Die Programmiersprache Awk
Einführung, Tips und Tricks

Version 1.32 — 8.1.2003

Schutzgebühr: 10 Euro

(inkl. Diskette mit 100 Beispiel-Programmen)

Hinweise auf inhaltliche Fehler, Schreibfehler und unklare Formulierungen sowie Ergänzungen, Kommentare, Wünsche und Fragen können Sie gerne an den Autor richten:

OSTC Open Source Training and Consulting GmbH
Thomas Birnthaler
eMail: tb@ostc.de
Web: www.ostc.de

Inhaltsverzeichnis

1 Einführung

1.1 Übersicht

Anwender verwenden oft viel Zeit für vergleichsweise einfache, mechanische Datenbearbeitungen:

- Änderungen am Format von Daten
- Gültigkeitsüberprüfungen
- Selektieren von Datensätzen mit bestimmten Eigenschaften
- Summieren von Zahlen
- Zusammenstellen von Berichten
- ...

Diese Tätigkeiten sollten eigentlich dem Rechner überlassen werden. Oft ist es aber zu umständlich, dafür jedesmal ein eigenes Programm in einer Standard-Programmiersprache wie *C* oder *Pascal* zu schreiben, wenn eine solche Aufgabe auftaucht.

Awk ist eine **Programmiersprache**, in der sich solche Aufgaben durch sehr kleine Programme — oft nur ein oder zwei Zeilen lang — erledigen lassen. Ein *Awk*-Programm besteht aus einer Reihe von **Regeln** (**Muster-Aktion Paaren**). *Awk* durchsucht eine oder mehrere Dateien nach Zeilen, die zu einem der **Muster** passen; wird eine passende Zeile gefunden, so wird die zugehörige **Aktion** ausgeführt.

Ein Muster kann aus einer beliebigen Kombination von **Regulären Ausdrücken** und Vergleichsoperationen auf Zahlen, Zeichenketten, Feldern, Variablen und Arrayelementen bestehen. Anschließend kann die zugehörige Aktion die ausgewählten Zeilen beliebig verarbeiten. Die Sprache, in der die Aktionen beschrieben werden, ähnelt *C*, es gibt allerdings **keine Deklaration** von Variablen und neben Zahlen sind **Zeichenketten** ein fest eingebauter Datentyp.

Awk liest automatisch alle Eingabedateien zeilenweise ein und zerlegt jede Eingabezeile automatisch in einzelne Wörter (**Felder**). Gerade weil vieles **automatisch** geschieht — Einlesen der Daten, Zerlegen in Felder, Speicherverwaltung, Variableninitialisierung — sind *Awk*-Programme in der Regel viel kleiner als das gleiche Programm geschrieben in einer konventionellen Programmiersprache. *Awk* ist daher ideal für die oben aufgezählten Arten der Datenverarbeitung geeignet. Ein oder zwei Zeilen lange Programme können direkt auf der Kommandozeile über die Tastatur eingegeben, sofort verwendet und anschließend wieder verworfen werden. Im wesentlichen ist *Awk* ein allgemein verwendbares Programmierwerkzeug, das viele andere Hilfsprogramme (wie z.B. *Sed*, *Grep*, *Tr*, *Expr*) ersetzen kann, allerdings aber auch langsamer als diese (spezialisieren) Werkzeuge ist.

Eine Weiterentwicklung des *Awk* stellt die Skript-Sprache *Perl* (*practical extraction and reporting language*) dar. Sie ist eine Zusammenfassung der UNIX-Programme *Sh*, *Awk*, *Sed*, *Grep*, *Tr*, *Sort*, *Uniq* und bietet zusätzlich noch extrem ausdrucksstarke Reguläre

Ausdrücke, C-Bibliotheks-Funktionen, UNIX-Systemaufrufe, Objektorientierung, Modularisierung, Sicherheitsaspekte, usw. Diese Programmiersprache ist daher sehr mächtig und sehr umfangreich und ersetzt in zunehmenden Maße die klassischen Skript-Sprachen *Sh*, *Awk*, *Sed* und sogar *C*. Warum sollte man dann überhaupt noch *Awk* erlernen?

- *Awk* ist einfach und seine Syntax ist übersichtlich (etwa 45 Seiten Spezifikation), *Perl* ist aufgrund seines Umfanges und der ungewöhnlichen Syntax schwerer zu erlernen (etwa 450 Seiten Spezifikation).
- Bei etwas *C*-Kenntnissen weiß man bereits sehr viel, um *Awk* zu beherrschen.
- *Awk* ist klein und überall verfügbar.
- Für die meisten Anwendungen reicht die Funktionalität von *Awk* vollständig aus, viele Konzepte von *Perl* sind auch dort bereits verfügbar.
- Vieles aus dem Awk ist auf andere Programmiersprachen wie *Perl*, *PHP*, *Python*, *Ruby*,... übertragbar, da er ihr Vorläufer war.

1.2 Entstehungsgeschichte

Der Name *Awk* steht für die Initialen seiner drei Programmautoren **A**ho, **W**einberger und **K**ernighan (*awkward* heißt „ungeschickt“ auf Englisch, hier schimmert die Ironie der Autoren durch, was den Entwurf der Sprache betrifft).

Die erste Version von *Awk* wurde 1977 entworfen und realisiert, teilweise als ein Experiment um zu überprüfen, wie die UNIX-Werkzeuge *Sed* und *Grep* auf die gleichzeitige Verarbeitung von Zahlen und Zeichenketten verallgemeinert werden können. Sie basierte auf der Begeisterung der Autoren für Reguläre Ausdrücke und programmierbare Editoren.

Obwohl nur zum Schreiben von kleinen Programmen gedacht, führten seine Fähigkeiten bald dazu, daß begeisterte Anwender bedeutend längere Programme verfaßten. Diese längeren Programme benötigten Eigenschaften, die nicht Teil der ersten Version waren, daher wurde *Awk* in einer zweiten Version erweitert, die 1985 verfügbar wurde und häufig als *Nawk* (*New Awk*) bezeichnet wird. Ein grundsätzlich neues Merkmal dieser Version war die **Definition eigener Funktionen** durch den Anwender. Weitere Erweiterungen waren dynamische Reguläre Ausdrücke in Funktionen zur Textersetzung und zum Mustervergleich, neue vordefinierte Funktionen und Variablen, einige neue Operatoren und Anweisungen, Verarbeitung mehrerer Eingabedateien gleichzeitig sowie Zugriff auf die Kommandozeilenparameter. Die Fehlermeldungen wurden ebenfalls verbessert.

1.3 Die verschiedenen Versionen

Unter dem Namen *Awk* werden in diesem Skript verschiedene Varianten zusammengefaßt, die unterschiedlich mächtig sind und mehr oder weniger unterschiedliche Verhaltensweisen besitzen (die sich teilweise sehr subtil unterscheiden und auch „**dunkle Ecken**“ (dark

corners) genannt werden). Auf jedem UNIX-System ist zumindestens der Ur-*Awk* *Oawk* vorhanden, der *Nawk* sollte ebenfalls überall vorhanden sein. Der *Gawk* als freie Referenzimplementation ist letztlich auf jedem System installierbar:

- *Awk*: Ursprüngliche Version (1977).
- *Nawk*: Erweiterte Version (*New Awk*: Funktionen, `getline`, ... 1985).
- *Oawk*: *Old Awk* im Gegensatz zu *New Awk*.
- *Gawk*: Freie GNU-Version von *Nawk* mit Erweiterungen (kaum mehr Beschränkungen und deutlich schneller).
- *Mawk*: Freier *Awk* von Michael Brennan.
- *MKS-Awk*: Teil des *MKS-Toolkit* von Mortice Kern Systems.
- *Tawk*: Thompson-Version von *Nawk* mit sehr vielen Erweiterungen + Compiler.
- *Awka*: Freier *Awk*-Compiler.

Hinweis: In dieser Beschreibung wird einheitlich der Name *Awk* verwendet. Beschrieben werden aber die erweiterten Versionen *Nawk* und *Gawk* (die Erweiterungen des *Gawk* sind jeweils gekennzeichnet). Beim Ausprobieren der Beispiele bitte darauf achten, daß statt `awk` immer `nawk` oder `gawk` auf der Kommandozeile einzugeben ist (oder ein entsprechender Link von `awk` auf `nawk`/`gawk` gesetzt ist).

1.4 Einsatzgebiete

Die *Awk*-Sprache ist sehr gut zur **Beschreibung von Algorithmen** geeignet. Da es keine Variablendeklaration gibt und die Speicherverwaltung automatisch erfolgt ist, hat ein *Awk*-Programm viel Ähnlichkeit mit **Pseudocode**, mit dem Unterschied, daß *Awk*-Programme — im Gegensatz zu Pseudocode — sofort lauffähig sind.

Die übersichtliche Sprache und die einfache Anwendung lassen *Awk* auch für den **Entwurf größerer Programme** geeignet erscheinen. Man beginnt mit einigen Programmzeilen, verfeinert dann das Programm, bis es die gewünschte Aufgabe erfüllt und kann dabei leicht alternative Entwürfe ausprobieren. Da die Programme klein sind, kommt man schnell zu Ergebnissen und kann ebenso schnell wieder von vorne anfangen, wenn die beim ersten Versuch gemachten Erfahrungen einen „weiteren“ Lösungsweg aufgezeigt haben. Außerdem ist es sehr einfach, ein vollständig entwickeltes und korrektes *Awk*-Programm nachträglich in eine andere Programmiersprache umzusetzen.

Da *Awk* unter UNIX entwickelt wurde, beruhen einige seiner Fähigkeiten auf Eigenschaften, die normalerweise nur unter diesem Betriebssystem verfügbar sind. Trotz dieser Einschränkung sollte *Awk* jedoch in jeder Betriebssystemumgebung verwendbar sein; insbesondere ist er auch unter Windows/MS-DOS lauffähig.

Awk ist sicher nicht perfekt; er besitzt aufgrund seiner Entstehungsgeschichte einige Widersprüche und Mängel, er beruht auf einigen äußerst schlechten Grundideen und ist außerdem

manchmal sehr langsam. Gleichzeitig ist er jedoch eine ausdrucksstarke und vielseitige Programmiersprache, die in einer bemerkenswerten Reihe von Fällen einsetzbar ist:

- Suchen und Ersetzen von Texten.
- Datenextraktion, -reduktion, -aufbereitung, -präsentation.
- Konverter für ASCII-Daten.
- Ersatz für *C* bei Schnellschüssen oder Einmal-Entwicklungen.
- **Prototyping** von Programmen bis diese fehlerfrei sind und die Anwenderanforderung−en erfüllen, dann Re-Implementierung in *C* (*falls dies überhaupt noch notwendig sein sollte bzw. überhaupt noch Zeit dazu ist*).
- Pretty-Printing (z.B. Formatieren der Ergebnisse von SQL- oder UNIX-Kommandos).
- Testdaten erzeugen.
- Programmgeneratoren.
- Compiler/Interpreter für kleinere Sprachen.

1.5 Eigenschaften

- **Interpreter**, kein Compiler, d.h. schnelle Entwicklung aber (relativ) langsame Abarbeitung (deutlich schneller als *Shell*-Skripte, aber wesentlich langsamer als *C*-Programme).
- *C*-**ähnliche Syntax** („wer *C* kann, kann auch *Awk*“).
- **Stream-orientierter** Editor (analog *Sed*).
- Als **Filter-Programm** in Pipelines verwendbar.
- Bietet eine **automatische Leseschleife**, d.h. es ist kein Öffnen von Dateien notwendig.
- Besitzt eine Reihe sinnvoller **Automatismen** (z.B. Zerlegung der Eingabezeilen in Wörter (Felder) und automatisches Initialisieren von Variablen), wodurch die Verarbeitung von Textdateien (zeilenorientierten ASCII-Dateien) und die Programmierung stark vereinfacht werden.
- Kennt (nur) 2 Datentypen (und Arrays davon):
  - ▷ **Gleitkommazahl**: double, emuliert Integer (maximal 16 Stellen).
  - ▷ **Zeichenkette**: dynamisch, kann beliebig lang sein.
- Bietet **mehrdimensionale, dynamische, assoziative Arrays** (auch **Hashes** genannt, d.h. als Index sind beliebige Zeichenketten zugelassen).
- Erlaubt Zeichenkettenvergleiche mit (erweiterten) **Regulären Ausdrücken**.

- Die **Verarbeitung von Zeichenketten** ist einfach + sicher (es ist keine Speicherplatzreservierung oder -freigabe notwendig).
- Bietet **(rekursive) Funktionen** mit Parametern (Funktions-Prototypen sind nicht notwendig).

1.5.1 Vorteile

- Erlaubt eine schnelle, interaktive Entwicklung.
- Automatische Zerlegung der Eingabedateien in **Sätze** (Satztrenner frei wählbar).
- Automatische Zerlegung der Eingabesätze in **Worte** (Felder, Feldtrenner frei wählbar).
- *C*-ähnliche Syntax und analoge Eigenschaften (`; { }`, Kontrollstrukturen, Operatoren, `printf`-Funktion).
- **Automatische Konvertierung** zwischen den Datentypen Zahl und Zeichenkette (*as needed*):

  ▷ `"123" + "456" -> 579`

  ▷ `"12abc" + 1 -> 13`

- **Automatische Speicherverwaltung** für Zeichenketten, Variablen und Arrays, d.h. explizites Belegen und Freigeben von Speicher ist nicht notwendig (*Garbage Collection*).
- **Keine Variablendeklaration** notwendig, beim ersten Auftreten werden sie automatisch mit `0/""` initialisiert (*d.h. der Aufwand für Deklarationen fällt weg*).

1.5.2 Nachteile

- **Relativ langsam** bei großen Datenmengen (etwa ab 10 MByte bzw. 100.000 Zeilen).
- Kann **nur Textdateien** (zeilenorientierte ASCII-Daten) verarbeiten, aber keine Binärdaten. Durch Vor- und Nachschalten entsprechender Konverter kann dies aber ausgeglichen werden (z.B. durch `bin2asc file1 | awk ... | asc2bin > file2`).
- Es ist **kein wahlfreies Positionieren** in Dateien möglich, d.h. sie können nur von vorne nach hinten durchgelesen werden.
- Kennt **keine Datenstrukturen** oder Zeiger (diese können aber über assoziative Arrays simuliert werden).
- Kennt keine **modullokalen Variablen**, nur globale/lokale zu Funktionen (in *Tawk* schon).
- **Keine Variablendeklaration** notwendig, beim ersten Auftreten werden sie automatisch mit `0/""` initialisiert (*d.h. Tippfehler führen neue Variablen ein*).

- Große Programme können leicht unübersichtlich werden (die Aufteilung eines Programms auf mehrere Dateien (= Module) ist aber möglich).
- Kein Präprozessor vorhanden (inbesondere keine `#include`-Anweisung).
- Kein Debugger vorhanden (in *Tawk* schon).
- Arrayelemente sind unsortiert (Hash-Funktion), Sortieren muß extern erfolgen.
- Zugriff auf einzelne Zeichen in Zeichenketten ist nicht per Index (wie in *C*) möglich, sondern nur per Funktion `substr` bzw. Verkettung (Konkatenation), d.h. ist etwas langsam und umständlich.
- Die Syntaxfehlermeldungen sind recht spartanisch (bisweilen sogar kryptisch).
- Es gibt nicht „den *Awk*“, sondern verschiedene Versionen, die sich mehr oder weniger stark in ihrem Verhalten unterscheiden.

