

# **Ein awk-Tutorium**

Martin Oehm

# Inhaltsverzeichnis

<b>Vorwort.....</b>	<b>1</b>
Weiterführende Links.....	1
<b>1 Erste Schritte.....</b>	<b>2</b>
1.1 Was ist awk?.....	2
1.2 Wie arbeitet awk?.....	3
1.3 awk findet Wörter.....	4
<b>2 Bedingungen und Variablen.....</b>	<b>5</b>
2.1 Bedingungen.....	5
2.2 Variablen.....	7
<b>3 Ein- und Ausgabeformate.....</b>	<b>10</b>
3.1 Formatierte Ausgabe.....	10
3.2 Formatgebundens Einlesen mit Substrings.....	11
3.3 Weitere Stringoperationen.....	12
<b>4 Komplexere Programme.....</b>	<b>14</b>
4.1 Wenn-dann Konstrukte.....	14
4.2 Schleifen.....	15
4.3 Funktionen.....	16
4.4 Weitere Funktionen.....	17
4.5 Felder.....	18
<b>5 Für Fortgeschrittene.....</b>	<b>20</b>
5.1 Reguläre Ausdrücke – die ganze Wahrheit.....	20
5.2 Weitere Variablen in awk.....	21
<b>Anhang A: Kurzübersicht.....</b>	<b>23</b>
<b>Anhang B: Lösungen.....</b>	<b>25</b>

# Vorwort

*awk* wurde Ende der Siebziger Jahre von Alfred V. Aho, Peter J. Weinberger und Brian W. Kernighan geschrieben und danach stets erweitert. Es ist eine leicht zu lernende Skriptsprache zum schnellen Bearbeiten von Texten.

Für meine Arbeit in der technischen Simulation ist *awk* sehr nützlich, um schnell Änderungen in Datensätzen für *Nastran*, *PAM-Crash* oder andere Solver vorzunehmen. Mit ein paar Zeilen wird das erledigt, was in einem *Pascal*- oder *C*-Programm viel komplizierter wäre. In letzter Zeit wird zunehmend *Perl* benutzt, das weitaus mächtiger ist als *awk*. Für die meisten kleinen Aufgaben finde ich *awk* wegen seiner speziell auf Textbearbeitung ausgerichteten und unkomplizierten Syntax dennoch geeigneter.

Dieses Tutorium soll die Grundlagen von *awk* vermitteln. Es richtet sich hauptsächlich an Ingenieure, denen *awk* ihre tägliche Arbeit mit großen Datensätzen erleichtern kann. Viele der Übungen und Beispiele sind aus dem Bereich der FEM-Berechnung.

*awk* ist in verschiedenen Versionen erhältlich, den meisten UNIX- und Linux-Distributionen liegt es bei. Wer einen Windows-PC hat, der erhält *awk* als Teil von Cygwin, der BASH-Shell für Windows.

## Weiterführende Links

The awk Manual:

[http://www.cs.uu.nl/docs/vakken/st/nawk/nawk\\_toc.html](http://www.cs.uu.nl/docs/vakken/st/nawk/nawk_toc.html)

GNU Awk User's Guide:

<http://www.gnu.org/manual/gawk/index.html>

comp.lang.awk FAQ:

<http://www.faqs.org/faqs/computer-lang/awk/faq/>

Cygwin:

<http://www.cygwin.com>

# 1 Erste Schritte

## 1.1 Was ist awk?

*awk* ist ein Werkzeug zum Bearbeiten von strukturierten Texten, wie z.B. Datensätzen für FE-Programme. Es bietet Such- und Ersetzungsfunktionen, formatiertes und unformatiertes Einlesen und Herausschreiben und eine mächtige Skriptsprache. All diese Dinge können je nach Gusto und Anforderungen des Benutzers beliebig einfach oder kompliziert gehalten werden.

*awk*-Skripte können auf verschiedene Weise aufgerufen werden:

### i) Aus der Shell heraus:

```
cmd> awk 'script' textdatei
```

*<script>* ist das Skript, das mit Hochkommas von der Shell geschützt wird. Die Hochkommas bewirken, dass das Skript als ein „Wort“ in der Shell betrachtet wird. wird.

*<textdatei>* ist die zu lesende Datei. Es können mehrere Dateien angegeben werden, die dann nacheinander abgearbeitet werden. Wird keine Datei angegeben, so erfolgt das Einlesen über **stdin**.

Die Ausgabe erfolgt über **stdout** und kann mit **>** in eine andere Textdatei umgelenkt werden.

### ii) Als separates Skript

```
cmd> awk -f script textdatei
```

Hier wird das Skript nicht auf der Kommandozeile angegeben, sondern steht in einer separaten Datei *<script>*. Dieses Vorgehen bietet sich insbesondere für längere Skripte an.

Die Shell ermöglicht es, den *awk*-Aufruf direkt in den Kopf der Skriptdatei zu schreiben. Steht in der ersten Zeile des Skripts

```
#!/bin/awk -f
```

so kann die Skriptdatei direkt mit Namen aufgerufen werden. Die Skriptdatei muss dann für den Benutzer ausführbar und lesbar sein (**r-x** oder oktal 5). Außerdem wurde hier angenommen, dass sich *awk* unter **/bin/** befindet. Wer Zweifel hat, wo sein *awk* liegt, sagt **which awk**.

## 1.2 Wie arbeitet awk?

*awk* liest die Daten zeilenweise aus der angegebenen Textdatei ein. In jeder Zeile wird das Skript ausgeführt. Die Anweisungen im Skript haben dabei die Form

```
⟨Bedingung⟩ { ⟨Anweisung(en)⟩ }
```

Ist die *⟨Bedingung⟩* erfüllt, so werden die *⟨Anweisungen⟩* in den geschweiften Klammern ausgeführt. Fehlt die Bedingung, so werden die Anweisungen für jede Zeile ausgeführt. Fehlt der Anweisungsteil in geschweiften Klammern, so wird die Zeile, die die Bedingung erfüllt, einfach ausgegeben.

Eine typische Anweisung ist **print**, das einfach die eingelesene Zeile ausgibt. Wie oben beschrieben, kann { **print** } einfach weggelassen werden.

Ein Beispiel für eine typische Bedingung ist */⟨text⟩/*. Diese Bedingung ist wahr, wenn der String *⟨text⟩* im der eingelesenen Zeile auftaucht.

