Visualisierung regulärer Ausdrücke

Martin Häcker und Felix Schwarz
April 2003

Zusammenfassung

Diese Ausarbeitung stellt verschiedene Techniken vor, mit deren Hilfe sowohl Einsteiger als auch fortgeschrittene Benutzer reguläre Ausdrücke besser und schneller verstehen und verwenden können.

I. Einleitung und Terminologie

Der Begriff eines „regulären Ausdrucks“ wird in unterschiedlichen Kontexten mit unterschiedlicher Bedeutung verwendet. Unter einem regulären Ausdruck verstehen wir Sprachkonstrukte von Programmiersprachen zur Erkennung von Textmustern, wie sie z.B. in Perl, awk, sed, Java und zahlreichen anderen Unix-Programmen verwendet werden.

Mit Hilfe von regulären Ausdrücken ist häufig das Parsing von Text sehr einfach zu programmieren, daher werden sie schon sehr lange v.a. in Unix-Programmen verwendet, wo man ihre Flexibilität und Ausdruckskraft sehr schätzt. Allerdings konnte sich bisher keine Standardisierung der Syntax durchsetzen (trotz eines entspr. POSIX-Standards), so dass sie im Wesentlichen von der Verbreitung der Programme definiert wird, die reguläre Ausdrücke verarbeiten. Im weiteren werden wir daher die Syntax von Perl 5 verwenden, Einführungen und Syntaxreferenzen finden sich u.a. in [1], [9] [10].

Das Problem

„Some people, when confronted with a problem, think »I know, I'll use regular expressions.« Now they have two problems.“

Jamie Zawinski, in comp.lang.emacs (from the fortune file)

```
# don't touch this stuff down here or you'll break the rfc822 matcher.
__END__
(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\n
\015()]|\\[^\x80-\xff])*\))*\))*(?:(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\
xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"(?:[^\\\x80-\xff\n\015"
]|\\[^\x80-\xff])*")(?:(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xf
f]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff])*\))*\)*\.(?:[\040\t]|\((?:[
^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\
xff])*\))*\))*(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;
:".\\\[\]\000-\037\x80-\xff])|"(?:[^\\\x80-\xff\n\015"]|\\[^\x80-\xff])*"))
*(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\
n\015()]|\\[^\x80-\xff])*\))*\))*@(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\
\[^\x80-\xff]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff])*\))*(?:[^(\04
0)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-
\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])(?:(?:[\040\t]|\((?
:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80
-\xff])*\))*\)*\.(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\(
(?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff])*\))*\))*(?:[^(\040)<>@,;:".\\\[\]
\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\[(?:[^\\
\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\]))*|(?:[^(\040)<>@,;:".\\\[\]\000-\0
37\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"(?:[^\\\x80-\xf
f\n\015"]|\\[^\x80-\xff])*")(?:[^()<>@,;:".\\\[\]\x80-\xff\000-\010\012-\03
7]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\n\015()]|\
\[^\x80-\xff])*\))*\)|"(?:[^\\\x80-\xff\n\015"]|\\[^\x80-\xff])*")*<(?:[\04
0\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\n\015()]
|\\[^\x80-\xff])*\))*\))*(?:@(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x
80-\xff]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff])*\))*\))*(?:[^(\040)<>@
,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]
```

Abbildung 1: Ausschnitt aus einem regulären Ausdruck [11] zur Erkennung korrekter E-Mail Adressen

Bisherige Programme zur Bearbeitung von regulären Ausdrücken (insbes. Texteditoren) arbeiten auf der reinen ASCII-Repräsentation des regulären Ausdrucks und beherrschen nur ein einfaches Syntax-Highlighting. Wie man oben sehr schön sieht, versagt diese Visualisierungsmethode aber bei komplexeren Ausdrücken sehr schnell. Offensichtlich werden weitergehende Visualisierungsmechanismen benötigt.

Menschen, die sich mit regulären Ausdrücken beschäftigen, lassen sich in drei Gruppen einteilen: Einsteiger, Programmierer und Maintainer.

Einsteiger beschäftigen sich erst seit kurzem mit der Materie und haben daher häufig noch Probleme mit der Verständnis der Elemente und der Strukturen eines vorgegebenen regulären Ausdrucks. Weiter-

hin ist ihnen oft nicht bewusst, wie die einzelnen Teile zusammenwirken bzw. wie ein regulärer Ausdruck im Detail arbeitet.

Programmierer hingegen erstellen einen neuen regulären Ausdruck auf der Basis eines Textes und einem gewünschten Ergebnis. Zunächst identifizieren sie die erwünschten Muster und formulieren diese dann als regulären Ausdruck. Eine Eingabeunterstützung für diesen Prozess würde eher in den Bereich „visuelle Programmierung“ fallen, doch machen Menschen Fehler und bei der Fehlersuche bzw. -vermeidung ist Visualisierung wieder von Nutzen, nämlich beim Testen der formulierten Ausdrücke und bei der Ablaufvisualisierung, um ein ev. fehlerhaftes Syntaxverständnis zu korrigieren.

Maintainer sind für uns Personen, die vorhandenen Code wieder verstehen sollen, um ihn zu erweitern oder seine Ausführungsgeschwindigkeit zu erhöhen. Für solche Benutzer liegen die Hauptprobleme nicht in dem Verständnis einzelner kleiner Code-Schnipsel, sondern vielmehr im Verständnis der Zusammenhänge auch größerer Ausdrücke, so dass die Visualisierung entsprechend skalieren muss.

Problemanalyse der bisherigen Darstellung

„Regular expressions tend to be easier to write than they are to read. This is less of a problem if you are the only one who ever needs to maintain the program (or sed routine, or shell script, or what have you), but if several people need to watch over it, the syntax can turn into more of a hindrance than an aid.“ (Stephen Ramsay, University of Virginia, [10])

Wie oben gezeigt, ist die Syntax selbst das Problem beim Verständnis eines regulären Ausdrucks. Aus diesem Grund funktioniert auch ein einfaches Syntax-Highlighting wie oben im Texteditor gesehen nur für sehr einfache Muster. Die hohe Flexibilität regulärer Ausdrücke wird mit einer hoch komplexen Syntax auf kleinstem Raum bezahlt.

Dies ist v.a. darin begründet, dass reguläre Ausdrücke meist in einer reinen Text-Repräsentation geschrieben und analysiert werden, wodurch sämtliche Elemente den gleichen Platz benötigen, obwohl sie für das Erkennen des zu Grunde liegenden Musters nicht alle gleich wichtig sind.

Herkömmliche Ansätze zur unterstützenden Visualisierung versagen darüber hinaus bei regulären Ausdrücken meist, weil der zur Verfügung stehende Platz viel geringer ist als z.B. bei imperativen Programmen wie wir weiter unten zeigen werden.

Eine effiziente Visualisierung muss daher über die üblichen Mittel der Darstellung regulärer Ausdrücke hinausgehen, um Skalierbarkeit und Übersicht zu gewährleisten. Problemlösungen werden sich daran messen lassen müssen, inwieweit es ihnen gelingt, die Elemente mit besonderer Bedeutung geeignet zu darzustellen und das durch den regulären Ausdruck festgelegte Muster verständlich zu machen sowie den Ablauf des regulären Ausdrucks über einem Text zu veranschaulichen.

II. Existierende Werkzeuge

Erstaunlicherweise gab es bis vor wenigen Jahren nicht einmal umfassende Dokumentation zu regulären Ausdrücken (was sich mit dem Erscheinen von Jeffrey Friedls Buch „Mastering Regular Expressions“ [1] glücklicherweise geändert hat)

Während unserer Recherche sind wir daher nur auf eine Handvoll Programme gestoßen, die erste Ansätze zur Visualisierung von regulären Ausdrücken implementieren.