1.6 Ein einfaches Beispiel

Das folgende Beispiel summiert alle Zahlen der 2. Textspalte in der Datei `data` mit folgendem Inhalt (die Spalten sind durch ein oder mehrere Leerzeichen getrennt):

```
Susanne  15.0
Thomas   23.0
Richard   0.0
Birgit   -2.0
Helmut   31.0
```

und gibt als Ergebnis `sum = 67` aus:

```
awk '{ sum = sum + $2 } END { print "sum =", sum }' data
```

Ablauf: Das *Awk*-Programm ist das 1. Argument, es ist durch einfache Anführungszeichen `'...'` geschützt (*quotiert*), da die Shell sonst einige Sonderzeichen selbst interpretieren würde, statt sie an den *Awk* zu übergeben. Das 2. Argument `data` ist die zu verarbeitende Datei. Die Variable `sum` ist zu Beginn mit dem Wert `0` vorbelegt. Die Datei `data` wird zeilenweise eingelesen, jede eingelesene Zeile wird in 2 Felder zerlegt (der Inhalt des 2. Feldes wird in `$2` abgelegt) und für jede Zeile (leeres Muster) wird das 2. Feld zur Variablen `sum` dazuaddiert. Nach dem Lesen aller Zeilen von `data` wird in der `END`-Regel die Endsumme ausgegeben.

Die obigen Anweisungen können statt auf der Kommandozeile z.B. auch in einer **Skript-Datei** namens `addcol2.awk` abgelegt werden (der Übersicht halber stehen die beiden Regeln jetzt in je einer eigenen Zeile und die Aktion der ersten Regel ist eingerückt):

```
    { sum = sum + $2 }
END { print "sum =", sum }
```

Der Aufruf erfolgt dann folgendermaßen:

```
awk -f addcol2.awk data
```

Als dritte Alternative kann das *Awk*-Skript mit dem Kommando `chmod +x addcol2.awk` unter UNIX direkt ausführbar gemacht werden. Der Aufruf erfolgt dann ohne Angabe des Kommandos `awk -f`:

```
addcol2.awk data
```

Dazu muß folgende Zeile zusätzlich als 1. Zeile in das *Awk*-Skript `addcol2.awk` aufgenommen werden (ab der 1. Spalte):

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

Diese Zeile sorgt dafür, daß der **UNIX-Kernel** bei der Ausführung des Skriptes automatisch das Programm *Awk* aus dem Verzeichnis `/usr/bin` startet und das Skript mit Hilfe der Option `-f` [**file**] an es übergibt. Diese Zeile wird vom aufgerufenen *Awk* anschließend ignoriert, da alle Zeichen nach `#` von ihm als Kommentar interpretiert werden.

1.7 10 typische Einzeiler

Obwohl sich der *Awk* auch zum Erstellen umfangreicher Programme eignet, sind viele nützliche Programme nur ein oder zwei Zeilen lang. Hier eine Auswahl von 10 typischen Einzeilern, zu ihrem Verständnis genügen folgende Informationen: Die Variable `$0` enthält die aktuelle Zeile, die Variable `NR` enthält die Nummer der aktuellen Zeile, die Variablen `$1`..`$`n enthalten das 1. bis n-te Wort der aktuellen Zeile, die Variable `NF` enthält die Anzahl der Worte der aktuellen Zeile, die `END`-Regel wird nach dem Einlesen aller Eingabedaten ausgeführt.

1. Die Anzahl der Eingabezeilen ausgeben:

   ```
   END { print NR }
   ```

2. Die zehnte Eingabezeile ausgeben:

   ```
   NR == 10
   ```

3. Jede Eingabezeile mit mehr als vier Feldern (Wörtern) ausgeben:

   ```
   NF > 4
   ```

4. Die Gesamtzahl aller Felder (Wörter) aller Eingabezeilen ausgeben:

   ```
   { nw += NF } END { print nw }
   ```

5. Die Anzahl aller Zeilen ausgeben, die `Barbara` enthalten:

```
/Barbara/ { ++cnt } END { print cnt }
```

6. Jede Zeile ausgeben, die mindestens ein Feld (Wort) enthält:

```
NF > 0
```

7. Jede Zeile ausgeben, die länger als 80 Zeichen ist:

```
length($0) > 80
```

8. Die beiden ersten Felder (Worte) jeder Zeile vertauschen und dann die Zeile ausgeben:

```
{ tmp = $1; $1 = $2; $2 = tmp; print }
```

9. Jede Zeile mit ihrer Zeilennummer davor ausgeben:

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

10. Jede Zeile ausgeben und vorher das zweite Feld (Wort) löschen:

```
{ $2 = ""; print }
```

1.8 Referenzen

1.8.1 Literatur

- Aho, Kernighan, Weinberger, *The AWK Programming Language*, Addison-Wesley.

  Die „Bibel“ zum *Awk* von den Autoren der Sprache selbst. Enthält neben einer kompakten Definition der Sprache (auf 45 Seiten) viele Beispiele zur Anwendung des *Awk* in vielen (auch anspruchsvollen) Gebieten der Informatik.

- Robbins, *Effective AWK Programming, 3rd Edition*, O'Reilly.

  Die „Bibel“ zum *Gawk*. Enthält eine Definition der Sprache, listet penibel die Erweiterungen des *Gawk* und die „dunklen Ecken“ der Sprache auf und enthält auch eine Reihe von Anwendungsbeispielen.

- Dougherty, *Sed & Awk, 2. Edition*, O'Reilly & Associates.

  Behandelt die Themen *Reguläre Ausdrücke*, *Sed* und *Awk* gemeinsam und enthält viele Anwendungsbeispiele für die beiden Programme.

- Thompson Automation Software, *Tawk Compiler*.

  Enthält die Beschreibung der *Tawk*-Sprache, die eine deutliche Erweiterung des *Awk* darstellt. Sie bietet insbesondere auch einen Compiler, mit dem direkt ausführbare *Awk*-Programme erstellt werden können.

- Gilly, *UNIX in a Nutshell, 2. Edition*, O'Reilly & Associates.

  Enthält eine Kurzbeschreibung der Regulären Ausdrücke, des *Sed* und des *Awk*.

- Staubach, *UNIX-Werkzeuge zur Textmusterverarbeitung*, Springer.

  Enthält eine kompakte deutsche Beschreibung der Regulären Ausdrücke und des *Awk*.

1.8.2 Links

- `www.ostc.de`: Dieses Skript und weitere Awk-Infos.
- `awka.sourceforge.net`: Awk-Compiler.
- `cm.bell-labs.com/cm/cs/awkbook`: Ergänzungen zum Buch „The AWK Programming Language“.
- `www.cs.hmc.edu/tech_docs/gref/awk.html`: Mit Awk loslegen.
- `www.canberra.edu.au/ sam/whp/awk-guide.html`: Handbuch zum Awk.
- `www.novia.net/ phridge/programming/awk`: Programmier-Beispiele zum Awk.
- `www.tasoft.com`: Thompson Automation Software — Tawk-Compiler.
- `www.oase-shareware.org/shell/goodcoding/awkcompat.html`: Awk Kompatibilitäts-Liste.
- `www.faqs.org/faqs/computer-lang/awk/faq`: Awk Frequently Asked Questions.
- `www.shelldorado.org`: Shell-Skriptsprachen-Seite die auch zu Awk einiges enthält.

2 Beschreibung

2.1 Aufruf, Parameter und Optionen

Entweder wird ein *Awk*-**Programm** `PROGRAM` direkt auf der Kommandozeile als 1. Argument angegeben (in einfachen Hochkommata, um zu verhindern, daß Metazeichen darin von der Shell interpretiert werden):

```
awk [OPTIONS] 'PROGRAM' [FILE...]
```

oder es steht in einer **Skript-Datei** `PROGFILE` und wird über die Option `-f` [**file**] ausgewählt (diese Option kann auch mehrfach angegeben werden, um ein in mehrere Module zerlegtes *Awk*-Programm einzulesen):

```
awk [OPTIONS] -f PROGFILE [FILE...]
```

Wenn **Dateinamen** `FILE...` angegeben sind, werden diese Dateien der Reihe nach zeilenweise gelesen, sonst wird zeilenweise von der *Standard-Eingabe* gelesen. Die Ausgabe erfolgt auf der *Standard-Ausgabe* (kann aber auch umgelenkt werden). Die möglichen **Optionen** `OPTIONS` sind:

Option	Bedeutung
-F *sep*	Feldtrenner FS auf *sep* festlegen (**Default:** "␣")
-f *progfile*	Programm von *progfile* einlesen (*mehrfach erlaubt*)
-v *var=text*	Variable *var* mit dem Wert *text* belegen (*vor* BEGIN)
-mf *n*	Maximale Anzahl Felder *n* festlegen (*im Gawk überflüssig*)
-mr *n*	Maximale Recordlänge *n* festlegen (*im Gawk überflüssig*)
--	Beendet die Optionen-Liste

Hat ein Dateiname in FILE... die Form *var=text*, so erhält die Variable *var* den Wert *text* zugewiesen und der *Awk* springt zum nächsten Argument. **Achtung:** Diese Zuweisung wird erst zu dem Zeitpunkt ausgeführt, an dem auf das Argument als Datei zugegriffen würde.

Folgende **Optionen** sind *Gawk*-spezifisch:

Gawk-**Option**	**Bedeutung**
--compat	*Gawk*-Erweiterungen abschalten
--traditional	*Gawk*-Erweiterungen abschalten
--copyleft	GPL (GNU General Public Licence) ausgeben
--copyright	GPL (GNU General Public Licence) ausgeben
--help	*Gawk*-Usage-Meldung ausgeben
--usage	*Gawk*-Usage-Meldung ausgeben
--version	*Gawk*-Versionsnummer ausgeben
--lint	Warnung bei gefährlichen/nicht portablen Konstrukten ausgeben
--lint-old	Warnung bei Konstrukten abweichend vom Ur-*Awk* ausgeben
--posix	Nur POSIX-Umfang zulassen
--re-interval	{n,m}-Wiederholung in Regulären Ausdrücken erlauben
--source '*program*'	Programmtext *program* auf der Kommandozeile angeben (*mehrfach erlaubt*)

2.2 Konstanten

Als Konstanten sind möglich: **Zahlen**, **Zeichenketten**, **Logische Werte** und **Reguläre Ausdrücke**. Zeichenkonstanten wie in *C* ('x') sind *nicht* verfügbar, sie sind allerdings durch einbuchstabige Zeichenketten ("x") ersetzbar.

2.2.1 Zahlen

Erlaubt sind Ganzzahlen oder Fließkommazahlen mit Dezimalpunkt und Exponent. Intern werden *alle* Zahlen als double mit 16-stelliger Genauigkeit gespeichert. Beispiele:

```
123   -1   -3.141592   +.0125   1e12   -987.654E-321
```

2.2.2 Zeichenketten

Zeichenketten sind in der Form "..." anzugeben, die **leere Zeichenkette** hat die Form "". Beispiele:

```
"abc"
"Dies ist ein Text.\n"
""                          (leere Zeichenkette)
```

2.2.2.1 Escape-Sequenzen

Folgende **Escape-Sequenzen** sind in Zeichenketten `"..."` und Regulären Ausdrücken `/.../` erlaubt (`*` = nur im *Gawk* vorhanden):

Gawk	Escape	Bedeutung
*	\a	Akustisches Signal [**alert**]
	\b	Backspace (Zeichen zurück)
	\f	Seitenvorschub [**formfeed**]
	\n	Zeilenvorschub [**newline**]
	\r	Wagenrücklauf [**carriage return**]
	\t	Tabulator
*	\v	Vertikaler Tabulator
	\ddd	Zeichen mit oktalem Wert ddd (Ziffern zwischen 0 und 7)
*	\x$dd$	Zeichen mit hexadezimalem Wert dd (Ziffern zwischen 0 und f/F)
	\"	Anführungszeichen
	\\	Backslash
	\/	Slash

2.2.3 Logische Werte

Die Werte `0` (Zahl 0) und `""` (leere Zeichenkette) werden als **falsch** interpretiert, alle anderen Zahlen bzw. Zeichenketten werden als **wahr** interpretiert. **Hinweis:** Insbesondere sind die Zahl `1` und die Zeichenkette `"0"` wahr (letzteres gilt in *Perl* nicht!).

2.2.4 Reguläre Ausdrücke

Reguläre Ausdrücke `REGEXP` sind in der Form `/REGEXP/` anzugeben. Sie beschreiben Eigenschaften von Zeichenketten (z.B. daß sie bestimmte Zeichen oder Zeichenfolgen enthalten oder nicht enthalten, eine bestimmte Form haben, einen bestimmten Anfang oder Ende haben, ...). Alternativ können sie auch in Form einer Zeichenkette `"..."` angegeben werden (**Hinweis:** Backslashes sind darin dann zu verdoppeln) oder in einer Variablen stehen. Sie können auf diese Weise sogar *dynamisch zur Laufzeit* erzeugt werden (dies führt allerdings zu Performance-Einbußen). Beispiele:

```
/abc/           Enthält "abc"
/^abc$/         Lautet exakt "abc"
/(abc|def)+/    Enthält "abc" oder "def" mind. 1x nacheinander
```

2.2.4.1 Metazeichen

Folgende **Metazeichen** sind in Regulären Ausdrücken möglich, sie sind nach **absteigendem Vorrang** geordnet. r, $r1$ und $r2$ sind Reguläre Ausdrücke, sie spiegeln die rekursive Definition von Regulären Ausdrücken wider (* = nur im *Gawk* vorhanden):

Gawk	Metazeichen	Bedeutung
	(r)	**Gruppierung**: jede Zeichenkette, auf die r paßt
	c	Zeichen c (kein Metazeichen)
	$\backslash c$	Eine Escape-Sequenz oder das (Meta)Zeichen c wörtlich
	^	Zeichenketten-/Zeilenanfang
	$	Zeichenketten-/Zeilenende
	.	Jedes beliebige Zeichen
	[abc], [a-z]	**Zeichenliste**: jedes Zeichen in abc, a–z
	[^abc], [^a-z]	**Negierte Zeichenliste**: jedes Zeichen außer abc, a–z
*	\y \B	Wortgrenze / Wortinneres
*	\< \>	Wortanfang / Wortende
*	\w \W	Wortzeichen ([A-Za-z_0-9]) / kein Wortzeichen ([^A-Za-z_0-9])
*	\` \'	Zeichenketten-/Zeilenanfang / -ende (alternative Form)
	r*	$0-\infty$ aufeinanderfolgende Zeichenketten, auf die r paßt (**Closure**)
	r+	$1-\infty$ aufeinanderfolgende Zeichenketten, auf die r paßt (**Closure**)
	r?	$0/1$ Zeichenketten, auf die r paßt (**Option**)
*	r{m,n}	$m-n$ Wiederholungen der Zeichenkette, auf die r paßt
*	r{m,}	$m-\infty$ Wiederholungen der Zeichenkette, auf die r paßt
*	r{m}	Genau m Wiederholungen der Zeichenkette, auf die r paßt
	$r1\,r2$	**Verkettung**: Jede Zeichenkette xy, wo $r1$ zu x **und** $r2$ zu y paßt
	$r1$|$r2$	**Alternative**: Jede Zeichenkette, die zu $r1$ **oder** $r2$ paßt

- Redundante Klammern können bei passendem Vorrang weggelassen werden.
- Die Metazeichen ( ) \ ^ $. [ ] * + ? { } | müssen mit \ **quotiert** werden, wenn sie wörtlich gemeint sind.
- Das Zeichen / muß mit \ quotiert werden, wenn es in einem Regulären Ausdruck vorkommt.
- Die Wiederholung mittels {m,n} ist nur bei Angabe der Option --re-interval erlaubt.
- Leerzeichen und Tabulatoren stehen für sich selbst, sie werden nicht ignoriert.
-

2.2.4.2 POSIX-Zeichenklassen

Innerhalb von Zeichenlisten sind **POSIX-Zeichenklassen** der Form [[:*class*:]] erlaubt, sie dienen zur Angabe von Zeichen unabhängig von der verwendeten Zeichencodierung (ASCII, EBCDIC, ...), aber z.B. abhängig von der verwendeten Landessprache. Folgende POSIX-Zeichenklassen *class* gibt es:

Klasse	Bedeutung
`alnum`	Alphanumerische Zeichen (Buchstaben + Ziffern)
`alpha`	Buchstaben
`blank`	Leerzeichen oder Tabulator
`cntrl`	Control-Zeichen
`digit`	Dezimalziffern
`graph`	Alle druckbaren und sichtbaren Zeichen
`lower`	Kleine Buchstaben
`print`	Druckbare Zeichen (keine Kontroll-Zeichen)
`punct`	Satzzeichen
`space`	Whitespace (Leerzeichen, Tabulator, Zeilenvorschub, ...)
`upper`	Große Buchstaben
`xdigit`	Hexadezimalziffern

2.3 Ausdrücke (Expressions)

Ein **Ausdruck** (engl. **Expression**) entsteht durch die Verknüpfung von **Operanden** (Konstanten, Feldern, Variablen, Arrayelementen und Funktionsaufrufen) über **Operatoren** (siehe Abschnitt 2.7 auf Seite 30). Jeder Ausdruck hat einen Wert. Bei **numerischen Ausdrücken** ergibt sich ein numerischer Wert, bei **Zeichenkettenausdrücken** eine Zeichenkette und bei **logischen Ausdrücken** ein logischer Wert `0` (**wahr**) oder `1` (**falsch**). Beispiele:

```
1 + (2 * exp(3) + sqrt(100)) ^ 2                          numerisch
substr($0, 1, 4) "String"                                 Zeichenkette
i <= 10                                                   logisch
user !~ /Dieter/ && (age >= 18 || permission == "yes")    logisch
```

Es ist jederzeit möglich, den Wert eines numerischen Ausdrucks oder eines Zeichenkettenausdrucks auch als logischen Wert zu interpretieren. Er wird dann gemäß der Definition in Abschnitt 2.2.3 auf Seite 14 in einen logischen Wert umgewandelt. Umgekehrt können auch logische Ausdrücke als numerischer Ausdruck oder Zeichenkettenausdruck interpretiert werden, sie haben dann den Wert `1/"1"` (wahr) oder `0/""` (falsch).