Unser erstes *awk*-Skript soll einfach alle Zeilen der Datei **xyz.dat** ausgeben, die das Wort „NODE“ enthalten:

```
awk '/NODE/ { print }' xyz.dat
```

Da { **print** } weggelassen werden kann, kann man dies verkürzen zu

```
awk '/NODE/' xyz.dat
```

Dieses *awk*-Skript ist natürlich äquivalent zum Shell-Befehl

```
grep 'NODE' xyz.dat
```

Das noch viel einfachere

```
awk '{ print }' xyz.dat
```

ist dasselbe wie

```
cat xyz.dat
```

wobei hier { **print** } nicht ausgelassen werden kann, da sonst das Skript leer wäre, was *awk* interpretiert als „mache gar nichts“.

## 1.3 awk findet Wörter

Wenn *awk* eine Zeile einliest, so schreibt es sie auf die Variable **\$0**. Außerdem teilt *awk* die Zeile automatisch in Wörter (oder „Felder“) auf, die **\$1**, **\$2**, **\$3** usw. heißen. Ein Wort ist dabei alles, was zwischen zwei Leerzeichen steht.

Weiterhin werden folgende Variablen belegt: **NF** (*Number of Fields*) ist die Anzahl der Wörter, in die die Zeile aufgeteilt wurde. **NR** (*Number of Records*) ist die Zeilennummer der momentan eingelesenen Zeile. Eine Zeile wird in *awk* oft als *Record*, Eintrag, bezeichnet.

✘ *awk* berücksichtigt Groß- und Kleinschreibung. So ist **NF** die Anzahl der Felder; **nf**, **Nf** und **nF** sind es nicht.

Wenn die Zeile also

```
(Fischers Fritz fischt frische Fische.)
```

heißt, so ist **NF** fünf. Satzzeichen gehören zu den Wörtern, deshalb heißt das letzte Wort "**Fische.**)", nicht bloß "**Fische**". *awk* hat also keine menschlichen Lesegewohnheiten.

Das Skript

```
awk '{print $NF}' xyz.dat
```

schreibt übrigens das letzte Wort jeder Zeile heraus. An diesem Skript kann man sehen, dass nach dem Dollar anstelle einer Zahl auch eine Variable stehen kann, und dass **print** nicht nur alleine stehen kann, sondern konkret angegeben werden kann, was ausgegeben werden soll. **print** ohne alles ist also nur eine Abkürzung für **print \$0**.

### Aufgaben:

#### Aufgabe 1a

Schreibe ein Skript, das die beiden ersten Wörter einer Zeile in umgekehrter Reihenfolge ausgibt.

#### Aufgabe 1b

Schreibe ein Skript, das alle Zeilen einer Datei ausgibt und dabei die Zeilennummer vor eine Zeile schreibt.

## 2 Bedingungen und Variablen

### 2.1 Bedingungen

Wir haben bereits die Bedingung  $/\langle text \rangle/$  kennengelernt, die nach dem Text  $\langle text \rangle$  sucht. Dabei kann  $\langle text \rangle$  auch einfache so genannte *reguläre Ausdrücke* enthalten:

•  
bedeutet ein beliebiges Zeichen:  $/\mathbf{H}\mathbf{a}\mathbf{.}\mathbf{s}/$  findet „Hans“, „Haus“, „Hass“, „Hals“.

^  
am Anfang des Ausdrucks bedeutet, dass der gesuchte Text am Anfang der Zeile stehen muss. Leerzeichen am Anfang werden berücksichtigt.  $/\mathbf{^a}/$  findet „alles“, aber nicht „ alles“.

\$  
am Ende des Ausdrucks bedeutet, dass der Text am Ende stehen muss.  $/\mathbf{z}\mathbf{\$}/$  findet „ganz“, nicht „ganzes“.

[ab]  
bedeutet, dass einer der Buchstaben „a“ oder „b“ an dieser Stelle stehen muss.  $/\mathbf{H}\mathbf{a}[\mathbf{u}]\mathbf{s}/$  findet „Hals“ und „Haus“, nicht aber „Hans“.

[a-z]  
bedeutet, dass hier ein Buchstabe von „a“ bis „z“ stehen muss. Das - bezieht sich auf die Nummer des Zeichens im ASCII-Code.  $/\mathbf{H}\mathbf{a}[\mathbf{i-n}]\mathbf{s}/$  findet „Hals“, „Hans“, aber nicht „Hass“ oder „Haus“.

✘ Will man eines dieser speziellen Zeichen im Klartext finden, so muss ein *Backslash*, der rückläufige Schrägstrich, vorangestellt werden:  $\backslash\mathbf{.}$  findet einen Punkt,  $\backslash\backslash$  einen Backslash. Der Backslash kann auch bei anderen Zeichen benutzt werden, er findet dann dieses Zeichen. Wer also nicht genau weiß, ob ein Zeichen eine besondere Bedeutung hat, stellt einfach einen Backslash voran:  $\backslash\mathbf{Y}$  findet ein großes Ypsilon.

Es gibt weitere Bedingungen, die meist in einer der Programmiersprache C ähnlichen Syntax angegeben werden:

$\mathbf{a == b}$  ist wahr, wenn **a** und **b** gleich sind.  
 $\mathbf{a != b}$  ist wahr, wenn **a** und **b** ungleich sind.  
 $\mathbf{a < b}$  ist wahr, wenn **a** kleiner als **b** ist.  
 $\mathbf{a <= b}$  ist wahr, wenn **a** kleiner als oder gleich **b** ist.  
 $\mathbf{a > b}$  ist wahr, wenn **a** größer als **b** ist.  
 $\mathbf{a >= b}$  ist wahr, wenn **a** größer als oder gleich **b** ist.

Das Skript

```
$1=="NODE" { print $2 }
```

schreibt also das zweite Wort heraus, wenn das erste „NODE“ ist. Zeichenketten, so genannte *Strings*, müssen hier immer in Anführungszeichen stehen, um deutlich zu machen, dass es sich um einen String und nicht um eine Variable handelt.

✗ Vorsicht bei der Prüfung auf Gleichheit:  $a = b$  belegt  $a$  mit dem Wert von  $b$ . In *awk* muss die Prüfung auf Gleichheit wie in C mit einem doppelten Gleichheitszeichen erfolgen!

(Es ist sogar noch schlimmer:  $a = b$  ist für *awk* wahr, wenn  $b$  nicht Null ist. Die Belegung von  $a$  erfolgt aber trotzdem: Etwas wie wie

```
$1="xxx" { print $1, $2, $3 }
```

hat merkwürdige Folgen.)

