

Die Programmiersprache Awk: Einführung, Tipps und Tricks

Version 1.40 — 7.11.2016

© 2001–2016 T. Birnthal, OSTC GmbH

Die Informationen in diesem Skript wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Der Autor übernimmt keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen.

Alle Rechte vorbehalten einschließlich Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Für Dokumente und Programme unter dem Copyright der OSTC GmbH gilt:

- Dürfen heruntergeladen und im privaten Bereich frei verwendet werden.
- Kommerzielle Nutzung bedarf der vorherigen Zustimmung durch die OSTC GmbH.
- Titelseite und Copyright-Hinweise darin dürfen nicht verändert werden.

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 Birnthal
E-Mail: tb@ostc.de
Web: www.ostc.de

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einführung | 4 |
| 1.1 | Übersicht | 4 |
| 1.2 | Entstehungsgeschichte | 5 |
| 1.3 | Die verschiedenen Versionen | 6 |
| 1.4 | Einsatzgebiete | 6 |
| 1.5 | Eigenschaften | 7 |
| 1.5.1 | Vorteile | 8 |
| 1.5.2 | Nachteile | 8 |
| 1.6 | Ein einfaches Beispiel | 9 |
| 1.7 | 10 typische Einzeiler | 10 |
| 1.8 | Referenzen | 11 |
| 1.8.1 | Literatur | 11 |
| 1.8.2 | Links | 13 |
| 2 | Beschreibung | 14 |
| 2.1 | Aufruf, Parameter, Optionen und Umgebungsvariablen | 14 |
| 2.1.1 | Exit-Status | 16 |
| 2.2 | Konstanten | 16 |
| 2.2.1 | Zahlen | 16 |
| 2.2.2 | Zeichenketten | 16 |
| 2.2.2.1 | Escape-Sequenzen | 16 |
| 2.2.3 | Logische Werte | 17 |
| 2.2.4 | Reguläre Ausdrücke | 17 |
| 2.2.4.1 | Metazeichen | 17 |
| 2.2.4.2 | POSIX-Zeichenklassen | 18 |
| 2.3 | Operatoren | 19 |
| 2.4 | Ausdrücke (Expressions) | 21 |
| 2.5 | Programmaufbau | 22 |
| 2.5.1 | Regeln (Muster-Aktion Paare) | 22 |
| 2.5.1.1 | Muster | 22 |
| 2.5.1.2 | Aktion | 23 |
| 2.5.2 | Funktionen | 26 |
| 2.5.2.1 | Definition | 26 |
| 2.5.2.2 | Aufruf | 27 |
| 2.5.3 | Kommentare, Whitespace, Leerzeilen und Zeilenumbruch | 28 |
| 2.5.4 | Strukturierung von Programmen | 29 |
| 2.6 | Programmablauf | 30 |
| 2.6.1 | Vereinfachte Version | 30 |
| 2.6.2 | Ergänzungen | 31 |
| 2.7 | Felder, Variablen und Arrays | 32 |
| 2.7.1 | Felder | 33 |
| 2.7.2 | Variablen | 33 |
| 2.7.3 | Assoziative Arrays | 33 |
| 2.7.4 | Datentypen, Datentypumwandlung und Defaultwerte | 35 |
| 2.8 | Vordefinierte Variablen und Arrays | 37 |

| | | |
|----------|--|-----------|
| 2.9 | Vordefinierte Funktionen | 41 |
| 2.9.1 | Arithmetik-Funktionen | 41 |
| 2.9.2 | Bit-Funktionen | 41 |
| 2.9.3 | Zeichenketten-Funktionen | 42 |
| 2.9.4 | Array-Funktionen | 43 |
| 2.9.5 | Ein/Ausgabe-Funktionen | 43 |
| 2.9.6 | Printf-Formatumwandlung | 45 |
| 2.9.7 | Zeit-Funktionen | 46 |
| 2.9.8 | Internationalisierung-Funktionen | 47 |
| 2.10 | Koprozesse | 47 |
| 2.11 | Grenzen | 47 |
| 3 | Tipps und Tricks | 48 |
| 3.1 | Aufrufparameter | 48 |
| 3.2 | Datentyp erzwingen | 48 |
| 3.3 | Felder | 49 |
| 3.4 | Arrays | 49 |
| 3.5 | Ausgabe | 50 |
| 3.6 | Dateien | 50 |
| 3.7 | BEGIN/END/next/nextfile/exit | 51 |
| 3.8 | Reguläre Ausdrücke | 52 |
| 3.9 | Mehrzeilige Eingabesätze (Datenblöcke) | 52 |
| 3.10 | Häufige Fehler | 53 |
| 3.10.1 | Syntax-Fehler | 53 |
| 3.10.2 | Besondere Verhaltensweisen | 53 |
| 3.10.3 | Flüchtigkeitsfehler | 54 |
| 3.11 | Sonstiges | 55 |
| 3.11.1 | Empfehlungen | 55 |
| 3.11.2 | Syntax im Vergleich zu C | 55 |
| 3.11.3 | UNIX | 55 |
| 3.11.4 | Die automatische Leseschleife | 56 |
| 3.11.5 | Awk-Compiler | 56 |
| 4 | Beispielprogramme | 57 |
| 4.1 | Standard-Funktionen | 57 |
| 4.2 | Erweiterte Funktionen | 58 |
| 4.3 | UNIX-Werkzeuge simulieren | 58 |
| 4.4 | Programme | 59 |
| 5 | ASCII Tabelle | 61 |

1 Einführung

1.1 Übersicht

Computeranwender verschwenden oft viel Zeit für vergleichsweise einfache, mechanische Datenverarbeitung:

- Änderungen am Format von Daten
- Gültigkeitsüberprüfungen
- Suchen nach Einträgen mit bestimmten Eigenschaften
- Summieren von Zahlen
- Zusammenstellen von Berichten
- usw.

Diese Tätigkeiten sollten eigentlich dem Computer überlassen werden. Oft ist es aber zu umständlich, dafür ein Programm in einer Standardprogrammiersprache 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 passende Datenzeilen über eine beliebigen Kombination von **Regulären Ausdrücken** und Vergleichsoperationen auf Zahlen, Zeichenketten, Feldern, Variablen und Arrayelementen ermitteln. Die zugehörige Aktion kann 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 zu Beginn aufgezählten Arten der Datenverarbeitung geeignet. Ein oder zwei Zeilen lange Programme werden direkt auf der Kommandozeile über die Tastatur eingegeben, sofort verwendet und anschließend wieder verworfen. Im wesentlichen ist *Awk* ein allgemein verwendbares Programmierwerkzeug, das viele andere Hilfsprogramme (wie z.B. *Sed*, *Grep*, *Tr*, *Expr*) ersetzen kann, allerdings etwas langsamer als diese (spezialisierten) Werkzeuge ist.

Anstellen von **Zeilen** spricht man auch von **(Daten)Sätzen** oder **Records**, da der Zeilentrenner beliebig wählbar ist (Std: `\n`). Anstelle von **Feldern** spricht man auch von **Wörtern**.

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 (*PCRE* = Perl Compatible Regular Expressions), C-Bibliotheks-Funktionen, UNIX-Systemaufrufe, Objektorientierung, Modularisierung, Sicherheitsaspekte, usw. Diese Programmiersprache ist daher sehr mächtig und sehr umfangreich und ersetzt in zunehmendem 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 Umfangs 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.
- Viele Konzepte aus dem *Awk* sind auf andere Programmiersprachen wie *Perl*, *PHP*, *Tcl/Tk*, *Python*, *Ruby*, *Lua*, ... 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**, 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 Referenzimplementierung ist letztlich auf jedem System installierbar:

| Variante | Beschreibung |
|----------------|--|
| <i>Awk</i> | Ursprüngliche Version (1977) |
| <i>Awka</i> | Freier <i>Awk</i> -Compiler |
| <i>lawk</i> | Shell-Skript, das <i>Awk</i> Datei-Include beibringt (per <code>@include file</code>) |
| <i>Gawk</i> | Freie GNU-Version von <i>Nawk</i> mit Erweiterungen (kaum mehr Beschränkungen und deutlich schneller) |
| <i>Mawk</i> | Freier <i>Awk</i> von Michael Brennan |
| <i>MKS-Awk</i> | Teil des <i>MKS-Toolkit</i> von Mortice Kern Systems |
| <i>Nawk</i> | Erweiterte Version (<i>New Awk</i> : Funktionen, <code>getline</code> , ... 1985) |
| <i>Oawk</i> | <i>Old Awk</i> (im Gegensatz zu <i>New Awk</i>) |
| <i>Pgawk</i> | Profilingversion von <i>Gawk</i> (erzeugt <code>awkprof.out</code> und ist langsamer) |
| <i>Tawk</i> | Thompson-Version von <i>Nawk</i> mit sehr vielen Erweiterungen + Compiler |
| <i>Xawk</i> | XML- <i>Awk</i> : <i>Gawk</i> mit Erweiterungen zur XML-Verarbeitung |

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, 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, falls 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 Anwenderanforderungen 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 **Minisprachen**.
- **Sugar-coating**, d.h. "Verpacken" einer bestehenden Programmier- oder Anwendungssprache in eine leichter einsetzbare Hülle ("Syntactic Sugar").

