Kapitel 2

sed

2.1 Einführung

In diesem Kapitel werden wir `sed`, den *Stream EDitor*, kennenlernen. `sed` ist ein Editor, aber anders als z.B. `vi` oder `emacs`, ist er nicht interaktiv, sondern läuft im Batch-Betrieb. D.h. man gibt `sed` ein Script mit, das er dann auf die Eingabedaten anwendet und das Ergebnis wieder ausgibt, siehe Abbildung 2.1.
`sed` arbeitet zeilenorientiert: Er lädt immer eine Zeile in seinen Puffer, wendet die Kommandos aus dem Script darauf an und gibt das Resultat dann am Schluß wieder aus[1].
Man kann `sed` das Script auf zwei Arten mitgeben:

- Man gibt nach der Option **-e** das Script auf der Kommandozeile an: **sed -e *script***.
- Man schreibt das Script in ein File und gibt den Namen des Files nach der **-f** Option an: **sed -f *file***.

Z.B. hat **sed -e 's/Uni Saarbruecken/Uni des Saarlandes/g' Antrag** zur Folge das alle Vorkommen von `Uni Saarbruecken` in dem File `Antrag` durch `Uni des Saarlandes` ersetzt werden. Das Kommando ist in diesem Fall das *Substitute* Kommando (erkenntlich an dem `s` vor dem ersten Schrägstrich)[2]. Dieses ersetzt Vorkommen des Pattern zwischen den ersten beiden Schrägstrichen durch den Text zwischen dem zweiten und dritten Schrägstrich. Das `g` Flag am Ende signalisiert dem Kommando, daß es alle Vorkommen in einer Zeile ersetzen soll, nicht nur das erste, was den Default darstellt.

[1]Nun ja, durch das Konzept des "Hold-Spaces" und Multi-Line Pattern ist es nicht ganz so einfach, siehe Abschnitt 2.3.3.

[2]Die Hochkommata "'" dienen nur zum Quoten der Scriptes in der Shell, da dieses Leerzeichen enthält.

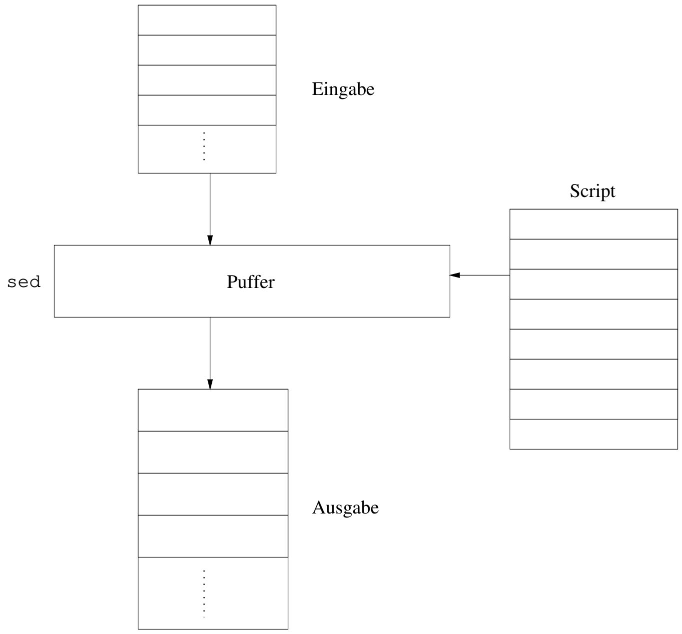

Abbildung 2.1: `sed`'s Arbeitsweise

Schreibt man das folgende Script in ein File `scr.sed`:

```
s/Uni Saarbruecken/Uni des Saarlandes/g
/XXX/d
```

und ruft dann z.B. **sed -f scr.sed Antrag** auf, dann werden wie oben alle Vorkommen von `Uni Saarbruecken` ersezt; danach wird jetzt aber auch noch jede Zeile gelöscht, in der der Text `XXX` vorkommt[3]. Das Löschen wird durch das zweite Kommando in dem Script ausgeführt, dem *Delete* Kommando, erkenntlich an dem `d`. Dem Kommando ist eine *Adresse* vorgestellt, die angibt, bei welchen Zeilen es ausgeführt werden soll. Adressen werden wir weiter unten noch besprechen.

Normalerweise nimmt `sed` seine Eingabedaten von der Standardeingabe, man kann aber auch die Namen von Files auf der Kommandozeile angeben, so daß deren Inhalt nacheinander bearbeitet wird: **sed -e *script file*$_1$ *file*$_2$ ...**.

