Kapitel 3

awk

3.1 Einführung

awk ist ein weiteres Tool, mit dem man Textfiles bearbeiten kann. Es bietet dabei weit über sed hinausgehende Möglichkeiten.
awk bietet, statt der simplen Manipulationskommandos von sed, eine eigene Programmiersprache, die stark an die Sprache **C** angelehnt ist.
awk spaltet die Eingabe in Records auf, diese wiederum in Felder. Dabei sind die Zeichen, die Records und Felder in der Eingabe trennen, einstellbar. Als Default werden die Zeilen der Eingabe als Records genommen und Felder in jeder Zeile durch Leerzeichen getrennt.
awk kann nun für jeden Record bestimmte Programmstücke ausführen, wobei der Inhalt und die Anzahl der Felder in dem aktuellen Record in speziellen Variablen zur Verfügung steht. Welches Programmstück für einen Record ausgeführt wird, kann man (ähnlich sed) durch vorgestellte Adressen auswählen.
Der Aufruf von awk geschieht ähnlich, wie bei sed: Entweder gibt man das komplette Programm auf der Kommandozeile an, oder man schreibt es in ein File und gibt den Namen des Files nach der **-f** Option mit.
Als Beispiel betrachten wird dieses File:

```
1 34.5
Versuch Nr 1
2 89.54
Versuch Nr 2
3 773.0
Versuch Nr 3: ...
4 333.11
5 993.11
```

Dies kann etwa die Nummern von Messungen, gefolgt von den Messwerten eines Experimentes darstellen. Unter diesen Zahlenpaaren folgt ggf. ein Kommentar. Wenn wir jetzt den Durchschnitt dieser Messwerte haben wollen, könnten wir folgendes awk Programm schreiben:

```
/^[0-9]/{ summe=summe+$2; anzahl++; }
END { durch=summe/anzahl; printf("%4.2f\n",durch); }
```

Dieses Programm zeigt schon den typischen Aufbau eines awk Programmes: Es besteht aus mehreren Adresse/Programmstück Paaren. Dabei muß das Programmstück immer in geschweiften Klammern eingeschlossen werden. In unserem Beispiel selektiert die Adresse des ersten Stückes alle Records, welche mit einer Ziffer beginnen. Im Programmstück wird dann einfach die Summe um das zweite Feld dieses Records erhöht. Die Felder eines Record sind in den zugehörigen Programmstücken mittels der speziellen Variablen '$\$i$' ansprechbar, wobei i die Nummer des Feldes ist. Als Besonderheit steht in `$0` der gesamte Record. Woher weiss awk aber nun, daß das zweite Feld eine Zahl darstellt? In awk gibt es eine Unterscheidung zwischen Strings und Zahlen eigentlich nicht. In Wirklichkeit steht in `$2` genau der Text des Feldes. Dieser wird bei Bedarf umgewandelt. Da wir hier eine Addition ausführen, wird versucht, `$2` in eine Zahl umzuwandeln, was in unserem Beispiel immer gelingt. Schlägt dieser Versuch aber fehl, weil z.B. in `$2` etwa 'foo' steht, wird nicht etwa ein Fehler ausgelöst, sondern einfach '0' als Ergebnis genommen.
In unserem Programm steht noch die merkwürdige Adresse 'END'. Diese, zusammen mit ihrem Gegenstück 'BEGIN', hat eine spezielle Rolle: Das zugehörige Programmstück wird *nach* der Verarbeitung des letzten Records, bevor awk beendet wird, ausgeführt. Dies ist besonders nützlich, um am Schluß Zusammenfassungen auszugeben. Im Unterschied dazu wird das Programmstück hinter einem 'BEGIN' *vor* dem Lesen des ersten Records ausgeführt und kann zu Initialisierungen genutzt werden. In unserem Fall berechnen wir am Schluß den Durchschnitt und geben ihn dann aus.
Unsere Eingabe enthält noch Zeilen, die nicht von einer Adresse selektiert werden, nämlich die drei Kommentare. awk gibt diese Zeilen nicht etwa einfach aus, sondern ignoriert sie schlichtweg.
Wichtig ist auch noch die Tatsache, daß man eine Adresse auch komplett weglassen kann. Dann wird das folgende Programmstück für jeden Record ausgeführt.
Gibt es mehrere passende Adress/Programmstück Paare, welche auf eine Zeile passen, dann werden sie in der Reihenfolge ihres Auftretens im Programm abgearbeitet.

3.1.1 Versionitis

Leider existieren eine ganze Reihe von Versionen von `awk`, welche sich doch stark unterscheiden. Wir werden hier nur GNU `awk` betrachten, das die meisten Features bietet und auch frei verfügbar ist.

