sed Tutorium

Eine Einführung in sed

Thomas Pircher

sed Tutorium: Eine Einführung in sed

Thomas Pircher

Veröffentlicht 2016-11-10

Kapitel 1. Einführung

Seit ich mir ein bisschen Zeit genommen habe mich in **sed** einzuarbeiten, hat dieses Tool mir sehr viele Aufgaben erleichtert. Es ist unglaublich, wie flexibel dieses Programm einsetzbar ist und welche komplexen Regeln man mit ein paar Zeichenkombinationen aufstellen kann, die für den Laien nur wie ein beliebiges Gekrösel von Hieroglyphen ausschauen. Außerdem macht es einfach Spaß, eigene Scripte zu erstellen oder die von anderen Leuten zu verstehen. Mit dieser Einführung will ich Lust auf **sed** machen und ich hoffe dass die Einarbeitung in das Programm nicht zur Frust ausartet.

Über dieses Tutorium

Ich bin kein **sed**-Guru, deshalb könnte manches Problem sicher mit weniger Befehlen gelöst werden. Das ist aber nicht der Sinn dieser Einführung die versucht die Beispiele so nachvollziehbar wie nur möglich zu halten. Ich bin aber dankbar für jedes Feedback, für Anregungen, Kritik, Fehlermeldungen, Verbesserungsvorschläge und Blankoschecks. Sollte diese Einführung gefallen, dann sind auf der Seite https://www.tty1.net/say-thanks_de.html ein paar Ideen aufgelistet, um sich erkenntlich zu zeigen.

Die hier vorgestellten Scripte wurden in der Regel getestet, doch kann ich nicht ausschließen, dass sich doch hie und da einige Fehler eingeschlichen haben In diesem Fall bitte nicht meckern sondern melden. Für Fragen stehe ich gerne zur Verfügung. Einfach eine E-Mail an die auf der ersten Seite angegebene Adresse schicken und in der Betreff-Zeile '*sed-tutorium*' unterbringen.

Neue Versionen dieses Dokumentes können von https://www.tty1.net/sed-tutorium_de.html bezogen werden. Der Quellcode ist nun auch auf GitHub [https://github.com/tpircher/sed-tutorium] gehostet.

Was ist sed

sed ist ein kleines aber feines UNIX Programm, um Texte zu verarbeiten. Es ist im Gegensatz zu gewöhnlichen Editoren nicht interaktiv; sed arbeitet eine Eingabedatei anhand vorher festgelegter Regeln ab und gibt das Ergebnis auf die Standard-Ausgabe aus. Das macht sed sehr nützlich um eine große Anzahl von Dateien Script-gesteuert als Batch zu verarbeiten.

Ganz nebenbei lernt man durch die Beschäftigung mit **sed** eine ganze Menge über Reguläre Ausdrücke (Regular Expressions), die grundlegenden Prinzipien von UNIX und einige kleine Tricks, welche in anderen Programmen wie **vim** sehr nützlich sind.

sed wurde 1973 oder 1974 von Lee E. McMahon geschrieben (mehr zur Geschichte der frühen UNIX-Kommandos im Netizens Netbook [http://www.columbia.edu/~hauben/book/ch106.x09]). Trotz seines Alters wird **sed** immer noch verwendet, da es ein weites Spektrum von einfachen bis sehr komplexen Aufgaben erledigen kann. Die Stärke von UNIX liegt zum Teil auch darin, dass es über kleine hochspezialisierte Tools verfügt die, miteinander kombiniert, fast jede Aufgabe lösen können.

Der Name **sed** steht für Stream EDitor, was andeuten soll, dass das Programm kein Editor im gewöhnlichen Sinn ist, sondern Zeile für Zeile von einer Eingabedatei oder der Tastatur in einen Zwischenspeicher (*pattern space*) einliest wo sie nach den Anweisungen im Script bearbeitet werden. Anschließend wird der bearbeitete *pattern space* auf `stdout` (standard output, meistens der Bildschirm), ausgegeben. Die Quelldatei bleibt dabei unverändert.

Zur Verdeutlichung der Arbeitsweise von **sed** hier ein kleines Beispiel:

```
sed -e '/ignore/d'
```

Die Option `-e` erwirkt dass **sed** das darauf folgende Argument als Script behandelt, und nicht etwa als Eingabedatei. Das eigentliche (an dieser Stelle nicht weiter erklärte) Script ist in einfachen Gänsefüßchen angegeben.

Anmerkung

Wird **sed**, wie im obigen Beispiel, ohne Angabe einer Eingabedatei aufgerufen, dann wird die Eingabe von `stdin` (in diesem Fall von der Tastatur) eingelesen. Zum Beenden des Programms drückt man **^D** (CTRL und **D**) was unter UNIX für Dateiende steht. Unter Windows muss man **^Z** gefolgt von **<Enter>** eingeben.

Wenn das Programm gestartet wird, können auf der Tastatur nun Zeichen eingegeben werden, und sobald das Zeilenende eingegeben wird, beginnt **sed** diese Zeile zu bearbeiten. In obigem Beispiel wird jede Zeile auf dem Bildschirm ausgegeben, es sei denn, sie enthält den String 'ignore'.

Das Programm

Bedingt durch die weite Verbreitung von **sed** gibt es eine Reihe von Implementationen, die sich in einigen Details unterscheiden. Dieses Tutorium versucht, sich so weit als möglich an die standardisierte Version von **sed** zu halten. Dem Autor liegt GNU **sed** vor, und alle Beispiele wurden damit getestet. Im Text wird aber jeweils darauf hingewiesen, wenn Erweiterungen verwendet werden. Auf Pements sedfaq [http://sed.sf.net/sedfaq.html] findet sich eine erschöpfende Liste der verschiedenen **sed**-Implementationen und deren Unterschiede.

Eine gute Hersteller-unabhängige Dokumentation stellt die Open Group [https://www.opengroup.org/onlinepubs/007908799/xcu/sed.html] bereit.

Dokumentationen

Zu nennen ist natürlich die man-page zu **sed**, die aber eher erfahrenen Benutzer zu empfehlen ist. Ausführlicher und systematischer ist die *info* page zu sed. Damit sollte man nach der Lektüre dieser Einführung keine Probleme mehr haben.

Eine Sammlung interessanter Fragen rund um **sed**, inklusive vieler Scripte sowie eine nicht enden wollende Liste von weiterführenden Links findet man auf **sed** $HOME [http://sed.sf.net]. Besonders hinweisen möchte ich auf die dort enthaltene sedfaq [http://sed.sf.net/sedfaq.html] von Eric Pement. Das Tutorial von Carlos Jorge Duarte (do it with sed [http://sed.sf.net/grabbag/tutorials/do_it_with_sed.txt]) ist sehr lesenswert, besonders wegen der vielen sehr gut dokumentierten und zum Teil trickreichen Scripte. Wer keine Dokumentation zu **sed** hat, findet in der Datei eine kurze aber durchaus brauchbare Referenz. Sehr gut ist auch u-sedit [http://www.pement.org/sed/u-sedit3.zip] von Mike Arst, welches ein Tutorial und jede Menge Beispiele beinhaltet.

Spezifische Fragen (sofern nicht in den FAQs behandelt) kann man auch in den *dafür zuständigen* Newsgruppen stellen. Man bekommt dort meistens eine fundierte und ausführliche Antwort - eine treffende Beschreibung des Problems und die Befolgung der jeweiligen Netiquette seitens des Fragenden vorausgesetzt. Eine zuständige deutschsprachige Newsgruppe könnte beispielsweise `news:de.comp.os.unix.shell` sein.

Kapitel 2. Grundlagen

Reguläre Ausdrücke

Reguläre Ausdrücke (Regular Expressions, REs) wurden von 1956 Stephen Cole Kleene eingeführt und sie erwiesen sich als sehr effektiv, um Zeichenketten zu beschreiben.

REs werden dann verwendet, wenn man die *Form* einer Zeichenkette (String) angeben will; sie beschreiben also *Klassen* von Strings. Es ist zum Beispiel einfacher die *natürlichen Zahlen* als "eine Zeichenkette, bestehend aus einer oder mehreren Ziffern aus der Menge {0,1,2,3,4,5,6,7,8,9}" zu definieren, als alle Zahlen von 0 bis unendlich aufzuzählen. Mit Hilfe von Regulären Ausdrücken kann man solche Klassen von Strings eindeutig beschreiben.

Man sagt eine RE passt auf eine Zeichenkette, wenn diese in der von der RE umrissenen Klasse enthalten ist. Häufig werden REs verwendet, um aus einem String einen Teilstring heraus zu picken, welcher von der RE beschrieben ist. Dabei gilt das Prinzip der längsten Übereinstimmung (*longest match*), was heißen soll dass dies der längste Teilstring ist, auf den die RE passt. In diesem Zusammenhang spricht man auch davon, REs sind gefräßig (*greedy*).

Tabelle 2.1. Erweiterte Reguläre Ausdrücke (Extended Regular Expressions)

Regulärer Ausdruck	Erklärung
`x`	das Zeichen 'x'
`\x`	Escape: wenn das Zeichen nach dem `\` eines eines der folgenden ist: `{a, b, f, n, r, t, v}` dann wird es als Spezielles Zeichen gemäß ANSI C interpretiert. Zum Beispiel '`\n`' für einen Zeilenumbruch. Jedes andere `x` verliert seine Sonderbedeutung (falls es eine hat) und wird als einfaches Zeichen interpretiert, z.B. `\*` wird als Stern interpretiert und nicht als Quantifikator.
`\123`	das Zeichen mit oktalem ASCII-Code 123
`\xe5`	das Zeichen mit hexadezimalem ASCII-Code e5
`.`	jedes beliebige Zeichen außer `\n` (newline)
`[xyz]`	eine "character class": *x* ODER *y* ODER *z*
`[ako-sP]`	eine "character class" mit einer Bereichsangabe, also *a* ODER *k* ODER ein Character aus dem Bereich *o* BIS *s* ODER *P*
`[^x-z]`	eine "negated character class": Jedes Zeichen außer *x* bis *z*
`(r)`	die RE *r* selber
`rs`	die RE *r* gefolgt von der RE *s*
`r|s`	die RE *r* ODER die RE *s*
`r*`	die RE *r* null oder mehrere male
`r+`	die RE *r* ein oder mehrere male
`r?`	die RE *r* null oder ein mal
`r{2,6}`	die RE *r* zwei bis sechs mal
`r{2,}`	die RE *r* zwei oder mehrere male
`r{,6}`	die RE *r* null bis sechs mal
`r{4}`	die RE *r* genau vier mal
`^r`	die RE *r* am Anfang der Zeile
`r$`	die RE *r* am Ende der Zeile

Regulärer Ausdruck	Erklärung
`[:str:]`	mit *`str`* eine der folgenden Bezeichner: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit` dann die betreffende Charakterklasse. Siehe *ctype(3)* für Details.