Wie oben gesagt, gibt `sed` nach der Abarbeitung des Scriptes die Zeile in seinem Puffer wieder aus. Man kann dieses Verhalten abschalten, indem man die **-n** Option auf der Kommandozeile angibt. Dann muß man allerdings den Kommandos

[3] Eine beliebte Methode, Stellen in Files zu kennzeichnen, an denen noch etwas bearbeitet werden muß, ist es, sie mit einem Kommentar der Form `XXX` *`Kommentar`* zu versehen.

im Script explizit sagen, daß sie etwas ausgeben sollen. Wir werden weiter unten sehen, welche Optionen man diesen Kommandos dazu geben muß.
`sed` gibt seinen Buffer immer auf der Standardausgabe aus[4]. `sed` verändert niemals Files, die man als Eingabe auf der Kommandozeil angegeben hat[5].
`sed` bietet die Möglichkeit, Kommandos im Script nur dann anzuwenden, wenn der Puffer auf ein bestimmtes *Pattern* paßt, respektive die zugehörige Zeile in einem bestimmten *Adreßbereich* liegt (etwa bestimmte Zeilennummern). Sowohl Pattern als auch Adressen werden wir weiter unten genauer besprechen.
Es gibt eine ganze Reihe von Versionen von `sed`. Wir werden im folgenden ausschließlich die GNU Version von `sed` betrachten.

2.2 Pattern Matching

Pattern begegnen einem in der UNIX Welt sehr häufig. `sed` benutzt die meisten der Standard-Pattern die auch die Shells anbieten, und noch einige weitere.
Wie üblich werden dazu bestimmte Zeichen zu *Metazeichen*, die dann für Wiederholungen, etc. stehen.
In `sed` werden Pattern an zwei Stellen verwandt:

- In Adressen. Adressen dienen dazu, einige Kommandos im Script nur auf bestimmte Zeilen der Eingabe anzuwenden.
- Im *Substitute* Kommando von `sed`. Dieses dient dazu, bestimmte Teile einer Eingabezeile durch andere Teile zu ersetzen. Siehe Abschnitt 2.3.2

Das Matchen des Patterns erfolgt dabei anders, als bei der Shell. Es wird geprüft, ob das Pattern *irgendwo* im Puffer passt, nicht vom Anfang an (außer, man legt das Pattern mit den Metazeichen ˆ oder `$` explizit an den Anfang/ans Ende des Puffers, s.u.). So passt etwa das Pattern `Pattern` auf die Zeile `Das Matchen des Patterns erfolgt...`. Diese Semantik führt oft zu ungewünschten Matches. Man sollte also, wo nötig, das Pattern explizit mit ˆ und `$` auf den Anfang/das Ende zwingen.
Besonders zu beachten ist, das `sed` immer versucht von links her den längsten Teil einer Zeile zu matchen. D.h. das Pattern **a*a*** matcht in der Eingabe `aaaa` die ersten vier '`a`' auf den ersten Teil und das leere Wort auf den zweiten Teil des Patterns. Dies ist für das Pattern im Substitute Kommando wichtig, da man dort die Teile, die von einem bestimmten Teil des Patterns gematcht wurden, wieder im Ersetzungstext einsetzen kann.

[4]D.h. man muß ihn mit >***file*** umlenken, wenn man das Ergebnis in einem File haben will.
[5]Es sei denn, man tut etwas törichtes in der Art von **sed -e *script* foo** >**foo**. Dies führt zu lustigen Effekten, allerdings nie den gewünschten.

Im folgenden Abschnitt werden nun zuerst die einzelnen Metazeichen vorgestellt und erläutert. In Abschnitt 2.2.2 werden dann die Adressen vorgestellt.

2.2.1 Pattern

`sed` kennt die folgenden Meta-Zeichen:

. Der Punkt steht für ein beliebiges Zeichen, ausgenommen ein Newline (\n).

[...] Dies definiert eine Klasse von Zeichen. Meta-Zeichen zwischen diesen Klammern verlieren ihren Meta Status, werden also wieder zu dem Zeichen selbst (`[.]` matcht also nur den Punkt selbst.). Ein ^ als erstes Zeichen nach der öffnenden Klammer dreht den Sinn des Matchings um: die Klasse matcht dann nur die Zeichen, die *nicht* zwischen den Klammern auftreten. Ein − gibt *Bereiche* von Zeichen an, so matcht `[0-9]` die Ziffern 0 bis 9. Taucht das Zeichen `]` als erstes Zeichen nach der öffnenden Klammer (und ggf. einem `^`) auf, steht es für die eckige rechte Klammer selbst, also matcht `[^]0-9]` alles ausser den Ziffern 0 bis 9 oder der rechten eckigen Klammer `]`.

^ Steht dieses Zeichen am Anfang des Patterns, paßt es auf den Anfang des Puffers. Steht es später im Pattern, hat es keine spezielle Bedeutung mehr und passt also nur auf das Zeichen ^ selbst.

$ Steht dieses Zeichen am Ende des Patterns, passt es auf das Ende ders Puffers es woanders, verliert es seine spezielle Bedeutung und passt also nur noch auf das Zeichen `$` selbst.