3.2 Adressen & Pattern

In `awk` werden ebenfalls Adressen der Form, wie sie in `sed` auftauchen (siehe Abschnitt 2.2.2), verwandt. Zusätzlich kennt `awk` aber noch weitere Adressangaben, so daß wir nochmal alle zusammenfassen:

/pattern/ Ein `sed`-ähnliches Pattern, allerdings sind mehr Operatoren erlaubt (s.u.).

rel-expr Ein relationaler Ausdruck (s. Abschnitt 3.3), der zu einem Wahrheitswert auswertet. Das Programmstück wird ausgeführt, wenn der Warheitswert 'wahr' ($\neq$`0`) ergibt.

adr && adr Der `&&` Operator steht für die Und-Verknüpfung zweier Adressen. Nur wenn beide auf den aktuellen Record passen, wird dieser bearbeitet.

adr* || *adr Ähnlich wie `&&`, nur für die Oder-Verknüpfung.

(*adr*) Klammerung von Adressen.

!*adr* Negation der Adresse. Der aktuelle Record wird nur bearbeitet, wenn die Adresse nicht passt.

adr?adr:adr Diese Adresse ist ähnlich dem `?:` Operator von **C**. Wenn die erste Adresse auf den aktuellen Record passt, dann wird noch die zweite Adresse getestet, sonst die dritte.

adr,adr Ein Bereich von Adressen, mit der gleichen Semantik wie bei `sed`.

BEGIN Diese spezielle Adresse steht vor Programmstücken, dir vor dem Einlesen des ersten Records ausgeführt werden sollen.

END Diese Adresse steht vor Programmstücken, die nach der Bearbeitung des letzen Records ausgeführt werden sollen.

Die Pattern von `sed` werden nun von `awk` erweitert, es gibt die unten aufgeführten zusätzlichen Metazeichen.

+ Der + Operator steht für ein- oder mehrmaliges Auftreten des vor im stehenden Patterns. Z.B. passt `[0-9]+` auf mindestens eine Ziffer.

? Der `?` Operator steht für null- oder genau einmaliges Auftreten des vor ihm stehenden Patterns. Also passt `lines?` auf '`line`' oder '`lines`' und sonst nichts.

| Dieser Operator steht für die Alternative zwischen zwei Pattern. So passt `a|b` entweder auf '`a`' oder auf '`b`'.

() `awk` bietet, anders als `sed`, Klammerung von Pattern. So passt `a(bc)*` auf '`a`', '`abc`', '`abcbc`', ...

3.3 Variablen und Ausdrücke

3.3.1 Konstanten

Konstanten treten in `awk` in zwei Formen auf:

- Als Strings, eingeschlossen in '`"`', z.B. `"Ein String"`.
- Als numerische Konstanten, egal ob als Ganzzahl ('`65535`') oder Fließkommazahl ('`3.14159265`').

Es sei aber angemerkt, daß `awk` bei Bedarf konvertiert. So weist das Kommando `s="9"-1` der Variablen `s` die Zahl `8` zu!
In String Konstanten stehen die von **C** her gewohnten Escape-Sequenzen zur Verfügung, um nicht-druckbare Zeichen darzustellen, etwa '`\n`' oder '`\f`'.

3.3.2 Variablen

`awk` kennt drei Typen von Variablen:

- Vom Benutzer definierte Variablen. Diese werden einfach durch ihre Verwendung im Programm definiert und müssen nicht deklariert werden (z.B. `summe` im obigen Beispiel).
- Eingebaute Variablen. Diese werden von `awk` vordefiniert. Ihre Namen bestehen immer komplett aus Großbuchstaben, so daß eine Verwechslung mit anderen Variablen schwierig ist. Weiter unten sind alle vordefinierten Variablen aufgelistet.
- Felder. Diese enthalten die Felder des aktuellen Records. Sie werden mittels `$`*expr* angesprochen, wobei *expr* zu einer Zahl auswertet, die die Nummer des Feldes angibt (von `1` an). Das 'Feld' `$0` enthält den gesamten Record.

Vordefinierte Variablen

Folgende Variablen werden von `awk` vordefiniert:

ARGC Die Anzahl von Kommandozeilenparametern.

ARGV Ein Array mit den Kommandozeilenparametern.

ENVIRON Ein assoziatives Array, das die Umgebungsvariablen enthält.

FILENAME Der Filename des aktuellen Eingabefiles.

FNR Die Nummer des aktuellen Records, relativ zum aktuellen File.

FS Der Feldseperator, also das Zeichen, das Felder trennt. Default: Leerzeichen

IGNORECASE Bestimmt, ob das Matching ohne Unterscheidung von Groß-/Kleinschreibung stattfindet. Dies ist der Fall, wenn diese Variable einen Wert ungleich 0 enthält. Default: 0

NF Die Anzahl der Felder im aktuellen Record.

NR Die Nummer des aktuellen Records (Zählung beginnt mit 1).