Für Strings gibt es zwei weitere Operatoren:  $\sim$  und  $!\sim$ , die prüfen, ob ein String einen regulären Ausdruck enthält oder nicht.  $/^[0-9]/$  und  $\$0 \sim /^[0-9]/$  bewirken dasselbe: Sie überprüfen, ob die Zeile mit einer Ziffer beginnt. Mit  $\sim$  kann man also beliebige Strings auf reguläre Ausdrücke hin überprüfen.

Bedingungen können mit so genannten *logischen Operatoren* miteinander verknüpft werden:

$a \ \&\& \ b$

ist wahr, wenn beide Bedingungen  $a$  und  $b$  wahr sind.

$a \ || \ b$

ist wahr, wenn mindestens eine der Bedingungen  $a$  und  $b$  wahr ist.

$!a$

ist wahr, wenn  $a$  falsch ist (Negation).

Bedingungen können weiterhin in Klammern geschachtelt werden, um Prioritäten bei der Auswertung zu setzen. Außerdem gibt es noch zwei nützliche Regeln aus der Logik, die Gesetze von *de Morgan*:

```
!(a && b) == !a || !b
!(a || b) == !a && !b
```

Außerdem können zwei Bedingungen hintereinander, durch ein Komma getrennt, angegeben werden. Dann werden die Anweisungen für alle Zeilen ausgeführt, die zwischen einer Zeile, für die die erste Bedingung gilt, und der nächsten, für die die zweite gilt, liegen.

```
/Anfang/, /Ende/
```

Schreibt alle Zeilen, die zwischen den Wörtern „Anfang“ und „Ende“ liegen. Die erste Bedingung ist also eine Start-, die zweite eine Abbruchbedingung.

Es gibt noch zwei weitere Bedingungen in *awk*, die sehr wichtig sind, die wir aber erst später benutzen werden:

**BEGIN**

Die Anweisungen, die nach **BEGIN** stehen, werden ganz am Anfang ausgeführt, bevor die erste Zeile eingelesen wird.

**END**

Analog dazu werden die Anweisungen nach **END** nach dem Einlesen aller Zeilen abgearbeitet.

## 2.2 Variablen

Das Konzept der Variablen in *awk* ist sehr mächtig. Variablen müssen nicht, wie in Programmiersprachen allgemein üblich, deklariert werden, bevor sie benutzt werden. Um eine Variable zu benutzen, setzt man sie einfach im Skript ein. Jede Variable hat zu Beginn des Skripts den Wert 0.

Dazu ein Beispiel:

```
awk '/NODE/ { n = n+1 }; END { print n }' xxx.dat
```

Dieses Skript zählt die Zeilen, in denen „NODE“ vorkommt, in der Variable *n* und gibt *n* am Ende aus. Dies ist äquivalent zu

```
grep -c NODE xxx.dat
```

Verschiedene Anweisungen können durch Semikola oder Zeilenumbrüche voneinander getrennt hintereinander angegeben werden. Letzteres funktioniert natürlich nur, wenn das Skript in einer separaten Datei steht. Wenn eine Anweisung über mehrere Zeilen geht, so müssen alle Zeilen der Anweisung außer der letzten mit einem *Backslash* enden.

✘ Anstelle der Anweisung `n = n + 1`, die wörtlich bedeutet: „Weise der Variable *n* den jetzigen Wert von *n* plus eins zu, und die nichts anderes macht, als *n* um eins zu erhöhen, kann man in *awk* abkürzend schreiben `n++`.

Variablen in *awk* können zwei Grundtypen besitzen: Zahl oder String. Dabei ist *awk* bei der Interpretation, um welchen Typ es sich handelt, flexibel.

Mit Zahlen können folgende Operationen durchgeführt werden:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo-Division
^	Potenzieren

Das Ergebnis der Division ist eine Fließkommazahl. Mit Modulo-Division bezeichnet man

den Rest einer Ganzzahldivision: `7%3` ist 1, da sieben geteilt durch drei einen Rest von eins lässt.

✘ Für alle diese Funktionen gibt es Abkürzungen, wenn dieselbe Variable auf beiden Seiten vorkommt. So kann man `x = x + 10` als `x += 10` schreiben. Analog dazu gibt es die Operatoren `--`, `*=`, `/=`, `%=` und `^=`. Diese Schreibweise ist für Anfänger aber oft verwirrend, deshalb ist es vermutlich besser, den Ausdruck komplett anzugeben.

✘ Die Operatoren `++` und `--` können vor oder hinter Variablen stehen. `++` vor einer Variable (*Präfix*) bedeutet „erhöhe die Variable um eins, und werte dann aus“, nach einer Variable (*Postfix*) bedeute es „werte die Variable aus, erhöhe sie dann um eins“.

Ist `x` 5, so erhöht sich der Wert bei `print ++x` und bei `print x++` in beiden Fällen auf 6. Im ersten Fall wird jedoch „6“, im zweiten „5“ ausgegeben.

Die einzige elementare Funktion für Strings ist das Anhängen eines anderen Strings, das einfach durch Hintereinanderschreiben der Strings, durch mindestens ein Blank getrennt, geschieht:

```
name = "Hans" "Meiser";
print name;
```

gibt „HansMeiser“ aus. Es wird also kein Leer- oder anderes Zeichen zwischen die verketteten Strings geschrieben.

Wie bereits erwähnt, können Variablen ihren Typ ändern. Führt man mit Strings eine mathematische Operation durch, so ist das Ergebnis eine Zahl. Andersherum geht es aber auch: Fügt man an eine Zahl einen String an, so ist das Ergebnis ein String.

Dazu ein Beispiel:

```
$1=="NODE" {x = x + $2; n = n + 1}
END {print x, n, x/n}
```

Hier werden für alle Zeilen, in denen das erste Wort „NODE“ ist, die Zahlenwerte der zweiten „Wörter“ aufaddiert. Am Ende wird die Summe, die Anzahl und der Durchschnitt ausgegeben.

Es gibt keine explizite Anweisung, um Strings in Zahlen umzuwandeln und umgekehrt. Um sicherzugehen, kann man folgendes machen: `x + 0` ist eine Zahl, `x ""` ist ein String. (`""` ist der Nullstring, ein String ohne Inhalt.)

✘ Der numerische Wert von Strings, die keine Zahl enthalten, ist Null: `"Anna"` hat einen numerischen Wert von Null. `awk` kann aber die Exponentialschreibweise für Fließkommazahlen erkennen: `"2.4E-3"` wird korrekt als 0,0024 interpretiert.