***** Der Stern markiert beliebig oftes Auftreten des Zeichens, das durch das vor im stehenden Pattern beschrieben wird. So passt `0*` auf `0`, `00`, etc. aber auch das leere Wort. Ebenso passt `[0-9][0-9]*` auf eine nicht leere Folge von Ziffern.

\\{n,m**\\}** Matcht n bis m maliges Auftreten des Zeichens, das im vorstehenden Pattern beschrieben wird. Als Spezialform passt \{n\} auf *genau* n Auftreten und \{n,\} auf *wenigstens* n Auftreten.

c Jedes andere Zeichen c matcht genau dieses Zeichen.

Besonders zu beachten ist, daß der `*` und der \{n,m\} Operator immer nur ein vorhergehendes *Zeichen* (welches auch durch eine Klasse angegeben werden kann) wiederholen, aber keine anderen vorhergehenden Pattern. So ist `a\{3\}*` ein illegales Muster und passt *nicht* auf beliebig ofte Wiederholungen von drei `a`'s. Dahingegen ist `[a-zA-Z]*` legal und passt auf beliebig viele Buchstaben.

2.2.2 Adressen

Adressen werden in `sed` benutzt, um bestimmte Script-Kommandos nur auf bestimmte Zeilenbereiche anzuwenden. Eine Adresse kann dabei sein:

- Ein Bereich, der durch eine Anfangszeile und eine Endzeile gegeben ist. Dieser hat die Form `Anfang,Ende`. Dabei können *Anfang* und *Ende* sein:
  - Eine Zeilennummer[6], wobei das Symbol `$` für die letzte Zeile steht (deren Nummer i.A. nicht im voraus bekannt ist).
  - Ein Pattern, eingeschlossen in '`/`': `/pattern/`.

  Der Bereich einer solchen Adressangabe wird durch folgende Schritte bestimmt.

  1. Es wird verglichen, ob die aktuelle Zeile auf den Anfang passt. Wenn ja, wird das Aktiv-Flag gesetzt.
  2. Ist das Aktiv-Flag gesetzt, wird die Zeile in den Puffer übernommen und von den folgenden Kommandos im Script bearbeitet.
  3. Es wird geprüft, ob die Zeile auf das Ende passt. Wenn ja, wird das Aktiv-Flag gelöscht.

- Eine einzelne Spezifikation, die wieder eine Zeilennummer oder eine Spezifikation der Form `/pattern/` sein kann. Hier wird jede Zeile mit dieser Adresse verglichen und dann durchgelassen, wenn die Zeilennummer stimmt, oder das Pattern in der Zeile passt (je nachdem, welche Adressart man angegeben hat.).

Wenn einer Adresse ein Ausrufungszeichen folgt, wird die Reichweite der Adresse negiert, die folgenden Kommandos also nur auf Zeilen angewandt, die nicht in den Adressbereich fallen.
Fehlt eine Adresse vor einem Kommando, wird dieses Kommando auf alle Zeilen angewandt, die bis zu ihm durchgelassen wurden.
Einige Beispiele für Adressen und ihre Reichweiten:

- `1,$` Reicht von der ersten bis zur letzten Zeile.
- `1,/^$/` Reicht von der ersten Zeile bis zur *ersten* Leerzeile[7].
- `/^%%%SOE/,/^%%%EOE/` Umfasst alle Blöcke im File, die durch ein `%%%SOE`, `%%%EOE` Paar geklammert sind.
- `1,10!` Umfasst alle Zeilen, *ausser* den ersten zehn.

[6] `sed` numeriert die Eingabeeilen von 1 an durch.
[7] Nützlich, um mittels `1,/^$/d` den Header einer Mail zu löschen.

2.3 Kommandos

Ein Kommando kann eine der folgenden Formen haben:

Kommando[*Argumente*]
adrKommando[*Argumente*]

Wobei *adr* eine Adresse in den oben dargelegten Formen ist, *Kommando* ein Buchstabe, welcher das Kommando beschreibt und *Argumente* die Argumente zu dem Kommando. Wir werden bei der Beschreibung der einzelnen Kommandos angeben ob und welche Argumente es benutzt. Es sei angemerkt, daß sich nicht alle Kommandos mit allen Adressangaben vertragen. So gibt es einige, denen höchstens eine einfache Adresse (Zeilennummer oder Pattern), aber kein Bereich vorangestellt werden kann. Dies wird ebenfalls bei der Beschreibung des Kommandos erwähnt werden.
Enthält ein Script mehrere Kommandos, müssen diese durch ein Semikolon oder einen Zeilenumbruch getrennt werden. Z.B.:

```
s/FB14/Fachbereich 14/g ; s/s\./siehe/g
```

oder, wenn das Script in einem File steht:

```
s/FB14/Fachbereich 14/g
s/s\./siehe/g
```

2.3.1 Gruppierung

In `sed` werden die geschweiften Klammern benutzt, um Kommandos einzugruppieren. In solchen Gruppen, kann man wiederum Kommandos mit den geschweiften Klammern gruppieren.
Beispiel:

```
1,/^$/ { s/gargoyle.cs.uni-sb.de/cs.uni-sb.de/
         /[Ww]arning/d
       }
```

Die Möglichkeit der Gruppierung erlaubt es, auch die Kommandos, die nur eine einfache Adressangabe vertragen (s.o.) auf Adressbereiche anzuwenden, man schreibt einfach statt ***adr1,adr2Kommando*** die Gruppierung ***adr1,adr2***{ ***Kommando***}.