OFMT Das Ausgabeformat für Zahlen. Default: `%.6g`

OFS Der Feldsperator für Ausgaben mittels des `print` Kommandos (siehe dort). Default: Leerzeichen

ORS Der Recordsperator für die Ausgabe mittels `print`. Default: Newline

RLENGTH Hier wird die Länge des Substrings abgelegt, der von der `match` Funktion gematcht wurde (siehe dort.)

RSTART Hier wird die Position des Beginns des Matches bei Aufruf der `match` Funktion abgelegt.

RS Der Recordseperator, also das Zeichen, das Records in der Eingabe trennt. Default: Newline

SUBSEP Das Zeichen, das Indices in Arrayindizierung trennt. Default: ASCII-28

3.3.3 Arrays

`awk` kennt neben skalaren Variablen auch Arrays. Im Unterschied zu **C** sind Arrays in `awk` allerdings *assoziativ*. Das heißt, der Index des Arrays ist nicht eine Zahl, sondern ein String, z.B. `foo["bar"]=99;`
Dies bedeutet, daß die Arrays dadurch realisiert sind, das eine Liste von Paaren (Schlüssel,Wert) abgelegt wird. Dabei ist der Schlüssel der Index. Man kann aber weiterhin Zahlen als Indices verwenden, da `awk` sie auch als Strings auffaßt.
`awk` kennt auch mehrdimensionale Arrays. Allerdings sind diese intern wiederum als eindimensionale Arrays realisiert. Dabei werden die Indices zu einem einzigen Index aneinandergehängt, getrennt durch das Zeichen, welches in der Variablen `SUBSEP` abgelegt ist. Z.B. weisen die beiden Ausdrücke `foo[1,2]=9` und `foo["1" SUBSEP "2"]=9` dem gleichen Element den Wert `9` zu[1].
Bei der Iteration über die Elemente eines Arrays benötigt man nun einen Weg, um alle Schlüssel aufzählen zu können, bzw. einen Test, ob ein bestimmter Schlüssel im Array bereits vorhanden ist. Dazu gibt es eine Sonderform der `for` Schleife (s.u.) und den `in` Operator.
`for (`*`var`*` in `*`array`*`)` iteriert über alle Paare in dem Array *array*, wobei der Wert des Schlüssels im Rumpf der Schleife in der Variablen *var* verfügbar ist. So gibt `for (j in foo) { print foo[j]; }` alle Werte des Arrays `foo` aus.
Der Ausdruck *`index`*` in `*`array`* prüft, ob der Schlüssel *index* in dem Array *array* vorhanden ist. So würde mit dem obigen Beispiel etwa `"bar" in foo` zu 'wahr' (`1`) auswerten.

3.3.4 Operatoren

Operatoren verknüpfen Teilausdrücke zu einem neuen Ausdruck. `awk` kennt die folgenden Operatoren (geordnet nach steigender Priorität):

[1]Man beachte, daß in `awk` String Konkatenation durch das Leerzeichen angegeben wird.

Operator	Beschreibung
`=`,`+=`, `-=`, `*=`, `/=`, `%=`, `^=`	Zuweisung
`?:`	**C** Conditional
`||`	Logisches Oder
`&&`	Logisches Und
`~`, `!~`	Matchen eines Patterns und Negation davon
`<`, `<=`, `>`, `>=`, `!=`, `==`	Relationale Operatoren
(Leerzeichen)	String Konkatenation
`+`, `-`	Addition, Subtraktion
`*`, `/`, `%`	Multiplikation, Division und Modulo
`+`, `-`, `!`	Unäres Plus, Minus und logische Negation
`^`	Exponentation
`++`, `--`	Inkrement und Dekrement (Postfix oder Prefix)
`$`	Feldzugriff

Darüber hinaus kann man Ausdrücke natürlich auch noch klammern, um die Auswertungsreihenfolge zu beeinflussen.
Da die meisten dieser Operatoren denen von **C** entsprechen, werden wir jetzt nur noch auf die `awk`-spezifischen Operatoren genauer eingehen.

Feldzugriff

Der Feldzugriff wurde schon im Abschnitt über Variablen beschrieben.

Exponentation

`awk` kennt, im Gegensatz zu **C**, einen primitiven Operator zum Exponentieren von Zahlen: *`expr^expr`*.
Z.B.: `s=2^4` weist `s` den Wert `16` zu.
Die Zuweisungvariante *`var^=expr`* steht als Abkürzung für *`var=var^expr`* bereit.

