Reguläre Ausdrücke und Textmanipulationstools

[Ein kurzer Überblick]

Thomas Leichtle
Universität Ulm
Ulm, Deutschland
Thomas.Leichtle@uni-ulm.de

1. EINLEITUNG

Die meisten Leute verwenden sie - bewusst oder unbewusst - und ebenso gibt es in den gängigsten Programmier- und Skriptsprachen eine entsprechende Implementierung dafür: zum einfachen Suchen von Zeichenketten eines bestimmten Musters, als Hilfsmittel zur schnellen Formatierung (vor allem für große Datenmengen) oder sogar als Teil von Compilern und Parsern. Reguläre Ausdrücke machen in vielen Fällen das Leben leichter!

In dieser kurzen Einführung in Reguläre Ausdrücke und Textmanipulationstools werden sowohl einige theoretische Grundlagen erläutert, als auch Beispiele aufgeführt und Implementierungsmöglichkeiten angedeutet und verglichen.

2. THEORETISCHE GRUNDLAGEN

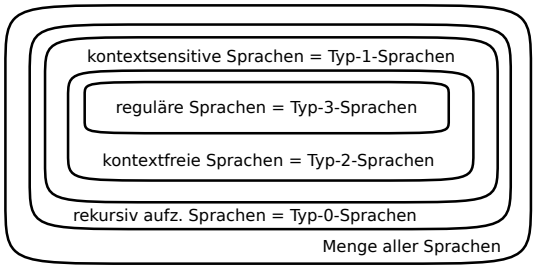
2.1 Definition

Was sind Reguläre Ausdrücke? Man kann sie z.B. wie folgt definieren:

- $\emptyset$ (die leere Menge) ist ein regulärer Ausdruck.
- ϵ (das leere Wort) ist ein regulärer Ausdruck.
- Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck, also jeder einzelne Buchstabe des Alphabets.
- Wenn α und β reguläre Ausdrücke sind, dann auch $\alpha\beta$, $(\alpha \mid \beta)$, sowie $(\alpha)^*$. Wobei $\alpha\beta$ der Konkatenation, $(\alpha \mid \beta)$ α oder β und $(\alpha)^*$ der beliebig häufigen Konkatenation von α mit α entspricht.

2.2 Einordnung / Klassifizierung

Wie in vielen Grundlagenvorlesungen in der Informatik behandelt, kann man Sprachen klassifizieren. Meist wird hierfür die Chomsky-Hierarchie [Sch04] verwendet, welche zwischen rekursiv aufzählbaren Sprachen, kontextsensitiven Sprachen, kontextfreien Sprachen und regulären Sprachen unterscheidet. Hierbei sind nach dem Satz von Kleene [1] genau die von den regulären Ausdrücken erzeugten Sprachen äquivalent zu den regulären Sprachen, den Typ-3-Sprachen [Sch04]. Aus der Definition der regulären Ausdrücke folgt auch die Abgeschlossenheit dieser Sprachen unter Vereinigung, Schnitt, Komplement, Produkt, Stern.



2.3 Weitere Äquivalenzen

Aus der Äquivalenz mit den regulären Sprachen folgen noch weitere Äquivalenzen, u.a. mit folgendem Satz:
Eine Sprache ist genau dann regulär, wenn es einen endlichen Automaten (Deterministic Finite Automaton, kurz: DFA) M gibt, der sie erkennt.
$M = (Z, \Sigma, \delta, z_0, E)$ mit endlicher Zustandsmenge Z, endlichem Eingabealphabet Σ, dem Startzustand $z_0 \in Z$, Endzustandsmenge E und $\delta : Z \times \Sigma \rightarrow Z$ als Überführungsfunktion.

Weiterhin gilt nach dem Satz von Rabin & Scott:
Jede von einem Nichtdeterministischen Automaten (Nondeterministic Finite Automaton, kurz: NFA [2]) akzeptierbare Sprache ist auch durch einen DFA akzeptierbar [Sch04].

