Kapitel 11

Korn-Shell

Die Korn-Shell (`ksh`) ist eine Weiterentwicklung der Bourne-Shell (`sh`), der Standard-Shell unter Unix. Die Korn-Shell bietet einiges mehr an Komfort, wenn auch nicht ganz so viel wie moderne Shells aus der GNU-Freeware-Familie, wie z.B. `tcsh` oder `bash`. Die Korn-Shell ist die Default-Shell auf Rechnern der Hersteller IBM und HP.

Viele Funktionen der Korn-Shell sind vergleichbar mit denen der C-Shell oder gar der Tenex-C-Shell. Auch die Konzepte, wie diese Funktionen umgesetzt sind, unterscheiden sich oft nur in einigen wenigen, dafür aber oft sehr wichtigen Details von den Konzepten der beiden anderen Shells, die in diesem Buch vorgestellt werden. Aus diesem Grund gibt es in den nachfolgenden Abschnitten sicherlich einige Wiederholungen aus dem vorangegangenen Kapitel, aber auch viele entscheidende Unterschiede.

Das wohl wichtigste neue Feature der Korn-Shell ist das *Command Line Editing*. Während bei der Bourne-Shell bereits abgesetzte Befehle nicht wiederholt werden können und bei der C-Shell nur ein sehr umständlicher History-Mechanismus zur Verfügung steht, können unter der Korn-Shell vorherige Befehle erneut auf die Kommandozeile geholt, dort verändert und erneut abgesetzt werden. Die Korn-Shell bietet hierfür zwei Editiervarianten an: Die Kommandozeile kann entweder mit `emacs` oder mit `vi`-Befehlen[1] bearbeitet werden.

Neben ihrer Funktion als interaktiver Kommandointerpreter ist die Shell auch ein Interpreter für ihre eigene Programmiersprache. Diese Programmiersprache dient der Formulierung von Kommandoprozeduren, die unter

1 `emacs` und `vi` sind die beiden am häufigsten benutzten Unix-Texteditoren.

Unix allgemein *Shell-Scripte* genannt werden. Die Scriptsprache der Korn-Shell ist bis auf einige Erweiterungen die gleiche wie die der Bourne-Shell.

11.1 Die Arbeitsweise der ksh

Ohne die Shell, den Kommandointerpreter, wären Unix-Benutzer ziemlich hilflos, und obwohl ich in den vorangegangenen Kapiteln immer nur von »der Shell« gesprochen habe, als gäbe es nur eine, bietet Unix auch hier wieder ein Beispiel für seine Flexibilität. Anders als bei den meisten bekannten Betriebssystemen, kann der Benutzer unter Unix zwischen verschiedenen Kommandointerpretern wählen.[2]

Egal, ob Sie sich bei seiner Wahl für komfortable Public-Domain-Shells wie die `bash` oder `tcsh` entscheiden, die kommerziell bevorzugten `ksh` oder `csh` wählen oder als ewig Gestriger an der guten alten `sh` festhalten – jede dieser Alternativen hat eine Reihe von Features auf Lager, die Ihnen den Umgang mit dem Betriebssystem erleichtern können. Damit Sie nicht genau das Gegenteil erfahren, sollte Ihnen die Arbeitsweise der Shell bekannt sein.

Einer der Mechanismen, den die Shell anbietet, sind die im Abschnitt über Dateiverwaltung vorgestellten *Wildcards*. Entgegen der Meinung vieler Anwender sorgen nämlich nicht die einzelnen Befehle für die richtige Handhabung dieser Sonderzeichen, sondern der Kommandointerpreter. Eigenen Programmen und Scripten wird der Mechanismus zur Behandlung von Wildcards dadurch auf einfache Art verfügbar gemacht. Dieses Vorgehen bringt aber auch Probleme mit sich. Werden bei der Auflösung von Wildcards zu viele (in der Regel sind das mehr als 256) Dateinamen von der Shell generiert, streikt diese mit der Fehlermeldung »arg list too long«.

Ungewollte Effekte können auch eintreten, wenn Wildcards, die nicht für die Shell bestimmt sind, nicht entsprechend unter Verwendung des *Escape Characters* (\) oder über den Quoting-Mechanismus, über den jede Shell verfügt (Abschnitt 11.4) geschützt werden.


```
UNIX-Kurs [~] 93 > find / -name \*.tmp -print
UNIX-Kurs [~] 94 > find / -name *.tmp -print    (falsch)
```

Das erste Kommando im Beispiel durchsucht den gesamten Dateibaum nach Dateien mit der Endung `.tmp`. Bei dem zweiten Kommando wird das Wildcard-Zeichen `*` bereits durch die Shell – und nicht durch das Kom-

2 Bei einigen Unix-Derivaten, wie z.B. dem IRIX von SGI, darf man sogar seine Default-Shell (die in der `/etc/passwd` eingetragen ist) umsetzen, wofür es normalerweise des Eingriffs eines Systemverwalters bedarf.

mando `find` – ausgewertet. Existierten z.B. im Working Directory genau zwei Dateien `a.tmp` und `b.tmp` mit der Endung `.tmp`, so setzt die Shell die Eingabe in das Kommando `find / -name a.tmp b.tmp -print` um. Da nach der Option `-name` nur ein Muster stehen darf, würde dieses Kommando also einen Syntaxfehler erzeugen.

Wann muß nun ein Sonderzeichen durch den Escape-Character geschützt werden? Merken Sie sich folgende Regel:

Soll das Sonderzeichen vor der Kommandoausführung interpretiert werden, braucht man es nicht zu schützen. Soll es erst während der Kommandoausführung interpretiert werden (also durch das Kommando selbst, wie bei dem Beispiel mit `find`), muß es geschützt werden.

Da die Shell als Kommandointerpreter des Unix-Betriebssystems im Gegensatz zu den Interpretern anderer Systeme den Großteil ihrer Funktionalität nicht über eingebaute Befehle umsetzt, muß sie zur Ausführung eines Kommandos zuerst ermitteln, wo das entsprechende auszuführende Programm steht. Damit hier nicht ganze Dateibäume durchsucht werden müssen, verfügen alle Shells über das Konzept des Suchpfads für Kommandos. Über den Suchpfad erfährt die Shell, in welchen Verzeichnissen sie nach den entsprechenden Programmen suchen soll. Erst das richtige Setzen dieses Pfades (durch Setzen der Environmentvariablen `PATH`) ermöglicht ein Arbeiten mit den Unix-Kommandos.


```
UNIX-Kurs [~] 95 > echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/usr/bin/X11:/etc:.
```

Die Verzeichnisse werden genau in der Reihenfolge, wie sie in der Variablen `PATH` stehen, nach einer ausführbaren Datei mit dem Namen des gesuchten Kommandos durchsucht. Die Suche ist beendet, sobald eine solche Datei gefunden wird oder alle Verzeichnisse im Suchpfad durchsucht wurden.

Um das Verhalten eines Programms zu beeinflussen, gibt es neben den Kommandozeilenparametern noch den Mechanismus der Environmentvariablen. Jedes unter Unix gestartete Programm bekommt eine Kopie des Satzes an Environmentvariablen, der zum Startzeitpunkt für das aufrufende Programm definiert war, als »Beigabe«. Die Shell bietet Möglichkeiten, diese Variablen zu setzen und ihre Werte abzurufen. In der Eingabezeile *substituiert* die Shell den *Variablennamen* durch den Wert der Variablen, sobald ein Dollarzeichen »$« vor den Variablennamen geschrieben wird. So enthält z.B. die Variable `HOME` den absoluten Pfad des Home Directory des

Benutzers. Die Eingabe von `cd $HOME` bewirkt daher den Wechsel in das Home Directory.

Neben der Ausgabe der Ergebnisse von Kommandos auf den Bildschirm bietet jede Shell auch eine Möglichkeit, die *Ausgaben* in eine Datei »*umzulenken*«. Leider hat sich auch hier keine einheitliche Syntax durchgesetzt, was die Nutzung dieses Mechanismus abhängig von der verwendeten Shell macht.

