst (d. h. der Backslash wird ignoriert).

Noch einmal: Innerhalb einer Zeichenklasse ist ein Backslash bei den meisten *egrep*-Versionen ein ganz normales Zeichen und leitet *kein* Escape ein.

- Dinge, die mit einem Fragezeichen oder dem Stern quantifiziert werden, brauchen gar kein Zeichen im Suchtext, damit die Behauptung erfüllt ist. Sie passen *immer*, auch wenn sie auf »kein Zeichen« passen (☞ 17).

Persönliche Einsprengsel

Das Problem der verdoppelten Wörter zu Beginn des Kapitels mag einschüchtern, aber reguläre Ausdrücke sind so mächtig, dass wir einen großen Teil des Problems mit einem so limitierten Werkzeug wie *egrep* lösen konnten – schon hier im ersten Kapitel. Ich würde an dieser Stelle ganz gerne ein paar verblüffende oder lustige Beispiele anbringen, gerade weil ich so ausdrücklich auf die Tatsache hingewiesen habe, dass noch viel mehr dahintersteckt. Ich befürchte, dass jemand, der all die Warnungen gelesen hat, aber noch kaum mit regulären Ausdrücken vertraut ist, sich langsam fragt: »Was soll das alles?«

Einmal haben meine Brüder versucht, ein paar Freunden *Schaffkopf* beizubringen, ein Kartenspiel, das in unserer Familie seit Generationen gespielt wird. Schaffkopf ist viel spannender, als es auf den ersten Blick scheint, aber die Regeln sind nicht ganz leicht zu erlernen. Nach etwa einer halben Stunde meinte meine Schwägerin Liz, normalerweise eine sehr geduldige Person: »Warum spielen wir nicht einfach Rummy[14]?« Trotzdem wurde weitergespielt, und zwar bis tief in die Nacht hinein. Nachdem die erste, schwierige Lernphase überwunden war, genügte die aufkommende Begeisterung, um bei der Sache zu bleiben. Meine Brüder wussten, dass sich der Lernaufwand lohnt, aber es brauchte seine Zeit, um Liz und die anderen Schaffkopf-Neulinge auf den Geschmack zu bringen.

Es mag seine Zeit dauern, um sich in der Welt der regulären Ausdrücke zu akklimatisieren. Bevor man die Begeisterung spürt, kann einem das alles etwas zu akademisch vorkommen. Falls dem so ist, hoffe ich sehr, dass *Sie* dem Verlangen, »Rummy zu spielen«, widerstehen können. Wenn Sie mit regulären Ausdrücken spielen können und wenn Sie deren Kraft ausnutzen können, dann wird Ihnen der anfängliche Lernaufwand vernachlässigbar erscheinen.

14 Eine sehr einfache Variante von Poker oder Rommé.