Leibniz Rechenzentrum

der Bayerischen Akademie der Wissenschaften

Reguläre Sprachen, reguläre Ausdrücke

Helmut Richter

2004-07-08

Inhalt:

Über diesen Artikel . . . 1
    Muss man das alles lesen? . . . 1
Was sind reguläre Sprachen und reguläre Ausdrücke? . . . 2
Reguläre Ausdrücke in der Programmierung . . . 3
    Metazeichen und andere Zeichen . . . 4
    Die verschiedenen Notationen für reguläre Ausdrücke . . . 5
    Übersicht über die Notationen . . . 7
    Welches Programm akzeptiert was? . . . 9
        Shell-Namensmuster . . . 9
        Eigentliche reguläre Ausdrücke . . . 10
    Maskierung und Mengen von Einzelzeichen . . . 11
    Mustersuche . . . 13
    Gier (*greed*) . . . 15
    Rückverweise . . . 17
    Beispiele . . . 18
        Wortsuche: Schreibungsvarianten, Ausschluss von Zufallstreffern . . . 18
        Texte, die an ihrem Anfang und ihrem Ende erkannt werden . . . 18
        Einfach maskierter Text . . . 19
        Mehrfach maskierter Text . . . 20
        Gregorianische Kalenderdaten . . . 20
Theorie . . . 21
    Endliche Automaten . . . 22
        Vervollständigung und Komplementierung endlicher Automaten . . . 25
        Kombination mehrerer endlicher Automaten zu einem einzigen . . . 27
        Nichtdeterministische endliche Automaten . . . 29
        Minimierung endlicher Automaten . . . 30
    Reguläre Ausdrücke . . . 30
        Reguläre Ausdrücke für endliche Automaten . . . 33
    Beispiele . . . 35
        Mehrfach maskierter Text (zweiter Versuch) . . . 35
        Durch drei teilbare Dezimalzahlen . . . 37
        Durch dreizehn teilbare Dezimalzahlen . . . 39
        Weder Fisch noch Fleisch . . . 41
    Weitere theoretische Betrachtungen . . . 42

Über diesen Artikel

Wer sich in die Benutzung von Programmierwerkzeugen wie der Scriptsprache `perl` oder der auf Unix-Systemen vorhandenen Programme `grep` oder `awk` einarbeiten will, sieht sich mit dem Begriff "regulärer Ausdruck" konfrontiert, der dem Normalbürger in der Regel unbekannt ist. Informatiker, also nicht ganz so normale Bürger, kennen diesen Begriff aus der Theorie der formalen Sprachen im Zusammenhang mit "regulären Sprachen" und "endlichen Automaten". Was verbirgt sich nun dahinter?

Dieser Artikel hat das Ziel, einen Leitfaden durch die verwirrende und widersprüchlich gebrauchte Begriffswelt zu bieten.

Der erste, mehr programmiertechnische Teil geht hinsichtlich der Notationen in den verschiedenen Programmen stark ins Detail und unterscheidet sich so von dem eher feuilletonistischen zweiten Teil. Wichtig war dabei, Ähnlichkeiten und Unterschiede zwischen ähnlich aussehenden Konstrukten herauszuarbeiten. Um also *ein* bestimmtes Werkzeug benutzen zu können, ist das möglicherweise der falsche Ansatz. Vielmehr sollen dem Leser Orientierungspunkte an die Hand gegeben werden, mit denen er sich in der verwirrenden Vielfalt zurechtfinden kann.

Der mehr theoretische Teil (der erst im Entstehen ist) soll nicht Bücher oder Vorlesungen über formale Sprachen ersetzen, sondern möglichst schlaglichtartig beleuchten, worauf es in dieser Theorie ankommt und was ihre besondere Schönheit ausmacht. Ganz entgegen den Gepflogenheiten der Mathematiker wird dabei fast ganz und gar auf formale Definitionen, Sätze und Beweise verzichtet. Damit dürften diese Teile für ein breiteres Publikum lesbar werden.

Muss man das alles lesen?

Nein. Dieser Artikel versteht sich eher als eine Art Enzyklopädie, wo man immer nur das liest, was einen gerade interessiert. Die verschiedenen Abschnitte bauen auch nicht alle aufeinander auf, so dass eine selektive Lektüre durchaus möglich ist. Hier sind ein paar mögliche Pfade durch diesen Artikel für Leser mit unterschiedlichen Interessen:

- **Neulinge**: Künftige Benutzer von Programmen, die mit regulären Ausdrücken umgehen (wie Unix-Shells, komfortable Editoren wie `emacs` oder `vi`, oder `perl`, `awk`, `grep`):

  Lesen Sie alles bis zur Tabelle der Notationen gründlich. Bei der Tabelle brauchen Sie die letzten paar Zeilen nicht unbedingt zu kennen, und was hinter den "Rückverweisen" steckt, heben Sie sich besser für später auf, wenn Sie den Rest sicher beherrschen. Die anschließende Tabelle der Programme, die mit regulären Ausdrücken arbeiten, muss man nicht lernen; man sollte sie aber überflogen haben und sich die Beschreibung des einen oder anderen Programms ansehen. Den Rest des Artikels, insbesondere die Theorie-Teile, können Sie getrost erst einmal weglassen.

- **Alte Hasen**: Erfahrene Benutzer solcher Programme, die ihre Kenntnisse auffrischen oder ordnen und vielleicht die eine oder andere Einsatzmöglichkeit für reguläre Ausdrücke neu entdecken wollen:

  Sie haben ungefähr dasselbe Leseprogramm wie die Neulinge, nur dass Sie vieles überfliegen können, wenn Sie sicher sind, dass Sie es kennen. Vor allem die Abschnitte nach der Tabelle der Notationen, bevor der Theorie-Teil beginnt, sind für Sie gedacht.

- **Gurus**: Leute, die alle Möglichkeiten ausreizen, die ein System bietet und nicht einmal vor dem Programm `lex` zurückschrecken:

  Wer reguläre Ausdrücke profimäßig einsetzen will, sollte wenigstens ein bisschen etwas von der Theorie kennen. Die Beurteilung, ob reguläre Ausdrücke für eine bestimmte Anwendung geeignet sind, ergibt sich damit oft viel einfacher. Auch lassen sich *manche* regulären Ausdrücke besonders leicht konstruieren, wenn man einen endlichen Automaten vor Augen hat. Also: von der Theorie wenigstens die ersten beiden Abschnitte lesen. Die Kenntnis des Alte-Hasen-Stoffes wird stillschweigend vorausgesetzt.

- **Theoretiker**: Menschen, die Freude an der einfachen Darstellung komplizierter Dinge haben und **Bildungshungrige**: Neugierige Menschen, die sich immer dafür interessieren, "was dahintersteckt":

  Die Theorie-Kapitel sollten auch für sich alleine lesbar sein. Alle Theorie ist nie ganz einfach; man muss schon ein wenig mitdenken. Um die Abschreckung minimal zu halten, ist die strenge Darstellung im mathematischen Jargon auf ein Minimum beschränkt worden. Die Nichtmathematiker unter den Lesern werden sie nicht vermissen, und die Mathematiker werden keine Mühe haben, die hier eher blumig dargestellten Ideen in mathematisch sauberer Darstellung selbst zu formulieren.

Was sind reguläre Sprachen und reguläre Ausdrücke?

Egal, ob Theorie oder Programmierpraxis, wir haben es in jedem Fall mit formalen *Sprachen* zu tun, wobei hier unter Sprache einfach eine Menge von *Zeichenreihen* (auch *Strings* genannt) verstanden wird, die ihrerseits wieder aus *Zeichen* bestehen, die einem *Alphabet* angehören. Die Zeichenreihen, die zu einer Sprache gehören, werden oft *Wörter*, manchmal (hier aber nicht) auch *Sätze* der Sprache genannt; das hat nichts damit zu tun, was diese beiden Begriffe im Zusammenhang mit natürlichen Sprachen bedeuten.

Formal heißen die Sprachen, um sie von natürlichen Sprachen zu unterscheiden. Außerdem geht es hier ausschließlich um *syntaktische* Fragen, also Fragen, die sich auf die Gestalt der Zeichenreihen beziehen, wie "Gehört diese Zeichenreihe zur Sprache?" oder "Kommt in dieser Zeichenreihe ein Muster vor, auf das diese Beschreibung passt?". *Semantische* Fragestellungen, also solche, die sich auf die Bedeutung der Zeichenreihen beziehen, spielen hier keine Rolle.

Nehmen wir als Beispiel diejenigen Zeichenreihen, die ganze oder Dezimalzahlen in der üblichen deutschen Notation darstellen. Verbal ließen sich diese etwa so beschreiben:

- Am Anfang kann, muss aber nicht, ein Vorzeichen stehen.
- Danach kommt mindestens eine Ziffer, vielleicht auch mehrere.
- Danach kann ein Komma stehen; dann muss aber mindestens noch eine Ziffer folgen, vielleicht auch mehrere.

In der Programmierung kann man das mit einem *regulären Ausdruck* so notieren:

```
[+-]?[0-9]+(,[0-9]+)?
```

Die Einzelheiten, was das bedeutet, werden später [S.4] erläutert, z.B. das Fragezeichen, das einen Teilausdruck markiert, der vorhanden sein kann, aber nicht muss.

Eine Sache sorgt anfangs immer wieder für Verwirrung, nämlich die Endlichkeit oder Unendlichkeit der ganzen Angelegenheit. An dem Beispiel eben kann man sich das klarmachen:

- Der Zeichenvorrat (das Alphabet), aus dem die Zeichenreihen gebildet werden, ist immer nur endlich groß.
- Der reguläre Ausdruck selbst ist immer nur endlich lang.
- Auch ist jede einzelne Zeichenreihe, die mit dem regulären Ausdruck beschrieben wird, nur endlich lang. Die Länge solcher Zeichenreihen kann aber, wie hier, unbeschränkt sein, d.h. jede vorgegebene Länge wird dann von mindestens einer der (selbst nur endlich langen) Zeichenreihen überschritten.
- Es kann, wie hier, unendlich viele Zeichenreihen geben, die mit dem regulären Ausdruck beschrieben werden. Wegen der Endlichkeit des Alphabets ist das genau dann der Fall, wenn die Längen unbeschränkt sind.

Also: Reguläre Ausdrücke dienen dazu, Sprachen, d.h. Mengen von Zeichenreihen zu beschreiben. Je nachdem, wie dabei die regulären Ausdrücke definiert sind, kann man damit eine bestimmte Sprache beschreiben oder auch nicht. Das Beispiel oben lässt sich mit allen Varianten regulärer Ausdrücke beschreiben, wenn auch die verschiedenen Varianten verschieden aussehen mögen, aber ansonsten ist die Verwirrung groß. Sie lässt sich so zusammenfassen:

- Die Theoretiker sind sich einig. Selbst wenn sie verschiedene Notationen verwenden, so können damit immer genau dieselben Sprachen, eben die *regulären Sprachen*, beschrieben werden.
- In der Programmierung gibt es verschiedene Notationen, mit denen verschiedene Mengen von Sprachen beschrieben werden können. Beileibe nicht alle davon reichen aus, um damit jede reguläre Sprache beschreiben zu können. Umgekehrt gehen viele weit über die regulären (sogar über die kontextfreien, aber das wird hier nicht erklärt) Sprachen hinaus.

Wie lässt sich unter diesen Umständen der Gebrauch, oder besser Missbrauch, des Begriffs "regulärer Ausdruck" verstehen? Das liegt zunächst einmal an der Geschichte: die regulären Ausdrücke der Theorie waren der Ausgangspunkt für die regulären Ausdrücke in der Programmierung, nur wurden anfangs seltener benötigte, schwerer zu implementierende Eigenheiten weggelassen und später komplexere hinzugefügt. Bis heute ist aber das wichtigste Einsatzgebiet der regulären Ausdrücke in der Programmierung die Erkennung von solchen Mustern in Zeichenreihen geblieben, die in der Tat reguläre Sprachen bilden. Insofern ist die Ähnlichkeit eben doch nicht zufällig.

Reguläre Ausdrücke in der Programmierung

Eine Warnung vorweg: reguläre Ausdrücke werden in der Programmierung nach *deutlich* verschiedenen Konventionen notiert. Diese lassen sich in mehrere verschiedene "Schulen" oder "Traditionen" einteilen; das wird Gegenstand des übernächsten Abschnittes sein. Vorher wird an einem Beispiel gezeigt, wie reguläre Ausdrücke *zum Beispiel* aussehen können. Damit wird das allen Notationen gemeinsame Prinzip erklärt und nebenbei das Material für die anschließende Diskussion der Gemeinsamkeiten und Unterschiede der Notationen bereitgestellt.

Metazeichen und andere Zeichen

Um zu verstehen, wie in der Programmierung reguläre Ausdrücke notiert werden, wenden wir uns noch einmal dem Beispiel [S.2] vom Anfang dieses Artikels zu:

```
[+-]?[0-9]+(,[0-9]+)?
```

Dabei fällt auf, dass es hier offensichtlich zwei Arten von Zeichen gibt:

- gewöhnliche Zeichen (der Fachausdruck heißt *terminale Zeichen*), die sich selbst bedeuten, z.B. das Komma, welches nur bedeutet, dass dort ein Komma steht, und
- *Metazeichen* (oben in blau dargestellt), die für die Bedeutung des regulären Ausdrucks selbst eine Bedeutung haben, hier:
  - die eckigen Klammern, die eine Liste von Zeichen umschließen, von denen genau *eines* an einer bestimmten Stelle vorkommen soll,
  - das Minuszeichen (aber nur wenn es innerhalb einer eckigen Klammer und dort nicht am Rand steht), das einen Bereich von Zeichen markiert, die dann nicht alle aufgelistet zu werden brauchen,
  - das Fragezeichen, das anzeigt, dass das letzte Zeichen oder der Inhalt der unmittelbar voranstehenden runden Klammer optional ist, d.h. einmal vorkommen kann, aber nicht muss,
  - das Pluszeichen, das anzeigt, dass das letzte Zeichen oder der Inhalt der unmittelbar voranstehenden runden Klammer mindestens einmal vorkommen muss aber auch mehrmals vorkommen darf,
  - die runde Klammer, die dafür sorgt, dass sich ein dahinter stehendes Frage- oder Pluszeichen auf einen längeren Text und nicht nur auf das letzte Zeichen bezieht, analog der Verwendung von Klammern in der Mathematik,

  sowie einige Metazeichen, die in diesem kurzen Beispiel nicht vorkommen:

  - der Stern, der anzeigt, dass das letzte Zeichen oder der Inhalt der unmittelbar voranstehenden runden Klammer ein- oder mehrmals vorkommen kann, aber nicht muss,
  - der Punkt, der für genau *ein* ganz beliebiges Zeichen steht,
  - der senkrechte Strich, der Alternativen voneinander trennt, von denen mindestens eine zutreffen muss, sowie
  - weitere, exotischere Zeichen, die weiter unten in einer Tabelle [S.7] stehen, hier aber nur verwirren würden.

  Bis auf das Minuszeichen sind diese Zeichen nur dann Metazeichen, wenn sie außerhalb von eckigen Klammern stehen. Das kann man auch dazu verwenden, beispielsweise eine Klammer als terminales Zeichen zu notieren: man schreibt dann einfach `[(]`, was, soviel heißt wie "eines der folgenden Zeichen: `(`". Alternativ kann man auch vor ein Metazeichen einen inversen Schrägstrich setzen, um es auf diese Weise zu *maskieren*, d.h. an dieser Stelle nur als terminales Zeichen zu verwenden - allerdings werden wir noch sehen, dass dieser inverse Schrägstrich

manchmal auch *gerade* Metazeichen bezeichnet, während die Maskierung mittels eckiger Klammern solche Tücken nicht hat.

Wer den Umgang mit formalen Notationen gewohnt ist, etwa aus der Mathematik, wird keine Schwierigkeiten haben, den Ausdruck oben in seine Bestandteile zu zerlegen und entsprechend zu verstehen. Für die anderen wird hier noch detaillierter dargelegt, wie die ineinander verschachtelten Teilausdrücke zusammenwirken:

- Das letzte Pluszeichen verlangt, dass die davorstehende Ziffer (notiert als `[0-9]`) mindestens einmal vorkommen *muss*; ebenso verlangt das Komma, dass das Komma genau einmal vorkommen *muss*.
- Das steht aber in einer runden Klammer, die mit einem Fragezeichen versehen ist, so dass doch alles *optional* ist. Wie passt das zusammen?

Man löst solche Fragen, indem man den Ausdruck strikt in seine Bestandteile gliedert. Das Fragezeichen sagt aus, dass der Teilausdruck `,[0-9]+` optional ist. Ist im zu untersuchenden String ein Teil vorhanden, der diesem Teilausdruck entspricht, ist es recht, wenn nicht, auch. *Wenn* ein solcher Teilstring vorhanden ist - und nur dann - interessieren wir uns dafür, wie er aussieht: und dann gibt es Zeichen, die er *zwingend* enthalten muss.

Jetzt brauchen wir nur noch die Vorrangregeln zu kennen und schon können wir nach Herzenslust reguläre Ausdrücke schreiben. Die Vorrangregeln sind:

- 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; eigentlich nicht recht einzusehen: man würde erwarten, dass `**`, `*+`, `*?`, `+*`, `+?`, `?*` und `?+` dasselbe bedeuten wie `*`, `++` dasselbe wie `+` und `??` dasselbe wie `?`. In einer wichtigen Erweiterung der regulären Ausdrücke kommen `*?` und `+?` vor; das wird aber erst dort [S.15] erklärt.

Die verschiedenen Notationen für reguläre Ausdrücke

Bei den regulären Ausdrücken erleben wir das Dilemma jeder Normung: wird etwas genormt, bevor es in allgemeinem Gebrauch ist, so ist die Norm weltfremd und wird nicht akzeptiert, ist es aber in allgemeinem Gebrauch, so ist es zu spät für die Normung. Bei Software-Schnittstellen, wie hier eben bei den regulären Ausdrücken, kann eine spätere Normung einen davon abweichenden früher üblichen Gebrauch nachträglich entwerten, was man natürlich vermeiden will. Trotzdem hat man sich bemüht, die unterschiedlichen Notationen für reguläre Ausdrücke zu vereinheitlichen, so dass es nicht beliebig viele, sondern im wesentlichen nur drei verschiedene Notationen gibt. In diesem Abschnitt werden diese drei in ihren Grundzügen dargestellt, im nächsten [S.7] in etwas mehr Detail, und dann folgt noch eine Übersicht [S.9] , welche diese Notationen von welchem Programm akzeptiert wird.