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. kein Öffnen von Dateien notwendig.
- Besitzt eine Reihe sinnvoller **Automatismen** (z.B. Zerlegung der Eingabedaten in Zeilen, 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, beliebig lang.

- 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 (keine Speicherplatzreservierung oder -freigabe notwendig).
- Bietet (**rekursive**) **Funktionen** mit Parametern (keine Funktions-Prototypen notwendig).

1.5.1 Vorteile

- Erlaubt 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 Datentyp 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 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 500 MByte bzw. 5.000.000 Zeilen).
- Kann **nur Textdateien** (zeilenorientierte ASCII-Daten) verarbeiten, aber keine Binärdaten, insbesondere das **NUL-Byte** '\0' nicht!). Durch Vor- und Nachschalten entsprechender Konverter kann dies aber ausgeglichen werden (z.B. durch `bin2asc file1 | awk ... | asc2bin > file2`).
- **Kein wahlfreies Positionieren** in Dateien möglich, d.h. können nur ein Mal von vorne nach hinten durchgelesen werden (auch mehrfach, falls sie erneut geöffnet werden).
- Kennt **keine Datenstrukturen** oder Zeiger (aber über assoziative Arrays simulierbar).

- Kennt keine **modullokalen Variablen**, nur globale/lokale zu Funktionen (im *Tawk* doch vorhanden).
- **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 (insbesondere keine `#include`-Anweisung).
- Kein Debugger vorhanden (im *Tawk* schon).
- Arrayelemente sind unsortiert (Hash-Funktion).
- Kann nicht sortieren, dies muss extern erfolgen (*Gawk* kennt Sortierfunktionen für Arrays).
- Zugriff auf einzelne Zeichen in Zeichenketten nicht per Index (wie in *C*) möglich, sondern nur per Funktion `substr` bzw. Verkettung (Konkatenation), d.h. etwas langsam und umständlich.
- Syntaxfehlermeldungen 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 *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      # Suchpfad PATH enthält AWK-Verzeichnis
./addcol2.awk data   # Skript liegt im aktuellen Verzeichnis
```

Dazu muss eine **Shee-Bang-Zeile** zusätzlich als 1. Zeile in das *Awk*-Skript `addcol2.awk` aufgenommen werden (beginnend in der 1. Spalte!):

```
#!/usr/bin/awk -f
{ sum = sum + $2 }
END { print "sum =", sum }
```

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. Der Aufruf lautet dann:

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. Anzahl Eingabezeilen ausgeben:

```
END { print NR }
```

2. Zehnte Eingabezeile ausgeben:

```
NR == 10
```

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

```
NF > 4
```

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

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

5. 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.
- Aho, Kernighan, Weinberger, *Die Programmiersprache AWK*, OSTC Verlag.
Deutsche Übersetzung der „Bibel“ zum *Awk* von den Autoren der Sprache selbst. Enthält zusätzlich zum Originalbuch einen Abschnitt zu den Erweiterungen des *Gawk*.

- Arnold 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.
- Dale Dougherty, *Sed & Awk, 2. Edition*, O'Reilly.
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.
- Daniel Gilly, *UNIX in a Nutshell, 2. Edition*, O'Reilly.
Enthält eine Kurzbeschreibung der Regulären Ausdrücke, des *Sed* und des *Awk*.
- Gottfried Staubach, *UNIX-Werkzeuge zur Textmusterverarbeitung*, Springer.
Enthält eine kompakte deutsche Beschreibung der Regulären Ausdrücke und des *Awk*.
- Ulrich Cuber, *Linux Scripting*, Franzis'.
Enthält eine übersichtliche Beschreibung der UNIX-Werkzeuge, der Regulären Ausdrücke, des *Awk*, der Shell sowie von Perl und Tcl/Tk.
- Stephan Thesing, *sed&awk GE-PACKT*, mitp.
Zusammenfassung der Befehle des *Sed* und des *Awk*.
- Arnold Robbins, *sed&awk kurz&gut*, O'Reilly.
Zusammenfassung der Befehle des *Sed* und des *Awk*.
- Jeffrey Friedl, *Reguläre Ausdrücke*, O'Reilly.
Enthält eine ausführliche Beschreibung zu Regulären Ausdrücken in ihren vielfachen Ausprägungen und eine vollständige Analyse ihres Verhaltens.
- Jeffrey Friedl, *Mastering Regular Expressions, 3. Edition*, O'Reilly.
Enthält eine ausführliche Beschreibung zu Regulären Ausdrücken in ihren vielfachen Ausprägungen und eine vollständige Analyse ihres Verhaltens.
- Tony Stubblebine, *Reguläre Ausdrücke kurz&gut*, O'Reilly.
Beschreibung verschiedener Varianten von Regulären Ausdrücken und ihres Verhaltens.
- Christian Wenz, *Reguläre Ausdrücke — schnell+kompakt*, entwickler.press.
Beschreibung verschiedener Varianten von Regulären Ausdrücken und ihres Verhaltens.

- John Bambenek, Agnieszka Klus, *grep - kurz&gut*, O'Reilly.
Beschreibung der verschiedenen grep-Varianten mit ihren Regulären Ausdrücken.
- Jan Goyvaerts, Steven Levithan, *Reguläre Ausdrücke Kochbuch*, O'Reilly.
Viele fix-und-fertig Lösungen für Standardprobleme des Suchens + Ersetzens mit Regulären Ausdrücken.
- Michael Fitzgerald, *Einstieg in Reguläre Ausdrücke*, O'Reilly.
Die Grundlagen von Regulären Ausdrücken erklärt (alles was man wissen muss).
- Michael Fitzgerald, *Introducing Regular Expressions*, O'Reilly.
Die Grundlagen von Regulären Ausdrücken erklärt (alles was man wissen muss).

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-Seite, die auch zum Awk einiges enthält.

2 Beschreibung

2.1 Aufruf, Parameter, Optionen und Umgebungsvariablen

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...]
```

Falls **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 |
|--------------------------|--|
| <code>-F seplere</code> | Feldtrenner FS auf <i>seplere</i> festlegen (Default: "_") |
| <code>-f pgm</code> | Programmdatei <i>pgm</i> einlesen (<i>mehrfach erlaubt</i>) |
| <code>-v var=text</code> | Variable <i>var</i> den Wert <i>text</i> vor Programmstart zuweisen |
| <code>-mf n</code> | Maximale Anzahl Felder <i>n</i> festlegen (<i>im Gawk überflüssig</i>) |
| <code>-mr n</code> | Maximale Recordlänge <i>n</i> festlegen (<i>im Gawk überflüssig</i>) |
| <code>--</code> | Beendet Optionen-Liste (erlaubt Argumente mit führendem -) |

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:

| <i>Gawk-Option</i> | Bedeutung |
|---|--|
| --field-separator= <i>sep</i> --assign <i>var=val</i> --file= <i>file</i> | Analog -F Analog -v Analog -f (<i>mehrfach erlaubt</i>) |
| --compat --traditional --posix | <i>Gawk</i> -Erweiterungen abschalten <i>Gawk</i> -Erweiterungen abschalten Nur POSIX-Umfang zulassen (siehe unten). |
| --copyleft --copyright | GPL (GNU General Public Licence) ausgeben GPL (GNU General Public Licence) ausgeben |
| --help --usage --version | <i>Gawk</i> -Usage-Meldung ausgeben <i>Gawk</i> -Usage-Meldung ausgeben <i>Gawk</i> -Versionsnummer ausgeben |
| --source= <i>program</i> --exec= <i>pgm</i> | Programmtext <i>program</i> auf der Kommandozeile angeben (<i>mehrfach erlaubt, in Hochkomma setzen</i>) Analog -f, aber letzte verarbeitete Option (für CGI-Skripte) |
| --lint[= <i>wert</i>] --lint-old | Warnung bei gefährlichen/nicht portablen Konstrukten (<i>wert=fatal</i> führt zu fatalem Fehler statt Warnung) (<i>wert=invalid</i> : nur Warnung bei wirklich ungültigen Dingen) Warnung bei Konstrukten abweichend vom Ur-Awk |
| --dump-variables[= <i>file</i>] --gen-po --profile[= <i>file</i>] | Sortierte Liste globaler Var. mit Typ + Endwert ausgeben Aus lokalisierbaren Strings .po-Datei erzeugen (<i>gentext</i>) Profildaten in Datei <i>awkprof.out</i> bzw. <i>file</i> ablegen (<i>pgawk</i>) |
| --disable-directories-fatal --enable-switch --non-decimal-data --re-interval --use-lc-numeric | Verz. als Kmdozeilen-Argument ignorieren (statt abzubrechen) <i>switch-case-default</i> -Anweisung aus C erlauben Oktal/Hexadezimalwerte in Eingabedaten erkennen (<i>Vorsicht!</i>) { <i>n,m</i> }-Wiederholung in Regulären Ausdrücken erlauben Dezimaltrennzeichen gemäß Locale-Einstellung beim Einlesen |

Angabe von `--profile` bei *Gawk* gibt das Awk-Programm in einer schön formatierten Form aus. Angabe von `--profile` bei *Pgawk* gibt zu jeder Anweisung die Anzahl ihrer Ausführungen und zu jeder Funktion die Anzahl ihrer Aufrufe aus. Standardname der Profildatei ist *awkprof.out*, er kann durch Angabe eines Namens nach `--profile` geändert werden.