Konkatenation

In `awk` werden Strings aneinandergehängt, indem man ein Leerzeichen zwischen ihnen läßt. So ergibt z.B. `"foo" "bar"` den String `"foobar"`.
Da `awk` aber auch Zahlen intern als Strings auffasst, ergibt `s=9 99` das selbe Ergebnis, wie `s="999"` oder `s="9" "99"`

Match

Der Match Operator (`~`) dient zum Test, ob ein String ein Pattern matcht. Er kann in zwei Formen aufreten:

expr ~ /pattern/ Es wird geprüft, ob der Wert des Ausdruckes auf der linken Seite von dem Pattern der rechten Seite gematcht wird. Falls dies zutrifft, wird der Wert `1` (wahr) zurückgegeben, sonst `0` (falsch).

expr ~ expr In dieser Variante wird der rechte Ausdruck ausgewertet und das Ergebnis als Pattern aufgefasst, mit dem dann auf den Match getestet wird (s.o.). Dies erlaubt variable Pattern in Ausdrücken.

Der Operator `!~` testet dann auf das Nicht-Matchen, ist ansonsten aber analog.
Z.B.:

```
if ($1 ~ /^[0-9]+$/) {print "Zahl";}
p=NR*2 ". Satz"; if ($1 ~ p) {print "OK";}
```

3.3.5 Funktionsaufrufe

`awk` kennt eine ganze Reihe vordefinierter Funktionen. Darüber hinaus kann man sich eigene Funktionen definieren (siehe Abschnitt 3.5.1).
Ein Funktionsaufruf sieht in `awk` (erwartungsgemäß) so aus:

$$ret=function(arg_1,\ arg_2...)$$

Gibt die Funktion keinen Wert zurück, entfällt natürlich der *ret*= Teil.
In Abschnitt 3.5.2 sind die vordefinierten Funktionen aufgeführt.

3.4 Kommandos

Kommandos in `awk` ähneln stark denen in **C**. Im Unterschied zu **C** muß man allerdings nicht jedes Kommando durch ein Semikolon beenden, kann dies aber tun. Zum Trennen von Kommandos benutzt man entweder ein Semikolon oder einen Zeilenumbruch. Wie in **C** kann man mehrere Kommandos zu einem Block fassen, der dann wie ein einzelnes Kommando behandelt wird, indem man sie in geschweifte Klammern einschließt.
Zuerst ist anzumerken, daß ein Ausdruck auch ein Kommando ist. So ist etwa `9+9; 34*67;` ein vollkommen legales Programmstück. Natürlich wird der Wert eines Ausdruckes, der in dieser Form als Kommando benutzt wird nur berechnet und dann weggeworfen. Der einzige Zweck eines Ausdruckes besteht in den Seiteneffekten, die in ihm auftreten. Dies sind in `awk` nur die Zuweisungen, s.o.

3.4.1 Einfache Kommandos

`awk` kennt die folgenden einfachen Kommandos:

break Dies verläßt eine `for` oder `while` Schleife (s.u.)

continue Beginnt die nächste Iteration einer `for` oder `while` Schleife.

exit Beendet das komplette `awk`-Programm. Wenn vorhanden wird eine `END` Aktion noch ausgeführt.

next Liest den nächsten Record ein und beginnt die Abarbeitung des `awk`-Programmes wieder am Anfang. Die spezielle Variante `next file` springt zum Beginn des nächsten Eingabefiles und liest den ersten Record daraus.

print *expr*[,*expr*...] [*out-expr*] Dient zur Ausgabe von Werten. Die durch Kommata getrennten Ausdrücke werden ausgewertet und nacheinander, durch den String, der in der Variablen `OFS` steht, getrennt ausgegeben. Am Ende wird der String, der in der Variablen `ORS` steht, ausgegeben. Durch den *out-expr* Ausdruck kann man die Ausgabe umleiten. Ist dieser von der Form >*`expr`*, dann erfolgt die Ausgabe in das File, dessen Namen durch den Ausdruck *expr* gebeben ist[2]. Steht statt dessen etwas der Form >>*`expr`* wird an das entsprechende File angehängt. Als letztes kann man noch die Form |*`expr`* verwenden. Hier wird die Ausgabe in eine Pipe geleitet, deren anderes Ende an das Programm geht, welches durch den Ausdruck spezifiziert ist, etwa `| sort -u >nix`.

printf(*f-expr*[, *expr*, ...]) [*out-expr*] Die von **C** her bekannte *printf(3)* Funktion existiert auch in `awk`. Hierbei wird *f-expr* ausgewertet und der resultierende String als eine Formatschablone genommen. Für die Argumente in der Schablone werden dann die ausgewerteten Ausdrücke eingesetzt. Als Formatangaben sind dieselben verfügbar, wie in *printf(3)* beschrieben. Auch dieses Kommando kann einen Umleitung enthalten (*out-expr*), die der des `print` Kommandos entspricht.

return [*expr*] Kehrt aus einen Funktion zurück und gibt ggf. den Wert des angegebenen Ausdruckes zurück.

[2]Dieser Ausdruck muß ein String sein. Man kann also nicht >`foobar` schreiben, sondern muß >`"foobar"` verwenden.

