```
Betriebssysteme - Werkzeuge und UNIX-Schnittstelle
==================================================

                 Werkzeuge
                 =========


8. awk - Alfred V. Aho, J. Weinberger, Brian W. Kernighan
          -       -                    -
=========================================================
  Ein Filter oder eine leistungsfähige Programmiersprache?
```


Historisches:

```
1977 von Alfred V. Aho, J. Weinberger, Brian W. Kernighan entwickelt
1985 grundlegend verbessert
1990 aufgenommen in POSIX 1003.2 Command Language  And  Utilities
     Standard
1992 gawk POSIX-konform mit Erweiterungen!!!
1995 verbesserte Version von gawk
1998 verbesserte Version von gawk (POSIX 1003.2 Regel 11.2)
```

Allgemeines:

- Mit awk lassen sich Dateien nach Textmustern durchsuchen und wenn diese erkannt werden, verschiedene Aktionen ausführen (ersetzen, streichen, merken).
- Syntax entspricht in groben Zügen der Syntax von C
- awk wird vorrangig in der Systemprogrammierung zum Bearbeiten von Konfigurationsdateien eingesetzt.
  Weiterhin ist awk für die einmalige komplexe Modifikation von großen Dateien geeignet.
- awk ist langsam!!!!! Wenn man häufig große Datenbestände mit awk durchmustern will, sollte man das awk-Programm in ein C-Programm umformen.
- awk-Programme sind in der Regel kurz!!!!!

awk-Einführung

Arbeitsweise von awk:

awk liest die Eingabedatei zeilenweise. Jede Eingabezeile wird in mehrere Eingabefelder zerlegt. Die Eingabefelder werden mit $1, $2, $3, ... bezeichnet. $0 bezeichnet die Eingabezeile. Danach wird für jede Zeile das awk-Programm abgearbeitet und die entsprechenden Anweisungen ausgeführt.

Aufruf von awk:

```
awk   [-F<Feldseparator>] '<awk-Programm>' {<Datei>}
awk   [-F<Feldseparator>] -f <awk-Programm-File> {<Datei>}
```

```
<awk-Programm>         - vollständiges awk-Programm, kann sich über
                         mehrere Zeilen erstrecken. <NL> sollte durch
                         '\' maskiert sein.

-F<Feldseparator>      - Feldseparator ist ein Zeichen. Dieses Zeichen
                         definiert das Trennzeichen zwischen zwei
                         Eingabefeldern in der Eingabedatei.
                         Standard ist das Leerzeichen.
                                                       a1,a2,a3,a4,a5
-f <awk-Programm-File> - File, das ein awk-Programm enthält
                                                                   a6
<Datei>                - Name der Datei, die bearbeitet werden soll. Wird
                         keine Datei angegeben, werden die Daten von der
                         Standardeingabe gelesen.                  a7
```


Einfache awk-Programme

awk-Programm:

Folge von Anweisungen der Form:

```
<Muster>  {  <Aktion> }
<Muster>  {  <Aktion> }
      .....
```

oder formal (unvollständig):

```
<awk-Programm> ::=  <Pattern> ["{" <Folge von Anweisung> "}"] |
     <awk-Programm> <NL> <Pattern> ["{"<Folge von Anweisung>"}"]
<Folge von Anweisungen> ::=  <Anweisung> { "<NL>" <Anweisung> }
<Anweisung> ::=   <print-Anweisung> | ...
<Pattern> ::=  <arithmetische Vergleiche> | <Textvergleiche> |
               "BEGIN" | "END" | ...
```

```
<print-Anweisung> ::= print [<Ausdruck> {"," <Ausdruck>}]
```

Ausgabe unformatiert auf die Standardausgabe. Zwischen den Ausdrücke wird ein Leerzeichen ausgegeben. Eine leere print-Anweisung bewirkt die Ausgabe der Eingabezeile.

a8,a9

Built-in-Variable:

NF - Anzahl der Eingabefelder in der aktuellen Zeile
NR - Aktuelle Zeilennummer

a10

```
<arithmetische Vergleiche>  -   "<" und ">"
                 Arithmetische Ausdrücke können verglichen werden.

<Textvergleich>  - "=="
                Nur Strings können verglichen werden.

BEGIN-Pattern - Die zum BEGIN-Pattern gehörende Folge von Anweisungen
                wird einmal vor dem Lesen der ersten Zeile der Eingabe-
                datei abgearbeitet.  Initialisierung!!!!
                Achtung: BEGIN muß wirklich groß geschrieben werden!!!!

END-Pattern   - Die zum END-Pattern gehörende Folge von Anweisungen
                wird einmal nach dem Lesen der letzten Zeile der
                letzten Eingabedatei abgearbeitet.
                Achtung: END muß wirklich groß geschrieben werden!!!!
```

Variable sind möglich. Werden beim Auftreten definiert. Arithmetische Variable werden mit Null initialisiert. String-Variable werden mit der leeren Zeichenkette initialisiert.

a11,a12 a13,a14

arithmetische Operationen: +, -, * , /

```
    <arithmetischer Ausdruck> := <variable> | <Konstante> |
          <arithmetischer Ausdruck> <Operator> <arithmetischer Ausdruck>
```

Verkettungsoperationen für Strings:

```
             <string> string>
```

a15

Die Sprache AWK

```
<awk-Programm> ::=  <Pattern> [ "{" <Folge von Anweisung> "}" ]          |
                    <Funktionsdefinition>                                |
                    <awk-Programm> "<NL>" <Pattern>
                                        [ "{" <Folge von Anweisung> "}" ] |
                    <awk-Programm>  "<NL>" <Funktionsdefinition>

<Folge von Anweisungen> ::=  <Anweisung> | "#" [<Zeichenkette>] "<NL>" |
                          <Folge von Anweisungen> { "<NL>" <Anweisung> }  |
                          <Folge von Anweisungen> { ";" <Anweisung> }
```

Das Pattern und die zugehörige "{" müssen immer auf der selben Zeile stehen.

Wird nach einem Pattern keine <Folge von Anweisungen> angegeben, erfolgt die Ausgabe der Zeile, auf die das Pattern passt auf Standardausgabe.

a16

Anweisungen werden durch ";" getrennt, wenn sie auf einer Zeile stehen.

Ein ";" am Ende einer einzelnen Anweisung ist nicht schädlich.

Leerzeichen und Tabulatoren können zur Strukturierung des awk-Programms benutzt werden.

"#" leitet einen Kommentar ein, der bis zum Zeilenende reicht.