Folgende **Umgebungsvariablen** sind *Gawk*-spezifisch:

- `AWKPATH` legt **Suchpfad** fest (analog `PATH`, d.h. eine Liste von durch `:` getrennten Verzeichnissen), in denen nach den per Option `-f` angegebenen Awk-Dateien gesucht wird, falls sie nicht im aktuellen Verzeichnis gefunden werden.
- `POSIXLY_CORRECT` hat die gleiche Bedeutung wie der Schalter `--posix` und schaltet folgende Erweiterungen des *Gawk* ab:
 - ▷ Escape-Sequenz `\x` wird nicht erkannt.
 - ▷ Nur Leerzeichen + Tabulator sind Feldtrenner, falls `FS` ein einzelnes Leerzeichen als Wert zugewiesen wird (Zeichen Newline nicht).
 - ▷ Zeilenumbruch nach `?` und `:` nicht erlaubt.
 - ▷ Synonym `func` für Schlüsselwort `function` nicht erlaubt.
 - ▷ Synonyme `**` und `**=` für Potenzierungsoperatoren `^` und `^=` nicht erlaubt.
 - ▷ Funktion `fflush` nicht erlaubt.

2.1.1 Exit-Status

Der Awk gibt am Skriptende folgende Exit-Status zurück:

| Status | Bedeutung |
|----------|---|
| 0 | Awk-Skript fehlerfrei ausgeführt |
| 1 | Während Ausführung trat ein Fehler auf (Syntax) |
| 2 | Während Ausführung trat ein fataler Fehler auf (Absturz) |
| <i>n</i> | Während Ausführung wurde <code>exit (n)</code> ausgeführt |

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 einbuchstabile 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. **Oktale Werte** werden durch 0 eingeleitet, **hexadezimale Werte** durch `0x` oder `0X` (nur *Gawk*). Beispiele:

```
123      -1      -3.141592   +.0125   1e12     -987.654E-321
0123     0777     07654321           # Oktale Werte
0XFF     0xAFFE   0X1234567890ABCDEF   # Hexadezimale Werte
0xff     0xAffe   0x1234567890abcdef   # Hexadezimale Werte
```

2.2.2 Zeichenketten

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

```
"abc"           # Ohne Zeilenvorschub
"Dies ist ein Text.\n" # Mit Zeilenvorschub
""              # 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 (horizontal) |
| * | \v | Vertikaler Tabulator |
| | \ddd | Zeichen mit oktalem Wert <i>ddd</i> (Zahlen zwischen 000 und 377) |
| * | \xdd | Zeichen mit hexadezimalen Wert <i>dd</i> (Zahlen zwischen 00 und ff/FF) |
| | \" | Anführungszeichen |
| | \\ | Backslash |
| | \/ | Slash |

2.2.3 Logische Werte

Die Werte 0 (Zahl 0) und "" (leere Zeichenkette) werden als **false** interpretiert, alle anderen Zahlen bzw. Zeichenketten werden als **true** 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, ...).

```
/abc/           # Enthält "abc"
/^abc$/        # Enthält exakt "abc"
/(abc|def)+/   # Enthält "abc" oder "def" mind. 1x nacheinander
/[ \t]+/       # Enthält Leerzeichen oder Tabulator
/\r$/         # Enthält Carriage Return am Zeilenende
```

Dynamische Reguläre Ausdrücke: Alternativ können sie auch in Form einer Zeichenkette "... " angegeben werden (**Hinweis:** Backslashes sind darin dann zu verdoppeln) oder in einer Variablen stehen. Sie werden auf diese Weise **dynamisch zur Laufzeit** erzeugt (unterliegen dann allerdings Performance-Einbußen). Beispiele:

```
"abc"          # Enthält "abc"
"^abc$"       # Enthält exakt "abc"
"(abc|def)+"  # Enthält "abc" oder "def" mind. 1x nacheinander
"[ \t]+"      # Enthält Leerzeichen oder Tabulator
"\\r$"        # Enthält Carriage Return am Zeilenende
```

2.2.4.1 Metazeichen

Folgende **Metazeichen** sind in Regulären Ausdrücken möglich, sie sind nach **absteigendem Vorrang** geordnet. r , r_1 und r_2 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 (höchster Vorrang) |
| | c $\backslash c$ \wedge $\$$ \cdot $[abc]$ $[a-z]$ $[\wedge abc]$ $[\wedge a-z]$ | Zeichen c (kein Metazeichen) Escape-Sequenz oder (Meta)Zeichen c wörtlich Zeichenketten/Zeilen-Anfang Zeichenketten/Zeilen-Ende 1 beliebiges Zeichen Zeichenklasse: 1 Zeichen aus Menge abc bzw. $a-z$ Invertierte Zeichenklasse: 1 Zeichen nicht aus Menge abc bzw. $a-z$ |
| * | $\backslash y$ $\backslash B$ | Wortgrenze / Wortinneres (Break) |
| * | $\backslash <$ $\backslash >$ | Wortanfang / Wortende |
| * | $[[:<:]]$ $[[:>:]]$ | Wortanfang / Wortende |
| * | $\backslash w$ $\backslash W$ | 1 Wortzeichen ($[A-Za-z_0-9]$) / kein Wortzeichen ($[\wedge A-Za-z_0-9]$) |
| * | $\backslash \wedge$ $\backslash \prime$ | Zeichenketten/Zeilen-Anfang/Ende (alternative Form) |
| | r^* r^+ $r^?$ | $0 - \infty$ aufeinanderfolgende Zeichenketten, auf die r paßt (Closure) $1 - \infty$ aufeinanderfolgende Zeichenketten, auf die r paßt (Positive Closure) $0/1$ 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 |
| * | $r\{m\}$ | m Wiederholungen der Zeichenkette, auf die r paßt (genau) |
| | $r_1 r_2$ $r_1 r_2$ | Verkettung: Jede Zeichenkette xy , wo r_1 zu x und r_2 zu y paßt Alternative: Jede Zeichenkette, die zu r_1 oder r_2 paßt (niedrigster Vorrang) |

- Jedes Zeichen außer den Metazeichen — insbesondere Leerzeichen und Tabulatoren — steht für sich selbst; sie werden **nicht ignoriert**.
- Die Metazeichen $() \backslash \wedge \$ \cdot [] * + ? \{ \} |$ müssen mit \backslash **quotiert** werden, falls sie wörtlich gemeint sind.
- Das Zeichen $/$ muss mit \backslash quotiert werden, falls es in einem Regulären Ausdruck vorkommt.
- Die Wiederholung mittels $\{m, n\}$ ist nur bei Angabe der Option `--re-interval` erlaubt (nur *Gawk*).
- Redundante Klammern können bei passendem Vorrang weggelassen werden.

2.2.4.2 POSIX-Zeichenklassen

Innerhalb von Zeichenlisten in $[\dots]$ sind auch **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:

| Gawk | 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 |

Beispiel:

```
/[[:alpha:]][[:alnum:]]_+/ # Ein Bst., dann bel. viele Zif.+Bst.+Unterstrich
```

2.3 Operatoren

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

| IC | Operator | Bedeutung |
|----|-------------------------|--|
| | (...) | Klammerung/Gruppierung |
| * | \$n | Feld-Zugriff auf Feld n ($\$1, \dots, \NF) |
| | ++ -- | Inkrementieren, Dekrementieren (<i>Präfix</i> und <i>Postfix</i>) |
| *r | ^ ** | Potenzierung x^y (** nur Gawk) |
| | + - ! | <i>Unäres</i> Plus/Minus (Vorzeichen), logisch NICHT |
| | * / % | Multiplikation, Division, Modulo (Divisionsrest) |
| | + - | Addition, Subtraktion |
| * | (space) | Verkettung von Zeichenket. (<i>kein expliziter Operator!</i>) |
| * | & | Ein/Ausgabe per Pipe (getline, print und printf) |
| | < > | Vergleich (<i>Zahlen und Zeichenketten</i>) |
| | <= >= | Vergleich (<i>Zahlen und Zeichenketten</i>) |
| | == != | Vergleich (<i>Zahlen und Zeichenketten</i>) |
| * | ~ !~ | Vergleich mit Reg. Ausdruck , negiert (<i>matched by</i>) |
| * | in | Test ob Index/Schlüssel in Array vorhanden |
| | && | Logisch UND (<i>short-cut evaluation</i>) |
| | | Logisch ODER (<i>short-cut evaluation</i>) |
| r | ?: | Bedingter Ausdruck (<i>cond ? true-case : false-case</i>) |
| 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 gesamte Eingabezeile, 1. Feld, letztes Feld, vorletztes Feld):

```
$0 $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
```

Dieser Unterschied ist nur in **Zuweisungen oder Ausdrücken** relevant. Stehen die Operationen für sich (so wie im Beispiel), dann besteht kein Unterschied zwischen der Präfix- und der Postfix-Form.

- Die Operatoren ^ und ** führen eine Potenzierung x^y durch (** nur *Gawk*) .
- 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 (**Modulo**).
- 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**: bitte durch ein Leerzeichen trennen) 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=" ")
```