Das Hauptproblem bei der Vereinheitlichung der Notation für reguläre Ausdrücke sind die Metazeichen: wird die Notation um zusätzliche Operatoren erweitert, so werden Zeichen, die bisher terminale Zeichen waren, plötzlich zu Metazeichen, ändern also in *bestehenden* Programmen ihre Bedeutung. Beispielsweise ist das Fragezeichen zur Bezeichnung optionaler Teile erst später dazugekommen, in früheren Programmen gab es das nicht. Mit dieser Neuerung ändert aber jeder reguläre Ausdruck, der zufällig ein Fragezeichen enthält, seine Bedeutung. Man hat nun, je nach Anwendung, drei verschiedene Wege beschritten:

- Die Möglichkeiten, gegenüber der Shell mehrere ähnlich benannte Dateien mit einem einzigen Ausdruck zu bezeichnen, der Sterne und Fragezeichen - beide in anderer Bedeutung als im vorangegangenen Abschnitt! - enthält, sind voll beibehalten worden. Das ist eine Notation, die sich grundsätzlich von den beiden folgenden unterscheidet [S.6] . Wir wollen sie, da mit ihr nur wenige reguläre Sprachen darstellbar sind, auch nicht als reguläre Ausdrücke bezeichnen, sondern *(Shell-)Namensmuster* (Shell Patterns) nennen. Die Metazeichen in dieser Notation, die jeweils in der Art von Jokern im Rommé- oder Canasta-Spiel für ganz Verschiedenes stehen können, werden dabei oft als *Wildcards* bezeichnet, was eben "Joker" bedeutet.

- Für Programme, die mit diesen beschränkten Möglichkeiten nicht auskommen, z.B. `grep`, hat sich früh eine andere Notation entwickelt. Als Erweiterungen dazugetreten sind, insbesondere die runden Klammern zur Bezeichnung von Teilausdrücken, ist man für die neuen Metazeichen von der generellen Regel abgewichen, diese Zeichen dann zu kennzeichnen, wenn sie als terminale Zeichen gebraucht werden, stattdessen kennzeichnet man ihren Gebrauch als Metazeichen: die runden Klammern haben hier die Gestalt `\(` und `\)`. Das hat den Vorteil, dass reguläre Ausdrücke, die vor der Einführung der Klammerung schon zufällig runde Klammern als terminale Zeichen enthalten haben, in ihrer Bedeutung unverändert geblieben sind. Dafür ist die entstandene Notation insgesamt sehr widersprüchlich: manche Zeichen bekommen den inversen Schrägstrich, weil sie Metazeichen sind, andere, zum Beispiel der Stern, damit sie gerade *nicht* als Metazeichen interpretiert werden. Diese Notation heißt *Einfache Reguläre Ausdrücke* oder *Basic Regular Expressions*, abgekürzt *BRE*.

- Um diese Inkonsistenz zu beseitigen, gibt es eine neuere Notation mit mehr Metazeichen, die alle nicht mehr besonders gekennzeichnet sind, sondern wieder, wie bei den Shell-Namensmustern oder den BREs *vor* der Einführung von Erweiterungen, nur dann durch einen inversen Schrägstrich davor oder eckige Klammern herum maskiert werden, wenn sie als terminale Zeichen verwendet werden. Diese Notation heißt *Erweiterte Reguläre Ausdrücke* oder *Extended Regular Expressions*, abgekürzt *ERE*.

Die im vorangegangenen [S.4] Abschnitt verwendete Notation war, wie man an der Form der runden Klammern (nämlich ohne Verzierung durch einen inversen Schrägstrich) leicht sehen kann, die der Erweiterten Regulären Ausdrücke. In einer BRE hätte es übrigens weder das Pluszeichen noch das Fragezeichen gegeben - diese Zeichen hat man trotz ihrer Nützlichkeit nicht zu den BREs dazugenommen, weil sie dann nach den BRE-Regeln mit inversem Schrägstrich geschrieben werden müssten, was sehr unübersichtlich wäre.

Während die Notationen ERE und BRE relativ ähnlich sind (weswegen man wohl immer in den Manualen der einzelnen Programme nachschauen muss, welche davon verwendet wird und mit welchen Einschränkungen oder Erweiterungen), unterscheiden sich Shell-Namensmuster von eigentlichen regulären Ausdrücken ganz wesentlich in der Art der Darstellung: Bei den ersteren gibt es ein Metazeichen, den Stern, für einen beliebigen (evtl. auch leeren) String, während es bei ERE und BRE ein Metazeichen mit einer solchen Bedeutung nicht gibt. Der Stern bei ERE und BRE hat seine Bedeutung nämlich nur im Zusammenhang mit dem unmittelbar vorhergehenden Zeichen oder

geklammerten Teilausdruck und zeigt *dessen* Wiederholung an. Um einen beliebigen String als ERE oder BRE darzustellen, wird der Punkt für ein beliebiges Zeichen mit dem Stern für dessen Wiederholung kombiniert.

Beispiel: `[ab]*` als ERE oder BRE steht für eine beliebige Folge von `a` und `b`, z.B. `aabbaba`; diese Menge erlaubter Strings ist nicht als Shell-Namensmuster darstellbar. `[ab]*` als Shell-Namensmuster steht für einen beliebigen String, der mit `a` oder `b` beginnt; diese Menge erlaubter Strings würde als ERE oder BRE mit dem Ausdruck `[ab].*` dargestellt.

Außerdem werden auch sonst in Shell-Namensmustern und eigentlichen regulären Ausdrücken unterschiedliche Metazeichen verwendet. Unten in der Tabelle [S.7] sind die Unterschiede zusammengestellt, wobei der von dieser wichtigen Unterscheidung betroffene Bereich durch gelbe Hintergrundfarbe besonders herausgestellt ist. Man sieht dabei auch, dass sich im Bereich der Dinge, die durch Shell-Namensmuster überhaupt darstellbar sind, EREs nicht von BREs unterscheiden.

Glücklicherweise treten Notationen in der Art von Shell-Namensmustern und eigentliche reguläre Ausdrücke nie miteinander vermengt auf. Man kann sich also darauf verlassen, dass eine Anwendung konsequent einer der beiden Konventionen folgt: keine Anwendung wird etwa den Stern im Sinne von ERE und BRE benutzen, aber ein einzelnes Zeichen mit einem Fragezeichen bezeichnen.

Übersicht über die Notationen

In der nachstehenden Tabelle sind die verbreitetsten Notationen für reguläre Sprachen zusammengestellt, und zwar:

- **Theorie:** die in der Theorie [S.30] der formalen Sprachen übliche Notation
- **ERE:** Extended Regular Expression (=erweiterter regulärer Ausdruck) gemäß *X/Open CAE Specification "System Interface Definitions", Issue 4*
- **BRE:** Basic Regular Expression (=einfacher regulärer Ausdruck) gemäß dem selben Dokument
- **Shell:** Wildcards (=Joker) nach den Konventionen von Unix-Shells
- **SQL:** Wildcards (=Joker) nach den Konventionen von SQL-Datenbankabfragen, die ansonsten hier nicht weiter diskutiert werden.

Nicht in dieser Tabelle enthalten sind:

- Konventionen zur Maskierung von Zeichen, d.h. zum Kennzeichnen, dass ein Zeichen mit einer Sonderbedeutung diese an einer Stelle *nicht* haben soll, sowie
- die genaue Syntax für die Inhalte der eckigen Klammern (Menge einzeln aufgezählter Zeichen oder deren Komplement).

Auf diese Dinge wird später [S.11] eingegangen.

<table>
<tr><th rowspan="2"></th><th rowspan="2">Theorie</th><th rowspan="2">ERE</th><th rowspan="2">BRE</th><th colspan="2">Wildcard</th></tr>
<tr><th>Shell</th><th>SQL</th></tr>
<tr><td>leere Menge</td><td>Ø</td><td>(fehlt)</td><td>(fehlt)</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>einzelnes Zeichen</td><td>a</td><td>a</td><td>a</td><td>a</td><td>a</td></tr>
<tr><td>Konkatenation (Hintereinanderschreiben)</td><td>xy</td><td>xy</td><td>xy</td><td>xy</td><td>xy</td></tr>
<tr><td>Alternative, Vereinigung</td><td>x | y</td><td>x | y</td><td>(fehlt)</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>Klammer</td><td>(. . .)</td><td>(. . .)</td><td>\(. . . \)</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>Rückverweis [S.17] auf Inhalt der i-ten Klammer</td><td>(fehlt)</td><td>(fehlt)</td><td>\i</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>einzelnes Zeichen [S.11] aus endlicher Menge</td><td>a | b | c</td><td>[abc]</td><td>[abc]</td><td>[abc]</td><td>(fehlt)</td></tr>
<tr><td>einzelnes Zeichen [S.11] nicht aus endlicher Menge</td><td>(fehlt)</td><td>[^abc]</td><td>[^abc]</td><td>[!abc]</td><td>(fehlt)</td></tr>
<tr><td>beliebiges einzelnes Zeichen</td><td>(fehlt)</td><td>.</td><td>.</td><td>?</td><td>_</td></tr>
<tr><td>beliebiger Text</td><td>(fehlt)</td><td>.*</td><td>.*</td><td>*</td><td>%</td></tr>
<tr><td>Kleenescher Abschluss (0..?-fache Wiederholung)</td><td>x*</td><td>x*</td><td>x*</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>optionaler Teilausdruck (0..1-fache Wiederholung)</td><td>(| x)</td><td>x?</td><td>(fehlt)</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>1..?-fache Wiederholung</td><td>xx*</td><td>x+</td><td>xx*</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>m-fache Wiederholung</td><td>(fehlt)</td><td>x{m}</td><td>x\{m\}</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>m..n-fache Wiederholung</td><td>(fehlt)</td><td>x{m,n}</td><td>x\{m,n\}</td><td>(fehlt)</td><td>(fehlt)</td></tr>
<tr><td>m..?-fache Wiederholung</td><td>(fehlt)</td><td>x{m,}</td><td>x\{m,\}</td><td>(fehlt)</td><td>(fehlt)</td></tr>
</table>

Erläuterungen zur Tabelle:

- Der Begriff "Wiederholung" wird hier so verwendet, dass damit nicht Wiederholungen *zusätzlich* zu einem Original gemeint sind, d.h. eine 5-fache Wiederholung von `a` ist ein String von fünf, nicht von sechs `a`.
- Der gelb unterlegte Teil der Tabelle zeigt den Bereich an, in welchem sich die Unterschiede [S.6] zwischen Shell-Namensmustern einerseits und ERE/BRE andererseits befinden.
- Die zur Darstellung aller regulären Sprachen notwendigen Sprachmittel sind hellgrün unterlegt (außer in den gelben Bereichen). Man sieht daraus, dass nur ERE die Bezeichnung "reguläre Ausdrücke" verdienen, während mit allen anderen Notationen nur ein Teil der regulären Sprachen dargestellt werden kann. Das rot unterlegte Feld deutet an, dass mit dieser Notation die Menge der regulären Sprachen überschritten wird.
- Die hellblau unterlegten Teile sind Abkürzungen für solche Ausdrücke, die zwar nur geringes theoretisches Interesse wecken können, aber in der Praxis oft vorkommen, z.B. `a{2,4}` als Abkürzung für `aa|aaa|aaaa`.

Welches Programm akzeptiert was?

Shell-Namensmuster

Die Shell (damit ist hier und im Folgenden `sh`, `ksh`, `bash` gemeint, für `tcsh` gelten zum Teil in Details abweichende Regeln) verwendet die Namensmuster-Notation (das ist die, wo der Punkt keine spezielle Bedeutung hat) an folgenden Stellen:

- Vor der Abarbeitung jedes Kommandos werden alle Wörter der Kommandozeile (Kommandoname, Parameter und Operanden) durch eine Liste der Dateipfade ersetzt, die auf das Muster passen; wird keine einzige Datei gefunden, so bleibt das Muster erhalten. Das gilt beileibe nicht nur für die Parameter, die wirklich Dateien bezeichnen sollen; die Shell ist ja auch gar nicht in der Lage, das zu entscheiden. Bei dieser Ersetzung gibt es die Sonderregel, dass ein Punkt, der am Anfang oder nach einem Schrägstrich steht, sowie jeder Schrägstrich explizit angegeben werden muss und nicht durch `*`, `?` oder einen Ausdruck in eckigen Klammern mit abgedeckt ist.

- Die Alternativen in der `case`-Anweisungen sind Listen von Namensmustern in Shell-Notation. Die Sonderregel für Punkte und Schrägstriche gilt hier nicht.

- Man kann bei der Substitution von Variablennamen durch ihre Werte das Ergebnis dahingehend abändern, dass ein Präfix oder Suffix des Variablenwertes abgeschnitten wird, das auf ein gegebenes Muster passt; solche Muster werden dann ebenfalls in Shell-Notation ohne die Sonderregel für Dateipfade geschrieben. Will man beispielsweise im Shell-Prompt nur den Namen, nicht aber den ganzen Pfad des aktuellen Verzeichnisses ausgeben, so schreibt man `${PWD##*/}` und meint damit den Wert `${PWD}`, verkürzt um das längste Präfix, das auf das Muster `*/` passt. Es gibt dabei die vier Möglichkeiten:

  % : verkürze um das kürzeste Suffix, das auf das Muster passt
  %% : verkürze um das längste Suffix, das auf das Muster passt
  # : verkürze um das kürzeste Präfix, das auf das Muster passt
  ## : verkürze um das längste Präfix, das auf das Muster passt

Außerdem haben die Programme `find`, `pax` und `cpio` Argumente, bei denen das Programm eine Suche nach Dateien entsprechend einem Muster durchführt, wobei das Muster in Shell-Notation geschrieben wird. Bei `pax` ist ausdrücklich spezifiziert, dass die Sonderregel für Dateipfade wie bei der Dateisubstitution ebenfalls gilt. Bei diesen Programmen muss man die Shell daran hindern, eine solche Substitution schon vorab durchzuführen:

```
find . -name 'abc*' -print
```

sucht nach allen Dateien, deren name mit `abc` beginnt und könnte beispielsweise `./abc1`, `./abc2` und `./sub/abc` als Ergebnis liefern. Demgegenüber würde

```
find . -name abc* -print
```

schon von der Shell durch

```
find . -name abc1 abc2 -print
```

ersetzt, was erstens syntaktisch falsch ist und zweitens, wenn es zufällig richtig wäre, nach etwas ganz anderem suchen würde.

Eigentliche reguläre Ausdrücke

Die meisten Programme, die reguläre Ausdrücke verarbeiten, richten sich nach den Regeln von Extended oder Basic Regular Expressions. Leider haben fast alle diese Programme Besonderheiten, in denen sie von der Norm abweichen. In der folgenden Tabelle sind nur die wichtigsten davon zusammengestellt; keinesfalls ersetzt die Tabelle die Lektüre des jeweiligen Manuals.

Programm	**Parameter**	**ERE/BRE/ Shell-Namensmuster**	**Besonderheiten**
`awk`		E	2
`cpio`	*Operanden*	Shell-Namensmuster	
`csplit`	*Operanden*	E	
`ed`		B	1
`emacs`, `xemacs`		B/E	2, 3, 4, 8
`ex`		B	1, 4, 5
`expr`	*Operanden*	B	6
`find`	`-name`	Shell-Namensmuster	
`grep` *ohne* `-E`	`-e`, `-f`	B	
`grep -E`, `egrep`	`-e`, `-f`	E	
`lex`		E	2, 7
`more`		B	
`nl`	`-b p`*pattern*	B	
`pax`	*Operanden*	Shell-Namensmuster	
`perl`		E	1, 2, 3, 4, 7
`sed`		B	1
`vi`		B	1, 4, 5

Erläuterung der Besonderheiten:

1. Mit dem Schrägstrich (in einigen Programmen auch dem Fragezeichen für Rückwärts-Suche) werden die Suchmuster begrenzt; deshalb werden diese beiden Zeichen innerhalb der Suchmuster mit dem inversen Schrägstrich maskiert.

2. Es gibt weitere mit `\` eingeleitete Ersatzdarstellungen von Zeichen, etwa `\\` für den inversen Schrägstrich selbst und `\a`, `\b`, `\f`, `\n`, `\r`, `\t` und `\v` für die ASCII-Steuerzeichen `BEL`, `BS`, `FF`, `LF`, `CR`, `TAB` und `VT`.

3. Es gibt neben den unter Punkt 2 beschriebenen, mit dem inversen Schrägstrich eingeleiteten Symbolen weitere solche, insbesondere für Zeichenklassen (z.B. `\d` für Ziffern, `\s` für alle Arten Zwischenraum).

4. Die Stellen, an denen Wörter aneinander grenzen, können beim Suchen bezeichnet werden, z.B. `\b` für Wortgrenze, `\B` für keine Wortgrenze, `\<` für Wortanfang, `\>` für Wortende, oder eine Teilmenge dieser Konventionen. Einzelheiten bitte im Manual des entsprechenden Programms nachlesen.

5. Die Tilde ist auch ein Sonderzeichen, das maskiert werden muss.

6. Der reguläre Ausdruck wird immer links verankert.

7. Das Programm hat zahlreiche Erweiterungen in den regulären Ausdrücken, die mit seiner Arbeitsweise zusammenhängen. Einzelheiten bitte im Manual nachlesen.

8. Die Schreibweisen entsprechen BRE; es gibt auch Rückverweise. Zusätzlich gibt es, wie sonst nur bei ERE, den senkrechten Strich zur Bezeichnung von Alternativen, der (in Analogie zu den Klammern) auch mit vorangestelltem inversen Schrägstrich versehen sein muss, um seine besondere Bedeutung zu erhalten.

Maskierung und Mengen von Einzelzeichen

In diesem Abschnitt geht es darum, welche Notation verwendet wird, um ein einzelnes Zeichen darzustellen, und zwar entweder ein bestimmtes oder ein beliebiges aus einem Vorrat verschiedener Zeichen, z.B. eine beliebige Ziffer.