Wir haben also identische Darstellungsmöglichkeiten:
Reguläre Spachen $\Leftrightarrow$ DFA $\Leftrightarrow$ NFA $\Leftrightarrow$ Reguläre Ausdrücke
Es würde noch weitere geben, jedoch sollen uns diese hierfür genügen.

[1] Stephen Cole Kleene: US-amerikanischer Mathematiker und Logiker (1909-1994). Gilt auch als Erfinder regulärer Ausdrücke.
[2] Ein NFA enthält in der Definition im Gegensatz zum DFA eine Startzustandmenge und die Zustandsübergänge müssen sich nicht gegenseitig ausschließen.

2.4 Beispiel

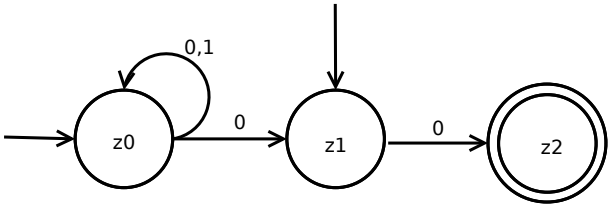
Zur Veranschaulichung ein kleines Beispiel:

2.4.1 Regulärer Ausdruck

Die Sprache der Worte über dem Alphabet $\{0,1\}$ die auf "00" enden oder $x = 0$ (das Wort, das genau aus einer "0" besteht), wäre als regulärer Ausdruck z.B.: $(0|(0|1)^*00)$.

2.4.2 Nichtdeterministischer endlicher Automat (NFA)

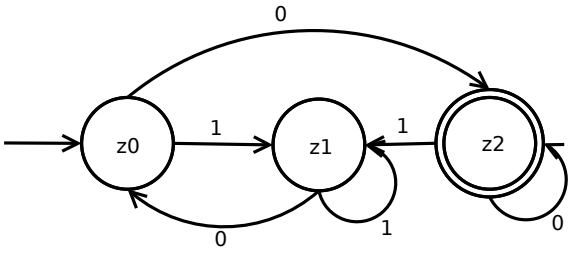
Ein Zustandsgraph eines passenden NFA ist dann ...



oder als Formel: $M = (Z, \Sigma, \delta, S, E)$ mit:
Zuständen $Z = \{z0, z1, z2\}$, Alphabet $\Sigma = \{0, 1\}$
Startzustände $S = \{z0, z1\}$, Endzustände $E = \{z2\}$
Überführungsfunktion $\delta : \delta(z0, 0) = z0$, $\delta(z0, 1) = z0$, $\delta(z0, 0) = z1$, $\delta(z1, 0) = z2$.

2.4.3 Deterministischer endlicher Automat (DFA)

Und wie bereits erwähnt, kann jeder NFA in einen DFA umgeformt werden. Eine Möglichkeit wäre folgende ...



Wobei für gewöhnlich dann (deutlich) mehr Zustände und Zustandsübergänge benötigt werden. Auch hier einmal noch die mathematische Beschreibung: $M = (Z, \Sigma, \delta, z0, E)$ mit: $Z = \{z0, z1, z2\}$, $\Sigma = \{0, 1\}$, $E = \{z2\}$, $\delta : \delta(z0, 0) = z2$, $\delta(z0, 1) = z1$, $\delta(z1, 0) = z0$, $\delta(z1, 1) = z1$, $\delta(z2, 0) = z2$, $\delta(z2, 1) = z1$.

2.5 Notation

Um auf die Notation einzugehen, welche bisher nicht sehr formal gehalten war und auch später nicht allzu genau eingehalten werden wird, trotzdem ein kurzer Einschub.
$\mathcal{L}(regex)$ bezeichnet die formale Sprache, die durch den regulären Ausdruck *regex* spezifiziert wird. Es gilt dann:

- $\mathcal{L}(\emptyset) = \emptyset$
- $\forall a_i \in \Sigma : \mathcal{L}(a_i) = \{a_i\}$
- $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$
- $\mathcal{L}(\alpha|\beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$

2.6 Das Pumping Lemma

Ebenfalls ergänzend wird nun noch das Pumping Lemma ≡ Schleifenlemma ≡ Iterationslemma ≡ uvw-Theorem aufgeführt. Es ist eines der wichtigsten Hilfsmittel, wenn man zeigen will, dass eine Sprache *nicht* regulär ist.