3.4.2 Konditional & Schleifen

Neben den einfachen Kommandos des vorherigen Abschnittes kennt `awk` auch noch ein `if-then-else` Konditional und vier Schleifenkonstrukte: zwei Varianten einer `for`-Schleife und zwei Varianten der `while`-Schleife:

if (*expr*) *stat* Der Ausdruck *expr* wird ausgewertet, wenn sein Wert `1` ergibt, wird das Kommando *stat* ausgeführt.

if (*expr*) $stat_1$ else $stat_2$ Der Ausdruck *expr* wird ausgewertet. Ergibt dies `1`, wird das Kommando $stat_1$ ausgeführt, sonst das Kommando $stat_2$.

do *stat* while (*expr*) Die Anweisung *stat* wird ausgeführt. Danach wird der Ausdruck *expr* ausgewertet. Ist das Ergebnis `1`, dann beginnt die Schleife von vorne. Sonst wird die Schleife verlassen.

while (*expr*) *stat* Der Ausdruck *expr* wird ausgewertet. Ist das Ergbnis `1`, dann wird die Anweisung *stat* ausgeführt und die Schleife beginnt danach von vorne. Andernfalls wird die Schleife sofort verlassen.

for (*i-expr*;*c-expr*;*l-expr*) *stat* Zuerst wird der Ausdruck *i-expr* ausgewertet. Danach wird der Ausdruck *c-expr* ausgewertet. Ist dessen Wert `0`, wird die Schleife verlassen. Anderenfalls wird das Kommando *stat* ausgeführt und danach der Ausdruck *l-expr* ausgewertet und der nächste Durchlauf der Schleife beginnt wieder mit der Auswertung des Ausdruckes *c-expr*.

Der Ausdruck *i-expr* dient zur Initialisierung der Schleife, etwa eines Schleifenzählers. Der Ausdruck *c-expr* ist die Schleifenbedingung. Der Ausdruck *l-expr* dient zum Weiterschalten von Schleifenzählern. Z.B.:

```
for (i=1;i<=NF;i++)
  printf("Feld %d hat den Wert %s\n",i,$i);
```

Diese Schleife gibt alle Felder des aktuellen Records aus.

for (*var* in *array*) *stat* Diese Variante dient zur Aufzählung aller Schlüssel des assoziativen Arrays *array*. Im Schleifenrumpf (*stat*) steht der jeweilige Schlüssel in der Variablen *var* zur Verfügung. Siehe Abschnitt 3.3.3.

In Schleifen können die Anweisungen `break` und `continue` auftreten. `break` bewirkt dabei, daß die innerste Schleife (in welcher sie steht), sofort verlassen wird. Die Programmausführung wird mit dem Kommando nach dem Schleifenrumpf fortgesetzt.
Das `continue` Kommando beginnt sofort die nächste Schleifeniteration, springt also in einer `while`-Schleife zur Auswertung der Schleifenbedingung. In einer

`for`-Schleife wird zuerst noch der *l-expr* Ausdruck ausgewertet, dann zur Auswertung der Schleifenbedingung gesprungen.

3.5 Funktionen

3.5.1 Benutzerdefinierte Funktionen

In `awk` kann man eigene Funktionen definieren. Diese werden im Programm an derselben Stelle definiert, wo sonst die Adressen/Programmstück Paare stehen (also nicht innerhalb eines Programmstückes). Die Form der Definition:

`function` *name*`(`$param_1$`,...,`$param_N$`) {` *stats* `}`

Als Beispiel folgendes kleines Programm:

```
function foo(bar, baz) {
  printf("Der %d. Parameter ist %s\n",bar,baz);
}
BEGIN { for (i=0;i<ARGC;i++) foo(i,ARGV[i]);}
```

Dieses Programm definiert eine Funktion `foo`, die einfach etwas Text mittels `printf` ausgibt und ihre Argumente an zwei Stellen einsetzt. Als Resultat dieses Programmes werden alle Kommandozeilenparameter von `awk` ausgegeben.
Innerhalb einer Funktion kann man mittels des `return` Kommandos Werte zurückgeben.

3.5.2 Eingebaute Funktionen

`awk` verfügt über eine ganze Reihe von eingebauten Funktionen, die wir hier alphabetisch geordnet auflisten:

atan2(x**,**y**)** Gibt den Wert des arcus tangens von y/x (in Radianten) zurück.

close(*f-expr*) Wir hatten bei dem `print` Kommando gesehen, daß man Ausgaben in Files umlenken kann. `awk` erlaubt aber nur (etwa) zehn offene Files. Um Files explizit wieder zu schliessen, ruft man die `close` Funktion mit dem Filenamen des geöffneten Files auf. Hatte man die Ausgabe in eine Pipe zu einem Programm geleitet, muß der Ausdruck zu `"|`*Programm*`"` auswerten, um diese Pipe zu schliessen.

cos(x**)** Gibt den Wert des Cosinus von x (in Radianten) zurück.