Kate

Abbildung 2

Traditionell werden reguläre Ausdrücke mit einfachen Texteditoren bearbeitet. Der KDE Advanced Text Editor (kurz: Kate) besitzt umfangreiche Funktionen für das Syntax-Highlighting verschiedener Strukturen, darunter auch für reguläre Ausdrücke. Die Visualisierung regulärer Ausdrücke unter Kate beschränkt sich aber auf die ASCII-Darstellung und färbt sämtliche Sonderzeichen lila ein, maskierte Elemente mit spezieller Bedeutung werden türkis dargestellt und alle anderen Zeichen in grün angezeigt.

Komodo

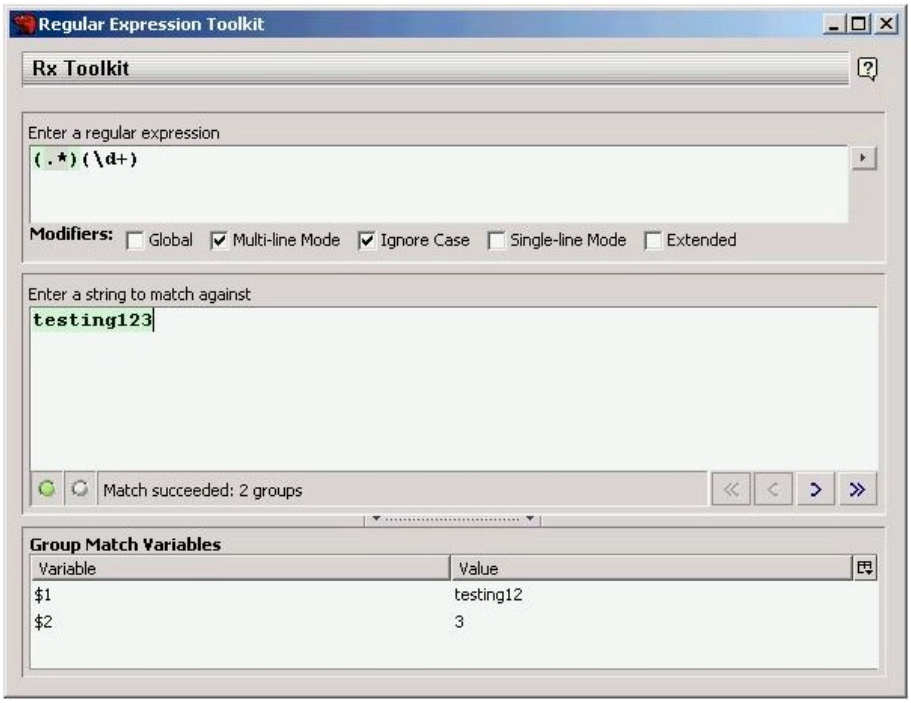

Abbildung 3

Komodo ist eine Entwicklungsumgebung für verschiedene Sprachen wie Perl, PHP und Python. Auf Abbildung 3 sieht man das in Komodo integrierte „Regular Expression Toolkit“, bei dem man oben einen regulären Ausdrücken und in der Mitte die Testdaten eingeben kann. Im unteren Bereich werden die herausgezogenen Variablen angezeigt. Zusätzlich gibt es noch ein paar Optionen, die die Ausführung des regulären Ausdrucks beeinflussen.

VisualRegexp

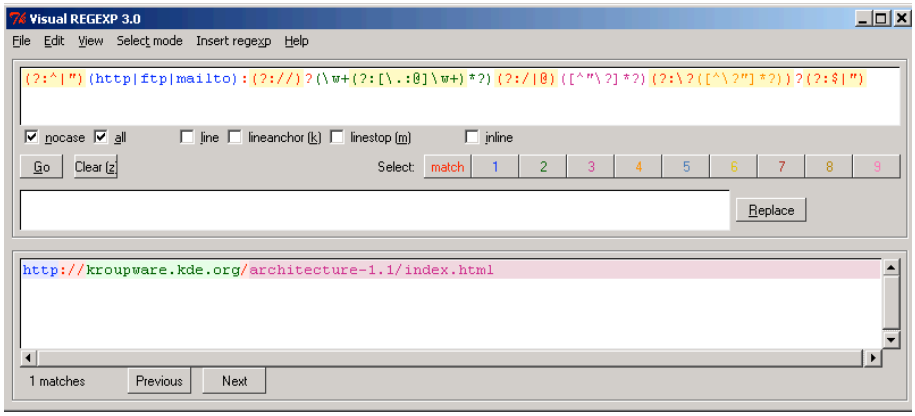

Abbildung 4

Ähnlich wie Komodo dient VisualRegex der Zuordnung von Teilen des regulären Ausdrucks zu den korrespondierenden Teilen des vorgegebenen Textes. Auch hier gibt es wieder einige Optionen, die das Verhalten des Musters beeinflussen.

kregexpeditor

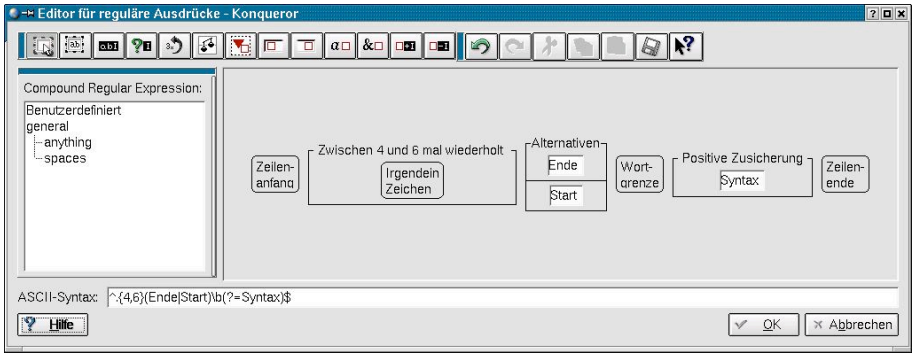

Abbildung 5

In KDE 3 ist der kregexpeditor enthalten, der u.a. über die Suchdialoge des Konqueror aufgerufen werden kann. Ziel des kregexpeditor ist es eigentlich, die Erstellung regulärer Ausdrücke zu vereinfachen. Allerdings kann man ihn auch für die Visualisierung eines regulären Ausdrucks verwenden, da einfache Strukturen grafisch aufbereitet werden.

III. Visualisierungstechniken

Aus der Problemanalyse ergeben sich mehrere sinnvolle Strategien zur Visualisierung regulärer Ausdrücke, die sich in die folgenden Kategorien untergliedern lassen.

Code-Visualisierung

Die Code-Visualisierung bezeichnet die statische grafische Aufbereitung des eigentlichen Codes, also des regulären Ausdrucks. Hierzu wird kein zusätzlicher Text benötigt. Grundgedanke dahinter ist, dass das Verständnis u.a. auch deswegen schwer fällt, weil die Sortierung nach wichtigen und unwichtigeren Elementen erst im Gehirn des Betrachters stattfindet, obwohl hier Unterstützung durch entsprechende Werkzeuge größere Effizienz bringen würde.

Wir untergliedern die verschiedenen Visualisierungsstufen und -funktionen in mehrere Unterpunkte.

Ablauf-Visualisierung

Der Ablauf eines regulären Ausdrucks unterscheidet sich in der Hinsicht von der Ausführung eines kompilierten Programms, dass es unmöglich ist, ihn ohne die zu Grunde liegenden Daten zu visualisieren. Reguläre Ausdrücke sind insgesamt viel stärker an die Daten gebunden als dies bei herkömmlichen Programmen der Fall ist. Daher verstehen wir unter „Ablauf-Visualisierung“ Darstellungen, die neben dem eigentlichen Code (dem regulären Ausdruck) auch noch eine starke zeitliche Komponente aufweisen. Zusätzlich werden immer die zu Grunde liegenden Daten angezeigt.