2.6.1 Definition

Sei L eine reguläre Sprache. Dann $\exists n_0$, so dass sich alle Wörter x ∈ L mit $|x| \geq n_0$ zerlegen lassen in $x = uvw$, so dass gilt:

1. $|v| \geq 1$,
2. $|uv| \leq n_0$,
3. $\forall i = 0, 1, 2 \ldots\ uv^i w \in L$.

2.6.2 Beispiel

Um das Prinzip davon zu erläutern, überprüft man, ob $L = \{a^n b^n | n \geq 1\}$ regulär ist. Unter der Annahme L sei regulär, existiert ein n_0, so dass es für alle Wörter x ∈ L, deren Länge $\geq n_0$, eine Zerlegung in uvw gemäß dem Pumping Lemma gibt:
Betrachtet man das Wort $a^{n_0}b^{n_0}$ der Länge $2n_0 \geq n_0$. Es folgt aus den Bedingungen 2 & 1, dass v aus a's bestehen muss und eben mindestens aus einem. Aus Bedingung 3 würde dann folgen, dass $\forall i \in \mathbb{N}_0$ auch $a^{|u|+|v|*i}b^{n_0}$, also z.B. auch $a^{n_0-|v|}b^{n_0}$ in L liegen müsste, was jedoch nicht zutrifft.
⇒ nicht regulär!

2.7 Folgerungen

Durch die bereits genannten Äquivalenzen, lässt sich nun leicht erkennen, dass man für ein gegebenes Wort x entscheiden kann, ob es einer bestimmten regulären Sprache angehört - oder in unserem Fall - auf einen regulären Ausdruck matcht. Dies ist sogar mit Hilfe eines DFAs in linearer Zeit möglich, da man nur den Zustandsübergängen folgen muss.

3. PRAKTISCHE ANWENDUNG

Verwendung von regulären Ausdrücken findet man vor allem im Bereich der Textanalyse und -manipulation. Sei dies nun zur Erkennung syntaktischer Merkmale oder eben zum Suchen (& Ersetzen) bestimmter Ausdrücke. Ken Thompson erstellte dafür bereits in den 1960ern eine Implementierung für den Editor "qed" [Rit] und inzwischen können bei sehr vielen Programmen reguläre Ausdrücke verwendet werden. *Anmerkung: Da in der Praxis auch Rückwärtsreferenzen unterstützt werden, stimmen die Ausdrücke jedoch nicht mehr ganz mit den vorher erwähnten Typ-3-Sprachen überein.* [3]

3.1 Das Tool "grep"

„Global search for a regular expression and print out matched lines - kurz g/re/p" [Wik]. Es versucht seinem Namen entsprechend reguläre Ausdrücke auf eine Texteingabe anzuwenden und die betreffenden Zeilen auszugeben.

[3] Dies kann unter anderem mit Hilfe des Pumping Lemmas gezeigt werden.

3.1.1 Beispiel

Wie oft passiert es einem, dass man in seiner ständig wachsenden Ordnerhierarchie irgendwelche Dateien „verliert“? Abhilfe kann geschaffen werden, indem man sich einfach alle Dateien eines Pfades anzeigen lässt und die erwünschten herausfiltert, z.B. so: *find . | grep “Tschebyschew“*. Aber wie war der Name nochmal genau? Tschebyschew, Tschebyscheff, Tschebyschow, Tschebyschev, Chebyshev? Also erstellt man sich den Ausdruck:
→ find . | grep -E “(Tsc|C)hebysc*h[eo]([vw]|ff)“, welcher auf alle diese Zeichenketten matcht. Natürlich kann dies auf jede beliebige Texteingabe angewendet werden.

3.1.2 Erläuterung: grep

Es gibt kleine Unterschiede zwischen den verschiedenen Implementierungen für GNU/plan9/BSD/usw, welche hier aber ausgelassen werden und bei Bedarf in der entsprechenden Literatur nachgelesen werden können.