Moderne Shells wie `ksh`, `csh` und `tcsh` erlauben es dem Benutzer, Kürzel für häufig benutzte Befehlsfolgen zu definieren. Diese Kürzel oder *Aliase* werden von der Shell durch die dahinter verborgenen Befehlsfolgen ersetzt. Eine der beliebtesten Abkürzungen ist `ll` für den häufig verwendeten Befehl `ls -l`. Falls Sie sehr an das DOS-Betriebssystem gewöhnt sind, möchten Sie vielleicht auch das Kommando `ls` über den Alias `dir` ansprechen.

Während die `csh` und die `tcsh` mit dem Alias nur ein Konzept zur Abkürzung von häufig benutzten Befehlsfolgen kennen, bietet die Korn-Shell einen weiteren, sehr flexiblen, Mechanismus an – die *Functions*. Eine Function ist ein Alias, bei dem die Kommandozeilenparameter, wie in einem Shell-Script einzeln angesprochen werden können. Solche Functions werden in den Startup-Dateien der Korn-Shell vereinbart.

11.2 Environmentvariablen

Ein wichtiger Mechanismus zur Übergabe von Parametern an Programme ist neben den Kommandozeilenparametern die Nutzung der *Environmentvariablen*.

Jeder Prozeß, der unter Unix läuft, wird, mit Ausnahme des Schedulers, von einem anderen Prozeß gestartet. Bei diesem Startvorgang wird dem Prozeß eine Kopie des Environments des aufrufenden Prozesses mitgegeben. Zu diesem Environment gehören neben dem Pfad des Working Directory auch ein Satz von Variablen: die Environmentvariablen.

Diese Variablen haben unterschiedliche Funktionen und Aufgaben. Einige haben rein informativen Charakter (wie z.B. `PWD`, die den Pfad des Working Directory beinhaltet), andere wirken steuernd (wie z.B. `DISPLAY`, die festlegt, welche Maschine die Ausgabe eines X-Programms anzeigt), und wieder andere ermöglichen das Ausführen von Programmen (`PATH` enthält die Suchpfade für Programme).

Der Benutzer kann sich das Environment durch entsprechende Befehle anzeigen lassen und es verändern.

env

Das Kommando `env` gibt die Namen aller Environmentvariablen, die zur Zeit definiert sind, und deren aktuelle Werte aus.

export

Das interne Shell-Kommando `export` setzt eine Variable auf einen neuen Wert und exportiert sie dann in das Environment. War die Variable bereits definiert, wird ihr Wert mit dem neuen Wert überschrieben. War sie noch nicht definiert, wird sie in das Environment eingefügt und mit dem neuen Wert belegt. Das führende Dollarzeichen ist nur bei der Ausgabe des Inhalts einer Variablen anzugeben, nicht aber bei ihrer Definition.

`export` *Variablenname=Wert*


```
UNIX-Kurs [~] 96 > export DISPLAY=:0.0
UNIX-Kurs [~] 97 > echo $DISPLAY
:0.0
```

In diesem Beispiel wird die Environmentvariable `DISPLAY`, die angibt, auf welchem Display die Ausgaben von X11-Programmen erscheinen sollen, auf den Bildschirm der lokalen Maschine (`:0.0` gelesen als X-Server 0 Screen 0) gesetzt. Anschließend wird mit `echo` überprüft, ob das Setzen erfolgreich war.

Allerdings kann das Belegen einer Variablen mit einem Wert und das Exportieren der Variablen in das Environment auch in zwei Schritten erfolgen.


```
UNIX-Kurs [~] 98 > DISPLAY= :0.0
UNIX-Kurs [~] 99 > export DISPLAY
UNIX-Kurs [~] 100 > echo $DISPLAY
:0.0
```

`unset`

Um definierte Environmentvariablen wieder zu löschen, wird das Kommando `unset` benutzt.

`unset` *Variablenname*

```
UNIX-Kurs [~] 101 > unset DISPLAY
```

Durch diesen Befehl wird das Setzen der Environmentvariablen `DISPLAY` wieder rückgängig gemacht.

11.3 Shell-Variablen

Neben den Environmentvariablen verfügen die meisten Shells auch noch über sogenannte Shell-Variablen. Diese werden für drei Zwecke benutzt:

- ✘ zur Abbildung von Environmentvariablen in die für die Shell spezifische Darstellung;
- ✘ zur Steuerung des Verhaltens der Shell;
- ✘ in Shell-Scripten zum Zwischenspeichern von Ergebnissen.

Auch die `ksh` verfügt über solche Shell-Variablen. Im Gegensatz zu anderen Shells werden bei der Korn-Shell die Shellvariablen aber fast ausschließlich in Korn-Shell-Scripten eingesetzt.

Um eine Shell-Variable zu definieren, muß folgende Zuweisung angegeben werden:

Variablenname=Wert

11.4 Quoting

Für große Verwirrung bei Unix-Einsteigern sorgen immer wieder die unterschiedlichen Anführungszeichen, die bei der Eingabe von einigen Unix-Kommandos nötig sind. Dieses Thema ist schon an der einen oder anderen Stelle angeschnitten worden, ohne allerdings den gesamten Umfang der Möglichkeiten und Aufgaben der Quotes offenzulegen. Nachdem nun die wichtigsten Unix-Kommandos ausführlich erläutert worden sind, ist es an

der Zeit, die Quotes in einem eigenen Abschnitt genau unter die Lupe zu nehmen. Das dies im Kapitel über die `csh` und `tcsh` erfolgt, ist dabei nicht so wichtig, denn dieser Mechanismus ist in allen Shells gleich.

Aufgabe der Quotes ist es generell, Teile der Kommandozeile zusammenzufassen. Unix verwendet für diese Aufgabe drei unterschiedliche Arten von Quotes:

- ✘ Double Quotes (doppelte Anführungszeichen[3]) `""`
- ✘ Single Quotes (einfache Anführungszeichen[4]) `''`
- ✘ Back Quotes (rückwärts gerichtete einfache Anführungszeichen[5]) ``` `` ```

Double Quotes haben zwei Aufgaben. Mit Hilfe von Double Quotes können Zeichenketten, die Leerzeichen enthalten, zu einem Kommandozeilenparameter zusammengefaßt werden. Dies wird immer dann notwendig, wenn eine Zeichenkette, die Leerzeichen enthält, an ein Kommando übergeben werden soll, z.B. als Suchmuster. Das Leerzeichen verliert dann seine Funktion als Trennzeichen von Kommandozeilenparametern.

Des weiteren verlieren innerhalb von Double Quotes Wildcards wie »*«, »~« und »[]« ihre Sonderbedeutung. Allerdings gibt es auch Zeichen, die von den Double Quotes nicht von ihrer Sonderfunktion befreit werden. So werden Variablen wie `$HOME` nach wie vor durch ihren Inhalt ersetzt und Kommandos in Back Quotes durch ihre Ausgaben substituiert.

Single Quotes sind eine verschärfte Form der Double Quotes. Sie haben genau die gleichen Aufgaben, aber innerhalb von Single Quotes werden keine Variablen mehr ersetzt, und auch die Kommmandosubstitution wird nicht mehr durchgeführt.

Zwischen den Back Quotes steht immer ein vollständiges Unix-Kommando. Dieses Kommando wird von der Shell ausgeführt, bevor die restliche Kommandozeile interpretiert wird. Die Ausgabe, die das in Back Quotes eingeschlossene Kommando erzeugt, wird an der Stelle in die Kommandozeile eingefügt, an der die Back Quotes stehen. Sinn dieser Quotes ist es, Unix-Kommandos miteinander zu verknüpfen.

3 Es muß das ASCII-Zeichen »Double Quote« verwendet werden, also nicht die in der Textverarbeitung üblichen typografischen englischen oder deutschen Anführungszeichen.