Ein *bestimmtes* einzelnes Zeichen wird einfach durch sich selbst dargestellt, es sei denn, dass es innerhalb regulärer Ausdrücke eine besondere Bedeutung hat, wie etwa der Punkt (beliebiges Zeichen), der Stern (Wiederholung des letzten Teilausdrucks), die öffnende eckige Klammer (Beginn der Notation einer Menge von Einzelzeichen) oder der inverse Schrägstrich (Maskierung, Beginn einer Klammerung). In diesem Fall muss es *maskiert* werden. Die einfachste Möglichkeit der Maskierung besteht darin, es in eckige Klammern zu setzen; dadurch wird eine Zeichenmenge definiert, die genau dieses eine Zeichen enthält. Das funktioniert für alle Zeichen, auch die eckigen Klammern selbst, nur nicht für den Zirkumflex-Akzent. Eine Alternative besteht darin, dem zu maskierenden Zeichen einen inversen Schrägstrich voranzustellen; es muss sich dann aber auch wirklich um ein Zeichen handeln, das sonst speziell behandelt würde, während das bei der Maskierung mit der eckigen Klammer gleichgültig ist.

Im Rest dieses Abschnittes geht es um eine vereinfachte Darstellung von Mengen einzelner Zeichen. Gemäß der ERE-Syntax kann man zwar die Sprache "eines der Zeichen `a`, `b` oder `c`" als `(a|b|c)` darstellen, aber spätestens bei der Sprache "ein Buchstabe aus dem Bereich von `a` bis `z`" wird diese Darstellung unhandlich und man greift lieber zu der abgekürzten Form `[a-z]`. Gemäß der BRE-Syntax ist dies sogar die einzige Möglichkeit, da der senkrechte Strich, mit dem Alternativen notiert werden, dort nicht existiert.

Innerhalb der eckigen Klammer verlieren die meisten Zeichen ihre besondere Bedeutung, die sie sonst in regulären Ausdrücken haben. Dafür gibt es leider auch Zeichen, die hier eine besondere Bedeutung erlangen:

- Die öffnende eckige Klammer hat eine Sonderbedeutung, wenn ihr ein Punkt, Gleichheitszeichen oder Doppelpunkt folgt.

- Die schließende eckige Klammer beendet die ganze Liste oder den von einer öffnenden eckigen Klammer begonnenen Teil, es sei denn, sie steht ganz vorne in der Liste.

- Mit dem Bindestrich (Minuszeichen) wird ein Bereich bezeichnet, es sei denn, er steht ganz am Anfang oder am Ende der Liste.

Man bekommt beim Aufbau der Liste also keine Probleme, wenn man die Zeichen wie folgt anordnet:

- Ist ein Bindestrich dabei, so kommt er ans Ende.

- Ist eine öffnende eckige Klammer dabei, so kommt sie ans Ende, jedoch vor einen gegebenenfalls ebenfalls vorhandenen Bindestrich.

- Ist eine schließende eckige Klammer dabei, so kommt sie an den Anfang.

- Ist ein Zirkumflex-Akzent dabei, so kommt er nicht an den Anfang; er verdrängt dabei notfalls Zeichen, die nach den voranstehenden Regeln an das Ende sollen.

Wenn die Liste fertig ist, kommt ein Zirkumflex-Akzent davor, wenn der Ausdruck die Zeichen darstellen soll, die *nicht* in der Liste vorkommen. Am Schluss kommen die eckigen Klammern drum herum.

Jetzt kann nur noch der etwas falsch machen, der meint, einen Bereich von Zeichen mit dem Bindestrich oder einer eckigen Klammer beginnen oder enden lassen zu müssen, aber der muss dann eben selbst in die Norm schauen, wie sein kryptischer Ausdruck verstanden wird.

Welche Zeichen in einem Bereich liegen, der mit Bindestrich notiert wird, richtet sich nicht nach der Codierung (etwa im ASCII- oder ISO8859-1-Code), sondern nach der Sortierreihenfolge, die systemweit oder benutzerspezifisch nach den Gepflogenheiten einzelner Länder eingestellt sein kann. Es ist also nicht von vorneherein klar, ob etwa das scharfe ß im Bereich `[a-z]` enthalten ist oder nicht. Spezifiziert man explizit die "POSIX Locale" durch Setzen der Environment-Variablem `LC_COLLATE` auf den Wert `POSIX`, so liegen wenigstens die Zeichen des ASCII-Codes in der durch diesen Code spezifizierten Reihenfolge; über die übrigen Schriftzeichen schweigt sich die Norm allerdings aus.

Kennt man Sortierreihenfolge und Zeichenklassen der eingestellten "Locale" (dieses Wort, das "Schauplatz" bedeutet, spricht sich mit langem "a" ähnlich dem deutschen Wort "Lokal"), so gibt es weitere Darstellungen von Zeichenmengen wie etwa

- `[.ch.]` (die Zeichenfolge `ch`, aber nur falls sie bei der Sortierung gemeinsam als ein Zeichen betrachtet wird),
- `[=a=]` (die Zeichen, die so wie `a` einsortiert werden) und
- `[:digit:]` (die Zeichen, die lokal als Ziffern betrachtet werden).

Dies hier im einzelnen zu erläutern, würde zu weit führen. Die letzte Möglichkeit könnte aber nützlich sein, will man etwa die Menge der Buchstaben explizit länderabhängig festlegen. Es gibt hier:

Zeichenklasse	Bedeutung	internationale Voreinstellung ("POSIX-Locale")
`[:alnum:]`	alphanumerisches Zeichen	`[[:alpha:][:digit:]]`
`[:alpha:]`	Buchstabe	`[[:upper:][:lower:]]`
`[:blank:]`	Zwischenraum	*Zwischenraum und horiz. Tabulator*
`[:cntrl:]`	Steuerzeichen	*Steuerzeichen ausschließlich Zwischenraum*
`[:digit:]`	Dezimalziffer	`[0123456789]`
`[:graph:]`	Schriftzeichen	`[[:alpha:][:digit:][:punct:]]`
`[:lower:]`	Kleinbuchstabe	`[abcdefghijklmnopqrstuvwxyz]`
`[:print:]`	druckbares Zeichen	`[ [:graph:]]`
`[:punct:]`	Satzzeichen	`[]!"#$%&’()*+,./:;<=>?@\^_‘{|}~[-]`
`[:space:]`	Freiraum	*alle Arten Zwischenraum, Zeilenwechsel und Tabulator*
`[:upper:]`	Großbuchstabe	`[ABCDEFGHIJKLMNOPQRSTUVWXYZ]`
`[:xdigit:]`	hexadekadische Ziffer	`[[:digit:]A-Fa-f]`

Man kann aber nicht voraussetzen, dass diese internationale Voreinstellung in Kraft ist, da sie durch Environment-Variable gesteuert wird, die anders gesetzt sein können. Im Zweifel muss man explizit die Environment-Variable `LC_CTYPE` auf den Wert `POSIX` setzen.

Mustersuche

Bisher hat uns nur die Frage beschäftigt, wie ein regulärer Ausdruck eine Menge von Zeichenreihen beschreiben kann. Es ging also um eine reine Ja/Nein-Entscheidung, ob eine Zeichenreihe in der durch den Ausdruck definierten Sprache liegt, wenn zum Beispiel das Filterprogramm `grep` entscheiden soll, ob die entsprechende Eingabezeile in die Ausgabe übernommen werden soll oder nicht.

Jetzt wenden wir uns der Frage zu, wie wir innerhalb einer längeren Zeichenreihe einen Teil finden und herauspräparieren können, der uns interessiert, wobei wir durch reguläre Ausdrücke sowohl beschreiben, *wo* wir suchen als auch *was* uns interessiert. Nehmen wir als Beispiel die Ausgabe des Unix-Kommandos "`ls -l abc`", die etwa so aussehen könnte:

```
rw-r-----   1 joeuser  staff        580 Feb 25 14:19 abc
```

Wenn wir einem menschlichen Zuhörer erklären würden, wo hier der Name des Benutzers steht, dem die Datei gehört, könnte diese Erklärung beispielsweise so lauten:

> Erst kommen vielleicht ein paar Zwischenräume, dann kommt ein Teil, der nur aus den Buchstaben `b`, `c`, `d`, `p`, `r`, `s`, `w` und `x` sowie aus Minuszeichen besteht, dann kommt mindestens ein Zwischenraum, dann mindestens eine Ziffer, dann mindestens ein Zwischenraum, und dann - hier fängt der Teil an, der mich interessiert - kommen Zeichen, die keine Zwischenräume sind und dann - aber das interessiert mich schon nicht mehr - kommt noch ein Zwischenraum ...

Genau diese Information kann man dem Programm `expr` mit Hilfe eines regulären Ausdrucks mitgeben, wobei der interessierende Teil durch die runden Klammern (die in BRE-Syntax mit voranstehendem \ geschrieben werden) gekennzeichnet wird. So würde in diesem Beispiel das Kommando

```
expr "`ls -l abc`" : ' *[bcdprswx-]*  *[0-9][0-9]*  *\([^ ]*\) '
```

als Ergebnis `joeuser` liefern.

Sucht man dabei nach einem Teilstring, der durch seine Gestalt eindeutig identifiziert ist, so dass man die Umgebung nicht zu beschreiben braucht, so wäre es natürlich, wenn der Ausdruck mit `.*` beginnen und auch enden würde, ebenso in dem Fall, dass nur eine kleine Umgebung vor und nach dem Teilstring mit herangezogen werden muss. Es hat sich aber als Konvention eingebürgert, dass umgekehrt vorgegangen wird: bei den meisten Programmen, die nach Mustern suchen, z.B. auch bei `grep`, wird *irgendwo* im String nach dem Muster gesucht, d.h. das Suchmuster wird *implizit* vorne und hinten mit `.*` ergänzt. Nur wenn man das *nicht* will, muss man explizit am Anfang einen Zirkumflex-Akzent oder am Ende ein Dollarzeichen schreiben, um die Suche auf den Anfang bzw. das Ende der Zeichenreihe zu beschränken; man sagt dann, das Muster sei am linken bzw. rechten Ende *verankert*. Warum diese Konvention ganz praktisch ist, wird im nächsten [S.15] Abschnitt diskutiert. Das Programm `expr` ist insofern besonders inkonsequent, als es implizit das Muster am linken, nicht aber am rechten Ende verankert. Im Beispiel wurde das benutzt: es wurde nämlich spezifiziert, was am Anfang der Zeile vor dem zu extrahierenden Teil steht, nicht aber, was ihm bis zum Ende der Zeile folgt.

Die Möglichkeit, mit Hilfe regulärer Ausdrücke Information aus Texten herauszufischen und später wiederzuverwenden, kommt sehr häufig in der Praxis vor. Beispiele sind:

- Aus der Ausgabe eines Programms, die als Text vorliegt, soll relevante Information herausgezogen und von einem anderen Programm weiterverarbeitet werden.

- Eine Anwendung soll eine bequem zu benutzende Eingabesprache ohne allzu starre Konventionen haben. Dazu ist es erforderlich, die Eingabe syntaktisch zu analysieren. In vielen Fällen reicht es dazu aus, die zulässigen Eingabetexte mit Hilfe eines regulären Ausdrucks zu beschreiben, in welchem die weiterzuverarbeitenden Teile entsprechend gekennzeichnet sind.

- Mit Hilfe eines Editors sollen dieselben Änderungen an vielen Stellen gemacht werden, z.B. von Wörtern, die in Klammern und dort in Anführungszeichen stehen, die Anführungszeichen entfernt werden. Es ist dann sehr praktisch, wenn der Editor bei seiner Ersetzungsfunktion reguläre Ausdrücke im Suchstring ebenso zulässt wie dort gefundene Teilstrings im Ersetzungsstring. Man würde in diesem Beispiel etwa die Ersetzung von `("\([^"]*\)")` durch `(\1)` verlangen, wobei `\1` für den String steht, der an *der* Stelle im Suchstring gefunden wurde, wo im regulären Ausdruck die Klammer stand. Die beiden verbreitetsten Editoren unter Unix, `vi` und `emacs`, können das beide.

Mit den Möglichkeiten der Shell, auch wenn sie um Programme wie `expr` oder `grep` erweitert sind, kommt man für die beiden ersten Anwendungen nicht weit; man verwendet dann Skriptsprachen wie `perl` (universell einsetzbar) oder `awk` (stärker auf den ersten Anwendungsfall beschränkt).

Die in diesem Abschnitt dargestellten Möglichkeiten sind es, die regulären Ausdrücken viele Anwendungsbereiche erschließen, wo es ohne sie sehr viel umständlicher zugegangen wäre. Man versuche etwa, die Analyse der Ausgabezeile von "`ls -l`" im obigen Beispiel selbst in Pascal oder einer anderen Programmiersprache (in C natürlich ohne Verwendung der Funktion `regexp`) zu

programmieren; das wird um einiges länger und meist auch unübersichtlicher als die Form mit regulären Ausdrücken, so kryptisch diese auf den ersten Blick auch aussehen mag.

Der Preis für diese Bequemlichkeit ist allerdings hoch, auch wenn das den Anwender nicht zu interessieren braucht: durch solche Erweiterungen des Begriffs "regulärer Ausdruck" ist das Gebiet verlassen worden, in dem die vielen schönen und einfachen Aussagen gelten, die im zweiten, theoretischen Teil dieses Artikels zusammengestellt sind. Insbesondere die wertvolle Eigenschaft, dass man immer deterministisch vorgehen kann, dass also das Programm, das nach einem Muster sucht, nie eine getroffene Entscheidung rückgängig machen muss, gilt hier nicht mehr. Dazu ein Beispiel: Mit

```
expr abaaababbaababaabaaaababc : '.*\(a*ba*ba*b\)c'
```

sollen aus dem links vom Doppelpunkt stehenden String die letzten drei `b`'s vor dem `c` mit den davorliegenden `a`'s herausgefunden werden. Bei der Analyse muss nun sowohl am Anfang als auch nach jedem `b` mit dem Beginn des herauszufindenden Strings gerechnet werden. Jeder dieser Kandidaten ist also noch aktiv, nachdem der nächste schon begonnen hat.

Außerdem ist die Definition nicht eindeutig: es ist in diesem Fall nicht klar, wo der Bereich des `.*` endet und der des ersten `a*` beginnt. In der Norm ist das leider nicht eindeutig definiert, so dass verschiedene Implementierungen verschiedene Ergebnisse liefern können. Im nächsten Abschnitt wird das eingehender diskutiert.

Gier (*greed*)

In manchen Anwendungsprogrammen, die mit regulären Ausdrücke umgehen, zum Beispiel in `perl`, lässt sich die "Gier" (engl. *greed*) regeln, mit der Zeichen von einem Wiederholungsoperator, also einem Stern oder einem Pluszeichen, "verschlungen" werden. Und zwar arbeiten diese Operatoren normalerweise mit maximaler Gier, können aber gezähmt werden, indem man ihnen ein Fragezeichen anhängt. Welche Wirkung das hat, soll anhand dreier kurzer perl-Skripts gezeigt werden. Das Skript

```
$s = 'abaaababbaababaabaaaababc';

if ($s =~ /^(.*)(a*ba*ba*b)c$/) {
  print "gieriger Stern\n erste Klammer: \"$1\"\nzweite Klammer: \"$2\"\n\n";
};

if ($s =~ /^(.*?)(a*ba*ba*b)c$/) {
  print "nicht gieriger Stern\n erste Klammer: \"$1\"\nzweite Klammer: \"$2\"\n\n";
};
```

liefert die folgende Ausgabe:

```
gieriger Stern
 erste Klammer: "abaaababbaababaa"
zweite Klammer: "baaaabab"

nicht gieriger Stern
 erste Klammer: "abaaababbaabab"
zweite Klammer: "aabaaaabab"
```

Dieses erste Beispiel zeigt, dass die Eindeutigkeit, die am Ende des letzten Abschnittes schmerzlich vermisst wurde, durch die Möglichkeit, die Gier zu regeln, wiederhergestellt wird. Die Gier (oder ihr Fehlen) geht aber nie soweit, dass ein Wort der Sprache nicht erkannt wird, wenn man es mit weniger (bzw. mehr) Gier hätte erkennen können. Im Beispiel verschlingt also auch ein gieriger Stern nicht

etwa das drittletzte b mit, weil das ja noch für die zweite Klammer gebraucht wird, und auch ein bescheidener Stern auf Hungerkur isst trotzdem brav alle überzähligen bs auf.

Keine Gier bedeutet nur, dass der Stern möglichst früh die Mahlzeit einstellt, nicht aber, dass er sie möglichst spät beginnt; das wird durch das zweite Beispiel demonstriert:

```
$s = 'aaaaaaaaaab';

if ($s =~ /(a*b)/) {
  print " (mit Gier) gefunden wurde: \"$1\"\n\n";
};

if ($s =~ /(a*?b)/) {
  print "(ohne Gier) gefunden wurde: \"$1\"\n\n";
};
```

mit der Ausgabe:

```
 (mit Gier) gefunden wurde: "aaaaaaaaaab"

(ohne Gier) gefunden wurde: "aaaaaaaaaab"
```

Der wichtigste Anwendungsfall für die Giervermeidung ist der, dass es wie im folgenden dritten Beispiel darum geht, einen Textteil zu extrahieren, der durch ein Muster an seinem Ende gegeben ist. Dasselbe Beispiel ohne die Möglichkeit der Giervermeidung wird recht kompliziert [S.18] . Dazu das folgende Beispiel

```
$s = 'Anfang Inhalt Ende, und noch ein Ende';

if ($s =~ /Anfang(.*)Ende/) {
  print " (mit Gier) zwischen Anfang und Ende steht \"$1\"\n\n";
};

if ($s =~ /Anfang(.*?)Ende/) {
  print "(ohne Gier) zwischen Anfang und Ende steht \"$1\"\n\n";
};
```

mit der Ausgabe:

```
 (mit Gier) zwischen Anfang und Ende steht " Inhalt Ende, und noch ein "

(ohne Gier) zwischen Anfang und Ende steht " Inhalt "
```

Bei der Einführung von Ankern [S.14] (also Zeichen, mit denen im regulären Ausdruck Anfang oder Ende des Textes bezeichnet werden) war gesagt worden, dass man im Grunde auf die Anker verzichten könnte, wenn man stattdessen bei einer Suche an beliebiger Stelle im Text den uninteressanten Teil explizit mit `.*` angeben würde. Für die bloße Entscheidung über das Vorliegen eines Musters ist das richtig, für die Mustersuche nicht: würde man diese Angabe machen, wäre dieser Stern (je nach Anwendung) möglicherweise gierig und man würde das letzte und nicht das erste Vorkommen des Musters im Text finden. Das dürfte auch ein Grund für die Konvention sein: es ist einfacher, die Verankerung am Textanfang durch ein Zeichen explizit zu bezeichnen, als dass man im umgekehrten Fall die Suche irgendwo im Text durch einen möglicherweise komplexen regulären

Ausdruck bezeichnen muss, der fast alles wegfressen soll, aber trotzdem nicht in seiner Gier die ersten Vorkommen des Suchmusters gleich mitverschlingen darf.