Für detailliertere Informationen siehe die man-page *regex(7)* oder, falls nicht vorhanden die man-page zu *awk(1)*, welche lange Zeit auch als die Referenz für REs galt, oder *flex(1)* oder die Online-Dokumentation der Open Group [https://www.opengroup.org/onlinepubs/007908799/xbd/re.html].

Basic Regular Expressions

sed verwendet *Basic* Regular Expressions, eine Art Untermenge der oben vorgestellten Erweiterten Regulären Ausdrücke. Die Unterschiede zu den Erweiterten Regulären Ausdrücken sind:

- Die Quantifikatoren '`|`', '`+`' und '`?`' sind normale Zeichen, und es gibt keine äquivalenten Operatoren dafür. GNU **sed** kennt diese Operatoren, wenn sie durch einen vorangestellten Backslash "escaped" werden.
- Die geschwungenen Klammern sind normale Zeichen, und müssen mit Backslashs "escaped" werden, werden also als '`\{`' und '`\}`' geschrieben. Das selbe gilt für runde Klammern; die Zeichen, die durch '`\(`' und '`\)`' eingeschlossen werden, können später mit '`\1`' usw. dereferenziert werden.
- '`^`' ist ein normales Zeichen, wenn es nicht am Beginn einer Zeile oder eines Klammerausdrucks steht.
- '`$`' ist ein normales Zeichen, wenn es nicht am Ende einer Zeile oder eines Klammerausdrucks steht.
- '`*`' ist ein normales Zeichen am Beginn einer Zeile oder eines Klammerausdrucks.

Reguläre Beispiele

Die Menge der Natürlichen Zahlen kann man mit einer Basic Regular Expression wie folgt umschreiben: '`[0-9][0-9]*`'. Die einfachere RE '`[0-9]*`' passt zwar auch auf die natürlichen Zahlen, aber auch auf einen leeren String der keine Ziffer enthält: der Quantifikator '`*`' steht für null oder mehrere male. Die Bereichsklasse '`[0-9]`' hätte auch als '`[[:digit:]]`' geschrieben werden können und mit Extended REs kann man ein paar Zeichen sparen indem man den '`+`'-Quantifikator verwendet: '`[0-9]+`'.

Hier ein berühmtes Shakespearezitat in Nerd-Schreibweise:

```
(bb|[^b]{2})
```

Diese RE passt auf Strings, die entweder aus zwei 'b' bestehen oder aus zwei Zeichen verschieden von 'b'. Auf Englisch liest sich das in etwa als "two b or not two b" (sprich: to be or not to be).

Streng genommen liest sich die RE '`(bb|[^b]{2})`' als "two b or two not b". Gönnen wir uns die dichterische Freiheit und lassen die RE trotzdem als Shakespearezitat durchgehen.

Tipp

Eine Reihe von Programmen helfen die ersten Experimente mit Regulären Ausdrücken zu erleichtern. pcretest (enthalten in der PCRE library) ist eines davon, oder kregexpeditor, mit grafischer Benutzeroberfläche für KDE. Aber es geht auch einfach mit **sed**. Das folgende Script-Gerüst schreibt alle Zeilen, auf die ein Regulärer Ausdruck passen, auf den Bildschirm (der String 'RE' muss durch den gewünschten Regulären Ausdruck ersetzt werden):

```
sed -ne '/RE/p'
```

Wenn man 'interaktiv' mit **sed** arbeitet, also wenn Ein- und Ausgabe mit Tastatur und Bildschirm erfolgen (so wie im obigen Beispiel) dann sieht man sowohl Eingabe als auch Aus-

gabe auf dem Bildschirm. Das bedeutet, dass Zeilen, die nicht auf die RE passen, einmal am Bildschirm auftauchen (als Eingabe). Zeilen die hingegen auf die RE passen, erscheinen zweimal (einmal als Eingabe, einmal als Ausgabe).

Das folgende Beispiel schreibt alle Zeilen mit mindestens einer Ziffer auf den Bildschirm:

```
sed -ne '/[0-9]/p'
```

Dieses Beispiel schreibt alle Zeilen die den String "Rumpelstilzchen" enthalten auf den Bildschirm:

```
sed -ne '/Rumpelstilzchen/p'
```

Tipp

sed kennt nur Basic Regular Expressions, aber das weit verbreitete GNU **sed** ist in der Lage, Extended REs auszuführen, wenn man die Option '`-r`' angibt.

Reguläre Ausdrücke sind sehr flexibel und sehr oft kann man ein gewünschtes Ergebnis auf mehreren verschiedenen Wegen erreichen. So ist zum Beispiel die Extended RE '`(x|y|z)`' äquivalent zu '`[xyz]`' und '`(a|b)`' ist äquivalent zu '`(b|a)`'.

Die RE '`[FB]all`' passt sowohl auf "Fall" als auch auf "Ball". Das selbe Ergebnis könnte mit einer Extended RE zum Beispiel auch so erreicht werden: '`(F|B)al{2}`'.

Die RE '`^#.*`' passt auf alle Zeilen, die mit einem '#' anfangen. Ist die deutlich kürzere RE '^#' äquivalent zur vorhergehenden? Zur reinen Mustersuche passen beide REs auf die selben Zeilen, der Unterschied kommt dann zu Tage, wenn man die gefundenen Muster weiterverarbeiten will. Erstere RE benennt die ganze Zeile, vom Beginn bis zum newline, zweitere benennt nur das Zeichen '#' am Anfang der Zeile.

Grenzen von REs

Mit REs lassen sich nicht alle Zeichenketten beschreiben. Es ist zum Beispiel unmöglich ein System von balancierten Klammern zu beschreiben, auch ist die Menge {wcw | w ist ein String bestehend aus 'a's und 'b's} als RE nicht auszudrücken. Mehr zu REs kann man im 'Drachenbuch', *Compilers - Principles, Techniques and Tools* von Aho, Sethi und Ullman nachlesen.

REs können sehr schnell unleserlich werden. Eine Gleitkommazahl wie 3.675E-15 kann man mit '`[[:digit:]]+\.[[:digit:]]*([eE][+-]?[[:digit:]]+)?`' beschreiben. Zu beachten ist der Backslash '\' vor dem Punkt, da jener eine Sonderbedeutung hat, die mit dem vorangestellten '\' unterbunden wird. Leider hat diese Beschreibung einen Nachteil: sie passt zwar auf die Zahl '1.', aber nicht auf die Zahl '.1', also noch einmal: '`(([[:digit:]]+\.[[:digit:]]*)|(\.[[:digit:]]+))([eE][+-]?[[:digit:]]+)?`' Wie man sieht, werden REs schnell unübersichtlich. Das Chaos wird in Verbindung mit **sed** und **bash** perfekt, da sich noch viele lustige '\' und '`/`' hinzugesellen werden. Dazu aber später.

Kapitel 3. Erste Schritte mit sed

Ein- und Ausgabe

sed liest gewöhnlich von `stdin` (standard input, normalerweise die Tastatur) und schreibt auf `stdout` (standard output, normalerweise der Bildschirm). Man kann aber nach den Kommandozeilenoptionen einen (oder mehrere) Dateinamen angeben, von dem die Eingabe gelesen werden soll. Weiters kann man sich der *Umleite-Operatoren* der Shell bedienen (`>`, `<`, `|`). Die drei folgenden Zeilen liefern das selbe Ergebnis:

```
sed -n -e '/root/p' /etc/passwd
sed -n -e '/root/p' < /etc/passwd
cat /etc/passwd | sed -n -e '/root/p'
```

Das Script '`/root/p`' liest die Eingabedatei ein (`/etc/passwd`) und schreibt nur jene Zeile(n) auf den Bildschirm die das Wort 'root' enthalten.

Anmerkung

Noch eine kleine Besserwisserei meinerseits, die mit **sed** eigentlich nichts zu tun hat, sondern mit Shell-Scripting. Von den beiden Zeilen

```
programm 2>&1 >file
```

und

```
programm >file 2>&1
```

ist die zweite Version vorzuziehen, da die erste Kommandozeile `stderr` auf den alte `stdout` setzt, und erst anschließend `stdout` auf `filename` umlenkt; `stderr` wird also i.d.R. nicht nach `filename` umgelenkt werden.

Die meisten UNIX-Kommandos lassen sich als *Filter* einsetzen. Filter werden dazu verwendet um einen Stream von Daten durch mehrere mit *pipes* (`|`) verkettete Programme zu jagen. Eine Pipe macht nichts anderes als `stdout` des Programms auf der linken Seite mit `stdin` des Programms auf der rechten Seite zu verknüpfen. Auf diese Weise lassen sich in Verwendung von verschiedenen spezialisierten Programmen sehr komplexe Aufgaben erledigen.

Kommandos

Das Programm

```
sed -e 'd' /etc/services
```

liefert erst mal gar nix.

Wie die Verarbeitung einer Zeile zu erfolgen hat, wird in einem *Script* oder *Programm*, festgelegt, das auf der Kommandozeile der Option '`-e`' folgen muss. Ein **sed**-Script enthält mindestens ein Kommando (in diesem Fall '`d`' für *delete*).

Anmerkung

Eigentlich ist die Angabe der Option '`-e`' nicht zwingend notwendig. Wenn diese Option weggelassen wird, dann wird das erste Argument als Script und alle folgenden Argumente als Dateinamen interpretiert. In dieser Einführung wird die Option '`-e`' aber immer explizit angegeben um Missverständnisse zu vermeiden.

Die Funktionsweise eines Scriptes ist wie folgt: eine Zeile des Eingabe-Streams wird in den Arbeitsspeicher (*pattern space*) geladen, welcher dann nach den angegebenen Regeln bearbeitet (im Beispiel oben wird er gelöscht) und anschließend ausgegeben wird. Die Eingabedatei wird dabei nicht verändert. Diese Schritte werden Zeile für Zeile wiederholt, bis das Dateiende erreicht ist.