4 Es muß das Apostroph-Zeichen verwendet werden, nicht der Acute-Akzent und auch nicht das typografische Anführungszeichen.

5 Auf europäischen Tastaturen gibt es meist keine eigene Taste, sondern man muß zuerst den Gravis-Akzent und dann die Leertaste drücken, um dieses ASCII-Zeichen zu erhalten.

```
UNIX-Kurs [~] 102 > ls -l ‘fgrep -l Meier *.c‘
```

Bei diesem Beispiel haben wir zwei Unix-Befehle auf der Kommandozeile. `ls -l` erzeugt ein Listing mit allen Attributen der angegebenen Dateien. `fgrep -l Meier *.c` listet alle Dateien mit der Endung `.c` auf, in deren Inhalt die Zeichenkette `Meier` vorkommt. Da das Kommando `fgrep` mit seinen Parametern in Back Quotes steht, wird es zuerst ausgeführt. Die Namen der Dateien, die es als Ergebnis liefert, werden in der Kommandozeile hinter das `ls -l` geschrieben. So werden die beiden Befehle zu dem neuen Befehl kombiniert: »Liste alle Attribute jener Dateien mit der Endung `.c` auf, welche die Zeichenkette `Meier` enthalten.«

Diese Variante des Einsatzes von Quotes heißt *Kommandosubstitution*.

11.5 Alias-Namen

Die Korn-Shell bietet durch den Alias-Mechanismus ein einfaches Hilfsmittel an, um häufig eingegebene Befehlsfolgen über einen Kurznamen auszuführen. Der Alias-Mechanismus der Korn-Shell ist eine einfache Textersetzung und soll in erster Linie Tipparbeit sparen helfen. Das Belegen von allen Buchstaben der Tastatur mit neuen Funktionen ist allerdings nicht die Intention des Alias-Mechanismus. Durch diese zusätzliche Verkürzung der ohnehin schon kurzen Unix-Befehlsnamen würde jeglicher Zusammenhang zwischen Benennung und Funktion eines Kommandos zerstört. Ein solches Vorgehen muß über kurz oder lang zu folgenschweren Fehlbedienungen führen.

alias

Alias-Namen werden mit dem shell-internen Kommando `alias` definiert.

alias *Alias-Name Kommando*

Wird `alias` ohne Parameter aufgerufen, listet es alle definierten Alias-Namen auf.

Wird ein Kommando abgesetzt, das als Alias definiert ist, wird dieses von der Shell bei der Alias-Substitution durch den Wert des Alias ersetzt.


```
UNIX-Kurs [~] 103 > alias ll="ls -la"
UNIX-Kurs [~] 104 > ll a*
```

In dem Beispiel wird der Alias `ll` textuell durch seine Definition (`ls -la`) ersetzt und die so entstandene neue Kommandozeile (`ls -la a*`) durch die Shell ausgeführt.

Umplazierung der Parameterliste

Nicht bei allen denkbaren Abkürzungen durch Alias-Namen wäre eine solche reine textuelle Ersetzung erfolgreich. Häufig muß die Parameterliste, die nach einem Alias-Namen beim Aufruf steht, in den Befehl eingefügt werden, um eine sinnvolle Ersetzung zu ergeben. Das ist häufig bei Alias-Namen für `find`-Kommandos oder bei Kommandoverkettungen über Pipes der Fall. Während der Alias-Mechanismus der `csh` und der `tcsh` eine Möglichkeit zur Umplazierung der Parameterliste anbietet, muß der Benutzer der Korn-Shell auf ein anderes Konzept, nämlich das der Functions, zurückgreifen, vgl. Abschnitt 11.6.

Permanente Alias-Namen

Wurde ein Alias interaktiv in der Shell definiert, ist er nur auf dieser einen Shell verfügbar und verschwindet, sobald die Shell beendet wird. Dies kann verhindert werden, indem solche Alias-Namen, die in jeder Shell verfügbar sein sollen, in die Startup Datei `.profile` oder – sofern vorhanden – in die Datei `.kshrc` der `ksh` eingetragen werden. Die Definition erfolgt hier ebenso wie auf der Kommandozeile.

`unalias`

Soll ein Alias gelöscht werden, so geschieht dies mit dem Kommando `unalias`.

`unalias` *Alias-Name*


```
UNIX-Kurs [~] 105 > unalias ll
UNIX-Kurs [~] 106 > ll
ll - Command not found
```

Das Löschen eines Alias-Namens hat in der Regel zur Folge, daß das entsprechende Kommando nicht mehr von der Shell gefunden wird.

Aber Vorsicht: Unter Umständen kann an die Stelle des Alias jetzt ein Kommando aus dem Suchpfad treten, das durch den Alias überdefiniert worden war.

Was man alles mit dem `alias`-Kommando der Korn-Shell machen kann, soll das folgende Beispiel, welches von einer Sun Solaris-Maschine stammt, zeigen.


```
UNIX-Kurs [~] 107 > cat .kshrc

......
export VISUAL=emacs

alias __A=‘echo "\020"‘
alias __B=‘echo "\015"‘
alias __C=‘echo "\006"‘
alias __D=‘echo "\002"‘
alias __P=‘echo "\004"‘
alias __H=‘echo "\001"‘
......
rm -f $HOME/.sh_history
```

In der Datei `.kshrc` werden permanente Aliasnamen für die Korn-Shell definiert. Der hier gezeigte Ausschnitt aus einer solchen Datei macht sich zunutze, daß man auch Aliasnamen definieren darf, die nicht druckbare Zeichen enthalten. In unserem Beispiel werden hier Aliasnamen für die Zeichenfolgen, die von den Cursortasten gesendet werden, definiert. So sendet z.B. die ↑-Taste einen Tastaturcode, der auf den Alias `__A` zutrifft. Als Wert für den Alias wird die Ausgabe von der Kommandosubstitution ‘`echo "\020"`‘ definiert. Dieses `echo`-Kommando gibt ein Control-P (Strg P) aus (`\020` ist die oktale Repräsentation für Control-P). Das heißt, jedesmal, wenn die ↑-Taste gedrückt wird, wird durch den Alias ein Control-P an die Shell gesandt.

Wozu das Ganze? Das Commandline Editing der Korn-Shell funktioniert entweder mit den Positionierungskommandos für den `vi` (die Environmentvariable `VISUAL` hat den Wert `vi`) oder mit denen für den `emacs` (die Environmentvariable `VISUAL` hat den Wert `emacs`). Diese beiden Editoren der Unix-Welt benutzen zur Positionierung nicht die Cursortasten, sondern bestimmte Tastenfolgen. So muß man z.B. Strg P eingeben, um den Cursor im `emacs` um eine Zeile nach oben zu positionieren. Da diese Positionierungskommandos nur schwer zu merken sind und man als Benutzer einfach mit den Cursortasten in der Kommandozeile arbeiten möchte, kann man durch

derartige Aliasdefinitionen die Cursortasten auf die Positionierungskommandos des verwendeten Kommandozeileneditors abbilden. Damit ist jetzt das Editieren der Eingaben mit Hilfe der Cursortasten möglich.

Wenn wir schon mal bei den netten kleinen Tricks im Umgang mit dem Commandline Editing der Korn-Shell sind, sollte auf jeden Fall der folgende Trick nicht unerwähnt bleiben. Die Command History – also die Liste der bereits abgesetzten Kommandos – wird unter der Korn-Shell in einer Datei abgelegt. Diese Datei trägt den Namen `.sh_history` und liegt im Home Directory des Benutzers. Leider werden die Kommandos aus allen laufenden Shells des Benutzers in dieser Datei gespeichert, sofern sie existiert. Das Resultat ist, daß in der Command History aus dem Fenster A auch Befehle auftauchen, die im Fenster B abgesetzt worden sind und umgekehrt.

Um dieses zu verhindern, kann der Benutzer in seiner Datei `.kshrc` einfach die beim Start der Shell angelegte Datei `.sh_history` löschen (`rm -f $HOME/.sh_history`). Als Folge hat nun jedes Fenster seine eigene Command History, welche auch nur Befehle aus diesem Fenster enthält.

