Arbeiten mit der Linux-Shell „BaSH”

Das Seminar wendet sich an Teilnehmerinnen mit geringen Linux- Kenntnissen; Vorerfahrungen können gleichwohl den Verständnis-Prozeß hie und da erleichtern. Es erstreckt sich über fünf Tage mit jeweils acht Unterrichts-Stunden à 45 Minuten bzw. vier Doppelstunden à 90 Minuten. Einschließlich der Pausen ergeben sich daraus das 40 Zeit-Stunden, die nach Absprache mit den Teilnehmerinnen aufgeteilt werden können auf pro Tag

9:00 Uhr bis 17:00 Uhr
oder

10.00 Uhr bis 18:00 Uhr.

Alle Zeit-Zuordnungen sind vage und vorläufig: Da dies das erste Seminar dieses Typs ist, müssen sie sich anhand der praktischen Arbeits-Erfahrungen erst noch genauer herausbilden. Aus dem gleichen Grunde sind die vorliegenden Seminar-Unterlagen noch nicht vollständig: Ihr genauer Inhalt und Umfang muß in der Validierungs-Phase erst noch ermittelt werden.

Bei den im Folgenden eingestreuten englisch-sprachigen Text-Teilen handelt es sich jeweils um Auszüge aus der einen oder anderen sog. *man-page* (einer ansonsten online einsehbaren Dokumentation), die es ermöglichen sollen, auch ohne Rechner-Zugriff auf bestimmte Informationen zurückzugreifen. Zugleich sollen sie dazu dienen, sich mit der englisch-sprachigen Terminologie vertraut zu machen. Da es sich hierbei jedoch stets um Text-Auszüge handelt, und jene Seiten zudem von Programm-Version zu Programm-Version überarbeitet oder erweitert werden, sollte bei der praktischen Arbeit stets auf die Original-Dateien zurückgegriffen werden.

Einführung / Allgemeines

[Montag:Vormittag]

Rechner-Hierarchie

Hardware - Software - Nochwer

Software-Hierarchie

BIOS / OS
Treiber / Dämonen
Anwendungs-Programme (u.a. sog. *Shells*)

OS - Gegenüberstellung

DOS : Interrupts
WinDoz : Botschaften
*nix : Minimalismus => externe Programme
(copy./.cp, ren./.mv, type./.cat, del./.rm &c.)

(!) Der grundlegende Aufbau eines PC-Systemes soll kennengelernt werden. Insbesondere die unterschiedliche „Philosophie'' von DOS- und Unix-basierten Betriebssystemen soll verstanden werden, um sich auch in heterogenen Rechner-Welten (Netzen) leichter orientieren zu können.

Das Linux-Datei-System

Während es unter Betriebssystemen wie DR-/MS-/PC-DOS (incl. seiner graphischen Benutzeroberflächen GEM und Windows) weitgehend der jeweiligen Nutzerin überlassen bleibt, welche Verzeichnis-Strukturen sie anlegt und welche Daten sie wo ablegt gibt es unter UNIX-basierten Betriebssystemen hierfür einige Konventionen. Um der Antwort auf die Frage: *Welche Dateien liegen wo?* näher zu kommen, hier zunächst eine schematische Darstellung eines Dateisystemes unter Linux:

```
/ --                the root directory
|
+-bin               Essential command binaries
+-boot              Static files of the boot loader
+-dev               Device files
+-etc               Host-specific system configuration
    |
    +-X11           Configuration for the X Window System
    +-opt           Configuration for /opt
+-home              User home directories
+-lib               Essential shared libraries and kernel modules
+-mnt               Mount point of temporary partitions
+-opt               Add-on application software packages
    |
    +-<package>     Static package objects
+-root              Home directory for the root user
+-sbin              Essential system binaries
+-tmp               Temporary files
+-usr               Secondary hierarchy
    |
    +-X11R6         X Window System, version 11 release 6
    +-X386          X Window System, version 11 release 5 on x86 platforms
    +-bin           Most user commands
        |
        +-mh        Commands for the MH mail handling system
        +-X11       Symlink to /usr/X11R6/bin
    +-games         Games and educational binaries
    +-include       Header files included by C programs
        |
        +-X11       Symlink to /usr/X11R6/include/X11
        +-bsd       BSD compatibility include files (if required)
        +-g++       GNU C++ include files
    +-lib           Libraries
    +-local         Local hierarchy (empty after main installation)
        |
        +-bin       Local binaries
        +-games     Local game binaries
        +-include   Local C header files
        +-lib       Local libraries
        +-sbin      Local  system binaries
        +-share     Local architecture-independent hierarchy
        +-src       Local source code
    +-sbin          Non-vital system binaries
    +-share         Architecture-independent data
        |
        +-dict      Word lists
        +-doc       Miscellaneous documentation
        +-games     Static data files for /usr/games
        +-info      GNU Info system's primary directory
        +-locale    Locale information
        +-man       Online manuals
            |
            +-man1  User programs
            +-man2  System calls
            +-man3  Library calls
            +-man4  Special files
            +-man5  File formats
            +-man6  Games
            +-man7  Miscellaneous
            +-man8  System administration
        +-nls       Native language support
        +-misc      Miscellaneous architecture-independent data
        +-terminfo  Directories for terminfo database
        +-tmac      troff macros not distributed with groff
        +-zoneinfo  Timezone information and configuration
    +-src           Source code
+-var               Variable data
    |
    +-account       Process accounting logs (if supported)
    +-cache         Application cache data
        |
        +-fonts     Locally-generated fonts
```

```
        +-man          Locally-formatted manual pages
        +-www          WWW proxy or cache data
        +-<package>    Package specific cache data
    +-crash            System crash dumps (if supported)
    +-games            Variable game data
    +-lock             Lock files
    +-log              Log files and directories
    +-mail             User mailbox files
    +-opt              Variable data for /opt
    +-run              Files relevant to running processes
    +-spool            Application spool data
        |
        +-lpd          Printer spool directory
            |
            +-<printer>   Spools for a specific printer (optional)
        +-mqueue       Outgoing mail queue
        +-news         News spool directory
        +-rwho         Rwhod files
        +-smail        Spool directories for smail
        +-uucp         Spool directory for UUCP
    +-state            Variable state information
        |
        +-<editor>     Editor backup files and state
        +-misc         Miscellaneous state data
        +-xdm          X display manager variable data
        +-<pkgtool>    Packaging support files
        +-<package>    State data for packages and subsystems
    +-tmp              Temporary files preserved between system reboots
    +-yp               Network Information Service (NIS) database files
```

Dieses Schaubild gibt gewissermaßen eine idealtypische Konfiguration wieder. Welche Verzeichnisse auf einer bestimmten Maschine hingegen tatsächlich vorhanden sind, richtet sich nach den dort installierten Programmen. Gründe und Hintergründe dieser Verzeichnisstruktur werden in den folgenden Auszügen aus dem „ *Filesystem Hierarchy Standard -- Version 2.0*“ der „Filesystem Hierarchy Standard Group“ vom Oktober 1997, detaillierter diskutiert wird.

The UNIX filesystem is characterized by:

o A hierarchical structure

o Consistent treatment of file data

o Protection of file data

This standard assumes that the operating system underlying an FHS-compliant file system supports the same basic security features found in most UNIX filesystems. Note that this standard does not attempt to agree in every possible respect with any particular UNIX system's implementation. However, many aspects of this standard are based on ideas found in UNIX and other UNIX-like systems.

This is after careful consideration of other factors, including:

o Traditional and well-considered practices in UNIX-like systems.

o The implementation of other filesystem structures

o Applicable standards

It is possible to define two orthogonal categories of files: shareable vs. unshareable and variable vs. static.

Shareable data is that which can be shared between several different hosts; unshareable is that which must be specific to a particular host. For example, user home directories are shareable data, but device lock files are not.

Static data includes binaries, libraries, documentation, and anything that does not change without system administrator intervention; variable data is anything else that does change without system administrator intervention.

For ease of backup, administration, and file-sharing on heterogenous networks of systems, it is desirable that there be a simple and easily understandable mapping from directories (especially directories considered as potential mount points) to the type of data they contain.

The distinction between shareable and unshareable data is needed for several reasons:

o In a networked environment (i.e., more than one host at a site), there is a good deal of data that can be shared between different hosts to save space and ease the task of maintenance.

o In a networked environment, certain files contain information specific to a single host. Therefore these filesystems cannot be shared (without taking special measures).

o Historical implementations of UNIX-like filesystems interspersed shareable and unshareable data in the same hierarchy, making it difficult to share large portions of the filesystem.

The "shareable" distinction can be used to support, for example:

o A **/usr** partition (or components of **/usr**) mounted (read-only) through the network (using NFS).

o A **/usr** partition (or components of **/usr**) mounted from read-only media. A CD-ROM can be considered a read-only filesystem shared with other FHS-compliant systems, using the postal mail system as a "network".

The "static" versus "variable" distinction affects the filesystem in two major ways:

o Since **/** contains both variable and static data, it needs to be mounted read-write.

o Since the traditional **/usr** contains both variable and static data, and since we may want to mount it read-only (see above), it is necessary to provide a method to have **/usr** mounted read-only. This is done through the creation of a **/var** hierarchy that is mounted read-write (or is a part of another read-write partition, such as **/**), taking over much of the **/usr** partition's traditional functionality.

Here is a summarizing chart. Since this chart contains generalized examples, it may not apply to every possible implementation of an FHS-compliant system.

```
+---------+-----------------+-------------+
|         | shareable       | unshareable |
+---------+-----------------+-------------+
|static   | /usr            | /etc        |
|         | /opt            | /boot       |
+---------+-----------------+-------------+
|variable | /var/mail       | /var/run    |
|         | /var/spool/news | /var/lock   |
+---------+-----------------+-------------+
```

3. The Root Directory

The contents of the root filesystem should be adequate to boot, restore, recover, and/or repair the system:

o To boot a system, enough must be present on the root partition to mount other filesystems. This includes utilities, configuration, boot loader information, and other essential start-up data. **/usr**, **/opt**, and **/var** are designed such that they may be located on other filesystems.

o To enable recovery and/or repair of a system, those utilities needed by an experienced maintainer to diagnose and reconstruct a damaged system should be present on the root filesystem.

o To restore a system, those utilities needed to restore from system backups (on floppy, tape, etc.) should be present on the root filesystem.

Software should never create or require special files or subdirectories in the root directory. Other locations in the FHS hierarchy provide more than enough flexibility for any package.

The operating system kernel image should be located in either **/** or **/boot**. Additional information on kernel placement can be found in the section regarding **/boot**, below.

3.1 **/bin** : Essential user command binaries (for use by all users)

/bin contains commands that may be used by both the system administrator and by users, but which are required in single user mode. It may also contain commands which are used indirectly by scripts.

There should be no subdirectories within **/bin**.

Command binaries that are not essential enough to place into **/bin** should be placed in **/usr/bin**, instead. Items that are required only by nonroot users (**mail**, **chsh**, etc.) are generally not essential enough to be placed into the root partition.

Required files for **/bin**:

o General commands

If **/bin/sh** is Bash, then **/bin/sh** should be a symbolic or hard link to **/bin/bash** since Bash behaves differently when called as **sh** or **bash**. The use of a symbolic link in these cases allows users to easily see that **/bin/sh** is not a true Bourne shell.

o Restoration commands

Restoration commands may appear in either **/bin** or **/usr/bin** on different systems.

o Networking commands

3.2 **/boot** : Static files of the boot loader

This directory contains everything required for the boot process except configuration files and the map installer. Thus **/boot** stores data that is used before the kernel begins executing user-mode programs. This may include saved master boot sectors, sector map files, and other data that is not directly edited by hand. Programs necessary to arrange for the boot loader to be able to boot a file should be placed in **/sbin**. Configuration files for boot loaders should be placed in **/etc**. The operating system kernel should be located in either **/** or **/boot**.

3.3 **/dev** : Device files

The **/dev** directory is the location of special or device files.

If it is possible that devices in **/dev** will need to be manually created, **/dev** shall contain a command named **MAKEDEV**, which can create devices as needed. It may also contain a **MAKEDEV.local** for any local devices. If required, **MAKEDEV** should have provisions for creating any device that may be found on the system, not just those that a particular implementation installs.

3.4 **/etc** : Host-specific system configuration

/etc contains configuration files and directories that are specific to the current system. No binaries should be located under **/etc**.

The following section is intended partly to illuminate the description of the contents of **/etc** with a number of examples; it is definitely not an exhaustive list. Required files for **/etc**:

o General files

o Networking files

3.4.1 **/etc/X11** : Configuration for the X Window System

/etc/X11 is the recommended location for all X11 host-specific configuration. This directory is necessary to allow local control if **/usr** is mounted read only. Files that should be in this directory include **Xconfig** (and/or **XF86Config**) and **Xmodmap**.

/etc/X11/xdm holds the configuration files for xdm. These are most of the files normally found in **/usr/lib/X11/xdm**. Some local variable data for xdm is stored in **/var/state/xdm**.

3.4.2 **/etc/opt** : Configuration files for **/opt**

Host-specific configuration files for add-on application software packages shall be installed within the directory **/etc/opt/<package>**, where **<package>** is the name of the subtree in **/opt** where the static data from that package is stored. No structure is imposed on the internal arrangement of **/etc/opt/<package>**.

3.5 **/home** : User home directories (optional)

/home is a fairly standard concept, but it is clearly a site-specific filesystem. The setup will differ from host to host. This section describes only a suggested placement for user home directories; nevertheless we recommend that all FHS-compliant distributions use this as the default location for home directories.

On small systems, each user's directory is typically one of the many subdirectories of **/home** such as **/home/smith**, **/home/torvalds**, **/home/operator**, etc. On large systems (especially when the **/home** directories are shared amongst many hosts using NFS) it is useful to subdivide user home directories. Subdivision may be accomplished by using subdirectories such as **/home/staff**, **/home/guests**, **/home/students**, etc.

3.6 **/lib** : Essential shared libraries and kernel modules

The **/lib** directory contains those shared library images needed to boot the system and run the commands in the root filesystem. This includes **/lib/libc.so.***, **/lib/libm.so.***, the shared dynamic linker **/lib/ld.so**, and other shared libraries required by binaries in **/bin** and **/sbin**.

Shared libraries that are only necessary for binaries in **/usr** (such as any X Window binaries) do not belong in **/lib**. Only the shared libraries required to run binaries in **/bin** and **/sbin** should be here.

The library **libm.so.*** may also be placed in **/usr/lib** if it is not required by anything in **/bin** or **/sbin**.

For compatibility reasons, **/lib/cpp** needs to exist as a reference to the C preprocessor installed on the system. The usual placement of this binary is **/usr/lib/gcc-lib/<target>/<version>/cpp**. **/lib/cpp** can either point at this binary, or at any other reference to this binary which exists in the filesystem.

3.7 **/mnt** : Mount point for temporarily mounted filesystems

This directory is provided so that the system administrator may temporarily mount filesystems as needed. The content of this directory is a local issue and should not affect the manner in which any program is run.

We recommend against the use of this directory by installation programs, and suggest that a suitable temporary directory not in use by the system should be used instead.

3.8 **/opt** : Add-on application software packages

/opt is reserved for the installation of add-on application software packages. A package to be installed in **/opt** shall locate its static files in a separate **/opt/<package>** directory tree, where **<package>** is a name that describes the software package. Programs to be invoked by users shall be located in the directory **/opt/<package>/bin**. If the package includes UNIX manual pages, they shall be located in **/opt/<package>/man** and the same substructure as **/usr/share/man** shall be used.

The directories **/opt/bin**, **/opt/doc**, **/opt/include**, **/opt/info**, **/opt/lib**, and **/opt/man** are reserved for local system administrator use. Packages may provide "front-end" files intended to be placed in (by linking or copying) these reserved directories by the local system administrator, but shall function normally in the absence of these reserved directories.

Package files that are variable (change in normal operation) should be installed in **/var/opt**.

Host-specific configuration files should be installed in **/etc/opt**.

No other package files should exist outside the **/opt**, **/var/opt**, and **/etc/opt** hierarchies except for those package files that must reside in specific locations within the filesystem tree in order to function properly. For example, device lock files must be placed in **/var/lock** and devices must be located in **/dev**.

3.9 **/root** : Home directory for the root user (optional)

/ is traditionally the home directory of the root account on UNIX systems. **/root** is used on many Linux systems and on some UNIX systems (in order to reduce clutter in the **/** directory). The root account's home directory may be determined by developer or local preference. Obvious possibilities include **/**, **/root**, and **/home/root.**

If the home directory of the root account is not stored on the root partition it will be necessary to make certain it will default to **/** if it can not be located.

Note: we recommend against using the root account for mundane things such as mail and news, and that it be used solely for system administration. For this reason, we recommend that subdirectories such as **Mail** and **News** not appear in the root account's home directory, and that mail for administration roles such as root, postmaster and webmaster be forwarded to an appropriate user.

3.10 **/sbin** : System binaries (binaries once kept in **/etc**)

Utilities used for system administration (and other root-only commands) are stored in **/sbin**, **/usr/sbin**, and **/usr/local/sbin**. **/sbin** typically contains binaries essential for booting the system in addition to the binaries in **/bin**. Anything executed after **/usr** is known to be mounted (when there are no problems) should be placed into **/usr/sbin**. Local-only system administration binaries should be placed into **/usr/local/sbin**.

Deciding what things go into "**sbin**" directories is simple: If a normal (not a system administrator) user will ever run it directly, then it should be placed in one of the "**bin**" directories. Ordinary users should not have to place any of the **sbin** directories in their path.

We recommend that users have read and execute permission for everything in **/sbin** except, perhaps, certain setuid and setgid programs. The division between **/bin** and **/sbin** was not created for security reasons or to prevent users from seeing the operating system, but to provide a good partition

between binaries that everyone uses and ones that are primarily used for administration tasks. There is no inherent security advantage in making **/sbin** off-limits for users.

Required files for **/sbin**:

o General commands

o Shutdown commands

o Filesystem management commands

o Networking commands

3.11 **/tmp** : Temporary files

The **/tmp** directory shall be made available for programs that require temporary files.

Although data stored in **/tmp** may be deleted in a site-specific manner, it is recommended that files and directories located in **/tmp** be deleted whenever the system is booted. Programs shall not assume that any files or directories in **/tmp** are preserved between invocations of the program.

4. The **/usr** Hierarchy

/usr is the second major section of the filesystem. **/usr** is shareable, read-only data. That means that **/usr** should be shareable between various hosts running FHS-compliant and should not be written to. Any information that is host-specific or varies with time is stored elsewhere.

No large software packages should use a direct subdirectory under the **/usr** hierarchy. An exception is made for the X Window System because of considerable precedent and widely-accepted practice.

4.1 **/usr/X11R6** : X Window System, Version 11 Release 6

This hierarchy is reserved for the X Window System, version 11 release 6, and related files.

4.2 **/usr/X386** : X Window System, Version 11 Release 5, on x86 platforms

This hierarchy is generally identical to **/usr/X11R6**; the /usr symbolic links for X11 should point to the desired version of the X Window System.

4.3 **/usr/bin** : Most user commands

This is the primary directory of executable commands on the system.

Because shell script interpreters (invoked with **#!<path>** on the first line of a shell script) cannot rely on a path, it is advantageous to standardize their locations. The Bourne shell and C-shell interpreters are already fixed in **/bin**, but Perl, Python, and Tcl are often found in many different places. **/usr/bin/perl**, **/usr/bin/python**, and **/usr/bin/tcl** should reference the perl, python, and tcl shell interpreters, respectively. They may be symlinks to the physical location of the shell interpreters.

4.4 **/usr/include** : Directory for standard include files.

This is where all of the system's general-use include files for the C and C++ programming languages should be placed.

4.5 **/usr/lib** : Libraries for programming and packages

/usr/lib includes object files, libraries, and internal binaries that are not intended to be executed directly by users or shell scripts. Applications may use a single subdirectory under **/usr/lib**. If an application uses a subdirectory, all architecture-dependent data exclusively used by the application should be placed within that subdirectory. For example, the perl5 subdirectory for Perl 5 modules and libraries.

Miscellaneous architecture-independent application-specific static files and subdirectories should be placed in **/usr/share**.

4.6 **/usr/local** : Local hierarchy

The **/usr/local** hierarchy is for use by the system administrator when installing software locally. It needs to be safe from being overwritten when the system software is updated. It may be used for programs and data that are shareable amongst a group of hosts, but not found in **/usr**.

This directory should always be empty after first installing a FHS-compliant system. No exceptions to this rule should be made other than the listed directory stubs. Locally installed software should be placed within **/usr/local** rather than **/usr** unless it is being installed to replace or upgrade software in **/usr**.

Note that software placed in **/** or **/usr** may be overwritten by system upgrades (though we recommend that distributions do not overwrite data in **/etc** under these circumstances). For this reason, local software should not be placed outside of **/usr/local** without good reason.

4.7 **/usr/sbin** : Non-essential standard system binaries

This directory contains any non-essential binaries used exclusively by the system administrator. System administration programs that are required for system repair, system recovery, mounting **/usr**, or other essential functions should be placed in **/sbin** instead.

Typically, **/usr/sbin** contains networking daemons, any non-essential administration tools, and binaries for non-critical server programs. Locally installed system administration programs should be placed in **/usr/local/sbin**.

4.8 **/usr/share** : Architecture-independent data

The **/usr/share** hierarchy is for all read-only architecture independent data files. Much of this data originally lived in **/usr** (**man**, **doc**) or **/usr/lib** (**dict**, **terminfo**, **zoneinfo**). This hierarchy is intended to be shareable among all architecture platforms of a given OS; thus, for example, a site with i386, Alpha, and PPC platforms might maintain a single **/usr/share** directory that is centrally-mounted.

Any program or package which contains or requires data that doesn't need to be modified should store that data in **/usr/share** (or **/usr/local/share**, if installed locally). It is recommended that a subdirectory be used in **/usr/share** for this purpose.

4.8.1 **/usr/share/dict** : Word lists

Recommended files for **/usr/share/dict**: { **words** }

Traditionally this directory contains only the English words file, which is used by **look**(1) and various spelling programs. words may use either American or British spelling. Sites that require both may link words to **/usr/share/dict/american-english** or **/usr/share/dict/british-english**.

Word lists for other languages may be added using the English name for that language, e.g., **/usr/share/dict/french**, **/usr/share/dict/danish**, etc. These should, if possible, use an ISO 8859 character set which is appropriate for the language in question; if possible the Latin1 (ISO-8859-1) character set should be used (this is often not possible).

4.8.2 **/usr/share/man** : Manual pages

Manual pages are stored in **<mandir>/<locale>/man<section>/<arch>**. An explanation of **<mandir>**, **<locale>**, **<section>**, and **<arch>** is given below.

The primary **<mandir>** of the system is **/usr/share/man**. **/usr/share/man** contains manual information for commands and data under the **/** and **/usr** filesystems. Obviously, there are no manual pages in **/** because they are not required at boot time nor are they required in emergencies.

The **<section>** describes the manual section.

Provisions must be made in the structure of **/usr/share/man** to support manual pages which are written in different (or multiple) languages. These provisions must take into account the storage and reference of these manual pages. Relevant factors include language (including geographical-based differences), and character code set.

This naming of language subdirectories of **/usr/share/man** is based on Appendix E of the POSIX 1003.1 standard which describes the locale identification string -- the most well-accepted method to describe a cultural environment. The **<locale>** string is:

<language>[_<territory>][.<character-set>][,<version>]

The **<language>** field shall be taken from ISO 639 (a code for the representation of names of languages). It shall be two characters wide and specified with lowercase letters only.

The **<territory>** field shall be the two-letter code of ISO 3166 (a specification of representations of countries), if possible. (Most people are familiar with the two-letter codes used for the country codes in email addresses.1) It shall be two characters wide and specified with uppercase letters only.