Ergebnis-Visualisierung

Die Visualisierung des Ergebnisses der „Ausführung“ eines regulären Ausdrucks könnte man als Spezialfall der Ablauf-Visualisierung einordnen, jedoch ist eine Implementierung unabhängig von der zeitlichen Darstellung und auch sonst aus Benutzersicht von der Ablauf-Visualisierung zu unterscheiden, daher führen wir sie als eigene Kategorie auf. Insgesamt werden in dieser Klasse alle Darstellungsformen subsumiert, die sich auf den Zustand der Daten am Ende der Ausführung beziehen.

Profiling

Die gesammelten Informationen über das Laufzeitverhalten können auch grafisch dargestellt werden. Hier geht es v.a. um die Analyse des Ressourcenverbrauchs und besonders interessant ist dabei die Darstellung des Ressourcenverbrauchs aufgeschlüsselt auf die verschiedenen Teile der regulären Ausdrücke, so dass verschiedene Optimierungsmaßnahmen überhaupt nach ihrem Erfolg bewertet werden können.

IV. Code-Visualisierung

Die Code-Visualisierung beschäftigt sich mit der statischen Visualisierung des Codes, also des regulären Ausdrucks. Grundlegendes Ziel ist es, das Muster verständlich zu machen, in dem die Struktur besser kenntlich gemacht wird.

Jeder reguläre Ausdruck ist aus Atomen aufgebaut. Ein Atom ist der kleinste zulässige reguläre Ausdruck, also z.B. `\d` oder `\s`. Jeder Teil eines regulären Ausdrucks, der alleine wieder ein formal korrekter regulärer Ausdruck ist, nennen wir Untermaschine. Eine Untermaschine kann also (muss aber nicht) aus mehreren Atomen bestehen. Häufig hat es sich als sinnvoll erwiesen, Blöcke als Untermaschinen zu definieren. Blöcke sind durch Klammern eingeschlossene Bereiche des regulären Ausdrucks, also z.B. `(\w+)` oder `(?:[abc]*).`

Weitere wichtige Strukturmerkmale eines regulären Ausdrucks sind Quantoren, Mengen von Zeichen, Alternativen, Rückwärtsreferenzen und positive bzw. negative Zusicherungen. Die Darstellung einer speziellen Visualisierung bezeichnen wir als Aspekt.

Wir gliedern das Thema „Code-Visualisierung" in folgenden Unterpunkte; Syntax-Highlighting, Syntax-Styling, Visualisierung über Muster, Alien Highlighting, Übersetzung in natürliche Sprache, visuelle Programmierung, Synergiefunktionen.

Syntax-Highlighting

Unter reinem Syntax-Highlighting verstehen wir die bloße Einfärbung der Strukturen des regulären Ausdrucks, wie von zahlreichen Texteditoren her bekannt. Alle anderen Aspekte des Musters bleiben unverändert.

Diese Ansicht hat insbesondere einige spezielle Vorteile: Zunächst etwas mehr Übersicht und da man immer nah am Code war, kann

auch die gewonnene Übersicht auch dann noch nutzen, falls man einmal mit einem Editor arbeiten, muss der diese Funktion nicht unterstützt (z.B. Notepad©). Da Syntax-Highlighting relativ einfach zu implementieren ist, steht sie auch in vielen verschiedenen Editoren zur Verfügung.

Historisch gesehen wurde das Syntax-Highlighting zunächst bei Editoren konventioneller Programmiersprachen implementiert, um größere Programme übersichtlicher zu machen. Wendet man aber die gleiche Technik auf reguläre Ausdrücke an, muss man feststellen, dass dieses reine Syntax-Highlighting nicht den gewünschten Effekt bringt.

Um das zu verstehen, muss man sich den Unterschied zwischen herkömmlichen Programmen und regulären Ausdrücken verdeutlichen. In Programmiersprachen nimmt der Code einen sehr viel größeren Raum ein, jedoch sind selbst sehr komplexe reguläre Ausdrücke meist Einzeiler. Das hat zur Folge, dass die farbliche Auszeichnung normaler Programme viel spärlicher eingesetzt werden kann, so dass diese Konstellation sinnvoll die menschliche Wahrnehmung unterstützt.

Im Unterschied dazu besitzen reguläre Ausdrücke sehr viele wichtige Strukturinformationen auf kleinstem Raum, so dass der positive Effekt der Farbkodierung praktisch entfällt bzw. bei komplexeren Mustern die Farbvielfalt sogar vom Inhalt ablenkt.

Daraus ergibt sich, dass farbliche Auszeichnungen nur sehr sparsam eingesetzt werden dürfen. Erste Ansätze wären z.B. maskierte Sonderzeichen und andere Elemente wie a, x, 42 usw. ausgegraut darzustellen, um die Aufmerksamkeit von ihnen wegzunehmen. Wichtige Sonderzeichen wie Klammern oder Alternativen können dagegen auffälliger sein. Je seltener ein Operator ist, desto eher sollte er andersfarbig dargestellt werden.

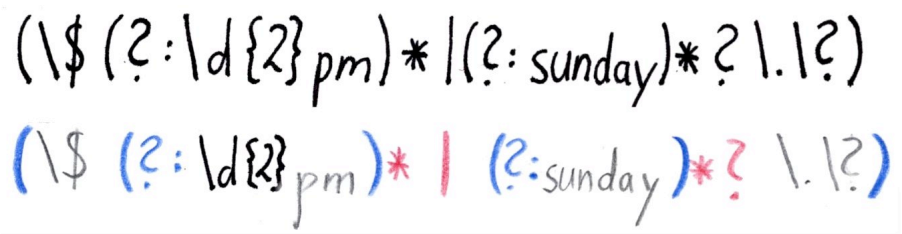

Abbildung 6

Syntax-Styling

Syntax-Styling ist eine Erweiterung des Syntax-Highlighting, in dem alle typographischen Möglichkeiten ausgeschöpft werden

können, ohne allerdings von der ASCII-Darstellung des regulären Ausdrucks abzuweichen. Als Beispiel sei hier die Variation von Farbe, Schriftart, Größe, Ausrichtung etc. pp. genannt. Beim Syntax-Highlighting werden die Zusatzinformationen auf zu engem Raum komprimiert. Syntax-Styling wirkt der Überfrachtung einer einzelnen Ebene wie z.B. Farbe entgegen, indem es die Informationen durch die Verteilung auf mehrere Ebenen „auseinander zieht“ Insbesondere können einzelne Elemente auch unauffälliger gemacht werden.

Folgende konkrete Vorschläge für Auszeichnungen wollen wir im Folgenden machen:

Quantoren wie z.B. `*, +, ?, {4}, {,3}` sind eigentlich nur Attribute des vorangehenden Blocks und haben keine eigenständige Bedeutung, sind für das Verstehen des Ausdrucks u.U. aber sogar im Weg. Durch hochgestellte Quantoren verringert man die Aufmerksamkeit, die ihnen zukommt, ohne diese durchaus wichtigen Zusatzinformationen gänzlich zu entfernen. Ein weiterer wichtiger Aspekt bei Quantoren ist der Scope, auf dem sie gelten. Eine farbige Unterstreichung des Scopes erleichtert das Verständnis. Quantoren, die maximale Zeichenketten finden („greedy matching“) werden zusätzlich farblich markiert.

Alternativen innerhalb eines regulären Ausdrucks (repräsentiert durch ein Pipe-Symbol), Nicht-Vorkommen der folgenden Musters sowie Zeilenanfang und -ende werden fett hervorgehoben, um diese Strukturmerkmale zu kennzeichnen.