11.6 Functions

Wer mit einer höheren Programmiersprache vertraut ist, wird mit einer Funktion eine feste Assoziation verbinden, die allerdings auf den ersten Blick nicht so gut in das Konzept eines interaktiven Kommandointerpreters zu passen scheint.

Functions haben ihren Ursprung in der Erstellung von *Shell-Scripten*. Während sich bei der `csh` und `tcsh` Shell-Scripte nur durch die Ablaufstrukturen (`if`, `for`, `while`, ...) strukturieren lassen, bietet die Korn-Shell die Möglichkeit, Teile eines Scripts, die sich wiederholen bzw. wiederholt aufgerufen werden, in Form einer Function zusammenzufassen. Eine solche Function kann wie ein Unterprogramm in einer Programmiersprache aufgerufen werden und einen Resultatwert zurückliefern. Im Gegensatz zu einem Unterprogramm in einer Programmiersprache haben Functions keine feste Parameterliste, können also theoretisch mit beliebig vielen Parametern aufgerufen werden. Die Aufrufparameter werden in den Variablen `$1` bis `$`*n* abgelegt und können einzeln angesprochen werden. Somit ist die Function im Prinzip nichts anderes als ein Script im Script.

Ist eine Function einmal definiert, wird sie von der Shell wie ein anderes Unix-Kommando auch behandelt. Das heißt, bevor die Function aufgerufen wird, wird die Parameterliste nach Metazeichen durchsucht und gegebenenfalls expandiert. Mit einer Function kann also im Prinzip das gleiche erreichen werden wie mit einem Script. An dieser Stelle kommt nun die Bedeu-

tung von Functions für interaktive Shells zum Tragen. Nicht für jede kleine sich wiederholende Aufgabe, die sich durch drei, vier Unix-Kommandos erledigen läßt, möchte man gleich ein Shell-Script schreiben. Also schreibt man eine Function, die man in einer der Startup-Dateien der Korn-Shell definiert. Wird eine neue Korn-Shell gestartet, werden diese Dateien durchlaufen und die Function in der interaktiven Shell definiert. Nun kann die Function, wie ein anderes Unix-Kommando auch, auf der Kommandozeile dieser Shell aufgerufen werden.

Da das Erstellen von Functions sehr viel mit der Programmierung von Shell-Scripten gemeinsam hat und an dieser Stelle kein Vorgriff auf den entsprechenden Abschnitt erfolgen soll, wird hier nur ein sehr einfaches Beispiel für eine Function gezeigt.


```
UNIX-Kurs [~] 108 > cat .kshrc

....
function ff {
   find / -name "$1" -print
}
....

UNIX-Kurs [~] 109 > ff "*.txt"
```

Die Function `ff` steht für ein `find`-Kommando, das Dateien in dem gesamten Unix-Dateibaum sucht. Ohne die Function müßte unser Kommando für den Anwendungsfall in der zweiten Beispielzeile wie folgt lauten:

```
find / -name "*.txt" -print
```

Dieses Kommando durchsucht den gesamten Dateibaum vom Root-Directory (/) an nach Dateien mit der Endung `.txt` und gibt alle Treffer mit absolutem Pfad aus. Das Suchkriterium lautet »`*.txt`« und muß zwischen dem `-name` und dem `-print` stehen, also zwischen zwei Bestandteilen des Kommandos, die eigentlich in der Function verankert sind. An dieser Stelle reicht eine simple Textersetzung wie beim Alias-Konzept der Korn-Shell nicht mehr aus, weil Text in die Parameterliste (»`*.txt`«) des Kommandos eingefügt werden muß.

Als Platzhalter für diesen Text, der als Parameter an die Function übergeben wird, wird die Variable `$1` benutzt, die den ersten Kommandozeilenparameter vom Aufruf der Function beinhaltet.

11.7 Die Datei .profile

Die Datei `.profile` ist die Startup-Datei der `ksh` und auch der `sh`. Bei Shells unter Unix wird zwischen den sogenannten *Login Shells* und den »normalen« interaktiven Shells, die z.B. aus einer anderen Shell heraus gestartet werden, unterschieden. Der Name der Login Shell ist in der `/etc/passwd` eingetragen. Diese Shell wird als erste beim Einloggen gestartet.

Wird eine `ksh` oder `sh` als Login Shell gestartet, sucht sie im Home Directory des Benutzers nach einer Datei `.profile`. Ist diese vorhanden (Ausführungsrechte müssen nicht gesetzt sein), wird der Inhalt der Datei Zeile für Zeile ausgeführt. Die Datei `.profile` wird verwendet, um Terminal-Charakteristika für die gerade gestartete Shell einzustellen (z.B. Definition der Taste, mit der auf der Kommandozeile gelöscht wird). Im Zusammenhang mit der Korn-Shell wird in der Regel über die Environmentvariable ENV in der Datei `.profile` der Name einer weiteren Datei vereinbart. Diese Datei wird von der Korn-Shell ebenfalls zu Initialisierungszwecken ausgewertet. Im Gegensatz zu der Datei `.profile`, welche nur von Login-Shells gelesen wird, wird die über die Environmentvariable ENV vereinbarte Datei von jeder Korn-Shell ausgewertet. In dieser Datei sollten deshalb Definitionen von Alias-Namen und Functions vorgenommen werden. In der Regel heißt diese Datei `.kshrc`, wie auch auf den vorangehenden Seiten vorausgesetzt, als es um die Definition von Alias-Namen ging.

11.8 Ein-/Ausgabeumlenkung

Obwohl auf jeder Shell verfügbar, ist das Feature der Ein-/Ausgabeumlenkung shellspezifisch. Das liegt daran, daß zwei unterschiedliche Konzepte verfolgt werden:

1. Das Konzept der `sh`, der `ksh` und der `bash`
2. und das der `csh` und der `tcsh`.

Das Konzept der `csh` ist nicht so flexibel wie das der `sh` und kann nur einen Überschreibschutz als Plus verbuchen, über den allerdings die Korn-Shell auch verfügt. Doch bevor in die Details eingestiegen werden soll, sollte zunächst einmal das Grundkonzept erläutert werden.

Grundidee der Ein-/Ausgabe unter Unix ist die Verwendung von speziellen *Kanälen*. Diese Kanäle sind verbunden mit Geräten, die wiederum durch Dateien – die Device Descriptoren – repräsentiert werden. Da die Schnitt-

stelle für den Zugriff auf Geräte identisch mit der für den Zugriff auf Dateien ist (gemäß dem Grundprinzip der Geräteunabhängigkeit), liegt nichts näher, als die vordefinierten Kanäle umzulenken, wenn z.B. Ausgaben nicht auf dem Bildschirm, sondern in eine Datei geschrieben werden sollen. Genau das macht die Ein-/Ausgabeumlenkung. Drei Standardkanäle sind vordefiniert und können umgelenkt werden. Die drei Standardkanäle können über ihre »Kanalnummern« – die sogenannten Dateideskriptoren – angesprochen werden.

Die *Standardeingabe* ist per Default mit der Tastatur verbunden. Liest ein Kommando von der Standardeingabe, liest es normalerweise von der Tastatur. Der Dateideskriptor für die Standardeingabe ist die 0.

Die *Standardausgabe* ist mit dem Bildschirm verbunden. Alle Ausgaben, die ein Kommando tätigt und welche die Ergebnisse einer fehlerfreien Befehlsausführung sind, werden über die Standardausgabe ausgegeben. Der Dateideskriptor für die Standardausgabe ist die 1.

Die *Standardfehlerausgabe* ist wie die Standardausgabe ebenfalls mit dem Bildschirm verbunden. Über diesen Kanal werden in der Regel alle Fehlermeldungen von Befehlen auf den Bildschirm geschrieben. Die Existenz dieses Fehlerkanals wird in der Regel vom Benutzer zunächst nicht erkannt, da die Fehlermeldungen wie auch die regulären Ausgaben auf dem Bildschirm erscheinen. Der Dateideskriptor für die Standardfehlerausgabe ist die 2.