## Aufgaben:

### Aufgabe 2a

Schreibe ein Skript, das nur jede zweite Zeile eines Datensatzes ausgibt.

### Aufgabe 2b

Wie sähe ein Skript aus, das den Shell-Befehl `wc` (*word count*) simuliert? `wc` gibt die Anzahl der Zeilen, Wörter und Zeichen in einer Textdatei aus.

### Aufgabe 2c

Wie sieht ein Skript aus, das das Produkt der zweiten und dritten Zahl einer Zeile bildet, wenn das erste Wort „mult“ ist, und diese dann aufsummiert. Die Ausgabe soll nur die Summe der Produkte sein, nicht die Produkte selbst.

### Aufgabe 2d

Wie sieht ein Skript aus, das vor jeder Zeile, die eingerückt ist, also mit einem Leerzeichen oder einem Tab beginnt, eine Leerzeile schreibt? (Ein Tab wird in *awk* durch die Zeichenfolge `\t` repräsentiert.)

### Aufgabe 2e

Schreibe ein Skript, das eine Zeile nur dann ausgibt, wenn das erste Wort von dem in der vorhergehenden Zeile verschieden ist.

## 3 Ein- und Ausgabeformate

### 3.1 Formatierte Ausgabe

Bisher kennen wir nur die Ausgabe mit `print`. Dies ist die einfachste Form, in der alle Argumente, die durch Kommas getrennt werden, nacheinander durch Leerzeichen getrennt ausgegeben werden.

Eine typische Ausgabe für das Beispiel oben, `print x, n, x/n`, wäre:

```
1524 108 14.1111
```

Jeder `print`-Befehl schreibt eine neue Zeile.

Gerade für Datensätze, die meist einem strengen Format unterliegen, möchte man aber eine bessere Formatierung der Ausgabe. Dazu gibt es den `printf`-Befehl (*print formatted*):

```
printf "<format>", <argumente>
```

Es kann eine beliebige Anzahl von Argumenten übergeben werden. Für jedes übergebene Argument muss im `<format>`-String ein Ausgabeformat, das mit dem Prozentzeichen `%` beginnt, angegeben werden.

Die wichtigsten Formate sind:

<code>%s</code>	String
<code>%d</code>	ganze Zahl ( <i>decimal</i> )
<code>%f</code>	Fließkommazahl ( <i>float</i> )
<code>%e</code>	Fließkommazahl immer in Exponentenschreibweise
<code>%g</code>	wie <code>%e</code> für Zahlen ab <code>1e06</code> , <code>%f</code> sonst
<code>%c</code>	Einzelnes ASCII-Zeichen ( <i>character</i> )
<code>%x</code>	ganze Zahl als Hexadezimalzahl

Buchstaben in Zahlen bei `%e`, `%g` und `%x` werden klein ausgegeben. Für Großbuchstaben gibt es die Notationen `%E`, `%G` und `%X`.

Zwischen dem Prozentzeichen und dem Ausgabebetyp können Angaben zur Breite des Ausgabefeldes gemacht werden. `%8d` passt eine ganze Zahl rechtsbündig in ein Feld von 8 Zeichen ein. Geht der Breite ein Minus voran, so erfolgt die Ausgabe ins Feld linksbündig: `%-8d`. Wird ein Plus vorangestellt, so wird auf jeden Fall ein Vorzeichen ausgegeben, auch wenn die Zahl positiv ist.

Für Fließkommazahlen kann nach einem Dezimalpunkt die Anzahl der signifikanten Nachkommastellen angegeben werden: `%16.4f` schreibt die Zahl mit vier Nachkommastellen in ein Feld von 16 Zeichen Breite.

Beginnt die Breitenangabe mit einer Null, so werden die unbenutzten Felder links nicht mit Leerzeichen, sondern mit Nullen aufgefüllt:

```
printf "Uhrzeit: %02d:%02d", hrs, min;
```

Alles im Formatstring, was nicht zu einer Formatangabe mit % gehört, wird wortwörtlich ausgegeben. Nach `printf` beginnt nicht automatisch eine neue Zeile wie bei `print`. Eine neue Zeile wird mit dem Sonderzeichen `\n` eingeleitet:

```
printf "ersteZeile\nzweiteZeile\ndritteZeile\n"
```

Es gibt mehrere solche Sonderzeichen, die mit einem Backslash beginnen:

<code>\n</code>	Zeilenumbruch ( <i>newline</i> )
<code>\f</code>	Seitenumbruch ( <i>form feed</i> )
<code>\t</code>	Tabulator
<code>\b</code>	Backspace
<code>\a</code>	Piepston ( <i>alert</i> )
<code>\\</code>	Backslash
<code>\"</code>	Anführungsstriche
<code>\'</code>	Hochkommas
<code>\0</code>	Das Nullzeichen am Ende des Strings

✘ Strings können in *awk* beliebig lang sein. Sie hören mit dem Zeichen `\0`, das dem ASCII-Code 0 entspricht, auf (*nullterminierte Strings*).

## 3.2 Formatgebundens Einlesen mit Substrings

*Substrings* sind Teile von Strings. In *awk* bezeichnet

```
substr (<string>, <anfang>, <länge>)
```

den Substring von `<String>` ab der Stelle `<anfang>` der Länge `<länge>`. Das erste Zeichen des Strings hat dabei die Position 1. `substr ("Schande", 3, 4)` ist also „hand“. Wenn `<länge>` weggelassen wird, geht der Substring bis zum Ende des Strings.

Betachten wir also eine Eingabekarte nach dem typischen Achter-Format, etwa eine Karte mit Knotenkoordinaten:

```
$.....|.....id.|.....x.|.....y.|.....z.|
NODE          12 12.0000 33.6773-992.762
```

Um nun den Durchschnitt aller Knotenkoordinaten zu bilden, lautet das Skript:

```
substr($0,1,4)=="NODE" {
    x += substr($0,17,8);
    y += substr($0,25,8);
```

```

    z += substr($0,33,8);
    n++;
}
END {
    printf "NODE      AVERAGE %8.2f%8.2f%8.2f",
           x/n ,y/n ,z/n;
}

```

✘ Da dieses Skript schon sehr lang ist, schreibt man es besser in eine eigene Datei. Dann kann man die Anweisungen zeilenweise angeben, das sieht übersichtlicher aus.

### 3.3 Weitere Stringoperationen

Es gibt noch weitere nützliche Stringoperationen, die hier kurz vorgestellt werden:

**sub**(*<alt>*, *<neu>*, *<string>*)

(*substitute*) ersetzt das erste Vorkommen von *<alt>* im String *<string>* durch *<neu>*. *<alt>* kann dabei ein regulärer Ausdruck in Schrägstrichen sein. Wird *<string>* nicht angegeben, so wird in **\$0** ersetzt.