2.3.2 Basis-Kommandos

Wir listen die Kommandos in alphabetischer Reihenfolge auf.

Append

Das Append Kommando, repräsentiert durch den Kommando-Buchstaben `a` fügt Text nach der aktuellen Zeile ein. Dieser Text wird am Schluß des Scriptes ausgegeben, ist jedoch bis dahin nicht weiter verfügbar. D.h. der Text wird nicht an den Puffer angehängt und verändert diesen also auch nicht. Er wird nur nach dem Pufferinhalt ausgegeben, wenn dieser schließlich ausgegeben wird. Dieses Kommando kann nur mit einer einfachen Adresse benutzt werden und nimmt eine spezielle Form von Argumenten:

[*adr*]`a\`
Text
Das Kommando muß sofort von einem Rückstrich und dem Zeilenende gefolgt werden. Der auszugebende Text folgt dann mit Beginn der nächsten Zeile. Wenn der Text Newlines enthält, also über mehrere Zeilen geht, muß vor dem Ende jeder Zeile (ausser natürlich der letzten) ein Rückstrich stehen.

Change

Das Change-Kommando (`c`) ersetzt die Zeilen, die durch eine ggf. vorstehende Adresse (auch einen Adressbereich) durch den Text, der als Argument folgt. Dabei ist die Form dieses Textes so, wie beim Append Kommando:

[*adr*][`,` *adr*]`c\`
Text

Delete

Das Delete Kommando (`d`) löscht einfach die Zeilen, die zu im durchgelassen werden. Das heisst, der Puffer wird verworfen, die nächste Zeile wird gelesen und das Script wieder vom Anfang an abgearbeitet. Allerdings wird Text, der durch ein Append oder Insert Kommando eingefügt wurde, noch ausgegeben:

[*adr*][`,` *adr*]`d`

Insert

Das Insert Kommando (`i`) ist ein Analogon zum Append Kommando, nur wird der Text am Schluß nicht nach, sondern vor dem Puffer ausgegeben:

[*adr*]`i\`
Text

Line-Number

Das Line-Number Kommando (=) gibt einfach die Zeilennummer der aktuellen Zeile aus. Es kann nur eine einfache Adresse vorgestellt haben:

[*adr*]=

List

Das List Kommando (`l`) gibt den Puffer aus (ähnlich wie das Print Kommando), nur werden nicht-druckbare Zeichen (etwa ESC) als ASCII Codes ausgegeben, etwa erzeugt **sed -n -e 'l'** bei der Eingabe von **ESCa** die Ausgabe `\033a\n$`. Die Form des Kommandos ist:

[*adr*][`,` *adr*]`l`

Next

Das Next Kommando gibt den Puffer aus, liest die nächste Zeile ein und fährt mit der Bearbeitung des Scriptes beim nächsten Kommando, nicht am Scriptanfang, fort:

[*adr*][`,` *adr*]`n`

Print

Das Print Kommando (`p`) gibt den Puffer aus. Wenn `sed` nicht mit der **-n** Option aufgerufen wurde, kann der Pufferinhalt mehrmals ausgegeben werden (etwa am Ende des Scriptes):

[*adr*][`,` *adr*]`p`

Quit

Das Quit Kommando (`q`) macht genau das, was sein Name vermuten lässt: Der Puffer wird ausgegeben (wenn `sed` keine **-n** Option hatte) und `sed` wird beendet. Dieses Kommando kann nur eine einfache Adresse vorgestellt haben:

[*adr*]`q`

Read

Das Read Kommando (`r`) ähnelt dem Append Kommando, nur wird der Text, der angehängt werden soll aus einem File gelesen, dessen Namen (durch genau einen

Zwischenraum vom Kommando getrennt) als Argument folgt. Diesem Kommando kann nur eine einfach Adresse vorgestellt werden:

[*adr*]`r` *`file`*

Substitute

Das Substitute Kommando (`s`) ist das wohl am häufigsten benutzte Kommando in `sed`. Es dient dazu, bestimmte Teile des Puffers zu ersetzen. Welche Teile ersetzt werden sollen, wird im ersten Argument als Pattern spezifiziert, die Ersetzung im zweiten Argument:

[*adr*][`,`*adr*]`s/`*`pattern`*`/`*`replacement`*`/`[*`flags`*]