- Der Koprozess-Operator |& ist nur im *Gawk* verfügbar.
- Die Operatoren <, >, <=, >=, == und != vergleichen numerisch oder textuell, je nach Typkombination der verglichenen Werte (siehe Abschnitt 2.7.4 auf Seite 36).
- 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. Ist der Wert des gesamten Ausdrucks bereits nach der Auswertung des ersten Elements (0 bei UND und 1 bei ODER) bekannt, dann werten sie den zweiten Operanden *nicht* mehr aus:

```
0 && printf("aaa")    # Ausgabe findet NICHT statt
1 && printf("bbb")    # Ausgabe findet statt
0 || printf("ccc")    # Ausgabe findet statt
1 || printf("ddd")    # Ausgabe findet NICHT statt
\end{exaprint}
```

\item

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

```
\begin{example}
if (COND)
    var = TRUE-CASE
else
    var = FALSE-CASE
\end{example}
```

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

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

- Alle Operatoren außer Zuweisungen, `?:`, `^` 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:** Zuweisungs-Operatoren `~=` `&=` `|=` `<<=` `>>=`, Bit-Operatoren `~` `&` `^` `|`, Shift-Operatoren `<<` `>>` und Komma-Operator `,` aus *C* existieren nicht.

2.4 Ausdrücke (Expressions)

Ein **Ausdruck** (engl. **Expression**) entsteht durch die Verknüpfung von **Operanden** (Konstante, Felder, Variable, Arrayelemente und Funktionsaufrufe) über **Operatoren** (siehe Abschnitt 2.3 auf Seite 19). Jeder Ausdruck hat einen **Wert**:

- Bei **numerischen Ausdrücken** ein numerischer Wert.
- Bei **Zeichenkettenausdrücken** eine Zeichenkette.
- Bei **logischen Ausdrücken** ein logischer Wert 0 (**true**) oder 1 (**false**).

Beispiele:

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

Hinweis: 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 17 in einen logischen Wert umgewandelt. Umgekehrt können auch logische Ausdrücke als numerischer Ausdruck oder Zeichenkettenausdruck interpretiert werden, sie haben dann den Wert 1/"1" (true) oder 0/" " (false).

2.5 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.*

2.5.1 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:

- **Muster fehlt:** Für alle Eingabesätze wahr, d.h. die Aktion wird für alle Eingabesätze ausgeführt.
- **Aktion fehlt:** Entspricht der Standardaktion `{ print $0 }`, d.h. die aktuelle Eingabezeile wird ausgegeben.

2.5.1.1 Muster

Folgende Muster sind möglich:

| Muster | Bedeutung |
|--|--|
| BEGIN END | Preprocessing <i>vor</i> dem Einlesen der Dateien Postprocessing <i>nach</i> dem Einlesen aller Dateien |
| Ausdruck /REGEXP/ | Logische Bedingung Matcht Eingabezeile |
| Ausdruck, Ausdruck /REGEXP/, /REGEXP/ | Bereichsmuster auf Basis logischer Bedingungen Bereichsmuster auf Basis Regulärer Ausdrücke |

- 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 2.4 auf Seite 21). Es ist wahr, falls es ausgewertet für die aktuelle Eingabezeile den Wert wahr ergibt.
- Das Muster `/REGEXP/` ist ein beliebiger Regulärer Ausdruck. Es ist wahr, falls es zur aktuellen Eingabezeile paßt.
- Die beiden letzten Muster heißen **Bereichsmuster**. Ein Bereichsmuster ist anfangs falsch. Es wird wahr, sobald die aktuelle Eingabezeile den ersten Ausdruck erfüllt und es wird falsch, nachdem die aktuelle Eingabezeile den zweiten Ausdruck erfüllt (einschließlich). Dies kann sich beliebig oft für die Eingabedaten wiederholen, auf diese Weise können „geklammerte“ = von Schlüsselbegriffen eingerahmte (zusammenhängende) Bereich der Eingabedaten bearbeitet werden. **Hinweis:** Im Extremfall trifft eine Eingabezeile auf beide Teile eines Bereichsmusters zu, dann wird nur sie von ihm erfasst.

2.5.1.2 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 |
|----|--|
| | <pre>break continue do <i>Anweisung</i> while (<i>Ausdruck</i>) exit [<i>Ausdruck</i>] for (<i>Ausdruck1</i>; <i>Ausdruck2</i>; <i>Ausdruck3</i>) <i>Anweisung</i> * for (<i>Variable</i> in <i>Array</i>) <i>Anweisung</i> if (<i>Ausdruck</i>) <i>Anweisung1</i> [else <i>Anweisung2</i>] * next * nextfile (Gawk) return [<i>Ausdruck</i>] while (<i>Ausdruck</i>) <i>Anweisung</i></pre> |
| * | <pre><i>Ein/Ausgabe-Anweisung</i> (getline, print und printf) delete <i>Arrayelement/Array</i> <i>Ausdruck</i> (mit Konstanten, Variablen, Zuweisungen, Funktionsaufrufen, ...) <i>Funktion</i> (<i>Argumente</i>) { <i>Anweisung</i>; ... } (Block)</pre> |
| * | <pre>switch (<i>Ausdruck</i>) { case <i>Wert Regex</i>: <i>Anweisung</i> ... default: <i>Anweisung</i> }</pre> |

- Da in den möglichen Anweisungen wieder *Anweisung* als Bestandteil auftaucht, handelt es sich um eine **rekursive Definition**. Für die Definition von Ausdrücken siehe Abschnitt 2.4 auf Seite 21.
- 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 einmal zu Beginn der Schleife ausgeführt (**Initialisierung**), *Ausdruck3* wird jedesmal nach der Ausführung von *Anweisung* ausgeführt (**Fortschaltung**). *Anweisung* wird eventuell überhaupt nicht ausgeführt (falls *Ausdruck2* bereits beim ersten Mal falsch ist).
 - ▷ `if (Ausdruck) Anweisung1 [else Anweisung2]` (**Verzweigung**) — führt *Anweisung1* aus, falls *Ausdruck* wahr ist. Ist der `else`-Teil vorhanden, so wird *Anweisung2* ausgeführt, falls *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 (falls *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. **Hinweis:** Kein Leerzeichen zwischen Funktionsname und (erlaubt).
 - ▷ { *Anweisung* ... } (**Block**) — faßt mehrere Anweisungen zusammen, die dann eine nach der anderen ausgeführt werden (Sequenz).
 - ▷ `switch` (*Ausdruck*) ... (**Fallunterscheidung**) ist nur bei Angabe der Option `--enable-switch` im *Gawk* erlaubt. Der *Ausdruck* wird ein Mal ausgewertet und sein Ergebnis der Reihe nach mit dem *Wert* bzw. dem Regulären Ausdruck *Regex* der einzelnen `case`-Fälle verglichen. Beim ersten Treffer wird die zugehörige Anweisung ausgeführt und anschließend zum Ende von `switch` gesprungen. Trifft kein einziger `case`-Fall zu, dann wird die Anweisung nach `default`: ausgeführt, falls sie vorhanden ist.
Hinweis: Es ist kein `break` zum Verlassen eines `case` notwendig und es ist nur eine Anweisung nach jedem `case` und nach `default` erlaubt.
- 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:** 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:** Wie in *C* oder *Perl* kann jede Anweisung durch ein Semikolon abgeschlossen werden (muss aber nicht).
 - Ein alleinstehendes Semikolon steht für die **leere Anweisung**, z.B. in einer leeren `for`-Schleife:

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

- Kommt mehrere 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, falls 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, falls sie in der gleichen Zeile wie das `while` auftritt:

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

2.5.2 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 **Argumente** (Ausdrücke) übergeben werden, deren Werte der Reihe nach in den **Positionsparametern** der Funktion landen. Sie beeinflussen das Verhalten der Funktion, 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 (sonst entsteht wegen dem Operator Leerraum zur Konkatenation von Zeichenketten eine Mehrdeutigkeit).

2.5.2.1 Definition

Eine **Funktionsdefinition** hat die Form (im *Gawk* statt `function` auch `func` erlaubt):

```
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. Das Schlüsselwort `function` *muss* bei der Definition einer Funktion vor dem Funktionsnamen angegeben werden.

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

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. Sie werden beim Aufruf der Funktion durch die an den entsprechenden Positionen übergebenen Argumente **initialisiert**. 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 = a # tmp = lokale Variable
    if (tmp < b) tmp = b # Alle 4 Variablen sind nur
    if (tmp < c) tmp = c # in dieser Funktion existent;
    return tmp # sie überdecken gleichnamige
                # globale Variablen darin.
}

```

2.5.2.2 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:

```

max3(100, 200, 300) # Ignorieren
max = max3(100, 2 * 53 - 1, -123) # Einer Variablen zuweisen
print 10 * max3(-1, -2, -3) # In einem Ausdruck verwenden

```

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 Argumente muss nicht mit der bei der Funktions-Definition vorgegebenen Anzahl von Parametern 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* Argumente ignoriert.