a17

```
<Pattern> ::= <BEGIN-Pattern> | <END-Pattern>   |
              <Ausdruck>                        |
              "/"<regulärer Ausdruck>"/"       |
              <zusammengesetztes Pattern>      |
              <Bereich>

  <BEGIN-Pattern> ::=  "BEGIN"

          Die zum BEGIN-Pattern gehörende Folge von Anweisungen wird einmal
          vor dem Lesen der ersten Zeile des ersten Eingabefiles abgearbeitet
          Der Anweisungsteil darf nicht leer sein!!!!

  <END-Pattern> ::=  "END"

          Die zum END-Pattern gehörende Folge von Anweisungen wird einmal
          nach dem Lesen der letzten Zeile des letzten Eingabefiles
          abgearbeitet.  Der Anweisungsteil darf nicht leer sein!!!!

  <Ausdruck>    ::=  <numerischer Ausdruck> | <String-Ausdruck>

          Jeder Ausdruck kann als Pattern benutzt werden. Ist der Wert
          des Ausdrucks ungleich Null, wird der Anweisungsteil ausgeführt,
          andernfalls nicht. Typische "Ausdrucks-Pattern" sind Vergleiche
          (arithmetische und string). Arithmetische Vergleiche werden
          ausgeführt, wenn beide Operanden arithmetisch sind, andernfalls
          wird ein String-Vergleich ausgeführt.
```



```
          Vergleichsoperatoren in Ausdrücken:

              Vergleichs-    Bedeutung                Type
              operator
                  <           kleiner als              n,s
                  <=          kleiner gleich           n,s
                  ==          gleich                   n,s
                  !=          ungleich                 n,s
                  >           größer als               n,s
                  >=          größer gleich            n,s
                  ~           rechter String im
                              linken String enthalten    s
                  !~          rechter String nicht im
                              linken String enthalten    s

           Tricks:
               Anhängen eines Nullstrings an eine Zahl bewirkt die
               Konvertierung der Zahl zu einem String.
               Addieren einer Null zu einem String bewirkt die
               Konvertierung eines Strings zu einer Zahl.
                                                                a18,a19

  <regulärer Ausdruck>   -  siehe oben
               Wie reguläre Ausdrücke:
                  Metazeichen:  \ ^ $ . [ ] | ( ) * + ?
                  Operatoren:   A | B (oder), AB (A gefolgt von B)
                                A* (0 bis n-mal A), A+ (1 bis n-mal A),
                                A? (0 oder 1-mal A)
                  Klassen von Zeichen:  [a-zA-z],  [^a-zA-Z]
                                                                 a20,a21
```

```
<zusammengesetztes Pattern>::=  <Pattern> || <Pattern>  |
                                  <Pattern> && <Pattern>  |
                                  "!" <Pattern> |
                                  "(" <Pattern> ")"
              Zusammensetzen von mehreren Pattern zu einem.
              ||     - oder
              &&     - und
              !      - Verneinung
              ( .. ) - Klammerung
                                                              a22,a23


<Bereich>  ::= <Pattern> "," <Pattern>

              Hierdurch wird ein Bereich festgelegt, der mit der Zeile
              beginnt, für die das erste Pattern paßt und endet mit
              der Zeile, für die das zweite Pattern paßt, bzw. bis
              zum Fileende. Die zugehörigen Anweisungen werden für alle
              Zeilen des Bereichs ausgeführt.
                                                             a24,a25


<Anweisung> ::=   <Ausdrücke> |
                  <print-Anweisung> | <printf-Anweisung> |
                  <getline-Anweisung> | <if-Anweisung> |
                  <for-Anweisung> | <for-array-Anweisung> | <do-Anweisung> |
                  <while-Anweisung> | <break-Anweisung> |
                  <continue-Anweisung> | <next-Anweisung> |
                  <exit-Anweisung> | "{" <Anweisung> { <Anweisung> }"}"
```



```
<Ausdrücke>

   <awk-Operatoren> - Übersicht

          Vorzeichen-Operatoren:       +, -

          Additions-Operatoren:        +, -

          Multiplikations-Operatoren:  *, /, %

          Potenz-Operator:             ^
                                               a^2   a zum Quadrat

          Zuweisungsoperatoren:        =, +=, -=, *=, /=, %=, ^=
                                               a+=3    -   a = a + 3

          Präfix-Operatoren:           ++, --
                                               y = ++z  -  z=z+1; y=z;

          Postfix-Operatoren:          ++, --
                                               y = z--  -  y=z; z=z-1;

          Relations-Operatoren:        <, <=, ==, !=, >=, >

          Match-Operatoren:            ~, !~

          Bediengungsoperator:         ?:
                                              (a?b:c - wenn a dann b sonst c)
```

Logische Operatoren: || - logisches oder
&& - logisches und
! - logische Verneinung

Enthaltens-Operator-Array: in - n in array - liefert True, wenn array[n] existiert

Klammerung: ( )

<Konstante>::=<String-Konstante> | <numerische Konstante>

<String-Konstante> : Zeichenkette in Doppelapostrophs geklammert
z.B. "Passwd", "\npasswd\n"

<numerische-Konstante>: ganze Zahlen, z.B. 123, 1234,4711
Gleitkommanzahlen, z.B. 3.1415, 1.0e-6, 1E-6

