Reguläre Ausdrücke

Ferdinand Beyer

Inhaltsverzeichnis

1 Allgemeines **2**
- 1.1 Beschreibung . . . 2
- 1.2 Bezeichnungen . . . 2
- 1.3 Einsatzgebiete in UNIX-Tools . . . 2
- 1.4 Notationsarten . . . 2

2 Notation **2**
- 2.1 Zeichen und Metazeichen . . . 2
- 2.2 Beliebige Zeichen . . . 3
- 2.3 Zeichenauswahlen . . . 3
- 2.4 Quantoren . . . 4
- 2.5 Unterausdrücke . . . 4
- 2.6 Alternativen . . . 5
- 2.7 Zeilenanfang und -ende . . . 5

3 Unterschiede in den Notationsarten **5**
- 3.1 Implementierungsspezifische Erweiterungen . . . 5
- 3.2 Unterschiede zwischen „Basic“ und “Extended“ Regular Expressions . . . 5

4 Anwendungsbeispiele **6**
- 4.1 grep . . . 6
- 4.2 sed . . . 7

1 Allgemeines

1.1 Beschreibung

Reguläre Ausdrücke dienen der Beschreibung von formalen Sprachen, das heißt sie beschreiben Mengen von Zeichenketten. Sie sind ein wichtiges Themengebiet der theoretischen Informatik. Sie werden üblicherweise als speziell kodierte Stringmuster notiert und als endliche Automaten implementiert.

1.2 Bezeichnungen

Meistens werden reguläre Ausdrücke mit ihrer englischen Entsprechung *regular expressions* bezeichnet. In Programmcode und Man-Pages finden sich oft die Abkürzungen `RegExp`, `Regex`, `RE` oder Umschreibungen wie *pattern*.

1.3 Einsatzgebiete in UNIX-Tools

In (UNIX-)Anwendungen werden sie vor allen zur Klassifizierung und Validierung von Zeichenketten (z.B. `lex`), zum Filtern von Daten z.B. `grep`), sowie zum Suchen und Ersetzen in Texten (z.B. `vi`, `sed`) eingesetzt.

1.4 Notationsarten

Für UNIX-Tools unterscheidet man grundsätzlich zwei Notationsarten für die Kodierung von regulären Ausdrücken: *Basic Regular Expressions* (BRE) und *Extended Regular Expressions* (ERE). *Basic regular expressions* werden oft als überholt angesehen und werden meist nur aus Gründen der Abwärtskompatibilität unterstützt. Dennoch (bzw. aus dem selben Grund) ist dies in vielen Tools wie `grep` oder *sed* die Standard-Notationsart.

Extended regular expressions haben eine konsistentere Syntax. Sie sind der de-facto Standard und werden auch von vielen Nicht-UNIX-Tools und Programmiersprachen (z.B. *Perl*) unterstützt.

2 Notation

2.1 Zeichen und Metazeichen

Reguläre Ausdrücke werden als eine Kette von Zeichenliteralen und Metazeichen notiert. Literale repräsentieren sich selbst. So steht beispielsweise der Ausdruck `wort` aus den vier Zeichenliteralen `w`, `o`, `r` und `t` für alle Zeichenketten, die „wort“ enthalten.

Metazeichen haben innerhalb des Ausdrucks eine besondere Kodierungsbedeutung. Diese sind: [ ] ( ) { } ? + ^ $ \.

Um Metazeichen als Zeichenliterale zu behandeln, müssen diese maskiert werden. Hierzu dient der Backslash „\“, der die Sonderbedeutung des unmittelbar folgenden Metazeichens aufhebt.

2.2 Beliebige Zeichen

Das Punkt-Metazeichen „.“ steht für ein beliebiges Zeichen außer Zeilenumbrüche (engl. *new line*). Viele Implementierungen unterstützen Optionen (engl. *flags*), um diese Ausnahme aufzuheben.

Beispiel: `.and` steht für „Hand“, „Wand“, „Rand“ und „Band“, aber nicht für „and“.