2.4 Programmaufbau

Ein *Awk*-Programm besteht aus beliebig vielen **Regeln (Muster-Aktion Paaren)** und **Funktionsdefinitionen**. Regeln und Funktionsdefinitionen werden durch Zeilenvorschübe getrennt und dürfen in beliebiger Reihenfolge auftreten. **Achtung:** *Die Reihenfolge der Regeln ist von Bedeutung, sie werden in der Reihenfolge ihrer Definition auf die Eingabedaten angewendet.*

2.4.1 Regeln (Muster-Aktion Paare)

Eine **Regel** hat folgenden Aufbau:

```
MUSTER { AKTION }
```

Ein **Muster** ist entweder `BEGIN`, `END`, ein logischer Ausdruck (analog *C*), ein Regulärer Ausdruck oder ein Bereichsmuster. Eine **Aktion** ist eine beliebig lange Folge von **Anweisungen**, die in geschweifte Klammern einzuschließen sind. Ist ein Muster (für die aktuelle Eingabezeile) **wahr**, so wird seine entsprechende Aktion ausgeführt. Entweder das Muster oder die Aktion darf auch fehlen:

- Ein fehlendes Muster ist für alle Eingabesätze wahr, d.h. die Aktion wird für alle Eingabesätze ausgeführt.
- Eine fehlende Aktion entspricht `{ print $0 }`, d.h. die aktuelle Eingabezeile wird ausgegeben.

2.4.1.1 Muster

Folgende Muster sind möglich:

Muster
`BEGIN` `END`
`Ausdruck` `/REGEXP/`
`Ausdruck, Ausdruck` `/REGEXP/, /REGEXP/`

- Das Muster `BEGIN` ist *vor* dem Lesen der ersten Datei wahr, das Muster `END` *nach* dem Lesen der letzten Datei. D.h. hier kann **Preprocessing** (z.B. Initialisierung) und **Postprocessing** (z.B. Ausgabe von Gesamtwerten) durchgeführt werden.
- Das Muster `Ausdruck` ist ein beliebiger (logischer) Ausdruck wie in *C* (siehe Abschnitt 2.3 auf Seite 16). Es ist wahr, wenn es ausgewertet für die aktuelle Eingabezeile den Wert wahr ergibt.
- Das Muster `/REGEXP/` ist ein beliebiger Regulärer Ausdruck. Es ist wahr, wenn es zur aktuellen Eingabezeile paßt.
- Die beiden letzten Muster heißen **Bereichsmuster**. Sie sind wahr, sobald die aktuelle Eingabezeile den ersten Ausdruck erfüllt und falsch, nachdem die aktuelle Eingabezeile den letzten Ausdruck erfüllt (einschließlich). Dies kann sich beliebig oft für die Eingabedaten wiederholen, auf diese Weise können geklammerte (zusammenhängende) Bereich der Eingabedaten bearbeitet werden.

2.4.1.2 Aktion

Eine Aktion ist eine Folge von **Anweisungen**, die meisten Anweisungen sind **Kontrollstrukturen**, die den Programmablauf steuern (Sequenz, Verzweigung, Schleife, Unterprogrammaufruf, Rücksprung). Folgende Anweisungen gibt es (* = in *C* nicht vorhanden):

!C	Anweisung
	`break`
	`continue`
	`do` *Anweisung* `while (`*Ausdruck*`)`
	`exit` [*Ausdruck*]
	`for (`*Ausdruck1*`;` *Ausdruck2*`;` *Ausdruck3*`)` *Anweisung*
*	`for (`*Variable* `in` *Array*`)` *Anweisung*
	`if (`*Ausdruck*`)` *Anweisung1* [`else` *Anweisung2*]
*	`next`
*	`nextfile` (*Gawk*)
	`return` [*Ausdruck*]
	`while (`*Ausdruck*`)` *Anweisung*
	Ein/Ausgabe-Anweisung (`getline`, `print` und `printf`)
*	`delete` *Arrayelement/Array*
	Ausdruck (mit Konstanten, Variablen, Zuweisungen, Funktionsaufrufen, ...)
	Funktion`(`*Argumente*`)`
	`{` *Anweisung*`;` `...` `}` (**Block**)

- Da in den möglichen Anweisungen wieder *Anweisung* als Bestandteil auftaucht, handelt es sich um eine **rekursive Definition**. Für die Definition von Ausdrücken siehe Abschnitt 2.3 auf Seite 16.
- Die nicht mit * gekennzeichneten Anweisungen verhalten sich wie die entsprechenden *C*-Anweisungen:
  - ▷ `break` (**Abbruch**) — verläßt die umschließende `do`-, `for`- oder `while`-Schleife.
  - ▷ `continue` (**Fortsetzung**) — beginnt die nächste Iteration der umschließenden `do`-, `for`- oder `while`-Schleife.
  - ▷ `do` *Anweisung* `while (`*Ausdruck*`)` (**Nicht abweisende Schleife**) — führt *Anweisung* aus, solange *Ausdruck* wahr ist, *Anweisung* wird mindestens einmal ausgeführt.
  - ▷ `exit` [*Ausdruck*] (**Abbruch**) — verläßt das Programm mit dem Exit-Code *Ausdruck* (oder `0`). Die Aktionen eines eventuell vorhandenen `END`-Musters werden vorher noch ausgeführt.
  - ▷ `for (`*Ausdruck1*`;` *Ausdruck2*`;` *Ausdruck3*`)` *Anweisung* (**Zählschleife**) — führt *Anweisung* aus, solange *Ausdruck2* wahr ist (**Bedingung**). *Ausdruck1* wird beim Erreichen der Schleife einmal ausgeführt (**Initialisierung)**, *Ausdruck3* wird jedesmal nach der Ausführung von *Anweisung* ausgeführt (**Fortschaltung**). *Anweisung* wird eventuell überhaupt nicht ausgeführt (wenn *Ausdruck2* bereits beim ersten Mal falsch ist).
  - ▷ `if (`*Ausdruck*`)` *Anweisung1* [`else` *Anweisung2*] (**Verzweigung**) — führt *Anweisung1* aus, wenn *Ausdruck* wahr ist. Ist der `else`-Teil vorhanden, so wird *Anweisung2* ausgeführt, wenn *Ausdruck* falsch ist, ansonsten wird gar nichts ausgeführt.
  - ▷ `return` [*Ausdruck*] (**Rücksprung**) — ist nur in Funktionen (mehrfach) erlaubt und verläßt diese, Rückgabewert ist *Ausdruck* (oder `0`).

- ▷ `while (`*Ausdruck*`)` *Anweisung* (**Abweisende Schleife**) — führt *Anweisung* aus, solange *Ausdruck* wahr ist. *Anweisung* wird eventuell überhaupt nicht ausgeführt (wenn *Ausdruck* bereits beim ersten Mal falsch ist).
- ▷ *Ein/Ausgabe-Anweisung* — führt die entsprechende Ein- oder Ausgabe durch (mit allen Seiteneffekten).
- ▷ *Ausdruck* — wertet den entsprechenden Ausdruck aus (mit allen Seiteneffekten).
- ▷ *Funktion*`(`*Argumente*`)` (**Unterprogrammaufruf**) — führt die Anweisungen der entsprechenden Funktion durch und kehrt dann zur folgenden Anweisung der Aufrufstelle zurück.
- ▷ `{` *Anweisung ...* `}` (**Sequenz**) — faßt einen **Block** von Anweisungen zusammen, die eine nach der anderen ausgeführt werden.

- Die mit `*` gekennzeichneten in *C* nicht bekannten Anweisungen haben folgendes Verhalten:
  - ▷ `for (`*Variable* `in` *Array*`)` *Anweisung* (**Indexschleife**) — durchläuft alle Elemente von *Array* und legt dabei ihren Index in *Variable* ab, bevor *Anweisung* ausgeführt wird (Achtung: Der Durchlauf erfolgt in keiner bestimmten Reihenfolge).
  - ▷ `next` — liest den nächsten Eingabesatz ein und beginnt die Verarbeitung wieder bei der ersten Regel.
  - ▷ `nextfile` — beendet das Einlesen der aktuellen Eingabedatei und beginnt mit dem Einlesen der nächsten Eingabedatei; die Verarbeitung beginnt wieder bei der ersten Regel (nur im *Gawk* vorhanden).
  - ▷ `delete` *Arrayelement/Array* — löscht je nach Aufruf ein oder alle Arrayelemente.
- Anweisungen sind durch Zeilenvorschübe, Semikolons „`;`„ oder beides zugleich zu trennen. **Hinweis:** Sinnvollerweise sollte jede Anweisung durch ein Semikolon abgeschlossen werden (auch wenn sie am Zeilenende steht), sonst bekommt man beim Umstieg auf *C* oder *Perl* Probleme mit der Syntax.
- Ein alleinstehendes Semikolon steht für die **leere Anweisung**, z.B. in einer leeren `for`-Schleife:

```
for (i = 0; i < 1000 && arr[i] != ""; ++i)
    ;
```

- Kommt mehr als eine Anweisung im Körper einer `do`-, `for`- oder `while`-Schleife oder in einem `if`- oder `else`-Zweig vor, so *müssen* sie in Blockklammern eingeschlossen werden:

```
for (i = 0; i < 100; ++i) {
    sum = sum + i
    squaresum = squaresum + i ^ 2      # oder i * i
}
```

- In einer `if-else`-Anweisung ist die Anweisung nach dem `if` durch ein Semikolon abzuschließen oder in geschweifte Klammern einzuschließen, wenn sie in der gleichen Zeile wie das `else` auftritt:

```
if (a > b) max = a; else max = b       # oder
if (a > b) { max = a } else max = b
```

- Ebenso ist in einer `do`-Anweisung die Anweisung durch ein Semikolon abzuschließen oder in geschweifte Klammern einzuschließen, wenn sie in der gleichen Zeile wie das `while` auftritt:

```
do --i; while (i > 0)      oder
do { --i } while (i > 0)
```

2.4.2 Funktionen

Eine **Funktionsdefinition** faßt eine Folge von Anweisungen unter einem frei wählbaren **Funktionsnamen** zusammen. Durch einen Aufruf über diesen Namen kann die Anweisungsfolge beliebig oft ausgeführt werden, Funktionen dürfen sich auch selbst (**rekursiv**) aufrufen. Beim Aufruf können an die Funktion Werte als **Argumente** übergeben werden, die ihr Verhalten beeinflussen, indem sie im **Funktionskörper** verwendet werden. Die Funktion kann weiterhin einen **Rückgabewert** festlegen, der an der Stelle des Funktionsaufrufes eingesetzt wird (Funktionen ohne Rückgabewert werden auch **Prozeduren** genannt). **Hinweis:** Bei der Definition und beim Aufruf einer Funktion darf zwischen dem Funktionsnamen und der öffnenden Klammer *kein* Leerzeichen stehen.

2.4.2.1 Definition

Eine **Funktionsdefinition** hat die Form:

```
function FUNCNAME(PARAM1, PARAM2, ...)
{
    Anweisung1
    Anweisung2
    ...
    return RESULT
}
```

Ein Funktionsname wird wie in *C* gebildet (`[A-Za-z_][A-Za-z_0-9]*`), Groß/Kleinschreibung wird beachtet. Bei der Definition einer Funktion *muß* das Schlüsselwort `function` vor dem Funktionsnamen angegeben werden.

Die Funktion kennt dann die **lokalen Parameter** `PARAM1`, `PARAM2`, `...`, besteht aus dem **Funktionskörper** `Anweisung1`, `Anweisung2`, `...` und gibt am Ende den Rückgabewert `RESULT` zurück.

Die Reihenfolge der Funktiondefinitionen ist beliebig, sie dürfen auch beliebig mit Regeln gemischt werden. Eine Funktion muß vor ihrem ersten Aufruf *nicht* definiert sein (es gibt keine Funktionsdeklaration wie in *C*).

Die in einer Funktionsdefinition aufgelisteten Parameter `PARAM1`, `PARAM2`, ... verhalten sich in dieser Funktion wie **lokale Variablen**, gleichnamige globale Variablen werden von ihnen innerhalb der Funktion überdeckt. Zusätzliche lokale Variablen können durch weitere (eigentlich nicht benötigte) Parameter simuliert werden (zur Kenntlichmachung z.B. durch 4 Leerzeichen abtrennen). Beispiel:

```
function max3(a, b, c,    tmp)    # a b c = 3 Parameter
{                                 # tmp = lokale Variable
    tmp = a
    if (tmp < b) tmp = b
    if (tmp < c) tmp = c
    return tmp
}
```

2.4.2.2 Aufruf

Der **Aufruf einer Funktion** hat folgende Form:

```
FUNCNAME(EXPR1, EXPR2, ...)
```

Die **lokalen Parameter** `PARAM1`, `PARAM2`, ... der Funktion werden mit dem Ergebnis der Ausdrücke `EXPR1`, `EXPR2`, ... belegt, bevor der Funktionskörper ausgeführt wird. Der Rückgabewert einer Funktion kann ignoriert, einer Variablen zugewiesen oder in einem Ausdruck verwendet werden. Beispiel:

```
max = max3(100, 2 * 53 - 1, -123)
```

Felder und Variablen werden **by value** (als Kopie), Arrays werden **by reference** (als Zeiger) übergeben. D.h. eine übergebene Variable selbst kann in einer Funktion *nicht* verändert werden (nur ihre lokale Kopie), die Elemente eines übergebenen Arrays *können* hingegen verändert werden.