<Variable>::=<Normale-Variable> | <Built-in-Variable> | <Felder>

<Normale-Variable> : vom Anwender definierte Variable, Bezeichner aus Buchstaben, Unterstrichen und Ziffern bestehen mit Buchstaben oder Unterstrich beginnend.
z.B. i, i_, i_23

<Built-in-Variable> : von awk vordefinierte Variable:

- ARGC - Anzahl der Argumente
- ARGV - Feld der Argumente
- ENVIRON - Feld der Umgebungsvariablen
- FILENAME - Filename des momentan bearbeiteten Feldes
- FNR - Zeilennummer des aktuellen Files
- FS - Feldseparator
- NF - Anzahl der Felder in der aktuellen Zeile
- NR - Anzahl der gelesenen Zeilen insgesammt
- OFMT - Ausgabeformat für Zahlen - "%.6g"
- OFS - Trennzeichen für Ausgabefelder - " "
- ORS - Trennzeichen für Ausgabezeilen - "\n"
- RS - Trennzeich für Eingabezeile - "\n"

a26

<Felder>:

```
$-Feld: bezeichnet die Eingabezeile
              $0 - gesammte Eingabezeile
              $1 - erstes Feldelement
              $2 - zweite Feldelement
                             ....
```

a27

Nutzerdefinierte Felder:

awk unterstützt assoziative Felder, die nicht explizite definiert werden müssen. Sie sind bei der ersten Benutzung mit Null oder "" initialisiert.

```
z.B.  feld["drei"] = 3; feld["vier"] = 4;
      feldn[3] = 3; feldn[4] = 4;
      felda[3] = "drei"; felda[4] = "vier";
```

a28

in-Operator für for-Anweisung:

Ermittelt alle Indizes eines Feldes. Reihefolge ist Implementationsabhängig. siehe Beispiel

delete-Operation für Felder:

delete feld[index] - das "indexte" Element des Feldes feld wird gelöscht.


```
<print-Anweisung>::=
            "print" {<Ausdruck1>}
            "print" {<Ausdruck1>} ">" <Ausgabefile>
            "print" {<Ausdruck1>} ">>" <Ausgabefile>
            "print" {<Ausdruck1>} "|" <Kommando>
```

print

Ausgabe der Eingabezeile auf die Standardausgabe
entspricht "print $0"

print Ausdruck1, Ausdruck2, ...

Ausgabe der Ausdrücke Ausdruck1, Ausdruck2, ... auf die Standardausgabe. Zwischen den einzelnen Ausdrücken wird OFS (Output Field Separator) geschrieben. Nach dem letzten Ausdruck wird ORS (Output Record Separator geschrieben.

print Ausdruck1, Ausdruck2, ... > Ausgabefile

Ausgabe der Ausdrücke Ausdruck1, Ausdruck2, ... in das Ausgabefile. Zwischen den einzelnen Ausdrücken wird OFS (Output Field Separator) geschrieben. Nach dem letzten Ausdruck wird ORS (Output Record Separator geschrieben. Der alte Inhalt des Ausgabefiles wird überschrieben.

a29

print Ausdruck1, Ausdruck2, ... >> Ausgabefile

Anfügen der Ausdrücke Ausdruck1, Ausdruck2, ... in das Ausgabefile. Zwischen den einzelnen Ausdrücken wird OFS (Output Field Separator) geschrieben. Nach dem letzten Ausdruck wird ORS (Output Record Separator geschrieben. Die neuen Daten werden an den Inhalt des Ausgabefiles angefügt.