2.5.3 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 (außer zwischen Funktionsname und ()). **Hinweis:** Zum Einrücken gemäß der **Blockstruktur** Tabulatoren verwenden.
- **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 <code>print</code> , <code>printf</code> , Funktionsdefinition und -aufruf) |
| { | Linke geschweifte Klammer (Blockbeginn) |
| ? | Bedingte Operation (nur <i>Gawk</i>) |
| : | Bedingte Operation (nur <i>Gawk</i>) |
| && | Logisch Und |
| | Logisch Oder |
| if (...) | Bedingung |
| else | Alternative |
| for (...) | Zählschleife |
| while (...) | Abweisende Schleife |
| do | Nicht abweisende Schleife |
| } | Rechte Klammer in einer <code>if</code> -, <code>for</code> - oder <code>while</code> -Anweisung |

- Anweisungen 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).
- Die geschweifte Klammer auf `{` zwischen einem Muster und den Anweisungen der zugehörigen Aktion **muss auf der gleichen Zeile** wie das Muster stehen:

```

MUSTER {      # MUSS auf der gleichen Zeile stehen
    AKTION
}

```

2.5.4 Strukturierung von Programmen

Die **Verschachtelungs-Struktur** eines Programms sollte durch geeignetes **Einrücken** 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 (Muster-Aktion Paaren) und Funktionen sollte folgende **Reihenfolge** eingehalten werden:

| |
|----------|
| BEGIN |
| Regel |
| ... |
| END |
| Funktion |
| ... |

2.6 Programmablauf

Die Anweisungen eines *Awk*-Programms werden ebenso wie die Eingabedaten in einer bestimmten Reihenfolge abgearbeitet. Dabei werden bestimmte **interne Variablen** benutzt bzw. automatisch mit Werten belegt.

2.6.1 Vereinfachte Version

Zunächst folgt eine vereinfachte Beschreibung des Programmablaufs, um die prinzipiellen Schritte und Verhaltensweisen besser überblicken zu können. Im nächsten Abschnitt [2.6.2](#) auf Seite [31](#) werden die einzelnen Punkte des Programmablaufes noch um die fehlenden Details ergänzt.

1. Initialisierung einiger interner Variablen findet statt.
2. Aktion des **BEGIN-Musters** wird ausgeführt (falls vorhanden).
3. Eingabedaten werden aus angegebenen Dateien oder von Standard-Eingabe zeilenweise eingelesen und verarbeitet:
 - (a) **Eine oder mehrere Eingabedateien als Argumente angeben:**
Variable `FILENAME` erhält Namen der aktuell eingelesenen Eingabedatei zugewiesen.

Eingabedateien werden der Reihe nach Satz für Satz (terminiert durch den **aktuellen Satztrenner** `RS`) in Variable `$0` eingelesen.

Variable `FNR` erhält Anzahl bisher aus aktueller Eingabedatei gelesenen Sätze zugewiesen.

(b) **Keine Eingabedatei als Argument angegeben:**

Variable `FILENAME` erhält Wert "-" zugewiesen.

Von Standard-Eingabe wird Satz für Satz (terminiert durch **aktuellen Satztrenner** `RS`) in Variable `$0` eingelesen.

Variable `FNR` erhält Anzahl bisher gelesener Sätze zugewiesen.

4. Jeder eingelesene Satz wird gemäß **aktuellem Feldtrenner** `FS` in die Felder `$1` bis `$NF` zerlegt und `NF` wird gleich Anzahl Felder gesetzt (siehe auch die Abschnitte 2.7.1/2.8 auf den Seiten 33/37).

Für jeden gelesenen Satz wird Variable `NR` um 1 erhöht, er entspricht also immer der aktuellen (Gesamt)Anzahl gelesener Sätzen.

Für jeden gelesenen Satz wird Variable `FNR` um 1 erhöht, zu Beginn jeder gelesenen Datei wird sie auf 0 zurückgesetzt.

5. Nachdem ein Satz eingelesen und in Felder zerlegt ist, werden *alle Muster in der Reihenfolge ihrer Definition* überprüft. Ist ein **Muster** (für 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 Standard-Eingabe eingelesen wurden, wird die Aktion des **END-Musters** ausgeführt (falls vorhanden).

2.6.2 Ergänzungen

- zu 1.** Interne Variablen `FS`, `RS`, `OFS`, `ORS`, `OFMT` und `SUBSEP` werden mit **Defaultwert** belegt (siehe Abschnitt 2.8 auf Seite 37).

Programmname und **Argumente** werden im Array `ARGV` abgelegt und ihre Anzahl in `ARGC` abgelegt, `ARGIND` erhält Wert 1 zugewiesen.

Umgebungsvariable werden im Array `ENVIRON` abgelegt.

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

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

Zu jeder `-f`-Option 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.

in Abschnitt 2.1 auf Seite 14 aufgeführte Optionen werden interpretiert und aus Kommandozeile entfernt.

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

Defaultwerte können in der `BEGIN`-Aktion geändert werden, bevor der ersten Satz eingelesen oder ausgegeben wird. Ebenso können dort 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. Defaultwerte für 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 Variable `var` 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 4

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.

2.7 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.

2.7.1 Felder

Die aktuelle Eingabezeile wird gemäß dem Feldtrenner `FS` automatisch in **Felder** zerlegt (Sonderfall: hat `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 komplette unveränderte Eingabezeile (inkl. aller Feldtrenner). 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 |
|------------------|--|
| <code>\$0</code> | Aktueller Eingabesatz |
| <code>\$n</code> | Feld <i>n</i> des aktuellen Eingabesatzes (<code>\$1..\$NF</code>) |

- 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)` darf angesprochen werden; es hat dann `0/" "` als Wert; durch eine Zuweisung kann es belegt werden

2.7.2 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.

Variablen haben keinen festen Datentyp und 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/" "`.

2.7.3 Assoziative 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 und ihre Länge 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. ein Array 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 Arrays erfolgt über einen durch `[...]` eingeschlossenen (**Array**)**Index**, als Index können **Zahlen und Zeichenketten** verwendet werden. Die **assoziativen Arrays** in Awk entsprechen also **Hashes** in anderen Programmiersprachen. Beispiel:

```
arr[123] = "Text1"
arr["abc"] = 789
```

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

aufzählen (*allerdings in keiner bestimmten Reihenfolge*). **Hinweis:** Bei **mehrdimensionalen Arrays** darf nur *ein* Index `i` angegeben werden, der anschließend per `split` in seine Komponenten zerlegt und dann erst 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` werden auf einen Schlag **gelöscht** durch:

```
delete arr
```

2.7.4 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:

- ▷ Zahlenkonstante oder Ergebnis eines numerischen Ausdrucks → Typ **num**.
- ▷ Zeichenkettenkonstante oder Ergebnis einer Zeichenkettenoperation → Typ **str**.
- ▷ Felder, `getline`-Eingaben, `FILENAME`, `ARGV`-Elemente, `ENVIRON`-Elemente und von `split` erzeugte Arrayelemente →
 - * Typ **strnum**, falls sie wie Zahlen aussehen.
 - * Typ **str** sonst.

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
"  -978.564" + 0  # --> -987.564
```

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

```
$1++
```

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

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

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

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

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

beschreibt folgende Tabelle den **Vergleichstyp** abhängig vom Typ der Argumente:

| | str | strnum | num |
|---------------|------------|---------------|------------|
| str | str | str | str |
| strnum | str | num | num |
| num | 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** ("Null") 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.

2.8 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 ARGIND ARGV | Anzahl Kommandozeilenargumente (Programmname in ARGV[0]) Index der aktuell eingelesenen Eingabedatei in ARGV[1..ARGC-1] Array der Kommandozeilenargumente (ARGV[0..ARGC-1]) |
| * | BINMODE | Binäre Ein/Ausgabe für Dateien (1/r = nur Eingabe-Dateien, 2/w = nur Ausgabe-Dateien, 3/rw = Ein+Ausgabe-Dateien) |
| * | CONVFMT | Format f. autom. Umwandlung Zahl → Zeichenkette (Default: "%.6g") |
| * | ENVIRON | Assoziatives Array mit Umgebungsvar. und ihrem Wert (Environment) |
| * | ERRNO | Systemfehlermeldung bei <code>getline</code> und <code>close</code> |
| * | FIELDWIDTHS FILENAME | Liste mit durch Leerz. getrennten festen Spaltenbreiten (statt FS) Name der aktuellen Eingabedatei |
| | FNR FS | Satznummer der aktuellen Eingabedatei Eingabefeldtrenner (Default: "_" = Whitespace) |
| * | IGNORECASE | Groß/Kleinschreibung beim Vergleich mit Regulärem Ausdruck und in Zeichenketten-Funktionen ignorieren (Default: <i>false</i>) |
| * | LINT | Schalter <code>--lint dyn.</code> steuern (<i>true</i> =Lint Warnungen, <i>false</i> =keine, <i>fatal</i> =Abbruch bei Lint) |
| | NF NR | Anzahl Felder im aktuellen Satz Anzahl bisher eingelesener Sätze insgesamt |
| | OFMT OFS ORS | Ausgabeformat für Zahlen in <code>print</code> (Default: "%.6g") Ausgabe-Feldtrenner in <code>print</code> (Default: "_") Ausgabe-Satztrenner in <code>print</code> (Default: "\n") |
| * | PROCINFO | Array mit Informationen zum Awk-Prozess (<i>gid, uid, ...</i>) |
| | RLENGTH RSTART | Länge des passenden Textes nach <code>match</code> (-1 falls nicht gefunden) Beginn des passenden Textes nach <code>match</code> (0 falls nicht gefunden) |
| * | RS RT | Eingabesatztrenner (Default: "\n") Zu RS passender Text nach dem Einlesen jedes Satzes |
| * | SUBSEP TEXTDOMAIN | Trennzeichen für mehrdim. Index [<i>i, j, ...</i>] (Default: "\034") Textdomäne des Programms für Übersetzung von Texten |

- **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] (**argument vector**) — enthalten die beim Programmaufruf angegebenen Argumente (Dateinamen). Sie können jederzeit verändert/gelöscht/erweitert werden (z.B. im BEGIN-Abschnitt).
- **BINMODE (binary mode)** — nur für Nicht-Posix-Systeme (z.B. Windows). Unterbindet die automatische Umwandlung von "\r\n" nach "\n" (Standard-Zeilentrenner in UNIX-Systemen ist "\n").
- **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"]` 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. Beispiel:

```
BEGIN { FIELDWIDTHS = "10 5 8" } # Sätze in 3 Felder der
{ ... } # Breite 10, 5, 8 zerlegen
```

- **FILENAME** — enthält den Namen der **aktuellen Eingabedatei**. Hat den Wert `"-"` (Standard-Eingabe), falls 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** `"c"` (außer Leerzeichen) wird selbst als Feldtrenner verwendet. Kommt dieses Zeichen 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 (nicht `"_"`).
 - ▷ Bei einem **Regulärer Ausdruck** werden die zu diesem Regulären Ausdruck passenden Textteile als Feldtrenner verwendet. Paßt dieser Reguläre Ausdruck am Zeilenanfang, mehrfach hintereinander oder am Zeilenende vor, so werden *leere Felder* erzeugt.
 - ▷ Eine **leere Zeichenkette** `"` sorgt dafür, dass die Zeichen der Eingabezeile einzeln in je einem Feld abgelegt werden (nur *Gawk*).
- **IGNORECASE** — belegt mit dem Wert *true* 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*).
- **LINT** — schärfere Prüfung des *Awk*-Programms durchführen und entsprechende Warnungen ausgeben. Ein Wert von `"fatal"` bricht das *Awk*-Programm ab, falls eine Lint-Warnung vorkommt (analog `--lint=fatal`).

- **NF (number of fields)** — enthält für die aktueller Eingabezeile die Anzahl der Felder.
- **NR (number of records)** — enthält die Anzahl der bisher insgesamt eingelesenen Sätze (über alle Eingabedateien hinweg).
- **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"`).
- **PROCINFO (process information)** — ist ein Array mit mehreren automatisch gefüllten Elementen mit Informationen zum Awk-Prozess:

| Element | Bedeutung |
|----------------------|---|
| <code>uid</code> | Benutzer-ID (Systemaufruf <code>getuid</code>) |
| <code>euid</code> | Effektive Benutzer-ID (Systemaufruf <code>geteuid</code>) |
| <code>gid</code> | Gruppen-ID (Systemaufruf <code>getgid</code>) |
| <code>egid</code> | Effektive Gruppen-ID (Systemaufruf <code>getegid</code>) |
| <code>pid</code> | Prozess-ID (Systemaufruf <code>getpid</code>) |
| <code>ppid</code> | Eltern-Prozess-ID (Systemaufruf <code>getppid</code>) |
| <code>pgrp</code> | Prozess-Gruppen-ID (Systemaufruf <code>getpgrp</code>) |
| <code>group1</code> | Weitere Gruppe, in der Prozess Mitglied ist |
| <code>group2</code> | Weitere Gruppe, in der Prozess Mitglied ist |
| <code>...</code> | Weitere Gruppen, in der Prozess Mitglied ist |
| <code>version</code> | Gawk-Version |
| <code>FS</code> | "FS" falls Feldzerlegung mit <code>FS</code> aktiv, "FIELDWIDTHS" falls Feldzerlegung mit <code>FIELDWIDTHS</code> aktiv |

- **RS (record separator)** — trennt die Eingabedaten in Sätze auf und kann durch folgende Werte definiert werden (**Default:** `"\n"`):
 - ▷ Ein **Einzelzeichen** als Wert sorgt dafür, dass dieses als Satztrenner verwendet wird.
 - ▷ Ein **Regulärer Ausdruck** als Wert sorgt dafür, dass die zu diesem Regulären Ausdruck passenden Textteile als Satztrenner verwendet werden (nur *Gawk*).
 - ▷ Die **leere Zeichenkette** `RS=""` sorgt dafür, dass eine oder mehrere *Leerzeilen* die Datensätze trennen und die einzelnen Zeilen eines durch Leerzeilen eingerahmten „Blocks“ die Datensatzelemente darstellen. Damit können **mehrzeilige Records** verarbeitet werden. Als Feldtrenner `FS` sollte dann `"\n"` verwendet werden. Die Zeilen eines solchen durch Leerzeilen begrenzten „Blocks“ werden dann zu Feldern der „Blockzeile“ `$0`.
- **RSTART (regex start)** und **RLENGTH (regex 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 (falls kein Treffer), `RLENGTH` enthält die Länge des passenden Textes (1..n) oder -1 (falls kein Treffer), `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/record terminator)** — ist nach jedem Einlesen eines Einlegesatzes 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
```

- **TEXTDOMAIN** — definiert die **Textdomäne** des Programms, d.h. in welche Sprache sollen mit `dcgettext()` und `dcngettext()` angegebene Zeichenketten konvertiert werden.

2.9 Vordefinierte Funktionen

2.9.1 Arithmetik-Funktionen

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

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

2.9.2 Bit-Funktionen

Diese Funktionen sind nur ab *Gawk 3.1* vorhanden! Sie wandeln Fließkommazahlen in vorzeichenlose Long-Ganzzahlen um, führen dann die Bitoperation aus und wandeln das Ergebnis anschließend wieder in eine Fließkommazahl um.

| Gawk | Funktion | Bedeutung |
|------|----------------------------|---|
| * | <code>and (a, b)</code> | <i>a</i> und <i>b</i> bitweise UND-verknüpfen |
| * | <code>or (a, b)</code> | <i>a</i> und <i>b</i> bitweise ODER-verknüpfen |
| * | <code>xor (a, b)</code> | <i>a</i> und <i>b</i> bitweise XOR-verknüpfen |
| * | <code>compl (a)</code> | <i>a</i> bitweise invertieren (Komplement) |
| * | <code>lshift (a, n)</code> | <i>a</i> um <i>n</i> Bits nach links schieben |
| * | <code>rshift (a, n)</code> | <i>a</i> um <i>n</i> Bits nach rechts schieben |

2.9.3 Zeichenketten-Funktionen

s und *t* sind Zeichenketten, *r* ein Regulärer Ausdruck, *n* und *i* sind Zahlen, *fmt* ist eine Zeichenkette mit %-Formatierungsangaben (siehe Abschnitt 2.9.6 auf Seite 45), *list* ist eine durch Kommas getrennte Liste von Argumenten, in eckigen Klammern eingeschlossene Parameter können weggelassen werden (* = nur im Gawk vorhanden):

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

- Wird bei `match` keine passende Zeichenkette gefunden, so enthält `RSTART` den Wert 0 und `RLENGTH` den Wert -1.
- `split` eignet sich 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.
- In `gensub` wird `\n` ($n = 1..9$) durch den im Suchmuster im n -ten Klammernpaar gemerkten Text ersetzt (analog `Sed`).
- In `match` wird Array `a[1]` mit dem Treffer des ersten Klammerpaars in `r` gefüllt, `a[2]` mit dem Treffer des zweiten Klammerpaars, usw. `a[0]` enthält den gesamten Treffer von `r` in `s`. `a[0, "start"]` und `a[0, "length"]` enthalten Startposition und Länge der in `r` geklammerten Treffer in der Zeichenkette `s`.

2.9.4 Array-Funktionen

Diese Funktionen sind nur im `Gawk` vorhanden (`delete` ist eigentlich keine Funktion, sondern eine Anweisung).

| Gawk | Funktion | Bedeutung |
|------|----------------------------------|---|
| * | <code>length(arr)</code> | Anzahl Elemente von Array <code>arr</code> |
| * | <code>delete arr</code> | Array <code>arr</code> vollständig löschen |
| * | <code>delete arr[ind]</code> | Arrayelement <code>arr[ind]</code> löschen |
| * | <code>asort(src[, dest])</code> | Werte von Array <code>src</code> direkt bzw. nach Array <code>dest</code> je nach Typ numerisch oder alphabetisch sortieren, Anzahl Elemente von Array <code>src</code> zurückgeben |
| * | <code>asorti(src[, dest])</code> | Indices von Array <code>src</code> direkt bzw. nach Array <code>dest</code> grundsätzlich alphabetisch sortieren, Anzahl Elemente von Array <code>src</code> zurückgeben |

- Bei `asort` werden die **Werte** von Array `src` sortiert und gemäß der Sortierung mit neuen Indices 1..n versehen, die **Originalindices gehen verloren**.
- Bei `asorti` werden die **Indices** von Array `src` sortiert und mit den Indices 1..n versehen zu Werten, die **Originalwerte gehen verloren**.