Eine kurze Erläuterung zur Syntax...

- anhand des Beispiels:
  - vorne steht entweder Tsc oder C
  - der * in der Mitte bezieht sich auf das c (für mehrere Zeichen muss man runde Klammern setzen).
  - nach dem h steht entweder ein e oder ein o, d.h. [eo] ≡ (e|o).
  - am Ende steht entweder v oder w oder ff ≡ (v|w|ff).
- ... und etwas mehr:
  - der Punkt . steht für ein beliebiges Zeichen.
  - [^a-z]≡ keines der Zeichen der Menge von a-z darf vorkommen.
  - bis auf ^ und] und * werden alle Zeichen in der Zeichenmenge in den eckigen Klammern auch als Zeichen interpretiert und haben keine Sonderbedeutung.
  - Quantifikatoren (1):
    ? ≡ 0-1 Mal, * ≡ ≥ 0 Mal, + ≡ > 0 Mal.
  - Quantifikatoren (2):
    {n} ≡ genau n Mal, {n,} ≡ min n Mal,
    {n,m} ≡ min n & max m Mal.
  - Positionierung:
    ^ matcht auf Zeilenanfang und $ auf Zeilenende.
  - Achtung: grep “α“ nimmt vor und nach α beliebig viele beliebige Zeichen an! Also grep “[0-9]“ matcht auch auf etwas wie *lisois193ajk21ksdf*.

Es würde noch viele weitere Möglichkeiten geben Einschränkungen vorzunehmen. Wer Interesse daran hat, sollte einfach eine Suchmaschine seiner Wahl um Rat fragen oder die Man-Pages dazu lesen.

3.2 Das Tool “sed“

Der StreamEDitor ist dazu gedacht Texteingaben in der Komandozeile zu bearbeiten und dahingehend auch vielfach in Skripten verwendet zu werden. Wer schonmal in mehreren 100 Zeilen Text Wörter suchen und ersetzen musste oder auch einfach nur alle Links in einer html-Datei gerne schön angezeigt gehabt hätte, hat hiermit einen sehr guten Freund gefunden!
Jedoch ist sed mit all seinen Features auch turing-äquivalent, d.h. man sollte auf keinen Fall das Potential unterschätzen! Es wurden bereits komplexere Programme wie Spiele oder Debugger damit geschrieben, wobei dies wohl eher die Ausnahmefälle sind.

3.2.1 Beispiel 1

Zurück zum Einfachen: Im Zeitalter der Emanzipation möchte man alle Superhelden eines Buches zu Heldinnen machen. Anstatt dies von Hand zu erledigen, kann man natürlich auf reguläre Ausdrücke (mit Backreferences) zurückgreifen. Mit sed sieht das dann z.B. so aus:
→ sed -r “s/(Super|Bat|Spider)man/\1woman/g“ < text

3.2.2 Beispiel 2

Oder man will alle Links der Startseite der Universität Ulm ausgegeben ...
→ sed -rn 's/.*href=“(.*)“.*/\1/p' < Home.html
→ sed -rn 's/.*href=“([^“]*)“.*/\1/p' < Home.html
Was stimmt nun? Um genau zu sein keines von beiden. Es wird beides mal nur ein Vorkommen *pro Zeile* gefunden - was uns hierfür allerdings genügt, um es nicht unnötig kompliziert zu machen. Ein weiterer Unterschied ist jedoch, dass bei der 1. Variante noch mehrere Attribute des html-Tags ausgegeben werden, wenn vorhanden. Dies ist auf die *gierige Implementierung* zurückzuführen, welche später noch erläutert wird.