Bitte die beiden Anführungszeichen ' ' um das **sed**-Script 'd' beachten. Diese verhindern eine Re-Interpretation der dazwischen liegenden Zeichen seitens der *Shell*. Diese Anführungszeichen sollten immer verwendet werden, da man sich dadurch unerwartetes Verhalten des Scriptes ersparen kann.

Anmerkung

Die zeilenorientierte Arbeitsweise von **sed** eignet es sich sehr gut um Texte zu bearbeiten. Binäre Daten werden kaum mit **sed** bearbeitet, da sie sich nur umständlich als Reguläre Ausdrücke angeben lassen und weil je nach Beschaffenheit der Eingabe-Daten sehr große Blöcke in den *pattern space* geladen werden müssen.

Adressen

Den meisten Kommandos kann man eine *Adresse* voranstellen. Eine solche Adresse bestimmt, welche Zeilen mit dem betreffenden Kommando zu bearbeiten sind. Somit kann man Kommandos selektiv auf bestimmte Zeilen, Blöcke oder wie wir später sehen werden, auf bestimmte Zeichenketten, anwenden.

Eine Adresse ist zum Beispiel eine fixe Zeilennummer in einer Datei oder ganze Bereiche, oder aber Zeilen die auf einen bestimmten Reguläre Ausdruck passen.

```
sed -e '1d' /etc/services
```

Hier wird das Kommando 'd' auf die Zeile mit der Adresse '1' angewendet. Der Effekt des Programms ist der, dass die erste Zeile von `/etc/services` in den *pattern space* geladen, dieser dann gelöscht und anschließend der leere *pattern space* (also nichts) ausgegeben wird. Alle anderen Zeilen werden in den *pattern space* geladen, der nicht bearbeitet wird da die Adresse nicht auf die Zeile zutrifft und anschließend wird der *pattern space* nach `stdout` geschrieben. Das Ergebnis des Scriptes ist eine Kopie der Eingabedatei, in der die erste Zeile fehlt.

Man kann auch Adressbereiche angeben wie in

```
sed -e '1,10d' /etc/services
```

was die ersten 10 Zeilen löscht oder man kann jede n-te Zeile bearbeiten wie in

```
sed -e '10~2d' /etc/services
```

wo jede zweite Zeile, ausgehend von der 10. Zeile gelöscht wird. Letzteres ist eine GNU-Erweiterung von **sed**; dort wo Portabilität auf andere Umgebungen wichtig ist, ist diese Adresse zu vermeiden.

Manchmal ist es interessant, nur solche Zeilen einer Konfigurationsdatei anzuzeigen, die nicht auskommentiert sind. Das kann mit folgender Zeile geschehen:

```
sed -e '/^#.*/d' /etc/inetd
```

Die Adresse `/re/` wendet das nachfolgende **sed**-Kommando auf jede Zeile an, auf die der Reguläre Ausdruck *re* passt. Nur zur Erinnerung - die angegebene RE passt auf jede Zeile, welche "mit einem '#' beginnt und danach null oder mehr beliebige Zeichen enthält". Das ist aber nicht das was wir eigentlich wollten. Denn enthält die Datei eine leere Zeile, dann wird diese auch ausgegeben. Also müssen wir unsere Strategie ändern und z.B. nur jene Zeilen ausgeben, die mit einem Zeichen beginnen, das nicht '#' ist:

```
sed -e '/^[^#].*/p' /etc/inetd
```

Das ist neu: das Kommando `p`, das für *print* steht, also den *pattern space* ausgeben. Die Ausgabe ist aber alles Andere als erwartet: jede Zeile wird ausgegeben, die erwünschten Zeilen sogar zwei mal. Was ist passiert? Noch einmal müssen wir die Funktionsweise von **sed** durchkauen: Zeile einlesen, wenn die Adresse passt, dann *pattern space* bearbeiten (in unserem Falle ausgeben), dann *pattern space* ausgeben. Wir müssen also den letzten Schritt unterbinden. Das geht mit der Option `-n` (portabel!) oder `--quiet` oder `--silent`, je nach Geschmack. Das richtige Programm schaut nun so aus:

```
sed -n -e '/^[^#].*/p' /etc/inetd
```

Wir haben gesehen, dass mit der Adresse '*n*,*m*' die *n*-te bis *m*-te Zeile bearbeitet wird. Das geht auch mit REs: die Adresse '*/BEGIN/*,*/END/*' selektiert alle Zeilen ab der ersten Zeile, auf die die RE '*BEGIN*' passt bis zu der Zeile auf die die RE '*END*' passt oder bis zum Dateiende, je nach dem was früher kommt. Wird '*BEGIN*' nicht gefunden, dann wird keine Zeile bearbeitet. Es ist oft so, dass man beim Compilieren eines umfangreichen Projektes regelrecht von Fehlermeldungen und Warnungen erschlagen wird. Das ist ein Job für **sed**: das folgende Beispiel liefert nur jene Ausgaben des **gcc** die zwischen der ersten Warnung und der ersten Fehlermeldung liegen.

```
gcc sourcefile.c 2>&1 | sed -n -e '/warning:/,/error:/p'
```

Und wenn '*n*,*m*' gilt und '*/BEGIN/*,*/END/*', warum nicht auch eine Kombination davon? Ein '*/BEGIN/*,*m*' heißt ab der Zeile, auf welche die RE '*BEGIN*' passt bis zur *m*-ten Zeile usw.

Das folgende Beispiel ist wohl das kürzeste sinnvolle Script in **sed**, das es gibt. Es gibt die Zeilenanzahl der bearbeiteten Datei aus (**wc** `-l`):

```
sed -n -e '$='
```

Das *Dollar*-Zeichen '$' ist in diesem Fall nicht das Zeilenende einer RE (es fehlen nämlich die `//`), sondern ist die Adresse der letzten Zeile der letzten Eingabe-Datei, und das Kommando '=' gibt die aktuelle Zeilennummer *vor* der Ausgabe aus.

Ein Rufezeichen '!' nach einer Adresse verkehrt diese in ihr Gegenteil um. Die Adresse *n*,*m*! trifft auf alle Zeilen außer Zeilen *n* bis *m* zu. Die Adresse '/awk/!' selektiert alle Zeilen die nicht die Zeichenkette 'awk' enthalten.

Mehr Kommandos

Neben den Kommandos 'd' und 'p' die wir schon kennen gibt es noch eine Reihe anderer Kommandos, die aber nicht alle in dieser Einführung beschreiben werden. Hat man erst einmal die Syntax eines **sed** Programms verstanden, findet man sich leicht in der man/info-page zurecht und man kann sie dort nachschlagen.

Ein einfaches Kommando ist 'q', das das Programm abbricht. Ob der *pattern space* noch geschrieben wird hängt davon ab, ob die Option `-n` angegeben wurde oder nicht. Als Beispiel folgen zwei funktionsmäßig äquivalente Emulationen des UNIX-Befehls **head**, wobei die zweite Lösung effizienter ist, da sie nur die ersten 10 Zeilen bearbeiten muss.

```
sed -n -e '1,10p'
sed -e '10q'
```

Weiters wird auch das Kommentarzeichen '#' als Kommando bezeichnet. **sed** ignoriert alle nachfolgenden Zeichen im *Script* bis zum Ende der Zeile. Das ist nützlich in Scripten, die in Dateien geschrieben wurden und die an trickreichen Stellen ein paar erklärende Worte verlangen.

Ein wichtiges Kommando ist 's*/re/rep/flag*'. Hierbei wird diejenige Portion im *pattern space*, auf welche die RE '*re*' passt durch die Zeichenkette '*rep*' ersetzt und zwar in der Modalität, die mit dem *flag* bestimmt wird. Ein 'd' ersetzt das erste Muster und fängt dann einen neuen Zyklus an. Das Flag 'g' ersetzt *alle* Muster in einer Zeile, eine Nummer '*n*' veranlasst sed, das *n*-te übereinstimmende Muster zu ersetzen. Mit dem Einzeiler

```
sed -e '/ich/s/€1500/€3000/g' Gehaltsliste.dat
```

kann man ein bisschen träumen. (Bei den Träumen wird es wohl bleiben, denn **sed** verändert die Datei nicht!) Wer jetzt denkt die Ausgabe mittels Ausgabeumleitung '>' wieder auf die Eingabedatei umzuleiten, der wird sich schön wundern: die Datei ist dann nämlich leer.

Tipp

Der richtige wenn auch umständliche Weg eine Datei mit **sed** zu verändern ist die Ausgabe in eine temporäre Datei umzuleiten und diese dann auf den Namen der Quelldatei umzube-

nennen. GNU**sed** kennt die Option `-i` welche das Programm veranlasst, die Eingabe-Datei direkt (inline) zu editieren.

Noch nicht verstanden was das vorherige Beispiel gemacht hat? In der Zeile, die 'ich' enthält wird das Gehalt von €1500 auf €3000 verdoppelt, alle anderen Zeilen werden unverändert ausgegeben.

Die folgende Zeile lässt die Ausgabe des Shell-Kommandos **ls** hingegen sehr 'l33t' aussehen:

```
ls -l /|sed -e 's/o/0/g'|sed -e 's/l/1/g'|sed -e 's/e/3/g'
```

Das ist als Kommandozeile ein wenig lang - könnte man nicht... ja man kann das alles kompakter schreiben, indem man mehrere **sed**-Kommandos durch Strichpunkte trennt.

```
ls -l /|sed -e 's/o/0/g;s/l/1/g;s/e/3/g'
```

Will man dem blinden Zorn des *Superusers* aus dem Wege gehen und eine Verhunzung seiner Homedirectory vermeiden, muss man die Adresse `'/ root$/!'` den Kommandos voranstellen. Diese Adresse selektiert jede Zeile, die nicht mit ' root' endet. Um mehrere Kommandos auf eine Adresse zu binden, müssen diese gruppiert werden. Das geschieht mit den geschwungenen Klammern `{}`. Wichtig: auch nach dem letzten Kommando muss ein Strichpunkt gesetzt werden.

```
ls -l /|sed -e '/ root$/!{s/o/0/g;s/l/1/g;s/e/3/g;}'
```

Das folgende Script zeigt dazu ein Beispiel und kann dazu verwendet werden, 8 Leerzeichen in ein Tabulatorzeichen zu verwandeln.

```
sed -e 's/ \{8\}/^t/g'
```

wobei das `^t` ein tab-Zeichen symbolisieren soll. Alles schön und gut, nur ist die tab-Taste unter der Shell für das schöne Wort Kommandozeilenvervollständigung reserviert, ein Tabulatorzeichen selber kann man nicht direkt eingeben. Der einfachste Weg dazu ist die Tastenkombination **^V^I** zu drücken, was für **CTRL-V CTRL-I** steht. Ein **^V** fügt das nachfolgende Zeichen ohne weitere Interpretation auf der Kommandozeile ein. Alternativ kann man also auch **^V**<tab> tippen. Mehr dazu in den infopages zu *bash*, *tcsh* oder *readline*, sowie bei Ihrem Arzt oder Apotheker.