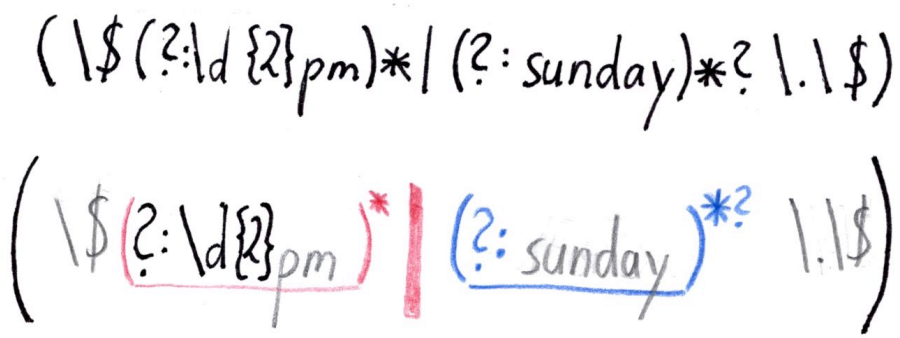

Abbildung 7

Visualisierung über Muster

Der nächste Schritt nach dem einfachen Syntax-Styling wie oben gezeigt besteht in einer weitergehenden Umformatierung des Codes. Sie ist immer noch statisch, jedoch stehen jetzt alle grafischen

Elemente zur Verfügung. Diese weitergehende Visualisierung löst sich von der zeilenbasierten Darstellung.

Ein Ansatzpunkt für eine weitergehende Darstellung ist die Zerlegung des regulären Ausdrucks in separate Untermaschinen. Ähnlich wie beim menschlichen Verstehen werden reguläre Ausdrücke auch im Computer in einzelne Untermaschinen zerlegt und in der Auswertung verknüpft. Dies macht die technische Realisierung der Visualisierung einfacher.

Insgesamt wird die Ansicht dadurch übersichtlicher, weil es eine gezielte Verschiebung der Informationen auf andere Ebenen gibt. Daher werden zum ersten Mal auch ASCII-Kodierungen weggelassen und z.B. durch Farbe oder Schriftart ersetzt.

In manchen regulären Ausdrücken werden z.B. bestimmte Zeichen hexadezimal kodiert `(\x092)`, eine Umwandlung in Unicode-Zeichen macht dieses Muster für Menschen lesbarer `(\x061-\x07a` wird zu `a-z`).

In Blöcken wie `(?: ...)` kann man `?:` weglassen, wenn man sie leicht grau hinterlegt. Auch die Kennzeichnung für nicht-gierige oder gierige Operatoren wie `.*` bzw. `.*?` können durch die gleichen Zeichen dargestellt werden, wenn die gierigen Elemente farblich hervorgehoben werden. Ansonsten schöpfen wir beim Visualisieren über Muster natürlich auch alle Zeichen des Syntax-Stylings aus, falls dies sinnvoll erscheint.

Weitergehende Visualisierungen bestehen darin, ein Muster auf mehrere Zeilen aufzuteilen, um die Informationsdichte etwas zu verringern. Eine sinnvolle Umbruchstrategie sollte sich natürlich an den Untermaschinen orientieren.

Besonderes Augenmerk wollen wir aber auf die Strukturvisualisierung legen, da jetzt weitere grafische Elemente verfügbar sind. Besondere Strukturelemente sind:

- Alternativen, dargestellt durch ein einfach zu übersehendes Pipe-Symbol, werden untereinander angeordnet werden.
- Blöcke können auch vertikal eingerückt werden, so dass die verschiedenen Ebenen besser zu erkennen sind. Durch unterschiedlich große Klammern (wie z.B. von LaTeX her bekannt) wird ebenfalls die Übersicht erhöht.
- Sonderzeichen werden durch einen zusätzlichen Backslash maskiert, so dass sie zu normalen Zeichen werden. Diese Maskierungssymbole blähen den ganzen Ausdruck stark auf. Wenn die entspr. Sonderzeichen ausgegraut dargestellt werden, können auch die maskierten Zeichen weggelassen werden.

- Rückwärtsreferenzen: Anstatt mühsam den regulären Ausdruck durchzusehen, um den referenzierten Teil zu finden, kann auch ein Pfeil einfach auf die entsprechende Stelle verweisen. Sollte der referenzierte Teil gerade ausgeblendet sein (s. dazu auch den nächsten Punkt), verweist der Pfeil auf das Symbol, mit dem die Untermaschinen wieder eingeblendet werden können. Zusätzlich findet der Beobachter an jedem Pfeil ein Symbol, mit dem das interessante Muster zeitweilig eingeblendet werden kann.

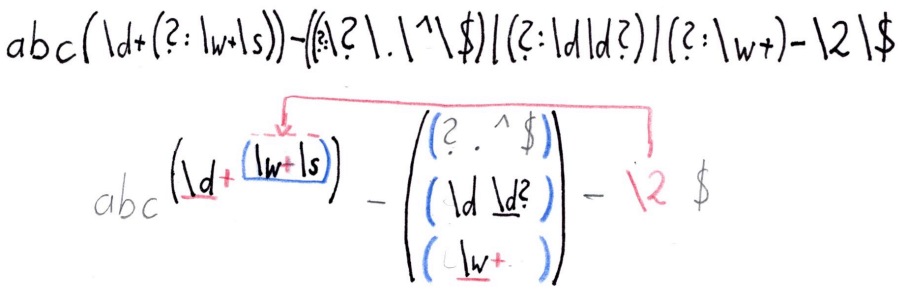

Abbildung 8

Gerade bei längeren regulären Ausdrücken interessiert sich der Beobachter häufig nur für einen Teil des regulären Ausdrucks. Zeitweise uninteressante Teile können daher ausgeblendet werden. Diese Technik ist auch von einigen Entwicklungsumgebungen her bekannt, die es erlauben, Funktionsrümpfe und andere Blöcke zusammenzuklappen und so die Übersichtlichkeit zu erhöhen. Naheliegenderweise geschieht dieses Einfalten bei regulären Ausdrücken an Hand der Untermaschinen.

Alien Highlighting

Hier standen reguläre Automaten Pate für die Idee, den regulären Ausdruck in einen Graph umzuformen, in dem man besonders einfach seine Struktur und die möglichen Datenpfade durch ihn sehen kann. Ein Knoten stellt dabei ein oder mehrere Elemente des regulären Ausdrucks dar, während eine Kante einen möglichen Ausführungspfad angibt.

Wichtig ist dabei, dass man explizit nicht alle Details erkennen können soll, sondern die Übersichtlichkeit als wichtigstes Ziel im Vordergrund steht. Den Aufbau dieses Graphen will ich im Folgenden rekursiv definieren:

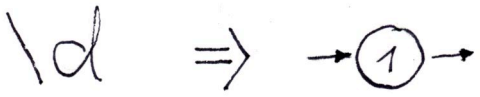

Abbildung 9

Als kleinste Einheit wählen wir die Atome eines regulären Ausdrucks. Für die Darstellung dafür verwenden wir einen Kreis, allerdings voraussichtlich ohne eine Angabe, was er genau darstellt (genau dieses soll aber in den anderen Aspekten hervorgehoben werden, sobald man ihn auswählt).

Abbildung 10

In diesem Kreis wird angegeben, wie oft sich dieses Muster wiederholen muss. Wichtig dabei: Soll sich ein Muster auch Null mal wiederholen können, wird soll entweder ein zusätzlicher Pfeil an dem „Token“ vorbeigeführt werden oder eine Null innerhalb des Kreises erscheinen.

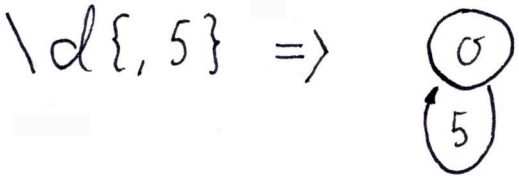

Abbildung 11

Soll ein Element maximal n-Mal auftreten, erscheint diese Zahl innerhalb der Schleife unterhalb des Elements.