Die Anzahl der beim Funktions-Aufruf angegebenen Parameter muß nicht mit der bei der Funktions-Definition vorgegebenen Anzahl übereinstimmen:

- Werden beim Aufruf einer Funktion *weniger* Argumente übergeben, als die Funktion Parameter besitzt, so werden die *überflüssigen* Parameter (= lokale Variablen) automatisch mit den Wert `0`/`""` initialisiert (*im Gegensatz zu C*).
- Werden beim Aufruf einer Funktion *mehr* Argumente übergeben, als die Funktion Parameter besitzt, so werden die *überflüssigen* Parameter ignoriert.

2.4.3 Kommentare, Whitespace, Leerzeilen und Zeilenumbruch

- **Kommentare** werden durch `#` eingeleitet und erstrecken sich bis zum Zeilenende, sie sind am Ende jeder Zeile erlaubt. Können überall eingefügt werden, um den *Awk*-Code übersichtlicher zu gestalten.
- **Whitespace** (Leerzeichen und Tabulatoren) kann überall in Anweisungen und Ausdrücken eingefügt werden, um den *Awk*-Code übersichtlicher zu gestalten. **Hinweis:** Zum Einrücken gemäß der **Blockstruktur** sollten nur Tabulatoren verwendet werden.
- **Leerzeilen** können überall (zwischen Regeln, Funktionsdefinitionen und Anweisungen) eingefügt werden, um den *Awk*-Code übersichtlicher zu gestalten.
- **Zeilenumbrüche** können beliebig *zwischen* Anweisungen eingefügt werden. *Innerhalb* von Anweisungen und Ausdrücken darf hingegen *nicht beliebig* umgebrochen werden (wie in *C*), sondern nur nach:

Token	Bedeutung
`,`	Komma (in `print`, `printf`, Funktionsdefinition und -aufruf)
`{`	Linke geschweifte Klammer (Blockbeginn)
`?`	Bedingte Operation
`:`	Bedingte Operation
`&&`	Logisch Und
`||`	Logisch Oder
`if (...)`	Bedingung
`else`	Alternative
`for (...)`	Zählschleife
`while (...)`	Abweisende Schleife
`do`	Nicht abweisende Schleife
`}`	Rechte Klammer in einer `if`-, `for`- oder `while`-Anweisung

- Anweisung und Ausdrücke können mit Hilfe von **Backslashes** am Zeilenende (direkt davor) *beliebig* umgebrochen werden, das Zeilenende wird dadurch „maskiert“:

```
for (i = 0; \
     i <= 100; \
     ++i)
   sum = i * \
         i
```

- `for`-Anweisungen können nicht umgebrochen werden (außer durch Backslash).

2.4.4 Strukturierung von Programmen

Die **Abhängigkeits-Struktur** eines Programms sollte durch geeignete Einrückungen kenntlich gemacht werden. Dies ist zwar für den Rechner nicht notwendig, für den Programmierer aber zum Verständnis eines Programms sehr hilfreich. Es gibt sehr viele **Einrückungsstile**, wichtig ist nur, sich für einen einfachen und nachvollziehbaren zu entscheiden und daran konsequent festzuhalten. Auch während der Entwicklung eines Programms sollten alle

Einrückungen sofort durchgeführt bzw. bei Änderungen sofort angepaßt werden. Auf diese Weise werden viele Programmierfehler bereits im Ansatz vermieden. Entscheiden Sie selbst, welches der drei folgenden identischen Programmstücke am einfachsten zu verstehen und zu warten ist:

- Nicht eingerückt:

```
for (i = begin; i < end; ++i) {
for (j = i+1; j <= end; ++j) if (arr[i] != arr[j]) break
if (j > i + 1) {
dup_cnt = j - 1 - i
for (k = j-1; k <= end; ++k)
arr[k - dup_cnt] = arr[k]
end -= dup_cnt
}
}
```

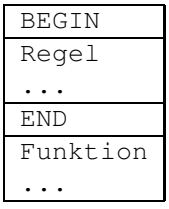
- Fehlerhaft eingerückt:

```
for (i = begin; i < end; ++i) {
    for (j = i+1; j <= end; ++j)
        if (arr[i] != arr[j])
    break
    if (j > i + 1) {
        dup_cnt = j - 1 - i
    for (k = j-1; k <= end; ++k)
        arr[k - dup_cnt] = arr[k]
        end -= dup_cnt
    }
}
```

- Richtig eingerückt:

```
for (i = begin; i < end; ++i) {
    for (j = i+1; j <= end; ++j)
        if (arr[i] != arr[j])
            break
    if (j > i + 1) {
        dup_cnt = j - 1 - i
        for (k = j-1; k <= end; ++k)
            arr[k - dup_cnt] = arr[k]
        end -= dup_cnt
    }
}
```

Bei der Definition von Regeln und Funktionen sollte folgende **Reihenfolge** eingehalten werden:



2.5 Programmablauf

2.5.1 Vereinfachte Version

1. Die Initialisierung einiger interner Variablen findet statt.
2. Die Aktion des **BEGIN-Musters** wird ausgeführt (falls vorhanden).
3. Die Eingabedaten werden aus den angegebenen Dateien oder von der Standard-Eingabe zeilenweise eingelesen und verarbeitet:

   (a) **Eine oder mehrere Eingabedateien als Argumente angegeben:**
   Die Variable `FILENAME` erhält den Namen der aktuell eingelesenen Eingabedatei zugewiesen.
   Die einzelnen Eingabedateien werden der Reihe nach Satz für Satz (terminiert durch den aktuellen Satztrenner `RS`) in die Variable `$0` eingelesen.
   Die Variable `FNR` erhält die Anzahl der bisher aus der aktuellen Eingabedatei gelesenen Sätze zugewiesen.

   (b) **Keine Eingabedatei als Argument angegeben:**
   Die Variable `FILENAME` erhält den Wert `"-"` zugewiesen.
   Von der Standard-Eingabe wird Satz für Satz (terminiert durch den aktuellen Satztrenner `RS`) in die Variable `$0` eingelesen.
   Die Variable `FNR` erhält die Anzahl der bisher gelesenen Sätze zugewiesen.

4. Jeder eingelesene Satz wird gemäß dem aktuellen Feldtrenner `FS` in die Felder `$1` bis `$NF` zerlegt und `NF` wird gleich der Anzahl der Felder gesetzt (siehe auch die Abschnitte 2.6.1/2.8 auf den Seiten 26/33).
   Für jeden gelesenen Satz wird der Satzzähler `NR` um 1 erhöht, er entspricht also immer der aktuellen (Gesamt)Anzahl an gelesenen Sätzen.
5. Nachdem ein Satz eingelesen und in Felder zerlegt ist, werden *alle Muster in der Reihenfolge ihrer Definition* überprüft. Ist ein **Muster** (für den aktuellen Eingabesatz) erfüllt, so wird die zugehörige **Aktion** ausgeführt.
   Sind *alle* Muster für den aktuellen Eingabesatz überprüft und eventuell ihre Aktionen ausgeführt worden, wird der nächste Satz eingelesen, in Felder zerlegt, ... (d.h. wieder bei Punkt 3 fortgefahren).
6. Nachdem sämtliche Sätze aus den angegebenen Dateien oder von der Standard-Eingabe eingelesen wurden, wird die Aktion des **END-Musters** ausgeführt (falls vorhanden).

2.5.2 Ergänzungen

zu 1. Die internen Variablen `FS`, `RS`, `OFS`, `ORS`, `OFMT` und `SUBSEP` werden mit ihrem **Defaultwert** belegt (siehe Abschnitt 2.8 auf Seite 33).

Der Programmname und die restlichen Argumente werden im Array `ARGV` abgelegt und ihre Anzahl im Array `ARGC` gezählt, `ARGIND` erhält den Wert 1 zugewiesen.

Die Umgebungsvariablen werden im Array `ENVIRON` abgelegt.

Gemäß der eventuell angegebenen Option `-F` wird die Variable `FS` initialisiert.

Gemäß eventuell angegebener `-v`-Optionen werden die Variablen initialisiert.

Zu jeder `-f`-Optionen wird das folgende Argument als Skriptname interpretiert und die Anweisungen daraus gelesen. Ist keine `-f`-Option angegeben worden, wird das 1. Argument auf der Kommandozeile als *Awk*-Programm interpretiert.

Das jeder `--source`-Option folgende Argument wird als Teil des *Awk*-Programms (in der angegebenen Reihenfolge) interpretiert.

Die in Abschnitt 2.1 auf Seite 12 aufgeführten Optionen werden interpretiert und aus der Kommandozeile entfernt.

zu 2. Mehrere `BEGIN`-Muster werden in der Reihenfolge ihres Auftretens abgearbeitet.

Die Defaultwerte können in der `BEGIN`-Aktion geändert werden, bevor der ersten Satz eingelesen oder ausgegeben wird. Ebenso können die Elemente des Argumentvektors `ARGV` und die Argumentanzahl `ARGC` beliebig geändert, erweitert oder gelöscht werden.

Enthält die `BEGIN`-Aktion einen `exit`-Befehl, so wird das Einlesen abgebrochen und zum `END`-Muster gesprungen (bzw. das *Awk*-Programm beendet, falls kein `END`-Muster vorhanden ist).

zu 3. Die Defaultwerte für die Feld- und Satztrenner sind:

Variable	Default	Bedeutung
`FS`	␣	**Whitespace** = beliebig lange Folge von Leerzeichen und/oder Tabulatoren; am Zeilenanfang/ende vorkommender Whitespace wird ignoriert [**field separator**]
`RS`	`\n`	Zeilenvorschub [**record separator**]

Diese Defaultwerte sind *jederzeit änderbar* (auch mitten während der Verarbeitung der Daten) und gelten dann für die *nächste* eingelesene Zeile.

zu 3b. Hat ein Dateiname die Form var=$text$, so erhält die Variable var den Wert $text$ zugewiesen und der Dateiname wird ignoriert. Diese Zuweisung wird erst zu dem Zeitpunkt ausgeführt, an dem auf das Argument als Datei zugegriffen würde (**Hinweis:** in `BEGIN` sind solche Variablen nicht definiert, auch wenn sie als die ersten Argumente angegeben werden).

zu 5. Enthält die Aktion einen `next`-Befehl, wird der nächste Eingabesatz gelesen und die Verarbeitung beginnt wieder bei der ersten Regel.

Enthält die Aktion einen `nextfile`-Befehl, wird das Einlesen der aktuellen Eingabedatei beendet und mit dem Einlesen der nächsten begonnen. Die Verarbeitung beginnt wieder bei der ersten Regel.

Enthält die Aktion einen `exit`-Befehl, so wird zum `END`-Muster gesprungen (bzw. das *Awk*-Programm beendet, falls kein `END`-Muster vorhanden ist).

zu 6. Mehrere `END`-Muster werden in der Reihenfolge ihres Auftretens abgearbeitet.
`exit` in der `END`-Aktion beendet das *Awk*-Programm.

2.6 Felder, Variablen und Arrays

Die Begriffe **Feld** und **Array** sind im *Awk* nicht wie üblich synonym verwendbar:

- **Feld** bezeichnet die durch die automatische Zerlegung der Eingabezeilen gebildeten Textstücke (**Worte**).
- **Array** hat die üblich Bedeutung einer mehrwertigen Variablen, deren Elemente über **Indices** ansprechbar sind.

2.6.1 Felder

Die aktuelle Eingabezeile wird gemäß dem Feldtrenner `FS` in **Felder** zerlegt (hat z.B. `FS` den Defaultwert ␣, so trennt **Whitespace**, d.h. beliebig lange Folgen von Leerzeichen und/oder Tabulatoren die Felder und am Zeilenanfang/ende vorkommender Whitespace wird ignoriert). Die Felder der aktuellen Eingabezeile sind über die Variablen `$1`, `$2`, ..., `$NF` ansprechbar, die Variable `$0` enthält die ganze Eingabezeile. Felder können wie Variablen behandelt werden, d.h. sie können in numerischen oder Zeichenketten-Ausdrücken verwendet werden und ihnen sind auch Werte zuweisbar.

Variable	Bedeutung
`$0`	Aktueller Eingabesatz
$\$n$	Feld n des aktuellen Eingabesatzes (`$1..$NF`)

- Wird `$0` durch Zuweisung oder Zeichenketten-Funktionen verändert, so werden die Felder `$1`, `$2`, ... und die Variable `NF` gemäß dem aktuellen Wert von `FS` neu erstellt.
- Wird eines der Felder `$1`, `$2`, ... verändert, dann wird `$0` mit `OFS` zur Trennung der Felder neu erstellt. **Hinweis:** Dabei kann Whitespace am Zeilenanfang, am Zeilenende und zwischen Feldern verschwinden.
- Auf Felder kann auch durch Ausdrücke zugegriffen werden, z.B. ist `$(NF-1)` das vorletzte Feld (die Klammern sind notwendig, `$NF-1` wäre der Wert des letzten Feldes minus eins).
- Ein nicht existierendes Feld wie z.B. `$(NF+1)` hat `0/""` als Startwert, es kann durch eine Zuweisung erzeugt werden.

2.6.2 Variablen

Ein Variablenname wird wie in *C* gebildet (`[A-Za-z_][A-Za-z_0-9]*`), Groß/Kleinschreibung wird beachtet. Beispiel:

```
v  v123  eine_variable  EineVariable  _var_  _VAR_  _Var_
```

Alle Variablen sind **global**, wenn sie nicht ein Parameter (**lokale Variable**) in einer Funktion sind. Der Datentyp von Variablen wird nicht unterschieden.

Variablen müssen nicht deklariert werden, sie können jeweils eine **Zahl** oder eine **Zeichenkette** enthalten. Ist der *erste Zugriff* auf eine Variable lesend, so hat sie automatisch den Wert `0/""`.

2.6.3 Arrays

Ein Arrayname wird wie in *C* gebildet (`[A-Za-z_][A-Za-z_0-9]*`), Groß/Kleinschreibung wird beachtet.

Alle Arrays sind **global**, wenn sie nicht durch einen gleichnamigen Parameter (**lokale Variable**) in einer Funktion überdeckt werden.

Arrays müssen nicht deklariert werden, sie werden *automatisch erweitert*, wenn auf ein neues Element (lesend oder schreibend) zugegriffen wird. Jedes Arrayelement entspricht einer Variablen (d.h. es kann eine beliebige Mischung aus Zahlen und Zeichenketten enthalten). Ist der erste Zugriff auf ein Arrayelement lesend, so hat es automatisch den Wert `0/""`.

Der Zugriff auf die Elemente eines Arrarys erfolgt über einen durch `[...]` eingeschlossen **(Array)Index**, als Index können **Zahlen und Zeichenketten** verwendet werden. Beispiel:

```
arr[123]   = "Text1"
arr["abc"] = "Text2"
```

Arrays können **mehrdimensional** sein, die Index-Komponenten werden durch `,` getrennt (*nicht wie in C durch mehrfache Angabe von* `[]...[]`). Mehrdimensionale Indices werden in eine Zeichenkette umgewandelt, wobei das Komma in das Zeichen `SUBSEP` (**Default:** `"\034"`) umgesetzt wird (es gibt keine Arrays von Arrays). Beispiel:

```
i = "A"; j = "B"; k = "C"
arr[i,j,k] = "hello world\n";   # Element "A\034B\034C" belegen
```

Ein Array hat also keine bestimmte Dimension, sondern die Indices können z.B. auch **gemischtdimensional** sein. In den folgenden Beispielen ist immer auch eine mehrdimensionale Form angegeben (falls sie möglich ist).

- Ein Zugriff auf das Arrayelement mit dem Index `i(,j,k)` im Array `arr` hat die Form:

  ```
  arr[i]
  arr[i,j,k]
  ```

  wobei `i(j/k)` eine beliebige Zeichenkette sein kann. Der Spezialfall der Indizierung über ganze Zahlen `0..n` wie in *C* ist darin natürlich enthalten. Die Zugriffe erfolgen sehr schnell per **Hash-Funktion**.