Im Pattern ist neben den allgemeinen Metazeichen noch Gruppierung erlaubt. D.h. man kann bestimmte Teile in `\(`, `\)` Paare einschliessen. Wenn das Pattern passt, wird der Teil zwischen diesen Paaren in speziellen Registern gespeichert. Im Ersetzungstext kann man dann mittels `\`n, wobei *n* eine Ziffer von 1 bis 9 ist, den Inhalt des zugehörigen Registers wieder einsetzen. Z.B. würde

s/^#define \([^][^]*\) \(.*\)$/\1 \2/

in einem C-Header File die Definiton eines Macros durch den Namen und den Wert des Macros ersetzen.
Als weiteres spezielles Zeichen wird '`&`' im Ersetzungstext durch den gesamten Match des Patterns ersetzt.
Das Kommando kennt die folgenden Flags:

n Ersetzt das n-te Auftreten des Patterns in der Zeile, nicht das erste.

g Ersetzt alle Auftreten des Patterns, nicht nur das erste.

p Gibt die Zeile aus, falls etwas in ihr ersetzt wurde. Wenn es mehrere Ersetzungen gab (etwa, weil das `g` Flag angegeben wurde), dann wird die Zeile mehrmals ausgegeben.

w $file$ Wenn eine Ersetzung gemacht wurdem wird die Zeile in das FIle $file$ geschrieben.

Man kann die beiden Argumente auch durch ein anderes Zeichen, als den Schrägstrich '`/`' trennen, z.B. **s+Uni Saarbruecken+Uni des Saarlandes+**. Dies ist besonders nützlich, wenn das Pattern oder der Ersetzungstext selbst einen Schrägstrich enthalten.
Einige weitere Beispiele dazu finden sich in Abschnitt 2.4.

Translate

Das Translate Kommando (`y`) dient dazu, Buchstaben umzusetzen, etwa Kleinbuchstaben in Großbuchstaben. Dazu wird das erste Zeichen aus dem ersten Argument durch das erste Zeichen aus dem zweiten Argument ersetzt:

[*adr*][`,`*adr*]`y/`*`Arg1`*`/`*`Arg2`*

Z.B. wird nach **sed -e 'y/abcdef/ABCDEF/'** jeder Kleinbuchstabe von 'a' bis 'f' durch den entsprechenden Großbuchstaben ersetzt[8].
Ebenso wie beim Substitute Kommando kann man statt des Schrägstriches ein anders Zeichen zur Trennung der beiden Argumente benutzen.

Write

Das Write Kommando (`w`) schreibt den Puffer (Aber *nicht* die mittels eines Append, Insert, etc. angefügten Texte) in das File, dessen Name (durch genau einen Zwischenraum getrennt) als Argument folgt:

[*adr*][`,`*adr*]`w` *`file`*

2.3.3 "Advanced Features"

In diesem Abschnitt wollen wir kurz drei etwas komplexere Bereiche von `sed` beleuchten:

- Sprünge im Script
- Den Hold-Space
- Multi-Line Pattern

Sprünge

Bisher wurde davon ausgegangen, daß `sed` die Kommandos des Scriptes sequentiell von oben nach unten abarbeitet. Manchmal ist es aber wünschenswert, zu anderen Teilen des Scriptes zu verzweigen und sei es nur, um nicht Teile duplizieren zu müssen. Hierfür gibt es in `sed` drei Kommandos:

- Das Label Kommando setzt Sprunkmarken in das Script.
- Das Branch Kommando verzweigt zu der durch ein Label gekennzeichneten Stelle im Script.

[8]Nützlich, um Hexadezimalzahlen zu normalisieren.

- Das Test Kommando prüft, ob im Puffer mittels des Substitute Kommandos eine Ersetzung stattgefunden hat und verzweigt in diesem Fall zu einer durch ein Label gekennzeichneten Stelle.

Label

Das Label Kommando (`:`) ist eigentlich kein Kommando, da es nichts tut. Es setzt nur eine Marke in das Script, die dann angesprungen werden kann:

`:`*`label`*

Der Name des Labels (*label*) darf bis zu sieben Buchstaben enthalten.

Branch

Das Branch Kommando (`b`) springt zu der Stelle des Scriptes an der das angegebene Label steht. Wenn man das Label fortlässt, wird zum Ende des Scriptes gesprungen:

[*adr*][`,`*adr*]`b` [*label*]

Test

Das Test Kommando (`t`) stellt einen bedingten Sprung dar. Als Bedingung wird geprüft, ob im Puffer mittels eines Substitute Kommandos bereits eine Ersetzung stattgefunden hat. Wenn ja, wird zu dem angegebenen Label, bzw. dem Ende des Scriptes gesprungen, wenn das Label fehlt:

[*adr*][`,`*adr*]`t` [*label*]

Der Hold-Space

Der Hold-Space stellt einen Zwischenspeicher dar, in welchen der Puffer mittels des Hold Kommandos kopiert werden kann. Das Get Kommando kopiert dann den Hold-Space wieder in den Puffer, während das Exchange Kommando die beiden vertauscht. Dies kann nützlich sein, um z.B. Zeilen in der Eingabe zu vertauschen. Angenommen, man hat eine Reihe von Files der Form

```
Titel: ...
Autor: ...
```