Rückverweise

Im vorletzten Abschnitt haben wir gesehen, wie man mit Hilfe regulärer Ausdrücke Teilstrings herausschneiden und anderweitig weiterverwenden kann. In den Editoren `vi` und `emacs` wird das jeweils so formuliert, dass `\1` den Teilstring bezeichnet, der bei der Analyse auf den Inhalt der ersten Klammer gepasst hat. Ebenso bezeichnen `\2` bis `\9` die Inhalte der weiteren Klammerpaare, die dabei in der Reihenfolge des Auftretens jeweils der öffnenden Klammern gezählt werden. Da bietet sich als eine abermalige Erweiterung der regulären Ausdrücke nachgerade an, unter Benutzung derselben Notation einen vorn herausgeschnittenen Teilstring weiter hinten im *selben* regulären Ausdruck zu verwenden. Beispielsweise bezeichnet dann `\(.*\)\1` in BRE-Notation alle diejenigen Strings, die aus zweimal demselben Text bestehen, z.B. `aabaaaba`.

Wir nehmen wieder das Beispiel [S.13] vom Anfang des letzten Abschnitts auf, wo mit Hilfe eines regulären Ausdrucks der Name des Dateieigentümers aus der Ausgabe von "`ls -l` *filename*" herausgeschnitten wurde. Diesmal verwenden wir diesen Teilausdruck dazu, um im selben Text, noch während der Analyse anhand desselben regulären Ausdrucks, den Namen des Eigentümers mit dem Dateinamen zu vergleichen. Wir können etwa ein ganzes Verzeichnis mit "`ls -l`" auflisten lassen und dann mit `grep` diejenigen Zeilen herausfiltern, in denen der Dateiname mit dem Namen des Dateieigentümers übereinstimmt:

```
ls -l | grep '^ *[bcdprswx-]*  *[0-9][0-9]*  *\([^ ]*\) .* \1$'
```

Die Stelle, wo der Name bei seinem ersten Auftreten erkannt wird, ist hier in roter Farbe gekennzeichnet und der Abgleich mit dem Namen am Ende der Zeile (also vor dem Anker rechts) in blauer. Dazwischen stehen zwei begrenzende einzelne Zwischenräume und die Zeichenfolge `.*` für den Teil des Textes, der hier nicht interessiert.

Solche Rückverweise gehören zum Standardrepertoire von BREs, *nicht* jedoch von EREs; insofern stellen EREs hier trotz ihres Namens keine Erweiterung, sondern eine *Einschränkung* gegenüber BREs dar. Wir hätten also das Kommando `grep` nicht zusammen mit der Option `-E` verwenden können. Der Grund dafür liegt darin, dass nur in BREs sichergestellt ist, dass die Klammer, auf die mit dem Rückverweis Bezug genommen wird, auch tatsächlich zur Analyse verwendet worden ist. Man betrachte etwa die Analyse von `abbd` mittels des Ausdrucks `(ab|c(a*))b\2`, wo an der Stelle, an der das `d` gelesen wird, nicht definiert ist, was `\2` bedeutet. Mit BREs tritt dieser Fall wegen der fehlenden Möglichkeit, Alternativen zu spezifizieren, nicht auf. Andere Werkzeuge wie etwa `perl` erlauben diesen Fall trotzdem; undefinierte Teile sind dort mit dem leeren String vorbesetzt.

Die Warnung am Ende des letzten Abschnittes, man würde das ruhige Fahrwasser der regulären Sprachen hier verlassen, gilt mit solcherart erweiterten regulären Ausdrücken natürlich umso mehr: schon die durch `\(.*\)\1` bezeichnete Sprache ist nicht regulär - übrigens nicht einmal kontextfrei -, und das dürfte auch für praktisch alle regulären Ausdrücke gelten, in denen Rückverweise vorkommen, es sei denn, die Ausdrücke in den dazugehörigen Klammern bezeichnen alle nur endliche Sprachen (d.h. sie enthalten keine unbeschränkten Wiederholungen). Außerdem gehört nicht viel Phantasie dazu, sich auszumalen, in wievielfacher Weise es geschehen kann, dass erst am Ende des Textes offenbar wird, worauf man am Anfang etwa hätte achten müssen, so dass der Analysator öfters und sogar in verschachtelten Analyseschritten zurücksetzen muss.

Beispiele

Diese Beispiele sind alle in der ERE-Notation [S.7] , wenn nicht etwas anderes explizit angegeben ist.

Wortsuche: Schreibungsvarianten, Ausschluss von Zufallstreffern

Mit regulären Ausdrücken kann man nach Wörtern suchen, die in unterschiedlicher Weise geschrieben werden: `(Ph|F)otogra(ph|f)ie`, `Känguruh?`, `internationali[sz]ation`, `Schwein[se]braten`, `Schifff?ahrt`, `(E|Ae|Ä)ther`, `Mo[çcsz]ambi(k|que)`. Wie das letzte Wort zeigt, darf man dabei umso großzügiger auch falsche Schreibungen mit zulassen, je unwahrscheinlicher es ist, *andere* gültige Wörter zu treffen. Auch eine gleichzeitige Suche nach mehreren Formen eines Wortes unabhängig von Beugungsendungen lässt sich so erreichen:
`eine?[mnrs]? geschenkte?[mnrs]? G[aä]ule?s?`

Umgekehrt kann man auch versuchen, bei der Suche nach Wörtern mit Hilfe regulärer Ausdrücke falsche Treffer zu eliminieren. Sucht man etwa das Wort `nett`, so gelten die Treffer `Nettopreise`, `Variablenettikett`, `Subnetting`, `Bajonettverschlüssen`, `nettype`, `NetTape` (alle mit Harvest auf dem LRZ-WWW-Server gefunden) nicht. Man schließt sie aus, indem man ein wenig Kontext mitgibt, etwa mit einem Kommando wie:

```
grep -Ei '(^|[ (])nett([^aoy]|$)' Dateiname
```

Die erste Klammer im regulären Ausdruck beschränkt die Suche darauf, den String `nett` nur am Anfang eines Wortes zu suchen: ganz am Textanfang (durch ^ als Anker dargestellt), nach einem Zwischenraum oder nach einer öffnenden Klammer (dem einzigen Satzzeichen, das üblicherweise ohne Zwischenraum vor einem Wort steht). Mit der zweiten Klammer sollen die zufälligen falschen Treffer ausgeschlossen werden; auch hier darf man den Fall nicht vergessen, dass das Suchwort unmittelbar am Textende steht, was für `grep` ja jedes Zeilenende ist. Im Gegensatz zur Verbreiterung der Suche lässt sich eine zweckmäßige Einschränkung nur selten vorher ausknobeln: man ermittelt sie durch Versuch und Irrtum.

Texte, die an ihrem Anfang und ihrem Ende erkannt werden

Eine häufige Anwendung regulärer Ausdrücke sind solche Texte, bei denen man zwar wenig über den Inhalt weiß, aber durchaus etwas über Anfang und Ende. Das erste Beispiel dafür sei ein beliebiger Text in runden Klammern. Der auf den ersten Blick naheliegendste reguläre Ausdruck dafür ist

```
[(].*[)]
```

Man beachte die Notwendigkeit, die Klammern durch einen vorangestellten inversen Schrägstrich oder wie hier durch Einschluss in eckige Klammern zu maskieren [S.11] . Dieser Ausdruck ist so unmittelbar einleuchtend, dass man gar nicht ahnt, welche Tücken er enthalten kann:

- Es kommt auf das Applikationsprogramm an, ob damit Klammern erkannt werden, die in einer Zeile geöffnet und erst in einer der nachfolgenden Zeilen wieder geschlossen werden. Umgekehrt kommt es auf die beabsichtigte Anwendung an, ob es überhaupt erwünscht ist, dass solche Klammerpaare erkannt werden. Was zu tun ist, wenn die Anforderungen nicht mit dem Verhalten des Applikationsprogramms übereinstimmen, hängt zu sehr von beiden ab, als dass es da generell gültige Tipps gäbe; man sollte aber wenigstens die Problematik immer mit bedenken.

- Es kommt ebenfalls auf das Applikationsprogramm an, welches Paar von Klammern gefunden wird, wenn es mehr als eines gibt. Man kann sich darauf verlassen, dass die öffnende Klammer des regulären Ausdrucks die *erste* öffnende Klammer des untersuchten Textes bezeichnet, der später eine schließende Klammer folgt. Keinesfalls darf man aber immer davon ausgehen, dass das mit der schließenden Klammer auch so ist. Ein durchaus vernünftiger Algorithmus zum Mustervergleich kann darin bestehen, dem Teilausdruck `.*` so *viel* Text wie möglich zuzuordnen, auch dann, wenn dieser dann selbst schließende und öffnende Klammern enthält. Das kann - besonders wenn gleichzeitig Klammerpaare über mehrere Zeilen hinweg erkannt werden - zu sehr bösen Überraschungen führen.

Wegen dieser immer lauernden Gefahr sollte man es sich daher angewöhnen, in solchen Fällen *immer* die nur unwesentlich komplexere Form

```
[(][^()]*[)]
```

zu wählen, mit der im eingeklammerten Text selbst keine öffnende oder schließende Klammer stehen darf, so dass man immer ein zusammengehöriges innerstes Klammerpaar erwischt. Besteht das Endezeichen allerdings aus mehreren Zeichen, kann das leider beliebig umständlich werden. Ist etwa der Text zwischen `Anfang` und `Ende` eingeschlossen, so müsste man eigentlich den regulären Ausdruck so schreiben:

```
Anfang([^E]|E[^n]|En[^d]|End[^e])*(E(nd?)?)?Ende
```

Der mit dem Stern wiederholte Teil akzeptiert nicht ganz alle Texte, die das Wort `Ende` nicht enthalten: wie man leicht sieht, ist z.B. `En` nicht dabei, und der String `AnfangEnEnde` würde ohne den eingeschobenen Teilausdruck `(E(nd?)?)?` nicht richtig erkannt.

Oft kann man da an der Präzision Abstriche machen, wenn klar ist, dass bestimmte Texte im Inneren nicht vorkommen. In Skripten sollte man aber immer den allgemeinsten Fall berücksichtigen: sie werden manchmal nach Jahren für einen anderen Zweck verwendet, wo eine stillschweigend vom ersten Programmierer eingebaute Voraussetzung nicht mehr erfüllt ist.

Fein heraus ist man in solchen Fällen, wenn das Anwendungsprogramm es gestattet, festzulegen, dass der Wiederholungsoperator seine Gier [S.15] zügeln soll, mit der er Text verschlingt.

Einfach maskierter Text

Nehmen wir als Beispiel eine E-Mail-Adresse, bei der der wirklich relevante Teil in spitze Klammern eingeschlossen ist (darüber hinaus gibt es weitere erlaubte Schreibweisen für E-Mail-Adressen, die jetzt nicht interessieren, siehe RFC822 (ftp://ftp.leo.org/pub/comp/doc/standards/rfc/rfc-0800-0899/rfc0822.gz)). Also etwa so:

```
LRZ-Hotline <hotline@lrz.de> (Return requested)
```

Ein regulärer Ausdruck dafür könnte einfach so aussehen:

```
.*<.*>.*
```

Hier durfte man, im Gegensatz zum vorangegangenen Abschnitt, die unbekannten Teile einfach mit `.*` bezeichnen, da ja genau ein < und ein > vorkommen müssen. Es ist allerdings erlaubt, diese beiden Zeichen auch sonst vor, zwischen und nach dem spitzen Klammernpaar zu verwenden, wenn sie dort durch Einschluss in Anführungszeichen maskiert sind. Ein regulärer Ausdruck, der dem Rechnung

trägt, ist ganz einfach zu realisieren: Der Punkt in `.*` muss durch etwas ersetzt werden, was neben einzelnen Zeichen auch maskierte Zeichenreihen enthält, nämlich durch `[^"<>]|"[^"]*"` (ein harmloses Zeichen oder ein in Anführungszeichen eingeschlossener Text). Damit wird der oben angegebene Ausdruck zu

```
([^"<>]|"[^"]*")*<([^"<>]|"[^"]*")*>([^"<>]|"[^"]*")*
```

Das sieht schlimm aus, aber nur für den, der bei der Entstehung nicht dabei war.

Mehrfach maskierter Text

Gibt es mehrere Möglichkeiten der Maskierung in der Weise, dass sich die zur Maskierung eingesetzten Zeichen sogar gegenseitig maskieren können, kann die Erstellung eines regulären Ausdrucks wie im letzten Beispiel schnell unübersichtlich werden. Als Beispiel soll nun ein regulärer Ausdruck für alle jene Unix-Kommandozeilen konstruiert werden, bei denen alle mit Apostroph oder Anführungszeichen begonnenen Texte ordnungsgemäß abgeschlossen sind. Im einzelnen gelten dafür die folgenden Regeln:

- Ein mit Apostroph begonnener Text muss mit Apostroph abgeschlossen werden.
- Ein mit Anführungszeichen begonnener Text muss mit Anführungszeichen abgeschlossen werden.
- Ausserhalb des von Apostrophen eingeschlossenen Textes maskiert ein inverser Schrägstrich das folgende Zeichen. (Diese Regel ist hier insofern ungenau wiedergegeben, als es eigentlich darauf ankommt, wann der inverse Schrägstrich für sich selbst steht und wann er das nachfolgende Zeichen maskiert, ob also `\a` für die zwei Zeichen `\` und `a` oder nur für ein unnötigerweise maskiertes `a` steht. Im Zusammenhang unserer Aufgabe spielt das aber keine Rolle: hier kommt es nur darauf an, welche der Zeichen `'`, `"` und `\` durch `\` maskiert werden, und dafür ist die angegebene vereinfachte Formulierung gut genug.)

Die Aufgabe kann man durchaus in derselben Weise zu lösen versuchen wie die vorige. Viel besser ist es aber, systematisch vorzugehen und sich alle Möglichkeiten aufzuschreiben, wie die verschiedenen Maskierungszeichen aufeinander einwirken. Nur: mit einer solchen Tabelle kommen wir bei unserem jetzigen Wissensstand nicht weiter. Deswegen wird die Darstellung des vollständigen Lösungsweges [S.35] in den Theorie-Teil dieses Artikels verlegt und hier nur die fertige Lösung verraten:

```
([^'"\]|[\].|['][^']*[']|["]([^"\]|[\].)*["])*
```

Gregorianische Kalenderdaten

Es soll ein regulärer Ausdruck angegeben werden, der genau die Kalenderdaten in der Schreibweise nach ISO 8601 (http://www.cl.cam.ac.uk/~mgk25/iso-time.html) (also Jahr-Monat-Tag mit vierstelligen Jahreszahlen und zweistelligen Monats- und Tagesangaben) beschreibt. Wir konstruieren diesen Ausdruck, indem wir zunächst Variable einführen und diese dann ersetzen, um den fertigen Ausdruck zu erhalten.

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|V00
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:

```
G = J-D|S-02-29 =
  = [1-9][0-9][0-9][0-9]-(M-T|N-(29|30)|L-31)|([1-9][0-9]V|V00)-02-29 =
  = [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
```

Der entstandene reguläre Ausdruck sieht zwar einigermaßen verwirrend aus, kennt man aber seine Entstehungsgeschichte, so muss man zugeben, dass er eigentlich gar nicht so schwer zu konstruieren war. So etwas ist typisch für reguläre Ausdrücke: sie sehen meist erheblich komplizierter aus als sie sind.

Theorie

Ganz am Anfang war eine Theorie von "besonderer Schönheit" versprochen worden. Über Schönheit lässt sich bekanntlich streiten, und die Schönheit mathematischer Theorien erschließt sich nicht jedem sofort, genauso wenig wie die Schönheit mancher anderen Kunstwerke. Mathematiker betrachten Theorien dann als besonders schön, wenn aus wenigen und einfachen Definitionen viele Aussagen hergeleitet werden können, die in einem inneren Zusammenhang stehen.

Was bisher geliefert wurde, entspricht sehr wenig diesem Bild. Das in der voranstehenden Einführung enthaltene Beispiel [S.2] eines regulären Ausdrucks wirkt einerseits kryptisch und damit hässlich, andererseits in gewisser Weise zufällig: warum hat man sich gerade für diesen Satz von Notationen entschieden, mit dem man manche Sprachen darstellen kann, andere aber nicht? In diesem Kapitel wird sich herausstellen, dass es darauf nur in geringem Maße ankommt. Auch andere Sätze von Notationen hätten vielleicht zum selben Ergebnis geführt. Es gibt nämlich eine ganze Reihe von Eigenschaften, die die regulären, also die durch reguläre Ausdrücke definierbaren Sprachen haben und andere Sprachen eben nicht. Viele Notationen, die man hinzufügen könnte, würden am beschreibbaren Sprachumfang nichts ändern, und man könnte umgekehrt manches auch weglassen und damit zwar Einbußen bei der Bequemlichkeit der Notation hinnehmen, nicht aber bei dem, was überhaupt darstellbar ist.

Der Begriff der regulären Sprache lässt sich durchaus auch ganz anders definieren als dadurch, dass es eben die Sprachen sind, die sich mit einem Satz mehr oder weniger willkürlich ausgewählter Operatoren beschreiben lassen. In diesem Kapitel werden ganz verschiedene Begriffsbildungen vorgestellt, von denen sich dann herausstellen wird, dass sie in ganz verschiedener Weise dasselbe aussagen. In der Mathematik ist so eine Situation, nämlich dass sich mehrere sehr verschieden aussehende Begriffswelten als äquivalent erweisen, oft ein Indiz dafür, dass der zugrunde liegende Begriff ein wichtiger Grundbegriff der Theorie ist. So ist auch hier: die Menge der regulären Sprachen weist eine so hohe Anzahl von Gesetzmäßigkeiten auf wie kaum eine andere Klasse formaler Sprachen.

Die Lektüre dieses Kapitels dient hauptsächlich der Bildung und nicht so sehr der Ausbildung des Lesers. Für die meisten praktischen Anwendungen kommt man ohne diese Kenntnisse aus. Höchstens bei der Konstruktion sehr verzwickter regulärer Ausdrücke können sie von Nutzen sein, wie die anschließenden Beispiele [S.35] belegen, besonders das erste.