The **<character-set>** field should represent the standard describing the character set. If the **<character-set>** field is just a numeric specification, the number represents the number of the international standard describing the character set. It is recommended that this be a numeric representation if possible (ISO standards, especially), not include additional punctuation symbols, and that any letters be in lowercase.

A parameter specifying a **<version>** of the profile may be placed after the **<character-set>** field, delimited by a comma. This may be used to discriminate between different cultural needs; for instance, dictionary order versus a more systems-oriented collating order. This standard recommends not using the **<version>** field, unless it is necessary.

Systems which use a unique language and code set for all manual pages may omit the **<locale>** substring and store all manual pages in **<mandir>**. For example, systems which only have English manual pages coded with ASCII, may store manual pages (the **man<section>** directories) directly in **/usr/share/man**. (That is the traditional circumstance and arrangement, in fact.)

Countries for which there is a well-accepted standard character code set may omit the **<character-set>** field, but it is strongly recommended that it be included, especially for countries with several competing standards.

Similarly, provision must be made for manual pages which are architecture-dependent, such as device-driver documentation or low-level system administration commands. These should be placed under an **<arch>** directory in the appropriate **man<section>** directory; for example, a man page for the i386 ctrlaltdel(8) command might be placed in **/usr/share/man/<locale>/man8/i386/ctrlaltdel.8**.

Manual pages for commands and data under **/usr/local** are stored in **/usr/local/man**. Manual pages for X11R6 are stored in **/usr/X11R6/man**. It follows that all manual page hierarchies in the system should have the same structure as **/usr/share/man**. Empty directories may be omitted from a manual page hierarchy. For example, if **/usr/local/man** has no manual pages in section 4 (Devices), then **/usr/local/man/man4** may be omitted.

The cat page sections (**cat<section>**) containing formatted manual page entries are also found within subdirectories of **<mandir>/<locale>**, but are not required nor should they be distributed in lieu of nroff source manual pages.

The numbered sections "**1**" through "**8**" are traditionally defined. In general, the file name for manual pages located within a particular section end with **.<section>**.

The practice of placing various language manual pages in appropriate subdirectories of **/usr/share/man** also applies to the other manual page hierarchies, such as **/usr/local/man** and **/usr/X11R6/man**. (This portion of the standard also applies later in the section on the optional **/var/cache/man** structure.)

A description of each section follows:

o **man1**: User programs. Manual pages that describe publicly accessible commands are contained in this chapter. Most program documentation that a user will need to use is located here.

o **man2**: System calls. This section describes all of the system calls (requests for the kernel to perform operations).

o **man3**: Library functions and subroutines. Section 3 describes program library routines that are not direct calls to kernel services. This and chapter 2 are only really of interest to programmers.

o **man4**: Special files. Section 4 describes the special files, related driver functions, and networking support available in the system. Typically, this includes the device files found in /dev and the kernel interface to networking protocol support.

o **man5**: File formats. The formats for many nonintuitive data files are documented in the section 5. This includes various include files, program output files, and system files.

o **man6**: Games. This chapter documents games, demos, and generally trivial programs. Different people have various notions about how essential this is.

o **man7**: Miscellaneous. Manual pages that are difficult to classify are designated as being section 7. The **troff** and other text processing macro packages are found here.

o **man8**: System administration. Documentation for programs used by system administrators for system operation and maintenance are documented here. Some of these programs are also occasionally useful for normal users.

4.8.3 **/usr/share/misc** : Miscellaneous architecture-independent data

This directory contains miscellaneous architecture-independent files which don't require a separate subdirectory under **/usr/share**. It is a required directory under **/usr/share**.

The following files, if present, should be located under **/usr/share/misc**:
{ **ascii**, **magic**, **termcap**, **termcap.db** }

Other (application-specific) files may appear here, but a distributor may place them in **/usr/lib** at their discretion.

4.9 **/usr/src** : Source code

Any non-local source code should be placed in this subdirectory.

5. The **/var** Hierarchy

/var contains variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files. Some portions of **/var** are not shareable between different systems. For instance, **/var/log**, **/var/lock**, and **/var/run**. Other portions may be shared, notably **/var/mail**, **/var/cache/man**, **/var/cache/fonts**, and **/var/spool/news**.

/var is specified here in order to make it possible to mount **/usr** readonly. Everything that once went into **/usr** that is written to during system operation (as opposed to installation and software maintenance) must be in **/var**.

Applications should generally not add directories to the top level of **/var**. Such directories should only be added if they have some systemwide implication, and in consultation with the FHS mailing list.

The **cache**, **lock**, **log**, **run**, **spool**, **state**, and **tmp** directories must be included and used in all distributions; the **account**, **crash**, **games**, **mail**, and **yp** directories must be included and used if the corresponding applications or features are provided in the distribution.

5.1 **/var/account** : Process accounting logs (if supported)

This directory holds the current active process accounting log and the composite process usage data (as used in some UNIX-like systems by lastcomm and sa).

5.2 **/var/cache** : Application cache data

/var/cache is intended for cached data from applications. Such data is locally generated as a result of time-consuming I/O or calculation. The application must be able to regenerate or restore the data. Unlike **/var/spool**, the cached files can be deleted without data loss. The data should remain valid between invocations of the application and rebooting the system.

Files located under **/var/cache** may be expired in an application specific manner, by the system administrator, or both. The application should always be able to recover from manual deletion of files (generally because of a disk space shortage). No other requirements are made on the data format of the cache directories.

5.2.1 **/var/cache/fonts** : Locally-generated fonts

The directory **/var/cache/fonts** should be used to store any dynamically-created fonts. In particular, **/var/cache/fonts/pk** will store all of the fonts which are automatically generated by MakeTeXPK. Other dynamically created fonts may also be placed in this tree, under appropriately-named subdirectories of **/var/cache/fonts**.

5.2.2 **/var/cache/man** : Locally-formatted manual pages (optional)

This directory provides a standard location for sites that provide a read-only **/usr** partition, but wish to allow caching of locally-formatted man pages. We recommend that most sites use one of the following options instead:

o Preformat all manual pages alongside the unformatted versions.

o Allow no caching of formatted man pages, and require formatting to be done each time a man page is brought up.

o Allow local caching of formatted man pages in **/var/cache/man**.

The structure of **/var/cache/man** needs to reflect both the fact of multiple man page hierarchies and the possibility of multiple language support.

Given an unformatted manual page that normally appears in **<path>/man/<locale>/man<section>**, the directory to place formatted man-pages in is **/var/cache/man/<catpath>/<locale>/cat<section>**, where **<catpath>** is derived from **<path>** by removing any leading **usr** and/or trailing share pathname components. (Note that the **<locale>** component may be missing.)

Man pages written to **/var/cache/man** may eventually be transferred to the appropriate preformatted directories in the source man hierarchy or expired; likewise formatted man pages in the source man hierarchy may be expired if they are not accessed for a period of time.

5.3 **/var/crash** : System crash dumps (if supported)

This directory holds system crash dumps. As of the date of this release of the standard, system crash dumps were not supported under Linux.

5.4 **/var/games** : Variable game data

Any variable data relating to games in **/usr** should be placed here. **/var/games** should hold the variable data previously found in **/usr**; static data, such as help text, level descriptions, and so on, should remain elsewhere, such as **/usr/share/games**.

As for **/var/state**, game variable data may be placed in /var/lib as a deprecated transitional measure. However, this allowance will be removed in a future release of the standard.

5.5 **/var/lock** : Lock files

Lock files should be stored within the **/var/lock** directory structure.

Device lock files, such as the serial device lock files that were originally found in either **/usr/spool/locks** or **/usr/spool/uucp**, must now be stored in **/var/lock**. The naming convention which must be used is "**LCK..**" followed by the base name of the device. For example, to lock **/dev/cua0** the file "**LCK..cua0**" would be created.

5.6 **/var/log** : Log files and directories

The directory contains miscellaneous log files.

Most logs should be written to this directory or an appropriate subdirectory.

lastlog record of last login of each user
messages system messages from syslogd
wtmp record of all logins and logouts

5.7 **/var/mail** : User mailbox files

This directory contains user mailbox files stored in the standard UNIX mailbox format.

5.8 **/var/opt** : Variable data for **/opt**

Variable data should be installed in **/var/opt/<package>**, where **<package>** is the name of the subtree in **/opt** where the static data from an add-on software package is stored, except where superseded by another file in **/etc**. No structure is imposed on the internal arrangement of **/var/opt/<package>**.

5.9 **/var/run** : Run-time variable files

This directory contains system information files describing the system since it was booted. Files in this directory should be cleared (removed or truncated as appropriate) at the beginning of the boot process.

Process identifier (PID) files, which were originally placed in **/etc**, should be placed in **/var/run**. The naming convention for PID files is **<program-name>.pid**.

The **utmp** file, which stores information about who is currently using the system, is located in this directory. Programs that maintain transient UNIX-domain sockets should place them in this directory.

5.10 **/var/spool** : Application spool data

/var/spool contains data which is awaiting some kind of later processing. Data in **/var/spool** represents work to be done in the future (by a program, user, or administrator); often data is deleted after it has been processed. UUCP lock files must be placed in **/var/lock**.

5.10.1 **/var/spool/lpd** : Line-printer daemon print queues

The lock file for **lpd**, **lpd.lock**, should be placed in **/var/spool/lpd**. It is suggested that the lock file for each printer be placed in the spool directory for that specific printer and named lock.

5.10.2 **/var/spool/rwho** : Rwhod files

This directory holds the rwhod information for other systems on the local net.

5.11 **/var/state** : Variable state information

This hierarchy holds state information pertaining to an application or the system. State information is data that programs modify while they run, and that pertains to one specific host. Users should never need to modify files in **/var/state** to configure a package's operation.

State information is generally used to preserve the condition of an application (or a group of inter-related applications) between invocations and between different instances of the same application. State information should generally remain valid after a reboot, should not be logging output, and should not be spooled data.

An application (or a group of inter-related applications) should use a subdirectory of **/var/state** for its data. There is one required subdirectory, **/var/state/misc**, which is intended for state files that don't need a subdirectory; the other subdirectories should only be present if the application in question is included in the distribution.

/var/state/<name> is the location that should be used for all distribution packaging support. Different distributions may use different names, of course.

5.11.1 **/var/state/<editor>** : Editor backup files and state

These directories contain saved files generated by any unexpected termination of an editor (e.g., **elvis**, **jove**, **nvi**).

Other editors may not require a directory for crash-recovery files, but may require a well-defined place to store other information while the editor is running. This information should be stored in a subdirectory under **/var/state**. Future editors may require additional state information beyond crash-recovery files and lock files -- this information should also be placed under **/var/state/<editor>**.

5.11.2 **/var/state/misc** : Miscellaneous variable data

This directory contains variable data not placed in a subdirectory in **/var/state**. An attempt should be made to use relatively unique names in this directory to avoid namespace conflicts.

5.12 **/var/tmp** : Temporary files preserved between system reboots

The **/var/tmp** directory is made available for programs that require temporary files or directories that are preserved between system reboots. Therefore, data stored in **/var/tmp** is more persistent than data in **/tmp**. Files and directories located in **/var/tmp** must not be deleted when the system is booted. Although data stored in **/var/tmp** is typically deleted in a site-specific manner, it is recommended that deletions occur at a less frequent interval than **/tmp**.

5.13 **/var/yp** : Network Information Service (NIS) database files

Variable data for the Network Information Service (NIS), formerly known as the Sun Yellow Pages (YP), shall be placed in this directory.

6. Operating System Specific Annex

6.1.1 **/** : Root directory

On Linux systems, if the kernel is located in **/**, we recommend using the names **vmlinux** or **vmlinuz**, which have been used in recent Linux kernel source packages.

6.1.2 **/dev** : Devices and special files

All devices and special files in **/dev** should adhere to the Linux Allocated Devices document, which is available with the Linux kernel source.

6.1.3 **/proc** : Kernel and process information virtual filesystem

The proc filesystem is becoming the de-facto standard Linux method for handling process and system information, rather than **/dev/kmem** and other similar methods.

6.1.4 **/sbin** : Essential system binaries

Linux systems place these additional files into /sbin.

o Second extended filesystem commands (optional)

o Boot-loader map installer

Optional files for **/sbin**:

o Static binaries

Static **ln** (**sln**) and static **sync** (**ssync**) are useful when things go wrong. The primary use of **sln** (to repair incorrect symlinks in **/lib** after a poorly orchestrated upgrade) is no longer a major concern now that the **ldconfig** program (usually located in **/usr/sbin**) exists and can

act as a guiding hand in upgrading the dynamic libraries. Static sync is useful in some emergency situations. The **ldconfig** binary is optional for **/sbin** since a site may choose to run **ldconfig** at boot time, rather than only when upgrading the shared libraries. (It's not clear whether or not it is advantageous to run **ldconfig** on each boot.)

o Miscellaneous

So as to cope with the fact that some keyboards come up with such a high repeat rate as to be unusable, kbdrate may be installed in **/sbin** on some systems. Since the default action in the kernel for the Ctrl-Alt-Del key combination is an instant hard reboot, it is generally advisable to disable the behavior before mounting the root filesystem in read-write mode. Some init suites are able to disable Ctrl-Alt-Del, but others may require the **ctrlaltdel** program, which may be installed in **/sbin** on those systems.

6.1.5 /usr/include : Header files included by C programs

These symbolic links are required if a C or C++ compiler is installed.

```
/usr/include/asm -> /usr/src/linux/include/asm-<arch>
/usr/include/linux -> /usr/src/linux/include/linux
```

6.1.6 /usr/src : Source code

The only source code that should be placed in a specific location is the Linux kernel source code. It is located in **/usr/src/linux**.

If a C or C++ compiler is installed, but the complete Linux kernel source code is not installed, then the include files from the kernel source code shall be located in these directories:

```
/usr/src/linux/include/asm-<arch>
/usr/src/linux/include/linux
```

<arch> is the name of the system architecture.

Note: **/usr/src/linux** may be a symbolic link to a kernel source code tree.

6.1.7 /var/spool/cron : cron and at jobs

This directory contains the variable data for the **cron** and **at** programs.

Die Shell „BaSH“

[Montag:Nachmittag]

Wenden wir uns nun der *bash - GNU Bourne-Again SHell* zu. Bevor wir sie „richtig“ benutzen, sollten wir uns einen Überblick verschaffen, wie sie funktioniert. Dies umfaßt den Aspekt ihres Aufrufs (ihrer Aktivierung) ebenso wie jenen ihrer Konfiguration.

Aufruf-Bearbeitung

Üblicherweise wird die *BaSH* implizit aufgerufen, entweder durch den **login**-Prozeß, der sie als interaktive *shell* startet, die solange aktiv bleibt, bis sie ausdrücklich beendet wird, oder durch den Aufruf eines *shell-scripts*, der einen nicht-interaktiven Prozeßstartet, der die einzelnen Anweisungen des *scripts* abarbeitet und dann automatisch beendet wird.

Da es eine ganze Reihe möglicher Dateien gibt bzw. geben kann , in denen Konfigurations-Daten abgelegt sein können, ist es wichtig, die *Reihenfolge* zu kennen, in denen diese Dateien ausgewertet werden. Zudem gibt es Unterschiede zwischen einem „normalen“, d.h. interaktiven Shell-Prozeß und einem nicht-interaktiven, wie er beispielsweise bei der Abarbeitung von *Shell-Scripts* vorkommt.

A *login shell* is one whose first character of argument zero is a **-**, or one started with the **-login** flag.

An *interactive shell* is one whose standard input and output are both connected to terminals (as determined by **isatty**(3)), or one started with the **-i** option. **PS1** is set and **$-** includes **i** if bash is interactive, allowing a shell script or a startup file to test this state.

Login shells:

On login (subject to the **-noprofile** option):
if **/etc/profile** exists, source it.
if **~/.bash_profile** exists, source it,
else if **~/.bash_login** exists, source it,
else if **~/.profile** exists, source it.

On exit:
if **~/.bash_logout** exists, source it.

Non-login interactive shells:

On startup (subject to the **-norc** and **-rcfile** options):
if **~/.bashrc** exists, source it.

Non-interactive shells:

On startup:
if the environment variable **ENV** is non-null, expand it and source the file it names, as if the command »**if ["$ENV"]; then . $ENV; fi**« had been executed, but do not use **PATH** to search for the pathname. When not started in Posix mode, bash looks for **BASH_ENV** before **ENV**.

If Bash is invoked as **sh**, it tries to mimic the behavior of **sh** as closely as possible. For a login shell, it attempts to source only **/etc/profile** and **~/.profile**, in that order. The **-noprofile** option may still be used to disable this behavior. A shell invoked as **sh** does not attempt to source any other startup files.

When bash is started in posix mode, as with the **-posix** command line option, it follows the Posix standard for startup files. In this mode, the **ENV** variable is expanded and that file sourced; no other startup files are read.

Konfiguration

Ausgehend von diesen Informationen kommen nun die einzelnen, erwähnten Dateien ins Blickfeld. Damit die *BASH überhaupt* funktioniert, muß im Grunde keine von ihnen tatsächlich vorhanden sein. Allerdings können sie Arbeitsweise und Komfortabilität der Arbeit mit der *shell* weitgehend beeinflussen,

und zumindest die Datei **/etc/profile** dürfte wohl ohnehin auf jedem Rechner vorhanden sein – und sei's auch nur, weil sie bei der Grundinstallation automatisch angelegt wurde.

System-global

Bei den systemweit verfügbaren Dateien ist grundsätzlich zu beachten, daß diese in der Regel zwar von allen *usern* gelesen, nicht aber verändert werden können. Um sie also den eigenen Bedürfnissen anzupassen, benötigt man Administratoren-Rechte (*root-access*).

Die wohl wichtigste systemweit verfüg- und nutzbare Konfigurations-Datei ist **/etc/profile**, da diese von allen Shells ausgewertet wird. Ihr Inhalt ist letztlich dafür verantwortlich, ob bestimmte Anwendungs-Programme oder Werkzeuge (*tools*) korrekt, d.h. erwartungsgemäß und fehlerfrei funktionieren. Mit anderen Worten: Hier hinein gehören alle Einstellungen, die nicht spezifisch für die verwendete *Shell* sind, sondern in gewissem Sinne spezifisch für eine bestimmte Maschine bzw. die darauf installierten Anwendungen.

```
[matthias@Barbara Bash-Kurs]$ cat /etc/profile
# /etc/profile
#  System wide environment and startup programs
#  Functions and aliases go in /etc/bashrc
#-----------------------------------------------------------------------------
PATH="$PATH:/opt/bin:/usr/local/bin:/usr/X11R6/bin"
PS1="[\u@\h \W]\$ "

# ulimit -c 1000000
ulimit -c 0
umask 002

HOSTNAME=`/bin/hostname`
HISTSIZE=1000
HISTFILESIZE=1000

# Default path for CRiSPlite
CRPATH=/usr/lib/CRiSPlite/macros

export PATH PS1 HOSTNAME HISTSIZE HISTFILESIZE CRPATH

[ "$TERM" = console ] && export MINICOM="-l -m -con -tmc"

# - mw-Erweiterungen:

export EDITOR="jstar"
export LESSCHARSET="latin1"

# fuer 'mpage':
export MPAGE="-PPostScript -4 -A -f -ml -C ISO-Latin.1"

# fuer WWW-Browser:
export WWW_HOME="http://www.dfg.de/index.html"

# fuer 'man' u.a.:
export NLSPATH="/usr/X11R6/lib/X11/nls"

# fuer NW-Utilities:
export NWCLIENT_PREFERRED_SERVER="Christine"
export NWCLIENT_PREFERRED_QUEUE="HP4"

# fuer NNTP-Klienten:
export NNTPSERVER=News.DFG.De
export SMTPSERVER=Mail.DFG.De

# fuer diesen und jenen:
export TZ="MET"


[matthias@Barbara Bash-Kurs]$ _
```

Speziell für die *BaSH* nun kommt eine weitere Datei hinzu, in welcher all jene Einstellungen vorgenommen werden können, die auf der jeweiligen Maschine für alle *BaSH*-Prozesse verfügbar sein sollen. Einen – gewichtigen – Haken hat die Sache indes: Diese Datei **/etc/bashrc** wird nicht automatisch ausgewertet! Vielmehr muß dafür gesorgt werden, daß dies von jedem gestarteten Shell-Prozeß seperat erledigt wird – das unkomplizierte *Wie?* werden wir gleich anschließend sehen.

```
[matthias@Barbara Bash-Kurs]$ cat /etc/bashrc
# /etc/bashrc
#   System wide functions and aliases
#   (Environment stuff goes in /etc/profile)
#-----------------------------------------------------------------------------
# For some unknown reason bash refuses to inherit
# PS1 in some circumstances that I can't figure out.
# Putting PS1 here ensures that it gets loaded every time.
# export PS1="\[\033[32m\][\u@\h]\[\033[0m\] \W \\$ "
export PS1="[\u@\h \W]\\$ "
export PS2="\W > "

# aliases for all shells
alias cp='cp -a -i'             # preserve attributes / interactive
alias mv='mv -i'            # interactive
alias rm='rm -i'            # interactive

alias ll='color-ls -ol'         # ls-Aufrufe zu 'color-ls' umleiten
alias l.='color-ls -o .[a-zA-Z]*'
alias ls='color-ls -Cfo'
alias la='color-ls -ola'

alias psl='ps -aux | less'
alias pst='pstree -apuh'

alias ..='cd ..'
alias .='echo $PWD'

alias which='type -path'
alias e="/usr/bin/jstar"
alias t="/usr/bin/tail -n 50 -f "
alias delbak='rm -f *~ .*~ ; /bin/ls'
alias mc="/usr/local/bin/mc -c -s "
alias rpm="/bin/rpm -vv "   # show diagnostic output
alias smb="/opt/samba/bin/smbstatus -d"

if [ -x /usr/bin/less ] ; then
    alias l='less'
else
    alias l='more'
fi

if [ -n "${DISPLAY}" ] ; then
    [ -x /usr/X11R6/xless ] && alias m='xless'
fi

# hilfreiche Funktionen:
# show filenames if the user has changed directory
function cd() {
    builtin cd $@
    ls
}

# create a new directory and change to
function nd() {
    if [ $# = "1" ] ; then
        mkdir $1; cd $1
    else
        echo "Usage: nd <directory>"
    fi
}


[matthias@Barbara Bash-Kurs]$ _
```

Hier werden also eine Fülle von **`alias`**-Funktionen definiert sowie zwei **`function`**s, die jeweils für alle Shell-Prozesse nutzbar sind, welche diese Datei auswerten. Zu einem späteren Zeit punkt im Seminar werden wir auf die hier vorhandenen Einzelheiten zurückkommmen.