2.3 Zeichenauswahlen

Eine Zeichenauswahl repräsentiert genau ein Zeichen einer Liste, die in eckige Klammern „[ ]“ eingeschlossen notiert wird. Mit einem Bindestrich „-“ können ganze Zeichenbereiche (z.B. `a-z` für Kleinbuchstaben von „a“ bis „z“) in die Zeichenauswahl aufgenommen werden.

Falls das erste Zeichen der Auswahl der Zirkumflex „^“ ist, handelt es sich um eine *negierte* Zeichenauswahl. Diese steht dann für ein beliebiges Zeichen, dass nicht in der Liste enthalten ist.

Besonderheiten der Notation:

- Falls die Auswahl das Zeichen „]“ enthalten soll, muss dieses als *erstes Zeichen* der Liste notiert werden. Leere Zeichenauswahlen existieren somit nicht!
- Um den Bindestrich „-“ in die Auswahl aufzunehmen, sollte er als erstes oder letztes Zeichen notiert werden, um seine Sonderbedeutung für die Notation von Zeichenbereichen aufzuheben.
- Damit der Zirkumflex „^“ von der Auswahl getroffen wird, darf er trivialerweise *nicht* das erste Zeichen der Liste sein.
- Andere Metazeichen (inklusive „\“) verlieren innerhalb der Zeichenauswahl ihre Sonderbedeutung und müssen folglich nicht maskiert werden.

Für einige Zeichenauswahlen existieren vordefinierte *Zeichenklassen*, die innerhalb einer Zeichenauswahl anhand ihres Namens zwischen „[:“ und „:]“ notiert werden.

Verfügbare Zeichenklassen:

`[:alnum:]`	Buchstaben & Ziffern	`[:blank:]`	Leerzeichen & Tabulator
`[:alpha:]`	Buchstaben	`[:space:]`	Jede Art von Leerzeichen
`[:lower:]`	Kleinbuchstaben	`[:punct:]`	Interpunktionszeichen
`[:upper:]`	Großbuchstaben	`[:print:]`	Alle druckbaren Zeichen
`[:digit:]`	Ziffern	`[:graph:]`	Druckbare, nicht-leere Zeichen
`[:xdigit:]`	Hexadezimal-Ziffern	`[:cntrl:]`	Kontrollzeichen

Zeichenklassen schließen je nach *Locale*-Systemeinstellung auch internationale Zeichen ein.

2.4 Quantoren

Quantoren definieren, wie oft ein Element des Ausdrucks hintereinander vorkommen darf oder muss. Sie werden stets unmittelbar *nach* dem Element notiert. In der allgemeinen Schreibweise werden Quantoren durch ein oder zwei natürliche Zahlen `n` und `m` in geschweiften Klammern notiert:

{`n`}	**genau** `n` Vorkommen
{`n,`}	**mindestens** `n` Vorkommen
{`n,m`}	**mindestens** `n` und **höchstens** `m` Vorkommen

Darüber hinaus gibt es noch Kurzschreibweisen für besonders häufig verwendete Quantoren:

`?`	Vorkommen **optional**	($\hat{=}$ {`0,1`})
`*`	**beliebig viele** Vorkommen	($\hat{=}$ {`0,`})
`+`	**mindestens ein** Vorkommen	($\hat{=}$ {`1,`})

2.5 Unterausdrücke

Mit runden Klammern „`(` `)`“ können beliebige Elemente zu Unterausdrücken (engl.: *sub pattern*) gruppiert werden. Dies ist insbesondere in Zusammenspiel mit Quantoren hilfreich, da so komplexe Ausdrücke wiederholt werden können.

In vielen Anwendungen kann man zudem durch die Notation `\1`, `\1`, ..., `\n` einen Rückbezug (engl.: *back reference*) auf den vom `i`-ten Unterausdruck getroffenen Teilstring innerhalb des Patterns herstellen. In Suchen-und-Ersetzen-Anwendungen (z.B. in Editoren) lässt sich dieser Rückbezug oft auch im Ersatz-String notieren.