Anmerkung

Es sei noch einmal angemerkt, dass für Basic Regular Expressions die Zeichen `+` und `?` keine Sonderbedeutung haben.
GNU**sed** kennt dagegen `\+` und `\?`. Alternativ dazu kann man mit GNU**sed** auch die Option `-r` verwenden. Diese Option bewirkt, dass alle REs als Extended REs interpretiert werden. Da die gewünschten Effekte sich aber relativ leicht mit Standard Bordmitteln von **sed** erreichen lassen, empfiehlt es sich, diese unportablen Erweiterungen selten oder gar nicht zu verwenden.

Ein weiteres nützliches Kommando ist '`y/SOURCE-CHARS/DEST-CHARS/`', welches alle Zeichen in *`SOURCE-CHARS`* in das entsprechende Zeichen in *`DEST-CHARS`* umwandelt. Unnütz zu sagen, dass beide Charakter-listen die gleiche Anzahl von Zeichen enthalten müssen. Das folgende Script 'verschlüsselt' den Text mit der sogenannten 'rot-13' Methode: alle Buchstaben werden um 13 Zeichen verschoben - aus 'a' wird 'n', aus 'b' wird 'o' usw, der Einfachheit halber hier nur für Kleinbuchstaben:

```
sed -e 'y/abcdefghijklmnopqrstuvwxyz/nopqrstuvwxyzabcdefghijklm/'
```

Das obige Beispiel ist auch ein schönes Exempel für eine umständliche Benutzung von **sed**. Den gewünschten Effekt kann man mit **tr** und weniger Getippe erreichen;

```
tr '[a-z]' '[n-za-m]'
```

Ein Nachtrag zu den geschwungenen Klammern '`{}`': Aus der Sicht von **sed** ist die öffnende Klammer '`{`' ein Kommando, dem eine Adresse oder ein Adressbereich vorangestellt werden kann. Das lässt sich für einen Trick missbrauchen, denn wenn man die Kommandos '`=`', '`a`', '`i`', oder '`r`' (erlauben höchstens eine Adresse; zur Bedeutung dieser Kommandos bitte die Dokumentation bemühen) auf

einen Adress*bereich* anwenden will, kann man sie in geschwungene Klammern setzen. So ist z.B. '`1,9=`' ein ungültiges Kommando, aber '`1,9{=;}`' ist ist nicht zu beanstanden. Der Effekt dieses Programms ist dass die Zeilen von 1 bis 9 mit vorangestellten Zeilennummern ausgegeben werden, der Rest der Datei wird unverändert wiedergegeben.

Tipp

Weil es oft gebraucht wird, stelle ich noch Scripte zur Umwandlung von Dateien im DOS-Format (CR/LF) ins UNIX-Format (LF) und umgekehrt vor. Sie wurden der schon erwähnten sedfaq [http://sed.sf.net/sedfaq.html] von Eric Pement entnommen.

```
# 3. Under UNIX: convert DOS newlines (CR/LF) to Unix format
sed 's/.$//' file    # assumes that all lines end with CR/LF
sed 's/^M$// file    # in bash/tcsh, press Ctrl-V then Ctrl-M
# 4. Under DOS: convert Unix newlines (LF) to DOS format
C:\> sed 's/$//' file                    # method 1
C:\> sed -n p file                       # method 2
```

Eine Randbemerkung: Ist keine `-e` Option angegeben, dann wird der erste Parameter, der keine Option ist als das auszuführende Programm genommen. Um Verwirrung zu vermeiden empfiehlt sich immer ein `-e` anzugeben. Einem Guru wie Herrn Pement sei es aber gestattet sich über diese Faustregel hinwegzusetzen.

UNIX wäre nicht UNIX, wenn es nicht unzählige andere Methoden dafür gäbe: beispielsweise die Programme **dos2unix** bzw. **unix2dos**, oder der Befehl `tr -d [^M] < inputfile > outputfile` um vom DOS- ins UNIX-Format zu konvertieren, oder `:set fileformat=dos` bzw. `:set fileformat=unix` unter **vim** oder...

Kapitel 4. Ein paar interessantere Beispiele

Wer bis hierher gekommen ist, sollte wirklich verstanden haben was *Adressen* und was *Kommandos* sind. Das ist wichtig, denn ab jetzt werden diese in einem **sed**-Script hintereinander gehängt, und das kann sonst schon für einige Verwirrung sorgen.

Hin und wieder trifft man in Scripten nicht die gewohnte Form '`/r/`' einer RE vor - die *Slash*es '/' scheinen zu fehlen. Das hat den Grund, dass es manchmal nötig ist in einer RE den *Slash* selber anzugeben. Damit dieser aber nicht fälschlicherweise interpretiert wird, muss er mit dem *Backslash* escaped werden, also '`\/`'. **sed** gibt einem die Möglichkeit, ein anderes Zeichen als den *Slash* als RE-Begrenzer zu verwenden. Man kann also '`/\/bin\/ls/`' oder beispielsweise '`\@/bin/ls@`' verwenden. In gleicher Weise kann das mit dem `s`- oder `y`-Kommando geschehen: '`s//`' ist gleichwertig zu '`s@@`' Hat man deshalb nicht genau verstanden, was Adresse was Kommando und was RE ist, kommt man da leicht ins Schleudern.

Probleme mit REs

Reguläre Ausdrücke sind greedy, finden also immer den längsten passenden String. Das kann manchmal unerwünscht sein. Will man zum Beispiel eine HTML-Seite in Text umwandeln, dann könnte man in Versuchung kommen folgendes Script zu verwenden:

```
sed -e 's/<.*>//g' text.html
```

Das liefert aber nicht den gewünschten Effekt, denn eine Zeile

```
Das <b>ist</b> ein <i>Beispiel</i>.
```

wird zu

```
Das.
```

verkrüppelt. Man muss also nur jene Zeichen bis zum *ersten* '>' löschen:

```
sed -e 's/<[^>]*>//g' text.html
```

Muss man einen Text nicht bis zum ersten Vorkommen eines Zeichens sondern einer Zeichenkette bearbeiten, wird die RE ein bisschen komplizierter. Im Kapitel mit den Beispielen findet sich dazu ein Lösungsansatz (Löschen von Kommentaren).

Selektives Ersetzen

Das `s///` Kommando kann nicht nur fixe Strings einsetzen, sondern auch den gefundenen String oder Substrings davon. Der *Ampersand* '&' steht dabei für den gesamten gefundenen String.

In meiner Kindheit hatten wir die *elleff*-Sprache, unsere Geheimsprache, bei der man jeden Vokal (oder Gruppe von Vokalen) in einem Wort mit `<VOKAL>l<VOKAL>f<VOKAL>` ersetzen muss. Kompliziert? Da ist die **sed**-Schreibweise einfacher:

```
sed -e 's/[aeiou][aeiou]*/&l&f&/g'
```

Die Mächtigen der Welt, als 'Bilifill Clilifintolofon' oder 'Boloforilifis Jelefelzilifin' ausgesprochen, gewinnen damit in meinen Augen sofort an Sympathie. Meine Hochachtung jedem, der ein ver*elleff*tes 'ukulele' aussprechen kann ohne es vom Bildschirm zu lesen.

Mit GNU **sed** kann man folgende Zeile schreiben:

```
sed -e 's/[aeiou]\+/&l&f&/g'
```

Bitte den *Backslash* '\' vor dem Plus beachten, da dieses Zeichen - weil GNU-Erweiterung - zuerst als normaler Charakter angesehen wird und seine Bedeutung die er bei REs inne hat, erst durch den Backslash gewinnt. Gleiches gilt auch für das *Fragezeichen* (*Questionmark*) '?', nicht aber für den *Asterisk*en '*'.

Hier weise ich noch einmal auf die Grenzen von Regulären Ausdrücken hin. Es ist nicht möglich, die Rücktransformation aus der *elleff*-Sprache mit REs auszudrücken. Ein `[aeiou]l[aeiou]f[aeiou]` kann man wohl angeben, nicht aber die Bedingung dass alle drei Vokale gleich sein müssen. Ob dies hinreichend ist um die *elleff*-Sprache als sichere Verschlüsselungsmethode zu bezeichnen, müssen wohl findigere Kryptologen entscheiden.

Mit **sed** ist es auch möglich, Teile von Strings heraus zu picken um diese später zu verwenden. Diese Teile werden mit '`\(`' und '`\)`' markiert, und man kann auf diese Strings mit '`\1`', '`\2`' usw. zugreifen. Nehmen wir einmal an wir hätten eine Datei, in dem verschiedene Namen eingetragen sind:

```
Alan Mathison Turing
Claude Elwood Shannon
Grace Murray Hopper
John von Neumann
Ada Lovelace
```

die in die Form `<VORNAME> [<INITIAL ZWEITER NAME>.] <NACHNAME>` gebracht werden soll. Dazu muss man erst die Bereiche definieren:

```
sed -e 's/^[^ ][^ ]* [[:alpha:]].* [^ ][^ ]*$//'
```

Nun gibt man um die gewünschten Zonen die Klammern und stellt sich das Ergebnis mit '\1' und '\2' und '\3' zusammen:

```
sed -e 's/\(^[^ ][^ ]*\) \([[:alpha:]]\).* \([^ ][^ ]*\)$/\1 \2. \3/'
```

und voilà das Ergebnis:

```
Alan M. Turing
Claude E. Shannon
Grace M. Hopper
John v. Neumann
Ada Lovelace
```

Will man das Ergebnis noch in eine Adressdatenbank importieren, dann muss man einen Feldbezeichner vor die Namen setzen. Ein erster Versuch wäre der, das gleich in einem Rutsch mit dem Script

```
sed -e 's/\(^[^ ][^ ]*\) \([[:alpha:]]\).* \([^ ][^ ]*\)$/name: \1 \2. \3/'
```

zu bewerkstelligen, das liefert aber genau da ein falsches Ergebnis, wenn der zweite Vorname fehlt.

```
name: Alan M. Turing
name: Claude E. Shannon
name: Grace M. Hopper
name: John v. Neumann
Ada Lovelace
```

Einem solchen nur teilweise formatierten Datenhaufen ist nur schwer beizukommen. Deshalb den Output ungetesteter Scripte immer zuerst auf eine temporäre Datei umleiten, diese auf Korrektheit prüfen und dann die Zieldatei ersetzen. Wie man die Namen nun richtig formatiert, wird im nächsten Kapitel beschrieben. Warum hat das Script aber nicht richtig gearbeitet? Damit die RE auf eine Zeile zutrifft, muss diese mindestens 3 Felder, durch Leerzeichen getrennt, enthalten. Das ist bei Frau Lovelace nicht der Fall, deshalb wird auch das Kommando nicht ausgeführt und der *pattern space* wird unberührt gelassen.