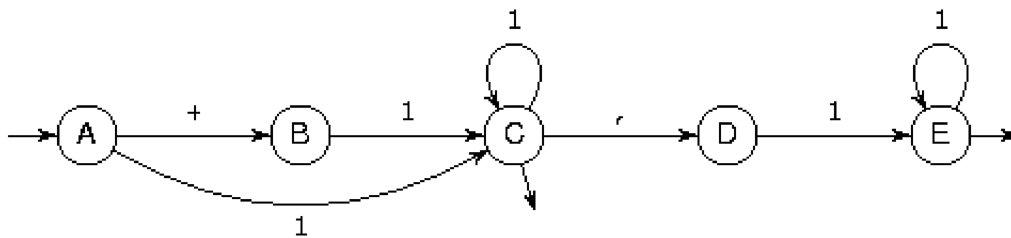
Endliche Automaten

Wir stellen uns zunächst eine Frage, die mit regulären Ausdrücken nicht zu tun hat - oder wenigstens nicht zu tun zu haben scheint. Bei manchen Sprachen, zum Beispiel der deutschen, kann man sich beliebig tief in Nebensätze verschachteln, und man braucht ein unbegrenzt großes Gedächtnis, um sich für jeden angefangenen Satz merken zu können, wie er potenziell weitergeht. In der *Praxis* gibt es natürlich nur endlich viele Fortsetzungen: jeder Redner wird schon aufgrund seiner beschränkten Redegeschwindigkeit und ebenfalls beschränkten Lebenszeit Sätze nur beschränkt tief verschachteln können - sogar nur endlich viele verschiedene Sätze sagen können -, aber von der *Sprache* her könnte man unbeschränkt viele Satzanfänge konstruieren, die alle nach unterschiedlichen Vervollständigungen verlangen. Ein viel einfacheres Beispiel sind mathematische Formeln: es müssen rechts von einer beliebigen Stelle in der Formel so viele Klammern geschlossen werden, wie links davon geöffnet wurden, was eine unendliche Anzahl von Möglichkeiten bedeutet. Wohlgemerkt: es geht nicht nur darum, dass auf ein gegebenes Anfangsstück unendlich viele Enden möglich sind (das ist immer erreichbar, wenn die Sprache selbst unendlich viele Wörter enthält), sondern dass man unendlich viele Anfangsstücke präsentieren kann, die alle verschiedene *Mengen* von Ergänzungen zu korrekten Wörtern der Sprache haben. Mit dem nächsten Beispiel wird das hoffentlich viel klarer. Nun ergibt sich die Frage, ob es nicht besonders einfache Sprachen gibt, bei denen das nicht vorkommt.

Endliche Sprachen, also Sprachen mit endlich vielen Wörtern, haben offensichtlich diese Eigenschaft. Aber auch die Sprache aus dem Beispiel [S.2] in der Einführung gehört dazu, obwohl sie unendlich ist. An jeder Stelle des Lesens ist man nämlich entweder:

A. ganz am Anfang
B. hinter einem einleitenden Vorzeichen: als restlicher Text darf nur noch eine vorzeichenlose Zahl kommen
C. nach mindestens einer Ziffer: die Zahl darf hier schon zu Ende sein oder es darf nur noch eine vorzeichenlose Zahl kommen, die aber (weil schon eine Ziffer gelesen wurde) auch gleich mit dem Komma beginnen darf
D. nach dem Komma: jetzt dürfen nur noch Ziffern (und zwar mindestens eine) kommen
E. nach Komma und mindestens einer Ziffer: die Zahl darf hier zu Ende sein oder es kommen nur noch Ziffern

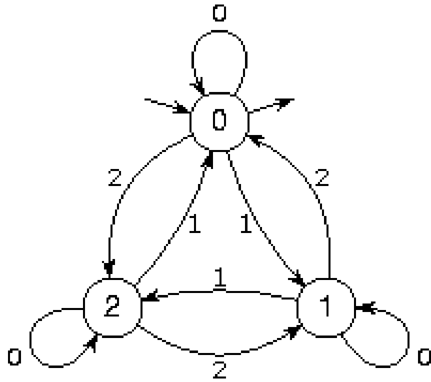
Es gibt also nur fünf, also endlich viele, verschiedene Möglichkeiten, wo man sein kann, im Gegensatz zu Sprachen mit Klammerstruktur, wo es unendlich viele Möglichkeiten gibt.



Dieser Sachverhalt lässt sich auch grafisch darstellen. Die fünf Stellen, wo man sich befindet, sind hier durch Kreise dargestellt, und die Pfeile deuten an, wie man durch Weiterlesen des nächsten Zeichens (das Pluszeichen steht hier abkürzend für ein beliebiges Vorzeichen und die Ziffer `1` für eine beliebige Ziffer) von einem dieser *Zustände* in einen anderen gelangt. Der hineingehende Pfeil beim Zustand A deutet an, dass es dort losgeht, und der herausgehende bei den Zuständen C und E sagt aus, dass dort der zu analysierende String zu Ende sein darf. Auf welchem Wege man auch immer beim hineingehenden Pfeil das Schema betritt, dort entlang der Pfeile weitergeht und es bei einem der beiden herausgehenden Pfeile verlässt: immer bilden die Beschriftungen der Pfeile hintereinander gelesen ein zulässiges Wort der Sprache. Umgekehrt gibt es auch zu jedem zulässigen String einen solchen Weg durch das Schema.

Man versuche einmal, ein solches Schema für die Sprache zu zeichnen, deren Wörter nur aus öffnenden und danach ebenso vielen schließenden Klammern bestehen, die also die Wörter `()`, `(())`, `((()))`, ... und nur diese enthält. Es wird nicht gelingen: entweder akzeptiert das Schema nicht alle diese Wörter oder darüber hinaus welche, bei denen die Zahl der schließenden Klammern nicht mit der der öffnenden übereinstimmt.

Ein solches Schema heißt *deterministischer endlicher Automat*. Man stellt sich also eine Art Maschine vor, die anfangs im *Anfangszustand* (dem mit dem hineingehenden Pfeil) ist, dann beim Lesen jeweils eines Zeichens des Textes von einem Zustand in einen anderen (oder auch wieder in denselben) übergeht und beim Erreichen des Textendes "akzeptiert" verkündet, wenn sie sich dann in einem der Endzustände (also einem mit einem herausgehenden Pfeil) befindet. In den übrigen Fällen, nämlich entweder dem Erreichen eines Nicht-Endzustandes am Textende oder dem Erreichen eines Zustandes unterwegs, bei dem es mit dem nächsten Zeichen nicht weitergeht, wird der String nicht akzeptiert. *Endlich* heißt der Automat, weil er nur endlich viele Zustände hat, und *deterministisch*, weil der Weg durch den String eindeutig festgelegt ist: es gibt nur einen Anfangszustand und zu jedem Zustand und jedem Zeichen höchstens einen Pfeil, der mit diesem Zeichen aus diesem Zustand in den nächsten führt.



Hier ist noch ein ganz anderes Beispiel für einen deterministischen endlichen Automaten: dieser Automat akzeptiert die durch drei teilbaren natürlichen Zahlen in Dezimalschreibweise, wobei der leere String auch akzeptiert wird. Dabei steht die Ziffer `0` (als Beschriftung eines Zustandsübergangs) abkürzend für eine der Ziffern `0`, `3`, `6` oder `9`, die `1` für eine der Ziffern `1`, `4` oder `7` und die `2` für eine der Ziffern `2`, `5` oder `8`. Die Idee ist ganz einfach: Im Zustand, der mit 0 bezeichnet ist, ist bisher noch gar nichts oder eine durch drei teilbare Zahl gelesen worden, und im Zustand 1 (oder 2) eine Zahl, die bei Division durch 3 den Rest 1 (bzw. 2) lässt. Jetzt muss man nur noch die Übergänge verifizieren. Beispielsweise besagt der mit `2` beschriftete Übergang vom Zustand 2 in den Zustand 1 (unten in der Mitte der Zeichnung), dass man beim Anhängen einer 2, 5 oder 8 an eine Zahl, die bei Division durch 3 den Rest 2 lässt, eine mit dem Rest 1 erhält - wie man unschwer nachrechnet. Dass das funktioniert, hat übrigens nichts mit der Dreierprobe aus der Grundschule zu tun: es geht ganz genauso für jeden beliebigen Divisor; nur wäre die Zeichnung schon für den Divisor 7 oder 13 mit 7 bzw. 13 Zuständen und 49 bzw. 130 Übergängen reichlich unübersichtlich [S.39] geworden.

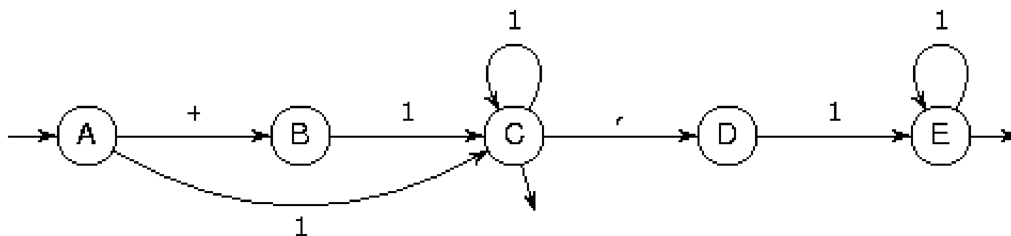
Ein deterministischer endlicher Automat lässt sich unmittelbar in ein Programm umschreiben, bei dem jeder Zustand als Marke und jeder Übergang als Sprung auf eine solche Marke dargestellt wird. Im allgemeinen gilt zwar das Programmieren mittels der `GOTO`-Anweisung als nicht gerade übersichtlicher Programmierstil; hier wollen wir aber einmal eine Ausnahme machen. Wir müssen natürlich eine Programmiersprache verwenden, in der, anders als etwa in Java, `GOTO` eine zulässige Anweisung ist. Aus Nostalgiegründen nehmen wir für das Beispiel die Sprache FORTRAN77 her; in anderen Sprachen sieht es auch nicht viel anders aus:

```
      subroutine drei (x)
        character*(*) x
      integer i, l
      character c
      logical rest0, rest1, rest2
      rest0(c) = c.eq.'0' .or. c.eq.'3' .or. c.eq.'6' .or. c.eq.'9'
      rest1(c) = c.eq.'1' .or. c.eq.'4' .or. c.eq.'7'
      rest2(c) = c.eq.'2' .or. c.eq.'5' .or. c.eq.'8'
      i = 0
      l = len (x)

C     Zustand 0
 1000 i = i+1
      if (i .gt. l) goto 9001
      if (rest0(x(i:i))) goto 1000
      if (rest1(x(i:i))) goto 1001
      if (rest2(x(i:i))) goto 1002
      goto 9000
```

```
C     Zustand 1
 1001 i = i+1
      if (i .gt. l) goto 9000
      if (rest0(x(i:i))) goto 1001
      if (rest1(x(i:i))) goto 1002
      if (rest2(x(i:i))) goto 1000
      goto 9000

C     Zustand 2
 1002 i = i+1
      if (i .gt. l) goto 9000
      if (rest0(x(i:i))) goto 1002
      if (rest1(x(i:i))) goto 1000
      if (rest2(x(i:i))) goto 1001
      goto 9000

C     String ablehnen
 9000 print *, '"', x, '" ist falsch.'
      return

C     String akzeptieren
 9001 print *, '"', x, '" ist richtig.'
      return
      end

      program probier
      call drei ('')
      call drei ('5317')
      call drei ('3')
      call drei ('2288962946180602796295895129457306782916305881')
      call drei ('2288962946180602796295895129457306782916305882')
      end
```

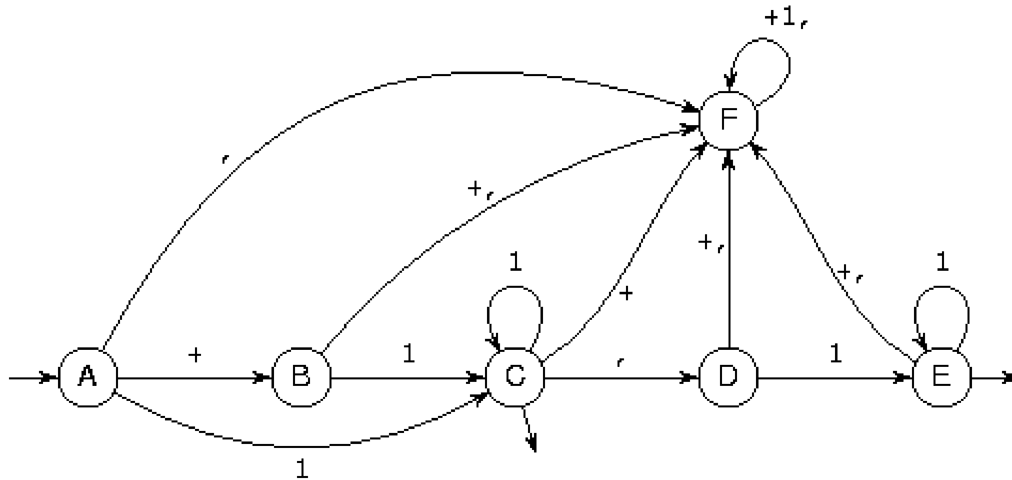
Gegenüber einer schlichten Probedivision hat das Programm den Vorteil, dass es beliebig lange Dezimalzahlen als Eingabe akzeptiert. Die für uns interessanteste Eigenschaft dieses Programms ist aber eine andere: der Speicherbedarf steht von vorneherein fest. Es werden keine Variablen dynamisch alloziert, es werden keine Unterprogramme rekursiv aufgerufen; kurz, der Speicherbedarf für die Variablen ist genau durch die statisch niedergeschriebenen Variablendeklarationen gegeben. Ein Rechner mit diesem Programm ist also wirklich ein "deterministischer endlicher Automat". Man könnte auch umgekehrt die Eigenschaft einer Sprache, durch ein Programm mit vorher fixiertem Speicherbedarf analysierbar zu sein, anstelle endlicher Automaten zur Definition verwenden, muss dann aber sehr aufpassen, den Speicherbedarf richtig zu definieren. Wir verfolgen diesen Weg hier nicht weiter.

Vervollständigung und Komplementierung endlicher Automaten

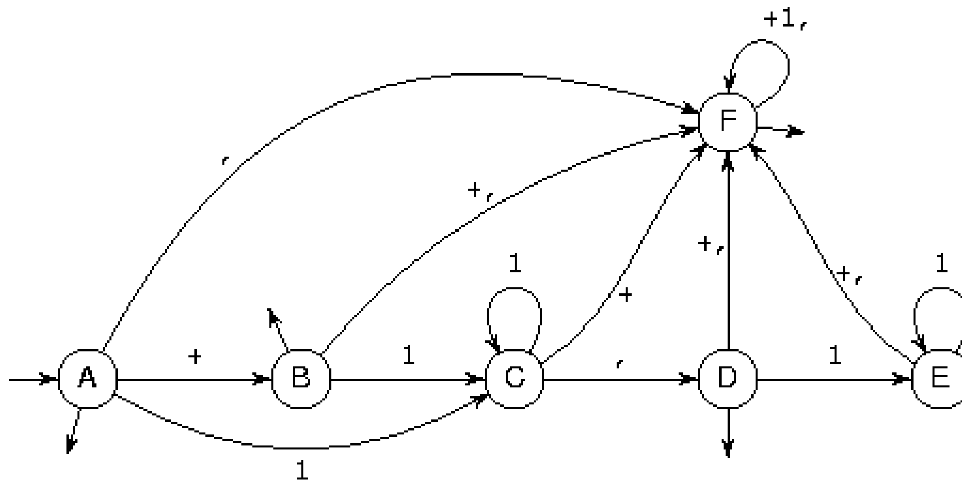


Bei der Einführung endlicher Automaten [S.23] gab es zwei Gründe, aus denen ein endlicher Automat einen vorgelegten Text nicht akzeptiert: entweder weil er sich am Textende in einem Zustand befindet, der

kein Endzustand ist, oder weil in einem Zustand unterwegs das anstehende Zeichen nicht unter denen ist, für die es einen Folgezustand gibt. Wir sehen uns dazu noch einmal den Automaten aus den ersten Beispiel an: hier würde man mit dem String `1,` im Zustand D landen, der kein Endzustand ist, und mit dem String `1+` in überhaupt keinem Zustand. Der zweite Fall ist ein sicheres Indiz dafür, dass der String auch nicht zu einem korrekten Wort fortgesetzt werden kann.



Man kann nun den endlichen Automaten durch einen zusätzlichen Zustand zu einem Automaten ergänzen, bei dem der zweite Fall nie eintritt: der nebenstehende Automat geht immer dann in den Zustand F über, wenn der ursprüngliche nicht mehr weitergewusst hätte. Mit dieser Art von Vervollständigung vereinfacht sich die Beschreibung der Funktionsweise des Automaten: jetzt gibt es immer Zustandsübergänge, egal, wo man ist und was kommt, und es kommt für die Akzeptanz eines Strings nur noch auf den erreichten Zustand am Textende an. Der Preis, den man dafür bezahlt, ist dass man sich jetzt auf ein bestimmtes Alphabet festgelegt hat, während vorher das Alphabet einfach implizit als die Menge der bei den Zustandsübergängen vorkommenden Zeichen gegeben war.



Bei der wichtigsten Anwendung vollständiger endlicher Automaten muss man sich ohnehin auf ein bestimmtes Alphabet festlegen, nämlich bei der Konstruktion von Automaten, die genau die Strings akzeptieren, die *nicht* in einer vorgegebenen Menge liegen. Will man etwa einen Automaten konstruieren, der genau die Strings akzeptiert, die *keine* Zahlen im Sinne des ersten Beispiels sind, so muss man trotzdem sagen, über welchem Alphabet diese Strings gebildet werden, weil sonst die Forderung nach endlichen Alphabeten verletzt wäre. Der nebenstehende Automat akzeptiert etwa

genau die Strings über dem Alphabet aus Ziffern, Vorzeichen und Komma, die der vorangegangene Automat gerade nicht akzeptiert. Hier wird von dessen Vollständigkeit Gebrauch gemacht, indem einfach nur die Zustände zu Endzuständen ernannt werden, die vorher keine waren und umgekehrt. Auf den zusätzlichen Zustand F kann man jetzt nicht mehr verzichten.