Anwender-spezifisch

Diese Dateien liegen –soweit überhaupt vorhanden –jeweils im **`HOME`**-Verzeichnis, das stets kurz mit »**`~/`**« angegeben werden kann. Um die jeweils eigenen Dateien zu bearbeiten, werden natürlich keine besonderen (Zugriffs-)Rechte benötigt. Der konkrete Inhalt dieser Dateien kann sehr stark variieren, denn jeder *user* wird nach und nach seine eigenen Bedürfnisse, Vorlieben und Abneigungen entdecken, die sich dann auch in seiner Konfiguration niederschlagen.

```
[matthias@Barbara Bash-Kurs]$ cat ~/.bash_profile
# ~/bash_profile
```

```
# Get the aliases and functions
[ -s ~/.bashrc ] && . ~/.bashrc

# User specific environment and startup programs

if [ $SHLVL = 1 ] ; then
    export PATH=$PATH:/sbin:/usr/sbin:$HOME/bin:.
else
    export PATH=$PATH:/sbin:/usr/sbin:$HOME/bin
fi

export ENV=$HOME/.bashrc
export USERNAME="matthias"
export EDITOR=/usr/bin/jstar
export VISUAL=/usr/bin/jstar

# Einstellung fuer's CVS:
export CVSROOT=/opt/cvsroot

# beim NW-Server anmelden:
nwlogin -b -s CHRISTINE -u MATTHIAS
nwwhoami
echo

# angemeldete User/Prozesse anzeigen:
w -f


[matthias@Barbara Bash-Kurs]$ _
```

Diese von der *BaSH* als erstes verarbeitete Konfigurations-Datei sorgt zum einen dafür, daßandere Konfig-Dateien («**. ~/.bashrc**«, dadurch wiederum **/etc/bashrc**) ausgewertet werden, desweiteren richtet sie einige Umgebungs-Variablen ein und sorgt in diesem Falle schließlich für eine automatische Anmeldung bei einem Netware-Server. –Wie gesagt: Der Inhalt einer sol chen Datei ist –sieht man von ihrer formalen Korrektheit einmal ab –im wahren Wortsinne „will kürlich". Was im übrigen auch für die nächste Datei gilt:

```
[matthias@Barbara Bash-Kurs]$ cat ~/.bashrc
# ~/.bashrc
#
# Source global definitions
[ -s /etc/bashrc ] && . /etc/bashrc

# remove ~/.mcwd (mtools)
if [ -f ~/.mcwd ] ; then
    rm -f ~/.mcwd
fi

# extract compressed tar-file
function xzip() {
    if [ $# -eq 1 ] ; then
        tar xvzf $1
    else
        echo "Usage: xzip <filename>"
    fi
} # xzip

# list contents of compressed tar-file
function lszip() {
    if [ $# -eq 1 ] ; then
        tar tzf $1
    else
        echo "Usage: lszip <filename>"
    fi
} # lszip

# search for file (default: beginning from current directory)
function ff() {
    case $# in
        1) find . -fstype ext2 . -name "$1" -print ;;
        2) find "$1" -fstype ext2 -name "$2" -print ;;
        *) echo "Usage: ff [<search_path>] <filename>" ;;
    esac
} # ff

# eine MAN-page anzeigen
function vm() {
    if [ $# -eq 1 ]; then
        nroff -Tascii -mandoc $1 | l
    else
        echo "Usage: vm <filename>"
    fi
```

```
} # vm

# HTML-Seiten in Text konvertieren
function h2t() {
    for F in $* ; do
        lynx -force_html -dump ${F} >${F%.*}.txt
    done
} # h2t


[matthias@Barbara Bash-Kurs]$ _
```

Hier werden nach Belieben und Gutdünken der jeweiligen Anwenderin (*user*) weitere **alias**- oder **function**-Deklarationen zusammengestellt. Zu erkennen ist auch, wie einfach mittels »**. /etc/bashrc**« die zuvor dargestellte, systemweit verfügbare Konfigurations-Datei einge lesen wird.

Beim Beenden einer *Login-shell* wird schließlich noch die folgende Datei abgearbeitet.

```
[matthias@Barbara Bash-Kurs]$ cat ~/.bash_logout
# ~/.bash_logout
#
# beim NW-Server abmelden:
nwlogout -a
# Bildschirm loeschen:
clear


[matthias@Barbara Bash-Kurs]$ _
```

Umgebungs-Variablen

Bereits bei den bislang betrachteten Konfigurations-Dateien war zu erkennen, daßneben Be sonderheiten wie **alias**- und **function**-Deklarationen vor allem das Setzen von *Variablen* benutzt wurde, um bestimmte Effekte zu erzielen. So wird beispielsweise die Variable **EDITOR** nicht nur von der *shell*, sondern auch von einer Reihe anderer Programme verwendet, um bei Bedarf einen Editor zu aktivieren. Neben den selbst angelegten und nach Belieben verwendbaren Variablen, richtet auch die *shell* einige ein bzw. füllt sie mit Werten:

PPID
The process ID of the shell's parent.

PWD
The current working directory as set by the cd command. **OLDPWD** The previous working directory as set by the **cd** command.

REPLY
Set to the line of input read by the **read** builtin command when no arguments are supplied.

UID
Expands to the user ID of the current user, initialized at shell startup.

EUID
Expands to the effective user ID of the current user, initialized at shell startup.

BASH
Expands to the full pathname used to invoke this instance of bash.

BASH_VERSION
Expands to the version number of this instance of bash.

SHLVL
Incremented by one each time an instance of bash is started.

OPTARG
The value of the last option argument processed by the **getopts** builtin command.

OPTIND
The index of the next argument to be processed by the **getopts** builtin command.

The following variables are used by the shell. In some cases, bash assigns a default value to a variable; these cases are noted below.

IFS
The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the read builtin command. The default value is »**<space><tab><newline>**«.

PATH
The search path for commands. It is a colonseparated list of directories in which the shell looks for commands. The default path is system-dependent, and is set by the administrator who installs bash. A common value is »**/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:.**«

HOME
The home directory of the current user; the default argument for the **cd** builtin command.

CDPATH
The search path for the **cd** command. This is a colon-separated list of directories in which the shell looks for destination directories specified by the **cd** command. A sample value is »**.:~:/usr**«.

ENV
If this parameter is set when bash is executing a shell script, its value is interpreted as a filename containing commands to initialize the shell, as in **.bashrc**. The value of **ENV** is subjected to parameter expansion, command substitution, and arithmetic expansion before being interpreted as a pathname. **PATH** is not used to search for the resultant pathname.

MAIL
If this parameter is set to a filename and the **MAILPATH** variable is not set, bash informs the user of the arrival of mail in the specified file.

MAILCHECK
Specifies how often (in seconds) bash checks for mail. The default is 60 seconds. When it is time to check for mail, the shell does so before prompting. If this variable is unset, the shell disables mail checking.

MAILPATH
A colon-separated list of pathnames to be checked for mail. The message to be printed may be specified by separating the pathname from the message with a »**?**«. **$_** stands for the name of the current mailfile. Example:
MAILPATH='/usr/spool/mail/bfx?"You have mail":~/shell-mail?"$_ has mail!"'
Bash supplies a default value for this variable, but the location of the user mail files that it uses is system dependent (e.g., **/usr/spool/mail/$USER**).

MAIL_WARNING
If set, and a file that bash is checking for mail has been accessed since the last time it was checked, the message »**The mail in mailfile has been read**« is printed.

PS1
The value of this parameter is expanded and used as the primary prompt string. The default value is »**bash\$** «.

PS2
The value of this parameter is expanded and used as the secondary prompt string. The default is »**>** «.

PS3
The value of this parameter is used as the prompt for the **select** command.

PS4
The value of this parameter is expanded and the value is printed before each command bash displays during an execution trace. The first character of **PS4** is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is »**+**«.

HISTSIZE
The number of commands to remember in the command history. The default value is **500**.

HISTFILE
The name of the file in which command history is saved. The default value is **~/.bash_history**. If unset, the command history is not saved when an interactive shell exits.

HISTFILESIZE
The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than that number of lines. The default value is **500**.

OPTERR
If set to the value 1, bash displays error messages generated by the **getopts** builtin command. **OPTERR** is initialized to 1 each time the shell is invoked or a shell script is executed.

PROMPT_COMMAND
If set, the value is executed as a command prior to issuing each primary prompt.

TMOUT
If set to a value greater than zero, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt. Bash terminates after waiting for that number of seconds if input does not arrive.

FIGNORE
A colon-separated list of suffixes to ignore when performing filename completion. A filename whose suffix matches one of the entries in **FIGNORE** is excluded from the list of matched filenames. A sample value is »**.o:~**«.

INPUTRC
The filename for the readline startup file, overriding the default of **~/.inputrc**.

notify
If set, bash reports terminated background jobs immediately, rather than waiting until before printing the next primary prompt.

history_control
HISTCONTROL
If set to a value of **ignorespace**, lines which begin with a **space** character are not entered on the history list. If set to a value of **ignoredups**, lines matching the last history line are not entered. A value of **ignoreboth** combines the two options. If unset, or if set to any other value than those above, all lines read by the parser are saved on the history list.

command_oriented_history
If set, bash attempts to save all lines of a multiple-line command in the same history entry. This allows easy re-editing of multi-line commands.

glob_dot_filenames
If set, bash includes filenames beginning with a »**.**« in the results of pathname expansion.

allow_null_glob_expansion
If set, bash allows pathname patterns which match no files to expand to a null string, rather than themselves.

histchars
The two or three characters which control history expansion and tokenization. The first character is the history expansion character, that is, the character which signals the start of a history expansion, normally »**!**«. The second character is the quick substitution character, which is used as shorthand for rerunning the previous command entered, substituting one string for another in the command. The default is »**^**«. The optional third character is the character which signifies that the remainder of the line is a comment, when found as the first character of a word, normally »**#**«. The history comment character causes history substitution to be skipped for the remaining words on the line. It does not necessarily cause the shell parser to treat the rest of the line as a comment.

nolinks
If set, the shell does not follow symbolic links when executing commands that change the current working directory. It uses the physical directory structure instead. By default, bash follows the logical chain of directories when performing commands which change the current directory, such as **cd**. See also the description of the **-P** option to the **set** builtin.

hostname_completion_file
HOSTFILE
Contains the name of a file in the same format as **/etc/hosts** that should be read when the shell

needs to complete a hostname. The file may be changed interactively; the next time hostname completion is attempted bash adds the contents of the new file to the already existing database.

noclobber

If set, bash does not overwrite an existing file with the **>**, **>&**, and **<>** redirection operators. This variable may be overridden when creating output files by using the redirection operator **>|** instead of **>** (see also the **-C** option to the **set** builtin command).

Im Prinzip kann man die meisten dieser Variablen als R/O (*readonly*) betrachten oder benötigt sie im Normalfalle überhaupt nicht. In besonderen Situationen aber kann es recht hilfreich sein, sie – zumindest zeitweise – zu manipulieren, um bestimmte Effekte zu erreichen. Ein Beispiel hierfür ist **IFS**: ersetzt oder erweitert man ihren Wert so, daß er auch – sagen wir: – den Doppelpunkt »**:**« enthält, so können die Zeilen einer eingelesenen Datei oder die Aufrufparameter einer **function** oder **subshell** die einzelnen Bestandteile einer solchen Zeile eben auch anhand des **:** unterscheiden.

Andere Beispiele für möglicherweise veränderte Umgebungs-Variablen sind die **PSx**-Werte (**PS1**, **PS2**, **PS3**, **PS4**). Das „normale" Befehls-Prompt, also jene Zeichenfolge mit der die *shell* am Anfang ihrer Befehlszeile ihre Arbeits-Bereitschaft signalisiert, wird durch **PS1** konfiguriert. Anstelle eines simplen »**$**« ist es – zumal bei der Arbeit in Netzwerken – sinnvoll, hier auch den gerade verwendeten *User-Namen* (**\u**) sowie den *Namen des Hosts* (**\h**), auf dem gerade gearbeitet wird, aufzunehmen, wie dies in der oben dargestellten Datei **/etc/bashrc** gezeigt ist:

```
export PS1="[\u@\h \W]\\$ "
```

Zusätzlich ist hier auch noch das gerade aktuelle Verzeichnis (**\W**) mit aufgenommen, so daß auch bei der Arbeit mit mehreren Konsolen gleich ersichtlich wird, wo man sich jeweils befindet.

(!) Die Kommando-Oberfläche (*shell*) ist weitgehend konfigurierbar. Die Teilnehmerinnen lernen kennen, was davon vom Administrator für das gesamte Rechner-System vorgegeben werden kann und welche Möglichkeiten die Anwenderinnen (*user*) haben, sich „ihre persönliche" Shell zu konfigurieren.

(?) Bei entsprechendem Interesse und gegebenen (Rechner-) Voraussetzungen kann eine *BaSH*-Portierung für Win32-Systeme (*CygWin*) installiert und konfiguriert werden. Diese könnte dann während des weiteren Seminar-Verlaufes parallel zur Arbeit mit der „echten" Shell benutzt werden, um gemeinsame Möglichkeiten, aber ggf. auch Unterschiede aufgrund des jeweils zu grundeliegenden Betriebssystemes zu erfahren.

Die „BaSH“ als Benutzer-Oberfläche

[Dienstag: Vormittag]

Um mehr zu sein als ein simpler „Programm-Starter“ für andere, „externe“ Werkzeuge (*tools*) muß eine Benutzer-Oberfläche auch über eine gewisse eigene „Intelligenz“ verfügen, die ihren Ausdruck zum einen findet in Art und Umfang der von ihr originär – also nicht über den Umweg separater Programme – zur Verfügung gestellten Befehle, den hier sogenannten *builtins*, und zum anderen in den Optionen, die zum Füllen und Manipulieren der Kommandozeile selbst angeboten werden. Dabei zerfällt das Letztgenannte wiederum in mehrere Bereiche: Das Erstellen und Manipulieren der Kommondozeile selbst (*readline*), die Interpretation dort eventuell auftretender Sonderzeichen (verschiedene Formen der *expansion*) sowie schließlich die Verarbeitung der so „auf bereiteten“ Befehle (*evaluation* und *execution*).

eingebaute Befehle

`: [arguments]`
No effect; the command does nothing beyond expanding **`arguments`** and performing any specified redirections. A zero exit code is returned.

`. filename [arguments]`
`source filename [arguments]`
Read and execute commands from **`filename`** in the current shell environment and return the exit status of the last command executed from **`filename`**. If **`filename`** does not contain a slash, pathnames in **`PATH`** are used to find the directory containing **`filename`**. The file searched for in **`PATH`** need not be executable. The current directory is searched if no file is found in **`PATH`**. If any **`arguments`** are supplied, they become the positional parameters when file is executed. Otherwise the positional parameters are unchanged. The return status is the status of the last command exited within the script (0 if no commands are executed), and false if **`filename`** is not found.

`alias [name[=value] ...]`
`Alias` with no arguments prints the list of aliases in the form **`name=value`** on standard output. When arguments are supplied, an **`alias`** is defined for each **`name`** whose **`value`** is given. A trailing space in **`value`** causes the next word to be checked for **`alias`** substitution when the **`alias`** is expanded. For each **`name`** in the argument list for which no **`value`** is supplied, the **`name`** and **`value`** of the **`alias`** is printed. **`Alias`** returns true unless a **`name`** is given for which no **`alias`** has been defined.

`bg [jobspec]`
Place **`jobspec`** in the background, as if it had been started with &. If **`jobspec`** is not present, the shell's notion of the current job is used. **`bg jobspec`** returns 0 unless run when job control is disabled or, when run with job control enabled, if **`jobspec`** was not found or started without job control.

`break [n]`
Exit from within a **`for`**, **`while`**, or **`until`** loop. If **`n`** is specified, break **`n`** levels. **`n`** must be >=1. If **`n`** is greater than the number of enclosing loops, all enclosing loops are exited. The return value is 0 unless the shell is not executing a loop when **`break`** is executed.

`builtin shell-builtin [arguments]`
Execute the specified shell **`builtin`**, passing it **`arguments`**, and return its exit status. This is useful when you wish to define a function whose name is the same as a shell **`builtin`**, but need the functionality of the **`builtin`** within the function itself. The **`cd builtin`** is commonly redefined this way. The return status is false if **`shell-builtin`** is not a shell **`builtin`** command.

`cd [dir]`
Change the current directory to **`dir`**. The variable **`HOME`** is the default dir. The variable **`CDPATH`** defines the search path for the directory containing **`dir`**. Alternative directory names are separated by a colon (:). A null directory name in **`CDPATH`** is the same as the current directory, i.e., »**`.`**«. If **`dir`** begins

with a slash (/), then **CDPATH** is not used. An argument of **-** is equivalent to **$OLDPWD**. The return value is true if the directory was successfully changed; false otherwise.

command [-pVv] command [arg...]

Run **command** with **arg**s suppressing the normal shell function lookup. Only builtin commands or commands found in the **PATH** are executed. If the **-p** option is given, the search for **command** is performed using a default value for **PATH** that is guaranteed to find all of the standard utilities. If either the **-V** or **-v** option is supplied, a description of command is printed. The **-v** option causes a single word indicating the command or pathname used to invoke **command** to be printed; the **-V** option produces a more verbose description. An argument of **--** disables option checking for the rest of the arguments. If the **-V** or **-v** option is supplied, the exit status is 0 if **command** was found, and 1 if not. If neither option is supplied and an error occurred or **command** cannot be found, the exit status is 127. Otherwise, the exit status of the **command** builtin is the exit status of **command**.

continue [n]

Resume the next iteration of the enclosing for, **while**, or **until** loop. If **n** is specified, resume at the nth enclosing loop. **n** must be >= 1. If **n** is greater than the number of enclosing loops, the last enclosing loop (the »top-level« loop) is resumed. The return value is 0 unless the shell is not executing a loop when **continue** is executed.

declare [-frxi] [name[=value]]
typeset [-frxi] [name[=value]]

Declare variables and/or give them attributes. If no names are given, then display the values of variables instead. The options can be used to restrict output to variables with the specified attribute.

-f
Use function names only

-r
Make names readonly. These names cannot then be assigned values by subsequent assignment statements.

-x
Mark names for export to subsequent commands via the environment.

-i
The variable is treated as an integer; arithmetic evaluation is performed when the variable is assigned a value.

Using »**+**« instead of »**-**« turns off the attribute instead. When used in a function, makes names local, as with the **local** command. The return value is 0 unless an illegal option is encountered, an attempt is made to define a function using »**-f foo=bar**«, one of the names is not a legal shell variable name, an attempt is made to turn off readonly status for a readonly variable, or an attempt is made to display a non-existant function with **-f**.

echo [-neE] [arg...]

Output the args, separated by spaces. The return status is always 0. If **-n** is specified, the trailing newline is suppressed. If the **-e** option is given, interpretation of the following backslashescaped characters is enabled. The **-E** option disables the interpretation of these escape characters, even on systems where they are interpreted by default.

\a
alert (bell)

\b
backspace

\c
suppress trailing newline

\f
form feed

\n
new line

\r
carriage return

\t
horizontal tab

\v
vertical tab

****
backslash

\nnn
the character whose ASCII code is **nnn** (octal)

enable [-n] [-all] [name...]
Enable and disable builtin shell commands. This allows the execution of a disk command which has the same **name** as a shell builtin without specifying a full pathname. If -**n** is used, each **name** is disabled; otherwise, names are enabled. For example, to use the test binary found via the **PATH** instead of the shell builtin version, type »**enable -n test**«. If no arguments are given, a list of all enabled shell builtins is printed. If only -**n** is supplied, a list of all disabled builtins is printed. If only -**all** is supplied, the list printed includes all builtins, with an indication of whether or not each is enabled. **enable** accepts **-a** as a synonym for **-all**. The return value is 0 unless a name is not a shell builtin.

eval [arg...]
The args are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of the **eval** command. If there are no **args**, or only null arguments, **eval** returns true.

exec [[-] command [arguments]]
If **command** is specified, it replaces the shell. No new process is created. The **arguments** become the arguments to **command**. If the first argument is **-**, the shell places a dash in the zeroth arg passed to **command**. This is what login does. If the file cannot be executed for some reason, a noninteractive shell exits, unless the shell variable **no_exit_on_failed_exec** exists, in which case it returns failure. An interactive shell returns failure if the file cannot be executed. If **command** is not specified, any redirections take effect in the current shell, and the return status is 0.

exit [n]
Cause the shell to **exit** with a status of **n.**.If **n** is omitted, the exit status is that of the last command executed. A **trap** on **EXIT** is executed before the shell terminates.

export [-nf] [name[=word]] ...
export -p
The supplied names are marked for automatic export to the environment of subsequently executed commands. If the **-f** option is given, the names refer to functions. If no names are given, or if the **-p** option is supplied, a list of all names that are exported in this shell is printed. The **-n** option causes the export property to be removed from the named variables. An argument of **--** disables option checking for the rest of the arguments. **export** returns an exit status of 0 unless an illegal option is encountered, one of the names is not a legal shell variable name, or **-f** is supplied with a name that is not a function.

fg [jobspec]
Place **jobspec** in the foreground,and make it the current job. If **jobspec** is not present, the shell's notion of the current job is used. The return value is that of the command placed into the foreground, or failure if run when job control is disabled or, when run with job control enabled, if **jobspec** does not specify a valid job or **jobspec** specifies a job that was started without job control.

getopts optstring name [args]
getopts is used by shell procedurestо parse positional parameters. **optstring** contains the option letters to be recognized; if a letter isfollowed by a colon, the option is expected to have an argument, which should be separated from it by white space. Each time it is invoked, **getopts** places the next option in the shell variable **name**, initializing **name** if it does not exist, and the index of the next argument to be processed into the variable **OPTIND**. **OPTIND** is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, **getopts** places that argument into the variable **OPTARG**. The shell does not reset **OPTIND** automatically; it must be manually reset between multiple calls to getopts within the same shell invocation if a new set of parameters is to be used. **getopts** can report errors in two ways. If the first character of **optstring** is a colon, silent error

reporting is used. In normal operation diagnostic messages are printed when illegal options or missing option arguments are encountered. If the variable **OPTERR** is set to 0, no error message will be displayed, even if the first character of **optstring** is not a colon.

If an illegal option is seen, **getopts** places **?** into **name** and, if not silent, prints an error message and unsets **OPTARG**. If **getopts** is silent, the option character found is placed in **OPTARG** and no diagnostic message is printed.

If a required argument is not found, and **getopts** is not silent, a question mark (**?**) is placed in **name**, **OPTARG** is unset, and a diagnostic message is printed. If **getopts** is silent, then a colon (**:**) is placed in **name** and **OPTARG** is set to the option character found.

getopts normally parses the positional parameters, but if more arguments are given in **args**, **getopts** parses those instead. **getopts** returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs.

help [pattern]

Display helpful information about builtin commands. If **pattern** is specified, **help** gives detailed help on all commands matching **pattern**; otherwise a list of the builtins is printed. The return status is 0 unless no command matches **pattern**.

history [n]
history -rwan [filename]

With no options, display the command **history** list with line numbers. Lines listed with a ***** have been modified. An argument of **n** lists only the last **n** lines. If a non-option argument is supplied, it is used as the name of the history file; if not, the value of **HISTFILE** is used. Options, if supplied, have the following meanings:

-a
Append the »new« history lines (history lines entered since the beginning of the current bash session) to the history file

-n
Read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current bash session.

-r
Read the contents of the history file and use them as the current history

-w
Write the current history to the history file, overwriting the history file's contents.

The return value is 0 unless an illegal option is encountered or an error occurs while reading or writing the history file.

jobs [-lnp] [jobspec...]
jobs -x command [args...]