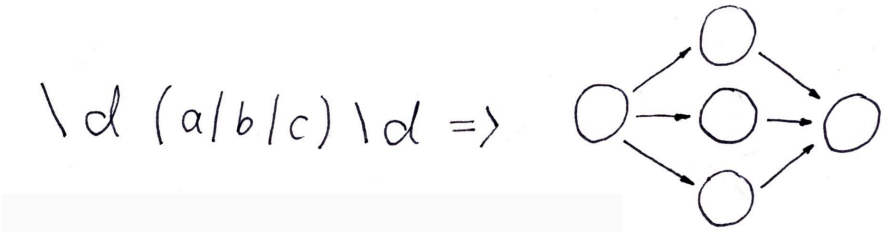

Abbildung 12

Alternativen werden schlicht als zwei (bis n) mögliche Ausgänge aus einem Muster dargestellt. Beispielsweise:„`(a|b|c)`“.

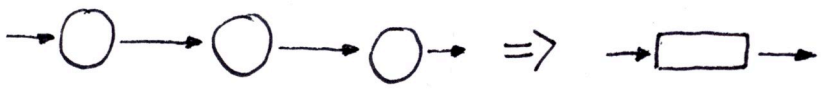

Abbildung 13

Eine Sequenz von Elementen kann als Rechteck zusammengefasst werden.

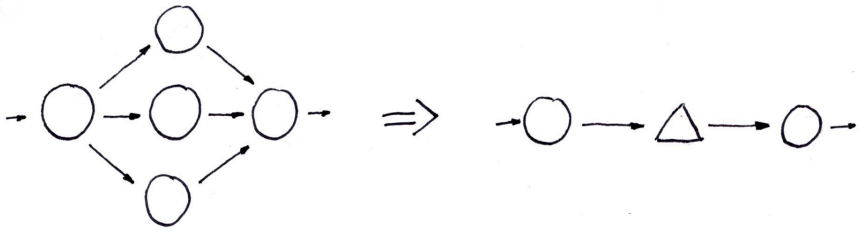

Abbildung 14

Verzweigungen können von dem Programm als Dreieck dargestellt werden.

In dieser Darstellung soll man sich frei bewegen können, um sie als eine Art Übersicht zu nutzen und um möglichst schnell in regulären Ausdrücken an die Stelle zu kommen, an denen man in anderen Aspekten arbeiten möchte.

Die regulären Ausdrücke werden in der Darstellung nach bestimmten Regeln zusammengefasst. Dafür reichen im Prinzip zwei Muster aus, das oben beschriebene Rechteck für eine Sequenz und das Dreieck für Alternativen (siehe Regeln 6. und 7.). Wie Elemente zusammengefasst werden, kann der Benutzer entweder selbst entscheiden oder dem System überlassen.

Weitere Maßnahmen, um die Menge der Information, die dieses Diagramm transportiert zu erhöhen, kann und sollte man diskutieren.

Denkbar ist beispielsweise eine farbliche Kodierung der Symbole, so dass man über die Farbe oder deren Helligkeit die Anzahl der darin zusammengefassten Atome erkennen kann, oder ein Eintrag in die Zusammenfassungen wie viele Elemente sie genau zusammenfassen.

Übersetzung in natürliche Sprache

Die einfachste Art für einen Einsteiger, einen regulären Ausdruck zu verstehen, ist die Verknüpfung mit bereits bekannten Darstellungsformen und hier eignet sich unserer Ansicht nach die natürliche Sprache am besten, da der Benutzer diese auf jeden Fall schon kennt.

Diese Visualisierung soll also in der Lage sein, jedem Atom eines regulären Ausdrucks die korrekte deutsche (englische, spanische, ...) Bedeutung zuzuordnen. Als logische Erweiterung kann diese Funktion einen (zumindest beinahe korrekten) deutschen Satz aus einem nicht zu großen regulären Ausdruck erzeugen.

Mehrere Möglichkeiten sind dafür vorstellbar: Man könnte entweder versuchen, tatsächlich Fließtext zu erzeugen oder man kann eine Erweiterung des Alien-Highlightings verwenden, in dem tatsächlich innerhalb der Elemente eine Erklärung der Elemente enthalten ist. Für eine experimentelle Implementierung dieser Variante sei hier auf den „kregexpeditor" [5] verwiesen

Diese „Übersetzung in eine natürliche Sprache" kann natürlich an vielen Stellen genutzt werden, um dort zusätzliche Hilfe anzubieten. Praktisch überall kann man sie integrieren, um per „Tooltip" einen Glossar anzubieten. Beispielsweise, um bei einer "Menge" tatsächlich sehen zu können, welche Zeichen sie enthält.

Beispiel: `\d` => "`0 1 2 3 4 5 6 7 8 9`"

Beispiel: `\x21-\x25` => "`! " # $ %`"

Diese Funktion ist hauptsächlich an den Einsteiger gerichtet, lässt sich aber auch für fortgeschrittene Benutzer (Programmierer und Maintainer) sehr gut als schnelle Referenz verwenden, da man sehr schnell nachprüfen kann, ob das geistige Bild einer Sequenz von Anweisungen tatsächlich der Realität entspricht.

Visuelle Programmierung

Unter visueller Programmierung fassen wir Funktionen zusammen, die eine iterative Erstellung von regulären Ausdrücken ermöglichen. Dabei soll diese Funktion vor allem auf die Bedürfnisse eines Programmierers zugeschnitten sein, der Muster, die er in einem Text gefunden hat, schnell in einen regulären Ausdruck zu überführen

möchte. Sie kann aber durchaus auch für den Einsteiger interessant sein, da diese Methode sicherstellt, dass das System viel mehr Fehler automatisch erkennen kann und an vielen Stellen auch bessere Vorschläge für die weitere Arbeit machen kann.

Konkret soll das so funktionieren, dass der Benutzer ein Beispieltext vorgibt, aus dem er bestimmte Merkmale extrahieren oder einfach erkennen möchte. In einem nächsten Schritt spezifiziert der Benutzer dann genau die Teile des Textes, die er daraus extrahieren möchte

Nun ist das System in der Lage, einen regulären Ausdruck vorzuschlagen, der genau diesen Zweck erfüllt und dann von dem Benutzer Schritt für Schritt so weit verfeinert werden kann, bis er damit zufrieden ist.

Besonders elegant an diesem Ansatz ist, dass er einerseits iterativ und zweitens „test-driven“ ist. Das heißt, dass man sich durch die Vorgabe der Beispiele „Proben“ definiert, die von dem regulären Ausdruck erkannt werden müssen, was dem System zusätzliche Möglichkeiten gibt, die Korrektheit der Ausdrücke zu überprüfen und den Programmierer so oft mit der Nase auf den Grund eines Problems stoßen kann.

Diese Funktion kann man aber natürlich auch umgekehrt nutzen, das heißt, man definiert einen regulären Ausdruck und lässt das System potenzielle Kandidaten generieren, die dieser Ausdruck erkennen würde. Natürlich können diese Beispiele nie besonders gut sein, aber sie können doch auch helfen, die erkennbaren Muster und speziell die Auswirkungen zu verdeutlichen, die eine Änderung an dem Ausdruck hat.

Synergiefunktionen

Bisher haben wir verschiedene Möglichkeiten aufgezeigt, wie man den Code eines regulären Ausdrucks visualisieren kann. Allerdings sind all diese Tools für einen erfahrenen Anwender von regulären Ausdrücken nur sehr begrenzt nutzbar, da jeder Aspekt für sich genommen ab einer bestimmten Komplexität der Ausdrücke immer weniger nützlich ist.

Genau diesem Problem aber ist lösbar, in dem man die Möglichkeiten der verschiedenen Visualisierungen auf raffinierte Weise kombiniert und verknüpft (oder zumindest lässt sich die Größe der

regulären Ausdrücke, die mit Hilfe dieses Systems noch sinnvoll bearbeitet werden können, damit wesentlich nach oben schieben).[1]

Essenziell ist hierbei, dass die Anzahl der gleichzeitig angebotenen Informationen nicht die Aufnahmefähigkeit des vor dem System sitzenden Benutzers überschreiten darf. Daher ist eine der wichtigsten Aufgaben des Editors, dem Benutzer zu erlauben, dass er jederzeit alle von ihm nicht benötigten Funktionen ausblenden kann. Sinnvoll erscheint uns hier eine drei- bis vierstufige Konzentrationsmöglichkeit durch die man entweder alles einblenden, Teile verdecken und nur den gerade notwendigen Kontext zeigen kann.