Gruppieren von Kommandos

Ein **sed**-Script kann mehrere Kommandos enthalten, die nach einander abgearbeitet werden. Das kann man auf mehrere Wege erreichen: Man kann zwei Kommandos im selben Script durch einen *Semicolon* (;) trennen oder man gibt mehrere Scripts mit der Option `-e` an. Für längere Scripte empfiehlt es sich, diese in eine Datei zu schreiben und diese Scriptdatei mit der Option `-f` aufzurufen.

Eine mögliche Lösung des obigen Problems benutzt zwei Kommandos: das erste kürzt den Namen, ein zweites setzt vor alle Zeilen den String *name:*.

```
sed -e 's/\(^[^ ][^ ]*\) \([[:alpha:]]\)..* \([^ ][^ ]*\)$/\1 \2. \3/' \
  -e 's/../name: &/'
```

oder man trennt die zwei Anweisungen durch einen Strichpunkt (`;`). Zu beachten ist in der zweiten Anweisung die RE '`..*`'; würde man nur einen Punkt schreiben, passte dieser Ausdruck auch auf leere Zeilen. Das wird mit zwei Punkten vermieden.

Diese Script, in eine Datei geschrieben, schaut so aus:

```
s/\(^[^ ][^ ]*\) \([[:alpha:]]\)..* \([^ ][^]*\)$/\1 \2. \3/
s/..*/name: &/
```

Anmerkung

Und wieder eine Bemerkung die nichts mit **sed** zu tun hat: Die *Shell* gibt einem die Möglichkeit Scripte wie normale Programme zu behandeln. Dazu muss man nur an den Anfang des Scriptes die Zeile '`#!`*`/pfad/zum/programm`*' setzen und die Scriptdatei als ausführbar markieren. Wenn diese Datei nun gestartet wird, ruft die Shell den angegebenen Interpreter mit dem Scriptnamen als Parameter auf. Auf das vorhergehende Beispiel angewandt sieht das so aus:

```
#!/bin/sed -f
s/\(^[^ ][^ ]*\) \([[:alpha:]]\)..* \([^ ][^]*\)$/\1 \2. \3/
s/..*/name: &/
```

Die Option `-f` weist **sed** an, den nachfolgenden Dateinamen (den die Shell hinzufügt) als Script zu nehmen. Dieser Trick funktioniert nur mit Scriptsprachen, bei denen das Zeichen '#' einen Kommentar einleitet, da sonst auch die erste Zeile als Programmcode interpretiert wird. Die Zeichenkombination '`#!`' nennt man *shebang*.

Wie weiter oben beschrieben, kann man die geschwungenen Klammern '`{}`' verwenden um mehrere Kommandos auf eine Adresse anzuwenden. Dies lässt sich auch für einen kleinen Trick missbrauchen. Will man zum Beispiel das *shebang* ('`#!`') in der ersten Zeile einer Datei entfernen, kann man das so machen:

```
sed -e '1{/^#!/d;}'
```

Dieses Script löscht die erste Zeile, aber nur wenn sie mit '`#!`' beginnt. Es ist ein schönes Beispiel für die Kombination von mehreren Adressen.

Eindeutige Kodierung der Eingabe

Gelegentlich will ein Script einfach nicht funktionieren, und man verbringt Stunden damit, zu rätseln warum eine RE partout nicht auf eine Zeile passen will. Manchmal liegt das an der Eingabe-Datei, nämlich wenn sie Zeichen enthält, die man sich nicht erwartet. In solchen Fällen ist das Kommando '`l`' nützlich. Es gibt den *pattern space* in einer eindeutigen Schreibweise (welche an ANSI C angelehnt ist) auf den Bildschirm:

```
echo "versuch mich zu haschen! " | sed -ne 'l'
versuch\tmich zu haschen\307\203 $
```

Jetzt wird klar warum die Adresse '`/^versuch mich zu haschen!$/`' nicht zur Eingabe passt: erstens ist das Zeichen zwischen den ersten beiden Wörtern kein Leerzeichen sondern ein Tabluator,

zweitens ist das vermeintlich Rufezeichen in Wahrheit das Unicode Zeichen "latin letter retroflex click", und schließlich hat sich ein Leerzeichen am Ende der Zeile eingeschlichen.

Anmerkung

Das obige Beispiel ist möglicherweise nicht durch Copy&Paste nachzuvollziehen, da Sonderzeichen nicht immer korrekt kopiert werden. Wenn auf einem System das Programm **base64** vorhanden ist, dann kann man das Beispiel mit der folgenden Eingabe nachvollziehen:

```
echo dmVyc3VjaAltaWNoIHp1IGhhc2NoZW7HgyAK | base64 -d | sed -ne 'l'
```

Kapitel 5. *space*balls

Ergänzungen zum *pattern space*

sed kennt noch weitere Kommandos zur Manipulation des *pattern space*. Das Kommando '`D`' löscht den Inhalt des *pattern space* bist zum ersten newline. Ist darin anschließend noch Text enthalten, wird ein neuer Zyklus gestartet, *ohne* eine neue Input-Zeile einzulesen. Ist der *pattern space* hingegen leer, beginnt ein normaler Zyklus. Das Kommando '`N`' hängt ein newline an den Inhalt des *pattern space*, liest eine neue Zeile ein, welche nach dem newline eingefügt wird. Kann keine neue Zeile mehr eingelesen werden (Dateiende) dann wird das Programm an dieser Stelle abgebrochen. Das Kommando '`P`' gibt den Inhalt des *pattern space* bis zum ersten newline aus.

Das Beispiel dazu löscht alle *konsekutiven* Leerzeilen in einer Datei. Ist am Dateianfang eine Leerzeile, so bleibt sie erhalten, am Dateiende werden alle Leerzeilen gelöscht.

```
sed -e '/^$/N;/\n$/D'
```

Einmal *hold space* und zurück

Neben dem *pattern space*, in den die Zeile geladen und dort manipuliert wird, kennt **sed** noch den *hold space*, der zu Programmbeginn leer ist, aber durch verschiedene Befehle manipuliert werden kann. Der *hold space* wird hauptsächlich dann verwendet wenn man das Operationsfeld eines einzigen Kommandos auf mehrere Zeilen ausdehnen will oder sich Zeilen für später aufheben muss.

Das Kommando '`h`' überschreibt den *hold space* mit dem Inhalt des *pattern space*; die umgekehrte Operation wird durch das Kommando '`g`' erreicht. Es gibt auch groß geschriebene Versionen dieser Kommandos, welche den Zielspace nicht überschreiben, sondern daran ein newline gefolgt vom Inhalt des Quellspace anhängen.

Zur Verinnerlichung des Konzepts des *hold space* ein sehr einfaches Beispiel, in dem die erste Zeile zurückbehalten wird und erst nach der letzten Zeile geschrieben wird. Das Programm kopiert also die erste Zeile in den *hold space*, gibt alle anderen aus, und nach Erreichen des Dateiendes wird der Inhalt des *hold space* in den *pattern space* kopiert, der dann noch ausgegeben werden muss. Das und nichts anderes tut der folgende Einzeiler.

```
sed -n -e '1h;1!p;${g;p;}'
```

Das folgende Beispiel gibt alle Zeilen sofort aus, die nicht in einem '`/begin/,/end/`'-Block liegen, den Rest erst bei Dateiende. Im Hinblick auf ein **sed**-Programm heißt das, Zeilen im Block '`/begin/,/end/`' werden an den *hold space* angehängt. Zu beachten ist nur, dass der Befehl '`H`' dem Inhalt des *hold space* zuerst ein newline und dann der *pattern space* anhängt. Deshalb muss man bei der Ausgabe das erste Zeichen (sicher ein newline) unterdrücken.

```
sed -n -e '/begin/,/end/H;/begin/,/end/!p;${g;s/^.//;p;}'
```

Anzumerken ist hierbei noch dass **sed** den Inhalt des *pattern space* als eine Zeile ansieht, egal ob da noch ein oder mehrere newline enthalten sind. Aus diesem Grund unterdrückt das Kommando '`s/^.//`' nicht alle Buchstaben nach einem newline, sondern wirklich nur das erste Zeichen im *hold space*.

Das Kommando '`G`' hat folgenden Effekt: es wird an den *pattern space* ein newline und anschließend der Inhalt des *hold space* angehängt. Das kann man für die verschiedensten Zwecke ausnützen. Das Script

```
sed -e 'G'
```

fügt nach jeder Zeile ein Leerzeichen ein (der *hold space* ist ja leer). Mit **sed** kann man auch die Funktionsweise von **tac** (ein umgekehrtes **cat**; dreht die Reihenfolge der Zeilen um) nachbilden:

```
sed -n -e 'G;h;$p'
```

mit dem kleinen Schönheitsfehler dass am Ende eine Leerzeile zu viel ausgegeben wird - sie ist die Leerzeile, die in der ersten Zeile dem *pattern space* unnötigerweise angehängt wurde. Diesen Fehler beheben gleich beide folgenden Programme.

```
sed -n -e 'G;h;$s/.$//p'
sed -n -e '1!G;h;$p'
```

Mit dem Kommando 'x' werden die Inhalte der beiden spaces ausgetauscht. Abschließend zu diesem Kapitel möchte ich ein längeres Beispiel (Danke an Ulf Bro) vorstellen, das umgebrochene Absätze in eine einzelne Zeile umwandelt:

```
# Zeilen, die nicht leer sind werden dem Hold-Raum angehängt
# Bei Leerzeilen wird der Inhalt des Hold-Raums in den
# Pattern-Raum verlagert. Der Hold-Raum wird entleert
# Erste Newline wird entfernt, die anderen in Leerzeichen
# umgewandelt
/^$/! H
/^$/ {
    x
    s/\n//
    s/\n/ /g
    p
}
# Letzte Zeile nicht vergessen
$ {
    g
    s/\n//
    s/\n/ /g
    p
}
```

Kapitel 6. Sprünge (*branches*)

Dieses Kapitel ist für *Stirb langsam* **sed**-Programmierer (frei aus dem Englischen übersetzt) geschrieben.