Kombination mehrerer endlicher Automaten zu einem einzigen

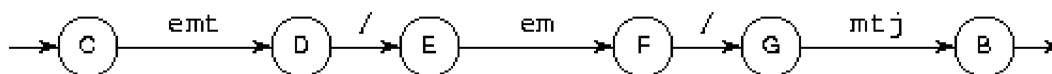
Wir beginnen mit einem Beispiel. Bekanntlich ist ja die anglo-amerikanische Schreibweise von Datumsangaben nicht immer besonders eindeutig: der 01/02/03 könnte der 2. Januar oder der 1. Februar des Jahres 03 sein oder vielleicht auch der 3. Februar 01. Manchmal lässt sich mehr sagen, wenn etwa ein Tagesdatum jenseits der 12 liegt oder eine Jahreszahl vierstellig geschrieben ist. Wir werden jetzt versuchen, mit Hilfe endlicher Automaten die verschiedenen Möglichkeiten zu beschreiben.

Dazu nehmen wir an, dass ein Programm die Datumsangabe schon in ihre Bestandteile gegliedert und dabei festgestellt hat, was jede der Ziffernfolgen möglicherweise bedeuten kann. Dabei ist die Einteilung so gewählt, dass jede Ziffernfolge *eindeutig* in eine der folgenden Klassen gehört, dass aber die meisten Klassen *mehrdeutig* sind hinsichtlich der Entscheidung, ob es sich um eine Tages-, Monats- oder Jahresangabe handelt. Die Klassen sind:

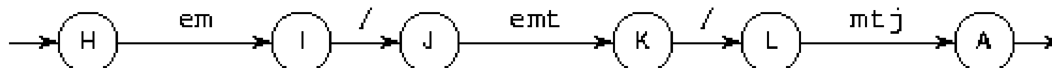
Klasse	Ziffernfolge	Bedeutung
`m`	2-stellige Zahl von 01 bis 12	Tag, Monat oder Jahr
`t`	2-stellige Zahl von 13 bis 31	Tag oder Jahr
`j`	andere 2-stellige Zahl oder 4-stellige Zahl	Jahr
`e`	1-stellige Zahl von 1 bis 9	Tag oder Monat

Als erstes bauen wir je einen endlichen Automaten für die drei Interpretationsmögichkeiten:

Der erste akzeptiert Datumsangaben der Form Tag/Monat/Jahr. Weil das die britische Variante ist, soll sein Endzustand B heißen.

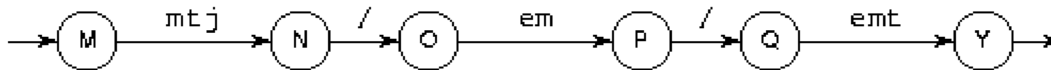


Die Tagesangabe ist nach obiger Tabelle ein String nach einem der Muster `e`, `m` oder `t`; deswegen ist der erste Übergang mit `emt` bezeichnet (das bedeutet also hier nicht den String `emt`!). Der dritte Übergang trägt entsprechend die Beschriftung `em`, weil Monate mit Strings bezeichnet werden, die entweder dem Muster `e` oder dem Muster `m` nach der Tabelle genügen. Schließlich ist der fünfte Übergang mit `mtj` bezeichnet, nach den drei Alternativen, eine Jahreszahl zu schreiben.



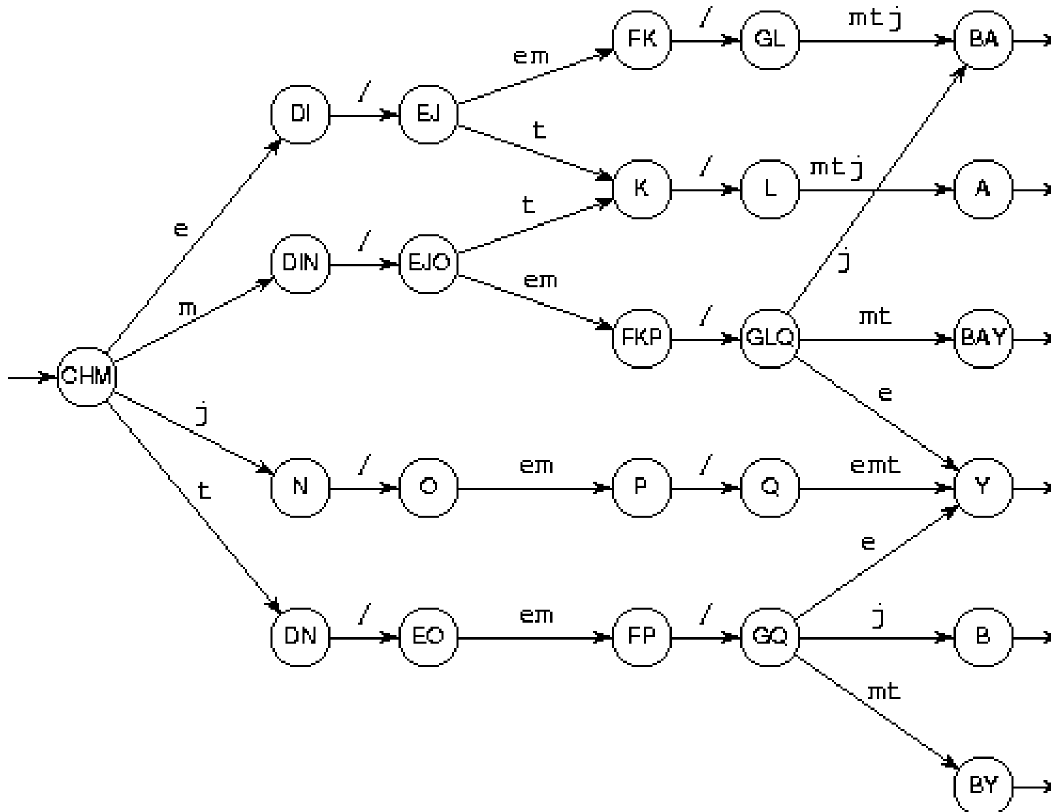
Der zweite Automat akzeptiert Datumsangaben der amerikanischen Form Monat/Tag/Jahr; dementsprechend sind die Beschriftungen des ersten und dritten Übergangs gegenüber dem voranstehenden Automaten vertauscht. Die Bezeichnungen der Zustände, die ja für die akzeptierte Sprache keine Rolle spielen, sind so gewählt worden, dass es keine Überschneidungen zwischen den

beiden Automaten gibt.



Schließlich gibt es noch einen dritten Automaten für Datumsangaben der Form Jahr/Monat/Tag. Hier sind die Beschriftungen der Übergänge gerade die vom ersten Automaten in umgekehrter Reihenfolge.

Jetzt wollen wir die in diesen drei Automaten *insgesamt* vorhandene Information durch einen einzigen Automaten darstellen. Die Zustände



des neuen Automaten sind jeweils Kombinationen von Zuständen der drei alten und geben an, in welchen Zustand man sich befinden könnte, je nachdem, mit welchem der Anfangszustände man gestartet ist. Der Anfangszustand ist also CHM, da man ja nicht wissen kann, welcher der drei Anfangszustände C, H und M gegolten hat. Kommt in diesem Zustand ein `m`, so ist damit kein Stück mehr Klarheit geschaffen, und man ist im Zustand DIN. Kommt hingegen ein `j`, so kann das überhaupt nur der Fall sein, wenn man sich im Zustand M des dritten Automaten befunden hat, und der Folgezustand ist eindeutig N. Mit den beiden anderen Eingabezeichen kann man jeweils einen der drei denkbaren Folgezustände ausschließen und erhält so die Zustände DI und DN, die nur noch jeweils zwei Interpretationen im Sinne der drei alten Automaten zulassen.

Der kombinierte Automat landet mit einem String genau dann in einem der Endzustände, der B (oder A oder Y) in seinem Namen hat, wenn der erste (bzw. zweite bzw. dritte) der einzelnen Automaten mit diesem String in seinem Endzustand B (bzw. A bzw. Y) gelandet wäre. Insofern erfüllt er wirklich die Aufgabe aller drei Automaten gleichzeitig.

Trotzdem sieht jetzt alles viel komplexer aus als zu der Zeit, als es noch drei getrennte Automaten waren. Was ist also damit gewonnen? Vier Dinge:

- Man sieht mit einem Blick, welche Mehrdeutigkeiten entstehen können. Beispielsweise gibt es keine Datumsschreibweise, die als Monat/Tag/Jahr oder Jahr/Monat/Tag, nicht aber als Tag/Monat/Jahr zu interpretierbar ist: das hätte nämlich einem Zustand AY entsprochen, den es nicht gibt.

- Man kann unmittelbar endliche Automaten daraus ableiten, die ganz bestimmte Teilmengen akzeptieren. Beispielsweise wäre ein Automat, der alle eindeutigen Datumsschreibweisen akzeptiert, einfach der, der aus dem der letzten Zeichnung dadurch hervorgeht, dass man die mehrdeutigen Endzustände BA, BAY und BY weglässt und nur die eindeutigen A, Y und B übriglässt. Die Zustände FK und GL werden damit auch überflüssig, aber es war ja kein Automat mit einer minimalen Anzahl von Zuständen versprochen.

- Man sieht damit, dass man aus den endlichen Automaten, die bestimmte Sprachen, also Mengen von Wörtern, akzeptieren, auch Automaten konstruieren kann, die die Vereinigungs-, Durchschnitts- und Differenzmengen akzeptieren. Das Ergebnis des vorangegangenen Abschnitts über Komplementierung erhalten wir hier also noch einmal, allerdings weniger anschaulich, was die tatsächliche Gestalt des Automaten für die komplementäre Sprache angeht.

- Ganz wichtig ist aber die Zeitersparnis: Interessiert uns, in welchen der drei Sprachen ein bestimmter String liegt, so hätten wir mit den drei Automaten am Anfang dazu dreimal so viele Übergänge gebraucht wie mit dem einen Automaten unten, also dreimal so viel Rechenzeit aufgewendet. Wir werden später sehen (allerdings in einem noch nicht geschriebenen Kapitel), dass diese *gleichzeitige* Abarbeitung mehrerer endlicher Automaten das wesentliche Hilfsmittel bei der syntaktischen Analyse in Compilern ist.

Nichtdeterministische endliche Automaten

Wie bei der Einführung der deterministischen Automaten [S.23] schon erwähnt, ergibt sich die einfache Prüfung eines Strings auf Zugehörigkeit zu der vom Automaten akzeptierten Sprache daraus, dass es an genau einer Stelle losgeht und dann mit jedem Zeichen an höchstens einer Stelle weitergeht. Was würde geschehen, wenn man diese Einschränkung fallen lässt, indem man erlaubt

- dass es mehr als einen Anfangszustand gibt,

- dass es von einem Zustand aus mehr als einen Übergang mit demselben Zeichen gibt, und

- dass es Übergänge gibt, die optional spontan durchlaufen werden können, ohne dass ein Zeichen verbraucht wird?

Für diesen jetzt *nichtdeterministischen* endlichen Automaten wird definiert, dass er einen String dann akzeptiert, wenn es wenigstens einen Weg von einem der Anfangszustände zu einem der Endzustände gibt, bei dem die Beschriftungen der Übergänge hintereinander gelesen den String ergeben.

Ohne es zu bemerken, haben wir im vorigen Abschnitt [S.27] schon einen nichtdeterministischen endlichen Automaten kennengelernt: die drei einzelnen Automaten für die drei Datumsnotationen bilden *zusammen* einen nichtdeterministischen Automaten mit drei Anfangszuständen, der alle Datumsnotationen akzeptiert. Der darunter stehende kombinierte Automat akzeptiert genau dieselbe Sprache. Die Art und Weise , wie hier aus einem nichtdeterministischen ein deterministischer endlicher Automat für dieselbe Sprache gemacht wurde, lässt sich ganz genauso auf *jeden*

nichtdeterministischen Automaten anwenden. Von der Tatsache, dass die drei Automaten nicht nur getrennt gestartet haben, sondern stets getrennt geblieben sind, haben wir nämlich an keiner Stelle Gebrauch gemacht. Wir ersparen uns, das im Detail vorzuführen; die Methode ist wirklich genau dieselbe wie bei der Kombination mehrerer endlicher Automaten zu einem einzigen.

Um zu einer gegebenen Sprache einen deterministischen endlichen Automaten anzugeben, genügt es also, einen nichtdeterministischen zu finden und diesen dann mit dem geschilderten Verfahren nachträglich deterministisch zu machen. Darin liegt auch die Hauptbedeutung nichtdeterministischer endlicher Automaten.

Minimierung endlicher Automaten

(wird später geschrieben)

Reguläre Ausdrücke

Das Beispiel [S.2] aus der Einleitung legt nahe, dass reguläre Ausdrücke eine gewisse Ähnlichkeit mit algebraischen Ausdrücken haben, wie wir sie aus der Schule kennen. Es gibt darin ganz genauso Operatoren, nur eben andere: es gibt kein Malzeichen für die Multiplikation, weil man Sprachen nicht multiplizieren kann, dafür gibt es andere Operatoren, die eben auf Sprachen sinnvoll angewendet werden können. Die Operanden dieser Operatoren sind dementsprechend keine Zahlen und auch keine Ausdrücke mit einem Zahlenwert, sondern Sprachen und Ausdrücke, die Sprachen beschreiben. Daran, dass Ausdrücke eben nichts mit Zahlen zu tun haben, muss man sich vielleicht ein wenig gewöhnen. Es gibt aber überhaupt keinen Grund anzunehmen, dass es sich bei Dingen, die mit algebraischen Ausdrucken beschrieben werden, immer um so etwas wie Zahlen handeln muss.

Auch aus der Mengenlehre kennt man ja Ausdrücke, die keine Zahlen, sondern Mengen darstellen. Da Sprachen auch Mengen sind, nämlich Mengen von Wörtern, wird man die Mengenoperationen wie Vereinigung, Durchschnitt und Mengendifferenz unter den Operationen finden, die auf Sprachen operieren. Aber diese reichen nicht aus, um Aussagen über die Gestalt der Wörter einer Sprache zu machen. Dazu müssen wir weitere Operatoren einführen.

Die wichtigste ist dabei die folgende: Unter der *Konkatenation* zweier Wörter versteht man das Wort, das durch Hintereinanderschreiben aus den beiden Wörtern entsteht, z.B. ist die Konkatenation der Wörter `reg` und `ulär` das Wort `regulär`. Die Konkatenation von *mehreren* Wörtern ist entsprechend definiert - es kommt ja nur auf die Reihenfolge der Bestandteile an, nicht aber auf die Reihenfolge der Konkatenationsvorgänge. Hier sieht man wieder, dass wir hier nur Syntax und keine Semantik betreiben: werden etwa die Wörter `mädchen`, `handels`, `schule` konkateniert, so lautet das Ergebnis `mädchenhandelsschule`, unabhängig davon, welche Zusammensetzung zuerst durchgeführt wurde; für die Semantik könnte das schon von Belang sein. Schließlich bemerken wir noch, dass die Konkatenation von null Wörtern das leere Wort ergibt.

Das war jetzt eine Operation auf Wörtern und nicht eine Operation auf Sprachen. Sie lässt sich aber unmittelbar auf ganze Sprachen übertragen (siehe Punkt 3 unten).

Ganz am Anfang dieses Artikels war eine nicht-mathematische Behandlung versprochen worden, ohne formale Definitionen, Sätze und Beweise. Mit der folgenden Definition des Begriffs *regulärer Ausdruck*, der ja bisher nur in Form eines Beispiels vorkam, wollen wir einmal eine Ausnahme machen. Zum einen mag es ganz interessant sein, zu sehen, *wie* man so etwas präzis definiert, zum zweiten ist ein gewisses Maß an Präzision notwendig, um Operationen auf Sprachen nicht mit Operationen auf Wörtern zu verwechseln, wie das eben schon beinahe geschehen wäre, zum dritten

schließlich kann diese Definition auch gleichzeitig als Beispiel für eine Definition einer nicht-regulären Sprache dienen, wie sie für spätere Erläuterungen (die noch nicht geschrieben sind) gebraucht wird.

Es handelt sich um eine *rekursive* Definition. Das bedeutet, dass der zu definierende Begriff selbst in der Definition verwendet wird. Eine solche rekursive Definition erhält immer Teile, bei denen das nicht vorkommt (hier die Punkte 1 und 2), aus denen dann die anderen aufgebaut werden. Das Beispiel nach der Definition wird das erläutern.

1. Ø ist ein regulärer Ausdruck, der die leere Sprache beschreibt, d.h. die Sprache, die kein Wort enthält, nicht einmal das leere.
2. Für jedes Zeichen *a* ist *a* ein regulärer Ausdruck, der die Sprache beschreibt, die genau ein Wort enthält, wobei dieses Wort genau aus dem Zeichen *a* besteht.
3. Sind E_1 und E_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 beschreiben, so ist $(E_1)(E_2)$ ein regulärer Ausdruck, der die Sprache beschreibt, deren Wörter jeweils aus einem Wort aus L_1 und einem Wort aus L_2 konkateniert sind.
4. Ist *E* ein regulärer Ausdruck, der die Sprache *L* beschreibt, so ist (*E*)* ein regulärer Ausdruck, der die Sprache beschreibt, deren Wörter sich als endliche (auch leere) Konkatenationen von Wörtern aus *L* schreiben lassen. Diese Sprache heißt auch der Kleene'sche Abschluss von *L* nach Stephen Cole Kleene (http://www-groups.dcs.st-and.ac.uk/history/Mathematicians/Kleene.html) (1909-1994); Aussprache: ['kli:ni].
5. Sind E_1 und E_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 beschreiben, so ist $(E_1)|(E_2)$ ein regulärer Ausdruck, der die Sprache beschreibt, deren Wörter in L_1 oder in L_2 vorkommen, also die Vereinigungsmenge der beiden Sprachen.
6. Sind E_1 und E_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 beschreiben, so ist $(E_1)\&(E_2)$ ein regulärer Ausdruck, der die Sprache beschreibt, deren Wörter in L_1 und in L_2 vorkommen, also den Durchschnitt der beiden Sprachen.
7. Sind E_1 und E_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 beschreiben, so ist $(E_1)\backslash(E_2)$ ein regulärer Ausdruck, der die Sprache beschreibt, deren Wörter in L_1, aber nicht in L_2 vorkommen, also die Differenzmenge der beiden Sprachen.