Die erste Stufe ist dabei trivial. Man bekommt als Benutzer alle Aspekte angeboten, die das System anzubieten hat. Beispielsweise im oberen Bereich des Fensters die verschiedenen Aspekte, während im unteren die Anzeige der Tests und der Laufzeit zu sehen ist.

Die zweite Stufe soll dazu dienen, einen Überblick erhalten zu können, hier werden viele Zusatzinformationen ausgeblendet, so dass der Benutzer sich besser auf die restlichen Aspekte konzentrieren kann,

Auf der dritten Stufe soll sich die Ansicht auf einen Aspekt beschränken, damit man diesen möglichst effizient bearbeiten kann. Mögliche Zusatzfunktionen könnten eine Bibliothek von Vorlagen und dort abgelegten „Code-Schnipseln“ sein.

Die vierte und letzte Stufe soll es dem erfahrenen Benutzer ermöglichen, dass er völlig ungestört an genau einer Ansicht arbeiten kann. Alle Zusatzfunktionen des Systems werden abgeschaltet.

Unsere Vorstellung ist, dass der erfahrene Benutzer die Stärken der verschiedenen Darstellungen optimal ausnutzt, um damit Schwächen der anderen auszugleichen, also beispielsweise zuerst die Laufzeitvisualisierung verwendet, um einen groben Überblick zu erhalten was überhaupt in dem regulären Ausdruck passiert. Danach das „Alien

[1] Zwar gehört der Editor nicht direkt zur Visualisierungen, dennoch legt er einen wichtigen Grundstock auf dem viele andere Visualisierungen aufbauen und erst richtig zum Tragen kommen. Wünschenswert wären unterstützende Funktionen wie unbeschränktes Undo/Redo, ein Verweis auf korrespondierende Klammern, automatische Maskierung von Sonderzeichen, eine Bibliothek von Vorlagen (Erkennen einer URL, eines Datums etc) und eine ständige Eingabeüberprüfung auf mögliche Fehler (ähnlich einer Rechtschreibkorrektur in modernen Textverarbeitungen).

Highlighting" für einen schnellen Überblick und Navigation innerhalb des Ausdrucks und auch um die anderen Aspekte gut koordinieren zu können. In den „Syntax Highlighting“ und „Visualisierung über Muster“ Aspekten kann der Anwender am Schnellsten etwas ändern oder einen „no nonsense“-Blick auf den Code werfen und sich mit der Übersetzung in eine natürliche Sprache schnell vergewissern, dass er keine Details übersieht.

Damit das alles funktioniert, müssen sich die verschiedenen Aspekte selbstverständlich ständig abgleichen, in dem sie zum Beispiel die gleichen Selektionen zeigen und auf die gleichen Bereiche „zoomen“.

Zwar wirken die hier beschriebenen Funktionen zuerst eher unscheinbar, sind aber ein sehr wichtiger Teil des ganzen Pakets, weil durch die verschiedenen gleichzeitigen Ansichten auf den regulären Ausdruck die Schwächen einer Ansicht durch die Stärken einer anderen ausgeglichen werden können. Dadurch wird es möglich, komplexe Ausdrücke zu bearbeiten, die in nur einem Aspekt betrachtet nicht mehr beherrschbar wären. Als Beispiel sei hier auf die „Speedbar“[2] von Emacs verwiesen, die für sich alleine zwar nicht viel Nutzen hat, es aber ermöglicht, viel größere Quelltexte komfortabel zu bearbeiten.

V. Ablaufvisualisierung / Debugging

Der Grundgedanke dieses Abschnittes ist, dass ein Benutzer, der die Funktionsweise eines regulären Ausdrucks verstehen möchte, ein mentales Modell erschaffen muss, wie dieser Abläuft. Genau diesen Vorgang des „Durchspielens“ eines regulären Ausdrucks wollen wir visualisieren, um das erstellen eines solchen Modells im Kopf des Benutzers zu unterstützen, Fehler darin aufzudecken und einfach den Benutzer zu entlasten, damit er den Kopf frei hat, um über andere schwierige Details nachzudenken.

Wichtig ist dabei, dass der Benutzer für diese Visualisierungen zu dem regulären Ausdruck noch einen oder mehrere Texte angeben muss, auf denen das Muster ausgeführt werden soll.

---

[2] Die Speedbar in Emacs stellt eine zusätzliche Ansicht auf den Quelltext zur Verfügung, indem sie es erlaubt, Code-Dateien als Unterverzeichnisse darzustellen. Dabei werden die darin deklarierten Methoden und Klassen ihrerseits wieder als Verzeichnisse und Dateien aufgefasst und im Editor nach einem Klick darauf angezeigt.

Für einen Einsteiger ist dies besonders praktisch, da er sich das mentale Modell, wie ein regulärer Ausdruck abläuft, sonst erst erarbeiten müsste. Ein Entwickler kann damit Tests definieren, die erfüllt werden müssen und sich knifflige Details auch im Detail noch einmal ansehen, hat aber gleichzeitig den Kopf für andere Dinge frei. Ein Maintainer kann diese Funktion gut nutzen, um schnell einen Einblick zu bekommen, was ein regulärer Ausdruck tut und wieso er so aufgebaut ist wie er es ist oder um Fehler darin zu finden.

Die tatsächliche Visualisierung des Ablaufs kann an mehreren Stellen und auf verschiedene Art und Weise erfolgen.

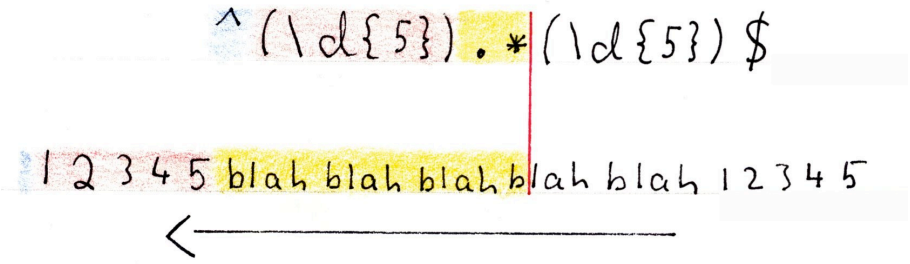

Abbildung 15

In Abbildung 15 ist im oberen Bereich der reguläre Ausdruck gezeigt, während im unteren der zu treffende Text an dem regulären Ausdruck vorbeigeführt wird. Der rote Strich bezeichnet dabei den „Carret“, also die Stelle im regulären Ausdruck und im Text der gerade evaluiert wird. Wie durch den Pfeil angedeutet, soll sich der Text nach links bewegen und dabei so eingefärbt werden, wie es dem Teil des regulären Ausdrucks entspricht, der diese Textpassage erkannt hat.

Relativ egal ist dabei, welche Visualisierung für den regulären Ausdruck oben gewählt wird, „Alien Highlighting“ kann in bestimmten Situationen durchaus Vorteile haben. Eine andere Variation dieser Visualisierung ist, die Rollen auszutauschen. D.h., dass nicht mehr der Text, sondern stattdessen der reguläre Ausdruck bewegt wird.