Sprungkommandos

Sprungziele (*labels*) werden durch einen Doppelpunkt, gefolgt vom Namen des Labels gekennzeichnet '`: label`' wobei `label` ein beliebiger Name sein kann. Einen *unbedingten Sprung* (es wird also immer gesprungen) kennzeichnet man mit '`b label`' (b für branch). Das Sprungziel `label` muss natürlich irgendwo im Script definiert sein. Wird kein Label angegeben, dann fängt unmittelbar der nächste Zyklus an. Das Kommando '`t label`' definiert einen *bedingten Sprung*. Gesprungen wird, wenn im aktuellen Zyklus eine erfolgreiche Substitution ('`s///`'-Befehl) durchgeführt werden konnte und außerdem seither kein '`t`'-Sprung durchgeführt wurde. Auch hier gilt, wenn das Sprungziel nicht angegeben wurde, beginnt ein neuer Zyklus.

Das sind alle Kommandos in diesem Zusammenhang. In diesem Sinne kann man **sed** als echten RISC-Editor bezeichnen (RISC = Reduced Instruction Set Computer).

Achtung bei der Verwendung von '`t`', die manches Kopfzerbrechen bereiten kann. Das Große Reformations-Script soll dies verdeutlichen.

```
#!/bin/sed -f
s/foo/bar/g
s/Bayern/Bayern/g;t noconversion
s/Katholik/Protestant/g
s/kathol/luther/g
: noconversion
# und weiter gehts im Code
```

Außer der Tatsache, dass man mit einem '`/Bayern/!{...}`' besser bedient wäre, sollte der Sinn des Scriptes klar sein: In jenen Zeilen, in denen das Wort 'Bayern' nicht vorkommt, soll alles Katholische durch Protestantisches ersetzt werden. Das eigentlich nutzlose '`s/Bayern/Bayern/g`' stellt die Bedingung für den nachfolgenden Sprung dar. Diese Zeile alleine? Nein, denn das Kommando '`s/foo/bar/g`' kann genau so gut ausgeführt werden und den 2 Zeilen entfernten Sprung einleiten. Denn obwohl dies durch die eigenwillige Formatierung des Scriptes so aussieht, ist das '`t`' Kommando nicht exklusiv an das unmittelbar davor stehende Kommando gebunden. Um einen Seiteneffekt durch das 'foo-bar' Kommando zu vermeiden sollte man es tunlichst irgendwo unterhalb des Sprungkommandos unterbringen oder wenn das nicht möglich ist, dann muss ein *dummy*-Sprung eingeführt werden.

```
#!/bin/sed -f
s/foo/bar/g
t dummy
: dummy
s/Bayern/Bayern/g;t noconversion
s/Katholik/Protestant/g
s/kathol/luther/g
: noconversion
# und weiter gehts im Code
```

Andere Sprünge

Auch andere Befehle wie '`q`', '`d`' und '`D`' (bei Dateiende auch '`n`' und '`N`') verändern den Programmfluss. Bedacht eingesetzt, kann man mit den Sprungbefehlen von **sed** ziemlich komplexe Programme schreiben. Unbedacht eingesetzt, kann man unnötigerweise noch viel komplexere Programme schreiben.

Kapitel 7. Vermischtes

Dateien

Mit **sed** kann man auch Dateien lesen und schreiben. Das geht mit den den Kommandos '`r` *`filename`*' und '`w` *`filename`*'. Beim Lesen wird die Datei nach dem gegenwärtigen Zyklus ausgegeben, oder wenn eine neue Zeile gelesen wird. Eine nicht vorhandene Datei wird als existent aber leer angesehen. Der '`w`' legt eine neue Datei an oder *überschreibt* eine schon vorhandene Datei und füllt sie mit dem Inhalt des *pattern space*. Das Kommando '`w`' kann auch als Flag zu '`s///`' angegeben werden, wobei in die Datei geschrieben wurde, wenn eine Substitution erfolgen konnte.

Das folgende **sed**-Script ist ein Ersatz für den UNIX-Befehl '**tee** *dateiname*',

```
sed -e 'w dateiname'
```

und wenn man es in der Form '`sed wdateiname`' schreibt, nur um ein 'w' länger als die Version mit **tee**

Um den Einsatz des Kommandos '`r`' zu demonstrieren möchte ich meinen *Tante Amalien*-Emulator vorstellen (der von Joseph Weizenbaums genialem ELIZA inspiriert ist). Meine Tante Amalie hat 3 Standardsätze die sie der Reihe nach verwendet. Diese sind 'Meinst du?', 'Früher war es besser - entschieden besser!' und 'Davon verstehst du nichts.'. Diese in die 3 Dateien `stdsatz1` - `stdsatz3` geschrieben ergeben mit dem (unportablen!) Script

```
#!/bin/sed -nf
1~3r stdsatz1
2~3r stdsatz2
3~3r stdsatz3
```

das interessante Gespräch

```
Schönes Wetter heute
Meinst du?
Ja natürlich. Schau doch raus!
Früher war es besser - entschieden besser!
Ich weiß nicht was du hast - blauer Himmel und strahlender Sonnenschein.
Davon verstehst du nichts.
Wie du meinst, Amalie.
Meinst du?
```

usw. Ich könnte mich stundenlang mit Amalie unterhalten. Für komplexere Charaktere wäre es denkbar, den Input nach bestimmten Mustern zu durchsuchen und dementsprechend zu reagieren. Dabei muss ja nicht jede Antwort in einer Datei gespeichert sein, man könnte ja den *pattern space* mit '`s/^.*$/Antwort/p`' füllen.

Noch mehr Kommandos

sed kennt noch einige meines Erachtens recht exotische Befehle wie '`a`', '`i`' und '`c`', welche einen gegebenen Text ausgeben oder '`w`', welcher den *pattern space* in eine Datei schreibt und noch derer mehr. Sollten diese gebraucht werden, dann gibt die info-page bereitwillig Auskunft dazu.

sed oder nicht sed?

Carlos Jorge G.duarte emuliert in seinem Tutorial sehr viele UNIX-Befehle mit **sed**. Das mag als Beispiel sehr interessant sein (und ich empfehle jedem, diese Programme anzuschauen und zu verstehen), ist aber in der Praxis zu kompliziert. **sed** eignet sich durch die kompakte Schreibweise seiner Scripte besonders gut für Einzeiler - wird das Problem größer, kann das Debuggen eines **sed**-Scriptes sehr nervenaufreibend sein. Eine sehr gute Alternative ist da **awk**, das eine ähnliche Syntax (`/regex/`

`{action}`) unterstützt, darüber hinaus noch strukturierte Programme (mit Konstrukten wie `'if (expr) statement else statement'` oder `'while (expr) statement'` u.Ä.) erlaubt, Funktionen zur Mustersuche, eigene Typen zur Behandlung von Gleitkommazahlen und vieles mehr besitzt. Leider kennt es die sehr bequemen `'\1'` Referenzen nicht, die **sed** bietet. **awk**-Programme sind in der Regel 3-10 mal so groß wie **sed**-Scripte.

Python oder **perl** bemüht man für größere und komplexere Probleme. Damit kann man ausgewachsene Programme schreiben, die zur Laufzeit interpretiert werden, wodurch sie etwas langsamer als **sed**- oder **awk**-Scripte abgearbeitet werden. Die Scripte können ungefähr die 8-40-fache Größe eines äquivalenten **sed**-Scriptes haben. Das muss kein Nachteil sein, denn die verlorene Zeit die ein mittelmäßiges **Python**-Script zur Laufzeit gegenüber einem **sed**-Script verliert, ist meistens durch eine lange Fehlersuche während des Schreibens eines solchen mehr als wett gemacht.

Keines dieser Tools soll verwendet werden, wenn das Betriebssystem ein dediziertes Programm für das jeweilige Problem bereitstellt, da dieses meist schneller und sicherer arbeitet und obendrein noch einfacher zu bedienen ist. UNIX bietet eine Vielzahl solcher Helferchen, die aber meistens nur einseitig oder überhaupt nicht verwendet werden. Ein Blick in die man-pages zu **bash**/**tcsh**, **xargs**, **tr**, **test**/**[**, **cat**/**tac**, **cut**, **[ef]grep**, **uniq**, **sort**, **wc**, **tail**, **column**/**colrm**, **paste**, **look**, **hexdump**, **basename** und Kumpanen, **seq**, **luser**, **lart**, **whack**, **bosskill**, [...] kann dem geneigten Leser nur zum Nutzen gereichen.

Andere Programme mit sed-Kommandos

Das am meisten verbreitete Programm, in dem man **sed**-Anweisungen angeben kann, ist sicher **vi**. Um beispielsweise in einer Zeile "bla" nach "blupp" zu ändern, schreibt man

```
:s/bla/blubb/g
```

Ein interessantes Flag zum `s///`-Kommando ist `c`, welches bewirkt, dass man bei jeder potentiellen Substitution um Bestätigung gefragt wird. Ein Blick in die Dokumentation von **vi** (**vim**) zu weiteren **sed**-Kommandos lohnt sich.

ed ist ein vollständiger Editor, und älter als **sed**. **ed** ist die richtige Wahl, wenn man Texte inline-editieren will. Zu beachten ist, dass, im Gegensatz zu **sed**, die Datei *nicht* tumb einmal von oben nach unten durchgeackert und eine Anweisung damit auf alle Zeilen angewandt wird. Das Kommando

```
s/bla/blupp/g
```

wird in **ed** nur auf die aktuelle Zeile angewandt. Der gewünschte Effekt wird mit

```
%s/bla/blubb/g
```

erreicht. Ein anderes neues Kommando is `g/re/command`. Es wendet das Kommando auf alle jene Zeilen an, auf welche der Reguläre Ausdruck 're' passt. Ein Beispiel dafür ist `g/re/p`, welche alle Zeilen ausgibt, welche auf die RE passen.