- Sämtliche Indices der aktuell vorhandenen Elemente eines Arrays lassen sich mit:

```
for (i in arr) ...
for (i,j,k in arr) ... # geht nicht!
```

aufzählen (*allerdings in keiner bestimmten Reihenfolge*). Bei mehrdimensionalen Arrays darf nur *ein* Index `i` angegeben werden, der anschließend per `split` in seine Komponenten zerlegt und dann erst mit mit Hilfe von `tmp[1]..tmp[n]` weiterverarbeitet werden kann:

```
for (i in arr) {
    n = split(i, tmp, SUBSEP)
    ...
}
```

- Die Existenz eines Arrayelementes `i(j/k)` kann durch:

```
if (i in arr) ...
if ((i,j,k) in arr) ... # ok!
```

überprüft werden, *ohne* das Element anzulegen.

- Einzelne Arrayelemente `i(j/k)` werden gelöscht durch:

```
delete arr[i]
delete arr[i,j,k]
```

- Sämtliche Elemente eines Arrays `arr` können auf einen Schlag gelöscht werden durch:

```
delete arr
```

2.6.4 Datentypen, Datentypumwandlung und Defaultwerte

- Der **Typ** eines Feldes, einer Variablen oder eines Arrayelementes kann **num** (Zahl), **str** (Zeichenkette) oder **strnum** (Mischung aus beidem) sein. D.h. jedes Feld, jede Variable und jedes Arrayelement kann eine Zahl, eine Zeichenkette oder beides zugleich enthalten:
  - ▷ Eine Zahlenkonstante oder das Ergebnis eines numerischen Ausdrucks hat den Typ **num**.
  - ▷ Eine Zeichenkettenkonstante oder das Ergebnis einer Zeichenkettenoperation hat den Typ **str**.
  - ▷ Felder, `getline`-Eingaben, `FILENAME`, `ARGV`-Elemente, `ENVIRON`-Elemente und von `split` erzeugte Arrayelemente haben den Typ **strnum**, wenn sie wie Zahlen aussehen, sonst haben sie den Typ **str**.

  Die Grundidee dahinter ist: Alle **Benutzereingaben**, die wie Zahlen aussehen — und zwar ausschließlich diese — werden auch wie Zahlen behandelt (auch wenn sie aus Zeichen bestehen und daher eigentlich Zeichenketten darstellen).

- Wird einer Variablen durch eine Zuweisung:

```
VAR = EXPR
```

  ein Wert zugewiesen, so wird der **Typ** der Variablen `VAR` gleich dem Typ des Ausdrucks `EXPR` gesetzt (Zuweisung schließt `+=` `-=` `...` ein). Ein *arithmetischer Ausdruck* hat den Typ **Zahl**, eine *Verkettung* hat den Typ **Zeichenkette**, usw.

- Ist eine Anweisung eine einfache Kopie, wie in:

```
VAR1 = VAR2
```

  so wird der Typ von `VAR1` gleich dem Typ von `VAR2` gesetzt. (d.h. der Typ einer Variablen wird durch Zuweisung weitergegeben).

- Durch Verwendung einer Variablen wird ihr Typ nicht verändert.

- Der numerische Wert einer beliebigen Zeichenkette ist der Wert ihres **numerischen Präfixes** (*führende Leerzeichen werden ignoriert*). Beispiel:

```
"123abc" + 1   -> 124
```

- Der **Typ eines Feldwertes** wird (wenn möglich) aus dem Kontext ermittelt; zum Beispiel führt:

```
$1++
```

  dazu, daß der Inhalt von `$1` (wenn nötig) in eine Zahl umgewandelt wird, und:

```
$1 = $1 "," $2
```

  führt dazu, daß die Inhalte von `$1` und `$2` (wenn nötig) in Zeichenketten umgewandelt werden.

- In Kontexten, in denen der Typ einer Variablen nicht sicher festgestellt werden kann, wie z.B. bei der Auswertung von:

```
if ($1 == $2) ...
```

  beschreibt folgende Tabelle den **Typ des Vergleichs** abhängig vom Typ der Argumente:

	str	**num**	**strnum**
str	str	str	str
num	str	num	num
strnum	str	num	num

- Der **Typ eines Ausdrucks** kann durch Tricks wie:

```
EXPR + 0
```

zu numerisch und durch:

```
EXPR ""
```

(d.h. Verkettung mit der leeren Zeichenkette) zu Zeichenkette umgewandelt werden.

- **Undefinierte Variablen** haben den numerischen Wert `0` und den Zeichenkettenwert `""`. Ist die Variable `x` undefiniert, so ist daher:

```
if (x) ...
```

falsch und:

```
if (!x) ...
if (x == 0) ...
if (x == "") ...
```

sind alle wahr. Bitte beachten, daß in diesem Fall:

```
if (x == "0") ...
```

falsch ist. Nach jeder dieser 5 Anweisungen ist `x` definiert und hat den Wert `0/""`.

- Die Verwendung eines Arrayelements in einem Ausdruck macht es existent, mit dem Wert `0/""`. Falls daher `arr[i]` aktuell nicht existiert, führt:

```
if (arr[i] == "") ...
```

zu seiner Existenz mit dem Wert `0/""` und die `if`-Bedingung ist daher erfüllt. Der Test:

```
if (i in arr) ...
```

ermittelt hingegen, ob `arr[i]` existiert, *ohne* das Element als Seiteneffekt zu erzeugen.

2.7 Operatoren

Folgende Operatoren sind vorhanden (nach fallendem Vorrang, `*` = in *C* nicht vorhanden, r = rechts-assoziativ, sonst links-assoziativ):

!C	Operator	Bedeutung
	`(...)`	Klammerung
*	`$`	Feld-Zugriff (`$1`, ...)
	`++ --`	Inkrementieren, Dekrementieren (*Präfix* und *Postfix*)
*r	`^ **`	Exponentiation x^y
	`+ - !`	*Unäres* Plus, *unäres* Minus, logisch NICHT
	`* / %`	Multiplikation, Division, Modulo (Divisionsrest)
	`+ -`	Addition, Subtraktion
*	*(space)*	Verkettung von Zeichenketten (*kein expliziter Operator!*)
	`< >`	Vergleich (*Zahlen und Zeichenketten*)
	`<= >=`	Vergleich (*Zahlen und Zeichenketten*)
	`== !=`	Vergleich (*Zahlen und Zeichenketten*)
*	`~ !~`	Vergleich mit Regulärem Ausdruck, negiert (*matched by*)
*	`in`	Index in Array testen
	`&&`	Logisch UND (*short-cut evaluation*)
	`||`	Logisch ODER (*short-cut evaluation*)
r	`?:`	Bedingte Operation ($cond$ `?` $true\text{-}case$ `:` $false\text{-}case$)
r	`= += -= *= /= %= ^= **=`	Zuweisung (inkl. Operation `+ - * / % ^ **`)

- Jeder Ausdruck darf geklammert werden, Klammern haben den höchsten Vorrang.
- Der Operator `$` greift auf die Feldinhalte (Worte) der aktuellen Eingabezeile zu (hier 1. Feld, letztes Feld, vorletztes Feld):

```
$1    $NF    $(NF-1)
```

- Die Operatoren `++` bzw. `--` inkrementieren bzw. dekrementieren den zugehörigen Variablenwert um den Wert 1 *vor* (in der Prefix-Form `++i`) bzw. *nach* (in der Postfix-Form `i++`) der Verwendung des Wertes:

```
++i   # i inkrementieren, dann Wert von i verwenden
i++   # Wert von i verwenden, dann i inkrementieren
--i   # i dekrementieren, dann Wert von i verwenden
i--   # Wert von i verwenden, dann i dekrementieren
```

- Die Operatoren `^` und `**` führen eine Exponentiation x^y durch.
- Die unären Vorzeichen-Operatoren `+` und `-` stehen direkt vor Zahlen und kennzeichnen ihr Vorzeichen.
- Der logische NICHT-Operator `!` kehrt den Wahrheitswert des folgenden Ausdrucks um.
- Die Operatoren `*`, `/` und `%` multiplizieren, dividieren und bilden den Restwert einer Division.
- Die Operatoren `+` und `-` addieren und subtrahieren.
- Für das mit *(space)* bezeichnete **Konkatenieren** (Aneinanderhängen) von Zeichenketten gibt es keinen Operator, diese Operation wird einfach durch Hintereinanderschreiben der zu verkettenden Zeichenketten oder Variablen (**Hinweis:** möglichst durch ein Leerzeichen getrennt) ausgedrückt:

```
var = "Dies" "ist" "ein" "Text."   ->  var = "DiesisteinText."
var = "Die Zahl ist " 12.3         ->  var = "Die Zahl ist 12.3"
var = 10; print "var=<" var ">"    ->  "var=<10>"
var = 10; print "var=<", var, ">"  ->  "var=< 10 >" (OFS=" ")
```

- Die Operatoren <, >, <=, >=, == und != vergleichen numerisch oder textuell, je nach Typkombination der verglichenen Werte (siehe Abschnitt 2.6.4 auf Seite 29).
- Die Operatoren ~ und !~ vergleichen einen Text mit einem Regulären Ausdruck, sie lassen sich als **matched by** und **not matched by** lesen:

```
if ($0 ~ /abc/) ...      # $0 matched by "abc"
if ($0 !~ /xyz/) ...     # $0 not matched by "xyz"
```

- Der Operator in prüft, ob ein Array einen bestimmten Index hat oder nicht, d.h. ob das entsprechende Arrayelement existiert oder nicht existiert:

```
if (123 in arr) ...
if (!("abc" in arr)) ...   # nicht (!"abc" in arr) !!!
```

- Die Operatoren && (logisch UND) und || (logisch ODER) führen eine **short-cut evaluation** (verkürzte Auswertung) durch. Wenn der Wert des gesamten Ausdrucks bereits nach der Auswertung des ersten Elements (0 bei UND und 1 bei ODER) bekannt ist, dann werten sie den zweiten Operanden *nicht* mehr aus:

```
0 && system("echo aaa")   # Ausgabe findet NICHT statt
1 && system("echo bbb")   # Ausgabe findet statt
0 || system("echo ccc")   # Ausgabe findet statt
1 || system("echo ddd")   # Ausgabe findet NICHT statt
```

- Der Operator ?: (bedingte Operation, Syntax $var\ =\ cond\ ?\ true\text{-}case\ :\ false\text{-}case$) ist eine kompakte Form der folgenden if-else-Anweisung (im Unterschied dazu ist er innerhalb von Ausdrücken direkt verwendbar und der $false\text{-}case$ kann nicht weggelassen werden):

```
if (COND)
    var = TRUE-CASE
else
    var = FALSE-CASE
```

- Die Operatoren += -= *= /= %= ^= **= stellen eine kompakte Form für Operationen auf einer Variablen zur Verfügung, indem sie die entsprechende Operation mit einer Zuweisung verbinden:

```
var += 10   # entspricht: var = var + 00
```

- Alle Operatoren außer Zuweisung, ?:, ^ und ** sind *linksassoziativ*, d.h. sie werden bei mehrfachem Vorkommen von links nach rechts ausgewertet:

```
4 - 3 - 2 = ((4 - 3) - 2) = 1 - 2 = -1        linksassoziativ
4 ^ 3 ^ 2 = (4 ^ (3 ^ 2)) = 4 ^ 9 = 262144    rechtsassoziativ
```

- **Hinweis:** Die *Zuweisungs-Operatoren* ~= &= |= <<= >>=, die *Bit-Operatoren* ~ & ^ |, die *Shift-Operatoren* << >> und der *Komma-Operator* , aus *C* existieren nicht.

2.8 Vordefinierte Variablen und Arrays

Folgende Variablen und Arrays sind vordefiniert, ihre Namen stellen jeweils eine sinnvolle Abkürzung für die Bedeutung ihres Inhalts dar (* = nur im *Gawk* vorhanden):

Gawk	Variable	Bedeutung
	ARGC	Anzahl Kommandozeilenargumente (Programmname in ARGV[0])
*	ARGIND	Index der aktuell eingelesenen Eingabedatei in ARGV[1..ARGC-1]
	ARGV	Array der Kommandozeilenargumente (ARGV[0..ARGC-1])
*	CONVFMT	Format f. autom. Umwandlung Zahl → Zeichenkette (**Default:** "%.6g")
*	ENVIRON	Assoziatives Array mit Umgebungsvariablen und ihrem Wert
*	ERRNO	Systemfehlermeldung bei getline und close
*	FIELDWIDTHS	Liste mit durch Leerz. getrennten festen Spaltenbreiten (statt FS)
	FILENAME	Name der aktuellen Eingabedatei
	FNR	Satznummer der aktuellen Eingabedatei
	FS	Eingabefeldtrenner (**Default:** "␣" = Whitespace)
*	IGNORECASE	Groß/Kleinschreibung beim Vergleich mit Regulärem Ausdruck und in Zeichenketten-Funktionen ignorieren (**Default:** *false*)
	NF	Anzahl Felder im aktuellen Satz
	NR	Anzahl bisher eingelesener Sätze ingesamt
	OFMT	Ausgabeformat für Zahlen in print (**Default:** "%.6g")
	OFS	Ausgabe-Feldtrenner in print (**Default:** "␣")
	ORS	Ausgabe-Satztrenner in print (**Default:** "\n")
	RLENGTH	Länge des passenden Textes nach match (-1 falls nicht gefunden)
	RS	Eingabesatztrenner (**Default:** "\n")
	RSTART	Beginn des passenden Textes nach match (0 falls nicht gefunden)
*	RT	Zu RS passender Text nach dem Einlesen jedes Satzes
	SUBSEP	Trennzeichen für mehrdim. Index [i,j,...] (**Default:** "\034")

- ARGC (**argument count**) — enthält die Anzahl Kommandozeilenargumente einschließlich dem Namen des aufgerufenen Programms in ARGV[0]. Kann jederzeit verändert werden.
- ARGIND (**argument index**) — enthält den Index der aktuell eingelesenen Eingabedatei, es gilt immer FILENAME == ARGV[ARGIND].
- ARGV[0] (**argument vector**) — enthält den Namen des aufgerufenen Programms (ohne Pfad, normalerweise awk), aber nicht sein(e) Programmargument(e) (-f FILE) oder seine Optionen (-F/-v).

  ARGV[1]..ARGV[ARGC-1] — enthalten die beim Programmaufruf angegebenen Argumente (Dateinamen). Sie können jederzeit verändert/gelöscht/erweitert werden (z.B. im BEGIN-Abschnitt).
- CONVFMT (**conversion format**) — bestimmt das Format, mit dessen Hilfe Zahlen in Zeichenketten konvertiert werden, falls dies für die automatische Typumwandlung benötigt wird (**Default:** "%.6g").
- ENVIRON (**environment (variables)**) — enthält zu allen Umgebungsvariablen deren Wert (z.B. kann ENVIRON["PATH"] den Wert "/bin:/usr/bin/:/usr/local/bin:." enthalten).