**gsub**(*<alt>*, *<neu>*, *<string>*)

(*globally substitute*) ist wie **sub**, ersetzt aber *alle* Vorkommnisse von *<alt>* im String *<string>* durch *<neu>*.

**length**(*<string>*)

gibt die Länge eines Strings an oder der ganzen Zeile, wenn kein String angegeben wurde.

**tolower**(*<string>*)

wandelt alle Großbuchstaben in *<string>* in kleine um.

**toupper**(*<string>*)

wandelt alle Kleinbuchstaben in *<string>* in große um. Umlaute und andere Zeichen mit Akzente werden bei **tolower** und **toupper** nicht berücksichtigt.

**index**(*<string>*, *<suchstring>*)

ist die Position in *<string>*, an der *<suchstring>* zum ersten Mal auftritt, oder Null, wenn der String überhaupt nicht auftritt.

**index**("Mississippi", "si") ist 4.

**match**(*<string>*, *<suchstring>*)

ist wie **index**, nur dass reguläre Ausdrücke angegeben werden können:

**match** ("Mississippi", /[si]/) ist 2.

#### Aufgaben:

#### **Aufgabe 3a**

Schreibe ein Skript, das die Felder einer Zeile formatiert in Achterfelder herausschreibt. Das erste Feld ist ein String, das zweite ein Integer und die drei nächsten sollen Fließkommazahlen sein.

#### **Aufgabe 3b**

Schreibe ein Skript, das den umgekehrten Weg geht: Ein formatierter Input (acht Zeichen String, acht Integer und drei mal acht Zeichen Fließkomma) soll unformatiert, d.h. nur durch Leerzeichen getrennt herausgeschrieben werden.

#### **Aufgabe 3c**

Schreibe ein Skript, das alle Zeilen, die mit einer Ziffer beginnen, in Großbuchstaben umwandelt. Dabei sollen die deutschen Umlaute berücksichtigt werden, „ß“ soll in „ss“ umgewandelt werden.

## 4 Komplexere Programme

### 4.1 Wenn-dann Konstrukte

Bedingungen können nicht nur außerhalb des Anweisungsblocks mit geschweiften Klammern stehen, sie können auch innerhalb abgefragt werden. Dazu gibt es die **if**-Anweisung:

```
if (<Bedingung>) <einzelne Anweisung>
if (<Bedingung>) {<mehrere Anweisungen>}
```

Wenn mehrere Anweisungen von der Bedingung abhängig ausgeführt werden sollen, so müssen sie zu einem Block in geschweiften Klammern zusammengefasst werden.

Also sind die beiden Zeilen

```
$1=="x" { print }
{ if ($1=="x") print }
```

gleichwertig. Für einfache Abfragen sind Abfragen außerhalb des `{ }`-Blocks wohl übersichtlicher, **if**-Anweisungen dienen eher zur weiteren Abfrage innerhalb des Blocks.

Einen großen Vorteil bietet die **if**-Anweisung jedoch: Mit dem Zusatz **else** können Ausführungen angegeben werden, die nur dann ausgeführt werden, wenn die Bedingung nicht erfüllt ist:

```
if (<Bedingung>) <Anweisung A> else <Anweisung B>
```

✘ Auch hier können Anweisungen zu Blöcken zusammengefasst werden. Wird nur eine einzelne Anweisung zu `<A>` angegeben, so muß diese mit einem Semikolon abgeschlossen werden, auch wenn dies auf den ersten Blick den „Satzfluss“ stört.

Also, wieder ein Beispiel:

```
{ if (/^\$/ ) print > "comment.dat"; else print; }
```

Hier werden alle Zeilen, die mit einem „\$“ beginnen, in eine separate Datei `comment.dat` geschrieben, alle anderen auf den normalen Output.

✘ Mit `>` können wie in der Shell Ausgaben in Dateien umgeleitet werden. Achtung: Dateinamen sind in `awk` immer Strings und müssen daher in Anführungsstrichen stehen.

## 4.2 Schleifen

Ein weiteres wichtiges Instrument sind Schleifen. Eine typische Schleife ist die `while`-Schleife:

```
while (<Bedingung>) <Anweisung>
```

Hier wird die Anweisung solange ausgeführt, bis die Bedingung falsch wird. Ist die Bedingung von Anfang an falsch, passiert nichts:

```
n = 1; while (n < 256) {print n; n*=2}
```

Natürlich muss in der Anweisung etwas passieren, das die Bedingung irgendwann einmal falsch werden lässt. Etwas wie

```
n = 0; while (n < 5) {print n}
```

führt zu einer so genannten *Endlosschleife*, die ununterbrochen den Wert 0 ausgibt, der natürlich immer kleiner als fünf ist.

Eine Abwandlung der `while`-Schleife ist die `do-while`-Schleife:

```
do <Anweisung> while (<Bedingung>)
```

Auch hier werden die Anweisungen solange durchlaufen, bis die Bedingung falsch wird. Allerdings wird die Anweisung immer mindestens einmal ausgeführt, was bei der reinen `while`-Schleife nicht sein muss.

Schleifen werden oft zum Zählen verwendet. Hierfür gibt es eine besondere Schleife, `for`:

```
for (<Init>; <Bedingung>; <Update>) <Anweisung>
```

Dies ist eine Abkürzung für

```
<Init>;  
while (<Bedingung>) {  
    <Anweisung>  
    <Update>  
}
```

Eine Schleife, die die Variable `i` von 0 bis 99 zählt, und sie ausgibt, wäre

```
for (i=0; i<100; i++) print i;
```

Jeder der drei Ausdrücke in der Klammer kann weggelassen werden. Fehlt z.B. `<Init>`, so wird der momentane Wert der Variable benutzt.

✘ Mit `break` kann aus jeder Schleife – auch aus einer Endlosschleife – herausgesprungen werden in die

nächsthöhere. Mit `continue` wird der Durchlauf übersprungen, es geht bei `<Update>` weiter.