Und möchte in diesen nun den Autor vor dem Titel stehen haben. Folgendes `sed` Script leistet gerade dies:

```
/^Titel:/{
h;d
}
/^Autor:/{
p; g
}
```

Der erste Teil kopiert jede Zeile, die mit 'Titel:' beginnt in den Hold-Space und löscht sie dann, so daß sie nicht ausgegeben wird. Sobald die Zeile mit 'Autor:' am Anfang erreicht ist, wird diese ausgegeben und dann der Inhalt des Hold-Spaces (die Titel-Zeile) in den Puffer kopiert, der dann am Schluß des Scriptes ausgegeben wird.

Get

Das Get Kommando (`g`) kopiert den Inhalt des Hold-Spaces in den Puffer dessen voheriger Inhalt verlorengeht:

[*adr*][`,` *adr*]`g`

Hold

Das Hold Kommando (`h`) kopiert den Inhalt des Puffers, aber keine mittels der Append oder Insert Kommandos angefügten Texte, in den Hold-Space, dessen vorheriger Inhalt verlorengeht:

[*adr*][`,` *adr*]`h`

Exchange

Das Exchange Kommando (`x`) vertauscht den Inhalt des Hold-Spaces mit dem Puffer:

[*adr*][`,` *adr*]`x`

Multi-Line Pattern

Wir hatten anfangs gesagt, daß `sed` nur jeweils eine Zeile bearbeiten kann. Manchmal ist es aber nötig, Aktionen über mehrer Zeilen auszuführen, etwa wenn man eine Phrase aus mehreren Wörtern ersetzen will, da zwischen den einzelnen Wörtern ja ein Zeilenumbruch stehen kann[9]. Man kann `sed` mittels spezi-

[9]Wie in der Ersetzung `Uni Saarbruecken` → `Uni des Saarlandes`

eller Kommandos anweisen, die nächste Eingabezeile (oder den Inhalt des Hold-Spaces) an den Puffer anzuhängen. Die Zeilen werden dann durch ein Newline getrennt, welches in einem Substitute Kommando dann durch `\n` gematcht werden kann. Auch kann man den Puffer an den Inhalt des Hold-Spaces anhängen. Alle bisher vorgestellten Kommandos arbeiten immer auf dem gesamten Puffer[10], etwa das Delete Kommando. `sed` stellt nun einige Kommando Varianten bereit, die nur auf der ersten Zeile im Puffer arbeiten.
Will man also z.B. 'Uni Saarbruecken' durch 'Uni des Saarlandes' auch über Zeilenumbrüche hinweg ersetzen, kann man folgendes Script benutzen:

```
s/Uni Saarbruecken/Uni des Saarlandes/g
/Uni$/ {
 N
 s/Uni\nSaarbruecken/Uni\
des Saarlandes/
P;D
}
```

Dieses Script ersetzt zunächst alle Vorkommen von 'Uni Saarbruecken', die auf einer Zeile sind. In dem Fall, daß ein 'Uni' am Ende der Zeile steht, könnte eine mehrzeilige Ersetzung nötig sein, also wird die nächste Zeile mittels des Next-ML Kommandos (`N`) an den Puffer angehängt. Das folgende Substitute Kommando ersetzt nun alle durch ein Newline getrennten Vorkommen von 'Uni Saarbruecken'. Als Besonderheit beachte man, daß die Ersetzung geschieht, indem das 'Uni' auf der ersten Zeile verbleibt und der neue Text 'des Saarlandes' an den Anfang der zweiten Zeile gesetzt wird. Da man in Ersetzungstexten keine Angaben der Form `\n` machen kann, muß man im Script einen Zeilenumbruch an dieser Stelle machen, der wiederum durch einen Rückstrich eingeleitet werden muß. Das Print-ML Kommando (`P`) gibt nun die erste Zeile aus. Das folgende Delete-ML Kommando (`D`) löscht dann die erste Zeile im Puffer und macht mit der zweiten Zeile am Anfang des Scriptes weiter, damit auch Vorkommen in dieser Zeile ersetzt werden können.