3.2.3 Erläuterung: sed

Sed hat eine Vielzahl von Parametern, Optionen und Anwendungsmöglichkeiten. Eine kleine Auswahl wird jetzt vorgestellt:

- Textersetzung (= **S**ubstitution):
  - Notation: sed 's/Exp1/Text1/g' *vgl. Beispiele*
  - Ersetzt den Ausdruck Exp1 durch Text1. In unserem 1. Beispiel eben (..)man durch (..)woman.
  - Mit \1,\2,...,\n kann auf den Inhalt der runden Klammern im Ausdruck zugegriffen werden, wobei bei (a(b)) \1 “ab“ liefert und \2 “b“.
- Zeilen löschen (= **D**elete):
  - Notation: sed '/Exp2/**d**'
  - Löscht alle Zeilen auf die Exp2 zutrifft.
- Allgemeines:
  - Die Notation der regulären Ausdrücke ähnelt sehr der von grep. Jedoch werden standardmäßig alle Zeilen (editiert oder unverändert) ausgegeben.
  - Mit dem Parameter -n wird die Standardausgabe unterdrückt. Dies ist sinnvoll in gemeinsamer Nutzung mit der Option /p, welche alle Zeilen ausgibt, auf die der Ausdruck matcht. *vgl. Beispiel 2*

- Mit dem Parameter -r können Erweiterte Reguläre Ausdrücke verwendet werden.
- Viele Optionen nehmen auch eine Adresse oder einen Adressbereich als zusätzliche Einschränkung: sed '/start/,#stop/s/a/b/' mit...
  - start: Regulärer Ausdruck, welcher die Startmarkierung matcht.
  - #stop: Zeilennummer der letzten Zeile des gewünschten "Adressbereichs".

  Adressierung über Zeilennummern & Ausdrücke beliebig vertauschbar

3.3 Erweiterte RegEx vs. Nicht-Erweiterte

Der Unterschied ist die Notation. In allen Beispielen bisher wurden *erweiterte Reguläre Ausdrücke (= ERE)* verwendet. Dies wurde bei grep durch Angaben des Parameters -E und bei sed mit -r festgelegt.
Der Grund warum es überhaupt verschiedene Notationen gibt, ist wie so oft die historische Entwicklung. Früher gab es noch weniger Metazeichen (z.B. Quantifikatoren wie ?,+,*). Wenn man nun neue Metazeichen einführt, so werden die ursprünglich als Zeichen interpretierten Symbole nicht mehr als solche erkannt, sondern übernehmen eben eine Sonderfunktion. Aus Kompatibilitätsgründen wurden diese neuen Metazeichen in den Nicht-Erweiterten oder eben *Basic Regular Expressions (= BRE)* escaped - ihnen wurde also ein "\" vorangestellt.
() werden also als Terminalsymbole betrachtet, wohingegen \(\) als Metazeichen interpretiert werden. Das Problem war nun allerdings folgendes: Die "alten" Metazeichen mussten escaped werden um als Terminale erkannt zu werden und die neuen um als Metazeichen gedeutet zu werden, was zur Verwirrung führt. Um dem entgegen zu wirken definierte man EREs, die mehr Metazeichen kennen, wie z.B. + oder ?. Unterschiede [LR]:

- ERE: (...) $\Leftrightarrow$ BRE: \(...\)
- ERE: x? $\Rightarrow$ BRE: gibt es nicht
- ERE: x+ $\Leftrightarrow$ BRE: xx* (+ wird als Terminal erkannt)
- ERE: x{m} $\Leftrightarrow$ BRE: x\{m\}
- ERE: x{m,n} $\Leftrightarrow$ BRE: x\{m\,n}
- ERE: x{m,} $\Leftrightarrow$ BRE: x\{m\,}

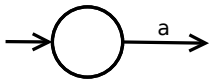
4. IMPLEMENTIERUNG

Es gibt verschiedene Ansätze Reguläre Ausdrücke zu implementieren. Alle haben Vor- und Nachteile gegenüber den anderen, aber darauf wird später noch etwas genauer eingegangen.