- `ERRNO` (**error number**) — enthält die Systemfehlermeldung (*nicht die Nummer!*) bei fehlerhaftem Aufruf der Funktionen `getline` und `close`.
- `FIELDWIDTHS` — kann eine durch Leerzeichen getrennte Liste von Spaltenbreiten enthalten. Dann werden die Feldinhalte nicht durch `FS`, sondern durch diese festen Spaltenbreiten bestimmt. Durch eine Zuweisung an `FS` (z.B. `FS = FS`) wird wieder das Standard-Verhalten „Trennung per `FS`“ eingeschaltet.
- `FILENAME` — enthält den Namen der aktuellen Eingabedatei. Hat den Wert `"-"` (Standard-Eingabe), wenn keine Eingabedatei auf der Kommandozeile angegeben wurde. Ist in `BEGIN` undefiniert und enthält in `END` den Namen der letzten eingelesenen Datei.
- `FNR` (**file number of records**) — enthält die Anzahl der bisher aus der aktuellen Eingabedatei gelesenen Sätze.
- `FS` (**field separator**) — trennt die Felder der Eingabezeile und kann durch folgende Werte definiert werden (**Default:** `"␣"`):
  - ▷ Der Defaultwert `"␣"` (ein Leerzeichen) sorgt dafür, daß Folgen von **Whitespaces** (Leerzeichen und Tabulator) die Felder trennen und am Zeilenanfang und -ende ignoriert werden.
  - ▷ Ein Einzelzeichen (oder ein Regulärer Ausdruck) wird selbst (oder die zu diesem Regulären Ausdruck passenden Textteile) als Feldtrenner verwendet. Kommt dieses Zeichen (oder paßt dieser Reguläre Ausdruck) am Zeilenanfang, mehrfach hintereinander oder am Zeilenende vor, so werden leere Felder erzeugt.

    Hinweis: Soll genau ein Leerzeichen als Feldtrenner verwendet werden, so ist `FS = "[␣]"` zu setzen.
  - ▷ Der Wert `""` (leere Zeichenkette) sorgt dafür, daß die Zeichen der Eingabezeile einzeln in je einem Feld abgelegt werden.
- `IGNORECASE` — belegt mit dem Wert *wahr* führt dazu, daß die Groß/Kleinschreibung beim Vergleich mit Regulären Ausdrücken oder in Zeichenketten-Funktionen ignoriert wird. Dies betrifft die Operatoren `~` und `!~`, die Funktionen `gensub`, `gsub`, `index`, `match`, `split`, `sub`, den Feldtrenner `FS` und den Satztrenner `RS`. (**Default:** *false*):
- `NF` (**number of fields**) — enthält für jede Eingabezeile die Anzahl der Felder.
- `NR` (**number of records**) — enthält die Anzahl der bisher insgesamt eingelesenen Sätze.
- `OFMT` (**output format**) — legt das Format fest, in dem Zahlen in `print`-Anweisungen ausgegeben werden (**Default:** `"%.6g"`).
- `OFS` (**output field separator**) — ist der Ausgabe-Feldtrenner für die durch `,` (Komma) getrennten Argumente in `print`-Anweisungen, er wird automatisch zwischen allen Argumenten einer `print`-Anweisung ausgegeben (**Default:** `"␣"`).

- `ORS` (**output record separator**) — ist der Ausgabe-Satztrenner für `print`-Anweisungen, er wird automatisch als Abschluß einer `print`-Anweisung ausgegeben (**Default:** `"\n"`).
- `RS` (**record separator**) — trennt die Eingabedaten in Sätze auf und kann durch folgende Werte definiert werden (**Default:** `"\n"`):
  - ▷ Ein Einzelzeichen (oder ein Regulären Ausdruck) als Wert sorgt dafür, daß dieses (oder die zu diesem Regulären Ausdruck passenden Textteile) als Satztrenner verwendet werden.
  - ▷ Der Wert `""` (leere Zeichenkette) sorgt dafür, daß eine oder mehrere *Leerzeilen* die Datensätze trennen. Damit können **mehrzeilige Records** verarbeitet werden. Neben Whitespace ist dann automatisch `"\n"` ein zusätzlicher Feldtrenner.
- `RSTART` (**regexp start**) und `RLENGTH` (**regexp length**) — werden von der Standard-Funktion `match(`s`,`r`)` belegt, die ein zum Regulären Ausdruck r passendes Textstück in s sucht. `RSTART` enthält anschließend den Startindex des passenden Textes (`1..n`) oder `0`, `RLENGTH` enthält die Länge des passenden Textes (`1..n`) oder `-1`, `RSTART` wird von `match` auch als Wert zurückgegeben. Beispiel:

  ```
  match("Testtext", /e.t+e/)
  ```

  ergibt `RSTART=2` und `RLENGTH=5` (`estte`).
- `RT` (**record (separator) text**) — ist nach jedem Einlesen eines Eingabesatzes mit der aktuell zu `RS` gefundenen Zeichenkette belegt.
- `SUBSEP` (**subindex separator**) — enthält das Trennzeichen, in das die Kommas eines mehrdimensionalen Array-Indexes umgewandelt werden, um eine Zeichenkette als Index zu erhalten (**Default:** `"\034"`):

  ```
  i = "A"; j = "B"; k = "C"
  arr[i,j,k] = "hello world\n";   # Element "A\034B\034C" belegen
  ```

2.9 Vordefinierte Funktionen

2.9.1 Arithmetik-Funktionen

x und y sind beliebige numerische Ausdrücke, in eckige Klammern eingeschlossene Parameter können weggelassen werden:

Funktion	Bedeutung
`sin(`x`)`	Sinus von x (x in Radiant)
`cos(`x`)`	Cosinus von x (x in Radiant)
`atan2(`y`,`x`)`	Arcustangens von y/x im Bereich $-\pi$ bis π (in Radiant)
`exp(`x`)`	Exponentialfunktion e^x von x
`log(`x`)`	Natürlicher Logarithmus (Basis e) von x
`int(`x`)`	Ganzzahliger Teil von x
`sqrt(`x`)`	Quadratwurzel von x
`rand()`	Zufallszahl x, mit $0 \leq x < 1$
`srand([`x`])`	x ist neuer Startwert von `rand`, gibt alten Startwert zurück (verwendet Uhrzeit ohne x)

2.9.2 Zeichenketten-Funktionen

s und t sind Zeichenketten, r ein Regulärer Ausdruck, n und i sind Zahlen, fmt ist eine Zeichenkette mit `%`-Formatierungsangaben (siehe Abschnitt 2.9.4 auf Seite 38), $list$ ist eine durch Kommas getrennte Liste von Argumenten, in eckige Klammern eingeschlossene Parameter können weggelassen werden (`*` = nur im *Gawk* vorhanden):

Gawk	Funktion	Bedeutung
*	`gensub(`r`,`s`[,`h`[,`t`]])`	Ersetzt r in t durch s, gibt Ergebnis zurück; h=`"g/G"`(global) oder Zahl(=Position), sonst nur 1. Position; t bleibt unverändert, ohne t wird `$0` verwendet; `()` ist in r und `\1-\9` ist in s verwendbar
	`gsub(`r`,`s`[,`t`])`	Ersetzt r überall in t durch s; gibt Anzahl Ersetzungen zurück ohne t wird `$0` verwendet [**global**]
	`index(`s`,`t`)`	Gibt erste Position (1..n) von t in s zurück; oder 0 wenn t nicht in s enthalten ist
	`length([`s`])`	Gibt Länge von s zurück, ohne s wird `$0` verwendet
	`match(`s`,`r`)`	Überprüft, ob s eine zu r passende Teilkette enthält; gibt Position (1..n) oder 0 wenn r nicht auf s paßt zurück; belegt `RSTART` und `RLENGTH`
*	`ord(`s`)`	Liefert ASCII-Code der 1. Zeichens von s
	`split(`s`,`a`[,`r`])`	Zerlegt s gemäß r in Arrayelemente, gibt ihre Anzahl zurück; legt diese in den Elementen 1 bis n des Arrays a ab; ohne r wird `FS` verwendet; r=`""` zerlegt s in Buchstaben
	`sprintf(`fmt`,`$list$`)`	Gibt $list$ formatiert gemäß fmt zurück (siehe `printf`)
	`sub(`r`,`s`[,`t`])`	Wie `gsub`, ersetzt aber nur die erste Teilkette
	`substr(`s`,`i`[,`n`])`	Gibt Teilstück von s der Länge n ab Position i zurück; ohne n wird der Suffix ab Position i zurückgegeben; das erste Zeichen hat die Position 1
*	`toupper(`s`)`	Wandelt s in Großschreibung um
*	`tolower(`s`)`	Wandelt s in Kleinschreibung um

- Wird bei `match` keine passende Zeichenkette gefunden, so enthält `RSTART` den Wert 0 und `RLENGTH` den Wert `-1`.
- `split` eignet sich sehr gut zum Initialisieren von Arrays, ein Array `arr` kann z.B. durch `n = split("mo di mi do fr sa so", arr)` mit den 7 Werten `"mo"`,

`"di"`, `"mi"`, `"do"`, `"fr"`, `"sa"`, `"so"` vorbelegt werden, die Indices laufen dann von `1..7`, die Variable `n` erhält den Wert 7.

- In `gensub`, `gsub` und `sub` wird ein `&` oder `\0` in der Ersetzungszeichenkette s durch die zum Suchmuster passende Zeichenkette ersetzt, `\&` in der Ersetzungszeichenkette ergibt das wörtliche Ampersand.

2.9.3 Ein/Ausgabe-Funktionen

$file$ ist eine Datei (muß in Anführungszeichen stehen, kann auch eine Textvariable sein), cmd ist ein Betriebssystem-Kommando (BS) (muß in Anführungszeichen stehen), var ist eine Variable, fmt ist eine Zeichenkette mit `%`-Formatierungsangaben (siehe Abschnitt 2.9.4 auf Seite 38), $list$ ist eine durch Kommas getrennte Liste von Argumenten, in eckige Klammern eingeschlossene Parameter können weggelassen werden (`*` = nur im *Gawk* vorhanden):

Gawk	Funktion	Bedeutung
	`close(`$file$`)`	Datei $file$ schließen
	`close(`cmd`)`	Kommando cmd (Pipe) schließen
*	`fflush([`$file$`])`	Gepufferte Daten der Datei $file$ schreiben
*	`fflush([`cmd`])`	Gepufferte Daten des Kommandos cmd schreiben
	`getline`	`$0` mit nächster Eingabezeile belegen; setzt `NF`, `NR`, `FNR`
	`getline <` $file$	`$0` mit nächster Eingabezeile aus $file$ belegen; setzt `NF`
	`getline` var	var mit nächster Eingabezeile belegen; setzt `NR`, `FNR`
	`getline` var `<` $file$	var mit nächster Eingabezeile aus $file$ belegen
	cmd `|` `getline`	Ergebnis von BS-Kommando cmd in `$0` ablegen; setzt `NF`
	cmd `|` `getline` var	... in var ablegen
	`print`	Aktuelle Eingabezeile `$0` ausgeben (mit Return)
	`print` $list$	Ausdrücke $list$ ausgeben
	`print` $file$ `>` $file$	... auf $file$ ausgeben ($file$ vorher löschen)
	`print` $list$ `>>` $file$	... an $file$ anhängen ($file$ vorher *nicht* löschen)
	`print` $list$ `|` cmd	... an BS-Kommando cmd übergeben
	`printf` fmt`,`$list$	Ausdrücke $list$ gemäß fmt formatieren und ausgeben
	`printf` fmt`,`$list$ `>` $file$	... und auf $file$ ausgeben
	`printf` fmt`,`$list$ `>>` $file$	... und an $file$ anhängen
	`printf` fmt`,`$list$ `|` cmd	... und an BS-Kommando cmd übergeben
	`system(`cmd`)`	BS-Kommando cmd ausführen, Exit-Status zurückgeben

- Auf die von einem *Awk*-Programm erzeugte Dateien kann im gleichen *Awk*-Programm erst dann zugegriffen werden, wenn sie mit `close` geschlossen (oder mit `fflush` definitiv geschrieben) wurden (da das Betriebssystem Ausgaben auf eine Datei oder Pipe zwischenpuffert und die Ausgabe erst dann sicher vollständig ist).
- Nur eine begrenzte Anzahl von Dateien/Pipes kann gleichzeitig geöffnet werden (d.h. nicht mehr benutzte Dateien/Pipes mit `close` schließen).
- `fflush("")` leert den Schreib-Puffer der Standard-Ausgabe, `fflush()` leert die Schreib-Puffer aller geöffneten Dateien oder Pipes.

- Die Funktion `getline` liest eine Zeile aus der aktuellen Eingabedatei, aus einer beliebigen Datei oder der Ausgabe eines UNIX-Kommandos in die Variable `$0` oder eine frei wählbare Variable ein. Sie gibt als Ergebnis zurück:

Code	Bedeutung
`1`	Beim erfolgreichen Lesen eines Satzes
`0`	Am Dateiende
`-1`	Bei Fehler (z.B. Datei existiert nicht)

  Das übliche Idiom zum Einlesen von Dateien per `getline` ist daher (wird der Vergleich `> 0` vergessen, so entsteht eine *Endlosschleife*, wenn die Datei nicht existiert):

```
while ((getline < "FILE") > 0)
    ...
```

  Das folgende Beispiel liest die Dateinamen des aktuellen Verzeichnisses sortiert nach dem Datum der letzten Änderung ein (Ergebnis des UNIX-Kommandos `ls -t`):

```
while (("ls -t" | getline) > 0)
    ...
```

- Die auf `print` folgende Argumentliste $list$ bzw. das auf `printf` folgenden Format fmt + die Argumentliste $list$ können auch in Klammern gesetzt werden (vorzugsweise bei `print` keine Klammern und bei `printf` Klammern verwenden). Beispiel:

```
print "Dies", "ist", "ein", "Beispiel"
printf("%d Beispiele\n", cnt)
```

- Bei Ausgaben mit `print` werden durch `,` (Komma) getrennte Argumente durch den Ausgabe-Feldtrenner `OFS` (**Default:** `"␣"`) getrennt und der ganze Satz wird mit dem Ausgabetrenner `ORS` (**Default:** `"\n"`) abgeschlossen. Die Werte von `OFS` und `ORS` sind *jederzeit änderbar* (auch mitten während der Verarbeitung der Daten) und gelten dann für die nächste auszugebende Zeile.

- `system` führt das angegebene Betriebssystem-Kommando cmd aus und gibt seinen Exit-Code zurück. Die Ausgabe von cmd wird in die Ausgabe des *Awk* eingefügt.

2.9.4 Printf-Formatumwandlung

Folgende `%`-Formatumwandlungen sind in `printf`- und `sprintf`-Anweisungen möglich:

Format	Bedeutung
`%c`	Zeichen [**character**]
`%d`/`i`	Dezimalzahl [**decimal**/**integer**]
`%e`/`E`	Gleitkommazahl `[-]d.dddddde[+-]dd` [**exponent**]; `e` klein oder `E` groß geschrieben
`%f`	Gleitkommazahl `[-]ddd.dddddd` [**float**]
`%g`/`G`	Verwendet `e`/`E`- oder `f`-Umwandlung, je nachdem welche kürzer ist; nicht signifikante Nullen werden unterdrückt [**general**]
`%o`	Oktalzahl ohne Vorzeichen [**octal**]
`%s`	Zeichenkette [**string**]
`%x`/`X`	Hexadezimalzahl ohne Vorzeichen [**hexadecimal**]; `a-f` klein oder `A-F` groß geschrieben
`%%`	Gibt ein `%` aus; verwendet kein Argument

Zwischen dem `%`-Zeichen und dem Formatbezeichner können folgende zusätzlichen Parameter stehen (in genau dieser Reihenfolge):

Parameter	Bedeutung
`-`	Ausdruck linksbündig ausrichten
␣	Leerzeichen vor positiven Zahlen, `-` vor negativen
`+`	`+` vor positiven Zahlen, `-` vor negativen
`#`	Alternative Form bei: `o` = Führende Null ausgeben `x`/`X` = Führendes `0x` oder `0X` ausgeben `eEf` = Dezimalpunkt immer ausgeben `gG` = Nullen am Ende nicht entfernen
$width$	Ausgabe auf Breite $width$ ausdehnen (mit Leerzeichen), eine führende `0` verwendet dazu Nullen
`.`$prec$	Anzahl Nachkommastellen $prec$ bei Zahlen, maximale Breite $prec$ bei Zeichenketten

- Anstelle von $width$ oder $prec$ ist auch ein `*` erlaubt, er wird durch den Wert des entsprechenden Parameters in der Parameterliste ersetzt (nur im *Gawk*). Damit können die Breite und die Anzahl Nachkommastellen von Zahlenausgaben *dynamisch* gesteuert werden.
- **Hinweis:** Da *Awk* keine unterschiedlichen Integer- und Gleitkomma-Datentypen wie *C* kennt, sondern alle Zahlen als `double` abgelegt werden, ist bei `printf` keine Unterscheidung in `f`/`lf` und `h`/`l` notwendig.