The first form lists the active jobs. The **-l** option lists process IDs in addition to the normal information; the **-p** option lists only the process ID of the job's process group leader. The **-n** option displays only jobs that have changed status since last notified. If **jobspec** is given, output is restricted to information about that job. The return status is 0 unless an illegal option is encountered or an illegal jobspec is supplied.
If the **-x** option is supplied, jobs replaces any **jobspec** found in **command** or **args** with the corresponding process group ID, and executes **command** passing it **args**, returning its exit status.

kill [-s sigspec | -sigspec] [pid | jobspec] ...
kill -l [signum]

Send the signal named by **sigspec** to the processes named by **pid** or **jobspec**. **sigspec** is either a signal name such as **SIGKILL** or a signal number. If **sigspec** is a signal name, the name is case insensitive and may be given with or without the **SIG** prefix. If **sigspec** is not present, then **SIGTERM** is assumed. An argument of **-l** lists the signal names. If any arguments are supplied when **-l** is given, the names of the specified signals are listed, and the return status is 0. An argument of **--** disables option checking for the rest of the arguments. **kill** returns true if at least one signal was successfully sent, or false if an error occurs or an illegal option is encountered.

let arg [arg...]
Each arg is an arithmetic expression to be evaluated. If the last arg evaluates to 0, let returns 1; 0 is returned otherwise.

local [name[=value] ...]
For each argument, create a local variable named **name**, and assign it **value**. When **local** is used within a function, it causes the variable **name** to have a visible scope restricted to that function and its children. With no operands, **local** writes a list of local variables to the standard output. It is an error to use **local** when not within a function. The return status is 0 unless local is used outside a function, or an illegal **name** is supplied.

logout
Exit a login shell.

pwd
Print the absolute pathname of the current working directory. The path printed contains no symbolic links if the **-P** option to the **set** builtin command is set. The return status is 0 unless an error occurs while reading the pathname of the current directory.

read [-r] [name...]
One line is **read** from the standard input, and the first word is assigned to the first **name**, the second word to the second **name**, and so on, with leftover words assigned to the last name. Only the characters in **IFS** are recognized as word delimiters. If no names are supplied, the line read is assigned to the variable **REPLY**. The return code is zero, unless end-of-file is encountered. If the **-r** option is given, a backslash-newline pair is not ignored, and the backslash is considered to be part of the line.

readonly [-f] [name...]
readonly -p
The given names are marked **readonly** and the values of these names may not be changed by subsequent assignment. If the **-f** option is supplied, the functions corresponding to the names are so marked. If no arguments are given, or if the **-p** option is supplied, a list of all readonly names is printed. An argument of **--** disables option checking for the rest of the arguments. The return status is 0 unless an illegal option is encountered, one of the names is not a legal shell variable name, or **-f** is supplied with a name that is not a function.

return [n]
Causes a function to exit with the **return** value specified by **n**. If **n** is omitted, the return status is that of the last command executed in the function body. If used outside a function, but during execution of a script by the **.** (source) command, it causes the shell to stop executing that script and return either **n** or the exit status of the last command executed within the script as the exit status of the script. If used outside a function and not during execution of a script by **.**, the return status is false.

set [--abefhkmnptuvxldCHP] [-o option] [arg...]

-a
Automatically mark variables which are modified or created for export to the environment of subsequent commands.

-b
Cause the status of terminated background jobs to be reported immediately, rather than before the next primary prompt.

-e
Exit immediately if a simple-command exits with a non-zero status. The shell does not exit if the command that fails is part of an **until** or **while** loop, part of an **if** statement, part of a **&&** or **||** list, or if the command's return value is being inverted via **!**.

-f
Disable pathname expansion.

-h
Locate and remember function commands as functions are defined. Function commands are normally looked up when the function is executed.

-k
All keyword arguments are placed in the environment for a command, not just those that precede the command name.

-m
Monitor mode. Job control is enabled. This flag is on by default for interactive shells on systems that support it. Background processes run in a separate process group and a line containing their exit status is printed upon their completion.

-n
Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored for interactive shells.

-p
Turn on privileged mode. In this mode, the **$ENV** file is not processed, and shell functions are not inherited from the environment. This is enabled automatically on startup if the effective user (group) id is not equal to the real user (group) id. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

-t
Exit after reading and executing one command.

-u
Treat unset variables as an error when performing parameter expansion. If expansion is attempted on an unset variable, the shell prints an error message, and, if not interactive, exits with a non-zero status.

-v
Print shell input lines as they are read.

-x
After expanding each simple-command, bash displays the expanded value of **PS4**, followed by the command and its expanded arguments.

-l
Save and restore the binding of name in a for name [in word] command.

-d
Disable the hashing of commands that are looked up for execution. Normally, commands are remembered in a hash table, and once found, do not have to be looked up again.

-C
The effect is as if the shell command »**noclobber=&**« had been executed.

-H
Enable ! style history substitution. This flag is on by default when the shell is interactive.

-P
If set, do not follow symbolic links when performing commands such as **cd** which change the current directory. The physical directory is used instead.

--
If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the args, even if some of them begin with a **-**.

-
Signal the end of options, cause all remaining args to be assigned to the positional parameters. The **-x** and **-v** options are turned off. If there are no args, the positional parameters remain unchanged.

The flags are off by default unless otherwise noted. Using **+** rather than **-** causes these flags to be turned off. The flags can also be specified as options to an invocation of the shell. The current set of flags may be found in **$-**. After the **option** arguments are processed, the remaining n args are treated as values for the positional parameters and are assigned, in order, to **$1**, **$2**, ... **$n**. If no options or args are supplied, all shell variables are printed. The return status is always true unlessan illegal option is encountered.

shift [n]
The positional parameters from **n+1 ...** are renamed to **$1 ....** Parameters represented by the numbers **$#** down to **$#-n+1** are unset. If **n** is 0, no parameters are changed. If **n** is not given, it is assumed to be 1. **n** must be a non-negative number less than or equal to **$#**. If **n** is greater than **$#**,

the positional parameters are not changed. The return status is greater than 0 if n is greater than **$#** or less than 0; otherwise 0.

suspend [-f]
Suspend the execution of this shell until it receives a **SIGCONT** signal. The **-f** option says not to complain if this is a login shell; just suspend anyway. The return status is 0 unless the shell is a login shell and **-f** is not supplied, or if job control is not enabled.

test expr
[expr]
Return a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression **expr**. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. Each operator and operand must be a separate argument. If file is of the form **/dev/fd/n**, then file descriptor **n** is checked.

-b file
True if **file** exists and is block special.

-c file
True if **file** exists and is character special.

-d file
True if **file** exists and is a directory.

-e file
True if **file** exists.

-f file
True if **file** exists and is a regular file.

-g file
True if **file** exists and is set-group-id.

-k file
True if **file** has its »sticky« bit set.

-L file
True if **file** exists and is a symbolic link.

-p file
True if **file** exists and is a named pipe.

-r file
True if **file** exists and is readable.

-s file
True if **file** exists and has a size greater than zero.

-S file
True if **file** exists and is a socket.

-t fd
True if **fd** is opened on a terminal.

-u file
True if **file** exists and its set-user-id bit is set.

-w file
True if **file** exists and is writable.

-x file
True if **file** exists and is executable.

-O file
True if **file** exists and is owned by the effective user id.

-G file
True if **file** exists and is owned by the effective group id.

file1 -nt file2
True if **file1** is newer (according to modification date) than **file2**.

file1 -ot file2
True if **file1** is older than **file2**.

file1 -ef file
True if **file1** and **file2** have the same device and inode numbers.

-z string
True if the length of **string** is zero.

-n string
True if the length of **string** is non-zero.

string1 = string2
True if the strings are equal.

string1 != string2
True if the strings are not equal.

! expr
True if **expr** is false.

expr1 -a expr2
True if both **expr1** AND **expr2** are true.

expr1 -o expr2
True if either **expr1** OR **expr2** is true.

arg1 OP arg2
OP is one of **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**. These arithmetic binary operators return true if **arg1** is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than **arg2**, respectively. **Arg1** and **arg2** may be positive integers, negative integers, or the special expression **-l string**, which evaluates to the length of string.

times
Print the accumulated user and system times for the shell and for processes run from the shell. The return status is 0.

trap [-l] [arg] [sigspec]
The command **arg** is to be read and executed when the shell receives signal(s) **sigspec**. If **arg** is absent or **-**, all specified signals are reset to their original values (the values they had upon entrance to the shell). If **arg** is the null string this signal is ignored by the shell and by the commands it invokes. **sigspec** is either a signal name defined in <signal.h>, or a signal number. If **sigspec** is **EXIT** (**0**) the command **arg** is executed on exit from the shell. With no arguments, **trap** prints the list of commands associated with each signal number. The **-l** option causes the shell to print a list of signal names and their corresponding numbers. An argument of **--** disables option checking for the rest of the arguments. Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created. The return status is false if either the trap name or number is invalid; otherwise **trap** returns true.

type [-all] [-type | -path] name [name...]
With no options, indicate how each name would be interpreted if used as a command name. If the **-type** flag is used, **type** prints a phrase which is one of **alias**, **keyword**, **function**, **builtin**, or **file** if **name** is an alias, shell reserved word, function, builtin, or disk file, respectively. If the **name** is not found, then nothing is printed, and an exit status of false is returned. If the **-path** flag is used, **type** either returns the name of the disk file that would be executed if **name** were specified as a command name, or nothing if **-type** would not return file. If a command is hashed, **-path** prints the hashed value, not necessarily the file that appears first in **PATH**. If the **-all** flag is used, **type** prints all of the places that contain an executable named **name**. This includes aliases and functions, if and only if the **-path** flag is not also used. The table of hashed commands is not consulted when using **-all**. **type** accepts **-a**, **-t**, and **-p** in place of **-all**, **-type**, and **-path**, respectively. An argument of **--** disables option checking for the rest of the arguments. **type** returns true if any of the arguments are found, false if none are found.

ulimit [-SHacdfmstpnuv [limit]]
ulimit provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The value of limit can be a number in the unit specified for the resource, or the value unlimited. The **H** and **S** options specify that the hard or soft limit is set for the given resource. A hard limit cannot be increased once it is set; a soft limit may be increased up to the value of the hard limit. If neither **H** nor **S** is specified, the command applies to the soft limit. If **limit**

is omitted, the current value of the soft limit of the resource is printed, unless the **H** option is given. When more than one resource is specified, the **limit** name and unit is printed before the value. Other options are interpreted as follows:

-a
all current limits are reported

-c
the maximum size of core files created

-d
the maximum size of a process's data segment

-f
the maximum size of files created by the shell

-m
the maximum resident set size

-s
the maximum stack size

-t
the maximum amount of cpu time in seconds

-p
the pipe size in 512-byte blocks (this may not be set)

-n
the maximum number of open file descriptors (most systems do not allow this value to be set, only displayed)

-u
the maximum number of processes available to a single user

-v
The maximum amount of virtual memory available to the shell

An argument of **--** disables option checking for the rest of the arguments. If **limit** is given, it is the new value of the specified resource (the **-a** option is display only). If no option is given, then **-f** is assumed. Values are in 1024-byte increments, except for **-t**, which is in seconds, **-p**, which is in units of 512-byte blocks, and **-n** and **-u**, which are unscaled values. The return status is 0 unless an illegal option is encountered, a non-numeric argument other than unlimited is supplied as **limit**, or an error occurs while setting a new **limit**.

umask [-S] [mode]
The user file-creation mask is set to **mode**. If **mode** begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by **chmod**(1). If **mode** is omitted, or if the **-S** option is supplied, the current value of the mask is printed. The **-S** option causes the mask to be printed in symbolic form; the default output is an octal number. An argument of **--** disables option checking for the rest of the arguments. The return status is 0 if the **mode** was successfully changed or if no mode argument was supplied, and false otherwise.

unalias [-a] [name...]
Remove names from the list of defined aliases. If **-a** is supplied, all alias definitions are removed. The return value is true unless a supplied name is not a defined alias.

unset [-fv] [name...]
For each **name**, remove the corresponding variable or, given the **-f** option, function. An argument of **--** disables option checking for the rest of the arguments. Note that **PATH**, **IFS**, **PPID**, **PS1**, **PS2**, **UID**, and **EUID** cannot be unset. If any of **RANDOM**, **SECONDS**, **LINENO**, or **HISTCMD** are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless a name does not exist or is non-unsettable.

wait [n]
Wait for the specified process and return its termination status. **n** may be a process ID or a job specification; if a job spec is given, all processes in that job's pipeline are waited for. If **n** is not given, all currently active child processes are waited for, and the return status is zero. If **n** specifies a non-exis-

tant process or job, the return status is 127. Otherwise, the return status is the exit status of the last process or job waited for.

Die Befehlszeile der „BaSH“

Die Befehlszeile selbst ist gewissermaßen die zentrale Schnittstelle zwischen der Anwenderin auf der einen in der *BaSH* mit ihren Möglichkeiten auf der anderen Seite. Hier werden die Tastatur-Eingaben durch mehrere Funktionen unterstützt bzw. aufbereitet: die *readline*-Bibliothek der *shell* stellt eine Art Kommandozeilen-Editor zur Verfügung, der u.a. Zugriff auf die Befehls- *history* bietet, sowie die verschiedenen *Expansionen* schließlich ermöglichen – vor allem im Rahmen von *shell-scripts* – das Arbeiten mit Variablen numerischen und alphanumerischen Inhaltes.

Sonderzeichen und reservierte Wörter

Einige Zeichen und Wörter haben bei der Arbeit mit der *shell* eine besondere Bedeutung. Sie sind wie folgt definiert:

blank
Ein Leer- oder Tabulator-Zeichen.

word
Eine Zeichenfolge, die von der *shell* als Einheit aufgefaßt wird, auch *token* genannt.

name
Ein *word*, das nur aus alphanumerischen Zeichen sowie dem Unterstrich besteht und mit einem alphabetischen Zeichen oder einem Unterstrich beginnt. Auch als *identifier* bezeichnet.

metacharacter
Ein Zeichen, das - wenn nicht zitiert (*quoted*) - einzelne *words* voneinander trennt. Es gibt die folgenden:
`| & ; ( ) < > space tab`

control operator
Ein *token*, das eine Kontroll-Funktion auslöst; es ist eines der folgenden Symbole:
`|| & && ; ;; ( ) | <newline>`

reserved words
sind Wörter, die in der *shell* eine spezielle Bedeutung haben. Die folgenden Wörter werden als reserviert erkannt, soweit sie nicht zitiert (*quoted*) sind und entweder das erste Wort eines einfachen Befehles oder das dritte Wort einer **`case`**- oder **`for`**-Anweisung sind:
`! case do done elif else esac fi for`
`function if in select then until while { }`

Während die reservierten Wörter wohl überwiegend im Rahmen von *shell-scripts* Verwendung finden, können Meta-Zeichen und Kontroll-Operatoren auch direkt in der Kommandozeile be nutzt werden.

Die readline-Bibliothek

This is the library that handles reading input when using an interactive shell, unless the **`-noline-editing`** option is given. By default, the line editing commands are similar to those of **`emacs`**. A **`vi`**-style line editing interface is also available.

In this section, the **`emacs`**-style notation is used to denote keystrokes. *Control keys* are denoted by **`C-key`**, e.g., **`C-n`** means *Control-N*. Similarly, *meta keys* are denoted by **`M-key`**, so **`M-x`** means *Meta-X*. (On keyboards without a meta key, **`M-x`** means **`ESC x`**, i.e., press the **`Esc`**ape key then the **`x`** key. This makes **`ESC`** the *meta prefix*. The combination **`M-C-x`** means *ESC-Control-x*, or press the Escape key then hold the Control key while pressing the x key.)

The default key-bindings may be changed with an **`~/.inputrc`** file. The value of the shell variable **`INPUTRC`**, if set, is used instead of **`~/.inputrc`**. Other programs that use this library may add their own commands and bindings.

For example, placing

M-Control-u: universal-argument

or

C-Meta-u: universal-argument

into the ~/.inputrc would make **M-C-u** execute the readline command *universal-argument*.

The following symbolic character names are recognized: *RUBOUT*, *DEL*, *ESC*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *SPC*, *SPACE*, and *TAB*. In addition to command names, readline allows keys to be bound to a string that is inserted when the key is pressed (a *macro*).

Readline is customized by putting commands in an initialization file. The name of this file is taken from the value of the **INPUTRC** variable. If that variable is unset, the default is **~/.inputrc**. When a program which uses the readline library starts up, the init file is read, and the key bindings and variables are set. There are only a few basic constructs allowed in the readline init file. Blank lines are ignored. Lines beginning with a **#** are comments. Lines beginning with a **$** indicate conditional constructs. Other lines denote key bindings and variable settings.

The syntax for controlling key bindings in the **~/.inputrc** file is simple. All that is required is the name of the command or the text of a macro and a key sequence to which it should be bound. The name may be specified in one of two ways: as a symbolic key name, possibly with Meta- or Control- prefixes, or as a key sequence. When using the form *keyname:function-name or macro*, keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: ">&output"
```

In the above example, **C-u** is bound to the function *universal-argument*, **M-DEL** is bound to the function *backward-kill-word*, and **C-o** is bound to run the macro expressed on the right hand side (that is, to insert the text **>&output** into the line).

In the second form, **"keyseq":function-name or macro**, *keyseq* differs from *keyname* above in that strings denoting an entire key sequence may be specified by placing the sequence within double quotes. Some GNU Emacs style key escapes can be used, as in the following example.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In this example, **C-u** is again bound to the function *universal-argument*. **C-x C-r** is bound to the function *re-read-init-file*, and **ESC[11~** is bound to insert the text *Function Key 1*. The full set of escape sequences is

\C-
: control prefix

\M-
: meta prefix

\e
: an escape character

****
: backslash

\"
: literal **"**

\'
: literal **'**

When entering the text of a macro, single or double quotes should be used to indicate a macro definition. Unquoted text is assumed to be a function name. Backslash will quote any character in the macro text, including **"** and **'**.

Bash allows the current readline key bindings to be displayed or modified with the **bind** builtin command. The editing mode may be switched during interactive use by using the **-o option** to the **set** builtin command.

Readline has variables that can be used to further customize its behavior. A variable may be set in the inputrc file with a statement of the form

set variable-name value

Except where noted, readline variables can take the values **On** or **Off**. The variables and their default values are:

horizontal-scroll-mode (**Off**)
When set to **On**, makes readline use a single line for display, scrolling the input horizontally on a single screen line when it becomes longer than the screen width rather than wrapping to a new line.

editing-mode (**emacs**)
Controls whether readline begins with a set of key bindings similar to **emacs** or **vi**. editing-mode can be set to either **emacs** or **vi**.

mark-modified-lines (**Off**)
If set to **On**, history lines that have been modified are displayed with a preceding asterisk (*****).

bell-style (**audible**)
Controls what happens when readline wants to ring the terminal bell. If set to **none**, readline never rings the bell. If set to **visible**, readline uses a visible bell if one is available. If set to **audible**, readline attempts to ring the terminal's bell.

comment-begin (»**#**«)
The string that is inserted in **vi** mode when the vi-comment command is executed.

meta-flag (**Off**)
If set to **On**, readline will enable eight-bit input (that is, it will not strip the high bit from the characters it reads), regardless of what the terminal claims it can support.

convert-meta (**On**)
If set to **On**, readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prepending an escape character (in effect, using escape as the meta prefix).

output-meta (**Off**)
If set to **On**, readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence.

completion-query-items (**100**)
This determines when the user is queried about viewing the number of possible completions generated by the *possible-completions* command. It may be set to any integer value greater than or equal to zero. If the number of possible completions is greater than or equal to the value of this variable, the user is asked whether or not he wishes to view them; otherwise they are simply listed on the terminal.

keymap (**emacs**)
Set the current readline keymap. The set of legal keymap names is **emacs**, **emacs-standard**, **emacs-meta**, **emacs-ctlx**, **vi**, **vi-move**, **vi-command**, and **vi-insert**. **vi** is equivalent to **vi-command**; **emacs** is equivalent to **emacs-standard**. The default value is **emacs**; the value of editing-mode also affects the default keymap.

show-all-if-ambiguous (**Off**)
This alters the default behavior of the completion functions. If set to **on**, words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell.

expand-tilde (**Off**)
If set to **on**, tilde expansion is performed when readline attempts word completion.

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are three parser directives used.

$if
The **$if** construct allows bindings to be made based on the editing mode, the terminal being used, or

the application using readline. The text of the test extends to the end of the line; no characters are required to isolate it.

mode

The **mode=** form of the **$if** directive is used to test whether readline is in **emacs** or **vi** mode. This may be used in conjunction with the set keymap command, for instance, to set bindings in the emacs-standard and emacs_ctlx keymaps only if readline is starting out in emacs mode.

term

The **term=** form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the **=** is tested against the full name of the terminal and the portion of the terminal name before the first **-**. This allows sun to match both sun and sun-cmd, for instance.

application

The application construct is used to include application-specific settings. Each program using the readline library sets the application name, and an initialization file can test for a particular value. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if Bash
    # Quote the current or previous word
    "\C-xq": "\eb\"\ef\""
$endif
```

$endif

This command, as you saw in the previous example, terminates an **$if** command.

$else

Commands in this branch of the **$if** directive are executed if the test fails.

Readline commands may be given numeric arguments, which normally act as a repeat count. Sometimes, however, it is the sign of the argument that is significant. Passing a negative argument to a command that acts in the forward direction (e.g., *kill-line*) causes that command to act in a backward direction. Commands whose behavior with arguments deviates from this are noted.

When a command is described as *killing text*, the text deleted is saved for possible future retrieval (*yanking*). The killed text is saved in a kill-ring. Consecutive kills cause the text to be accumulated into one unit, which can be yanked all at once. Commands which do not kill text separate the chunks of text on the kill-ring.

```
[matthias@Barbara Bash-Kurs]$ cat ~/.inputrc
# ~/.inputrc

# 8-Bit Eingaben (Umlaute) erlauben:
set meta-flag on

# 8-Bit Ausgaben (Umlaute) erlauben:
set output-meta on

# 8-Bit Zeichen nicht konvertieren:
set convert-meta off

$if term=xterm
    # wir sind in einem xterm-Prozess
    # Home:
    "\e[b": beginning-of-line
    # Del:
    "\e[d": delete-char
    # End:
    "\e[e": end-of-line
$else
    # wir sind auf der Konsole
    # Home:
    "\e[1~": beginning-of-line
    # Del:
    "\e[3~": delete-char
    # End:
    "\e[4~": end-of-line
$endif
```

```
[matthias@Barbara Bash-Kurs]$ _
```

The following is a list of the names of the commands and the default key sequences to which they are bound.

Commands for Moving

beginning-of-line (**C-a**)
Move to the start of the current line.

end-of-line (**C-e**)
Move to the end of the line.

forward-char (**C-f**)
Move forward a character.

backward-char (**C-b**)
Move back a character.

forward-word (**M-f**)
Move forward to the end of the next word. Words are composed of alphanumeric characters (letters and digits).

backward-word (**M-b**)
Move back to the start of this, or the previous, word. Words are composed of alphanumeric characters (letters and digits).

clear-screen (**C-l**)
Clear the screen leaving the current line at the top of the screen. With an argument, refresh the current line without clearing the screen.

redraw-current-line
Refresh the current line. By default, this is unbound.

Commands for Manipulating the History

accept-line (Newline, **Return**)
Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list according to the state of the **HISTCONTROL** variable. If the line is a modified history line, then restore the history line to its original state.

previous-history (**C-p**)
Fetch the previous command from the history list, moving back in the list.

next-history (**C-n**)
Fetch the next command from the history list, moving forward in the list.

beginning-of-history (**M-<**)
Move to the first line in the history.

end-of-history (**M->**)
Move to the end of the input history, i.e., the line currently being entered.

reverse-search-history (**C-r**)
Search backward starting at the current line and moving »up« through the history as necessary. This is an incremental search.

forward-search-history (**C-s**)
Search forward starting at the current line and moving »down« through the history as necessary. This is an incremental search.

non-incremental-reverse-search-history (**M-p**)
Search backward through the history starting at the current line using a non-incremental search for a string supplied by the user.

non-incremental-forward-search-history (**M-n**)
Search forward through the history using a non-incremental search for a string supplied by the user.

history-search-forward
Search forward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

history-search-backward
Search backward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

yank-nth-arg (**M-C-y**)
Insert the first argument to the previous command (usually the second word on the previous line) at point (the current cursor position). With an argument **n**, insert the **n**th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the **n**th word from the end of the previous command.

yank-last-arg (**M- ,M-_**)
Insert the last argument to the previous command (the last word on the previous line). With an argument, behave exactly like *yank-nth-arg*.

shell-expand-line (**M-C-e**)
Expand the line the way the shell does when it reads it. This performs alias and history expansion as well as all of the shell word expansions.

history-expand-line (**M-^**)
Perform history expansion on the current line.

insert-last-argument (**M- ,M-_**)
A synonym for *yank-last-arg*.

operate-and-get-next (**C-o**)
Accept the current line for execution and fetch the next line relative to the current line from the history for editing. Any argument is ignored.