a30

```
print Ausdruck1, Ausdruck2, ... | kommando
    Ausgabe der Ausdrücke Ausdruck1, Ausdruck2, ... auf die
    Standeingabe des Kommandos  kommando. Zwischen den einzelnen
    Ausdrücken wird OFS (Output Field Separator) geschrieben.
    Nach dem letzten Ausdruck wird ORS (Output Record Separator
    geschrieben.
```

a31

printf-Anweisung

```
<printf-Anweisung>::=
        "printf" "("<format> {,<Ausdruck>} ")" |
        "printf" "("<format> {,<Ausdruck>} ")" ">"  <Ausgabefile> |
        "printf" "("<format> {,<Ausdruck>} ")" ">>" <Ausgabefile> |
        "printf" "("<format> {,<Ausdruck>} ")" "|"  <Kommando>
```

Formatgesteuerte Ausgabe von Ausdrücken. Die Formate sind ähnlich wie bei C.
Der format-String besteht aus Zeichen ungleich "%", die unverändert ausgegeben werden und Formatsteuerungselemente.
Es folgen die für den format-String notwendigen Ausdrücke.

Ein Formatsteuerungselement hat folgenden Aufbau:

```
    %FWGU

F - Formatierungszeiche: "-"  -  linksbündige Justierung
W - Weite: Mindestanzahl der auszugebenden Zeichen
G - Genauigkeit: für Strings - Maximalzahl der auszugebenden
                               Zeichen
                 für Zahlen  - Anzahl der Stellen nach dem
                               Komma
    Der Wert wird als .<Ziffern> ausgegeben.

U - Umwandlungszeichen:
         c  - ASCII-Zeichen
         d  - Dezimalzahl mit Vorzeichen
         e  - Geitkommanzahl der form [-]d.dddddE[+-]dd
         f  - Geitkommanzahl der form [-]ddd.dddddd
         g  - die kürzere Darstellung von e oder f Umwandlung
         o  - vorzeichenlose Oktalzahl
         s  - String
         x  - vorzeichenlose Hexadezimalzahl
         %  - "%"-Zeichen

printf (format,Ausdruck1, Ausdruck2, ... )
    Ausgabe der Ausdrücke Ausdruck1, Ausdruck2, ... unter
    Berücksichtigung der Formatelemente in  format auf die
    Standardausgabe.  Es werden keine OFS und ORS geschrieben.
```

a32

printf (format,Ausdruck1, Ausdruck2, ...) > Ausgabefile
Ausgabe der Ausdrücke Ausdruck1, Ausdruck2, ... unter Berücksichtigung der Formatelemente in format in das Ausgabefile. Es werden keine OFS und ORS geschrieben. Der alte Inhalt des Ausgabefiles wird überschrieben.

printf (format,Ausdruck1, Ausdruck2, ...) >> Ausgabefile
Anfügen der Ausdrücke Ausdruck1, Ausdruck2, ... unter Berücksichtigung der Formatelemente in format in das Ausgabefile. Es werden keine OFS und ORS geschrieben. Die neuen Daten werden an den Inhalt des Ausgabefiles angefügt.

printf (format,Ausdruck1, Ausdruck2, ...) | Kommando
Ausgabe der Ausdrücke Ausdruck1, Ausdruck2, ... unter Berücksichtigung der Formatelemente in format auf die Standeingabe des Kommandos kommando. Es werden keine OFS und ORS geschrieben.

Sowohl bei print als auch bei printf kann die Verbindung zur Ausgabefile bzw. zum Kommando durch:

```
close(Ausgabefile)
     bzw.
close(Kommando)
```

abgebrochen werden.

getline-Anweisung

```
<getline-Anweisung>::= "getline" |
                       "getline" <Variable> |
                       "getline" "<" <Dateiname> |
                       "getline" <Variable> "<" <Dateiname> |
                       <Kommando> "|" "getline" |
                       <Kommando> "|" "getline" <Variable>
```

getline-Anweisung wird benutzt, um von der momentanen Eingabe, von einem File oder von der Standardausgabe eines Kommandos ein Zeile zu lesen. Die Zeile kann dabei in den Standardeingabepuffer oder in eine Variable gelesen werden.
Wird in den Standardeingabepuffer gelesen, wird die Zeile wie gewohnt in Felder aufgeteilt.
Wird die Zeile in eine Variable gelesen, so findet keine Aufteilung statt.
Die Variablen NR und FNR werden jeweils weitergezählt.
Die Variable NF wird nur bei der Eingabe in den Standardeingabepuffer gefüllt.

```
Rückkehrwert: 1 - eine Zeile gelesen
              0 - Dateiende erreicht
             -1 - Fehler beim Lesen
```