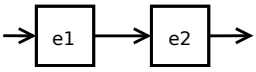
4.1 Thompson Automat

Durch die bereits beschriebenen Äquivalenzen wissen wir, dass wenn wir es schaffen einen passenden Automaten zu implementieren, damit auch Reguläre Ausdrücke matchen können. Dieser Ansatz wurde auch von Ken Thompson gewählt, welchen er 1968 in seinem CACM-paper beschrieb [Tho68].
Eine Konvertierung eines Ausdrucks in einen Automaten kann demnach wie folgt aussehen:

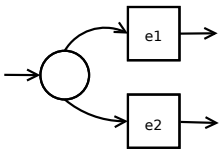
- $a \in \Sigma$:



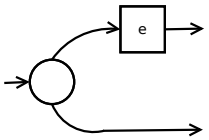
- e_1e_2:



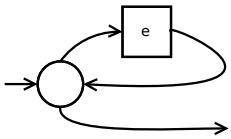
- $e_1|e_2$:



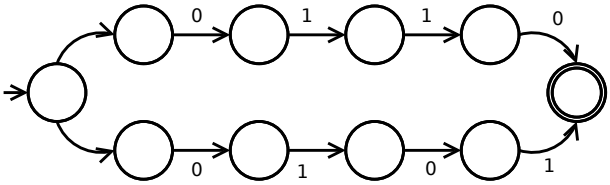
- $e?$:



- $e*$:



Ein so aufgebauter Automat (Thompson NFA) für den Ausdruck 0110|0101 wäre:



Da es sich um einen NFA handelt, muss man sich immer wieder für einen Pfad entscheiden. Allerdings kann sich ein NFA in maximal so viel Zuständen befinden, wie er besitzt. D.h. es ist möglich alle Pfade parallel abzuarbeiten und sollte es in einem Pfad keinen gültigen Übergang mehr geben, so wird dieser gestrichen. Erreicht man mit einem Pfad den Endzustand, so akzeptiert man.[4]

Die Realisierung eines Thompson Automaten in Quellcode ist für etwas geübte Programmierer nicht sehr schwer und kann ausführlicher nachgelesen werden [Cox].[5]

[4] Anmerkung: Dies ist sicher auch interessant im Angesicht der immer mehr gewünschten Parallelisierung
[5] Russ Cox macht dies in ungefähr 400 Zeilen C-Code.

Ein anderer Ansatz ist Backtracking:
Man arbeitet den Automaten soweit ab, bis eine Entscheidung getroffen werden muss. Diese Stelle merkt man sich um bei einem späteren fehlgeschlagenen Schritt wieder zurückspringen zu können. Der Nachteil dieser Implementierung ist, dass im Falle eines Nicht-Zutreffens jeder Pfad abgelaufen werden muss, was das ganze sehr langsam machen kann (Exponentielle Laufzeit). Ein weiteres Manko ist die Speicherung der jeweiligen Rücksprungmarken, das vorallem bei längeren Ausdrücken Probleme bereiten kann. Trotzdem ist dies die Implementierung, welche in Perl, PCRE, Python, Ruby und anderen Sprachen verwendet wird [Cox].

4.2 Backreferences

Der Grund für diese langsame Implementierung sind Backreferences. Wie bereits angedeutet sind Reguläre Ausdrücke mit Backreferences mächtiger als NFAs. D.h. wenn man Backreferences verwenden möchten, muss man ineffizientere Algorithmen verwenden. Um genau zu sein, ist dieses Problem der Mustererkennung mit Rückverweisen sogar *NP-Vollständig*. Es existieren keine bekannten, effizienten Algorithmen um solche Probleme zu lösen!
Fazit: Backreferences nur verwenden, wenn sie auch wirklich gebraucht werden!

4.3 Greedy vs. Reluctant

In den meisten Implementierungen sind die Quantifikatoren ?, *, + so definiert, dass sie versuchen die *maximale* Anzahl zu matchen - sie sind also gierig (= greedy). Beispiel: Der Ausdruck (.+)(.+) auf den String 01234 angewendet würde (0123) und (4) ergeben.
Perl hat zusätzlich zurückhaltende (= reluctant) Operatoren eingeführt, die durch ein nachgestelltes ? kenntlich gemacht werden, also: ??, *?, +?. Diese versuchen die *minimale* Anzahl von Zeichen zu matchen. Beispiel: (.+?)(.+?) auf 01234 angewendet ist im Gegensatz zu vorher (0) und (1234).