## 4.3 Funktionen

`awk` erlaubt die Definition eigener *Funktionen*, in anderen Sprachen auch *Subroutinen* oder *Prozeduren* genannt. Eine Funktion wird folgendermaßen definiert:

```
function <Name> (<Argumente>) {
    <Anweisungen>
}
```

Die Definition der Funktion muss außerhalb aller Blöcke in geschweiften Klammern stehen. Als Argumente können beliebig viele Variablen, durch Kommas getrennt, angegeben werden. Wenn es keine Argumente gibt, muss trotzdem ein leeres Klammerpaar angegeben werden:

```
function multiply(i,j) {
    print i "*" j, "=", i*j;
}
```

Der Aufruf der Funktion aus einem Anweisungsblock heraus wäre

```
multiply($4,$5);
```

zum Beispiel. Alle Variablen, die als Argumente übergeben werden, sind *lokal*. Das heißt, sie sind nur innerhalb des Anweisungsblocks der Funktion bekannt, nicht jedoch außerhalb.

Es können jedoch *globale Variablen*, die außerhalb der Funktion definiert wurden, benutzt werden. Wenn eine Variable innerhalb einer Funktion zum ersten Mal benutzt, also implizit initialisiert wird, ist sie eine globale Variable.

✘ Die Ausnahme ist hier, wenn es eine lokale Variable gibt, die denselben Namen wie eine globale Variable hat. In diesem Fall ist die lokale gemeint. Alle Variablen, die außerhalb von Funktionen definiert werden, sind global.

✘ Achtung! Die erste Klammer muss direkt an den Funktionsnamen anschließen, sonst hat `awk` Schwierigkeiten bei der Interpretation des Aufrufs!

Eine nützliche Sache ist die *Rückgabe von Werten* aus einer Funktion. Dies geschieht mit `return`:

```
function max(i,j) {
    if (i > j) return i;
    else return j;
}
```

Diese Funktion schreibt nichts auf den Bildschirm, sondern gibt nur einen Wert zurück. Der Aufruf

```
max(1,3)
```

bringt nichts, obwohl die Anweisungen der Funktion abgearbeitet werden. Um den Rückgabewert der Funktion nutzen zu können, muß der Aufruf lauten:

```
j = max(1,3)
```

Dann hat `j` den Wert des Maximums von 1 und 3, also 3. Man kann den Wert aber auch direkt in einem Aufruf verwenden:

```
print max(1,3)
```

schreibt eine 3. Man kann sogar soweit gehen, den Rückgabewert als Argument in einem weiteren Funktionsaufruf zu benutzen:

```
print max(max(1,3),8)
```

schreibt eine „8“.

Man kann die Funktion auch aus sich selbst heraus aufrufen, dies nennt man Rekursion:

```
function factorial(n) {
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

ist eine rekursive Funktion zum Berechnen von Fakultäten. Aber Vorsicht: Da die lokalen Variablen für jeden Aufruf extra angelegt werden müssen, kann es bei langen Rekursionen zum Überlauf des Stapelspeichers kommen.

## 4.4 Weitere Funktionen

Neben den bereits bekannten Stringoperationen gibt es folgende Funktionen:

**exit** *<Abbruchstatus>*

bricht das Skript sofort ab und gibt den Abbruchstatus an das System zurück. **exit** ohne Status gibt Null zurück, was soviel heißt wie „alles in Ordnung“.

**next**

überspringt alle weiteren Bedingungs-Anweisungsblöcke für die momentane Zeile und geht zur nächsten Zeile über. Das ist natürlich nur sinnvoll, wenn es mehrere Blöcke gibt oder wenn man alle anderen Anweisungen des jetzigen Blocks ignorieren möchte.

**close** *<filename>*)

Schließt die Datei mit dem Namen *<filename>* und beendet das Lesen von ihr. Mit **close** kann man auch Dateien schließen, die man durch Dateiumleitung

mit `>` geöffnet hat. Die ist manchmal notwendig, wenn man zu viele Dateien gleichzeitig geöffnet hat und das Limit erreicht ist.

**getline**  $\langle string \rangle < \langle filename \rangle$  Liest die nächste Zeile von  $\langle filename \rangle$  ein und schreibt sie auf die String-Variablen  $\langle string \rangle$ . Wenn die Angabe von  $\langle string \rangle$  fehlt, wird auf `$0` geschrieben, fehlt  $\langle filename \rangle$ , so wird von der momentan offenen Datei gelesen. **getline** gibt auch einen Wert zurück: 1 für erfolgreiches Einlesen der Zeile, 0 für Dateiende oder -1, wenn ein Fehler aufgetreten ist.

**system** ( $\langle cmd \rangle$ )

Führt das Systemkommando aus, das durch den String  $\langle cmd \rangle$  beschrieben wird. Ist  $\langle cmd \rangle$  z.B. `"ls -lrt"`, so wird der Inhalt des momentanen Verzeichnisses in langer Form, abwärts nach Datum sortiert, ausgegeben. Diese Anweisung ist natürlich dann besonders nützlich, wenn man anstelle eines fixen Strings eine Stringvariable einsetzt. Die Syntax des Kommandos richtet sich natürlich nach dem verwendeten Betriebssystem.

## 4.5 Felder

Felder sind Werte, die unter einem Namen zusammengefasst sind und mit einem Index versehen werden. Üblicherweise haben Felder ganzzahlige, aufeinanderfolgende Indizes. Daten in einem Feld werden mit dem gemeinsamen Feldnamen und ihrem Index in eckigen Klammern angesprochen:

$\langle Feld \rangle [ \langle Index \rangle ]$

Ein Feldeintrag ist wie eine Variable: Er kann mit anderen Werten verglichen werden und einen Wert zugewiesen bekommen. Felddaten sind zu Anfang Null oder der Nullstring. (Genau wie Variablen können Felddaten Zahlen oder Strings sein.)

Das besondere an Feldindizes in *awk* ist, dass sie nicht unbedingt aufeinanderfolgen und nicht numerisch sein müssen. Das macht das Arbeiten mit Feldern in *awk* besonders flexibel:

```

/^NODE/ {
    nodeID = substr ($0,9,8) + 0;
    x[nodeID] = substr($0,17,8) + 0;
    y[nodeID] = substr($0,25,8) + 0;
    z[nodeID] = substr($0,33,8) + 0;
}

```

Für Felder gibt es eine besondere Bedingung,

$( \langle Index \rangle \text{ in } \langle Feld \rangle )$

prüft, ob es in  $\langle Feld \rangle$  einen Eintrag unter  $\langle Index \rangle$  gibt. Diese Notation kann auch für **for**-Schleifen verwendet werden:

```
for (<Index> in <Feld>) <Anweisung>
```

arbeitet die Anweisung für alle Indizes im Feld ab. Das ist oft sehr praktisch:

```
 /^[A-Za-z]/ { a[$1]++ }
END { for (i in a) print a[i], "Mal", i; }
```

zählt zu Beispiel die Anzahl und Typen der verwendeten Keywords, wenn man davon ausgeht, dass Keywords mit Buchstaben anfangen.

Feldeinträge können mit

```
delete <Feld> [<Index>]
```

wieder aus dem Feld gelöscht werden. In einem Feld können auch mehrere Indizes verwendet werden, die dann durch Kommas getrennt innerhalb der eckigen Klammern stehen:

```
<Feld> [<Index1>, <Index2>, <Index3>]
```

### Aufgaben:

#### Aufgabe 4a

Wie sieht ein Skript aus, das alle Wörter einer Zeile in umgekehrter Reihenfolge ausgibt?  
Wie eines, das die ganze Zeile verkehrtherum schreibt?

#### Aufgabe 4b

Wie kann man vor Bearbeiten einer Datei eine Menge von Zahlen von einer externen Datei **id.dat** lesen und sie auf ein Feld schreiben? Die Datei **id.dat** enthält eine Zahl je Zeile und die gelesene ID soll Index des Felds sein, der Eintrag 0.

#### Aufgabe 4c

Schreibe ein Skript, das einen Datensatz kopiert, aber zwei Zeilen herausnimmt, wenn die erste mit dem Keyword „SOLID“ beginnt. (Eine Anwendung hierfür wäre *PAM-Crash*, bei dem die Definition eines Solid-Elements über zwei Zeilen geht.)

## 5 Für Fortgeschrittene

### 5.1 Reguläre Ausdrücke – die ganze Wahrheit

In Kapitel 2 haben wir bereits einige reguläre Ausdrücke kennengelernt. Dies sind die am häufigsten verwendeten, aber das Konzept des regulären Ausdrucks reicht noch viel weiter:

- Ein regulärer Ausdruck besteht aus einem oder mehreren „Zweigen“, die voneinander durch `|` abgetrennt sind. Der Ausdruck wird gefunden, wenn einer der Zweige gefunden wird (*Oder-Verknüpfung*):

`/groß|mittel|klein/` wird in „Der kleine Gernegroß“ gefunden. Der erste Treffer ist „groß“, obwohl „klein“ zuerst auftaucht: Links stehende Zweige haben ein stärkeres Gewicht. (Das ist wichtig, wenn die Position des ersten Treffers ausgegeben wird, wie z.B. in `match`.)

- Ein Zweig besteht aus mehreren „Stücken“, die einfach aneinandergereiht werden. Damit ein Zweig gefunden wird, müssen alle Stücke hintereinander im String gefunden werden.
- Ein Stück ist ein „Atom“, dem eines der Zeichen `*`, `+` oder `?` nachfolgen kann. Die Bedeutung der Zeichen wird gleich erklärt.
- Ein Atom ist ein regulärer Ausdruck in runden Klammern oder einer der Ausdrücke, die wir bereits kennen: Ein einzelner Buchstabe, ein Backslash mit einem nachfolgenden Buchstaben, die Wildcard `.`, ein Bereich in `[ ]`, der Zeilenanfang `^` oder das Zeilenende `$`.
- Ein Atom ohne nachgestelltes Zeichen findet exakt ein Vorkommen des Atoms: `/ein/` findet „einer“, „Bein“ und „Schweinebacke“.
- Ein Atom mit nachgestelltem `*` findet kein oder beliebig viele Vorkommnisse des Atoms. `/14*5/` findet „15“, „145“, „1445“, „14445“ usw.
- Ein Atom mit nachgestelltem `+` findet ein oder mehrere Vorkommnisse des Atoms: `/14+5/` findet „145“, „14445“, aber nicht „15“.
- Ein Atom mit nachgestelltem `?` findet das Atom selbst oder den leeren String, d.h. nichts: `/14?5/` findet „15“ und „145“, mehr nicht.
- Wenn es mehrere Treffer gibt, so wird der im Suchstring zuerst auftretende gewählt. Bei Ausdrücken mit `*`, `+` und `?` wird der Treffer gewählt, in dem die meisten Wiederholungen eines Atoms auftauchen (*longest match*).

Die Anwendung regulärer Ausdrücke ist oft hilfreich, kann aber auch zu Extremen wie

```
/-?[0-9]*(\.[0-9]*)?([eE][+-]?[0-9]+)?/
```

geführt werden. (Der Ausdruck findet übrigens eine Dezimalzahl in Exponentenschreibweise.) Für die meisten Anwendungen sind die Atome jedoch ausreichend, insbesondere dann, wenn man nicht die genaue Position des Auftretens eines regulären Ausdrucks benötigt.

## 5.2 Weitere Variablen in *awk*

Neben den bekannten  $\$x$ , **NR**, **NF** gibt es noch weitere Variablen in *awk*:

**FILENAME** ist der Name der momentan geöffneten Datei. Das kann hilfreich sein, wenn mehrere Dateien mit *awk* bearbeitet werden sollen. **FILENAME** ist in den Blöcken **BEGIN** und **END** nicht definiert.

**ARGC** (*argument count*) ist die Anzahl der aus der Shell übergebenen Argumente. Der Aufruf von *awk* zählt dabei als Argument. **ARGV** (*argument value*) ist ein Feld, das die übergebenen Argumente als String enthält. **ARGV[0]** ist "**awk**" (oder welche Version von *awk* benutzt wurde), **ARGV[1]** bis **ARGV[ARGC-1]** die übergebenen Dateien.

**ENVIRON** ist ein Feld, das die Umgebungsvariablen des Systems als Strings enthält. Der Index ist der Name der Umgebungsvariable als String: **ENVIRON["DISPLAY"]** enthält das momentane Display-Setting.

**FNR** (*file number of records*) ist die Zeilennummer in der momentanen Datei. **NR** wird dateiübergreifend gezählt.

**RS** (*record separator*) ist das Zeichen, das Zeilen voneinander trennt. Die Voreinstellung ist hier "**\n**", das *newline*-Zeichen, was bedeutet, dass Zeilen in *awk* auch normalen Zeilen entsprechen. Man könnte aber z.B. mit **RS = ";"** Zeilen auch durch Semikola trennen.

**FS** (*field separator*) ist der reguläre Ausdruck, der die einzelnen Wörter voneinander trennt. Die Voreinstellung ist, dass Tabulatoren und Leerzeichen die Wörter trennen. Achtung: Wird **FS** im Skript definiert, so werden Zeichen am Anfang und am Ende berücksichtigt. Fängt eine Zeile dann mit einem der Trennzeichen an, so ist das erste Wort der leere String, der als Wort zwischen Zeilenanfang und dem ersten Auftreten des Separators gewertet wird!

**ORS** (*output record separator*) ist das Zeichen, das nach jedem **print**-Befehl geschrieben wird, per Default ein Zeilenumbruch mit *newline* ("**\n**").

**OFS** (*output field separator*) ist das Zeichen, das zwischen zwei Ausgabeargumenten von **print** geschrieben wird, per Default ein Leerzeichen.

**OFMT** (*output format*) ist das Ausgabeformat für Zahlen, wenn kein anderes angegeben wird. Voreinstellung ist hier **"%.6g"**.

### **Aufgaben:**

#### **Aufgabe 5a**

Schreibe ein Skript, das ein HTML-Dokument lesbar macht: Alle Ausdrücke zwischen spitzen Klammern sollen gestrichen werden. (Gehen wir einmal davon aus, dass eine spitze Klammer immer auf derselben Zeile geschlossen wird.)

## Anhang A:

# Kurzübersicht

Angaben in eckigen Klammern müssen nicht unbedingt gemacht werden, Ausdrücke in spitzen Klammern sind Variablen.

Aufbau eines *awk*-Skripts:

```
{ function <Name>(<Arg>) {<Anweisungen>} }  
  
{ <Bedingung> } { {<Anweisungen>} }
```

Es können beliebig viele Funktionen und Bedingungs-Anweisungs-Paare angegeben werden, Sie werden durch Zeilenumbrüche oder Semikola voneinander getrennt.

Bedingungen:

```
/<reg. Ausdruck>/  
<Ausdruck> <Operator> <Ausdruck>  
<Ausdruck> <reg. Operator> <reg. Ausdruck>  
<Ausdruck> in <Feld>
```

Operatoren:

<b>==</b>	ist gleich
<b>!=</b>	ist nicht gleich
<b>&gt;</b>	größer als
<b>&gt;=</b>	größer als oder gleich
<b>&lt;</b>	kleiner als
<b>&lt;=</b>	kleiner als oder gleich

reguläre Operatoren:

<b>~</b>	enthält
<b>!~</b>	enthält nicht

Anweisungen:

```
<Anweisung>  
<Variable> <Zuweisung> <Ausdruck>  
{ <mehrere Anweisungen, durch ; oder Zeilenumbrüche getrennt> }  
if (<Bedingung>) <Anweisung> { else <Anweisung> }  
while (<Bedingung>) <Anweisung>  
do <Anweisung> while (<Bedingung>)  
for (<Init>; <Bedingung>; <Update>) <Anweisung>  
for (<Variable> in <Feld>)
```

Rechenoperatoren:

+	Addition
-	Subtraktion
*	Multiplikation
/	Dezimal-Division
%	Modulo-Division (Rest)
^	Potenz

Zuweisungsoperatoren:

=	ist gleich
++	erhöhe um eins
--	vermindere um eins
<i>&lt;op&gt;</i> =	führe Operation mit linker Seite aus ( <b>x += 5</b> entspricht <b>x = x + 5</b> )

Ausdrücke:

*<Zahl>* – Dezimalzahl  
*"<String>"* – Zeichenkette in Anführungszeichen  
*<Variable>* – Variable  
*<Feld>*[*<Index>*] – Feldeintrag  
*<Fkt>*(*<Arg>*) – Rückgabewert einer Funktion

*awk*-Funktionen:

```

print <Liste der Argumente>
printf "<Format>" { , <Argumente> }
getline { <Variable> } { < <Filename> }
close(<Filename>)
system(<cmd>)
exit { <Ausdruck> }
next
substr(<String>, <Anfang> { , <Länge> } )
length { (<String>) }
toupper(<String>)
tolower(<String>)
sub(<alt>, <neu> { , <string> } )
gsub(<alt>, <neu> { , <string> } )
index(<String>, <String>)
match(<reg. Ausdr.>, <String>)

```

## Anhang B:

# Lösungen

### 1a

```
{ print $2, $1 }
```

`print` kann mehrere Argumente haben, die mit Kommas getrennt werden. Bei der Ausgabe wird zwischen den Argumenten ein Leerzeichen geschrieben.

### 1b

```
{ print NR, $0 }
```

**2a** Es gibt zwei Möglichkeiten:

```
NR%2==0
```

für Zeilen mit geraden Nummern,

```
NR%2==1
```

für ungerade.

### 2b

```
{l++; w += NF; c += length + 1};  
END {print l, w, c};
```

Die 1 muss zur Länge addiert werden, da `wc` das Zeichen `\n` am Ende der Zeile mitberücksichtigt.

### 2c

```
$1=="mult" { n += $2*$3 }; END { print n }
```

### 2d

```
/^[ \t]/ { print " "; print }
```

### 2e

```
$1 != x { print; x = $1 }
```

**3a**

```
{ printf "%-8s%8d%8.2f%8.2f%8.2f", $1, $2, $3, $4, $5 }
```

**3b**

```
{
  x1 = substr ($0,1,8);
  x2 = substr ($0,9,8);
  x3 = substr ($0,17,8);
  x4 = substr ($0,25,8);
  x5 = substr ($0,33,8);
  print x1, x2, x3, x4, x5;
}
```

**3c**

```
$0 ~ "^[0-9]" {
  gsub("ä","Ä");
  gsub("ö","Ö");
  gsub("ü","Ü");
  gsub("ß","SS");
  print toupper ($0);
}
$0 !~ "^[0-9]"
```

**4a** Alle Wörter in umgekehrter Reihenfolge:

```
{ for (i=NF;i>0;i--) printf "%s " $i; print "" }
```

Die Zeile verkehrtherum:

```
{
  for (i=length;i>0;i--)
    printf "%s" substring ($0,i,1);
  print "";
}
```

Das abschließende `print ""` bewirkt nur einen Zeilenumbruch.

**4b**

```
BEGIN {
  while (getline < "id.dat" == 1) a[$1] = 0
}
```

**4c**

```
{ if ($1=="SOLID") getline; else print; }
```

Auch wenn es nicht so aussieht, werden hier tatsächlich zwei Zeilen nicht geschrieben: **getline** liest die nächste Zeile ein, mit der aber nichts gemacht wird. Erst die übernächste Zeile wird wieder auf „SOLID“ überprüft. Eine alternative Lösung wäre:

```
/^SOLID/ { getline; next; }  
{ print; }
```

**5a**

```
{ gsub (/\<[^>]*\>/, ""); print }
```