Klammerung von Anweisungen:

```
<Anweisung>::="{" <Anweisung> { <Anweisung> }"}"| ....
```

Durch die Klammerung in geschweiften Klammern können mehrere Anweisungen zu einer Anweisung zusammengefaßt werden. Dies wird für die nachfolgenden komplexen Anweisungen benötigt.

if-Anweisung

```
<if-Anweisung>::= "if" "(" <Ausdruck> ")" <Anweisung 1>
                                [ "else" <Anweisung 2> ]
```

Bemerkung zur Syntax:
Für <Anweisung 1> und <Anweisung 2> darf genau eine Anweisung stehen. Diese kann auch auf einer neuen Zeile stehen. Stehen <Anweisung 1> und der "else"-Teil auf der selben Zeile, müssen sie durch ein ";" getrennt werden. Sollen mehrere Anweisungen auf der Position von <Anweisung 1> bzw. <Anweisung 2> abgearbeitet werden, so sind diese zu klammern (siehe oben).

Funktionsweise: Der Ausdruck wird ausgewertet. Ist der Wert des Ausdrucks ungleich Null oder ein Nicht-Null-String, so wird die erste Anweisung abgearbeitet, andernfalls wird die Anweisung nach dem "else"-Teil abgearbeitet, falls dieser vorhanden ist.

a33,a34

for-Anweisung

```
<for-Anweisung>::="for" "(" <Ausdruck 1> ";" <Ausdruck 2> ";"
                             <Ausdruck 3> ")" <Anweisung>
```

Bemerkung zur Syntax:
Für <Anweisung> darf genau eine Anweisung stehen. Diese darf auf einer neuen Zeile stehen. Sollen mehrere Anweisungen auf der Position von <Anweisung> abgearbeitet werden, so sind diese zu klammern (siehe oben).

Funktionsweise:

1. <Ausdruck 1> wird berechnet.
2. <Ausdruck 2> wird berechnet. Ist der Wert des Ausdrucks ungleich Null (True) wird bei 3. fortgesetzt, sonst wird die for-Anweisung beendet.
3. Abarbeitung des Anweisungsteil
4. Berechnung des <Ausdruck 3> und fortsetzen bei 2.

a27

for-array-Anweisung

```
<for-array-Anweisung>::= "for" "(" <Variable> "in" <Feld> ")"
                                   <Anweisung>
```

Funktionsweise:
Der Variablen wird nacheinander die existierenden Indizes des Feldes zugewiesen. Jedesmal wird anschließend die Anweisung abgearbeitet. Die Reihenfolge des Indizes ist unbestimmt (implementations-abhängig).

a28

do-Anweisung

<do-Anweisung>::="do" <Anweisung> "while" "("<Ausdruck>")"

Funktionsweise:

1. **Abarbeiten der Anweisung.**
2. **Berechnung des Ausdrucks.**
3. **Wenn der Ausdruck ungleich Null ist (True), wird bei 1. fortgesetzt, sonst wird die do-Anweisung beendet.**

a35

while-Anweisung

<while-Anweisung>::="while" "("<Ausdruck>")" <Anweisung>

Funktionsweise:

1. **Berechnung des Ausdrucks.**
2. **Wenn der Ausdruck ungleich Null ist (True), wird 3. ausgeführt, sonst wird die while-Anweisung beendet.**
3. **Abarbeiten der Anweisung und fortfahren bei 1.**

a35

break-Anweisung

<break-Anweisung>::="break"

Funktionsweise:

Sofortiges Verlassen der aktuellen Schleifenanweisung (for-Anweisung, while-Anweisung, do-Anweisung). Bei mehrfach ineinander geschachtelten Schleifenanweisungen wird nur die innerste Schleife verlassen.