Da in Perl sowieso Backtracking verwendet wird, benötigt dies keine zusätzliche Laufzeit, da einfach gesagt werden kann, dass mit dem Versuch des kürzesten Teilstrings begonnen werden soll. Jedoch ist es ebenfalls möglich eine Variante des Thompson's Algorithmus entsprechend anzupassen.

Es gibt auch Möglichkeiten/Tricks normale (greedy) Ausdrücke so zu formulieren, dass sie ein identisches Ergebnis zu den zurückhaltenden liefern. In Beispiel 2 zu sed wurde dies bereits gemacht. Man schließt einfach das Zeichen (in unserem Beispiel das Anführungszeichen ") aus: "([^"]*)"

4.4 Look-around Assertions

Look-around Assertions wurden in Perl mit Version 5 eingeführt und werden inzwischen auch von PCRE, Python und dem .NET-Framework unterstützt [Goy]. Sie ermöglichen es kontextsensitive Bedingungen zu formulieren - sowohl durch Beschreibung des vorausgehenden Ausdrucks, als auch durch einen folgenden.

4.4.1 Notation

- Look-ahead Assertions:
  - positive Formulierung: Ausdruck1(?=Ausdruck2)
  - negative Formulierung: Ausdruck1(?!Ausdruck2)
- Look-behind Assertions:
  - positive Formulierung: (?<=Ausdruck2)Ausdruck1
  - negative Formulierung: (?<!Ausdruck2)Ausdruck1

4.4.2 Beispiel

Der Ausdruck Sport(?=verein) trifft beim Matchen mit "Ein Sportler betreibt Sport im Sportverein" nur auf das letzte Vorkommen von "Sport" zu, da nur hier "verein" folgt - *nicht* jedoch auf den Gesamtstring "Sportverein".

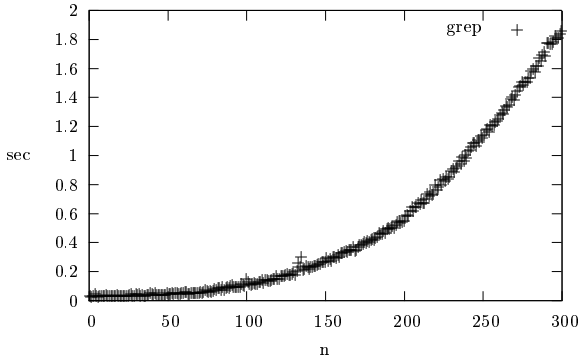
5. PERFORMANCE - VERGLEICH

In diesem Abschnitt wird noch ein wenig genauer auf die Unterschiede der Implementierungen eingegangen.

5.1 Der Test

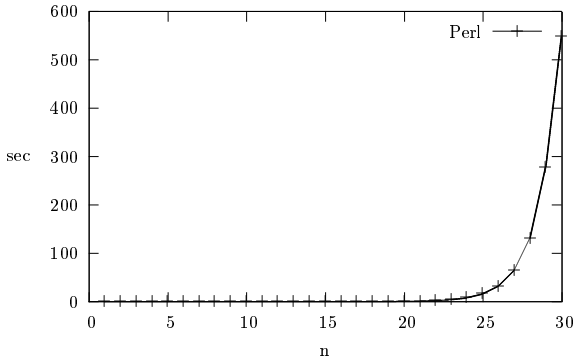
Als kleine Benchmark versucht man hier mit verschiedenen Tools den Ausdruck $(a?)^n a^n$ auf den Text a^n anzuwenden. Dazu kann man sich ein paar kleine Bash-Skripte erstellen, welche die Eingabe und den Ausdruck generieren und dann die Ausführungszeit mit 'time' messen. Eine Möglichkeit zur Visualisierung der Ergebnisse bietet gnuplot.