Delete-ML

Das Delete-ML (für Multi-Line) Kommando (`D`) löscht den Teil des Puffers bis einschließlich des ersten Newlines. Wird dadurch alles gelöscht (d.h. es war nur eine Zeile im Puffer), dann wird die nächste Eingabezeile gelesen. Das Script wird in jedem Fall wieder von der ersten Position an abgearbeitet:

[*adr*][`,`*adr*]`D`

[10]Der jetzt mehrere Zeilen enthalten kann:-)

Get-ML

Das Get-ML Kommando (`G`) hängt ein Newline, gefolgt vom Inhalt des Hold-Spaces an den Puffer an. Ist der Hold-Space leer, wird trotzdem ein Newline angehängt:

[*adr*][`,` *adr*]`G`

Hold-ML

Das Hold-ML (`H`) Kommando fügt ein Newline und den Puffer an den Inhalt des Hold-Spaces an. Das Newline wird auch angefügt, wenn der Hold-Space komplett leer ist:

[*adr*][`,` *adr*]`H`

Next-ML

Das Next-ML Kommando (`N`) liest die nächste Eingabezeile und hängt sie, durch ein Newline getrennt, an den Puffer an:

[*adr*][`,` *adr*]`N`

Print-ML

Das Print-ML Kommando (`P`) gibt den Teil des Puffers bis zum ersten Newline aus. Ansonsten identisch zum Print (`p`) Kommando:

[*adr*][`,` *adr*]`P`

2.4 Beispiele

2.4.1 Konfigurationsfiles

Oft kann man Files generisch anlegen, d.h. große Teile der Files sind konstant, nur einige Teile müssen je nach der Umgebung oder den Eingabedaten angepaßt werden.
Makefiles sind ein Beispiel hierfür; einige Teile, z.B. die Pfade zu Installations-directories müssen je nach Installation angepaßt werden. `sed` kann nun benutzt werden, um diese Einfügungen zu machen, indem man für solche Parameter Platzhalter vorsieht, die dann mittels des Substitute Kommandos durch die wirklichen

Plätze ersetzt[11]. Man erstellt also eine Schablone (z.B. `Makefile.in`), die die Platzhalter, z.B. in der Form `@INSTDIR@`, enthält und lässt `sed` dann ein Script mit dem folgenden Kommando ausführen: **s/@INSTDIR@/*real-inst-dir*/g**.
Dies kann man einfach in ein Bourne-Shell Script integrieren:

```
#! /bin/sh

....

#$instdir contains the dir  the stuff should be put to
#$programs contains the list of programs to build
...

# generate sed script
cat >/tmp/sed.scr <<EOM
s/@INSTDIR@/${instdir}/g
s/@PROGS@/${programs}/g
s/@VERSION@/${progversion}/g
EOM

for f in Makefile.in lib/Makefile.in doc/Makefile.in;
 do
 outfile=`echo $f | sed -e 's/\.in$//'`
 sed -f /tmp/sed.scr $f >$outfile
 echo created $outfile
done

rm /tmp/sed.scr
```

Dieses Shell Script generiert ein temporäres `sed` Script `/tmp/sed.scr`, wobei die Shell Variablen (`$instdir`, ...) durch deren Werte ersetzt sind. Danach wird für jede Schablone (`Makefile.in`, ...) die Ersetzung durchgeführt und ein File erzeugt, das denselben Namen wie die Schablone hat, ausser der Endung `.in`. Hier wurde `sed` benutzt, um die Endung abzuschneiden. Eine Alternative wäre ``outfile=`dirname $f`/`basename $f .in` `` gewesen. Allerdings kann man `sed` auch bei beliebigen Endungen benutzen (`s/\.[^.]*$//`).

[11]Dies ist der Weg, den das GNU `configure` Tool geht, welches automatisch Software-Packete für die vorhandene Architektur konfiguriert.

2.4.2 Extraktion von Daten

Angenommen, man hat Datenfiles (`dat.1`, `dat.2`, ...), in denen z.B. der Name des Datensatzes in folgender Weise vermerkt ist:

```
....
%%%Datensatz: Experiment_mit_Optimierung_1
....
```

Bei der Auswertung möchte man nun z.B. wieder in einem Shell Script über alle Datenfiles laufen, den Namen des Datensatzes ausgeben und danach einige Ergebnisse. Ein Fragment des Shell Scriptes könnte dann so aussehen:

```
....
# Zsfsg. generieren
for f in dat.*; do
 dname=`sed -n -e 's/^%%%Datensatz: \(.*\)$/\1/p' $f`
 echo Datensatz $dname >>Zsfsg

 # gen. Datenzsfsg.
 ....
done
```

Hier wird die `-n` Option benutzt, die verhindert, daß Zeilen automatisch ausgegeben werden. Das Substitute Kommando ersetzt dann den Namen des Datensatzes und gibt diesen aus (durch die `p` Option).