Commands for Changing Text

delete-char (**C-d**)
Delete the character under the cursor. If point is at the beginning of the line, there are no characters in the line, and the last character typed was not C-d, then return EOF.

backward-delete-char (**Rubout**)
Delete the character behind the cursor. When given a numeric argument, save the deleted text on the kill-ring.

quoted-insert (**C-q**, **C-v**)
Add the next character that you type to the line verbatim. This is how to insert characters like C-q, for example.

tab-insert (**C-v TAB**)
Insert a tab character.

self-insert (**a**, **b**, **A**, **1**, **!**, ...)
Insert the character typed.

transpose-chars (**C-t**)
Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative arguments don't work.

transpose-words (**M-t**)
Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

upcase-word (**M-u**)
Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

downcase-word (**M-l**)
Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

capitalize-word (**M-c**)
Capitalize the current (or following) word. With a negative argument, do the previous word, but do not move point.

Killing and Yankin

kill-line (**C-k**)
Kill the text from the current cursor position to the end of the line.

backward-kill-line (**C-x C-Rubout**)
Kill backward to the beginning of the line.

unix-line-discard (**C-u**)
Kill backward from point to the beginning of the line.

kill-whole-line
Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

kill-word (**M-d**)
Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by forward-word.

backward-kill-word (**M-Rubout**)
Kill the word behind the cursor. Word boundaries are the same as those used by backward-word.

unix-word-rubout (**C-w**)
Kill the word behind the cursor, using white space as a word boundary. The word boundaries are different from backward-kill-word.

delete-horizontal-space
Delete all spaces and tabs around point. By default, this is unbound.

yank (**C-y**)
Yank the top of the kill ring into the buffer at the cursor.

yank-pop (**M-y**)
Rotate the kill-ring, and yank the new top. Only works following yank or yank-pop.

Numeric Arguments

digit-argument (**M-0**, **M-1**, ..., **M--**)
Add this digit to the argument already accumulating, or start a new argument. **M--** starts a negative argument.

universal-argument
Each time this is executed, the argument count is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four. By default, this is not bound to a key.

Completing

complete (**TAB**)
Attempt to perform completion on the text before point. Bash attempts completion treating the text as a variable (if the text begins with **$**), username (if the text begins with **~**), hostname (if the text begins with **@**), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted.

possible-completions (**M-?**)
List the possible completions of the text before point.

insert-completions
Insert all completions of the text before point that would have been generated by possible-completions. By default, this is not bound to a key.

complete-filename (**M-/**)
Attempt filename completion on the text before point.

possible-filename-completions (**C-x /**)
List the possible completions of the text before point, treating it as a filename.

complete-username (**M-~**)
Attempt completion on the text before point, treating it as a username.

possible-username-completions (**C-x ~**)
List the possible completions of the text before point, treating it as a username.

complete-variable (**M-$**)
Attempt completion on the text before point, treating it as a shell variable.

possible-variable-completions (**C-x $**)
List the possible completions of the text before point, treating it as a shell variable.

complete-hostname (**M-@**)
Attempt completion on the text before point, treating it as a hostname.

possible-hostname-completions (**C-x @**)
List the possible completions of the text before point, treating it as a hostname.

complete-command (**M-!**)
Attempt completion on the text before point, treating it as a command name. Command completion attempts to match the text against aliases, reserved words, shell functions, builtins, and finally executable filenames, in that order.

possible-command-completions (**C-x !**)
List the possible completions of the text before point, treating it as a command name.

dynamic-complete-history (**M-TAB**)
Attempt completion on the text before point, comparing the text against lines from the history list for possible completion matches.

complete-into-braces (**M-{**)
Perform filename completion and return the list of possible completions enclosed within braces so the list is available to the shell (see Brace Expansion above).

Keyboard Macros

start-kbd-macro (**C-x (**)
Begin saving the characters typed into the current keyboard macro.

end-kbd-macro (**C-x)**)
Stop saving the characters typed into the current keyboard macro and save the definition.

call-last-kbd-macro (**C-x e**)
Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Miscellaneous

re-read-init-file (C-x C-r)
Read in the contents of your init file, and incorporate any bindings or variable assignments found there.

abort (C-g)
Abort the current editing command and ring the terminal's bell (subject to the setting of bell-style).

do-uppercase-version (**M-a**, **M-b**, ...)
Run the command that is bound to the corresponding uppercase character.

prefix-meta (**ESC**)
Metafy the next character typed. ESC f is equivalent to Meta-f.

undo (**C-_**, **C-x C-u**)
Incremental undo, separately remembered for each line.

revert-line (**M-r**)
Undo all changes made to this line. This is like typing the undo command enough times to return the line to its initial state.

tilde-expand (**M-~**)
Perform tilde expansion on the current word.

dump-functions
Print all of the functions and their key bindings to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an inputrc file.

display-shell-version (**C-x C-v**)
Display version information about the current instance of bash.

Befehls-Geschichte („History“)

Alle eingegebenen Befehle werden in einer sog. *history* gespeichert. Dabei handelt es sich um eine Datei im **HOME**-Verzeichnis namens **.bash_history**, deren Namen in der Umgebungsvariable **HISTFILE** abgelegt ist und derenUmfang man durch die Variable **HISTFILESIZE** beeinflussen kann. Obzwar diese Datei auch eine gewisse Protokollierungs- und damit Überwachungs-Funktion wahrnehmen kann, dient sie doch v.a. dazu, das mehrfache Eingeben gleicher –oder, mithilfe der *readline*-Möglichkeiten: ähnlicher –Befehlsfolgen zu erübrigen bzw. zu vereinfachen. Die Möglichkeiten der *history* sind bei interaktiven Shells standardmäßig aktiviert, können aber mit der **+H** Option des eingebauten **set**-Befehles ausgeschaltet werden. Nicht-interaktive Shells nutzen die Befehls-*history* nicht.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words. It takes place in two parts. The first is to determine which line from the previous history to use during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is the event, and the portions of that line that are acted upon are words. The line is broken into words in the same fashion as when reading input, so that several metacharacter-separated words surrounded by quotes are considered as one word. Only backslash (\) and single quotes can quote the history escape character, which is **!** by default. The shell allows control of the various characters used by the history expansion mechanism.

Event Designators

An *event designator* is a reference to a command line entry in the history list.

!
Start a history substitution, except when followed by a blank, newline, **=** or **(**.

!!
Refer to the previous command. This is a synonym for »**!-1**«.

!n
Refer to command line **n**.

!-n
Refer to the current command line minus **n**.

!string
Refer to the most recent command starting with **string**.

!?string[?]
Refer to the most recent command containing **string**.

^string1^string2^
Quick substitution. Repeat the last command, replacing **string1** with **string2**. Equivalent to »**!!:s/string1/string2/**« (see Modifiers below).

!#
The entire command line typed so far.

Word Designators

A **:** separates the event specification from the word designator. It can be omitted if the *word designator* begins with a **^**, **$**, *****, or **%**. Words are numbered from the beginning of the line, with the first word being denoted by a **0** (zero).

0 (zero)
The zeroth word. For the shell, this is the command word.

n
The **n**th word.

^
The first argument. That is, word 1.

$
The last argument.

%
The word matched by the most recent »**?string?**« search.

x-y
A range of words; »**-y**« abbreviates »**0-y**«.

*****
All of the words but the zeroth. This is a synonym for »**1-$**«. It is not an error to use ***** if there is just one word in the event; the empty string is returned in that case.

x*
Abbreviates **x-$**.

x-
Abbreviates **x-$** like **x***, but omits the last word.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following *modifiers*, each preceded by a »**:**«.

h
Remove a trailing pathname component, leaving only the head.

r
Remove a trailing suffix of the form **.xxx**, leaving the basename.

e
Remove all but the trailing suffix.

t
Remove all leading pathname components, leaving the tail.

p
Print the new command but do not execute it.

q
Quote the substituted words, escaping further substitutions.

x
Quote the substituted words as with q, but break into words at blanks and newlines.

s/old/new/
Substitute **new** for the first occurrence of **old** in the event line. Any delimiter can be used in place of **/**. The final delimiter is optional if it is the last character of the event line. The delimiter may be quoted in **old** and **new** with a single backslash. If **&** appears in **new**, it is replaced by **old**. A single backslash will quote the **&**.

&
Repeat the previous substitution.

g
Cause changes to be applied over the entire event line. This is used in conjunction with »**:s**« (e.g., »**:gs/old/new/**«) or »**:&**«. If used with »**:s**«, any delimiter can be used in place of **/**, and the final delimiter is optional if it is the last character of the event line.

```
[matthias@Barbara Bash-Kurs]$ /bin/ls -l
total 7
drwxr-xr-x 2 matthias matthias  1024 Aug 19 18:27 Scripts
-rw-rw-r-- 1 matthias matthias  4700 Aug 20 15:29 hist.log
```

```
-rwxr-xr-x 1 matthias matthias    40 Aug 20 15:14 log.scr


[matthias@Barbara Bash-Kurs]$ !!
/bin/ls -l
total 7
drwxr-xr-x 2 matthias matthias  1024 Aug 19 18:27 Scripts
-rw-rw-r-- 1 matthias matthias  4944 Aug 20 15:30 hist.log
-rwxr-xr-x 1 matthias matthias    40 Aug 20 15:14 log.scr


[matthias@Barbara Bash-Kurs]$ !-1
/bin/ls -l
total 8
drwxr-xr-x 2 matthias matthias  1024 Aug 19 18:27 Scripts
-rw-rw-r-- 1 matthias matthias  5189 Aug 20 15:30 hist.log
-rwxr-xr-x 1 matthias matthias    40 Aug 20 15:14 log.scr


[matthias@Barbara Bash-Kurs] > !:0
/bin/ls
Scripts
hist.log
log.scr


[matthias@Barbara Bash-Kurs]$ !-2
/bin/ls -l
total 8
drwxr-xr-x 2 matthias matthias  1024 Aug 19 18:27 Scripts
-rw-rw-r-- 1 matthias matthias  5503 Aug 20 15:30 hist.log
-rwxr-xr-x 1 matthias matthias    40 Aug 20 15:14 log.scr


[matthias@Barbara Bash-Kurs]$ !!:t
ls -l
total 8
drwxr-xr-x 2 matthias matthias  1024 Aug 19 18:27 [[32mScripts[[0m/
-rw-rw-r-- 1 matthias matthias  5744 Aug 20 15:31 [[0mhist.log[[0m
-rwxr-xr-x 1 matthias matthias    40 Aug 20 15:14 [[35mlog.scr[[0m*


[matthias@Barbara Bash-Kurs]$ !-3
/bin/ls
Scripts
hist.log
log.scr


[matthias@Barbara Bash-Kurs]$ ^s^s -la
/bin/ls -la
total 10
drwxr-xr-x 3 matthias matthias  1024 Aug 20 15:14 .
drwxr-xr-x 4 matthias matthias  1024 Jul 29 15:07 ..
drwxr-xr-x 2 matthias matthias  1024 Aug 19 18:27 Scripts
-rw-rw-r-- 1 matthias matthias  6092 Aug 20 15:31 hist.log
-rwxr-xr-x 1 matthias matthias    40 Aug 20 15:14 log.scr


[matthias@Barbara Bash-Kurs]$ host -a barbara
Trying domain "dfg.de"
rcode = 0 (Success), ancount=3
barbara.dfg.de  86400 IN HINFO   AMD486  Linux/CND
barbara.dfg.de  86400 IN MX (pri=10) by Mail.dfg.de
barbara.dfg.de  86400 IN A   192.168.192.200
For authoritative answers, see:
dfg.de  86400 IN     NS  NS.dfg.de
Additional information:
Mail.dfg.de   86400 IN   A   192.168.192.194
NS.dfg.de  86400 IN   A   192.168.192.197


[matthias@Barbara Bash-Kurs]$ ^host -a^ping
ping barbara
PING Barbara.dfg.de (192.168.192.200): 56 data bytes
64 bytes from 192.168.192.200: icmp_seq=0 ttl=255 time=7.0 ms
64 bytes from 192.168.192.200: icmp_seq=1 ttl=255 time=3.7 ms
64 bytes from 192.168.192.200: icmp_seq=2 ttl=255 time=3.6 ms
--- Barbara.dfg.de ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 3.6/4.7/7.0 ms


[matthias@Barbara Bash-Kurs]$ _
```

häufig genutzte Dienstprogramme

> [Mittwoch]

Hilfe finden

Viele Programme geben bei einem Aufruf mit dem Parameter »**-h**« oder »**--help**« eine knappe Übersicht über ihre Aufgabe und ihre Aufruf-Optionen auf die Konsole aus. Ein solcher Aufruf ist also immer einen Versuch wert.

Zum Formatieren und Anzeigen der online-verfügbaren *man-pages* steht das **man**-Programm zur Verfügung. Dieser Befehl ist somit zentral, wenn es darum geht, Informationen zu einem Programm zu erhalten. Die Aufruf-Optionen:

```
[matthias@Barbara Bash-Kurs]$ man -h
man, version  1.4d
usage: man [-adfhktwW] [section] [-M path] [-P pager] [-S list] [-m system] [-p string] name ...
    a : find all matching entries
    c : do not use cat file
    d : print gobs of debugging information
    D : as for -d, but also display the pages
    f : same as whatis(1)
    h : print this help message
    k : same as apropos(1)
    t : use troff to format pages for printing
    w : print location of man page(s) that would be displayed
        (if no name given: print directories that would be searched)
    W : as for -w, but display filenames only

    C file : use `file' as configuration file
    M path : set search path for manual pages to `path'
    P pager  : use program `pager' to display pages
    S list : colon separated section list
    m system : search for alternate system's man pages
    p string : string tells which preprocessors to run
        e - [n]eqn(1)   p - pic(1)    t - tbl(1)
        g - grap(1)     r - refer(1)  v - vgrind(1)


[matthias@Barbara Bash-Kurs]$ _
```

Üblicherweise wird der Aufruf nach dem Muster **man program** vorgenommen, sofern es indes zu einem Stichwort mehrere *man-pages* gibt, ist es wichtig, zur Unterscheidung die gewünschte Seite durch Angabe der *Sektion* näher zu bezeichnen, z.B. **man 7 signal**.

Schließlich bietet die *BaSH* durch ihren eingebauten **help**-Befehl die Möglichkeit, sich zu ihren eigenen Befehlen kurze Beschreibungen anzeigen zu lassen:

```
[matthias@Barbara Bash-Kurs]$ help help
help: help [pattern ...]
    Display helpful information about builtin commands. If PATTERN is
    specified, gives detailed help on all commands matching PATTERN,
    otherwise a list of the builtins is printed.


[matthias@Barbara Bash-Kurs]$ _
```

Daneben gibt es noch die Befehle **apropos** und **info**, mit denen sich zu einer Reihe von Programmen Hilfetexte abrufen lassen.

Anzeigen von Dateien

Um den Inhalt von Dateien einzusehen, stehen gleich mehrere Programme zur Verfügung.

cat

```
[matthias@Barbara Bash-Kurs]$ cat --help Usage: cat [OPTION] [FILE]...
Concatenate FILE(s), or standard input, to standard output.

   -b, --number-nonblank    number nonblank output lines
   -e                       equivalent to -vE
   -n, --number             number all output lines
   -s, --squeeze-blank      never more than one single blank line
   -t                       equivalent to -vT
   -u                       (ignored)
   -v, --show-nonprinting   use ^ and M- notation, save for LFD and TAB
   -A, --show-all           equivalent to -vET
   -E, --show-ends          display $ at end of each line
   -T, --show-tabs          display TAB characters as ^I
       --help               display this help and exit
       --version            output version information and exit

With no FILE, or when FILE is -, read standard input.


[matthias@Barbara Bash-Kurs]$ _
```

Dieser Befehl, der hier auch zur Anzeige der Beispiel-Scripts verwendet wird, gibt die im als Aufruf-Parameter übergebenen Dateien auf die Konsole aus. Er kann aber auch im Rahmen einer Pipeline verwendet werden, beispielsweise um mehrere Dateien in einem sog. **pager** anzuzeigen:

```
cat *.sh | less
```

more und less

Häufig sind Dateien zu lang, um sie am Bildschirm „am Stück" lesen zu können. Hier kommen die pager ins Spiel, die es ermöglichen, eine lange Datei seitenweise anzusehen. Traditionell wird hierfür **more** verwendet:

```
[matthias@Barbara Bash-Kurs]$ more --help
usage: more [-dflln] [+linenum | +/pattern] name1 name2 ...


[matthias@Barbara Bash-Kurs]$ -
```

Unter Linux dürfte unterdessen sein komfortabler Nachfolger **less** häufiger verwendet werden, bietet dieses Programm doch eine Fülle von Optionen – zum Beispiel zum Blättern oder Suchen im angezeigten Text –, die seine Verwendung weitaus komfortabler machen.

```
[matthias@Barbara Bash-Kurs]$ less --help

   SUMMARY OF COMMANDS

   Commands marked with * may be preceded by a number, N.
   Notes in parentheses indicate the behavior if N is given.

   h  H                 Display this help.
   q  :q  :Q  ZZ        Exit.

   e  ^E  j  ^N  CR  *  Forward  one line (or N lines).
   y  ^Y  k  ^K  ^P  *  Backward one line (or N lines).
   f  ^F  ^V  SPACE  *  Forward  one window (or N lines).
   b  ^B  ESC-v      *  Backward one window (or N lines).
   z                 *  Forward  one window (and set window to N).
   w                 *  Backward one window (and set window to N).
   d  ^D             *  Forward  one half-window (and set half-window to N).
   u  ^U             *  Backward one half-window (and set half-window to N).
   F                    Forward forever; like "tail -f".
   r  ^R  ^L         Repaint screen.
   R                 Repaint screen, discarding buffered input.
               ---------------------------------------------------
               Most flags may be changed either on the command line,
               or from within less by using the - command.

   -?            Display help (from command line).
   -a            Forward search skips current screen.
   -b [N]        Number of buffers.
   -B            Don't automatically allocate buffers for pipes.
```

```
    -c  -C        Repaint by scrolling/clearing.
    -d            Dumb terminal.
    -e  -E        Quit at end of file.
    -f            Force open non-regular files.
    -g            Don't highlight matches for previous search pattern.
    -G            Highlight ALL matches for previous search pattern.
    -h [N]        Backward scroll limit.
    -i            Ignore case in searches.
    -I            Ignore case in searches and in search patterns.
    -j [N]        Screen position of target lines.
    -k [file]     Use a lesskey file.
    -m  -M        Set prompt style.
    -n  -N        Use line numbers.
    -o [file]     Log file.
    -O [file]     Log file (unconditionally overwrite).
    -p [pattern]  Start at pattern (from command line).
    -P [prompt]   Define new prompt.
    -q  -Q        Quiet the terminal bell.
    -r            Output "raw" control characters.
    -s            Squeeze multiple blank lines.
    -S            Chop long lines.
    -t [tag]      Find a tag.
    -T [tagsfile] Use an alternate tags file.
    -u  -U        Change handling of backspaces.
    -V            Display the version number of "less".
    -w            Display ~ for lines after end-of-file.
    -x [N]        Set tab stops.
    -X            Don't use termcap init/deinit strings.
    -y [N]        Forward scroll limit.
    -z [N]        Set size of window.


[matthias@Barbara Bash-Kurs]$ _
```

Schon anhand dieser (hier aus Platzgründen stark gekürzten) Übersicht (die zugehörige *man-page* umfaßt 23 Seiten) ist zu erahnen, welch vielfältige Möglichkeiten dieses „Anzeige-Programm" bietet.

head und tail

Zuweilen kommt es vor, daß lediglich der Beginn oder das Ende einer Datei von Interesse ist. Diesem Zwecke dienen die Programme **head** (Datei-Beginn zeigen) und **tail** (Datei-Ende zeigen).

```
[matthias@Barbara Bash-Kurs]$ head --help Usage: head [OPTION]... [FILE]...
Print first 10 lines of each FILE to standard output.
With more than one FILE, precede each with a header giving the file name.
With no FILE, or when FILE is -, read standard input.

    -c, --bytes=SIZE  print first SIZE bytes
    -n, --lines=NUMBER   print first NUMBER lines instead of first 10
    -q, --quiet, --silent never print headers giving file names
    -v, --verbose always print headers giving file names
        --help display this help and exit
        --version  output version information and exit

SIZE may have a multiplier suffix: b for 512, k for 1K, m for 1 Meg.
If -VALUE is used as first OPTION, read -c VALUE when one of
multipliers bkm follows concatenated, else read -n VALUE.


[matthias@Barbara Bash-Kurs]$ tail --help
Usage: tail [OPTION]... [FILE]...
Print last 10 lines of each FILE to standard output.
With more than one FILE, precede each with a header giving the file name.
With no FILE, or when FILE is -, read standard input.

    -c, --bytes=N output the last N bytes
    -f, --follow  output appended data as the file grows
    -n, --lines=N output the last N lines, instead of last 10
    -q, --quiet, --silent never output headers giving file names
    -v, --verbose always output headers giving file names
        --help display this help and exit
        --version  output version information and exit

If the first character of N (the number of bytes or lines) is a `+',
print beginning with the Nth item from the start of each file, otherwise,
print the last N items in the file.  N may have a multiplier suffix:
b  for 512, k for 1024, m for 1048576 (1 Meg).  A first OPTION of -VALUE
or +VALUE is treated like -n VALUE or -n +VALUE unless VALUE has one of
the [bkm] suffix multipliers, in which case it is treated like -c VALUE
or -c +VALUE.
```

```
[matthias@Barbara Bash-Kurs]$ _
```

Insbesondere die **-f** Option von **tail** kann nützlich sein, wenn es darum geht, „wachsende" Dateien (wie z.B. Protokoll-Dateien, *logfiles* u.ä) zu beobachten. So sorgen etwa die folgenden Zeilen in der Datei **/etc/inittab** dafür, daß die angegebenen logfiles ständig auf verschiedenen Konsolen angezeigt werden:

```
[matthias@Barbara Bash-Kurs]$ grep 'tail' /etc/inittab
c6:345:respawn:nice -n 10 tail -n 50 -f /var/log/samba.log >>/dev/tty6
c7:345:respawn:nice -n 10 tail -n 50 -f /var/tmp/named.run >>/dev/tty7
c8:345:respawn:nice -n 10 tail -n 50 -f /var/log/ppp.log >>/dev/tty8
c9:345:respawn:nice -n 10 tail -n 50 -f /var/spool/smail/log/Debug.log >>/dev/tty9
cA:345:respawn:nice -n 10 tail -n 50 -f /var/spool/smail/log/logfile >>/dev/tty10
cB:345:respawn:nice -n 10 tail -n 50 -f /var/log/news/news.notice >>/dev/tty11


[matthias@Barbara Bash-Kurs]$ _
```

Übrigens kennt auch **less** eine solche **-f** Option, die durch **F** auch nachträglich aktiviert werden kann, wenn **less** bereits eine Datei anzeigt (**Ctrl-C** bricht diesen Modus dann wieder ab).