Eingabeumlenkung

Bei der *Eingabeumlenkung* werden die Eingaben für einen Befehl in einer Datei vorbereitet. Würde der Befehl von der Tastatur lesen, würde das Ende der Eingaben an der Übermittlung eines Control-D (ASCII-Wert 4) erkannt, das unter Unix mit dem Dateiendezeichen identisch ist. Bei der Eingabeumlenkung sorgt das Ende der Datei somit dafür, daß der Befehl keine weiteren Eingaben erwartet.

Unix-Kommando < Eingabedatei

Bei der Angabe der Eingabedatei dürfen keine Wildcards benutzt werden, da die Umlenkung von der Shell vor Interpretation der Wildcards aus der Kommandozeile gefiltert wird. Die Eingabe kann nur einmal pro Befehl umgelenkt werden.


```
UNIX-Kurs [~] 110 > mail Lutz.Brockmann@yasc.de < brief.txt
```

Ein klassisches Beispiel für die Verwendung der Eingabeumlenkung ist der `mail`-Befehl, mit dem elektronische Post auf der lokalen Maschine, aber auch in die weite Welt verschickt werden kann. In unserem Beispiel wurde der Text für die E-Mail in der Datei `brief.txt` vorbereitet (z.B. mit einem Texteditor) und wird nun dem Kommando `mail` per Eingabeumlenkung zur Verfügung gestellt.

Ausgabeumlenkung

Bei der *Ausgabeumlenkung* werden die Ergebnisse einer fehlerfreien Befehlsausführung in eine Datei geschrieben. Dabei kann eine Datei entweder neu angelegt oder überschrieben werden:

Unix-Kommando > *Ausgabedatei*

oder es können die Ausgaben an eine bereits bestehende Datei angehängt werden (append):

Unix-Kommando >> *Ausgabedatei*

Beide Varianten bergen ein gewisses Risiko in sich. Die Shell prüft nämlich nicht, was zuvor in der Umlenkungsdatei stand, d.h., es kann mit der Ausgabeumlenkung durchaus Text an eine Binärdatei angehängt bzw. ein Programm oder eine andere wichtige Datei per Ausgabeumlenkung überschrieben werden. Sollen Ausgaben an eine noch nicht bestehende Datei angehängt werden, wird diese von der Shell angelegt.

Alle drei aufgezählten Verhaltensformen sind im allgemeinen nicht erwünscht. Allerdings läßt sich nur gegen zwei davon etwas tun: gegen ungewolltes Überschreiben und gegen ungewolltes Anlegen einer Datei beim append. Erreicht wird dieser Schutz durch das Setzen der *Schreibschutzvariable* der `ksh`. Diese heißt `noclobber` und verhindert genau diese beiden Fälle.


```
UNIX-Kurs [~] 111 > ls -l >verzeichnis.txt
UNIX-Kurs [~] 112 > noclobber=
UNIX-Kurs [~] 113 > ls -l >verzeichnis.txt
verzeichnis.txt - File exists
```

Stellt die Shell bei gesetztem `noclobber` fest, daß eine der beiden Schutzvorschriften verletzt ist, sorgt sie dafür, daß das Kommando nicht ausgeführt wird – also unterbleibt nicht nur die Umlenkung, sondern der ganze Befehl wird nicht ausgeführt!

Ausgabeumlenkung mit Fehlerkanal

Bisher wurde nur die Umlenkung der Standardeingabe und -ausgabe beschrieben, was auch die beiden häufigsten Anwendungsfälle sind. Aus diesem Grund spielen auch die bereits erwähnten Dateideskriptoren noch keine Rolle, da die beiden Varianten auch ohne zusätzliche Kanalkennungen an der Richtung des Umlenkungsymbols zu erkennen sind. Wenn allerdings Gebrauch von der Ausgabeumlenkung gemacht wird, kann festgestellt werden, daß, obwohl die Ausgabe in eine *Ausgabedatei* umgelenkt ist, ab und zu dennoch Ausgaben auf dem Bildschirm erscheinen. Dabei handelt es sich um solche Meldungen, die über die Standardfehlerausgabe auf den Bildschirm gelangen. Soll dies unterbunden werden, kann unter der `ksh` die Standardfehlerausgabe entweder getrennt in eine *Fehlerdatei* ausgegeben werden oder *Standardausgabe und Standardfehlerausgabe* werden gemeinsam in eine Datei umgelenkt. Auch hier gibt es wieder die Varianten des Überschreibens bzw. Anlegens:

Unix-Kommando >Ausgabedatei 2>Fehlerdatei

Unix-Kommando >Ausgabedatei 2>&1

und die des Anhängens:

Unix-Kommando >>Ausgabedatei 2>>Fehlerdatei

Unix-Kommando >>Ausgabedatei 2>&1


```
UNIX-Kurs [~] 114 > find / -name .kshrc -print >rcs.txt 2>&1
```

In diesem Beispiel wird der gesamte Dateibaum des Unix-Betriebssystems nach Dateien mit dem Namen `.kshrc` durchsucht. Das Ergebnis dieser Suche wird in die Datei `rcs.txt` im Working Directory geschrieben. Da `find` Fehlermeldungen produziert, wenn es auf Verzeichnisse stößt, für die der Benutzer kein Ausführungsrecht oder Leserecht besitzt (mit solchen Verzeichnissen ist bei der Durchsuchung des kompletten Dateibaums zu rechnen), werden Standard- und Standardfehlerausgabekanal umgelenkt.

Die Korn-Shell bietet im Gegensatz zur csh und tcsh allerdings auch die Möglichkeit, nur die Standardfehlerausgabe (Dateidescriptor 2) umzulenken. Auch hier kann ein Beispiel am besten zeigen, wie das funktioniert.

```
UNIX-Kurs [~] 115 > find / -name .kshrc -print 2>/dev/null
```

In diesem Beispiel werden die Fehlerausgaben, die der find-Befehl produziert, wenn er in bestimmte Verzeichnisse nicht wechseln darf, einfach auf das NULL-Device umgelenkt. Alles was auf dieses Device umgelenkt wird, wird einfach weggeschmissen.

Es erscheinen also nur die gültigen Suchergebnisse des find-Kommandos auf dem Bildschirm.

11.9 Subshells

Wie auch die csh und die tcsh ist die Korn-Shell in der Lage, Kommandos oder Kommandofolgen in einer eigenen sogenannten Subshell zu starten. Diese Umhüllung durch eine Shell führt dazu, daß z.B. die Ausgaben der Kommandofolge zusammengefaßt werden und dann entweder mit der Ein-/Ausgabeumlenkung gemeinsam behandelt oder per Pipe an ein Filterkommando weitergeleitet werden können.

Eine Subshell wird dann für ein Kommando oder eine Kommandofolge gestartet, wenn sie durch runde Klammern eingeschlossen werden. Die Syntax ist dabei identisch mit der in der csh und tcsh.

```
(Kommando1;Kommando2;...)
```

Im Gegensatz zur csh und tcsh, wo die Subshell oft Verwendung findet, um die in diesen Shells fehlende vollständige Trennung von Standardausgabe- und Standardfehlerausgabeumlenkung zu kompensieren, ist dies in der Korn-Shell nicht nötig. Wie im vorangegangenen Abschnitt beschrieben, lassen sich in der Korn-Shell Standardausgabe und Standardfehlerausgabe separat umlenken und bedürfen deshalb keiner Subshell für diese Aufgabe.

Dennoch gibt es auch in der Korn-Shell einige sinnvolle Anwendungsbeispiele für die Funktion der Subshell.

```
UNIX-Kurs [~] 116 > (cd /etc;echo "Inhalt /etc";ls) > dir.txt
```

Dieses Beispiel zeigt, wie die Ausgaben von mehreren Unix-Kommandos mit nur einer Ausgabeumlenkung in eine Datei geschrieben werden können.