delete *array*[*element*] Diese Funktion löscht das angegebene Element aus dem Array. Nützlich, um Kommandozeilenparameter zu elimieren, siehe Abschnitt 3.6. Beispiele: `delete foo[1],delete ARGV[i]`

exp(x) Gibt den Wert von e^x zurück.

gsub(r,s,t) Diese Funktion ersetzt alle Vorkommen des Patterns r durch den String s. Die Ersetzung findet in dem String t statt. Das Pattern kann entweder als `/`*`pattern`*`/` oder als Variablennamen gegeben sein. In letzterm Fall wird der Inhalt der Variablen als Pattern benutzt. Man kann den String t weglassen, in welchem Fall `$0`, also der aktuelle Record, benutzt wird. Das Ergebnis dieser Funktion ist *nicht* das Resultat der Ersetzung, sondern die Anzahl Ersetzungen, die gemacht wurden. Das Resultat der Ersetzung steht in t selbst.

index(s, t) Liefert die Position des Substrings t in dem String s. Die Zählung beginnt dabei mit der `1`: `index("laber","aber")`=2.

int(x) Wandelt die Zahl x zu einer Ganzzahl um, indem die Nachkommastellen abgeschnitten werden.

length(s) Gibt die Länge des Strings s zurück.

log(x) Errechnet ln x.

match(s,r) Diese Funktion versucht, das Pattern r (in der gleichen Form, wie bei der `gsub` Funktion) in dem String s zu matchen. Der Rückgabewert ist `0`, falls kein Match existiert, oder die Position des Beginns des Matches. Diese Position wird auch in der Variablen `RSTART` abgelegt. Die Länge des Matches wird in der Variablen `RLENGTH` abgelegt. So liefert also z.B. `match("aber 100 Tiere",/[0-9]+/)` den Wert `6`. In `RLENGTH` wird der Wert `3` abgelegt.

rand() Diese Funktion liefert eine Pseudo-Zufallszahl im Bereich $[0, 1]$ zurück. Man erhält bei jedem Programmaufruf immer die gleiche Folge von Zahlen, deshalb sollte man die `srand` Funktion zum Setzen eines anderen Startwertes benutzen (s.u.).

sin(x) Liefert den Sinus von x (in Radianten) zurück.

split(s,***array***,***sep***) Diese Funktion spaltet einen String in Felder auf, die durch die im String *sep* stehende Zeichenfolge getrennt sind. Die einzelnen Felder werden der Reihe nach in dem Array *array* abgelegt. Fehlt der String *sep*, wird der Inhalt der Variablen `FS` benutzt. Die Funktion gibt die Anzahl Felder zurück, die in *array* abgelegt wurden. Z.B. liefert `split("a::b::c::d",foo,"::")` den Wert `4` zurück. `foo` enthält nachher die Elemente `foo[1]="a"`, `foo[2]="b"`, etc.

sprintf(*f-expr*[,*expr*...] Diese Funktion ähnelt dem `printf` Kommando, nur gibt sie ihr Ergebnis nicht aus, sondern liefert es als String zurück.

sqrt(x) Diese Funktion gibt die Quadratwurzel von x zurück.

srand(*expr*) Diese Funktion setzt den Startwert des Zufallszahlengenerators auf den Wert von *expr*.

sub(r,s,t) Diese Funktion ähnelt der Funktion `gsub` mit dem einzigen Unterschied, daß nur das erste Auftreten des Patterns ersetzt wird.

substr(s,b,l) Diese Funktion liefert den Teilstring des Strings s zurück, der an Position b startet und l Zeichen lang ist. Die Position zählt dabei von `1` an. Wird l weggelassen, wird der gesamte Rest des Strings von Position b an zurückgegeben.

system(*expr*) Führt das in dem String *expr* angegebene Programm in einer Shell aus. Die Funktion gibt den Exit-Code des Programmes zurück, oder `-1`, falls bei der Ausführung Fehler auftraten. Die Ausgaben des Programmes landen in `/dev/null` und können nicht im `awk`-Programm benutzt werden.

tolower(s) Diese Funktion wandelt alle Großbuchstaben in dem String s in Kleinbuchstaben und gibt dann den resultierenden String zurück.

toupper(s) Diese Funktion wandelt alle Kleinbuchstaben in s in Großbuchstaben und gibt das Ergebnis zurück.

3.6 Sonstiges

3.6.1 Kommandozeilenparameter

`awk` kennt, ähnlich `sed`, die **-f** Option, mit der man ein Programm zur Ausführung angeben kann. Darüberhinaus gibt es aber noch einige Optionen:

-Fc Setzt das Trennzeichen für Felder (`FS`) auf das Zeichen c.

-f *Prog* Nimmt das Programm im File *Prog* zur Ausführung.

var=value Weist der Variablen *var* den Wert *value* zu. Die Zuweisung erfolgt vor der Abarbeitung des ersten Kommandos im Programm (also auch vor einem `BEGIN` Stück).