Beispiel: Ersetze „`(([[:alpha:]]+) \1`“ mit „`Doppeltes Wort: \1`“

2.6 Alternativen

Mit dem Pipe-Zeichen “ lässt sich der (Unter-)Ausdruck in „oder“-verknüpfte Alternativen aufteilen. Der gesamte Ausdruck steht dann für alle Strings, die eines der Alternativ-Muster abdeckt.

Beispiel: `true|false` steht für einen booleschen Wert aus Kleinbuchstaben.

2.7 Zeilenanfang und -ende

Der Zirkumflex „^“ steht innerhalb eines regulären Ausdrucks für den Zeilenanfang, das Dollar-Zeichen „$“ für das Zeilenende.

Beispiel: `^$` steht für eine Leerzeile.

3 Unterschiede in den Notationsarten

3.1 Implementierungsspezifische Erweiterungen

Viele Implementierungen von regulären Ausdrücken – insbesondere die *Perl-komaptiblen regulären Ausdrücke* (PCRE) – unterstützen weitere Notationsmöglichkeiten, Kurzschreibweisen und Flags. Beispielsweise kann häufig die Kurzschreibweise `\w` für einen „Wortbuchstaben“ verwendet werden (entspricht in etwa `[[:alnum:]_]`) und `\b` repräsentiert Wortgrenzen.

Einige dieser Schreibweisen werden auch von vielen UNIX-Tools unterstützt. Für weitere Informationen sei hier auf die MAN-Pages der jeweiligen Tools verwiesen.

3.2 Unterschiede zwischen „Basic“ und “Extended“ Regular Expressions

Wie bereits erwähnt unterscheiden viele UNIX-Tools zwischen *basic regular expressions* (BRE) und *extended regular expressions* (ERE). Oft sind auch beide Notationsarten möglich und lassen sich über geeignete Parameter (z.B. `-E`) umstellen.

Bisher bezogen sich alle vorgestellten Notationen auf ERE. In folgenden Punkten unterschiedet sich die BRE-Notation:

- Die Zeichen „`| + ? { } ( )`“ sind keine Metazeichen, sondern normale Zeichenliterale.

- Die Klammern für Quantoren und Unterausdrücke werden mit vorangestellten Backslash „\“ notiert: „\{“, „\}“, sowie „\(“ und „\)“
- Zirkumflex „^“ und Dollar „$“ zur Notation von Zeilenanfang und -ende besitzen ihre Sonderbedeutung nur am Anfang, bzw. am Ende des (Unter-)Ausdrucks, ansonsten werden sie als Literale interpretiert.
- Es gibt keine Notation für Alternativen (ERE: Pipe-Zeichen „—“)
- Die Quantoren „+“ und „?“ müssen in der allgemeinen Schreibweise als `{1,}` und `{0,1}` notiert werden.

4 Anwendungsbeispiele

4.1 grep

Grep ist ein Kommandozeilenprogramm zur Suche und Filterung von Text unter der Verwendung von Regulären Ausdrücken. Der Name leitet sich vom Kommando „g/re/p“ des UNIX-Editors `ed` ab: „Suche global (`g`) nach dem regulären Ausdruck `re` und gebe ihn aus (print – `p`)“.

Das Programm durchsucht Eingabedateien oder die Standardeingabe (z.B. bei „Pipe-Operationen“ auf der UNIX-Shell) und schreibt jede Zeile mit Suchtreffern in die Standardausgabe.

Die Einsatzgebiete von `grep` umfassen u.a. das Durchsuchen mehrerer Dateien und das Filtern von Informationen. Es wird auch sehr häufig im Zusammenspiel mit anderen Programmen verwendet – z.B. mit `find`, um ganze Verzeichnisbäume zu durchsuchen oder mit `ls`, um Dateien eines bestimmten Musters herauszufiltern.

Standardmäßig verarbeitet `grep` *basic regular expressions*. Durch den Aufruf als `egrep` oder mit dem Parameter `-E` werden statt dessen *extended regular expressions* verwendet.