2.9.5 Zeit-Funktionen

Die Zeit-Funktionen sind *Gawk*-spezifisch, sie sind vor allem dazu gedacht, Zeiten zu vergleichen bzw. Datums- und Zeiteinträge in Log-Dateien zu machen. Die Zeit wird im **UNIX-Timestamp**-Format dargestellt, dies ist die Anzahl Sekunden seit `1.1.1970 1:00 UTC`.

Funktion	Bedeutung
`systime()`	Aktuelle Zeit in Sekunden (sec) seit 1.1.1970 1:00 UTC
`strftime(`[fmt[`,` sec]]`)`	Sekunden sec gemäß fmt formatieren Defaultformat: `"%a %b %d %H:%M:%S %Z %Y"` Defaultzeit: `systime()`

Folgende Formatangaben sind in fmt möglich:

Format	Bedeutung
`%A` / `%a`	Wochentagname vollständig / abgekürzt auf 3 Zeichen
`%B` / `%b`	Monatsname vollständig / abgekürzt auf 3 Zeichen
`%c`	Datum + Zeit gemäß lokalem Standard
`%d`	Monatstag (01-31)
`%H` / `%I`	24 / 12-Stunde (00-23)
`%j`	Jahrestag (001-366)
`%m`	Monat (01-12)
`%M`	Minute (00-59)
`%p`	AM/PM
`%S`	Sekunden (00-61, wegen Korrektursekunden)
`%U` / `%W`	Wochennummer (00-53, erster Sonntag/Montag ist erster Wochentag 1)
`%w`	Wochentagnummer (0-6, 0=Sonntag)
`%X` / `%x`	Zeit / Datum gemäß lokalem Standard
`%Y` / `%y`	Jahr 4- / 2-stellig (1999/00-99)
`%Z`	Zeitzone
`%%`	`%` wörtlich

2.10 Begrenzungen

Jede konkrete Implementierung von *Awk* kennt einige Begrenzungen. Typische Werte sind (dem *Awk*-Buch entnommen; moderne Implementierungen, insbesondere der *Gawk*, haben weit großzügigere Begrenzungen):

Grenze	Bedeutung
100	Felder
3000	Zeichen pro Eingabesatz
3000	Zeichen pro Ausgabesatz
1024	Zeichen pro Feld
3000	Zeichen pro `printf`-Zeichenkette
400	Zeichen maximale Zeichenkettenlänge
400	Zeichen in einer Zeichenliste
15	Offene Dateien gleichzeitig
1	Offene Pipes gleichzeitig
	Doppeltgenaue Gleitkommaarithmetik

3 Tips und Tricks

3.1 Aufrufparameter

- Argumente der Form *var=text* direkt zu Beginn der Argumentliste werden von verschiedenen *Awk*-Versionen unterschiedlich behandelt. Bei einigen sind diese Variablen bereits im `BEGIN`-Muster gesetzt, bei anderen nicht. Mit der Option `-v` *var=text* ist eine Variable *sicher* bereits im `BEGIN`-Muster gesetzt.
- Setzt man in der `BEGIN`-Aktion ein Element von `ARGV` gleich der leeren Zeichenkette `""`, so wird es beim Abarbeiten der Argumente ignoriert.
- Setzt man in der `BEGIN`-Aktion ein Element von `ARGV` gleich `"-"`, so wird beim Erreichen dieses Arguments von der Standard-Eingabe gelesen. Dies ist auch das *Defaultverhalten*, wenn kein einziges Argument angegeben wird.

3.2 Datentyp erzwingen

- Durch `EXPR + 0` kann das Ergebnis des Ausdrucks `EXPR` als *numerischer Wert*, durch `EXPR ""` (d.h. Verkettung mit der leeren Zeichenkette) als *Zeichenkettenwert* erzwungen werden.
- Beim Vergleich von zwei Feldern `$1 > $2` kann durch Anhängen von `+ 0` oder `""` an beide Felder die Art des Vergleichs (ob numerisch oder Zeichenketten) erzwungen werden, d.h. entweder numerisch:

```
if ($1 + 0 > $2 + 0)
    ...
```

  oder als Zeichenkette:

```
if ($1 "" > $2 "")
    ...
```

  Normalerweise wählt der *Awk* den richtigen Vergleichstyp aber automatisch abhängig vom Typ der Variablen aus (siehe Abschnitt 2.6.4 auf Seite 28).

3.3 Felder

- Die Nummer einer Feldvariablen kann durch einen numerischen Ausdruck ermittelt werden (z.B. `$(NF-1)`, dies ist ungleich `$NF-1`!)
- Eine Zuweisung an `$0` ist erlaubt, dabei findet erneut eine Zerlegung in Felder gemäß dem Feldtrenner `FS` statt.
- Eine Zuweisung an ein Feld `$1`, ..., `$NF` ist erlaubt, dabei wird `$0` aus allen Feldern mit dem Ausgabe-Feldtrenner `OFS` dazwischen neu erstellt.

- Eine Zuweisung an ein nicht existierendes Feld `$(NF+1),...` ist erlaubt, dabei werden alle Felder zwischen `$NF` und diesem neuen Feld mit der leeren Zeichenkette belegt und `$0` wird aus allen Feldern mit dem Ausgabe-Feldtrenner `OFS` dazwischen neu erstellt.
- Der Feldtrenner `FS` darf ein Regulärer Ausdruck sein (z.B. `"+␣*:␣*"`), der Recordtrenner `RS` nur im *Gawk*.

3.4 Arrays

- Arrayindices sind nicht sortiert (**Hash-Funktion**), d.h. eine Bearbeitung aller Elemente durch `for (i in arr) ...` durchläuft die Elemente in keiner bestimmten Reihenfolge. Es gibt keine eingebaute Möglichkeit, Arrayelemente zu sortieren (außer durch eine selbst geschriebene Sortierfunktion oder externes Sortieren per und wieder einlesen `sort`).
- Die **Nichtexistenz eines Arrayelements** kann durch `!(i in arr)` überprüft werden (`!i in arr` ist falsch, da erst `i` negiert wird und das Ergebnis dann als Index in `arr` gesucht wird)
- Ein Array kann durch `delete arr` oder `split("", arr)` vollständig gelöscht werden (*dies ist deutlich schneller als das Löschen der einzelnen Elemente!*).

3.5 Ausgabe

- Eine Leerzeile wird mit `print ""` oder `printf("\n")` erzeugt, `print` alleine druckt den aktuellen Eingabesatz `$0` aus.
- Bei Ausgaben mit `print` werden durch `,` (Komma) getrennte Argumente durch den Ausgabe-Feldtrenner `OFS` (**Default:** `"␣"`) getrennt und der ausgegebene Satz wird mit dem Ausgabe-Satztrenner `ORS` (**Default:** `"\n"`) abgeschlossen (bei Ausgaben mit `printf` haben `OFS` und `ORS` keine Bedeutung):

```
print ..., ... -> ...OFS...ORS   (Trennzeichen OFS dazw.)
print ... ...  -> ......         (Verkettung, ohne Trennz.)
```

- *Awk* kennt im `printf`-Format die `*`-Syntax für Breitenangaben per Variable *nicht* (*Gawk* schon). Als Ersatz kann aber die Zeichenkettenkonkatenation verwendet werden:

```
wid = 5
val = 123
printf("%0" wid "d", val) = printf("%05d", val) -> "00123"
printf("%0*d", wid, val)  = printf("%05d", val) -> "00123"
```

3.6 Dateien

- Das Einlesen von der Standard-Eingabe kann unter UNIX durch `Ctrl-D` und unter MS-DOS durch `Ctrl-Z` beendet werden (Dateiende = end of line = EOF).
- Ausgaben können mit `print ... | "cat 1>&2"` auf den Standard-Fehlerkanal umgelenkt werden.
- Im *Gawk* sind die Standard-Eingabe, die Standard-Ausgabe und der Standard-Fehlerkanal in `print`, `printf` und `getline` durch folgende Dateinamen ansprechbar:

Datei	Bedeutung
`/dev/stdin`	Standard-Eingabe (auch `-`, `/dev/fd/0`, `/dev/tty`)
`/dev/stdout`	Standard-Ausgabe (auch `/dev/fd/1`)
`/dev/stderr`	Standard-Fehlerkanal (auch `/dev/fd/2`)

- Folgende **Spezialdateien** stellen im *Gawk* Informationen über die Prozeßumgebung zur Verfügung (sie enthalten jeweils einen Satz mit 1 oder 4 Feldern). **Hinweis:** Ab *Gawk*-Version 3.1 sind diese Dateien nicht mehr vorhanden, sondern werden durch das vordefinierte Array `PROCINFO` ersetzt.

Datei	Inhalt
`/dev/pid`	Aktuelle Prozeß-ID
`/dev/ppid`	Prozeß-ID des Elternprozesses
`/dev/pgrpid`	Aktuelle Prozeß-Gruppen-ID
`/dev/user`	Folgende 4 oder mehr Arrayelemente: 1. Real User-ID 2. Effective User-ID 3. Real Group-ID 4. Effective Group-ID 5.-n: Group-IDs

- `print ... > "`$file$`"` löscht beim ersten Vorkommen die Datei $file$ zuerst, bevor die Ausgabe erfolgt. Jeder weitere Aufruf hängt an die Datei an (*nicht analog zur Shell!*).
- `print ... >> "`$file$`"` hängt grundsätzlich an die Datei $file$ an.
- Der Dateiname nach `<`, `>` und `>>` oder der Kommandoname bei `|` darf auch in einer Variablen stehen.
- Eine von einem *Awk*-Programm erzeugte Datei kann im gleichen Programm erst dann wieder eingelesen werden, wenn sie mit `close` abgeschlossen wurde.
- Lesen und Schreiben in einer Datei gleichzeitig ist nicht möglich (kein wahlfreies Positionieren).
- Das Argument von `close` muß *exakt* die gleiche Zeichenkette sein, wie die zu schließende Datei oder Pipe (am besten in einer Variablen ablegen und diese verwenden). Beispiel:

```
CMD = "sort -nr | uniq -c"
print "..." | CMD
close(CMD)
```

3.7 BEGIN/END/next/nextfile/exit

- Mehrere `BEGIN/END`-Muster werden in der Reihenfolge ihres Auftretens abgearbeitet.
- Besteht ein *Awk*-Programm *nur* aus einer **BEGIN**-Regel, so versucht es nicht, Eingabedaten einzulesen, sondern wird nach der Ausführung der BEGIN-Aktion beendet. Dies kann dazu verwendet werden, ein *reines Berechnungsprogramm* ohne Eingabedaten auszuführen.
- Für jeden Eingabesatz werden nacheinander *alle Muster* geprüft und gegebenenfalls die zugehörige Aktion ausgeführt. Enthält eine der Aktionen eines Musters eine `next`-, `nextfile`- oder `exit`-Anweisung, so gilt folgendes:
  - ▷ `next` in einer Aktion verwirft den aktuellen Satz, liest den nächsten Satz von der Eingabe ein und beginnt die Verarbeitung wieder bei der ersten Regel.
  - ▷ `nextfile` in einer Aktion verwirft den aktuellen Satz und die aktuelle Eingabedatei, liest den ersten Satz der nächsten Eingabedatei ein und beginnt die Verarbeitung wieder bei der ersten Regel.
  - ▷ `exit` in der `BEGIN`- oder einer Regel-Aktion springt zum `END`-Muster (oder beendet das Programm, falls das `END`-Muster fehlt). `exit` in der `END`-Aktion beendet das Programm.

3.8 Reguläre Ausdrücke

- Ein Regulärer Ausdruck muß nicht unbedingt in `/.../` stehen, sondern kann auch in Form einer Zeichenkette `"..."` angegeben werden (*Backslashes sind darin dann zu verdoppeln*) oder in einer Variablen stehen. Der Reguläre Ausdruck kann auf diese Weise sogar *dynamisch zur Laufzeit* erzeugt werden (leichte Performance-Einbuße). Das folgenden Beispiel enthält z.B. 3 identische `match`-Vergleiche:

```
match($0, /test/)
match($0, "test")
PAT = "te" "st"
match($0, PAT)
```

- Eine durch eine `match`-Operation in `$0` gefundene zu `REGEXP` passende Zeichenkette (erstreckt sich von `RSTART` bis `RSTART + RLENGTH - 1`) kann gegen eine andere Zeichenkette `SUBST` ausgetauscht werden durch:

```
match($0, /REGEXP/)
$0 = substr($0, 1, RSTART-1) "SUBST" substr($0, RSTART+RLENGTH)
```

3.9 Mehrzeilige Eingabesätze

- Nach der Zuweisung `RS = ""` (leere Zeichenkette) können **mehrzeilige Records** verarbeitet werden, da dann eine oder mehrere *Leerzeilen* die Datensätze trennen. Neben Whitespace ist dann automatisch `\n` ein zusätzlicher Feldtrenner.

- Wird zusätzlich der Feldtrenner `FS = "\n"` gesetzt, dann werden nur die einzelnen Zeilen eines mehrzeiligen Records als Felder betrachtet.

3.10 Häufige Fehler

3.10.1 Syntax-Fehler

- Generelle Aktion (ohne Muster) nicht in `{...}` eingeschlossen.
- `{` folgt nicht direkt auf das Muster (z.B. `BEGIN` und `END`), sondern steht erst in der nächsten Zeile.
- Abschließendes `}` nach Regel-Aktion vergessen.
- Abschließendes `"` bei Zeichenkette vergessen.
- Schlüsselwort `function` vor einer Funktionsdefinition vergessen (*C*).
- `strlen` statt `length` verwendet (*C*).
- `'x'` statt `"x"` bei Zeichenkonstante verwendet (*C*).
- In einzeiligem `if-else` vor dem `else` den `;` vergessen.
- Zeilenvorschub in Zeichenkette (abschließendes `"` erst in nächster Zeile).
- Zeilenumbruch innerhalb eines numerischen Ausdrucks oder einer `for`-Anweisung verwendet (*Zeilenumbruch mit* `\` *geht immer*).
- Einfache Variable auch als Array verwendet.
- Name einer (vordefinierten) Funktion als Variablenname verwendet (z.B. `index`).
- Ein Array-Zugriff a la *C* auf die einzelnen Zeichen einer Zeichenkette `string` per Indizierung `[i]` ist nicht möglich, sondern muß mit `substr` durchgeführt werden (z.B. den 10-ten Buchstaben merken und gegen den Buchstaben `'x'` austauschen):

```
ch     = substr(string, 10, 1)
string = substr(string,  1, 10-1) "x" substr(string, 10+1)
```

- `next` und `nextfile` sind nur in Aktionen erlaubt, nicht in Funktionen.
- `next` und `continue` verwechselt.