Der erste Kommandozeilenparameter, der keine Option ist, wird als Programm aufgefasst (nur wenn keine **-f** Option gegeben wurde). Alle weiteren Parameter werden als die Namen von Files aufgefasst, welche bearbeitet werden sollen.
Man hat hier aber ein Problem: `awk` bietet das assoziative Array `ARGV`, in welchem die Kommandozeilenparameter (ausser den oben genannten), gespeichert sind. Manchmal will man noch zusätzliche Optionen für das `awk`-Programm mitgeben, die nur von diesem interpretiert werden sollen. Dabei dürfen diese aber nicht als die Namen von Files aufgefasst werden, welche zur Bearbeitung geöffnet werden sollen. `awk` fasst daher den Inhalt des `ARGV` Arrays *nach* Abarbeitung des `BEGIN` Programmstücks als Namen der Eingabefiles auf. Will man eigene Parameter benutzen, muß man diese in dem `BEGIN` Stück aus `ARGV` entfernen, z.B.:

```
BEGIN{ for (i=1;i<ARGC;i++) {
           if (ARGV[i] ~ /--ignore/) {
               iflag=1;
               delete ARGV[i];
           }
       }
}
```

Dieses Programmstück sucht nach der Option `--ignore` und setzt, falls sie gegeben wurde, die Variable `iflag` auf `1` und löscht dann diese Option aus den Kommandozeilenparametern.

3.6.2 Spezielles

Wie oben erwähnt, kann man die Variable `RS` auf einen String setzen, der Records in der Eingabe trennt. Setzt man diese Variable auf den leeren String, `RS=""`, dann steht dies für eine Leerzeile als Trennung zwischen Records. So kann man z.B. unter Zuhilfenahme von `FS="\n"`, einen ganzen Absatz in einen Record lesen, wobei die einzelnen Felder jetzt aus den Zeilen des Absatzes bestehen.
`awk` kennt eine Funktion, `getline`, mit der man den nächsten Record einlesen und im Programm weiter fortfahren kann. Diese kommt in zwei Varianten:

getline [***var***] [<***file***] Dies nächste Zeile wird aus dem File *file* gelesen (oder dem aktuellen Eingabefile) und in Felder aufgespalten. Das Ergebnis wird in `$`i und `$0` abgelegt, `NR`, `NF` und `FNR` werden gesetzt. Wird eine Variable *var* angegeben, wird der neue Record nicht in `$0` abgelegt, sondern in dieser Variablen. Diese Funktion liefert `0` als Ergebnis, wenn kein neuer Record mehr in dem File steht, `1`, wenn ein Record gelesen werden konnte und sonst `-1`.

***command* | getline** [*var*] Diese Variante liest nicht aus einem File, sondern führt das angegebene Programm *command* aus und liest dessen Ausgabe ein. Ansonsten identisch zur ersten Version.

Es sei angemerkt, daß das angegebene File nur bei der ersten Ausführung der Funktion geöffnet wird. Weitere Ausführungen der Funktion (mit demselben File natürlich), benutzen das bereits geöffnete File und lesen die nächsten Records.
Ebenso wird das Programm bei der zweiten Variante nur bei der ersten Ausführung gestartet und die Ausgabe dann bei den folgenden Aufrufen Recordweise gelesen.

3.7 Beispiele

3.7.1 Filelängen aufsummieren

Angenommen, man möchte in einem Directory die Längen bestimmter Files aufsummieren, etwa die aller Java-Programme. Dazu kann man folgendes in die Shell tippen[3]:

```
ls -l `find . -name '*.java'`|
  awk '{summe+=$5}
       END{ printf("Summe: %d bytes\n",summe);}'
```

Das `ls` Kommando listet alle Files auf, deren Namen mit `.java` enden (die Namen werden vom `find` Kommando geliefert). Im fünften Feld steht dabei in der Ausgabe die Filelänge. Das `awk` Programm summiert nun in jeder Zeile diese Länge auf und gibt am Schluß die Summe aus.

3.7.2 Sortieren

Ein Weg, Files (auch über Unterdirectorygrenzen hinweg) nach ihrer Länge sortiert auszugeben bietet das folgende Konstrukt:

```
ls -R -l .| awk '{printf("%08d %s\n",$5,$0)}'|
 sort|
 awk '{for (i=2;i<=NF;i++)
        printf("%s%s",(i==2)?"":" ",$i);
       printf("\n");}'
```

[3]Hier nur der Lesbarkeit wegen auf mehrer Zeilen verteilt.