Aufrufsyntax und Optionen (Auszug):

```
[e]grep [OPTION] PATTERN [FILE]
```

`-E`	Verwende EREs statt BREs (vgl. `egrep`)
`-o`	Nur tatsächliche Treffer statt ganzer Zeilen ausgeben
`-v`	Nur Zeilen *ohne* Treffer ausgeben
`-i`	Groß-/Kleinschreibung ignorieren
`-c`	Nur Anzahl der gefundenen Zeilen ausgeben
`-l`	Nur die Dateinamen mit Treffern zeigen
`-n`	Zusätzlich Zeilennummern ausgeben
`-s`	Unterdrückt Fehlermeldungen in der Ausgabe

4.2 sed

Das UNIX-Tools `sed` ist ein nicht-interaktiver, zeilenorientierter Texteditor. Nicht-interaktiv bedeutet, dass der Benutzer im Gegensatz zu anderen Editoren wie `vi` oder `emacs` die Dateien nicht selbstständig editiert, sondern vorher eine Reihe von Editierkommandos festlegt und diese dann von `sed` abarbeiten lässt. Damit eignet sich `sed` vor allem für wiederkehrende, automatisierte Textmanipulationen – z.B. für Textmanipulationen in mehreren Dateien. Wie `grep` eignet sich `sed` daher besonders für die Shellprogrammierung im Zusammenspiel mit anderen Tools.

Die Arbeitsweise von `sed` umfasst üblicherweise folgende Schritte:

- Lese die nächste Zeile in den Eingabepuffer
- Entscheide, welche Befehle für diese Zeile ausgeführt werden müssen
- Führe sukzessive alle Befehle aus
- Kopiere die bearbeitete Zeile in die Standardausgabe

Durch die Bearbeitung überschreibt `sed` die Eingabedateien nicht, sondern gibt sie nur auf der Standardausgabe aus. Um eine Rückspeicherung auszuführen, kann die Ausgabe von `sed` beispielsweise mit dem „`>`“-Operator der Shell wieder in die Eingabedatei zurückgeschrieben werden.

Aufrufsyntax und Optionen (Auszug):

```
sed [-E] [-n] [-e COMMAND] [-f SCRIPT] [FILE]
```

`-E`	Verwende EREs statt BREs
`-n`	Zeilenpuffer nicht implizit in die Ausgabe schreiben
`-e COMMAND`	Füge den Befehl COMMAND der Befehlsliste hinzu
`-f SCRIPT`	Lese Befehle aus der Skript-Datei SCRIPT

Falls nur ein Befehl ausgeführt werden soll, kann `-e` weggelassen werden. Jede Zeile im Befehlstext (bzw. dem Befehlsskript) kann einen `sed`-Befehl enthalten.

Aufbau eines `sed`-Befehls:

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

Eine Adresse ist entweder eine Zeilennummer (beginnend ab 1), das Dollar-Zeichen „$“ als Indikator für die letzte Zeile oder ein in Schrägstriche „/“ eingefasster regulärer Ausdruck. Falls die Endadresse nicht angegeben wird, wird die Funktion nur auf die Startadresse angewendet. Falls beide Adressen weggelassen werden, wird die Funktion auf *alle* Zeilen angewendet.

Der Funktionsumfang von `sed` ist recht groß und würde den Rahmen dieses Vortrags sprengen. Deshalb gebe ich hier nur einen kleinen Überblick über die verfügbaren `sed`-Funktionen:

`p`	Zeile explizit ausgeben
`d`	Zeile löschen (bzw. nicht ausgeben)
`q`	Scriptausführung nach dieser Zeile beenden
`s/re/str/[flags]`	Ersetzt den ersten durch den regulären Ausdruck `re` getroffenen Teilstring mit `str` (kann Back-References beinhalten). Mit dem Flag `g` (*global*) werden *alle* passenden Strings in der Zeile ersetzt

Literatur

[1] Man-Pages zu `grep`, `sed` und `re_format` (Mac OS X)

[2] Wikipedia-Artikel „Regulärer Ausdruck“
http://de.wikipedia.org/wiki/Regulärer_Ausdruck

[3] Herold, H: „Linux-Unix-Profitools“ (3. Auflage), Addison-Wesley, Bonn