Die Klammern dürfen weggelassen werden, wenn trotzdem Eindeutigkeit besteht, zum Beispiel bei der Konkatenation oder Vereinigung von mehr als zwei Sprachen nach Regel 3 bzw. 5. Außerdem bindet der Stern für den Kleene'schen Abschluss stärker als die anderen Operatoren und das Hintereinanderschreiben für die Konkatenation stärker als die explizit hingeschriebenen zweistelligen Operatoren. Das Zeichen \ klammert nach links: $K\backslash L\backslash M$ bedeutet also $(K\backslash L)\backslash M$. Man kann diese Klammerungs- und Vorrangregeln auch direkt in die Definition aufnehmen, was aber zur leichten Verständlichkeit nicht unbedingt beiträgt; dafür lassen sich aus so geschriebenen Definitionen leichter auf automatischem Weg Programme generieren, die Ausdrücke in ihre Bestandteile zerlegen.

Als Beispiel wollen wir die Sprache aus dem Beispiel [S.2] in der Einleitung mit einem so definierten regulären Ausdruck beschreiben. Dabei gehen wir schrittweise vor:

- Die vorkommenden Zeichen (genauer: die Sprachen, die nur ein Wort der Länge 1 enthalten) werden nach Regel 2 durch reguläre Ausdrücke beschrieben, die einfach aus dem jeweiligen Zeichen bestehen.

- Die Sprachen, die aus lauter einzelnen Zeichen (genauer: aus lauter Wörtern der Länge 1) bestehen, werden mit Regel 5 daraus gebildet. Das sind hier:

    - die Vorzeichen: +|−

    - die Ziffern: 0|1|2|3|4|5|6|7|8|9

- Ketten beliebiger Länge können daraus nach Regel 4 mit Hilfe des Kleene'schen Abschlusses gebildet werden:

    - Ziffernketten, evtl. auch leer: (0|1|2|3|4|5|6|7|8|9)*

    - Daraus mit Regel 3 die Ziffernketten der Mindestlänge 1:
      (0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*

- der leere String, der für optionale Teile gebraucht wird, ebenfalls nach Regel 4: Ø*
  Das erschließt sich zugegebenermaßen erst beim Nachdenken; in der Praxis erweitert man oft deswegen die Definition der regulären Ausdrücke um ein Zeichen für den leeren String, meist das kleine griechische Epsilon, oder man erlaubt in bestimmten Zusammenhängen, den leeren String einfach durch sich selbst zu bezeichnen.

- ein optionales Vorzeichen, nach dem Voranstehenden: Ø*|+|−
  oder mit der eben vorgeschlagenen Erweiterung: |+|−

- und das ganze Ding, vor allem mittels Regel 3 gebildet:
  (Ø*|+|−)(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*(Ø*|,(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)

Wie man sieht, stört es kein bisschen, dass die Definition den zu definierenden Begriff selbst enthält, schließlich wird ja ein regulärer Ausdruck höchstens unter Berufung auf einen bereits gebildeten *anderen*, nie aber auf sich selbst konstruiert. Manchmal wird übrigens bei einer rekursiven Definition explizit dazugeschrieben, dass *nur* das damit definiert ist, was sich durch endlich viele Anwendungen der Regeln bilden lässt; aber auch wenn es wie hier nicht dabeisteht, ist es immer so gemeint.

Der reguläre Ausdruck ist viel länger geworden als in der Einleitung [S.2] . In der praktischen Anwendung bevorzugt man nämlich abkürzende, übersichtliche Notierungen, während in der Theorie möglichst wenige verschiedene Operatoren eingesetzt werden, um die Herleitung von Sätzen einfacher zu gestalten.

Diesem Ziel dient auch die folgende Festlegung: Wir nennen den regulären Ausdruck *einfach*, wenn er die Operatoren Durchschnitt und Mengendifferenz nicht enthält; ein *verallgemeinerter* regulärer Ausdruck ist einer, der diese beiden Operatoren auch enthalten darf. Es wird sich herausstellen, dass man mit verallgemeinerten regulären Ausdrücken keine Sprachen beschreiben kann, die man nicht auch mit einfachen regulären Ausdrücken beschreiben kann. Die mit regulären Ausdrücken beschreibbaren Sprachen heißen *reguläre Sprachen*. Das sind gerade diejenigen Sprachen, die von deterministischen endlichen Automaten akzeptiert werden, womit der Zusammenhang des vorangegangenen Abschnittes zum Thema des gesamten Artikels endlich hergestellt wäre.

Die hier eben aufgestellten Behauptungen bilden einen der Kernpunkte der Theorie und werden deshalb hier noch einmal in fetter Schrift wiederholt und, wenn schon nicht bewiesen, so doch wenigstens plausibel gemacht:

- **Die mit verallgemeinerten regulären Ausdrücken beschreibbaren Sprachen lassen sich auch mit nicht notwendig deterministischen endlichen Automaten beschreiben.**

  Für die nicht-rekursiven Teile der Definition der regulären Ausdrücke ist das unmittelbar zu sehen: Die Sprache, die nur das leere Wort enthält und die leere Sprache werden jeweils durch einen Automaten mit nur einem Zustand dargestellt, der *dann* auch Endzustand ist, wenn wenigstens das leere Wort akzeptiert werden soll. Die Sprachen, die nur ein Wort der Länge 1 enthalten, werden mit einem Automaten mit zwei Zuständen und einem Übergang dargestellt.

  Hat man schon einen endlichen Automaten für eine Sprache, so ergibt sich der Automat für ihren Kleene'schen Abschluss dadurch, dass man von allen Endzuständen einen Spontanübergang [S.29] auf alle Anfangszustände hinzufügt und alle Anfangszustände auch zu Endzuständen macht. Ebenso baut man aus zwei Automaten für zwei Sprachen einen neuen für die Konkatenation der beiden Sprachen, indem man alle Endzustände des ersten Automaten durch Spontanübergänge mit den Anfangszuständen des zweiten verbindet; die Endzustände des ersten Automaten bleiben dabei *nicht* Endzustände des neu konstruierten Automaten.

  Den Rest haben wir schon im Abschnitt über endliche Automaten gesehen. Die Mengenoperationen werden direkt mit Hilfe kombinierter [S.27] endlicher Automaten dargestellt.

- **Die mit nichtdeterministischen endlichen Automaten beschreibbaren Sprachen lassen sich auch mit deterministischen endlichen Automaten beschreiben.**

  Das war bei den nichtdeterministischen [S.29] endlichen Automaten schon kurz erläutert worden.

- **Die mit deterministischen endlichen Automaten beschreibbaren Sprachen lassen sich auch mit einfachen regulären Ausdrücken beschreiben.**

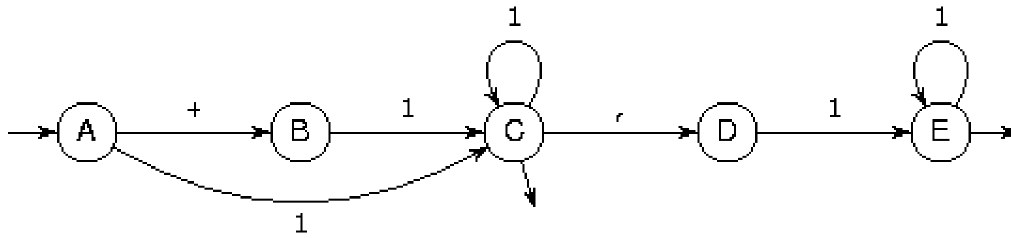
  Das ist die schwierigste der vier Aussagen: ihr wird der nächste Abschnitt [S.33] dieses Artikels gewidmet sein.

- **Die mit einfachen regulären Ausdrücken beschreibbaren Sprachen lassen sich auch mit verallgemeinerten regulären Ausdrücken beschreiben.**

  Das ist natürlich trivial, weil einfache reguläre Ausdrücken auch verallgemeinerte reguläre Ausdrücken *sind*. Es steht hier nur der Vollständigkeit halber, damit sich der Kreis zwischen den vier Darstellungen für reguläre Sprachen schließt.

Diese Zusammenstellung legt nahe, dass der Weg, aus einem verallgemeinerten regulären Ausdruck einen einfachen zu erhalten, über den endlichen Automaten führt. So sehr das wie ein Umweg aussieht, ist es doch tatsächlich oft einer der kürzesten Wege zum Ziel.

Reguläre Ausdrücke für endliche Automaten



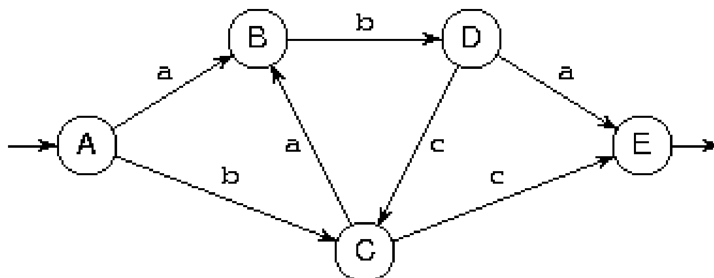
Der endliche Automat, der als unser erstes Beispiel diente, lässt sich ziemlich einfach in einen regulären Ausdruck umschreiben. Drei wiederholt einsetzbare Schritte reichen dafür aus, bei denen der Automat jeweils vereinfacht wird, wobei allerdings die Übergänge mit regulären Ausdrücken statt mit einzelnen Zeichen versehen werden:

1. Hat ein Zustand die Eigenschaft, dass nur ein Übergang von außen hineingeht und ein Übergang nach außen herausführt, so wird der Zustand mit seinen beiden Übergängen durch einen neuen Übergang ersetzt, der mit der Konkatenation der Ausdrücke auf den alten Übergängen markiert wird.

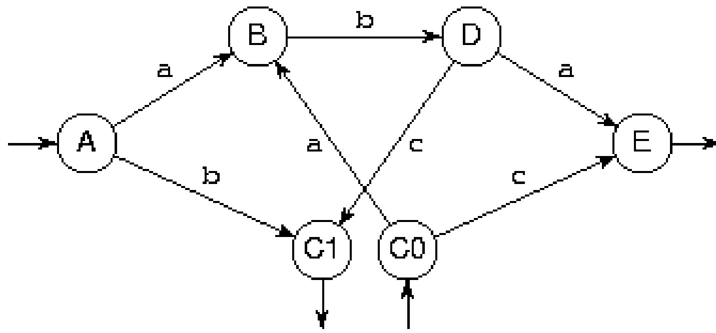
2. Hat ein Zustand eben diese Eigenschaft, aber zusätzlich eine Schlinge, also einen Übergang auf sich selbst, so wird der Zustand mitsamt seiner Schlinge und seinen beiden äußeren Übergängen durch einen neuen Übergang ersetzt, der mit der Konkatenation der Ausdrücke auf den äußeren Übergängen markiert wird, zwischen die der Kleene'sche Abschluss des Ausdrucks auf der Schlinge eingefügt wird.

3. Gibt es zwei parallele Übergänge, so werden sie durch einen einzigen mit der Vereinigung der Sprachen auf den Übergängen ersetzt.

Man ersetzt also den Zustand B mit seinen Übergängen nach Regel 1 durch einen Übergang von A nach C mit `+1`, ebenso den Zustand D mit seinen Übergängen durch einen Übergang von C nach E mit `,1`. Nach Regel 3 können die beiden parallelen Übergänge von A nach C durch einen einzigen mit `+1|1` ersetzt werden. Nach der ersten Ersetzung erfüllt der Zustand E die Voraussetzung für Regel 2, wenn man die Außenwelt, in die der Pfeil ganz rechts führt, wie einen Zustand behandelt: Man kann den Weg von C über E nach außen durch einen Übergang direkt von C nach außen ersetzen, der mit `,11*` bezeichnet ist. Es bleibt dem Leser überlassen, dieses Beispiel zu Ende zu führen, indem auch noch die restlichen Zustände eliminiert werden.



Für viele in der Praxis vorkommenden Anwendungen regulärer Ausdrücke reichen diese Regeln aus, leider aber nicht für den allgemeinen Fall. Ihre Anwendung setzt nämlich voraus, dass die Zyklen, also die Wege, die von einem Zustand nach einem oder mehreren Schritten wieder in denselben Zustand führen, sich auf einfache Wege reduzieren lassen. Im Beispiel links ist das nicht der Fall: keiner der Zustände lässt sich hier mit den obigen Regeln eliminieren.



In so einem Fall splittet man einen Zustand auf; wir nehmen hier beispielsweise den Zustand C, den wir in C0 und C1 splitten. Einer der beiden neuen Zustände erhält alle hineingehenden Übergänge und wird Endzustand, hier C1, der andere alle herausgehenden Übergänge und wird Anfangszustand, hier C0. Das Ganze ist jetzt nicht als nichtdeterministischer Automat zu lesen, wie es die Zeichnung eigentlich nahelegt, sondern nur als gemeinsame Zeichnung für vier verschiedene deterministische endliche Automaten:

- Der Automat, dessen Anfangszustand A und dessen Endzustand E ist, beschreibt die Wege im ursprünglichen Automaten, die nicht bei C vorbeikommen. Die Sprache ist `aba`.
- Der Automat, dessen Anfangszustand A und dessen Endzustand C1 ist, beschreibt die Wege von A nach C im ursprünglichen Automaten, die unterwegs nicht bei C vorbeikommen. Die Sprache ist `abc|b`.
- Der Automat, dessen Anfangszustand C0 und dessen Endzustand E ist, beschreibt die Wege von C nach E im ursprünglichen Automaten, die unterwegs nicht bei C vorbeikommen. Die Sprache ist `aba|c`.
- Der Automat, dessen Anfangszustand C0 und dessen Endzustand C1 ist, beschreibt die Wege von C nach C im ursprünglichen Automaten, die unterwegs nicht bei C vorbeikommen. Die Sprache ist `abc`.

Die Gesamtheit der Wege von A nach E im ursprünglichen Automaten besteht offenbar aus allen Wegen, die überhaupt nicht bei C vorbeikommen, sowie aus allen Wegen, die von A direkt (d.h. nicht über C) nach C führen, danach beliebig oft direkt von C nach C und schließlich direkt von C nach E. Damit sieht man unmittelbar die Sprache `aba|(abc|b)(abc)*(aba|c)`.

In komplizierten Fällen [S.37] wird man mehr als einmal einen Zustand splitten müssen. Man kann aber leicht zeigen, dass die resultierenden deterministischen endlichen Automaten zwar immer zahlreicher, aber auch jeweils immer einfacher werden, so dass das Verfahren terminiert.

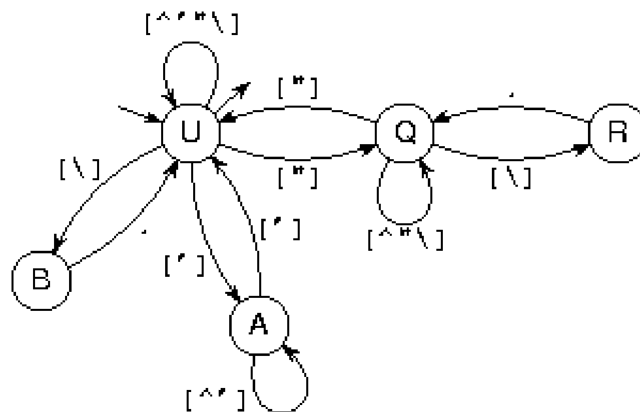
Beispiele

Mehrfach maskierter Text (zweiter Versuch)

Wir wenden uns noch einmal [S.20] der Aufgabe zu, einen regulären Ausdruck für alle jene Unix-Kommandozeilen zu konstruieren, bei denen alle mit Apostroph oder Anführungszeichen begonnenen Texte ordnungsgemäß abgeschlossen sind. Die Spezifikation entnehmen wir der ersten Beschreibung [S.20] dieser Aufgabe. Jetzt erstellen wir die dort angekündigte Übersicht über "alle Möglichkeiten, wie die verschiedenen Maskierungszeichen aufeinander einwirken":

	Situation	Beurteilung des Textes bei Textende in dieser Situation	Situation nach			
			`[']`	`["]`	`[\]`	`[^'"\]`
U	unmaskierter Text	Text in Ordnung	A	Q	B	U
A	in Apostrophen	schließender Apostroph fehlt	U	A	A	A
B	nach Backslash außen	Text endet mit Backslash als Maskierungszeichen	U	U	U	U
Q	in Anführungszeichen	schließendes Anführungszeichen fehlt	Q	U	R	Q
R	nach Backslash in Anführungszeichen	schließendes Anführungszeichen fehlt	Q	Q	Q	Q

Diese Tabelle liest sich so: anfangs befinden wir uns im unmaskierten Text, also an einer Stelle, an der alle Apostrophe und Anführungszeichen noch nicht geöffnet oder aber schon wieder wieder geschlossen sind; diesen Fall kürzen wir mit U ab. Kommt dann eines der drei Maskierungszeichen, so



verändert sich auf jeden Fall die Situation; die drei neuen Situationen kürzen wir mit A (für Apostroph), B (für Backslash) und Q (für Anführungszeichen, engl. "quote") ab. Die folgenden drei Tabellenzeilen machen dann Aussagen über die jeweils dort entstandene Situation. Man mache sich die übrigen Tabellenzeilen ebenso klar.

Wenn man eine solche Tabelle anlegt, stellt man auf jeden Fall sicher, dass man keine möglicherweise auftretende Situation vergisst. Im Zweifelsfall ist es besser, mehr verschiedene Situationen zu unterscheiden; man kann sie ja nachträglich noch zusammenfassen, wenn sich herausstellt, dass in ihnen jeweils dieselben nächsten Zeichen dieselben Reaktionen auslösen.