2.4.3 Umformungen

Angenommen, man hat in einer Shell Variablen eine Liste von Directories, die durch Doppelpunkte getrennt sind, etwa `$MANPATH` oder `$PATH`. Wenn man jetzt in jeder dieser Directories z.B. nach einem bestimmten File suchen will kann man (für die tcsh) z.B. folgendes eingeben:

```
foreach d (`echo $MANPATH|sed -e 's/:/ /g'`)
 if ( -f $d/whatis ) echo $d hat apropos Datenbank
end
```

Das `sed` Script ersetzt einfach die Doppelpunkte durch Leerzeichen, das Ergebnis wird dann von der Schleife als eine Liste von Directories benutzt.
Angenommen, man will einen Namen daraufhin prüfen, ob er ein legaler Shell Variablenname ist. Folgender Test in einem Bourne Shell Script kann dies leisten (der Name steht in der Variablen `$aname`):

```
if [ -n "`echo $aname|sed -e 's/[-a-zA-Z0-9_]//g'`" ];
 then
 echo illegaler name
else
 ...
fi
```

Das Script ersetzt jeden Buchstaben, jede Ziffer, jeden Trennstrich und Unterstrich durch nichts, löscht sie also. Ist das Ergebnis nicht leer (der `-n` Test in der Shell), existiert ein illegales Zeichen, der Name ist also ungültig.
Manchmal will man bei einem Namen alle Schrägstriche am Ende entfernen. Dies kann einfach durch die folgende Anweisung erledigt werden:

```
newname=`echo $name | sed -e 's+/*$++'`
```

Da `sed` immer den längsten Match vornimmt werden durch diese Substitution alle Schrägstriche am Ende entfernt. Man beachte auch die Verwendung von + statt `/` als Trenner des Patterns von der Ersetzung.
Angenommen, man möchte einen Namen, der u.a. Trennstriche ('`-`'), Punkte, Schrägstriche oder Pluszeichen enthält in einen Namen verwandeln, der dem C-Präprozessor als Macroname angeboten werden kann. Hier die Lösung:

```
cppname=`echo $aname| sed -e 'y%./-+%___p%'`
```

Hierbei werden Pluszeichen durch ein 'p' ersetzt, alles andere durch Unterstriche.

2.5 Falltüren

2.5.1 Pattern

Die häufigste Tretangel in `sed` sind dessen Pattern und das Matching. `sed` kennt keine Gruppierungen in Pattern mittels Klammerung, Meta-Operatoren, z.B. '`*`', wirken immer nur auf ein Zeichen. Auch kann man in `sed` keine Sonderzeichen, etwa TAB mittels der gewohnten Escape-Sequenzen darstellen (etwa `\t`). Diese Zeichen müssen roh eingegeben werden, wozu man also den Quote-Mechanismus des Editors oder der Shell bemühen muss: CTRL-V beim vi und der Shell, CTRL-Q beim emacs. Die einzige Ausnahme sind Multi-Line Pattern, in denen ein Newline mittels `\n` gematcht werden kann. Allerdings kann man in dem Ersetzungstext ein Newline *nicht* mittels `\n` einfügen, sondern muß dies tun, indem man einen Rückstrich ('`\`') eingibt, sofort gefolgt von dem Newline.
Verwirrung schafft häufig auch die Tatsache, daß `sed` immer den längsten Match von links her vornimmt, somit spätere Pattern nicht auf den Teil der Eingabe passen, den man im Sinn hatte, etwa: `s/\(.*\) \(.*\)/\1-\2/` Dies führt

auf der Eingabe 'abcd efg hij' *nicht* zu der vielleicht beabsichtigten Ausgabe 'abcd-efg hij', sondern zu 'abcd efg-hij'. Richtig wärte gewesen 's/\([^]*\) \(.*\)/\1-\2/'.

2.5.2 Zusatztext

Oft tritt man auch auf die Tretmine, daß die Texte, welche mittels eines Append, Insert oder Read Kommandos angehängt wurden, nicht von den folgenden Kommandos des Scriptes bearbeitet werden. Sie werden auch nicht von dem Hold Kommando in den Hold-Space übernommen.

2.5.3 Reihenfolgen

Man sollte auch berücksichtigen, daß Substitutionen von späteren Substitutionen wieder verändert werden können, z.B. wenn man in einem File alle Variablen namens '$A' nach '$B' und alle Variablen namen '$B' nach '$C' umbenennen will:

```
s/\$A/$B/g
s/\$B/$C/g
```

Dieses Script führt leider dazu, daß alle '$A' nach '$C', über den Umweg '$B', umbenannt werden. Richtig wäre gewesen:

```
s/\$B/$C/g
s/\$A/$B/g
```

2.6 Literatur

Als erstes sind natürlich die Manual-Pages und die Info Seiten von sed zu nennen. Wer es etwas ausführlicher mag, dem sei das gute Buch [Dou90] empfohlen, an welches sich auch dieses Kapitel angelehnt hat.
Beispiele, an denen man lernen kann, finden sich z.B. in den configure Shell Scripten, die fast jedem GNU Programm im Source-Archiv beiliegen. Unter NetBSD kann man unter /usr/src/usr.bin/sed/TEST ein sed Script bestaunen, welches die Lösung zum 'Turm-von-Hanoi' berechnet. Außerdem findet man dort ein Script, das mathematische Ausdrücke ausrechenen kann (in sed). Der GNU Source-Distribution von sed liegen unter dem Directory testsuite eine Reihe von Scripten bei, die man sich durchaus ansehen sollte.