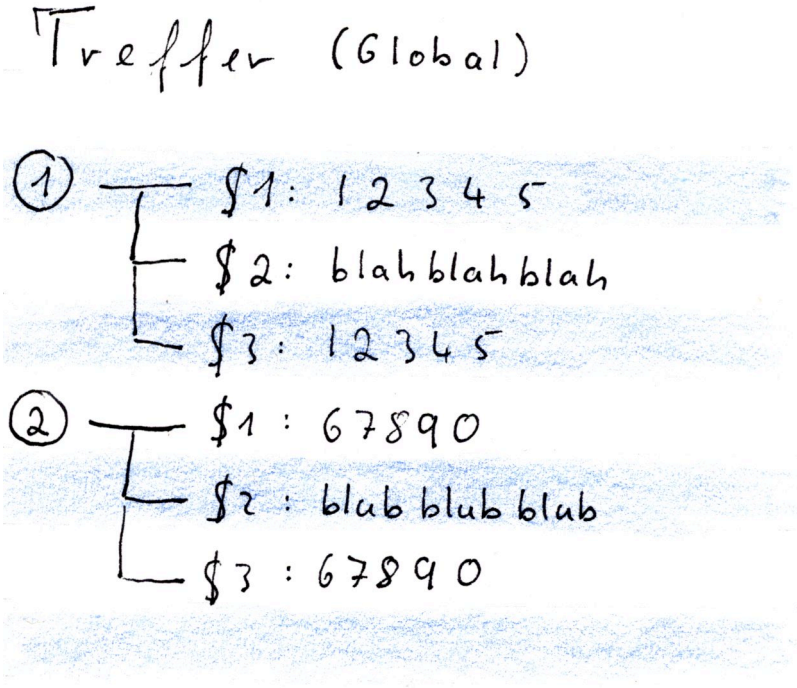

Abbildung 16

Abbildung 16 zeigt eine Möglichkeit den Inhalt der Variablen separiert vom regulären Ausdruck und Text anzuzeigen. Der Vorteil dieser Darstellung liegt in der Übersichtlichkeit, da sehr viele, möglicherweise irrelevante Informationen weggelassen werden.

Ablaufvisualisierung umfasst auch die Fehlersuche bzw. das Debugging von regulären Ausdrücken. Kurz zur Definition, Debuggen heißt Fehler suchen, d.h. etwas funktioniert nicht so wie es soll, obwohl man selbst den Fehler nicht im Quelltext sieht. Ein Fehler in diesem Kontext könnte sein, dass ein bestimmtes Muster im Text gar nicht oder fälschlicherweise doch erkannt wird. Also setzt man „Breakpoints“, entweder im Text oder im regulären Ausdruck, um mit ihnen die Quelle des Fehlers einzugrenzen. Trifft das Programm auf einen „Breakpoint“ kann es verschiedene Aktionen auslösen, z.B. den Eintritt in den Einzelschrittmodus oder einfach eine Veränderung der Ablaufgeschwindigkeit.

Grundsätzlich wird der reguläre Ausdruck nach jeder Modifikation im Hintergrund automatisch auf den Text angewendet und die Ausführung aufgezeichnet. Sollten dadurch Geschwindigkeitsprobleme

auftreten, kann das System die regulären Ausdrücke aber auch erst auf explizite Anforderung des Benutzers anwenden.

All das soll den Benutzer in die Lage versetzen, die Ablaufgeschwindigkeit der regulären Ausdrücke frei anzugeben, an beliebiger Stelle eine Pause einzulegen, die Ausführung vorwärts und rückwärts zu betrachteten und sich auch schrittweise zu genau der Stelle zu bewegen, die ihn interessiert.

VI. Ergebnisvisualisierung

Unter diesem Punkt verstehen wir statische Visualisierungen, die keine Informationen über das Ablaufverhalten der regulären Ausdrücke nutzen. In der Regel ist dies relativ leicht, da die meisten Implementierungen regulärer Ausdrücke dies direkt unterstützen. Im Prinzip ist die Ergebnisvisualisierung ein Sonderfall der Ablaufvisualisierung, wobei nur deren Endergebnis zu sehen ist.

Wesentlich ist die Anzeige, welche Untermaschinen zu welchen Teilen des Textes gepasst haben. Auch die Darstellung der herausgezogenen Variablen ist sinnvoll, als Beispiel verweisen wir hier auf Komodo [1].

Da man verschiedene Implementierungen regulärer Ausdrücke in das Framework einbinden kann, ist es eventuell nicht möglich, die Ablaufvisualisierung durchzuführen. In diesem Fall steht nur diese Ansicht zur Verfügung. Ein weiterer Vorteil ist, dass man sich mit diesem Mechanismus relativ leicht vor Fehlern in der eigenen Implementierung schützen oder diese testen kann. Zusätzlich ist es damit relativ leicht, Geschwindigkeitsvergleiche zwischen verschiedenen Implementierungen durchzuführen.

VII. Profiling: Zeitverbrauch

In herkömmlichen Programmen ist das Profiling zur Messung des Ressourcenverbrauchs (Rechenzeit, Speicherverbrauch, I/O-Wartezeit, ...) sehr verbreitet, insbesondere, um lokale Engpässe zu erkennen und zu beseitigen. Relevant ist bei regulären Ausdrücken v.a. die benötigte Rechenzeit, um ein Muster auf einen Text anzulegen.

Zeitkritisch sind insbesondere Muster, die zu exponentiellem Aufwand führen. Dies mag in Verbindung mit „regulären Ausdrücken“ zunächst widersprüchlich erscheinen und in der Tat sind „reguläre“ Ausdrücke mit Rückwärtsreferenzen oder bestimmten Operatoren nicht mehr im formalen Sinne „regulär“.

In seinem Buch hat Jeffrey Friedl[3] auch verschiedene Optimierungstechniken für reguläre Ausdrücke vorgestellt, insbesondere das Kapitel „Unrolling the Loop“ und das entsprechende Pattern ist hier interessant.

Die Messung des Zeitverbrauchs kann helfen, ein tiefer gehendes Verständnis für reguläre Ausdrücke zu erhalten und die einzelnen Muster bewusster einzusetzen. Insbesondere könnten Profilingergebnisse auch dazu führen, sich für eine anderweitige Realisierung zu entscheiden oder den Erfolg einer Optimierungsmaßnahme abzuschätzen.

Wie schon weiter oben gezeigt, ist der Begriff der Ausführungsgeschwindigkeit eines regulären Ausdrucks sinnlos, wenn die Daten außer Acht gelassen werden, auf denen der Ausdruck ausgeführt wird. Daher gibt es auch keinen definitiven Zeitverbrauch, sondern nur einen in Verbindung mit ausgewählten Datensätzen. Hier bieten sich vergleichende Diagramme an, die den Zeitverbrauch eines Durchlaufs über den verschiedenen Texten auftragen und so das Verhalten eines regulären Ausdrucks über unterschiedlichen Eingabedaten visualisieren.

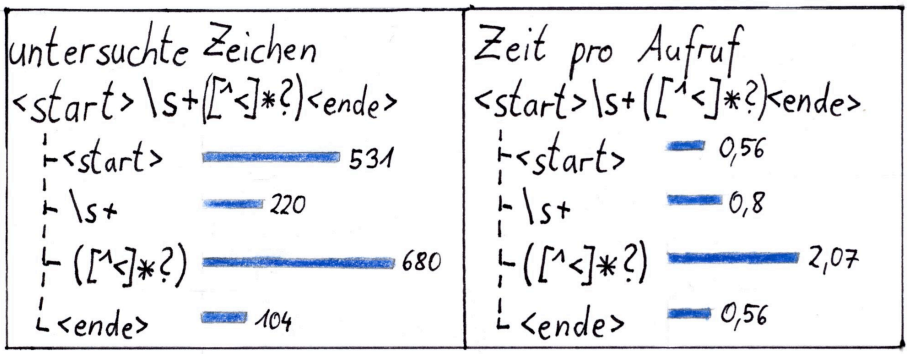

Abbildung 17

[3] Man kann sogar zeigen, dass das Matching von regulären Ausdrücken mit Rückreferenzen ein NP-vollständiges Problem ist, s. [4]. Für die Implementation einer Rückreferenz benötigt man daher auch eine Turingmaschine, während normale reguläre Ausdrücke selbstverständlich mit regulären, endlichen Automaten möglich sind. Damit ist natürlich klar, dass es einige Kombinationen von regulären Ausdrücken gibt, bei denen sich eine ausführlichere Laufzeitanalyse lohnt. Weiterführende Informationen über die verschiedenen Implementationen von regulären Ausdruecken ist insbesondere in [1] zu finden.