Die Verwendung des `cd` innerhalb der runden Klammern der Subshell führt dazu, daß das Working Directory der Subshell vor der Ausführung der beiden folgenden Befehle auf das Verzeichnis `/etc` gesetzt wird. Diese Änderung des Working Directory hat keinen Einfluß auf das Working Directory der Shell, von der die Kommandofolge in den runden Klammern abgesetzt wird. Das Verhalten bei einer Änderung von Environmentvariablen innerhalb der runden Klammern wäre identisch – die Auswirkungen der Änderungen blieben auf die Subshell begrenzt und hätten keinen Einfluß auf die aufrufende Shell.

In dieser Hinsicht verhalten sich Subshells also genauso wie jene Shell, die zur Interpretation eines Shell-Scripts gestartet werden.

11.10 Von der Eingabe eines Kommandos bis zur Ausführung

Nachdem nun die vielen unterschiedlichen Aufgaben der Shell bei der Ausführung eines Kommandos besprochen worden sind, stellt sich nur noch die Frage nach der Reihenfolge, die ja unter Umständen ausschlaggebend für das Ergebnis des Kommandos sein kann. Die `ksh` arbeitet dabei nach den in Abb. 11.1 dargestellten Schritten.

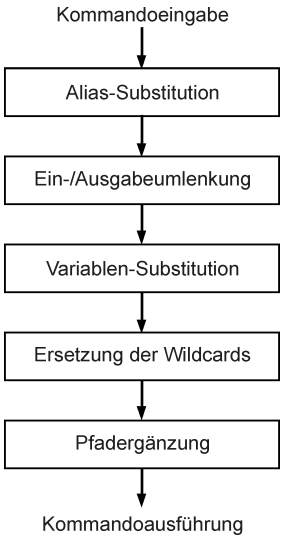

Abb.11.1: Arbeitsschritte der `ksh`

Nachdem die Kommandozeile eingegeben und mit ↵ zur Ausführung freigegeben worden ist, wird von der Shell zuerst die *Alias-Substitution*

vorgenommen Bei dieser Ersetzung werden vom Benutzer definierte Kürzel (Alias-Namen) durch die eigentliche Schreibweise der Befehle ersetzt.

Nach der Interpretation der Ein-/Ausgabeumlenkung erfolgt die *Variablen-Substitution*. Bei dieser werden Zeichenketten der Form `$xxx`, wobei für *xxx* ein Variablenname aus dem Environment oder der Shell stehen kann, durch den Inhalt der entsprechenden Variablen ersetzt.

Anschließend wird die so veränderte Eingabezeile nach *Ein-/Ausgabeumlenkungszeichen* durchsucht. Mit diesem Feature können die Ausgaben eines Befehls vom Bildschirm in Dateien umgelenkt und die Eingaben eines Befehls statt von der Tastatur aus einer Datei gelesen werden.

Als vorletzter Schritt werden die *Wildcards* in der Eingabezeile interpretiert. Die in der Kommandozeile enthaltenen Muster werden dabei – soweit möglich – durch passende Dateinamen ersetzt.

Der letzte Schritt vor der Kommandoausführung ist die Zuordnung von Pfaden zu den Kommandos, die in der Kommandozeile stehen. Zu diesem Zweck wird der Suchpfad aus dem Environment benutzt. Kann eine Zuordnung zu einem Pfad vollzogen werden, wird das entsprechende Programm mit der in den vorherigen Schritten ermittelten Kommandozeile von der Shell gestartet.

11.11 Prozeßverwaltung der Shell

Als letztes Feature der Korn-Shell soll ihre erweiterte Prozeßverwaltung vorgestellt werden. Grundsätzlich lassen sich auch wie unter den anderen Shells Vordergrund- und Hintergrundprozesse erzeugen. Die Kommandos `ps` und `kill` sind nach wie vor zur Anzeige und Beendigung von Prozessen verfügbar. Allerdings haben diese beiden Kommandos Schwächen, welche die Entwickler der `ksh` dazu bewogen haben, Erweiterungen einzuführen.

Werden Kommandos sequentiell (verknüpft mit »;«) oder parallel (verknüpft mit »|«) verkettet, können oft nicht alle Prozesse gleichzeitig mit `kill` beendet werden. Insbesondere die sequentielle Verkettung bereitet hier Probleme, weil für die noch nicht gestarteten Prozesse noch keine PIDs bekannt sind. Um diesem Problem beizukommen, werden unter der `ksh` verkettete Kommandos über eine extra zu diesem Zweck gestartete Shell verwaltet. Diese Shell bildet den Vaterprozeß für alle Kommandos. »Stirbt« diese Shell, »sterben« auch alle Kommandos der Verkettung mit ihr. Durch diesen Trick muß nur ein Prozeß kontrolliert werden, der mit Sicherheit über eine PID verfügt. Einen Einzelprozeß oder eine mehrere Kommandos kontrollierende Shell wird unter der `ksh` als *Job* bezeichnet.

Da aber die PIDs auf Systemen mit vielen Prozessen mit `ps` nur recht umständlich zu ermitteln sind, wurden die *Job-IDs* eingeführt. Jeder Job, der gestartet wird, bekommt, egal ob es sich um ein Einzelkommando oder um eine Verkettung mit kontrollierender Shell handelt, eine eindeutige Job-ID von der Shell zugeteilt.

jobs

Mit dem internen Kommando `jobs` der Korn-Shell können alle von dieser Shell aus gestarteten Hintergrundprozesse mit ihren Job-IDs aufgelistet werden. Angezeigt werden die Kommandozeile, mit der der Befehl gestartet wurde, und der Zustand, in dem er sich gerade befindet.

Soll nun ein Job beendet werden, so kann dies über das `kill`-Kommando und die zugehörige Job-ID mit einem vorangestellten Prozentzeichen »%« geschehen.

```
UNIX-Kurs [~] 117 > kill %1
```

In diesem Beispiel wird dem Job mit der Job-ID 1 mit `kill` das Signal `SIGTERM` zugestellt.

`csh` und `tcsh` bieten aber noch mehr in Sachen *Job Control*. So lassen sich z.B. gerade laufende *Vordergrundprozesse* mit Strg Z *stoppen*, um sie später fortzusetzen.

bg

background – Die Fortsetzung kann mit dem Kommando `bg` als Hintergrundprozeß erfolgen. Diese Variante wird häufig benutzt, wenn Rechenprozesse zu Beginn ihrer Arbeit interaktiv vom Benutzer Angaben abfragen, um dann minuten- oder gar stundenlang vor sich hinzurechnen. Ein solcher Prozeß wird als Vordergrundprozeß gestartet, dann werden interaktiv die gewünschten Angaben gemacht, und der Prozeß wird gestoppt und im Hintergrund fortgeführt. Als Ergebnis des Kommandoaufrufs werden die Job-ID in eckigen Klammern und die PID ausgegeben. `bg` kann auch als zusätzlichen Parameter eine Job-ID erhalten, wenn mehr als ein Job gestoppt wurde. Diese ist dann mit vorangestelltem Prozentzeichen anzugeben.

```
UNIX-Kurs [~] 118 > find / -name "*.c" -print >& result.txt
```

Strg Z

```
UNIX-Kurs [~] 119 > bg
 [1]    52314
```

Das Durchsuchen des gesamten Dateibaums nach Dateien mit der Endung `.c` wie in dem obigen Beispiel kann je nach Größe des Dateibaums recht lange dauern. Aus diesem Grund wurde das als Vordergrundprozeß gestartete Kommando in diesem Beispiel mit Strg Z gestoppt und mit dem Kommando `bg` als Hintergrundprozeß fortgesetzt.