Auswerten von Dateien

Wenn es nicht ausreicht, die interessierenden Dateien einfach nur anzuschauen, oder wenn sie so umfangreich sind, daß man Schrierigkeiten hat, die interessanten Passagen zu finden bzw. auszuwerten, kann man Programme verwenden, deren Zweck es ist, definierte Passagen zu suchen und auszugeben. Diese „definierten Passagen" werden durch sog. Muster (*pattern*) angegeben. Derartiges kennen wir bereits von der Kommandozeile: das ***** beispielsweise dient als Muster für „alles". Das Arbeiten mit Mustern kann zu einer eigenen Kunstform werden, insbesondere dann, wenn die *pattern* durch sogenannte „reguläre Ausdrücke" (*regular expressions*, kurz: *regex*) dargestellt werden.

grep

Im letzten Beispiel wurde das Programm **grep** verwendet, um die gewünschten Zeilen aus einer Datei gewissermaßen „auszuschneiden". Eine Hilfe-Funktion des Programmes ist indes nicht vorhanden:

```
[matthias@Barbara Bash-Kurs]$ grep --help
grep: illegal option -- -
usage: grep [-[[AB] ]<num>] [-[CEFGVchilnqsvwx]] [-[ef]] <expr> [<files...>]

[matthias@Barbara Bash-Kurs]$ _
```

Für unsere Zwecke beschränken wir uns auf die Option **-i**, wodurch bei Bedarf die Beachtung der Groß-/Kleinschreibung ausgeschaltet werden kann, und als *pattern* auf ganz normale (Such-)Wörter.

awk

Sehr viel weitergehende Möglichkeiten als **grep** besitzt **awk**: es kann nicht nur einzelne Zeilen lesen, sondern auch deren Bestandteile (*fields*) separieren und beispielsweise zum Rechnen benutzen. Auch hier ist die eingebaute Hilfe nicht sonderlich ergiebig:

```
[matthias@Barbara Bash-Kurs]$ awk --help
Usage:  awk [POSIX or GNU style options] -f progfile [--] file ...
awk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:          GNU long options:
    -f progfile             --file=progfile
    -F fs                   --field-separator=fs
    -v var=val              --assign=var=val
    -m[fr]=val
    -W compat               --compat
    -W copyleft             --copyleft
    -W copyright            --copyright
    -W help                 --help
    -W lint                 --lint
    -W posix                --posix
    -W source=program-text  --source=program-text
```

```
    -W usage                --usage
    -W version              --version

[matthias@Barbara Bash-Kurs]$ _
```

Genaugenommen ist **awk** eine eigene Programmiersprache, in der sich eigene *akw-scripts* erstellen lassen. Im Rahmen dieses Seminares indes beschränken wir uns auf die einfachsten Möglichkeiten, wie etwa das Extrahieren eines bestimmten Teiles einer Zeile.

Verändern von Dateien

Wenn weder das schlichte Anschauen – und sei's mit Unterstützung eines *pagers* – ausreicht, und auch das „Ausschneiden" relevanter Passagen aus einer Datei – mit **grep** oder **awk** – nicht zum gewünschten Ziele führt, könnte es daran liegen, daß die Daten erst noch „aufbereitet" werden müssen. Zwei häufig benötigte „Aufbereitungen" sind zum einen das Sortieren einer Datei nach bestimmten Kriterien (**sort**) und zum anderen das Ersetzen von bestimmten Zeichenfolgen durch andere (**sed**).

sort

Im einfachsten Falle wird eine Datei einfach anhand ihrer Zeilen sortiert. Der **sort**-Befehl bietet indes auch Möglichkeiten, die Art der Sortierung (auf - oder absteigend) und die Sortier-Kriterien festzulegen:

```
[matthias@Barbara Bash-Kurs]$ sort --help
Usage: sort [OPTION]... [FILE]...
Write sorted concatenation of all FILE(s) to standard output.

    +POS1 [-POS2] start a key at POS1, end it before POS2
    -M                compare (unknown) < `JAN' < ... < `DEC', imply -b
    -T DIRECT         use DIRECT for temporary files, not $TMPDIR or /tmp
    -b                ignore leading blanks in sort fields or keys
    -c                check if given files already sorted, do not sort
    -d                consider only [a-zA-Z0-9 ] characters in keys
    -f                fold lower case to upper case characters in keys
    -i                consider only [\040-\0176] characters in keys
    -k POS1[,POS2]    same as +POS1 [-POS2], but all positions counted from 1
    -m                merge already sorted files, do not sort
    -n                compare according to string numerical value, imply -b
    -o FILE           write result on FILE instead of standard output
    -r                reverse the result of comparisons
    -s                stabilize sort by disabling last resort comparison
    -t SEP            use SEParator instead of non- to whitespace transition
    -u                with -c, check for strict ordering
    -u                with -m, only output the first of an equal sequence
       --help         display this help and exit
       --version      output version information and exit

POS is F[.C][OPTS], where F is the field number and C the character
position in the field, both counted from zero.  OPTS is made up of one
or more of Mbdfinr, this effectively disable global -Mbdfinr settings
for that key.  If no key given, use the entire line as key.  With no
FILE, or when FILE is -, read standard input.

[matthias@Barbara Bash-Kurs]$ _
```

sed

Aus **sed** ist im Grunde eine eigene Programmiersprache und entsprechend mager (denn dazu gibt's eigene Literatur) ist denn auch der Hilfe-Text:

```
[matthias@Barbara Bash-Kurs]$ sed --help
Usage: sed [-nV] [--quiet] [--silent] [--version] [-e script]
    [-f script-file] [--expression=script] [--file=script-file] [file...]

[matthias@Barbara Bash-Kurs]$ _
```

Als Beispiel für ein *Shell-script*, daß u.a. **sed** für seine Zwecke verwendet, folgen hier Auszüge aus **process.logs**. Dieses Script wird als „Mail-Filter" eingesetzt und wandelt die eingehenden Emails in HTML-Seiten um, so daß sie im Rahmen einer Web-Präsentation angesehen werden können.

```
[matthias@Barbara Bash-Kurs]$ cat process.logs
#! /bin/bash
           #
           # [ Kode gekuerzt ]
           #

# \<!-- HTML-Kommentar-Start
#
declare -r HTMLescape1='-e s/\\</kLeInErAls/g -e s/\\>/gRoEsSeRaLs/g' #
declare -r HTMLescape2='-e s/kLeInErAls/</g -e s/gRoEsSeRaLs/>/g'
#
declare -r HTMLfilter='-e s/&/aMpErSaNdamp;/g -e s/</aMpErSaNdlt;/g
-e s/>/aMpErSaNdgt;/g -e s/ä/aMpErSaNdauml;/g -e s/Ä/aMpErSaNdAuml;/g
-e s/ö/aMpErSaNdouml;/g -e s/Ö/aMpErSaNdOuml;/g -e s/ü/aMpErSaNduuml;/g
-e s/Ü/aMpErSaNdUuml;/g -e s/ß/aMpErSaNdszlig;/g -e s/aMpErSaNd/\&/g'
#
# HTML-Kommentar-Ende --\>

#-----------------------------------------------------------------------------
#
function SplitMail() {
#            Mail in Head und Body zerlegen
#
    ${SED} -e '/^$/q' ${InFile} > ${MailHead}
    ${SED} -e '1,/^$/d' ${InFile} > ${MailBody}
    [ -s  ${MailBody} ] || return 1 # leere Mail signalisieren
    return 0
}   # SplitMail

#-----------------------------------------------------------------------------
#
function ProcessFile() {
#            die Mail ins Logfile-Verzeichnis legen
           #
           # [ Kode gekuerzt ]
           #
    #
    ${SED} ${HTMLescape1} ${HTMLfilter} ${HTMLescape2} ${MailBody} >>${Dateiname}
    echo -e "
</PRE>\n<HR>\n<P> Zur&uuml;ck zur <A HREF=\"index.html\">&Uuml;bersicht</A> </P>
</BODY>\n</HTML>">>${Dateiname}
    Title="${Dateiname%/*}"
    [ -x ${NeueIndizes} ] && ${NeueIndizes} "d" ${Title} &
    return 0
}   # ProcessFile

           #
           # [ Kode gekuerzt ]
           #

#   _EoF_

[matthias@Barbara Bash-Kurs]$ _
```

(!) Infolge der Grund-„Philosophie" Unix-basierter Systeme müssen häufig sogenannte „externe" Befehle (Programme) verwendet werden, um Dateien sowie Verzeichnisse zu verwalten oder Daten auswerten zu können. Die Teilnehmerinnen lernen die wichtigsten (d.h. meistverwen deten) Dienstprogramme (*tools*) und ihre jeweiligen Aufruf-Parameter kennen.

(!) Programme wie **`awk`**, **`grep`** oder **`sed`** beherrschen neben dem auch der Shell bekannten „pattern-matching" auch sog. reguläre Ausdrücke („regular expressions", kurz *regex*). Die Teilnehmerinnen lernen diese Möglichkeit kennen, Zeichenfolgen abstrakt auszudrücken, um un bekannte Datenbestände zu durchsuchen bzw. auszuwerten.

„BaSH“ als Programmier-Plattform

[Donnerstag: Vormittag]

Wie andere (Programmier-)Sprachen auch verfügt die *BaSH* nicht nur über einen gewissen „Wortschatz“ (die eingebauten Befehle, *builtins*, und *reserved words*), sondern auch eine Art Syntax und Semantik, die vorgeben, welche Zeichen an welchen Stellen mit welcher Bedeutung – bzw. mit welchen Auswirkungen – verwendet werden können.

positionale Parameter

A *positional parameter* is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the set builtin command. Positional parameters may not be assigned to with assignment statements. The positional parameters are temporarily replaced when a shell **`function`** is executed. When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces.

spezielle Parameter

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

`*`

Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the **`IFS`** special variable. That is, **`"$*"`** is equivalent to **`"$1c$2c..."`**, where **`c`** is the first character of the value of the **`IFS`** variable. If **`IFS`** is null or unset, the parameters are separated by spaces.

`@`

Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands as a separate word. That is, **`"$@"`** is equivalent to **`"$1 $2 ..."`** When there are no positional parameters, **`"$@"`** and **`$@`** expand to nothing (i.e., they are removed).

`#`

Expands to the number of positional parameters in decimal.

`?`

Expands to the status of the most recently executed foreground pipeline.

`-`

Expands to the current option flags as specified upon invocation, by the **`set`** builtin command, or those set by the shell itself (such as the **`-i`** flag).

`$`

Expands to the process ID of the shell. In a **`()`** subshell, it expands to the process ID of the current shell, not the subshell.

`!`

Expands to the process ID of the most recently executed background (asynchronous) command.

`0`

Expands to the name of the shell or shell script. This is set at shell initialization. If bash is invoked with a file of commands, **`$0`** is set to the name of that file. If bash is started with the **`-c`** option, then **`$0`** is set to the first argument after the string to be executed, if one is present. Otherwise, it is set to the pathname used to invoke bash, as given by argument zero.

_

Expands to the last argument to the previous command, after expansion. Also set to the full pathname of each command executed and placed in the environment exported to that command.

```
[matthias@Barbara Bash-Kurs]$ params.sh 1 2 3 4 5 6 7 8 9 10 11

    die positionalen Parameter dieses Aufrufs:

    "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11"

    die speziellen Parameter dieses Aufrufs:

    *: "1+2+3+4+5+6+7+8+9+10+11"
    @: "1 2 3 4 5 6 7 8 9 10 11"
    #: "11"
    ?: "0"
    -: ""
    $: "6970"
    !: ""
    0: "./params.sh"
    _: ""


[matthias@Barbara Bash-Kurs]$ cat params.sh
#!/bin/bash
declare -r cgIFS="${IFS}"              # IFS sichern
IFS="+${IFS}"                          # ein '+' in IFS voransetzen
declare -r cgStern="$*"                # hier wird das '+' wirksam
declare -r cgAffe="$@"                 # hier aber nicht
declare -r igZaun="$#"
declare -r igFrage="$?"
declare -r cgMinus="$-"
declare -r igDollar="$$"
declare -r igAusruf="$!"
declare -r cgNull="$0"
declare -r cgUnter="$_"
IFS="${cgIFS}"                         # IFS restaurieren
cgP=""

echo -en "\n\tdie positionalen Parameter dieses Aufrufs:\n\n\t"

for cgP in $* ; do                     # die uebergebenen Parameter durchgehen
    echo -en "\"${cgP}\" "             # und einzeln anzeigen
done

cat <<_EoT_


    die speziellen Parameter dieses Aufrufs:
    *: "${cgStern}"
    @: "${cgAffe}"
    #: "${igZaun}"
    ?: "${igFrage}"
    -: "${cgMinus}"
    \$: "${igDollar}"
    !: "${igAusruf}"
    0: "${cgNull}"
    _: "${cgUnter}"
_EoT_

#_EoF_


[matthias@Barbara Bash-Kurs]$ _
```

(Umgebungs-)Variablen

Schon die *BaSH* selber verwendet eine Reihe von Variablen, hinzu kommen – zumindest in Scripts – noch jene, die für die je eigenen Zwecke benötigt werden. Standardmäßig haben sie alle einen *globalen* Gültigkeitsbereich, d.h. alle Teile eines Scripts können auf sie lesend wie schreibend zu greifen. Die *shell* bietet v.a. zwei Möglichkeiten, dies ein wenig einzuschränken, d.h. sicherer zu machen: die **`declare`** und **`local`** *builtins*. So kann eine Variable, die zwar global lesbar, nicht jedoch veränderbar sein soll, durch

`declare -r VarName="konstanter Inhalt"`

zur einer R/O (readonly) Konstanten gemacht werden. Ähnlich können Variablen, die lediglich innerhalb eines Unter-Programmes (**`function`**) benötigt werden, durch

`local lokaleVar="irgendein Init-Wert"`
auf einen Funktions-lokalen Gültigkeitsbereich eingeschränkt werden. Schon durch diese beiden einfachen Maßnahmen läßt sich die Gefahr unbeabsichtigter Nebeneffekte in einem Skript drastisch reduzieren. – Die schlechte Nachricht: Damit sind die Möglichkeiten einer – sagen wir: – automatischen Fehlervermeidung auch schon ausgeschöpft. Alle anderen versehentlichen Fehler – wie etwa der Versuch, mit einer Variablen, die eine Zeichenfolge enthält, zu rechnen –, resultieren in einer mehr oder (eher) minder aussagekräftigen Fehlermeldung und dem Abbruch des Scripts. Es obliegt mithin der je eigenen Sorgfalt, die Zahl möglicher (Programmier-)Fehler zu minimieren, beispielsweise mithilfe einer konsistenten Variablen-Benamsung (wobei der Name einer Variablen gleich auf den Typ ihrer Werte hinweist, z.B. **`cgName`** bzw. **`clName`** für globale bzw. lokale Zeichenfolgen (*character*) oder **`igName`** bzw. **`ilName`** für globale bzw. lokale Zahlenwerte (*integer*).

Bevor die Variablen-Werte indes tatsächlich für die „eigentliche" Arbeit des Scripts verwendet werden, unterzieht sie die *BaSH* einer Reihe von Manipulationen, den sog. *Expansionen*. Und genau dies ist die Stelle, ist der Zeitpunkt, an der und zu dem die Fähigkeiten der *shell* die Intentionen der Programmiererinnen kräftig unterstützen können.

Expansionen

Expansion is performed on the command line after it has been split into words. There are seven kinds of expansion performed: *brace expansion*, *tilde expansion*, *parameter and variable expansion*, *command substitution*, *arithmetic expansion*, *word splitting*, and *pathname expansion*.

The order of expansions is: *brace expansion*, *tilde expansion*, *parameter*, *variable*, *command* and *arithmetic substitution* (done in a left-to-right fashion), *word splitting*, and *pathname expansion*.

On systems that can support it, there is an additional expansion available: *process substitution*.

Only *brace expansion*, *word splitting*, and *pathname expansion* can change the number of words of the expansion; other expansions expand a single word to a single word. The single exception to this is the expansion of **`"$@"`** as explained above.

Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *pathname expansion*, but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional *preamble*, followed by a series of comma-separated strings between a pair of braces, followed by an optional *postamble*. The *preamble* is prepended to each string contained within the braces, and the *postamble* is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved.

Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. Bash does not apply any syntactic interpretation to the context of the expansion or the text between the braces.

A correctly-formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma. Any incorrectly formed brace expansion is left unchanged. This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example:

`mkdir /usr/local/src/bash/{old,new,dist,bugs}` or

`chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}`

Brace expansion introduces a slight incompatibility with traditional versions of **`sh`**, the Bourne shell. **`sh`** does not treat opening or closing braces specially when they appear as part of a word, and preserves them in the output. Bash removes braces from words as a consequence of brace expansion. For example, a word entered to **`sh`** as **`file{1,2}`** appears identically in the output. The same word is output as **`file1 file2`** after expansion by *bash*. If strict compatibility with **`sh`** is desired, start *bash* with the **`-nobraceexpansion`** flag or disable brace expansion with the **`+o braceexpand`** option to the **`set`** command.

```
[matthias@Barbara Bash-Kurs]$ echo a{b,c,d}e
abe ace ade
```

```
[matthias@Barbara Bash-Kurs]$ /bin/ls -l /usr/local/{bin,lib,man,sbin}
/usr/local/bin:
# (Eintraege gekuerzt)

/usr/local/lib:
# (Eintraege gekuerzt)

/usr/local/man:
# (Eintraege gekuerzt)

/usr/local/sbin:
# (Eintraege gekuerzt)


[matthias@Barbara Bash-Kurs]$ _
```

Tilde Expansion

If a word begins with a tilde character (»**~**«), all of the characters preceding the first slash (or all characters, if there is no slash) are treated as a possible login name. If this login name is the null string, the tilde is replaced with the value of the parameter **HOME**. If **HOME** is unset, the home directory of the user executing the shell is substituted instead.

If a »**+**« follows the tilde, the value of **PWD** replaces the tilde and »**+**«. If a »**-**« follows, the value of **OLDPWD** is substituted. If the value following the tilde is a valid login name, the tilde and login name are replaced with the home directory associated with that name. If the name is invalid, or the tilde expansion fails, the word is unchanged.

Each variable assignment is checked for unquoted instances of tildes following a **:** or **=**. In these cases, tilde substitution is also performed. Consequently, one may use pathnames with tildes in assignments to **PATH**, **MAILPATH**, and **CDPATH**, and the shell assigns the expanded value.

```
[matthias@Barbara Bash-Kurs]$ echo ~
/home/staff/matthias


[matthias@Barbara Bash-Kurs]$ echo ~+
/home/staff/matthias/projects/Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ~-
/home/staff/matthias


[matthias@Barbara Bash-Kurs]$ _
```

Parameter Expansion

The »**$**« character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

${parameter}
: The value of **parameter** is substituted. The braces are required when parameter is a *positional parameter* with more than one digit, or when parameter is followed by a character which is not to be interpreted as part of its name.

In each of the cases below, **word** is subject to *tilde expansion*, *parameter expansion*, *command substitution*, and *arithmetic expansion*. Bash tests for a parameter that is unset or null; omitting the colon results in a test only for a parameter that is unset.

${parameter:-word}
: Use Default Values. If **parameter** is unset or null, the expansion of **word** is substituted. Otherwise, the value of **parameter** is substituted.

${parameter:=word}
: Assign Default Values. If **parameter** is unset or null, the expansion of **word** is assigned to **para-**

meter. The value of **parameter** is then substituted. *Positional parameters* and *special parameters* may not be assigned to in this way.

${parameter:?word}
Display Error if Null or Unset. If **parameter** is null or unset, the expansion of **word** (or a message to that effect if **word** is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of **parameter** is substituted.

${parameter:+word}
Use Alternate Value. If **parameter** is null or unset, nothing is substituted, otherwise the expansion of **word** is substituted.

${#parameter}
The length in characters of the value of **parameter** is substituted. If parameter is ***** or **@**, the length substituted is the length of ***** expanded within double quotes.

${parameter#word}
${parameter##word}
The **word** is expanded to produce a pattern just as in *pathname expansion*. If the pattern matches the beginning of the value of **parameter**, then the expansion is the value of **parameter** with the shortest matching pattern deleted (the »**#**« case) or the longest matching pattern deleted (the »**##**« case).

${parameter%word}
${parameter%%word}
The **word** is expanded to produce a pattern just as in **pathname expansion**. If the pattern matches a trailing portion of the value of **parameter**, then the expansion is the value of **parameter** with the shortest matching pattern deleted (the »**%**« case) or the longest matching pattern deleted (the »**%%**« case).

```
[matthias@Barbara Bash-Kurs]$ echo ${GibbetNich}



[matthias@Barbara Bash-Kurs]$ echo ${PWD}
/home/staff/matthias/projects/Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ${GibbetNich:-"neuer Default-Wert"}
neuer Default-Wert


[matthias@Barbara Bash-Kurs]$ echo ${PWD:-"neuer Default-Wert"}
/home/staff/matthias/projects/Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ${GibbetNich:="noch n Default"}
noch n Default


[matthias@Barbara Bash-Kurs]$ echo ${PWD:="noch n Default"}
/home/staff/matthias/projects/Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ${GibbetNich:?"Variable gibbet nich"}
noch n Default


[matthias@Barbara Bash-Kurs]$ echo ${GibbetAuchnich:?"Variable gibbet nich"}
bash: GibbetAuchnich: Variable gibbet nich


[matthias@Barbara Bash-Kurs]$ echo ${PWD:?"Variable gibbet nich"}
/home/staff/matthias/projects/Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ${GibbetNich:+"Variable gibbet nich"}
Variable gibbet nich


[matthias@Barbara Bash-Kurs]$ echo ${GibbetNich}
noch n Default


[matthias@Barbara Bash-Kurs]$ echo ${#GibbetNich}
14


[matthias@Barbara Bash-Kurs]$ echo${#PWD}
```

```
39


[matthias@Barbara Bash-Kurs]$ echo ${PWD#*/}
home/staff/matthias/projects/Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ${PWD##*/}
Bash-Kurs


[matthias@Barbara Bash-Kurs]$ echo ${PWD%/*}
/home/staff/matthias/projects


[matthias@Barbara Bash-Kurs]$ echo ${PWD%%/*}



[matthias@Barbara Bash-Kurs]$ echo ${PWD%%projects/*}
/home/staff/matthias/


[matthias@Barbara Bash-Kurs]$ _
```

Command Substitution

Command substitution allows the output of a command to replace the command name. There are two forms:

$(command)

or

‘command‘

Bash performs the expansion by executing **command** and replacing the command substitution with the standard output of the **command**, with any trailing newlines deleted. When the old-style backquote form of substitution is used, backslash retains its literal meaning except when followed by **$**, **‘**, or ****. When using the **$(command)** form, all characters between the parentheses make up the **command**; none are treated specially.

Command substitutions may be nested. To nest when using the old form, escape the inner back-quotes with backslashes. If the substitution appears within double quotes, word splitting and pathname expansion are not performed on the results.

```
[matthias@Barbara Bash-Kurs]$ echo $(date)
Fri Aug 20 01:23:47 MET DST 1999


[matthias@Barbara Bash-Kurs]$ echo `date`
Fri Aug 20 01:23:57 MET DST 1999


[matthias@Barbara Bash-Kurs]$ _
```

Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. There are two formats for *arithmetic expansion*:

$[expression]

$((expression))

The **expression** is treated as if it were within double quotes, but a double quote inside the braces or parentheses is not treated specially. All tokens in the **expression** undergo *parameter expansion*, *command substitution*, and *quote removal*. *Arithmetic substitutions* may be nested. If **expression** is invalid, bash prints a message indicating failure and no substitution occurs.

```
[matthias@Barbara Bash-Kurs]$ echo $[ ${SHVL} + 1 ]
1


[matthias@Barbara Bash-Kurs]$ echo $[ $[ $? + 1 ] * 7 ]
7
```

```
[matthias@Barbara Bash-Kurs]$ echo $[ ${GibbetNich} + 3 ]
bash: noch n Default + 3 :
syntax error in expression (remainder of expression is "n Default + 3 ")


[matthias@Barbara Bash-Kurs]$ _
```

Word Splitting

The shell scans the results of *parameter expansion*, *command substitution*, and *arithmetic expansion* that did not occur within double quotes for *word splitting*.