VIII. Bewertung der existierenden Werkzeuge

Nachdem wir die verschiedenen Kategorien zur Visualisierung regulärer Ausdrücke vorgestellt und beschrieben haben, können wir jetzt die vorhin erwähnten, bereits existierenden Werkzeuge in Bezug auf ihre Nützlichkeit für die drei verschiedenen Benutzergruppen bewerten.

Kate

Kate beschränkt sich strikt auf die reine ASCII-Syntax und bietet daher nur einfaches Syntax-Highlighting, wodurch er auch allen Schwächen unterliegt, die bei der Beschreibung des Syntax-Highlighting genannt wurden. Insbesondere mangelt es an der strukturierten Aufbereitung regulärer Ausdrücke und der Skalierbarkeit bei komplexeren Mustern. Letztlich sind die farblichen Hervorhebungen von Kate nur für sehr kleine Ausdrücke zu gebrauchen, ansonsten ist der Zusatznutzen sehr gering. Keine der Benutzergruppen profitiert in größerem Maße.

Komodo

Komodo beschränkt sich im Wesentlichen auf die Ergebnis-Visualisierung. Entsprechend sind auch die mehrfachen Treffer innerhalb eines Textes, die Anzeige, welche Untermaschine welchen Block im Text gefunden hat und die Anzeige der herausgezogenen Variablen sehr von Nutzen. Hinzu kommt die Erklärung der einzelnen Atome, die ein erster Schritt zur natürlichsprachlichen Darstellung ist. Leider fällt bei Komodo der gesamte Bereich der Visualisierung über Muster weg und auch die Strukturansicht kommt zu kurz, so dass viele der hier vorgestellten Ideen damit nicht zu verwirklichen sind.

kregexpeditor

Die Visualisierung regulärer Ausdrücke ist hierbei unter den vorgestellten Programmen die am weitesten gehende, gerade die Verbindung von natürlichsprachlicher Erklärung und dem eigentlichen Code hilft v.a. dem Einsteiger und zum Teil auch dem Maintainer. Insgesamt gibt es aber Probleme bei der Skalierung der regulären Ausdrücke, wie man auf Abbildung 18 sieht. Grund dafür ist, dass immer alle Details zur Verfügung stehen und nicht ausgeblendet werden können. Weiterhin sind die Erklärungen in Worten manchmal

schlicht falsch, z.B. wird auch „?“ schlicht als „mindestens ein Mal“ anstatt von „ein Mal oder kein Mal“ beschrieben.

Abgesehen davon ist die eigentliche Aufgabe des kregexpeditors die visuelle Programmierung eines regulären Ausdrucks, wo er den Programmierer unterstützt. Aber auch hier ist der Nutzen bei komplexeren Mustern auf Grund der fehlenden Testmöglichkeiten etwas eingeschränkt.

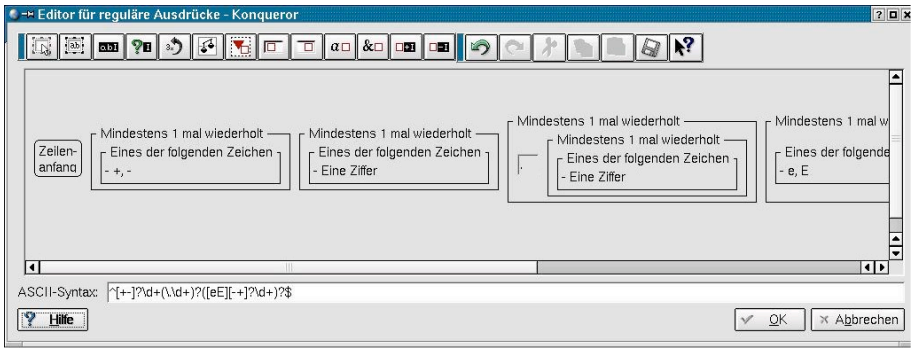

Abbildung 18

VisualRegexp

Auch VisualRegexp beschränkt sich wie Komodo auf die Ergebnis-Visualisierung, bei einem Treffer wird auch über die farblichen Zuordnungen sehr schön deutlich, welche Untermaschine welchen Teil des Textes erkannt hat. Leider werden die herausgefilterten Variablen nicht dargestellt und es werden auch nicht mehrere Treffer in einem Fließtext dargestellt. Dadurch ist dieses Programm vornehmlich für einfache Debugging-Aufgaben geeignet, aber ansonsten für keine der drei Benutzergruppen wirklich brauchbar.

Zusammenfassend kann man also sagen, dass die vorgestellten Werkzeuge bisher reguläre Ausdrücke nur mangelhaft visualisieren und echte Synergie-Effekte nicht zu stande kommen. Allerdings sind viele der Programme auch recht neu – die Funktionen für reguläre Ausdrücke wurden erst innerhalb der letzten zwei Jahre erstellt, so dass hier noch Hoffnung auf weitere Verbesserungen besteht. Bis dahin sind sie aber noch weit von einer optimalen Unterstützung entfernt.

IX. Fazit

Wir sind der Meinung, dass die Implementierung unserer Vorschlaege zusammengefasst die Bearbeitung und das Verständnis regulärer Ausdrücke enorm beschleunigen könnten. Reguläre Ausdrücke bieten gerade bei der Mustererkennung in Texten viele Vorteile, die aber bisher nur schwer zugänglich waren. Dies kann sich mit den von uns erarbeiteten Visualisierungstechniken aber ändern.

Ansätze einer Entwicklung in diesem Gebiet gibt es bereits (siehe Abschnitt „Existierende Werkzeuge“). Wir denken, dass sich dieser Bereich in Zukunft noch stark entwickeln wird und heutige Programme noch längst nicht alle Möglichkeiten ausschöpfen.

X. Future Work

Selbstverständlich gibt es weitere Themen die zum verbesserten Verständnis von regulären Ausdrücken beitragen könnten:

- Vertonung (Aurelization) als weiteren weiteren Feedback-Mechanismus für den Benutzer
- Interaktive Generierung von regulären Ausdrücken anhand von Textbeispielen (visuelle Programmierung).
- Erzeugen von Beispielen aus regulären Ausdrücken
- Implementierung einer entsprechenden Umgebung :-)

XI. Quellen

1) Mastering Regular Expressions, Jeffrey E.F. Friedl, O'Reilly, 6. Nachdruck 1998
2) Visuelle Programmierung – Grundlagen und Einsatzmöglichkeiten, Steffan Schiffer, Addison-Wesley, 1998
3) How Regexes Work, Mark-Jason Dominus, http://perl.plover.com/Regex/article.html
4) Perl Regular Expression Matching is NP-Complete http://perl.plover.com/NPC/index.html
5) kregexpeditor, KDE 3.1, http://www.kde.org/und http://www.blackie.dk/KDE/KRegExpEditor/
6) Komodo, http://www.activestate.com/komodo
7) Kate, http://www.kde.org und http//kate.kde.org
8) VisualRegexp, http://laurent.riesterer.free.fr/regexp/
9) SelfHTML, zu regulären Ausdrücken http://selfhtml.teamone.de/cgiperl/sprache/regexpr.htm
10) Using Regular Expressions, Stephen Ramsay http://etext.lib.virginia.edu/helpsheets/regex.html
11) Erkennung gültiger E-Mail-Adressen, Tom Christiansen http://www.perl.com/CPAN/authors/Tom_Christiansen/scripts/ckaddr.gz