`fg`

foreground – Ein gestoppter Prozeß kann aber auch als Vordergrundprozeß fortgesetzt werden. Das Stoppen eines Prozesses wird nämlich nicht grundsätzlich »von Hand« durch den Benutzer vorgenommen, sondern kann auch durch die Shell selbst erfolgen, z.B. wenn ein Hintergrundprozeß versucht, von der Standardeingabe zu lesen. Daß ein Prozeß gestoppt wurde, wird in der Liste der Prozesse, die mit `jobs` abgefragt werden kann, angezeigt. Da in der Regel nicht immer der zuletzt gestartete Job angehalten wurde, sollte bei der Verwendung von `fg` immer mit einer Job-ID gearbeitet werden.

```
UNIX-Kurs [~] 120 > fg %2
```

In diesem Beispiel wird der Job mit der Job-ID 2, der bisher im Hintergrund lief oder gestoppt war, als Vordergrundprozeß weitergeführt.

11.12 Übungen

11.12.1 Aufgaben

1. Suchen Sie nach einer Möglichkeit, den folgenden `find`-Befehl statt mit `-exec` mit einer Kommandosubstitution (Back Quotes) zu lösen.

   ```
   find ~ -type f -exec file {} \;
   ```

2. Welches der beiden Kommandos aus 1.) ist schneller – die Variante mit `-exec` oder die Kommandosubstitution? Begründen Sie Ihre Antwort.
3. Kann es bei der Kommandosubstitution aus 1.) zu Problemen kommen?
4. Versuchen Sie Beispiele zu finden, welche die Unterschiede zwischen Double und Single Quotes verdeutlichen.
5. Definieren Sie einen Alias `dir` für das Kommando `ls -la`.
6. Erstellen Sie eine Function mit dem Namen `wer_ist`, die Ihnen zu einer Benutzerkennung das zugehörige fünfte Feld aus der Datei `/etc/passwd` (den Kommentar) ausgibt.

7. Schauen Sie sich den Inhalt der Startup-Dateien Ihrer Shell an.
8. Versuchen Sie, mit Hilfe von `echo` und der Ausgabeumlenkung, einen Brief in eine Datei zu schreiben.
9. Starten Sie einen `find`-Befehl als Hintergrundprozeß, und lenken Sie die Suchergebnisse in eine Datei um. Werden wirklich alle Ausgaben in die Datei umgelenkt? Was müssen Sie tun, damit auf keinen Fall Ausgaben auf dem Bildschirm erscheinen?

11.12.2 Lösungen

1. Zur Lösung schlagen Sie bitte in dem vorangegangenen Kapitel zur C-Shell nach. Es bestehen keine Unterschiede.
2. Gleiches gilt für die Übungsaufgaben 2 und 3.
4. Um den Umgang mit den diversen Quoting-Mechanismen zu üben, eignet sich das `echo`-Kommando hervorragend. Raten Sie, vollziehen Sie nach und vor allem: Probieren Sie selbst. Nach ein paar Übungsrunden dürfte das Quoting seinen Schrecken verloren haben.

   Hier ein paar Beispiele für die Korn-Shell.

   Definieren Sie zwei Shell-Variablen:

   ```
   UNIX-KURS [~] 50 > export VORNAME=Albert
   UNIX-KURS [~] 51 > export NACHNAME=Nagel
   ```

   Ohne das $-Zeichen werden die Variablennamen selbst ausgegeben:

   ```
   UNIX-KURS [~] 52 > echo VORNAME NACHNAME
   VORNAME NACHNAME
   ```

   Double Quotes erhalten Leerzeichen:

   ```
   UNIX-KURS [~] 53 > echo "$VORNAME        $NACHNAME"
   Albert          Nagel
   UNIX-KURS [~] 54 > echo $VORNAME         $NACHNAME
   Albert Nagel
   ```

   Single Quotes hingegen verhindern das Auswerten von Variablen:

   ```
   UNIX-KURS [~] 55 > echo '$VORNAME        $NACHNAME'
   $VORNAME        $NACHNAME
   ```

Back Quotes dienen der Kommandosubstitution, das heißt, der darin stehende Unix-Befehl wird ausgeführt:

```
UNIX-KURS [~] 56 > echo pwd
pwd
UNIX-KURS [~] 57 > echo ‘pwd‘
/home/lutz
```

Im Gegensatz zu Double Quotes verhindern Single Quotes die Kommandosubstitution:

```
UNIX-KURS [~] 58 > echo "‘pwd‘"
/home/lutz
UNIX-KURS [~] 59 > echo '‘pwd‘'
‘pwd‘
```

5. Die Syntax für alias lautet: `alias` *Alias-Name*="*Kommando*"

   Als *Alias-Namen* wollen Sie `dir` benutzen und ihm das *Kommando* `ls -la` zuweisen. Also lautet die Lösung:

   ```
   UNIX-KURS [~] 60 > alias dir="ls -la"
   ```

   Die Anführungszeichen sind an dieser Stelle zwingend notwendig (im Gegensatz zum Alias-Kommando der C-Shell). Von nun an wird, immer dann, wenn Sie `dir` eintippen, von der Korn-Shell das Kommando `ls -la` abgesetzt. Noch eine Anmerkung dazu: Der Alias-Befehl ist kein Unix-Kommando, sondern ein Befehl, der in dem Kommandointerpreter implementiert ist. Daher ist es auch verständlich, daß die Syntax des Befehls von dem Kommandointerpreter abhängig ist. Außerdem kann darum das Unix-Kommando `which` die entsprechende ausführbare Datei nicht finden.

6. Bevor wir die Function erstellen, sollten wir uns zunächst einmal überlegen, welcher Unix-Befehl bzw. welche Kombination von Unix-Befehlen die eigentliche Aufgabenstellung – nämlich das Extrahieren des Kommentars für eine Benutzerkennung – lösen kann.

   Dazu wollen wir zunächst den Kommentar für die Benutzerkennung `root` aus der Datei `/etc/passwd` filtern. Die gesuchte Zeile liefert uns der Befehl `grep root /etc/passwd`.

   Wird der Befehl so abgesetzt, werden allerdings sämtliche Zeilen ausgegeben, die die Buchstabenfolge `root` enthalten, egal ob im Kommentar oder als Teil der Benutzerkennung oder des Heimatverzeichnisses. Sicherer ist es, den Befehl folgendermaßen zu formulieren:

   ```
   grep ^root: /etc/passwd
   ```

Durch das vorangestellte Sonderzeichen ^ wird nach der Zeichenkette nur am Anfang jeder Zeile der Datei gesucht. Der nachgestellte Doppelpunkt integriert das Trennzeichen der Paßwortdatei in das Suchermuster, wodurch verhindert wird, daß auch eine Zeile für eine Benutzerkennung der Form `rootmail` ausgegeben wird.

Von der ausgegebenen Zeile, die etwa folgendes Aussehen haben wird

```
root:x:0:0:UNIX-Systemadministrator:/home/root:/bin/sh
```

interessiert uns laut Aufgabenstellung nur der Kommentar im fünften Feld.

Aus dem Ergebnis des `grep`-Kommandos ist daher dieses Feld herauszuschneiden. Diese Aufgabe können wir getrost dem Kommando `cut` überlassen. Es findet Felder in einer Zeile anhand eines über die Option `-d` vereinbarten Trennzeichens und kann einzelne Felder aus der Standardeingabe oder einer Datei herausfiltern. Das Ergebnis des `grep`-Befehls ist also über eine Pipe an die Standardeingabe weiterzuleiten. Der zusammengesetzte Befehl lautet:

```
grep ^root: /etc/passwd | cut -d: -f5
```

Damit sind wir nun fast am Ziel unserer Wünsche. Wir müssen nur noch eine Function definieren, die genau diesen komplexen Befehl ausführt. Allerdings soll nicht immer nach der Benutzerkennung `root` gesucht werden, sondern die Benutzerkennung soll an die Function `wer_ist` übergeben werden. Wir wünschen uns also einen Aufruf der Form `wer_ist lutz` und `wer_ist root`. Schwierig ist das nicht; es ist nur die Zeichenkette `root` in dem Befehl durch das Platzhalterzeichen `$1` für den ersten Aufrufparameter zu ersetzen. Die Function wird damit folgendermaßen definiert:

```
function wer_ist {
    grep ^$1: /etc/passwd | cut -d: -f5
}
```

Der Befehl kann in der angegebenen Form mit den Zeilenumbrüchen eingegeben werden. Nach dem ersten Betätigen der ↵-Taste erhalten Sie einen anderen Prompt. Der Prompt der Shell erscheint wieder, nachdem die schließende Klammer eingegeben wurde.