5.2 Grep



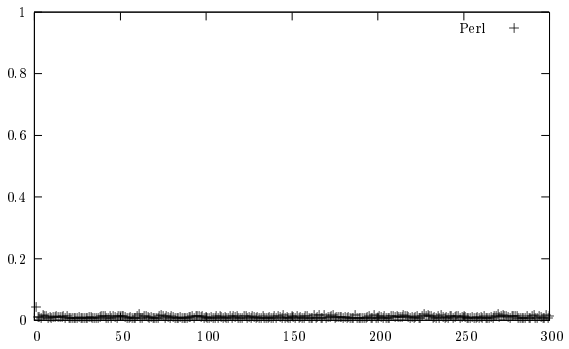
Wie man in dem Schaubild erkennen kann verarbeitet grep den Ausdruck für ein n $\leq$ 300 in unter 2 Sekunden.
Anmerkung: grep verwendet kein Backtracking, sondern eine Art Thompson NFA.

5.3 Perl (greedy)



Im Vergleich hierzu der gleiche Ausdruck angewendet auf die gleiche Eingabe in Perl. Da hier eine Greedy-Implementierung über Backtracking benutzt wird, ist die der worst-case. Es werden bereits über 500 Sekunden für n = 30 benötigt! *Anmerkung: Man achte auf die Skalierung!*

5.4 Perl (reluctant)



Dass Perl nicht immer so langsam ist und es durchaus auf die Formulierung der Ausdrücke ankommt, kann hier erkannt werden. Es handelt sich um die gleiche Perl-Implementierung, wobei nun die zurückhaltende Variante des ? Operators, also in Perl eben ?? verwendet wurde. Man hat nun den Best-Case erreicht und es sind auch bei zunehmendem n keine Performance-Einbusen zu erkennen, da pro zusätzlichem Operator nur ein zusätzlicher Schritt ausgeführt werden muss.

6. SCHLUSSFOLGERUNG

Reguläre Ausdrücke sind unumgänglich und auch sehr nützlich. Jedoch sollte man bei ihrer Formulierung mitdenken und auf einige Sachen achten, wie z.B. Backreferences nur verwenden, wenn es unbedingt sein muss und auch auf die Notation achten. Auch ist es durchaus sinnvoll sich zu informieren welche Implementierung die benutzte Programmiersprache oder das entsprechende Programm verwendet um vorhersagen zu können was man machen kann und was nicht - sowohl von der syntaktischen, als auch der effizienztechnischen Seite betrachtet.
Aufbauend auf den in dieser Ausarbeitung dargestellten Mitteln, können nun - bei Bedarf - weiterführende Implementierungen entwickelt werden.

7. REFERENCES

[Cox] Cox, Russ: *Regular Expression Matching Can Be Simple And Fast.* http://swtch.com/~rsc/regexp/regexp1.html.

[Goy] Goyvaerts, Jan: *Regular Expressions - Lookaround.* http://www.regular-expressions.info/lookaround.html.

[LR] Leibniz-Rechenzentrum: *Reguläre Sprachen, reguläre Ausdrücke.* http://www.lrz.de/services/schulung/unterlagen/regul/.

[Rit] Ritchie, Dennis: *An incomplete history of the QED text editor.* http://plan9.bell-labs.com/~dmr/qed.html.

[Sch04] Schoening, Prof. Dr. Uwe: *Boolesche Funktionen, Logik, Grammatiken und Automaten.* Theo.Inf.II - Skript, 2004.

[Tho68] Thompson, Ken: *Regular expression search algorithm.* Communications of the ACM 11(6), 1968.

[Wik] Wiki.Unixboard: *Linuxfible - Unix-Werkzeuge.* http://wiki.unixboard.de/index.php/Linuxfibel_-_Unix-Werkzeuge_-_Grep.

8. WEITERE LITERATUR

1. Folien zur Vorlesung Theoerische Informatik (6b) *http://theo.cs.uni-magdeburg.de/lehre05s/ti_iif/folien/* von B. Reichel, R. Stiebe
2. Reguläre Ausdrücke - SelfLinux.org *http://www.selflinux.org/selflinux/html/regex.html* von Dennis Roch
3. Reguläre Ausdrücke und Konstruktionen auf Endlichen Automaten *http://tfs.cs.tu-berlin.de/lehre/SS05/UML-Java/regexp-4.pdf* von Benjamin Braatz
4. Manpages grep, sed, ...