The shell treats each character of **IFS** as a delimiter, and splits the results of the other expansions into words on these characters. If the value of **IFS** is exactly **<space><tab><newline>**, the default, then any sequence of **IFS** characters serves to delimit words. If **IFS** has a value other than the default, then sequences of the whitespace characters space and tab are ignored at the beginning and end of the word, as long as the whitespace character is in the value of **IFS** (an **IFS** whitespace character). Any character in **IFS** that is not **IFS** whitespace, along with any adjacent **IFS** whitespace characters, delimits a field.

A sequence of **IFS** whitespace characters is also treated as a delimiter. If the value of **IFS** is null, no word splitting occurs. **IFS** cannot be unset.

Explicit null arguments (**""** or **''**) are retained. Implicit null arguments, resulting from the expansion of parameters that have no values, are removed.

Note that if no expansion occurs, no splitting is performed.

Pathname Expansion

After *word splitting*, unless the **-f** option has been set, bash scans each word for the characters *****, **?**, and **[**. If one of these characters appears, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of pathnames matching the pattern. If no matching pathnames are found, and the shell variable **allow_null_glob_expansion** is unset, the word is left unchanged. If the variable is set, and no matches are found, the word is removed. When a pattern is used for pathname generation, the character »**.**« at the start of a name or immediately following a slash must be matched explicitly, unless the shell variable **glob_dot_filenames** is set. The slash character must always be matched explicitly. In other cases, the »**.**« character is not treated specially.

The special pattern characters have the following meanings:

*****

Matches any string, including the null string.

?

Matches any single character.

[...]

Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a range; any character lexically between those two characters, inclusive, is matched. If the first character following the **[** is a **!** or a **^** then any character not enclosed is matched. A **-** or **]** may be matched by including it as the first or last character in the set.

```
[matthias@Barbara Bash-Kurs]$ /bin/ls -l
total 3
drwxr-xr-x 2 matthias matthias      1024 Aug 19 18:27 Scripts
-rw-r--r-- 1 matthias matthias         0 Aug 20 19:55 a
-rw-rw-r-- 1 matthias matthias       518 Aug 19 20:00 hist.log
-rwxr-xr-x 1 matthias matthias       539 Aug 19 17:32 params.sh


[matthias@Barbara Bash-Kurs]$ echo *
Scripts a hist.log params.sh


[matthias@Barbara Bash-Kurs]$ echo ?
a


[matthias@Barbara Bash-Kurs]$ echo ?[ia]*
```

```
hist.log params.sh


[matthias@Barbara Bash-Kurs]$ echo ?[^ia]*
Scripts


[matthias@Barbara Bash-Kurs]$ _
```

(!) Neben den Optionen während der unmittelbaren Arbeit in der Kommando-Zeile bietet die Erstellung sog. *shell-scripts* weitere Möglichkeiten, aufeinanderfolgende Arbeitsschritte zu orga nisieren und zu automatisieren. Die Teilnehmerinnen lernen die Grundlagen der eingebauten „Pro grammiersprache“ kennen.

Programm-Strukturen

[Donnerstag: Nachmittag]

sequentielle Abarbeitung

Die Grundstruktur eines jeden Programmes ist die *Anweisungs-Sequenz*, d.h. eine Folge von Anweisungen, die nacheinander –also: Zeile für Zeile, von links nach rechts –abgearbeitet werden. Insofern unterscheidet sich der Quell-Text eines Programmes nicht von –sagen wir: –einem Aufsatz, der Wort für Wort, Zeile für Zeile, Seite um Seite gelesen wird.

Im Unterschied zu Aufsätzen oder Büchern kommen bei Programm-Texten jedoch noch sogenannte *Kontroll-Strukturen* hinzu. Sie ermöglichen es, nach der Prüfung zuvor definierter Bedingungen einzelne Text-Bereiche zu „überspringen" (bei der Programm-Ausführung zu ignorieren) oder sie umgekehrt nur dann auszuführen, wenn eine beim Programmieren defi nierte Bedingung während der Programm-Ausführung tatsächlich gegeben ist.

bedingte Verzweigungen

Die **`if`**-Prüfung ist gewissermaßen die Grundform bedingter Verzweigungen. Hier wird eine definierte Bedingung geprüft und in Abhängigkeit vom Ergebnis dieses Tests der darauf folgende Programm-Kode entweder ausgeführt oder eben nicht. Eine **`case`**-Anweisung ist insofern lediglich eine Sonderform einer Gruppe von mehreren **`if/else`**-Folgen, sie bietet mithin nichts prinzipiell Neues, sondern dient –interne, Programmiersprachen-spezifische Optimierungen einmal außer acht ge lassen –vor allem einer leichteren Les- und Wartbarkeit des Programm-Kodes.

if / elif / else

`if` *`list`* `then` *`list`* `[elif` *`list`* `then` *`list`*`] ... [else` *`list`*`] fi`

The **`if` *`list`*** is executed. If its exit status is zero, the **`then` *`list`*** is executed. Otherwise, each elif list is executed in turn, and if its exit status is zero, the corresponding **`then` *`list`*** is executed and the command completes. Otherwise, the **`else` *`list`*** is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true.

```
[matthias@Barbara Bash-Kurs]$ iftest.sh
/iftest.sh: 1: bitte mit einem Zahlenwert aufrufen!


[matthias@Barbara Bash-Kurs]$ !! 0
iftest.sh  0
0 ist 0 (Null)


[matthias@Barbara Bash-Kurs]$ ^0^1
iftest.sh  1
1 ist ein 8-Bit-Wert


[matthias@Barbara Bash-Kurs]$ ^1^12
iftest.sh  12
12 ist ein 8-Bit-Wert


[matthias@Barbara Bash-Kurs]$ ^2^23
iftest.sh  123
123 ist ein 8-Bit-Wert


[matthias@Barbara Bash-Kurs]$ ^3^34
iftest.sh  1234
1234 ist ein 16-Bit-Wert


[matthias@Barbara Bash-Kurs]$ ^4^45
iftest.sh  12345
12345 ist ein 16-Bit-Wert
```

```
[matthias@Barbara Bash-Kurs]$ ^5^56
iftest.sh  123456
123456 ist ein positiver 32-Bit-Wert


[matthias@Barbara Bash-Kurs]$ ^1^-1
iftest.sh  -123456
-123456 ist ein negativer Wert


[matthias@Barbara Bash-Kurs]$ cat iftest.sh
#!/bin/bash

declare -i igN=${1:?"bitte mit einem Zahlenwert aufrufen!"}

if [ ${igN} -gt 65535 ] ; then
    echo "${igN} ist ein positiver 32-Bit-Wert"
elif [ ${igN} -gt 255 ] ; then
    echo "${igN} ist ein 16-Bit-Wert"
elif [ ${igN} -gt 0 ] ; then
    echo "${igN} ist ein 8-Bit-Wert"
elif [ ${igN} -eq 0 ] ; then
    echo "${igN} ist 0 (Null)"
else
    echo "${igN} ist ein negativer Wert"
fi
#_EOF_


[matthias@Barbara Bash-Kurs]$ _
```

case

`case word in [pattern [| pattern] ... ) list;; ] ... esac`

A **`case`** command first expands **`word`**, and tries to match it against each **`pattern`** in turn, using the same matching rules as for pathname expansion. When a match is found, the corresponding **`list`** is executed. After the first match, no subsequent matches are attempted. The exit status is zero if no patterns are matches. Otherwise, it is the exit status of the last command executed in **`list`**.

```
[matthias@Barbara Bash-Kurs]$ Check.It
    Das Script muss ueber Sym-Links aufgerufen werden:
    Check.{active|history|newsfeeds}
    Diese Sym-Links werden jetzt angelegt!

-rwxrwx--- 1 matthias matthias 520 Aug 20 01:01 ./Check.It
lrwxrwxrwx 1 matthias matthias  10 Aug 20 01:02 ./Check.active -> ./Check.It
lrwxrwxrwx 1 matthias matthias  10 Aug 20 01:02 ./Check.history -> ./Check.It
lrwxrwxrwx 1 matthias matthias  10 Aug 20 01:02 ./Check.newsfeeds -> ./Check.It


[matthias@Barbara Bash-Kurs]$ Check.active
puefe active-Datei


[matthias@Barbara Bash-Kurs]$ Check.history
puefe history-Datei


[matthias@Barbara Bash-Kurs]$ Check.newsfeeds
puefe newsfeeds-Datei


[matthias@Barbara Bash-Kurs]$ ln -s Check.It Check.WasAnderes


[matthias@Barbara Bash-Kurs]$ Check.WasAnderes
unbekannter Aufruf: WasAnderes


[matthias@Barbara Bash-Kurs]$ cat Check.It
#!/bin/bash

Me="${0##*/}"                              # Pfadnamen entfernen
Who="${Me#*.}"                             # Dateinamen entfernen, Endung bleibt

if [ "${Who}" = "It" ] ; then              # mit Original-Endung aufgerufen?
    echo "
```

```
Das Script muss ueber Sym-Links aufgerufen werden:
${Me%%.*}.{active|history|newsfeeds}
Diese Sym-Links werden jetzt angelegt!
    "
    Who=${0%/*}
    for F in active history newsfeeds ; do
        ln -sf ${0} ${Who}/${Me%.It}.${F}   # Sym-Links erzeugen
    done
    exec ls -la ${Who}/${Me%.It}.*          # Links anzeigen
fi
#
# allgemeiner Kode
#
case ${Who} in                              # mit welcher Endung aufgerufen?
    active)
        echo "puefe active-Datei"
        #
        # Spezielles fuer active-Datei
        ;;
    newsfeeds)
        echo "puefe newsfeeds-Datei"
        #
        # Spezielles fuer newsfeeds-Datei
        ;;
    history)
        echo "puefe history-Datei"
        #
        # Spezielles fuer history-Datei
        ;;
    *)
        echo "unbekannter Aufruf: ${Who}"
        # Script abbrechen:
        exit 1
        ;;
esac
#
# weiter allgemeiner Kode
#

#_EoF_


[matthias@Barbara Bash-Kurs]$ _
```

Schleifen

Schleifen sind Programm-Konstrukte, die immer dann Verwendung finden, wenn eine bestimmte Folge von Anweisungen mehrfach abgearbeitet werden soll. Dabei gibt es mehrere Arten von Schleifen, die sich darin unterscheiden, welche Art von Test vorgenommen wird, um ihren Beginn und ihr Ende zu bestimmen.

Der erste – weil: einfachste – Fall ist der einer Schleife, bei der zum Zeitpunkt der Programmierung bereits bekannt ist, wie oft sie durchlaufen werden soll, d.h. es gibt einen festen Anfangs-Wert, mit dem die Schleife gestartet wird, und einen festen End-Wert, mit dem sie beendet wird. Dies ist die **`for`**-Schleife („von 1 bis 10 mach irgendwas“).

Den zweiten Fall haben wir in einer Situation, in der nur eine möglicherweise zutreffende *Bedingung* bekannt ist, unter der die fraglichen Anweisungen auszuführen sind. Dies ist die **`while`**-Schleife („so lange Bedingung erfüllt mach irgendwas“). Dies ähnelt der **`if`**-Verzweigung, mit dem Unterschied, daß die Anweisungen nach der **`while`**-Anweisung nicht nur einmal ausgeführt werden, sondern solange, wie die fragliche Bedingung gegeben ist/bleibt.

Der dritte Fall ist dem zweiten verwandt, dreht jedoch die Prüfungs-Logik gewissermaßen um: führe bestimmte Anweisungen aus, *bis* eine Bedingung erfüllt ist. Das ist die **`repeat-until`**-Schleife („mach irgendwas bis Bedingung erfüllt“).

Während also eine **`for`**-Schleife stets eine vorgegebene Anzahl von Malen durchlaufen wird, wird eine **`while`**-Schleife möglichweise *gar nicht* abgearbeitet (wenn nämlich ihre Eingangs-Bedingung nicht zutrifft), während eine **`repeat-until`**-Schleife mindestens *einmal* durchlaufen wird.

Insbesondere bei der letzten beiden Schleifenformen ist es sehr wichtig, beim Programmieren sicherzustellen, daß die ex - oder impliziten Abbruch-Bedingungen auch tatsächlich erreicht werden, andernfalls hat man eine den berüchtigten End los-Schleifen, die sich nur noch durch den Abbruch des ganzen Programmes (**`kill -9 PID`**) beenden lassen.

for ...

`for name [in word; ] do list; done`

The list of words following **`in`** is expanded, generating a list of items. The variable **`name`** is set to each element of this list in turn, and **`list`** is executed each time. If the **`in word`** is omitted, the **`for`** command executes **`list`** once for each positional parameter that is set.

```
[matthias@Barbara Bash-Kurs]$ fortest.sh
Datei: Check.It
Link: Check.WasAnderes
Link: Check.active
Link: Check.history
Link: Check.newsfeeds
Verzeichnis: Scripts
Datei: casetest.sh
Datei: fortest.sh
Datei: hist.log
Datei: iftest.sh
Datei: params.sh


[matthias@Barbara Bash-Kurs]$ cat fortest.sh
#!/bin/bash

declare cgF=""                      # Laufvariable deklarieren

for cgF in * ; do
    if [ -L ${cgF} ] ; then
        echo Link: ${cgF}
    elif [ -d ${cgF} ] ; then
        echo Verzeichnis: ${cgF}
    elif [ -s ${cgF} ] ; then
        echo Datei: ${cgF}
    else
        echo unbekannt: ${cgF}
    fi
done

# _EoF_


[matthias@Barbara Bash-Kurs]$ _
```

while ... / ... until

`while list do list done`

`until list do list done`

The **`while`** command continuously executes the **`do list`** as long as the last command in **`list`** returns an exit status of zero. The **`until`** command is identical to the **`while`** command, except that the test is negated; the **`do list`** is executed as long as the last command in **`list`** returns a non-zero exit status. The exit status of the **`while`** and **`until`** commands is the exit status of the last **`do list`** command executed, or zero if none was executed.

```
[matthias@Barbara Bash-Kurs]$ whiletest.sh
0
1 2 3 4 5 6 7 8 9 10
10
9 8 7 6 5 4 3 2 1 0
0


[matthias@Barbara Bash-Kurs]$ cat whiletest.sh
#!/bin/bash

declare -i igN=0                # Variable 'igN' mit '0' initialisieren

echo ${igN}                     # Start-Wert anzeigen

while [ ${igN} -lt 10 ] ; do    # ist 'igN' kleiner '10'?
    let igN+=1                  # Ja: 'igN' um eins erhoehen
    echo -n "${igN} "           # ... und anzeigen
done

echo -e "\n${igN}"              # Zwischenergebnis anzeigen

until [ ${igN} -eq 0 ] ; do     # ist 'igN' gleich '0'?
```

```
    let igN-=1                  # Nein: 'igN' ums eins erniedrigen
    echo -n "${igN} "           # ... und anzeigen
done

echo -e "\n${igN}"              # End-Wert anzeigen

#_EoF_


[matthias@Barbara Bash-Kurs]$ _
```

Unter-Programme (Funktionen)

Obgleich die *BaSH* natürlich längst nicht so weitreichende Möglichkeiten zur strukturierten Programmierung von Bibliotheken-Verwendung bietet wie moderne Hochsprachen macht sie doch einige Angebote, die beim Einsatz von *shell-scripts* recht hilfreich sein können.

Ein Weg, um Bibliotheks-Dateien mit (von verschiedenen *scripts*) häufig verwendeten Vari ablen-Deklarationen oder Programm-Anweisungen zu verwenden, bietet der eingebaute **source**-Befehl, der auch mit einem einfachen **.** abgekürzt werden kann. Dieser Befehl liest den Inhalt der angebenen Datei ein und macht in gleichsam zum Bestandteil des gerade abgearbeiteten *scripts*. Eine andere Möglichkeit, die dieser Befehl eröffnet, ist das Anlegen bzw. Einlesen von Kon figurations-Dateien, so daßes nicht nötig ist, die *script*-Datei selbst zu bearbeiten, um sie –sagen wir: –auf einer andere Maschine im Netz lauffähig zu machen. Aber auch Status-Informationen könn ten bei Beendigung eines *scripts* in eine solche Datei geschrieben und bei einem Neustart von dort wieder eingelesen werden. Die Möglichkeiten des **source**-Befehls sind somit beinahe von der Kreativität der Programmiererin begrenzt.

Um Kode-Passagen, die sich womöglich zwar in den jeweils bearbeiteten Daten (Variablen-Wer ten) unterscheiden, nicht aber in ihrer Anweisungs-Folge (Verarbeitungs-Schritten), nicht mehrfach schreiben zu müssen –was nicht nur die *script*-Datei unnötig aufbläht, sondern auch die Möglichkeit von (Schreib-)Fehlern erhöhte –, bietet die *BaSH* die Möglichkeit, **funktions** zu implementieren. Dies sind Unterprogramme im eigentlichen Sinne, in gewissem Sinne selbständig funktionierende Programme, denen Parameter übergeben werden können, und deren Arbeits ergebnis über einen *exit-code* abgefragt werden kann. Übrigens können auch solche Funktionen in Dateien abgelegt werden, die mittels **source** erst zur Laufzeit eingebunden werden.

functions

A shell **function** stores a series of commands for later execution. Functions are executed in the context of the current shell; no new process is created to interpret them (contrast this with the execution of a shell script). When a **function** is executed, the arguments to the **function** become the *positional parameters* during its execution. The *special parameter* **#** is updated to reflect the change. *Positional parameter* **0** is unchanged.

Variables local to the function may be declared with the **local** builtin command. Ordinarily, variables and their values are shared between the function and its caller.

If the builtin command **return** is executed in a **function**, the function completes and execution resumes with the next command after the **function** call. When a function completes, the values of the *positional parameters* and the *special parameter* **#** are restored to the values they had prior to **function** execution.

Function names and definitions may be listed with the **-f** option to the **declare** or **typeset** builtin commands. Functions may be exported so that subshells automatically have them defined with the **-f** option to the **export** builtin.

Functions may be recursive. No limit is imposed on the number of recursive calls.

```
[matthias@Barbara Bash-Kurs]$ functest.sh
/home/staff/matthias/projects/Bash-Kurs: Check.It Scripts fortest.sh functest.sh hist.log
iftest.sh params.sh pingtest.sh tree.sh whiletest.sh  10
/home/staff/matthias/projects/Bash-Kurs/Scripts:  0
10 Dateien verarbeitet


[matthias@Barbara Bash-Kurs]$ cat functest.sh
#!/bin/bash
```

```
set -h -p

declare -i igCOUNT=0                    # globale Variable mit '0' initialisieren

function DoDir() {
local clNewDir="${1}"                   # lokale Variablen deklarieren/initialisieren
local clMyDir=$(pwd)
local ilDone=0
local clF=""
    builtin cd ${clNewDir}                  # ins neue Verzeichnis wechseln
    echo -n "${PWD}: "                  # Namen ausgeben
    for clF in * ; do                   # zuerst Dateien durchgehen
        [ -s ${clF} -a ! -L ${clF} ] && \
            echo -n "${clF##*/} " && let ilDone+=1
    done
    echo " ${ilDone}"
    for clF in * ; do                   # und nun Verzeichnisse bearbeiten
        [ -d ${clF} ] && DoDir ${clF}
    done
    builtin cd ${clMyDir}
    let igCOUNT=$[ ${igCOUNT} + ${ilDone} ]
    return ${iDone}
} # DoDir

# main()
if [ $# -eq 0 ] ; then
    DoDir .
else
    DoDir ${1}
fi
echo ${igCOUNT} Dateien verarbeitet

#_EoF_


[matthias@Barbara Bash-Kurs]$ _
```

Rekursionen

Rekursionen können in zweifacher Weise auftreten bzw. verwendet werden, die zudem häufig gemeinsam auftreten:

Rekursionen im Datei-System

Wenn etwa in mehreren (Unter-)Verzeichnissen gleiche oder ähnliche Aktionen durchgeführt werden (sollen), beispielsweise Dateien gelöscht oder umbenannt werden.

Rekursionen im Script

Wenn eine Programmteil, in der Regel eine **`function`**, sich selbst aufruft, um seine Arbeit unter veränderten Bedingungen (Parametern) fortzusetzen.

Bereits im vorherigen Beispiel hatten wir es mit beiden Arten von Rekursion zu tun: Die **`function DoDir()`** rief sich selber auf und übergab dabei an ihre neue „Instanz" einen jeweils anderen Verzeichnisnamen. Das folgende *script* nun arbeitet nach dem gleichen Prinzip, ist jedoch etwas weniger trivial, indem es die Anzahl der durchwanderten Verzeichnisse verwaltet und für seine Bildschirm-Ausgaben verwertet.

```
[matthias@Barbara Bash-Kurs]$ tree.sh ..
Initial directory = /home/staff/matthias/projects
|
+---Bash-Kurs
|   +---Scripts
+---virtualit
|   +---CVS
|
+  Total directories = 4


[matthias@Barbara Bash-Kurs]$ cat tree.sh
#!/bin/bash
set -h -p

declare -r cgLS="/bin/ls"                       # 'color-ls' wg. ESC-Codes vermeiden!
declare -i igDepth=0                            # globale Zaehler ...
declare -i igDirs=0                             # ... mit '0' initialisieren

function Search() {
local ilCount=0                                 # lokale Variablen ...
local clDir=""                                  # ... deklarieren und initialisieren
    for clDir in $(${cgLS} -dABFU *) ; do
```

```
    if [ "${clDir}" != "${clDir%/}" ] ; then      # ist's ein Verzeichnis?
       ilCount=0                                  # Zaehler zuruecksetzen
       while [ ${ilCount} -lt ${igDepth} ] ; do
          echo -n "|   "
          let ilCount+=1
       done
       echo "+---${clDir%/}"                      # Namen ohne 'Kennung' schreiben
       if builtin cd ${clDir} 2>/dev/null ; then
          let igDepth+=1
          Search                                  # Obacht! Rekursion ...
          let igDirs+=1
       fi
    fi
    done
    builtin cd ..
    let igDepth-=1
} # Search

# Main()
if [ $# -gt 0 ] ; then
    builtin cd ${1} || ${aError:?"can't find ${1}!"}
fi
echo "
Initial directory = $(pwd)
|"
Search                                            # beginnen wir die Suche ...
echo "|
+  Total directories = ${igDirs}
"
# _EoF_


[matthias@Barbara Bash-Kurs]$ _
```

Hier finden wir eine Menge von dem, was bislang diskutiert wurde: *globale* und *lokale* Variablen, ein *Unterprogramm*, **if**-Prüfungen, eine **for**-Schleife und, darin verschachtelt, eine **while**-Schleife, unterschiedliche Arten von *Expansionen*, weiter *numerische* Operationen sowie schließlich *Rekursion*. Hinzu kommt etwas, das bislang noch nicht gesondert angesprochen wurde.

Ein-/Ausgabe-Umleitungen und Pipelines

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. *Redirection* may also be used to open and close files for the current shell execution environment. The following *redirection operators* may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is **<**, the redirection refers to the *standard input* (file descriptor **0**). If the first character of the redirection operator is **>**, the redirection refers to the *standard output* (file descriptor **1**).

The **word** that follows the redirection operator in the following descriptions is subjected to *brace expansion*, *tilde expansion*, *parameter expansion*, *command substitution*, *arithmetic expansion*, *quote removal*, and *pathname expansion*. If it expands to more than one word, bash reports an error.