Die von Ihnen erstellte Function ist nun in der Shell, in der Sie den Befehl abgesetzt haben, verfügbar. Um die Function permanent zu machen, muß die Definition in eine Startup-Datei der Korn-Shell, im all-

gemeinen in die Datei `.profile` in ihrem Heimatverzeichnis, eingetragen werden.

7. Die Standard-Startup-Datei der Korn-Shell ist die Datei `.profile` im Heimatverzeichnis des jeweiligen Benutzers. Diese wird allerdings nur für die Login Shells ausgeführt. Sobald eine Korn-Shell aufgerufen wird (dies kann entweder explizit durch den Aufruf `ksh` geschehen oder automatisch beim Öffnen eines Shell-Fensters, wenn die Korn-Shell als Login-Shell für den Benutzer in der Datei `/etc/passwd` eingetragen ist), wertet die Shell die Environmentvariable ENV aus und führt die Datei aus, deren Name in ENV vermerkt ist. In dieser Startup-Datei werden dann in der Regel Umgebungsvariablen exportiert, insbesondere der Suchpfad, Alias-Namen und Functions definiert sowie Shell-Variablen gesetzt.

   Eine Startup-Datei `.profile` könnte folgenden Inhalt haben:

```
export ENV=$HOME/.kshrc
export HISTSIZE=100
export PS1="UNIX-Kurs[‘pwd‘] >"
PATH=$PATH:/sbin:/usr/bin:/usr/local/bin:/etc:/usr/bsd
PATH=$PATH:/usr/proc/bin:/usr/sbin:$HOME/bin
MANPATH=/usr/catman:/usr/share/catman
export PATH MANPATH

. /home/db2inst2/sqllib/db2profile
```

   In der ersten Zeile der Datei wird vereinbart, daß die Datei `.kshrc` im Heimatverzeichnis als zweite Startup-Datei ausgeführt wird. Die zweite Zeile bewirkt, daß die letzten 100 abgesetzten Befehle in der Befehlshistorie zur Verfügung stehen. Die Umgebungsvariable PS1 bestimmt das Aussehen des Prompts. Schließlich werden der Suchpfad (Umgebungsvariable PATH) und der Suchpfad für die Manual-Pages (Umgebungsvariable MANPATH) gesetzt und exportiert. Der Suchpfad wird in dem Beispiel nicht neu definiert, sondern nur erweitert. Dies verhindert, daß Voreinstellungen, die der Systemadministrator in der globalen Initialisierungsdatei der Korn-Shell (üblicherweise ist dies die Datei `/etc/profile`) gemacht hat, überschrieben werden.

   Ohne den `export`-Befehl werden die Umgebungsvariablen PATH und MANPATH nicht auf die zuvor zugewiesenen Werte gesetzt, und die zuvor definierten Variablen bleiben nur Shell-Variablen. Dieser Effekt ist allerdings nur dann sichtbar, wenn die Shell-Variablen nicht von Umgebungsvariablen überdeckt werden. Die letzte Zeile beginnt mit einem Punkt, dem durch ein Leerzeichen getrennt ein Dateiname folgt. Dieser

Befehl bewirkt, daß die angegebene Datei von der aktuellen Shell interpretiert wird. Das heißt, es wird keine Shell zur Ausführung der Datei gestartet; man sagt, die Datei wird »gesourct«. Werden in der Datei Umgebungsvariablen gesetzt, so verändert dies die Umgebung der aktuellen Shell. Dies wäre nicht der Fall, wenn zur Ausführung der Datei eine neue Shell gestartet würde. Für Programmierer: Dieser Mechanismus ist vergleichbar mit einem `#include` in einem C-Programm.

In der Datei `.kshrc` könnten sich Einträge der folgenden Form befinden:

```
alias dir="ls -la"

function ff {
   find / -name "$1" -print 2>/dev/null
}
```

Hier werden der Alias `dir` für das Kommando `ls -la` und eine Funktion `ff` definiert. Der Funktion ist beim Aufruf ein Dateiname als Parameter mitzugeben. Der find-Befehl in dieser Function durchsucht dann den gesamten Unix-Dateibaum nach Dateien mit diesem Namen. Beachtenswert ist die Umlenkung eventueller Fehlerausgaben (`2>`) auf das sogenannte Null-Device `/dev/null`. Dieses Device schluckt alle Ausgaben; sie werden weder auf dem Bildschirm ausgegeben noch irgendwo gespeichert.

8. Zur Lösung siehe die entsprechende Aufgabe im vorangegangenen Kapitel zur C-Shell.

9. Wie Bildschirmausgaben in eine Datei umgelenkt werden, haben wir schon in Aufgabe 8 gesehen. Sollen wirklich keine Ausgaben mehr auf dem Bildschirm erfolgen, so müssen auch die eventuell auftretenden Fehlermeldungen in eine Datei umgelenkt werden.

   Schauen wir uns folgenden Befehl an:

   ```
   UNIX-KURS [~] 68 > find / -name core -print > ergebnis.txt
   ```

   Dieser Befehl durchsucht den gesamten Dateibaum nach Dateien mit dem Namen `core`. Die Pfadnamen aller gefundenen Dateien werden dann in eine Datei namens `ergebnis.txt` gesichert. Setzen Sie diesen Befehl ruhig einmal so ab. Sie werden schnell merken, daß Sie trotz der Umlenkung eine Menge Ausgaben auf Ihren Bildschirm erhalten: Da Sie bei der Suche *alle* Verzeichnisse durchforsten wollen, bekommen Sie Fehlermeldungen für jedes Verzeichnis, für das Sie keine Zugriffsrechte haben. Und das dürften im Normalfall, d.h., solange Sie nicht über

Superuser-Berechtigung verfügen, einige sein. Sollen diese Fehlermeldungen in eine separate Datei umgelenkt werden, so geben Sie einen Dateinamen für den Dateideskriptor 2 (Standardfehlerausgabe) an:

```
UNIX-KURS [~] 69 > find / -name core -print > ergebnis.txt
2> err.txt
```

Sollen die Fehlermeldungen mit in der Datei `ergebnis.txt` protokolliert werden, so muß die Standardfehlerausgabe (2) auf die Standardausgabe (1) verweisen, die ja bereits auf die Datei `ergebnis.txt` umgelenkt ist:

```
UNIX-KURS [~] 70 > find / -name core -print > ergebnis.txt
2>&1
```

Beachten Sie, daß vor dem `&1` kein Leerzeichen stehen darf. Der Befehl ist sonst syntaktisch nicht korrekt.

Unter Umständen interessieren Sie die Fehlermeldungen überhaupt nicht. Dann gibt es eine sehr elegante Lösung, diese Meldungen zu »vernichten«: Alle Daten, die auf das Null-Device `/dev/null` umgelenkt werden, erscheinen weder auf der Standardausgabe noch werden sie irgendwo gespeichert.

```
UNIX-KURS [~] 71 > find / -name core -print > ergebnis.txt
2> /dev/null
```

Da diese Suche relativ lange dauern kann, empfiehlt es sich, das Kommando als Hintergrundprozeß zu starten. Andernfalls erscheint Ihr Eingabe-Prompt erst dann wieder, wenn der Befehl komplett abgearbeitet ist. Um ein Kommando im Hintergrund ablaufen zu lassen, geben Sie am Ende der Eingabezeile ein Ampersand (`&`) ein:

```
UNIX-KURS [~] 72 > find / -name core -print > ergebnis.txt
2> /dev/null &
```