Diese Konstruktion listet erst alle Files mittels `ls`, danach wird über ein `awk`-Programm die Filelänge vor jeden Eintrag geschrieben, und zwar mit führenden Nullen aufgefüllt. Danach wird diese Ausgabe mittels `sort` sortiert, das jetzt nach dem ersten Feld sortiert, der normalisierten Filelänge. Danach wird in einem zweiten `awk`-Programm die führende Filelänge wieder entfernt, indem einfach nur die Felder ab dem zweiten Feld ausgegeben werden.
Etwas unschön ist, das die Namen der Files in der Ausgabe nicht mit vollem Pfad stehen, sondern nur mit einfachem Namen. Das Auffinden gesuchter Files wird dadurch schwerer. Abhilfe schafft, statt des ersten `ls` Kommandos das folgende einzugeben: `ls -l 'find . -type f'`.

3.7.3 Statistik

In der Organisation eines Workshops reichten die Gutachter ihre Bewertungen über die Paper elektronisch ein. Dabei wurden, neben den textuellen Reviews, u.a. Benotungen über die allgemeine Qualität (`Overall`) oder die technische Qualität des Papers (`Technical`) abgegeben (Noten von 0-9). Für jedes Paper wurden diese Gutachten nun in einem File gesammelt. Zum Schluß sollten zur Auswertung und Einstufung der Paper diese Benotungen ausgewertet werden: Sowohl der Durchschnitt, als auch die Standardabweichung sollten für jedes Paper ausgegeben werden. Dabei waren die Datenfiles so ausgelegt, daß die Bewertungen in Zeilen der Form `%%Overall:` n_o bzw. `%%Technical:` n_t standen.
Dazu wurden ein Shell-Script und ein `awk`-Programm verwandt:

```
#! /bin/sh

echo Auswertung der Paper
for p in Papers/*; do
 echo Paper  'basename $p'
 awk -f auswert.awk $p
 echo
done

# auswert.awk
/^%%Overall:/{overall+=$2; anzahl++; ov[anzahl]=$2;}
/^%%Technical:/{technical+=$2; te[anzahl]=$2;}
END{ durcho=overall/anzahl; durcht=technical/anzahl;
     for (i=1;i<=anzahl;i++) {
        abweicho+=(ov[i]-durcho)^2;
        abweicht+=(te[i]-durcht)^2;
     }
```

```
    abweicho=sqrt(abweicho);
    abweicht=sqrt(abweicht);

    printf("Durchschnitte:\n");
    printf("\tOverall: %2.2f\n\tTechnical: %2.2f\n",
           durcho,durcht);
    printf("StdAbweichungen:\n");
    printf("\tOverall: %2.2f\n\tTechnical: %2.2f\n",
           abweicho, abweicht);
}
```

Das Shell-Script iteriert einfach über die einzelnen Datenfiles und ruft für jedes das `awk`-Programm auf. Das `awk`-Programm summiert in jeder Zeile, die eine Bewertung enthält diese auf (die Anzahl der Bewertungen wird nur bei der `Overall` Bewertung erhöht, die immer zuerst kommt). Am Ende werden dann der Durchschnitt und die Standardabweichung berechnet und ausgegeben.

3.8 Falltüren

In `awk` gibt es meiner Erfahrung nach weniger Falltüren, als in `sed`. Natürlich kann auch hier das Matching von Pattern verblüffen (siehe Beschreibung bei `sed`).
Eine immer mal wieder auftauchende Tretmine ist die Tatsache, daß ein adressloses Programmstück immer ausgeführt wird, auch wenn eine Zeile vorher schon von anderen Adressen gematcht und in deren Programmstücken bearbeitet wurde. Abhilfe schafft hier die Verwendung des `next` Befehls am Ende der spezielleren Programmstücke.
Die automatische Konvertierung von Strings zu Zahlen und zurück kann gewöhnungsbedürftig sein. Auch die assoziativen Arrays sorgen am Anfang für einige Überraschungen, insbesondere in Verbindung mit der automatischen String/Zahl-Konvertierung.
Man sollte aufpassen, sich bei Variablennamen nicht zu vertippen. Dies hat nämlich keine Fehlermeldung zur Folge, sondern es wird eine neue Variable angelegt. Natürlich ist der Effekt dann nicht ganz der gewünschte.
Eine extrem lästige Falltür ist die sich um `awk` rankende Versionitis. Es gibt mindestens drei verschiedene Versionen von `awk`, die sich alle in den unterstützen Features unterscheiden (`awk`, `nawk` und GNU `awk`). Allgemein kann man sagen, daß GNU `awk` die meisten Features bietet und dazu noch frei verfügbar ist. Deshalb haben wir uns hier nur auf diese Version beschränkt. Sollte man auf andere

Versionen von `awk` treffen, sollte man immer daran denken, daß sie vielleicht nur einen Teil der hier erwähnten Features bieten.

3.9 Literatur

Auf für `awk` empfiehlt sich das schon im `sed` Kapitel erwähnte Buch [Dou90]. Ebenso sollte man sich die Manual-Pages von `awk` zu Gemüte führen.