3.10.2 Besondere Verhaltensweisen

- `NF` hat im `BEGIN/END`-Teil den Wert 0.
- `NR` hat im `BEGIN`-Teil den Wert 0, im `END`-Teil enthält es die Anzahl der gelesenen Sätze.
- `FILENAME` ist im `BEGIN`-Teil nicht belegt, im `END`-Teil enthält es den Namen der letzten eingelesenen Datei.
- Wird keine Datei angegeben, so enthält `FILENAME` den Wert `"-"` (Standard-Eingabe).
- Wenn eine Eingabedatei `FILE` nicht vorhanden ist, erfolgt kein Programmabbruch, sondern es wird auf dem Standard-Fehlerkanal folgende Warnung ausgegeben:

```
input file "FILE": no such file or directory
```

- Bei erstmaligem Auftreten des Ausdrucks `arr[var++]` kann es sein, daß als Arrayindex `""` statt 0 verwendet wird (in älteren *Awk*-Versionen). Abhilfe: `var` in `BEGIN` mit `0` initialisieren oder `0 + var++` oder `++var` schreiben.
- `split` beginnt im Arrayelement mit Index 1 abzulegen (nicht 0 wie in *C*).
- `index`, `match` und `substr` numerieren die Zeichen in einer Zeichenkette beginnend mit 1 (nicht 0 wie in *C*).

3.10.3 Flüchtigkeitsfehler

- Tippfehler bei Variablennamen → es wird automatisch eine neue Variable erzeugt.
- Eine Funktion mit zu wenig/zu vielen Argumenten aufgerufen (wird nicht erkannt).
- Die Anzahl der Parameter in einer `printf`-Anweisung stimmt nicht mit der Anzahl der Formatanweisungen überein (→ Laufzeitfehler).
- In `sub` oder `gsub` vergessen, ein wörtliches `&` in der Ersetzungszeichenkette zu quotieren (`\&`).
- Argumente in `gensub`, `gsub`, `index`, `match`, `split`, `sub` und `substr` verwechselt.
- Bedingung `> 0` bei `getline` vergessen (→ Endlosschleife bei nicht vorhandener Datei).
- = statt == bei Vergleichen verwendet (wird nicht erkannt).

3.11 Sonstiges

3.11.1 Empfehlungen

- *Awk*-Programmdateien sollten die Extension `.awk` erhalten.
- Lokale Variablen in einer Funktion sollten durch 4 Leerzeichen von den Parametern abgetrennt werden:

```
function FUNC(PARAM1, PARAM2,    LOCAL1, LOCAL2, ...)
```

3.11.2 Syntax im Vergleich zu C

- Leerzeichen, Tabulatoren und Leerzeilen sind wie in *C* zur „Auflockerung“ von Anweisungen verwendbar.
- Bei Definition und Aufruf einer Funktion darf zwischen dem Funktionsnamen und der öffnenden Klammer *kein* Leerzeichen stehen.
- Da Semikolons nur zwischen Anweisungen auf derselben Zeile notwendig sind, können sie nach einer einzelnen Anweisung weggelassen werden. Man sollte sie aber trotzdem hinschreiben, sonst gewöhnt man sich daran und handelt sich bei der *C*- und *Perl*-Programmierung ständig Syntax-Fehler wegen fehlender Semikolons ein.
- Zeichen sind als Zeichenketten der Länge 1 zu schreiben (`"x"`), die *C*-Syntax für Zeichen (`'x'`) gibt es nicht.
- Um die gleichzeitige Programmierung in *C* zu erleichtern, sollte `printf` immer mit Klammern um die Argumente geschrieben werden.
- Eine Mehrfachverzweigung a la `switch-case-default-break` wie in *C* fehlt, kann aber durch `if (...) else if (...) ... else ...` ersetzt werden.

3.11.3 UNIX

- *Awk*-Programmdateien sollten in der 1. Zeile durch eine Shee-Bang-Zeile der Form `#!/usr/bin/awk` eingeleitet und mit `chmod u+x` ausführbar gemacht werden.
- Unter UNIX sucht *Gawk* nicht gefundene *Awk*-Programme in den Pfaden, die in der Shell-Variablen `AWKPATH` angegeben sind (Verzeichnisse darin durch `:` trennen). **Default:** `.:/usr/lib/awk:/usr/local/lib/awk` oder `.:/usr/local/share/awk`.
- Die Übergabe von Shell-Variablen oder Shell-Kommandoergebnissen an den *Awk* kann durch die `-v`-Option (oder auch per `ENVIRON` über Umgebungsvariable) erfolgen:

```
awk -v PREFIX="$PREFIX" '{ print PREFIX, $0 }'
awk -v RESULT=`CMD` '{ print RESULT, $0 }'
```

Die Übergabe von *Sh*-Variablen an *Awk*-Programme durch folgenden **Quotierungstrick** wird dadurch überflüssig (*war auch sehr unübersichtlich)*:

```
awk '{ print "'"$PREFIX"'", $0 }'
```

3.11.4 Die automatische Leseschleife

Der Pseudocode für die automatische Leseschleife lautet (*dies ist kein sinnvolles Awk-Programm!*):

```
# BEGIN-Aktionen ausführen
if (exist(BEGIN-Muster))
    call BEGIN-Aktion
# Über alle Dateinamen der Kommandozeile
for (ARGIND = 1; ARGIND < ARGC; ++ARGIND) {
    # Dateiname hat Form VAR=TEXT? -> Zuweisung + nächste Datei
    if (ARGV[ARGIND] ~ /^[A-Za-z_][A-Za-z_0-9]*=/) {
        eval ARGV[ARGIND]; # Zuweisung ausführen
        continue
    }
    FILENAME = ARGV[ARGIND]
    FNR = 0
    # Dateiinhalt zeilenweise lesen und in Felder zerlegen
    while ((getline < FILENAME) > 0) {
        ++FNR
        ++NR
        NF = split($0, $1-$NF, FS)
        # Eigener Programmcode
        ...
    }
}
# END-Aktionen ausführen
if (exist(END-Muster))
    call END-Aktion
```

3.11.5 Awk-Compiler

Auf dem PC und unter UNIX gibt es auch *Awk*-Compiler (MKS-Toolkit, Thompson Automation Software, awka), die Binär-Programme erstellen. Allerdings läuft bei den meisten das erzeugte Programm nicht schneller, da nur der *Awk*-Zwischencode mit dem *Awk*-Interpreter zu einem ausführbaren Programm zusammengebunden wird. Trotzdem sind sie ganz brauchbar, da so der Code nicht preisgegeben werden muß und der Aufruf des Programms einfacher ist. Der Aufruf zur Übersetzung von `PROGFILE.awk` lautet:

```
awkc -l -O -f PROGFILE.awk -o PROGFILE.exe   oder
tawkc -xe -o PROGFILE.exe PROGFILE.awk
```

Erzeugt aus `PROGFILE.awk` das `EXE`-File `PROGFILE.exe`, die Option `-l` (`large`) erzeugt ein Programm im large-Modell (nutzt bis zu 640 KByte), die Option `-O` (`optimize`) optimiert den erzeugten Code (wenn möglich), die Option `-o` (`output`) legt die Ausgabedatei fest.

Das Programm `awka` übersetzt Awk-Programm wirklich in C-Programme (bis auf wenige kleine Einschränkungen alle Funktionalitäten), die dann natürlich viel schneller ablaufen.

4 Beispielprogramme

Zu den mit DATA gekennzeichneten *Awk*-Programmen gibt es jeweils eine **Datendatei** gleichen Namens mit der Endung `.dat`. Die mit SH gekennzeichneten Programme sind in ein **Shell-Skript** verpackt. Die Beispiel-Programme sind aufzurufen per:

```
awk -f PROGRAM.awk      oder
sh PROGRAM.sh
```

oder bei den Programmen mit einer Datendatei per:

```
awk -f PROGRAM.awk PROGRAM.dat      oder
sh PROGRAM.sh PROGRAM.dat
```

4.1 Standard-Funktionen

- `args` — Zugriff auf Kommandozeilenargumente (`ARGC`, `ARGV`)
- `array` — Anlegen, Zugriff, Testen und Löschen von Arrayelementen
- `beginend` — Verhalten von `exit` bei `BEGIN/END`
- `concat` — Konkatenation von Zeichenketten
- `control` — Die verschiedenen Kontrollstrukturen (`if`, `for`, `while`, `do`)
- `environ` — Zugriff auf Umgebungsvariablen (`ENVIRON`)
- `fields` — Zerlegung in Felder demonstrieren (`FS`) — DATA
- `function` — Funktionsdefinition und -aufruf (`function`)
- `bool` — Welche Werte sind wahr/falsch (`0`, `1`, `""`, `"0"`, ...)
- `nofile` — Verhalten falls Datei aus Kommandozeile fehlt
- `numcmp` — Wann wird numerischer Vergleich/Zeichenkettenvergleich gemacht?
- `printf1` — Formatumwandlungen
- `printf2` — Alternative Formatumwandlungen (`#`)
- `range` — Zeilen zwischen `AAA` und `BBB` ausgeben (ohne `AAA` und `BBB`) — DATA
- `regexpr` — Reguläre Ausdrücke matchen

- `stdout` — Ausgabe auf Standard-Ausgabe, Standard-Fehlerkanal und das Terminal
- `string` — Zeichenketten-Funktionen
- `work1` — Wie *Awk* arbeitet (Standard-Eingabe oder eine Datei) — DATA
- `work2` — Wie *Awk* arbeitet (mehrere Dateien) — DATA
- `work3` — Wie *Awk* arbeitet (mehrere Dateien + Variablenbelegung) — DATA
- `work4` — Wie *Awk* arbeitet (`exit`-Befehl) — DATA
- `work5` — Wie *Awk* arbeitet (`next`-Befehl) — DATA

4.2 Erweiterte Funktionen

- `fixfield` — Lesen von Feldern mit fester Breite — DATA
- `getline` — Lesen externer Dateien — DATA
- `multidim` — Mehrdimensionale Arrays
- `multline` — Mehrzeilige Datensätze — DATA
- `pipe` — Pipe in/aus Shell-Kommando
- `system` — Aufruf von System-Kommandos
- `time` — Zeit-Funktionen und -formate

4.3 Simulierte UNIX-Programme

- `cat` — `cat` simulieren
- `cut` — `cut` simulieren (nur ein Zeichenbereich)
- `grep` — `grep` simulieren
- `head1` — `head` simulieren (eine Datei)
- `head2` — `head` simulieren (mehrere Dateien)
- `make` — `make` simulieren (ohne Makros, implizite Regeln, . . .) — DATA
- `nl` — Zeilen einer Datei durchnumerieren
- `split` — `split` simulieren
- `tail1` — `tail` simulieren (eine Datei)
- `tail2` — `tail` simulieren (mehrere Dateien)

- `uniq` — `uniq` simulieren
- `wc1` — `wc` simulieren (eine Datei)
- `wc2` — `wc` simulieren (mehrere Dateien)

4.4 Programme

- `asciitab` — ASCII-Tabelle erstellen
- `awkhead` — *Awk*-Beispieldateien überprüfen und Programmkopf extrahieren
- `bundle1` — Mehrere Dateien zu einer Datei zusammenfassen (→ `unbundl1`)
- `bundle2` — Mehrere Dateien zu einer Datei zusammenfassen (→ `unbundl2`)
- `calc1` — Taschenrechner für ganze Zahlen mit den Operatoren `+-*/()`
- `calc2` — Taschenrechner für Dezimalzahlen mit den Operatoren `+-*/()^`
- `calc3` — Taschenrechner mit Variablen, Funktionen, ...
- `calc4` — Taschenrechner mit selbstdefinierten Funktionen, ...
- `childcnt` — Zählen von Ziffern wie Kinder es tun (alles wörtlich nehmen)
- `colfmt` — Spalten gemäß breitestem Inhalt mit `|` dazwischen ausgeben (1 Durchlauf) — DATA
- `collect` — Dateien getrennt durch 3 Leerzeilen hintereinanderhängen
- `datechk` — Erkennen von Zeilen, die nur Datumswerte enthalten — DATA
- `eanchk` — EAN überprüfen (Präfix, Länge, Prüfziffer korrekt) — DATA
- `eanfreq1` — Häufigkeiten der EAN an Pos 13-25 zählen und ausgeben — DATA
- `eanfreq2` — Häufigkeitsverteilung nach absteigender Häufigkeit + kumul. Proz.
- `empty1ln` — Leerzeilen am Dateianfang/ende entfernen, mehrere zu einer reduzieren — DATA
- `genltr` — Generator für Briefe — DATA
- `guessnum` — Zahlenratespiel (Zahl aus 1..LIMIT erraten)
- `headline` — Unterstrichene Überschriften herausziehen
- `lotto` — Lottozahlen ermitteln (Std: 6 aus 49)
- `mwst` — Numerischen Ausdruck und Mehrwertsteuer ausrechnen
- `numchk` — Erkennen von Zeilen, die nur Gleitkommazahlen enthalten — DATA

- `randline` — Eine Zeile aus einer beliebig langen Datei zufällig auswählen — DATA
- `randnum` — Zahlen zufällig aus einer Zahlenmenge ermitteln (minimaler Aufwand)
- `tblfmt` — Spalten gemäß breitestem Inhalt mit | dazwischen ausgeben (2 Durchläufe, Zwischendatei, Programmdatei) — SH,DATA
- `textfmt` — Text auf bestimmte Breite umbrechen (Std: 60) — DATA
- `thousand` — Tausendertrennzeichen „,“ in Zahlen einfügen — DATA
- `unbundl1` — Dateizusammenfassung in Einzeldateien zerlegen (→ `bundle1`)
- `unbundl2` — Dateizusammenfassung in Einzeldateien zerlegen (→ `bundle2`)
- `wordfreq` — Worthäufigkeiten ermitteln — DATA

5 ASCII-Tabelle

Der ASCII-Zeichencode definiert die Standardbelegung der Codes 0–127 mit Zeichen (kennt keine landesspezifischen Sonderzeichen wie z.B. Umlaute). Die Codes 128–255 werden je nach Zeichensatz unterschiedlich belegt (mit Sonderzeichen wie z.B. Umlauten). Die wichtigsten ASCII-Zeichen und ihre Reihenfolge sind:

- Steuer-Zeichen (Control) (0–31, *zusammenhängend*)
- Leerzeichen (32)
- Ziffern `0-9` (48–57, *zusammenhängend*)
- Großbuchstaben `A-Z` (65–90, *zusammenhängend*)
- Kleinbuchstaben `a-z` (97–122, *zusammenhängend*)
- Tilde `~` (126)
- Druckbare Zeichen `SPACE-~` (32–127, *zusammenhängend*)

d.h. es gelten folgende Beziehungen: `SPACE` < `0-9` < `A-Z` < `a-z` < `~`

	00	10	20	30	40	50	60	70
0	^@	^P	[SPACE]	0	@	P	`	p
1	^A	^Q	!	1	A	Q	a	q
2	^B	^R	"	2	B	R	b	r
3	^C	^S	#	3	C	S	c	s
4	^D	^T	$	4	D	T	d	t
5	^E	^U	%	5	E	U	e	u
6	^F	^V	&	6	F	V	f	v
7	^G [BEL]	^W	'	7	G	W	g	w
8	^H [BS]	^X	(	8	H	X	h	x
9	^I [TAB]	^Y	)	9	I	Y	i	y
A	^J [LF]	^Z	*	:	J	Z	j	z
B	^K [VTAB]	[ESC]	+	;	K	[	k	{
C	^L [FF]	^\	,	<	L	\	l	|
D	^M [CR]	^]	-	=	M	]	m	}
E	^N	^^	.	>	N	^	n	~
F	^O	^_	/	?	O	_	o	[DEL]

Hinweis:

- `^X` steht für `Ctrl-X` (Control) oder `Strg-X` (Steuerung) und beschreibt die Terminal-Steuerzeichen.
- Zeichennamen: `BEL` = Glocke, `BS` = Backspace, `CR` = Carriage Return, `DEL` = Delete, `ESC` = Escape, `FF` = Formfeed, `LF` = Linefeed, `SPACE` = Leerzeichen, `TAB` = Tabulator, `VTAB` = Vertikaler Tabulator.