a35

continue-Anweisung

<continue-Anweisung>::= "continue"

Funktionsweise:

Die continue-Anweisung bewirkt das Beenden des aktuellen Schleifendurchlaufs bei einer Schleifenanweisung (for-Anweisung, while-Anweisung, do-Anweisung). Anschließend werden die notwendigen Ausdrücke berechnet und eventuell mit dem nächsten Schleifendurchlauf fortgesetzt.

a36

next-Anweisung

<next-Anweisung>::="next"

Funktionsweise:
Es wird sofort die nächste Eingabezeile gelesen und mit der Auswertung des ersten Patterns des awk-Programms fortgesetzt. Bei EOF wird awk beendet.

a37

exit-Anweisung

<exit-Anweisung>::="exit" [<Ausdruck>]

Funktionsweise:
Beenden des awk-Programms. Befindet sich die exit-Anweisung in einem END-Anweisungs-Teil, wird die das awk-Programm sofort beendet. In allen anderen Teilen eines awk-Programms bewirkt die exit-Anweisung die sofortige Verzweigung zum END-Anweisungs-Teil, wenn dieser vorhanden ist, sonst ebenfalls die sofortige Beendigung des awk-Programms. Ist ein Ausdruck in der exit-Anweisung spezifiziert, wird dieser als Rückkehrkode von awk benutzt.

a38,a39

built-in-Funktionen

In awk vordefinierte Funktionen. Werden von awk mitgeliefert. Die Zahl ist unterschiedlich.

arithmetische built-in-Funktionen

```
atan2(x,y)    - Arcustangens von x/y

cos(x)        - Cosinus von x im Bogenmaß

sin(x)        - Sinus von x im Bogenmaß

exp(x)        - Exponential-Funktion e hoch x

log(x)        - natürlicher Logarithmus von x: ln x

int(x)        - ganzzahliger Anteil von x

sqrt(x)       - Quadratwurzel von x

rand()        - Zufallszahl im Intervall von 0 bis 1

srand(x)      - neuer Startwert x für den Zufallszahlen-
                generator
```

a40

String built-in-Funktionen

```
re - regulärer Ausdruck     s1,s2 - Strings       ar - Feld
tr - Trennzeichen           n1,n2 - ganze Zahlen

gsub(re,s1,s2)   - Ersetzt im String s2 den regulären Ausdruck re
                   durch den String s1
                   Rückkehrwert: Anzahl der Ersetzungen

gsub(re,s1)      - wie gsub, aber anstelle von s2 wird $0
                   benutzt
                   Rückkehrwert: Anzahl der Ersetzungen

index(s1,s2)     - Sucht das erste Auftreten des Strings s2
                   in s1
                   Rückkehrwert: Anfangsposition von s2 in s1

length(s1)       - Berechnet die Länge von s1
                   Rückkehrwert: Länge des Strings s1

match(s1,re)     - prüft, ob der String s1 einen Teilstring
                   enthält, der mit re "matchet".
                   Rückkehrwert: Anfangsposition des Teilstrings
                          oder 0

split(s1,ar,tr)  - Zerlegt den String s1 entsprechend dem Trennzeichen
                   tr in ein Feld ar.
                   Rückkehrwert: Anzahl der Feldelemente
```



```
split(s1,ar)     - wie split(s1,ar,tr), aber als Trennzeichen wird FS
                   genommen.
                   Rückkehrwert: Anzahl der Feldelemente

sprintf(fmt,format-liste) - Formatierte Ausgabe
                   Rückkehrwert: Ergebnis der Ausgabe als String

sub(re,s1,s2)    - Substitution des am weitesten links stehenden
                   längsten Strings re durch den String s1 im String s2
                   Rückkehrwert: Anzahl der Ersetzungen

sub(re,s1)       - wie  sub(re,s1,s2), aber in $0
                   Rückkehrwert: Anzahl der Ersetzungen

substr(s1,n1,n2) - schneide aus String s1 ab der Position n1 einen
                   String der Länge n2 aus
                   Rückkehrwert: Ausgeschnittene Teilstring

substr(s1,n1)    - liefert ab Position n1 den Rest des Strings s1
                   Rückkehrwert: Rest des Strings ab Position n1

tolower(s1)      - Konvertieren der Großbuchstaben in Kleibuchstaben
                   s1 bleibt unverändert
                   Rückkehrwert: Konvertierte String s1

toupper(s1)      - Konvertieren der  Kleibuchstaben in Großbuchstaben
                   s1 bleibt unverändert
                   Rückkehrwert: Konvertierte String s1
```