Anmerkung

Eine geschichtliche Randnotiz: das UNIX-Kommando **grep** wurde bequemlichkeitshalber aus **ed** extrahiert, und der Name deutet an, was das Programm tut: global, Regular Expression, print.

Kapitel 8. Ein paar Beispiele

Diese Sektion sollte mehr Beispiele enthalten. Ich bin ständig auf der Suche nach Beispielen, welche zum Verständnis von **sed** beitragen. Sollte der Leser Scripte wissen, die mit noch nicht vorgestellten Tricks arbeiten, welche eines Kommentars bedürfen, einem ganze Mannjahre an Handarbeit ersparen, einfach nur schön sind oder irgend einen anderen AHA!-Effekt auszulösen imstande sind, dann bitte ich darum, mir diese zu schicken. Sie werden mit Angabe des Autors hier veröffentlicht.

In diesem Kapitel werden die GNU-Erweiterungen scham- und vor allem kommentarlos verwendet, da sie die Lesbarkeit eines Scriptes sehr verbessern. Die Beispiele ließen sich auch ohne diese Erweiterungen beschreiben (und es wird empfohlen das auch zu tun, sobald ein Script auf andere Systeme übertragen werden könnte) das ginge aber auf Kosten der Verständlichkeit.

Entfernen von Kommentaren

Die fiktiven Programmiersprachen K und K++ kennen zwei Arten von Kommentaren. Da wäre die nur in K++ verwendete Art '`kk.*`' (zwei einleitende 'k'), oder die in beiden Sprachen verwendete Form '`ko.*ok`', wobei sich ein solcher Kommentar über mehrere Zeilen erstrecken kann. Es soll ein **sed**-Script erstellt werden, das solche Kommentare (warum auch immer!) entfernen soll.

Anmerkung

Soll das folgende Script für in den archaischen Sprachen C/C++ geschriebene Programme funktionieren, dann muss man es mit dem **sed**-(Pseudo-)Einzeiler '`sed -e '/^#/!{s/k/\//g;s/o/\*/g;}' k-kommentar > c-kommentar`' ummodeln.

Das Entfernen von K-Kommentaren benötigt ein paar Erklärungen. Nehmen wir einmal an, wir hätten den gesamten Kommentar im *pattern space*. Das Kommando '`s/ko.*ok//`' geht aus dem Grund nicht, weil die ansonsten nützliche Eigenschaft von REs, den längsten zutreffenden String zu nehmen, hier unerwünscht ist. Sind zwei vollständige Kommentare in einer Zeile vorhanden, dann würde auch der unschuldigerweise *dazwischen* stehende Code entfernt werden.

Der zweite Anlauf ist ein '`s/ko\([^o][^k]\)*ok//g`'. Achtung bei Konstrukten, welche Quantifikatoren ('*', '\+', '\{\}' ...) auf zwei oder mehrere Zeichen anwenden! Das Script arbeitet nur bei der Hälfte der Kommentare, und zwar bei jener Hälfte welche eine gerade Anzahl von Zeichen beinhaltet. Vom Zorn gepackt, schreibt man dann Sachen wie '`s/ko\([^o]*\(o[^k]\)*[^o]*\)*ok//g`' welche zwar korrekt sind, aber völlig Praxisuntauglich. Ein solches Monsterprogramm kann allerhöchstens auf einem Großrechner vernünftig arbeiten. In diesen Situationen hilft es, eine verbale Beschreibung des Musters zu finden. Die könnte so aussehen: "Ein K-Kommentar beginnt mit 'ko', ihm folgen null oder mehr der oben beschriebenen Lesevorgängen `[^o]\|o\+[^ok]` plus einem Abschluss `o\+k`'. Auf **sed**isch übersetzt bekommt man '`ko\([^o]\|o\+[^ok]\)*o\+k`'.

Nun muss nur noch sicher gestellt werden, dass nach einem 'ko'-Muster auch ein 'ok' im *pattern space* ist. Ist dem nicht so, dann sorgt der innere Loop (um das label `append`) dafür, dass ständig neue Zeilen mit dem '`N`'-Befehl an den *pattern space* angehängt werden. Ein äußerer Loop (um das label `test`) sorgt dafür dass jene Zeilen richtig behandelt werden, in denen ein Kommentar geschlossen und anschließend ein neues mehrzeiligen Kommentar wieder aufgemacht wird.

```
#!/bin/sed -f

#lösche K++-Kommentare
/^[[:blank:]]*kk.*/d
s/kk.*//g

#Wenn kein Kommentar gefunden wurde, dann nächster Zyklus.
: test
/ko/!b

#Hänge so lange neue Zeilen an den pattern space an,
```

```
#bis ein vollständiger Kommentar zusammengebracht wurde.
: append
/ok/!{N;b append;}

#lösche K-Kommentare die sich vollständig im pattern space befinden
s/ko\([^o]\|o\+[^ok]\)*o\+k//g

t test
```

Ein K-Kommentar beginnt mit 'ko', soweit ist alles klar. Anschließend folgt die längst mögliche Zeichenkette, die kein 'ok' enthält. Hier liegt der Hund begraben. Das abschließende 'ok' ist wieder trivial.

Nun zum Hund: Gesucht ist die längst mögliche Zeichenkette, auf welche der reguläre Ausdruck /ok/ *nicht* zutrifft. Es ist also gewissermaßen das Gegenteil von /ok/ gesucht.

Man kann sich nun in Anlehnung an ein prozedurales Vorgehen vorstellen, man suche das erste Auftreten von 'ok' innerhalb einer Zeichenkette. Dazu lese man immer wieder neue Zeichen von der Zeichenkette ein und untersuche die eingelesenen Zeichen.

Die gesuchte Teil-Zeichenkette besteht dann aus null oder mehreren "Lesevorgängen": *Längste Zeichenkette ohne 'ok' = \(Lesevorgang\)**

Nach jedem Lesevorgang trifft man dann eine Fallunterscheidung, etwa von der Art: Es wurde kein 'o' eingelesen, es wurde ein 'o' aber kein 'k' eingelesen, etc. Man erhält so:

```
Lesevorgang = Fall_1 \| Fall_2 \| ... \| Fall_x
```

Wieder in Anlehnung an das prozedurale Vorgehen wird man zu Beginn der Überlegungen davon ausgehen, es werde pro Lesevorgang nur ein einzelnes Zeichen eingelesen. Ist dieses Zeichen dann von 'o' verschieden, trifft also der Ausdruck [^o] darauf zu, kann kann man mit dem nächsten Lesevorgang fortfahren, und man hat:

```
Fall_1 = [^o]
```

Ist das eingelesene Zeichen hingegen gleich 'o', dann könnte man in die Versuchung kommen zu prüfen, ob sich das nächste Zeichen von 'k' unterscheidet, in der Annahme, damit einen weiteren Fall eines Lesevorganges vollständig abgehandelt zu haben: Den Fall o[^k] nämlich! Träfe dieser reguläre Ausdruck auf die immerhin bereits zwei eingelesenen Zeichen zu, dann ginge man zum nächsten Lesevorgang über.

Aber hoppla! Das vorhin auf [^k] überprüfte Zeichen könnte ja wieder gleich 'o' sein, was zur Folge hätte, dass man beim nächsten Lesevorgang das zuerst eingelesene Zeichen auf auf 'k' überprüfen müsste. Solche Abhängigkeiten zwischen den Lesevorgängen sprengen aber das Konzept dieser Vorgehensweise und deuten darauf hin, dass der vorhergehende Lesevorgang im Prinzip weiter geführt werden muss.

Ist das erste Zeichen eines Lesevorganges also ein 'o', dann könnte diesem 'o' gleich eine ganze Folge weiterer 'o's folgen. Man muss also einen Ausdruck der Form o\+ einlesen, und zwar solange, bis man endlich ein Zeichen findet, das sich von 'o' unterscheidet. Ist dann dieses Zeichen nicht nur von 'o', sondern auch von 'k' verschieden, dann hat man insgesamt einen Ausdruck der Form o\+[^ok] eingelesen:

```
Fall_2 = o\+[^ok]
```

Von da aus kann man nun problemlos mit dem nächsten Lesevorgang fortfahren. Da das erste Zeichen aber nur entweder 'o' oder dann eben nicht 'o' sein kein, treten neben Fall_1 und Fall_2 keine weiteren Fälle mehr hinzu:

```
Lesevorgang = Fall_1 \| Fall_2 = [^o]\|o\+[^ok]
```

Bei jedem Lesevorgang findet man also ein einzelnes Zeichen [^o] oder einen Ausdruck o\+[^ok]. Erst wenn eine 'o-Folge" mit dem Zeichen 'k' endet, wenn man also auf den "Abschluss" o\+k trifft, ist man am Ende.

Die null oder mehr Lesevorgänge `[^o]\|o\+[^ok]` liefern damit die längste Zeichenkette, die den Abschluss `o\+k` nicht enthalten. Obwohl damit das anfängliche Ziel, die längste Zeichenkette ohne '`ok`' zu finden, knapp verfehlt worden ist, kann man mit diesen Überlegungen bequem den letztlich gesuchten Ausdruck eines K-Kommentars hinschreiben: *Ein K-Kommentar beginnt mit '`ko`', ihm folgen null oder mehr der oben beschriebenen Lesevorgängen `[^o]\|o\+[^ok]` plus einem Abschluss `o\+k`*:

```
K-Kommentar = ko\([^o]\|o\+[^ok]\)*o\+k
```

Übrigens kann man mit dem bekannten Editor vim einen K-Kommentar einfach durch

```
K-Kommentar = ko.\{-}ok
```

definieren. Dabei bedeutet der Ausdruck `.\{-}`, dass, ähnlich wie bei `.*`, eine beliebige Zeichenkette gesucht ist, aber nicht längste, sondern die kürzeste.