2.9.5 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, kann auch eine Textvariable sein), `var` ist eine Variable, `fmt` ist eine Zeichenkette mit %-Formatierungsangaben (siehe Abschnitt 2.9.6 auf Seite 45), `list` ist eine durch Kommas getrennte Liste von Argumenten, in eckigen Klammern eingeschlossene Parameter können weggelassen werden (* = nur im `Gawk` vorhanden):

| Gawk | Funktion | Bedeutung |
|------|---|--|
| | <code>close (file)</code> | Datei <i>file</i> schließen |
| | <code>close (cmd)</code> | Kommando <i>cmd</i> (Pipe) schließen |
| * | <code>close (cmd)</code> | Koprozess <i>cmd</i> (Pipe) schließen |
| * | <code>close (cmd, "to")</code> | Koprozess <i>cmd</i> (Pipe hin) schließen |
| * | <code>close (cmd, "from")</code> | Koprozess <i>cmd</i> (Pipe zurück) schließen |
| * | <code>fflush ([file])</code> | Gepufferte Daten der Datei <i>file</i> schreiben |
| * | <code>fflush ([cmd])</code> | Gepufferte Daten des Kommandos <i>cmd</i> schreiben |
| | <code>getline</code> | §0 mit nächster Eingabezeile belegen; setzt NF, NR, FNR |
| | <code>getline < file</code> | §0 mit nächster Eingabezeile aus <i>file</i> belegen; setzt NF |
| | <code>getline var</code> | <i>var</i> mit nächster Eingabezeile belegen; setzt NR, FNR |
| | <code>getline var < file</code> | <i>var</i> mit nächster Eingabezeile aus <i>file</i> belegen |
| | <code>cmd getline</code> | Ergebnis von BS-Kommando <i>cmd</i> in §0 ablegen; setzt NF |
| | <code>cmd getline var</code> | ... in <i>var</i> ablegen |
| * | <code>cmd & getline</code> | Ergebnis von BS-Koprozess <i>cmd</i> in §0 ablegen; setzt NF |
| * | <code>cmd & getline var</code> | ... in <i>var</i> ablegen (Koprozess) |
| | <code>print</code> | Aktuelle Eingabezeile §0 ausgeben (mit Return) |
| | <code>print list</code> | Ausdrücke <i>list</i> ausgeben |
| | <code>print list > file</code> | ... auf <i>file</i> ausgeben (<i>file</i> vorher löschen) |
| | <code>print list >> file</code> | ... an <i>file</i> anhängen (<i>file</i> vorher <i>nicht</i> löschen) |
| | <code>print list cmd</code> | ... an BS-Kommando <i>cmd</i> übergeben |
| * | <code>print list & cmd</code> | ... an BS-Koprozess <i>cmd</i> übergeben (läuft parallel) |
| | <code>printf fmt, list</code> | Ausdrücke <i>list</i> gemäß <i>fmt</i> formatieren und ausgeben |
| | <code>printf fmt, list > file</code> | ... auf <i>file</i> ausgeben (<i>file</i> vorher löschen) |
| | <code>printf fmt, list >> file</code> | ... an <i>file</i> anhängen (<i>file</i> vorher <i>nicht</i> löschen) |
| | <code>printf fmt, list cmd</code> | ... an BS-Kommando <i>cmd</i> übergeben |
| | <code>printf fmt, list & cmd</code> | ... an BS-Koprozess <i>cmd</i> übergeben (parallel) |
| | <code>system (cmd)</code> | BS-Kommando <i>cmd</i> 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).
- `close()` gibt den Wert von `fclose(3)` oder `pclose(3)` zurück, wenn eine Ausgabedatei oder eine Ausgabepipe geschlossen wird. Es gibt den Exit-Status des Prozesses zurück, wenn eine Eingabepipe geschlossen wird. Der Rückgabewert ist `-1`, falls die Datei, Pipe oder der Koprozess nicht geöffnet werden konnte (nur *Gawk*).
- `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 | Bei erfolgreichem 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"      # Ohne Klammern
printf("%d Beispiele\n", cnt)              # Mit Klammern
```

- Bei Ausgaben mit `print` werden durch `,` (Komma) getrennte Argumente durch den Ausgabe-Feldtrenner `OFS` (**Default:** `"_ "`) getrennt und der ganze Satz wird mit dem Ausgabebtrenner `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.

Bei Ausgaben mit `printf` sind `OFS` und `ORS` nicht relevant. Sämtliche Formatierungszeichen (insbesondere der Zeilenvorschub `\n`) müssen im Format-String stehen.

- `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.
- Bei Koprozessen per `|&` ist ein Aufruf von `close()` notwendig, wenn EOF gefunden wird (macht *Gawk* nicht automatisch).

2.9.6 Printf-Formatumwandlung

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

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

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

| Gawk | Parameter | Bedeutung |
|------|--------------|---|
| | - | Ausdruck linksbündig ausrichten (Std: rechtsbündig) |
| | ␣ | Leerzeichen vor positiven Zahlen, - vor negativen |
| | + | + vor positiven Zahlen, - vor negativen |
| | 0 | Führende Nullen statt Leerzeichen (nur bei <i>width</i> relevant) |
| * | # | Alternatives Format bei: <ul style="list-style-type: none"> o = Führende Null ausgeben x/X = Führendes 0x oder 0X ausgeben eEf = Dezimalpunkt immer ausgeben gG = Nullen am Ende nicht entfernen |
| | <i>width</i> | Ausgabe in Breite <i>width</i> (rechtsbündig mit Leerzeichen), eine führende 0 verwendet dazu Nullen |
| | <i>.prec</i> | Anzahl Nachkommastellen <i>prec</i> bei Zahlen, maximale Breite <i>prec</i> bei Zeichenketten |
| * | * | Statt <i>width</i> oder <i>prec</i> den nächsten Parameter in der <code>printf</code> -Argumentliste verwenden (nur <i>Gawk</i>) |
| * | <i>n</i> \$ | Positionsparameter: <i>n</i> -tes Argument der Argumentliste einsetzen |

- 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.

Die **Positionsparameter** *n*\$ sind zur Internationalisierung von Zeichenketten gedacht. Beispiel:

```
printf("%3$s %1$s %2$s", "eins", "zwei", "drei") # --> drei eins zwei
```

2.9.7 Zeit-Funktionen

Die Zeit-Funktionen sind *Gawk*-spezifisch, sie sind vor allem dazu gedacht, Zeiten zu vergleichen bzw. Datum- 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 |
|---|---|
| <code>systeme()</code> | Aktuelle Zeit in Anzahl Sekunden (<i>sec</i>) seit 1.1.1970 1:00 UTC |
| <code>strftime([<i>fmt</i>[, <i>sec</i>]])</code> | Sekunden <i>sec</i> gemäß <i>fmt</i> formatieren Defaultformat: "%a %b %d %H:%M:%S %Z %Y" Defaultzeit: <code>systeme()</code> |
| <code>mktime(<i>datespec</i>)</code> | Datum der Form "YYYY MM DD HH MM SS [DST]" (6/7 Elemente) in Zeitstempel (Anzahl Sek. seit 1.1.1970 1:00 UTC) konvertieren. |

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 | Wochenummer (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 |

2.9.8 Internationalisierung-Funktionen

Die Funktionen zur Internationalisierung (Übersetzung) von Texten sind erst ab *Gawk 3.1* verfügbar.

| Funktion | Bedeutung |
|---|--|
| <code>bindtextdomain (dirpath[, dom])</code> | Verzeichnis <i>dirpath</i> für <i>.mo</i> -Dateien festlegen gemäß Textdomäne <i>dom</i> (Std: TEXTDOMAIN) |
| <code>dcgettext (s[, dom[, cat]])</code> | Übersetzung von <i>s</i> gemäß Textdomäne <i>dom</i> (Std: TEXTDOMAIN) und Lokalkategorie <i>cat</i> (Std: LC_MESSAGES) |
| <code>dcngettext (s, p, n[, dom[, cat]])</code> | Singularform <i>s</i> ($n = 1$) oder Pluralform <i>p</i> ($n > 1$) gemäß Zahl <i>n</i> übersetzen gemäß Textdomäne <i>dom</i> (Std: TEXTDOMAIN) und Lokalkategorie <i>cat</i> (Std: LC_MESSAGES) |

Mit der Option `--gen-po` wird aus den lokalisierbaren Strings eine *.po*-Datei erzeugen.

2.10 Koprozesse

Koprozesse werden durch `print ... |& command` und `command |& getline` erzeugt. An einen Koprozess können sowohl Daten gesendet als auch von ihm gelesen werden. Mit `close(command)` kann der Koprozess beendet werden, mit `close(command, "to")` nur dessen Eingabekanal, mit `close(command, "from")` nur dessen Ausgabekanal.

2.11 Grenzen

Jede konkrete Implementierung von *Awk* hat einige Grenzen. Typische Werte sind (dem *Awk*-Buch entnommen; moderne Implementierungen, insbesondere der *Gawk*, haben weit

größzügigere Grenzen):

| Bedeutung | Awk | Gawk |
|---|------|----------|
| Anzahl Felder | 100 | ∞ |
| Zeichen pro Eingabesatz | 3000 | ∞ |
| Zeichen pro Ausgabesatz | 3000 | ∞ |
| Zeichen pro Feld | 1024 | ∞ |
| Zeichen pro <code>printf</code> -Zeichenkette | 3000 | ∞ |
| Zeichen maximale Zeichenkettenlänge | 400 | ∞ |
| Zeichen in einer Zeichenliste | 400 | ∞ |
| Anzahl offene Dateien gleichzeitig | 55 | ∞ |
| Anzahl offene Pipes gleichzeitig | 15 | ∞ |

3 Tipps und Tricks

3.1 Aufrufparameter

- Argumente der Form `var=text` direkt zu Beginn der Argumentliste werden von verschiedenen *Awk*-Versionen unterschiedlich behandelt. Dateinamen dieser Form führen nicht zum Einlesen einer Datei, sondern setzen **zum Zeitpunkt des Lesens** die entsprechende Variable `var` auf den Wert `text` und werden aus der Argumentliste entfernt. Bei einigen *Awk*-Versionen sind diese Variablen bereits im `BEGIN`-Muster gesetzt, bei anderen nicht, wenn sie am Anfang der Argumentliste vorkommen. 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*, falls 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 2.7.4 auf Seite 35).

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, \dots, \NF ist erlaubt, dabei wird $\$0$ aus allen Feldern mit dem Ausgabe-Feldtrenner *OFS* dazwischen neu erstellt.
- Eine Zuweisung an ein nicht existierendes Feld $\$(NF+1), \dots$ 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

- Im *Gawk* ergibt die eingebaute Funktion `length` angewendet auf ein Array die Anzahl der Elemente eines 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 `sort` und wieder einlesen).
- Im *Gawk* gibt es die eingebauten Funktionen `asort` und `asorti` zum **Sortieren von Arraywerten bzw. Arrayindices**. `for (i in arr) ...` durchläuft auch nach dem Sortieren mit `asort` bzw. `asorti` die Array-Elemente in keiner bestimmten Reihenfolge.

Durchlauf in Wert-Reihenfolge nach Sortieren per `asort` muss erfolgen durch:

```
for (i = 1; i <= length(sorted); ++i) print sorted[i].
```

Durchlauf in Index-Reihenfolge nach Sortieren per `asorti` muss erfolgen durch:

```
for (i = 1; i <= length(sorted); ++i) print unsorted[sorted[i]].
```
- 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 Zeichenketten-Konkatenation verwendet werden:

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

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 |
|--------------------------|--|
| <code>/dev/stdin</code> | Standard-Eingabe (auch <code>-</code> , <code>/dev/fd/0</code> , <code>/dev/tty</code>) |
| <code>/dev/stdout</code> | Standard-Ausgabe (auch <code>/dev/fd/1</code>) |
| <code>/dev/stderr</code> | Standard-Fehlerkanal (auch <code>/dev/fd/2</code>) |
| <code>/dev/fd/n</code> | File verbunden mit Dateideskriptor/Kanal <code>n</code> |

- Der `Gawk` benutzt folgende speziellen Dateinamen für TCP/IP-Netzwerkverbindungen im Rahmen von `|&`-Koprozessen (Portnummer `0` = Port automatisch wählen):

| Datei | Bedeutung |
|--|--|
| <code>/inet/tcp/lport/rhost/rport</code> | TCP/IP-Verbindung auf lokalem Port <code>lport</code> zu Rechner <code>rhost</code> mit remote Port <code>rport</code> |
| <code>/inet/udp/lport/rhost/rport</code> | UDP/IP-Verbindung auf lokalem Port <code>lport</code> zu Rechner <code>rhost</code> mit remote Port <code>rport</code> |
| <code>/inet/raw/lport/rhost/rport</code> | Reserviert |

- 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 |
|--------------------------|--|
| <code>/dev/pid</code> | Aktuelle Prozeß-ID |
| <code>/dev/ppid</code> | Prozeß-ID des Elternprozesses |
| <code>/dev/pgrpid</code> | Aktuelle Prozeß-Gruppen-ID |
| <code>/dev/user</code> | 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` geschlossen wurde.
- Lesen von und Schreiben in eine Datei gleichzeitig ist nicht möglich (kein wahlfreies Positionieren in Dateien möglich).
- 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/)    # Regulärer Ausdruck in /.../
match($0, "test")   # Regulärer Ausdruck als String
PAT = "te" "st"     # Variable zusammenbauen
match($0, PAT)      # Regulärer Ausdruck als Variable
```

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

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

Im *Gawk* kann statt `sub/gsub` die Funktion `gensub` verwendet werden. Diese merkt sich geklammerte Teile im Regulären Ausdruck, die dann im Ersetzungsteil per `\1... \9` angesprochen werden können.

3.9 Mehrzeilige Eingabesätze (Datenblöcke)

- 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 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*).
- Name einer normalen Variable auch als Array- oder Funktionsname verwendet.
- Name einer (vordefinierten) Funktion als Variablenname verwendet (z.B. `index`).
- Array-Zugriff a la C auf einzelne Zeichen einer Zeichenkette `string` per Indizierung `[i]` ist nicht möglich. 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.
- `break` und `continue` außerhalb von Schleifen verwendet.
- `case` und `default` außerhalb von `switch` verwendet.

3.10.2 Besondere Verhaltensweisen

- `$0` und `$1...$NF` sind im `BEGIN`-Teil nicht belegt, im `END`-Teil enthält `$0` die letzte gelesene Zeile.
- `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.
- `FNR` hat im `BEGIN`-Teil den Wert 0, im `END`-Teil enthält es die Anzahl der gelesenen Sätze der letzten Eingabedatei.
- `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).
- Ist eine Eingabedatei `FILE` nicht vorhanden, 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 Array-index `" "` 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 → automatisch neue Variable erzeugt.
- Funktion mit zu wenig/zu vielen Argumenten aufgerufen (wird nicht erkannt).
- Anzahl der Parameter in einer `printf`-Anweisung stimmt nicht mit 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, Zuweisung statt Vergleich → immer wahr).

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. Im *Gawk* kann sie durch die Option `--enable-switch` aktiviert 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 -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 'BEGIN { PREFIX="" "$PREFIX" " " }
     { 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 < ARGV; ++ARGIND) {
    # Dateiname hat Form VAR=TEXT? --> Zuweisung + nächste Datei
    if (ARGV[ARGIND] ~ /^[A-Za-z_][A-Za-z_0-9]*=/) {
        eval ARGV[ARGIND]; # Zuweisung VAR = TEXT 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
```

Die implizite Eingabeschleife und das Muster-Aktion-Paradigma vereinfachen den Kontrollfluss bzw. machen ihn oft unnötig:

```
FORALL (Eingabesätze) DO
    IF Pattern1 THEN { Action1 }
    IF Pattern2 THEN { Action2 }
    ...
    IF PatternN THEN { ActionN }
END DO
```

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 (falls 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 sehr 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 `true/false` (`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 + -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
- `multiline` — Mehrzeilige Datensätze — DATA
- `pipe` — Pipe in/aus Shell-Kommando
- `system` — Aufruf von System-Kommandos
- `time` — Zeit-Funktionen und -formate

4.3 UNIX-Werkzeuge simulieren

- `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 durchnummerieren
- `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 breitem 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 breitem 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) und sind hier nicht dargestellt. 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 < ~

| Dez | 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 | |
|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|------------|
| 0 | ^@ [NUL] | ^P | [SPACE] | 0 | @ | P | ` | p | 0 |
| 1 | ^A | ^Q | ! | 1 | A | Q | a | q | 1 |
| 2 | ^B | ^R | " | 2 | B | R | b | r | 2 |
| 3 | ^C | ^S | # | 3 | C | S | c | s | 3 |
| 4 | ^D | ^T | \$ | 4 | D | T | d | t | 4 |
| 5 | ^E | ^U | % | 5 | E | U | e | u | 5 |
| 6 | ^F | ^V | & | 6 | F | V | f | v | 6 |
| 7 | ^G [BEL] | ^W | ' | 7 | G | W | g | w | 7 |
| 8 | ^H [BS] | ^X | (| 8 | H | X | h | x | 8 |
| 9 | ^I [TAB] | ^Y |) | 9 | I | Y | i | y | 9 |
| 10 | ^J [LF] | ^Z | * | : | J | Z | j | z | A |
| 11 | ^K [VTAB] | ^[[ESC] | + | ; | K | [| k | { | B |
| 12 | ^L [FF] | ^\ ^_ | , | < | L | \ | l | | C |
| 13 | ^M [CR] | ^] | - | = | M |] | m | } | D |
| 14 | ^N | ^^ | . | > | N | ^ | n | ~ | E |
| 15 | ^O | ^_ | / | ? | O | _ | o | [DEL] | F |
| | 00 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | Hex |

Hinweise:

- ^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.