a41

```
<Funktionsdefinition>

   Benutzer definierte Funktionen. Können bei Bedarf in das awk-Program
   eingefügt werden, auch als File mit Funktionsdefinitionen.

    <Funktionsdefinition>::="function" <Funktionsname>"("<Parameterlist>")"
                             "{" <Anweisung> { <Anweisung> }"}"

     "return" <Ausdruck>
                                                                           a42
   Funktionsbibliothek

   Eine Funktionsbibliothek ist ein File mit nützlichen Funktionen, das wie
   das eigentliche awk-Script mittels einer -f Option beim awk-Aufruf
   angegeben wird, es sind also mehrere -f Optionen zulässig.

      awk -f /home/unixsoft/bell/awklib -f prog.awk beispieldatei

                                                                           a43
```



```
Vollständiger Programmaufruf von awk
------------------------------------

 awk [POSIX-Optionen oder GNU-Optionen] [--] 'program' file ...

    Optionen:
        -f progfile, --file=progfile
             File mit awk-Programm, mehrer derartiger Optionen sind
             möglich

        -F<fs>, --field-separator=fs
             Feldseparator, Standard ist Leerzeichen

        -v var=val, --assign=var=val
             Definition einer awk-Variablen <var> mit dem Wert <val> vor
             Begin der Abarbeitung des awk-Programms

        -mf val
             Maximalwert für die Anzahl von Feldern in einer Zeile

        -mr val
             Maximalwert für die Länge eines Satzes

        -W compat, --compat
             Kompatibilitätsmodus, gawk verhält sich wie awk

        -W copyleft, --copyleft
             Ausgabe des kurzen Copyrights
```

```
-W copyright, --copyright
     Ausgabe des langen Copyrights

-W version, --version
   Ausgabe der Versionsinformationen

-W dump-variables[=file], --dump-variables[=file]

-W gen-po, --gen-po

-W help , --help
-W usage, --usage
     Ausgabe einer kurzen Hilfe-Information

-W lint[=fatal], --lint[=fatal]
     Ausgabe von Warnungen für awk-Konstruktionen, die nicht
     portable sind.

-W lint-old, --lint-old
     Ausgabe von Warnungen für awk-Konstruktionen, die nicht mit
     dem ursprünglichen awk verträglich sind.

-W non-decimal-data, --non-decimal-data

-W profile[=file], --profile[=file]

-W posix, --posix
   Posix-Kompatibilitätsmodus - nur reguläre Ausdrücke im POSIX-
   Format werde unterstützt.
```



```
-W re-interval, --re-interval
   Intervall-Ausdrücke in Regulären Asudrücken erlauben

-W source=program-text, --source=program-text
   wie -f Option - awk-Programmtext

-W traditional, --traditional
   Reguläre Ausdrücke entsprechend dem ursprünglichen awk-Format
   werden unterstützt.
```

Abschlußbeispiele
----

1. Verwaltung der Datenbasis eines DNS-Servers:

```
add_dns <Hostname>  - hinzufügen eines neuen Hosts zur Datenbasis
  add_dns.awk   - modifizieren von db.informatik
  add_dns1.awk  - modifizieren von db.141.20.xx
del_dns <Hostname>    -  streichen eines Hosts aus der Datenbasis
```