Vielen herzlichen Dank an Mathias Michaelis für dessen Beitrag zu diesem Tipp.

elleff-Rücktransformation

Gelogen habe ich nicht, als ich behauptete, mit REs könne man keinen *elleff*-verschlüsselten Vokal beschreiben - das stimmt schon. Aber **sed** kann. Und das auf eine sehr trickreiche Weise. Zuerst das Script, kommentiert wird danach.

```
sed -e 's/\([aeiou]\+\)l\1f\1/\1/g'
```

Wenn man diesen Kniff nicht schon einmal gesehen hat, muss man 2 (ich 3) mal hinschauen um zu verstehen, warum das funktioniert. Was mir an diesem Beispiel so gut gefällt ist, dass sobald die Klammer geschlossen wird, der Inhalt des eingeschlossenen Bereiches schon in '`\1`' bereit steht und somit verwendet werden kann - auch innerhalb der RE. Die RE wird somit zur Laufzeit verändert. Das zeigt einerseits wie leistungsfähig **sed** ist und andererseits dass es auch manchmal recht knifflig sein kann die Scripte anderer zu verstehen.

Diesen Trick verdanke ich Carlos Duarte - ein weiterer Anreiz, in sein sed tutorial [http://sed.sf.net/grabbag/tutorials/do_it_with_sed.txt] hineinzuschauen.

Verschachtelte Klammern

In manchen Fällen muss man auf das *n*-te Feld einer Zeile zugreifen. Die Quantifikatoren '`\{n\}`' sind dabei sehr nützlich. Will man beispielsweise das 3. Wort einer Zeile an den Zeilenanfang setzen, ist das Konstrukt

```
sed -e 's/^\([^ ]* *\)\{2\}\([^ ]* \)/\2\1/g'
```

nicht richtig, da die Referenz '`\1`' nur das zweite Wort enthält, nicht aber das erste und zweite. Abhilfe schafft da ein weiteres Klammernpaar, wie im folgenden Script:

```
sed -e 's/^\(\([^ ]* *\)\{2\}\)\([^ ]* \)/\3\1/g'
```

Dabei ist '`\3`' die Referenz auf das zweite Wort ('`\2`' referenziert das letzte Wort in '`\1`'). Hierbei ist man schon an die Grenzen der Verständlichkeit eines **sed**-Scriptes gegangen, und es ist zu überlegen, ob man mit anderen Programmen wie zum Beispiel **awk** nicht besser bedient ist.

Danke an Tillmann Bitterberg für diesen Tip.

Kapitel 9. Kurzreferenz

Diese Kurzreferenz kann und will nicht die man- oder info-page zu **sed** ersetzen, sondern ist nur als eine Gedächtnisstütze gedacht.

Adressen

sed kann mehrere Dateien abarbeiten, wenn man diese als Argumente übergibt. Diese Dateien werden als ein einziger Input-Stream behandelt. Achtung deshalb bei Zeilenangaben. Die Adresse '1' gibt deshalb nicht die erste Zeile in jeder Datei an, sondern die erste Zeile im Input-Stream; der nachfolgende Befehl wird also nur einmal ausgeführt.

Tabelle 9.1. Adressen

Adresse	Beschreibung
n	Selektiert die Zeile *n* im Eingabestream.
$	Letzte Zeile im Input-Stream.
/regex/	Alle Zeilen, auf die *regex* passt. Alternativ kann auch '\ %*regex*%' geschrieben werden, wobei '%' ein beliebiges Zeichen ist.
adresse1,*adresse2*	Adressbereich: Alle Zeilen zwischen *adresse1* und *adresse2*, einschließlich der beiden Adressen.
adresse!	Alle Zeilen ausschließlich der in *adresse* angegebenen Zeilen.

Kommandos

Tabelle 9.2. Allgemeine Kommandos

Kommando	Anzahl Adressen	Beschreibung
#	0	Kommentar, alle nachfolgenden Zeichen bis zum Newline werden nicht als Programmcode interpretiert.
{	0-2	Beginnt einen Block, der mehrere Kommandos beinhalten kann. Muss mit } abgeschlossen werden. Auf jedes Kommando innerhalb des Blocks muss ein *semicolon* ';' folgen.
=	0-1	Gibt die aktuelle Position im Input-Stream aus.
q	0-1	Beendet das Programm. Der *pattern space* wird nur dann geschrieben, wenn die Option -n nicht gesetzt wurde.
l	0-2	Gibt den *pattern space* in einer unmissverständlichen Form aus, die an die ANSI C-Screibweise angelehnt is. Sehr nützlich zur Fehlersuche.
d	0-2	Löscht den *pattern space* und startet sofort einen neuen Zyklus.
p	0-2	Schreibt den *pattern space* nach stdout. (wird normalerweise nur in Verbindung mit der Option -n verwendet.)
n	0-2	Schreibt den *pattern space* (wenn -n nicht gesetzt ist) und ersetze den *pattern space* mit der nächsten Zeile. Wenn keine Zeile mehr zu lesen ist, beende das Programm.
s/*regex*/*rpl*/*flg*	0-2	Ersetzt *regex* durch *rpl*. Null oder mehrere *flg* geben an, wie das geschehen soll: 'g' ersetzt alle Zeichenketten in

Kommando	Anzahl Adressen	Beschreibung
		einer Zeile, auf die *regex* passt, 'p' führt eine Substitution durch und schreibt den *pattern space*, '*nummer*' ersetzt nur die *nummer*-te Fundstelle.
y/*src*/*rpl*/*flg*	0-2	Ersetzt jedes Zeichen im *pattern space*, das in *src* vorkommt, mit dem entsprechenden Zeichen in *rpl*.

Tabelle 9.3. Sprung-Kommandos

Kommando	Anzahl Adressen	Beschreibung
: *label*	0	Definiert das Sprungziel *label*. Siehe Kommandos b oder t, wie man Labels anspringt.
b *label*	0-2	Branch; unbedingter Sprung.
t *label*	0-2	Bedingter Sprung. Es wird zum *label* gesprungen, wenn auf den aktuellen *pattern space* eine s///- oder y///-Substitution erfolgt wurde.

Tabelle 9.4. Kommandos im Zusammenhang mit dem Hold-buffer

Kommando	Anzahl Adressen	Beschreibung
D	0-2	Löscht den Text im *pattern space* bis zum ersten newline. Ist noch Text im *pattern space* enthalten, starte einen Zyklus mit diesem Text, ansonsten starte einen normalen Zyklus.
N	0-2	Hängt eine newline an den *pattern space* an, gefolgt von der nächsten Zeile des Inputs. Ist das Ende der Datei erreicht, wird das Programm abgebrochen, ohne weitere Befehle abzuarbeiten.
P	0-2	Gibt den *pattern space* bis zum ersten newline aus.
h	0-2	Ersetzt den Inhalt des *hold space* mit dem des *pattern space*.
H	0-2	Hängt ein newline an den *hold space*, gefolgt vom Inhalt des *pattern space* an.
g	0-2	Ersetzt den Inhalt des *pattern space* mit dem des *hold space*.
G	0-2	Hängt ein newline an den *pattern space*, gefolgt vom Inhalt des *hold space* an.
x	0-2	Tauscht den Inhalt von *pattern space* und *hold space* aus.

Daneben gibt es noch weitere Befehle wie a, i, c, r, w. Der Leser sei diesbezüglich mit einem freundlichen RTFM auf die man-page verwiesen.

Kapitel 10. Versionsgeschichte

Versionsgeschichte

Version 1.14 2016-11-10 thp

Neue Lizenz: Creative Commons Attribution-ShareAlike 4.0 International.

Version 1.13 2016-05-11 thp

Neue e-Mail Adresse.

Version 1.12 2014-03-12 thp

Neuer Link zu Mike Arsts "u-sedit".

Version 1.11 2014-01-02 thp

Bessere Beschreibung in "was ist sed".

Version 1.10 2013-03-21 thp

Beschreibung des Kommandos 'l'. Das Tutorium ist nun auch auf GitHub.

Version 1.9 2013-03-03 thp

Update auf Docbook 5, die Quelldatei ist nun UTF-8 kodiert.

Version 1.8 2012-06-25 thp

Kleine Korrekturen. Updated Links, Link auf Feedback-Seite.

Version 1.7 2010-06-12 thp

Eine kleine Anmerkum zum Shakespeare-Zitat (vielen Dank an Daniel). Ein paar Querverweise eingefügt.

Version 1.6 2009-04-13 thp

Verwendung des Docbook Stils "book" anstelle von "article". Erweiterung des Indexes und bessere Beschreibung der Beispiele. Vielen Dank an "Max" (Rem Remedy).

Version 1.5 2008-11-16 thp

Index eingefügt.

Version 1.4 2008-08-11 thp

Einige Rechtschreibkorrekturen. Vielen Dank an Constantin Hagemeier. Umstellung von db2pdf auf fop.

Version 1.3 2008-04-13 thp

Viele Rechtschreibkorrekturen. Vielen Dank an Kate (KDE Advanced Text Editor).

Version 1.2 2008-03-03 thp

Neue Lizenz: Creative Commons Attribution-Share Alike 3.0 Unported.

Version 1.1 2007-07-19 thp

Kleinere Verbesserungen und Rechtschreibkorrekturen. Vielen Dank an Alexander Kriegisch.

Version 1.0 2007-02-20 thp

Kleinere Verbesserungen und Rechtschreibkorrekturen.

Version 0.9 2005-05-03 thp

Verbesserungen von Mathias Michaelis zu den K-Kommentaren.

Version 0.8 2004-03-17 thp

Umstieg auf xml.

Version 0.7 2003-01-09 thp

Bessere Unterscheidung Basic/Erweiterte Reguläre Ausdrücke; erster Versuch, GNU-ismen aus den Scripten zu verbannen.

Version 0.6 2002-11-10 thp

Beispiel für den x-Befehl in der Space-ball Sektion eingefügt. Danke an Ulf Bro.

Version 0.5 2002-09-06 thp

History als Kapitel angelegt, wie es die FDL verlangt; Detailänderungen.

Version 0.4 2002-03-02 thp

Detailverbesserungen und Kurzreferenz.

Version 0.3 2001-09-11 thp

Gründlich überarbeitet und neu strukturiert.

Version 0.2 2001-01-09 thp

wurde nie freigegeben.

Version 0.1 2001-09-07 thp

Beginn der Arbeit am Tutorium.

Index

Symbole

A

B

D

E

F

G

H

K

L

M

N

O

P

R

S

U

V