Note that the order of redirections is significant. For example, the command
ls > dirlist 2>&1
directs both *standard output* and *standard error* to the file **dirlist**, while the command
ls 2>&1 > dirlist
directs only the *standard output* to file **dirlist**, because the *standard error* was duplicated as *standard output* before the *standard output* was redirected to **dirlist**.

Redirecting Input

Redirection of input causes the file whose name results from the expansion of **word** to be opened for reading on file descriptor **n**, or the *standard input* (file descriptor **0**) if **n** is not specified. The general format for redirecting input is:

[n]<word

Redirecting Output

Redirection of output causes the file whose name results from the expansion of **word** to be opened for writing on file descriptor **n**, or the *standard output* (file descriptor **1**) if **n** is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.The general format for redirecting output is:

[n]>word

If the redirection operator is **>|**, then the value of the **-C** option to the **set** builtin command is not tested, and file creation is attempted.

Appending Redirected Output

Redirection of output in this fashion causes the file whose name results from the expansion of **word** to be opened for appending on file descriptor **n**, or the *standard output* (file descriptor **1**) if **n** is not specified. If the file does not exist it is created. The general format for appending output is:

[n]>>word

Redirecting Standard Output and Standard Error

Bash allows both the *standard output* (file descriptor **1**) and the *standard error output* (file descriptor **2**) to be redirected to the file whose name is the expansion of **word** with this construct. There are two formats for redirecting standard output and standard error:

&>word

and

>&word

Of the two forms, the first is preferred. This is semantically equivalent to

>word 2>&1

Here Documents

This type of redirection instructs the shell to read input from the current source until a line containing only **word** (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input for a command.

The format of *here-documents* is as follows:

<<[-]word

here-document

delimiter

No *parameter expansion*, *command substitution*, *pathname expansion*, or *arithmetic expansion* is performed on **word**.

If any characters in **word** are quoted, the **delimiter** is the result of quote removal on **word**, and the lines in the here-document are not expanded. Otherwise, all lines of the *here-document* are subjected to *parameter expansion*, *command substitution*, and *arithmetic expansion*. In the latter case, the pair **\<newline>** is ignored, and **** must be used to quote the characters ****, **$**, and **'**.

If the redirection operator is **<<-**, then all leading **tab** characters are stripped from input lines and the line containing **delimiter**. This allows *here-documents* within shell scripts to be indented in a natural fashion.

Duplicating File Descriptors

The redirection operator

[n]<&word

is used to duplicate input file descriptors. If **word** expands to one or more digits, the file descriptor denoted by **n** is made to be a copy of that file descriptor. If **word** evaluates to **-**, file descriptor **n** is closed. If **n** is not specified, the *standard input* (file descriptor **0**) is used.

The operator
[n]>&word
is used similarly to duplicate output file descriptors. If **n** is not specified, the *standard output* (file descriptor **1**) is used. As a special case, if **n** is omitted, and **word** does not expand to one or more digits, the *standard output* and *standard error* are redirected as described previously.

Opening File Descriptors for Reading and Writing

The redirection operator
[n]<>word
causes the file whose name is the expansion of **word** to be opened for both reading and writing on file descriptor **n**, or as the *standard input* and *standard output* if **n** is not specified. If the file does not exist, it is created.

```
[matthias@Barbara Bash-Kurs]$ echo * > echo.log


[matthias@Barbara Bash-Kurs]$ cat echo.log
Check.It Check.WasAnderes Check.active Check.history Check.newsfeeds Scripts fortest.sh
functest.sh hist.log iftest.sh params.sh pingtest.sh tree.sh whiletest.sh


[matthias@Barbara Bash-Kurs]$ _
```

Hier wird einfach die Ausgabe der **echo**-Anweisung in eine Datei namens **echo.log** umgeleitet, so daßsie nicht mehr am Bildschirm angezeigt wird. Um also das Ergebnis zu sehen, mußman sich den Inhalt der erzeugten Datei anschauen (hier mithilfe des externen Programmes **cat**).

Möchte man das Ergebnis eines Befehles *auch* in einer Datei protokollieren, es aber dennoch auf der Konsole (d.h. am Bildschirm) sehen, kann dies beispielsweise mit einer *Pipeline* realisiert werden. Dabei wird wie bei einer „normalen“ Umleitung die Ausgabe des ersten Befehles als Eingabe eines zweiten verwendet, der dann seinerseits seine Arbeitsergebnisse anzeigt (sofern sie nicht ihrerseits umgeleitet werden).

```
[matthias@Barbara Bash-Kurs]$ echo * | tee echotee.log
Check.It Check.WasAnderes Check.active Check.history Check.newsfeeds Scripts echo.log fortest.sh
functest.sh hist.log iftest.sh params.sh pingtest.sh tree.sh whiletest.sh


[matthias@Barbara Bash-Kurs]$ cat echotee.log
Check.It Check.WasAnderes Check.active Check.history Check.newsfeeds Scripts echo.log fortest.sh
functest.sh hist.log iftest.sh params.sh pingtest.sh tree.sh whiletest.sh


[matthias@Barbara Bash-Kurs]$ _
```

Hier wird die Ausgabe des **echo**-Befehls nicht ausdrücklich umgeleitet; stattdessen wird sie mittels des Pipe-Operators **|** als Eingabe an den externen **tee**-Befehl weitergereicht. Dieser Befehl akzeptiert einen Dateinamen als Aufruf-Parameter (hier **echotee.log**); in diese Datei schreibt er alles, was er als Eingabe erhält und zeigt es zugleich auf der Konsole (dem Bildschirm) an. Die Dateinamen, die wir als Ergebnis der Pipeline sehen, sind also nicht Ausgaben des **echo**-Befehles, sondern jene des **tee**-Befehls. –Um die Konfusion noch ein wenig zu steigern, können wir auch die Ausgaben des **tee**-Befehls umleiten (obgleich das dem Sinn dieses Programmes widerspricht).

```
[matthias@Barbara Bash-Kurs]$ echo * | tee echotee.log >tee.uml


[matthias@Barbara Bash-Kurs]$ cat tee.uml
Check.It Check.WasAnderes Check.active Check.history Check.newsfeeds Scripts echo.log
echotee.log fortest.sh functest.sh hist.log iftest.sh params.sh pingtest.sh tree.sh whiletest.sh


[matthias@Barbara Bash-Kurs]$ _
```

Wie bei der ersten, einfachen Umleitung sehen wir hier auf der Konsole gar nichts: die Ausgabe von **echo** dient als Eingabe von **tee**, dessen Ausgabe wiederum ist umgeleitet in die Datei **tee.uml** ...

Alldieweil sowohl die eingebauten Befehle der *BaSH*, als auch die weitaus meisten externen Dienstprogramme des Betriebssystems (*tools*) ihre Eingaben von *standard input* lesen (können) und ihre Ausgaben nach *standard output* schreiben (können), sind sie alle Umleitungs - und Pipeline-fähig. Damit können auch längere Befehls-Konstrukte entstehen wie

```
$ Befehl1 Optionen1 < Eingabe.Datei | Befehl2 Optionen2 | \
   Befehl3 Optionen3 | Befehl4 Optionen4 > Ausgabe.Datei
```

Das folgende shell-script verwendet neben anderen Fähigkeiten der BaSH sowohl die Ausgabe-Umleitung als auch Pipelines, um seine Aufgabe zu erfüllen:

```
[matthias@Barbara Bash-Kurs]$ cat pingtest.sh
#!/bin/bash
#
#   $Id: pingtest.sh,v 1.6 1997/10/27 16:40:48 matthias Exp $
#
#          Script zum Testen von Netz-Verbindungen
#   (C) 1996,97 DFG/M.Watermann <matthias@OLN.ComLink.APC.org>
#
#   kann bspw. regelmaessig durch Cron aufgerufen werden;
#   verwendete Programme: awk, date, hostname, mail, ping, rm;
#
#-----------------------------------------------------------------------------
set -h -p
#
# konfigurierbare Variablen:
#
PingHosts="backup7 erebus ipn-l nt4s uluru inx.de"       # zu testende Hosts
#
MailEMP="dlogs@Orakel.IPN.De"                            # Mail-Empfaenger bei Fehlern
#
PagerEMP="01723001633@d2-message.de"                     # Empfaenger fuer Teflon-Mails
PagerStart=7                                             # Start-Stunde fuer Pager-Mails
PagerEnd=22                                              # End-Stunde fuer Pager-Mails
#
Versuche=2                                               # Anzahl der Ping-Pakete
#
#-----------------------------------------------------------------------------
#   (you shouldn't change anything beyond this point!)
#
# interne Arbeitsvariablen:
declare -r HostName=`hostname`
declare -r Me="${0##*/}"
declare -r MyFiles="/tmp/${Me}"
declare -r MyPID="$$"
declare -r MailFile="${MyFiles}.${MyPID}.Mail"
declare -i PageHour=-1
declare -r RCS="\$Id: pingtest.sh,v 1.6 1997/10/27 16:40:48 matthias Exp $"
declare -i Result=0
declare -r Versuche=${Versuche:-3}
declare -r WorkFlag="${MyFiles}.MenAtWork"
Datum=""
aHost=""
#----------------------------------------------------------------------------

#   Main()
#
[   -f ${WorkFlag} ] && exit 1                           # andere Programm-Kopie aktiv

trap " rm -f ${MyFiles}* ; exit " 0 1 2 3 9 15           # Aufraeumen nicht vergessen

echo ${MyPID} >${WorkFlag}                               # Aktivitaet signalisieren
[   -w ${WorkFlag} ] || exit 1

if [ "${HostName%%.*}" = "Barbara" ] ; then              # bei Matthias
    PingHosts="christine ftp katharina m1 mail morgaine www"
    PagerEMP="mw" ; PagerStart=0 ; PagerEnd=23
    MailEMP="dlogs"
else
    PagerStart=${PagerStart:-0}
    PagerEnd=${PagerEnd:-23}
fi

if [ -n "${PagerEMP}" ] ; then
    PageHour=`date '+%k'`                                # aktuelle Stunde
    if [ ${PagerEnd} -ge ${PagerStart} ] ; then          # ohne Tageswechsel
        if [ ${PageHour} -lt ${PagerStart} ] ; then
            PageHour=-1
        else
            [ ${PageHour} -gt ${PagerEnd} ] && PageHour=-1
        fi
    else                                                 # Tageswechsel
        if [ ${PageHour} -lt ${PagerStart} ] ; then
            [ ${PageHour} -gt ${PagerEnd} ] && PageHour=-1
```

```
        else
            [ ${PageHour} -ne ${PagerStart} -a ${PageHour} -gt ${PagerEnd} ] && PageHour=-1
        fi
    fi
fi
(
for aHost in ${HostName} ${PingHosts} ; do
    Result=`ping -c ${Versuche} ${aHost} 2>/dev/null | awk '/packets received/ {print $4}'`
    Result=${Result:-0}
    if [ ${Result} -ne ${Versuche} ] ; then
        Datum=`date '+%d.%m.%y %T'`
    echo -en "\n\t${Datum} ${Me%.sh}@${HostName%%.*} kann \"${aHost}\" nicht erreichen!
        (${Result} Pakete erhalten statt ${Versuche})\n"
        if [ -n "${PagerEMP}" -a ${PageHour} -ge 0 ] ; then  # Teflon-Ruf abschicken
            if [ ${Result} -lt 1 ] ; then                    # =kein= Ping durchgekommen!
                mail -s "${Datum} ${Me%.sh}@${HostName%%.*} kann \"${aHost}\" nicht erreichen!" \
               ${PagerEMP} </dev/null &>/dev/null
            fi
        fi
    fi
done
)>${MailFile}
if [ -s ${MailFile} -a -n "${MailEMP}" ] ; then          # gibt's was?
    echo -en "\n\n\nNachricht erzeugt von:\n${RCS}" >>${MailFile}
    mail -s "Netzfehler!" ${MailEMP} <${MailFile} &>/dev/null
fi

exit 0                                                   # wg. Cron-Start immer 0
#_EoF_


[matthias@Barbara Bash-Kurs]$ _
```

Neben den bereits bekannten Dingen – der Deklaration von Variablen, verschiedenen Expansionen, unterschiedlichen Prüfungen und einer Schleife –finden wir hier mehrere Ein - und Ausgabe-Umleitungen sowie eine Pipeline. Darüberhinaus aber wird hier ein neuer eingebauter Befehl verwendet – **`trap`** –und eine *Sub-Shell* benutzt (hier einfach, um sämtliche Ausgaben der **`for`**-Schleife „auf einmal" in eine Datei umzuleiten).

Der Befehl **`trap`** wird benutzt, wenn *Signale*, die an die *shell* bzw. das aktive *script* geschickt werden, „abgefangen" werden sollen, um darauf in besonderer Weise zu reagieren. Welche *Signale* es gibt, verrät **`trap`** mit der Aufruf-Option **`-l`**:

```
[matthias@Barbara Bash-Kurs]$ trap -l
 0) EXIT          1) SIGHUP      2) SIGINT      3) SIGQUIT
 4) SIGILL        5) SIGTRAP     6) SIGIOT      7) SIGBUS
 8) SIGFPE        9) SIGKILL    10) SIGUSR1    11) SIGSEGV
12) SIGUSR2      13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGJUNK(16)  17) SIGCHLD    18) SIGCONT    19) SIGSTOP
20) SIGTSTP      21) SIGTTIN    22) SIGTTOU    23) SIGURG
24) SIGXCPU      25) SIGXFSZ    26) SIGVTALRM  27) SIGPROF
28) SIGWINCH     29) SIGIO      30) SIGPWR     31) SIGJUNK(31)


[matthias@Barbara Bash-Kurs]$ _
```

Eine Liste der unter Linux definierte Signale erhält man durch » **`man 7 signal`**«:

```
    Linux supports the following signals:

    Signal name | Value | Action | Comment
    ------------+-------+--------+----------------------------------
    SIGHUP      |   1   |   A    | Hangup detected
    SIGINT      |   2   |   A    | Interrupt from keyboard
    SIGQUIT     |   3   |   A    | Quit from keyboard
    SIGILL      |   4   |   A    | Illegal Instruction
    SIGTRAP     |   5   |   CG   | Trace/breakpoint trap
    SIGABRT     |   6   |   C    | Abort
    SIGUNUSED   |   7   |   AG   | Unused signal
    SIGFPE      |   8   |   C    | Floating point exception
    SIGKILL     |   9   |  AEF   | Termination signal
    SIGUSR1     |  10   |   A    | User-defined signal 1
    SIGSEGV     |  11   |   C    | Invalid memory reference
    SIGUSR2     |  12   |   A    | User-defined signal 2
    SIGPIPE     |  13   |   A    | Write to pipe with no readers
    SIGALRM     |  14   |   A    | Timer signal from alarm(1).
    SIGTERM     |  15   |   A    | Termination signal
    SIGSTKFLT   |  16   |   AG   | Stack fault on coprocessor
    SIGCHLD     |  17   |   B    | Child terminated
    SIGCONT     |  18   |        | Continue if stopped
```

```
SIGTSTOP    |  19   |  DEF   | Stop process
SIGTSTP     |  20   |   D    | Stop typed at tty
SIGTTIN     |  21   |   D    | tty input for background process
SIGTTOU     |  22   |   D    | tty output for background process
SIGIO       |  23   |   AG   | I/O error
SIGXCPU     |  24   |   AG   | CPU time limit exceeded
SIGXFSZ     |  25   |   AG   | File size limit exceeded
SIGVTALRM   |  26   |   AG   | Virtual time alarm (???)
SIGPROF     |  27   |   AG   | Profile signal
SIGWINCH    |  29   |   BG   | Window resize signal

The letters in the "Action" column have the following meanings:

A  Default action is to terminate the process.
B  Default action is to ignore the signal.
C  Default action is to dump core.
D  Default action is to stop the process.
E  Signal cannot be caught.
F  Signal cannot be ignored.
G  Not a POSIX.1 conformant signal.
```

Der Befehl »**`trap " rm -f ${MyFiles}* ; exit " 0 1 2 3 9 15`**« im obigen *pingtest-script* fängt neben dem „normalen“ *0* (fehlerfreies Ende) also die Signale *SIGHUP*, *SIGINT*, *SIGQUIT*, *SIGGKILL* und *SIGTERM* ab, um sicherzustellen, daß alle temporär erzeugten Dateien auch dann gelöscht werden, wenn das *script* durch eines dieser Signale abgebrochen wird. Ohne diese Vorsichtsmaßnahme würden bei jedem „außerplanmäßigen“ Programm-Abbruch Dateien im **`/tmp`**-Verzeichnis übrigbleiben ...

(!) Bei Skripts, die mehr als bloß eine Art vereinfachender Shell um ansonsten komplexe Programm-Aufrufe sind, müssen die zu erledigenden Arbeitsschritte „algorithmisiert“ werden, d.h. in einzelne Sequenzen unterteilt werden, die nacheinander abgearbeitet werden können. Die Teilnehmerinnen lernen die wesentlichen Programm-Strukturen kennen, die von der *BaSH* hierfür zur Verfügung gestellt werden.

praktische „Shell"-Programmierung

[Freitag]

Wenn öfter einmal *shell-scripts* zu schreiben sind, empfiehlt es sich, eine Art „Rumpf-Programm" zu entwerfen, das die immer wiederkehrenden Dinge enthält, wie in diesem Beispiel:

```
[matthias@Barbara Bash-Kurs]$ cat skeleton.sh
#!/bin/bash
#
#   $Id$
#
#          Script-Beschreibung/-Zweck
#
#          verwendete Programme: rm
#
#-----------------------------------------------------------------------------
set -h -p -x

#
# konfigurierbare Variablen:
#

#-----------------------------------------------------------------------------
# interne Arbeitsvariablen:
declare -r Me="${0##*/}"                      # Pfadnamen entfernen
declare -r Who="${Me#*.}"                     # Datei-Endung entfernen
declare -r ConfigFile="~/${Who}.conf"         # Konfigurations-Datei: privat
# oder:
# declare -r ConfigFile="/etc/${Who}.conf"    # Konfigurations-Datei: System
declare -r MyPID="$$"                         # PID des aktuellen Prozesses
declare -r MyFiles="/tmp/${Me}"               # Grund-Name fuer tmp.-Dateien
declare -r WorkFlag="${MyFiles}.MenAtWork"    # LOCK-Datei

#
# weitere Variablen-Deklarationen
#

#-----------------------------------------------------------------------------
# Unterprogramme/Funktionen
#

#-----------------------------------------------------------------------------
# Main()
#
[ -f ${WorkFlag} ] && exit 1                  # andere Programm-Kopie aktiv

trap " rm -f ${MyFiles}* ; exit " 0 1 2 3 9 15 # Aufraeumen sicherstellen

echo ${MyPID} >${WorkFlag}                    # Aktivitaet signalisieren
[ -w ${WorkFlag} ] || exit 1                  # sollte nie passieren ...

[ -s ${ConfigFile} ] && . ${ConfigFile}       # Konfig-Datei einlesen

#
# Anweisungen
#

#-----------------------------------------------------------------------------
#_EoF_


[matthias@Barbara Bash-Kurs]$ _
```

Dieses Grundgerüst verfügt bereits über die Möglichkeit, eine Konfigurations-Datei einzulesen (was natürlich längst nicht bei jedem *shell-script* nötig ist). Es stellt sicher, daß es zu einem gegebenen Zeitpunkt nur einmal aktiv ist, indem es eine *Flag-Datei* erzeugt und prüft. Und es sorgt schließlich dafür, daß die von ihm angelegten temporären Dateien auch wieder gelöscht werden. Wird dieses Script-Skelett aufgerufen, sehen wir dies:

```
[matthias@Barbara Bash-Kurs]$ skeleton.sh
+ declare -r Me=skeleton.sh
+ declare -r Who=skeleton
+ declare -r ConfigFile=~/skeleton.conf
+ declare -r MyPID=6770
+ declare -r MyFiles=/tmp/skeleton.sh
```

```
+ declare -r WorkFlag=/tmp/skeleton.sh.MenAtWork
+ [ -f /tmp/skeleton.sh.MenAtWork ]
+ trap  rm -f /tmp/skeleton.sh* ; exit  0 1 2 3 9 15
+ echo 6770
+ [ -w /tmp/skeleton.sh.MenAtWork ]
+ [ -s ~/skeleton.conf ]
+ rm -f /tmp/skeleton.sh.MenAtWork
+ exit


[matthias@Barbara Bash-Kurs]$ _
```

Der Grund für diese Bildschirm-Ausgaben liegt in der **-x** Option des eingebauten **set**-Befehls, die oben gemeinsam mit den **-h** und **-p** Optionen zu sehen ist. Durch das Setzen der **-x** Option wird die *BaSH* veranlaßt, alle Anweisungen, die sie ausführt, auf die Konsole (d.h. den Bildschirm) auszugeben. Dies geschieht, nachdem alle Expansionen durchgeführt worden sind. Bei umfänglicheren *scripts* empfiehlt es sich, die Debug-Ausgaben in eine Datei umzuleiten (**scriptname &>script.log**) oder sie via Pipeline an Programme wie **tee** oder **less** weiterzuleiten (**scriptname 2>&1 | tee script.log**)

Beim Schreiben von *shell-scripts* ist dieser **-x** Schalter ein unentbehrliches Hilfsmittel beim Testen und Aufspüren von Fehlern. Er sollte daher solange gesetzt bleiben, bis alle möglichen Fehlerquellen lokalisiert und beseitigt sind.

(!) Abschließend sollen die Teilnehmerinnen ihr erworbenes Wissen anhand kleinerer Pro grammier-Aufgaben erproben.

ARBEITEN MIT DER LINUX-SHELL „BASH'' 1
EINFÜHRUNG / ALLGEMEINES 2
Rechner-Hierarchie 2
Software-Hierarchie 2
OS - Gegenüberstellung 2
Das Linux-Datei-System 3
DIE SHELL „BASH" 15
Aufruf-Bearbeitung 15
Konfiguration 15
System-global 16
Anwender-spezifisch 17
Umgebungs-Variablen 19
DIE „BASH" ALS BENUTZER-OBERFLÄCHE 23
eingebaute Befehle 23
DIE BEFEHLSZEILE DER „BASH" 32
Sonderzeichen und reservierte Wörter 32
Die readline-Bibliothek 32
Commands for Moving 36
Commands for Manipulating the History 36
Commands for Changing Text 37
Killing and Yankin 38
Numeric Arguments 38
Completing 38
Keyboard Macros 39
Miscellaneous 39
Befehls-Geschichte („History") 40
Event Designators 40
Word Designators 40
Modifiers 41
HÄUFIG GENUTZTE DIENSTPROGRAMME 43
Hilfe finden 43
Anzeigen von Dateien 43
cat 44
more und less 44
head und tail 45
Auswerten von Dateien 46
grep 46
awk 46
Verändern von Dateien 47
sort 47
sed 47
„BASH" ALS PROGRAMMIER-PLATTFORM 49
positionale Parameter 49
spezielle Parameter 49
(Umgebungs-)Variablen 50
Expansionen 51
Brace Expansion 51
Tilde Expansion 52
Parameter Expansion 52
Command Substitution 54
Arithmetic Expansion 54
Word Splitting 55
Pathname Expansion 55
PROGRAMM-STRUKTUREN 57
sequentielle Abarbeitung 57
bedingte Verzweigungen 57
if / elif / else 57
case 58
Schleifen 59
for 60
while ... / ... until 60

Unter-Programme (Funktionen) *61*
functions 61
Rekursionen 62
Ein-/Ausgabe-Umleitungen und Pipelines *63*
Redirecting Input 63
Redirecting Output 64
Appending Redirected Output 64
Redirecting Standard Output and Standard Error 64
Here Documents 64
Duplicating File Descriptors 64
Opening File Descriptors for Reading and Writing 65
PRAKTISCHE „SHELL“-PROGRAMMIERUNG 69