Diese Tabelle können wir natürlich auch grafisch darstellen, und mit unserem mittlerweise erworbenen Wissen über endliche Automaten erkennen wir, dass eine Tabelle wie diese nur eine alternative Schreibweise für einen endlichen Automaten darstellt. Die Umwandlung eines endlichen Automaten in einen regulären Ausdruck ist für uns inzwischen eine leichte Standardaufgabe: in diesem einfachen Fall genügt sogar das vereinfachte Verfahren, das im Abschnitt [S.33] über diese Umwandlung zuerst dargestellt wurde. In wenigen Schritten sind wir bei einem regulären Ausdruck:

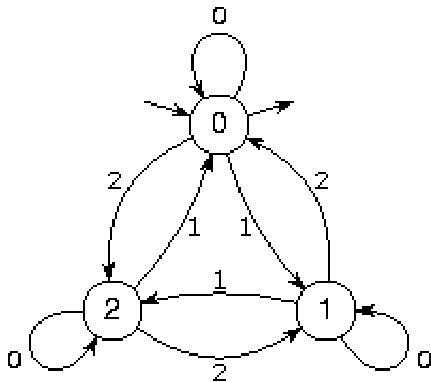
- von Q einmal über R nach Q zurück: `[\].`
- von Q nach Q, alle Wege nicht über U: `([^"\]|[\].)*`
- von U einmal über Q nach U zurück: `["]([^"\]|[\].)*["]`

- von U nach U, alle Wege: `([^'"\]|[\].|['][^']*[']|["]([^"\]|[\].)*["])*`

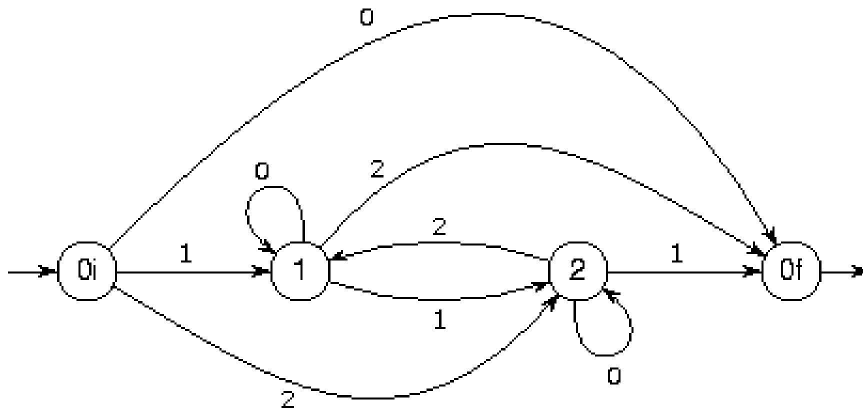
Der letzte Ausdruck ist dann, wie man sieht (da U einziger Anfangs- und Endzustand ist), gleich der gesuchte reguläre Ausdruck. Nicht alle Maskierungen mit eckigen Klammern sind notwendig; es ist aber sicherer, sie bei solchen Zeichen, die wie diese in verschiedenen Umgebungen (Shells, Programmiersprachen) zur Maskierung verwendet werden, immer dazuzufügen.

Durch drei teilbare Dezimalzahlen

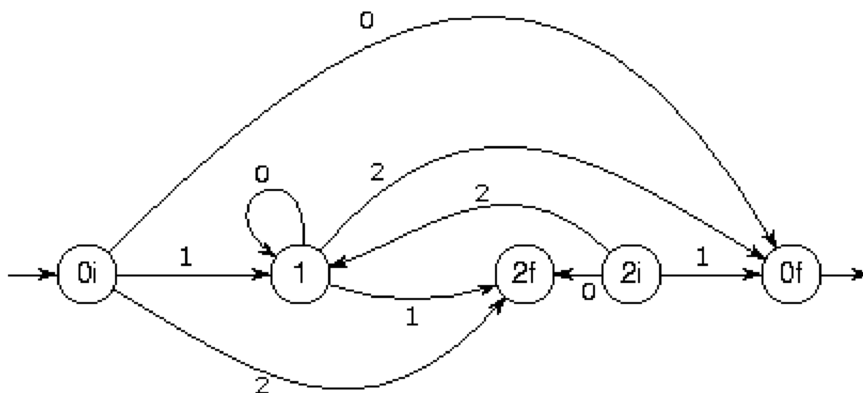
Wir haben eingangs durch Konstruktion [S.24] eines endlichen Automaten gezeigt, dass die durch drei teilbaren Dezimalzahlen eine reguläre Sprache bilden. Jetzt wollen wir den dazugehörigen regulären Ausdruck angeben. Wir beginnen dabei zunächst mit der schon dort durchgeführten Vereinfachung, die Ziffern 0, 1 und 2 stellvertretend für alle Dezimalziffern zu betrachten und den leeren String auch zuzulassen; beide Vereinfachungen lassen wir erst am Schluss der Konstruktion wegfallen.



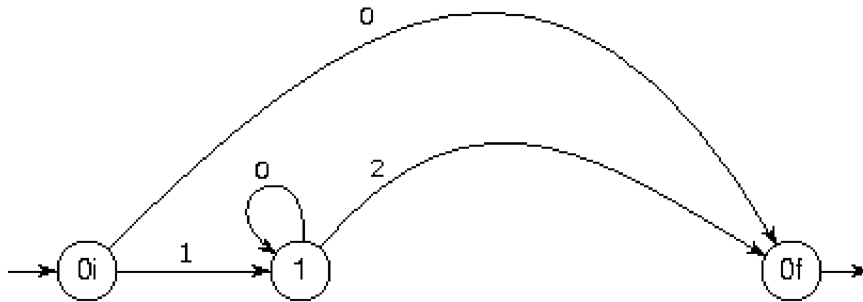
Das ist zunächst der endliche Automat, den wir schon betrachtet haben; die von ihm akzeptierte Sprache heiße *L*. Er ist zu eng vermascht, als dass wir mit dem vereinfachten Verfahren aus dem Abschnitt [S.33] über die Konstruktion regulärer Ausdrücke aus endlichen Automaten etwas anfangen könnten. Also müssen wir zunächst einen Zustand splitten [S.34] ; wir wählen Zustand 0. Das ist sehr naheliegend, weil das der Anfangs- und Endzustand ist, so dass die Wege aus dem Anfangs- in den gesplitteten und die aus dem gesplitteten in den Endzustand einfach wegfallen.



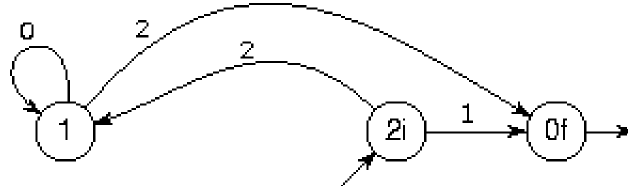
Hier haben wir den Zustand 0 in zwei Zustände gesplittet: in einen, 0i, für die herausgehenden und einen anderen, 0f, für die hineingehenden Übergänge. Ist L_1 die Sprache, die vom nebenstehenden Automaten akzeptiert wird, so ist $L{=}L_1{}^*$. Dieser Automat sieht nicht sehr ähnlich zum voranstehenden aus, weil er zur besseren Übersichtlichkeit auseinandergezogen wurde und daher die beiden Teile des alten Zustands 0 nicht mehr beieinander liegen. Es kommt aber nur auf die Übergänge an, und man verifiziert leicht, dass es noch dieselben sind wie vorher.



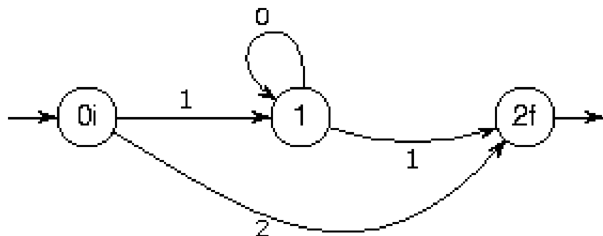
Weil auch der Automat vom letzten Schritt noch keine unmittelbare Darstellung als regulärer Ausdruck zulässt, muss abermals ein Zustand in derselben Weise gesplittet werden. Diesmal wählen wir den Zustand 2, den wir in 2i und 2f splitten. Sei nun L_2 die Sprache, die auf dem Weg von 0i nach 0f akzeptiert wird, L_3 die von 0i nach 2f, L_4 die von 2i nach 0f und L_5 die von 2i nach 2f. Dann ist, wie oben erläutert [S.34] , $L_1{=}L_2|L_3L_5{}^*L_4$.



Das ist der Teil des Automaten, der für die Sprache L_2 relevant ist; die übrigen Zustände werden auf dem Weg von 0i nach 0f ohnehin nicht erreicht. Man sieht unmittelbar, dass L_2=0|10*2.



Analog der Teil des Automaten für die Sprache L_4. Es ist also L_4=1|20*2.



Schließlich dasselbe für L_3=2|10*1. Für L_5 sparen wir uns die Extrazeichnung, weil schon im Ausgangsbild alles so schön nah beieinander liegt, dass man sieht L_5=0|20*1.

Jetzt muss man nur noch alle Formeln ineinander einsetzen und gewinnt so einen regulären Ausdruck für L:

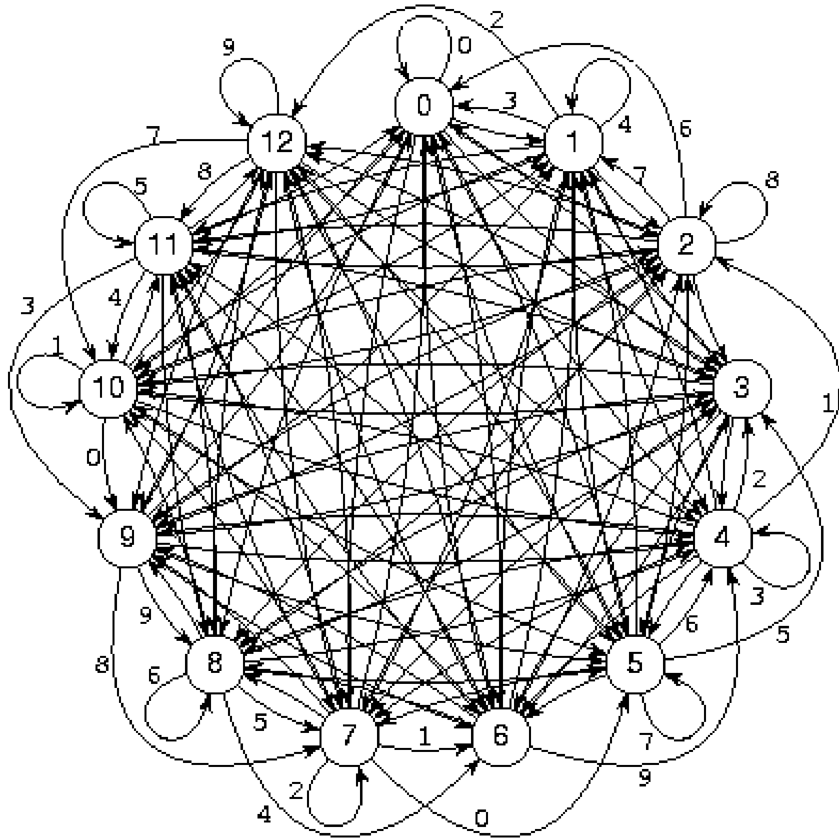
$L = L_1{}^* =$
$= (L_2 | L_3 L_5{}^* L_4)^* =$
= (0|10*2|(2|10*1)(0|20*1)*(1|20*2))*

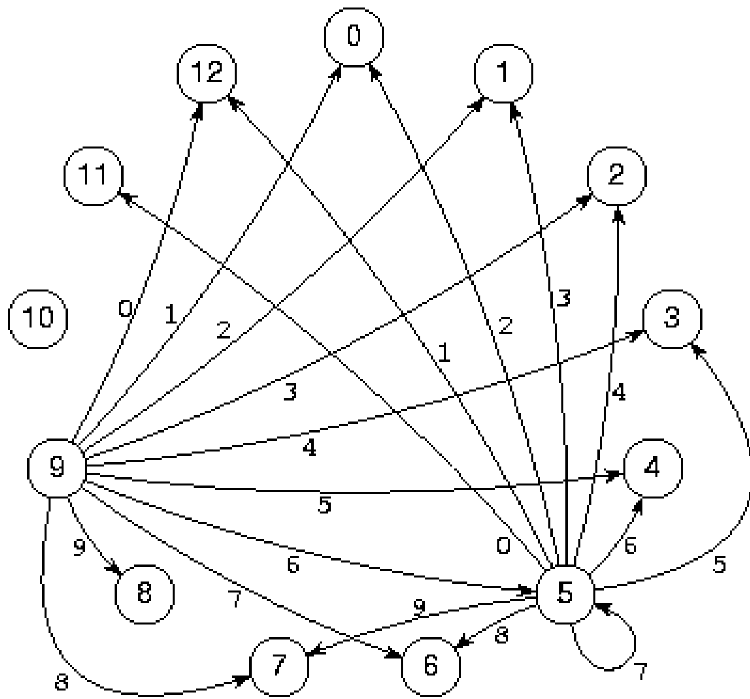
Schließlich lassen wir die beiden Vereinfachungen vom Anfang weg. Das Zeichen (nicht der Zustand!) 0 beim Automaten stand für eine der Ziffern 0, 3, 6 oder 9, also müssen wir die 0 im regulären Ausdruck durch [0369] ersetzen und analog die 1 durch [147] und die 2 durch [258]. Den leeren String schließen wir aus, indem ganz außen der Stern durch ein Pluszeichen ersetzt wird. Somit lautet der fertige reguläre Ausdruck für die durch drei teilbaren Dezimalzahlen:

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

Durch dreizehn teilbare Dezimalzahlen

Hier ist dasselbe Beispiel für einen größeren Teiler, nämlich 13, um zu demonstrieren, dass das Funktionieren der Konstruktion nicht davon abhängt, dass es eine einfache "Probe", etwa eine Quersummenbildung, für die Teilbarkeit gibt.


0
1
2
3
4
5
6
7
8
9
10
11
12


0
1
2
3
4
5
6
7
8
9
10
11
12

Die erste Zeichnung stellt den gesamten endlichen Automaten dar, wobei im Inneren die Bezeichnungen der Zustandsübergänge weggelassen wurden, da man sie ohnehin nicht hätte lesen können. Man sieht eigentlich nur ein wildes Knäuel, in welchem von jedem Zustand aus zehn Übergänge für die möglichen zehn Ziffern ausgehen. Die zweite Zeichnung zeigt anhand aller von zwei willkürlich ausgewählten Zuständen ausgehenden Übergänge, wie sich die Gesamtheit der Übergänge aus dreizehn fächerartigen Bündeln von je zehn Übergängen zusammensetzt. In der zweiten Zeichnung kann man sich auch von der Richtigkeit der dargestellten Übergänge leicht überzeugen. Beispielsweise sagt der Bogen rechts außen aus: Hängt man an eine Zahl, die bei Division durch 13 den Rest 5 lässt, die Ziffer 5 an, so entsteht eine, die bei Division durch 13 den Rest 3 lässt. Ist nämlich die Zahl vorher $13k+5$, so entsteht daraus $10\cdot(13k+5)+5 = 130k+55 = 13\cdot(10k+4)+3$.

Die Konstruktion eines regulären Ausdrucks dazu überlassen wir gerne dem geneigten Leser - der muss dazu allerdings schon *sehr* geneigt sein.

Weder Fisch noch Fleisch

Warnung: Dies ist ein eher pathologisches Beispiel und wohl kaum zu irgend einer praktischen Anwendung geeignet. Es ist aber insofern lehrreich, als die Grenzen der Darstellbarkeit regulärer Sprachen durch die ERE-Notation ausgereizt werden.

Einen regulären Ausdruck anzugeben, mit dem der String `Fisch` und der String `Fleisch` und sonst nichts beschrieben wird, ist ganz einfach: entweder einfach `Fisch|Fleisch` oder, unter Berücksichtigung der Ähnlichkeit der beider Wörter, `F(le)?isch`. Wie aber konstruiert man einen regulären Ausdruck für "weder Fisch noch Fleisch", also für alle Strings außer diesen beiden?

Mit regulären Ausdrücken kann man umgehen wie mit algebraischen Ausdrücken in der Mathematik: es *sind* algebraische Ausdrücke, deren Umformungsregeln im Theorie-Teil dieses Artikels stehen werden. Die Negation ¬ gehört zwar nicht zu den Operatoren, die in der ERE-Notation vorkommen können; das soll uns aber nicht hindern, sie für die Konstruktion eines regulären Ausdrucks bei den Zwischenschritten heranzuziehen.

Als erstes überlegen wir uns, wie die Negation eines einzelnen Strings, etwa des Strings `sch`, aussieht. Ist etwas nicht der String `sch`, so gibt es folgende Möglichkeiten:

- Es fängt mit einem anderen Zeichen als `s` an.
- Es fängt zwar mit `s` an, geht aber mit einem anderen Zeichen als `c` weiter.
- Es fängt zwar mit `sc` an, geht aber mit einem anderen Zeichen als `h` weiter.
- Es fängt zwar mit `sch` an, geht aber danach weiter.
- Es ist ein Anfang von `sch`, der aber zu früh abbricht, also der leere String, `s` oder `sc`.

In der Sprache der regulären Ausdrücke sieht das so aus:

```
¬(sch)=([^s]|s[^c]|sc[^h]|sch.).*|(sc?)?
```

Analog dazu, aber unter Verwendung des Obigen:

```
¬(eisch)=([^e]|e[^i]).*|ei¬(sch)|e?
```

Jetzt ein klein wenig komplizierter:

```
¬((le)?isch) = l(¬(eisch))|i(¬(sch))|([^le].*)? =
= l(([^e]|e[^i]).*|ei¬(sch)|e?)|i(¬(sch))|([^le].*)? =
= (l([^e]|e[^i]).*|lei¬(sch)|le?)|i(¬(sch))|([^le].*)? =
= (l([^e]|e[^i]).*|(le)?i¬(sch)|le?)|([^le].*)?
```

Damit haben wir das Material für den gewünschten regulären Ausdruck:

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

Wir lernen daraus zum einen, dass die regulären Ausdrücke auch in ERE-Notation das Versprechen einhalten, dass mit jeder darstellbaren Menge von Zeichenreihen auch ihr Komplement darstellbar ist, zum anderen aber, dass das nicht gerade die Stärke dieser Notation ist. Die Programme, die mit regulären Ausdrücken arbeiten, haben deswegen Möglichkeiten, auch außerhalb des regulären Ausdrucks anzugeben, dass man das Komplement wünscht, z.B. die Option `-v` beim `grep`-Kommando.

Der wesentliche Grund für diese Beschränkung der regulären Ausdrücke dürfte darin liegen, dass man sie zur Mustersuche einsetzen will und es sinnlos ist, zu fragen, was man an einer bestimmten Stelle *nicht* gefunden hat. Für globale Fragestellungen vom ja/nein-Typ genügen aber Optionen außen im Kommando meistens.

Weitere theoretische Betrachtungen

Die folgenden Themen sollen in eine spätere Version des Artikels aufgenommen werden:

- Abschlusseigenschaften
- nicht-einbettende Grammatiken
- Zusammenhänge mit der Algebra
- Zusammenhänge mit dem Relationenkalkül
- axiomatische Behandlung
- gsm mappings