5

Interaktive Shellskripte

Inhalt

5.1 Einleitung 78
5.2 Das Kommando `read` 78
5.3 Menüauswahl mit `select` 80
5.4 »Grafische« Oberflächen mit `dialog` 84

Lernziele

- Techniken zur Steuerung von Shellskripten erlernen
- Die Bash-Methoden zur Benutzerinteraktion kennen
- Mit dem `dialog`-Paket umgehen können

Vorkenntnisse

- Kenntnisse über Shellprogrammierung (aus den vorigen Kapiteln)

5.1 Einleitung

In den vorigen Kapiteln haben Sie gelernt, wie Sie Shellskripte schreiben können, die Dateinamen und andere Informationen über die Kommandozeile erhalten. Das ist in Ordnung für Personen, die an die Kommandozeile gewöhnt sind – »normale« Benutzer schätzen aber oft einen »interaktiveren« Stil, bei denen ein Skript Fragen stellt oder eine Menüoberfläche anbietet. In diesem Kapitel lernen Sie, wie Sie dies mit der Bash realisieren können.

5.2 Das Kommando `read`

Das Shell-Kommando `read` haben wir ja schon im Vorübergehen kennengelernt: Es liest Zeilen von seiner Standardeingabe und weist sie an Shellvariable zu, in Konstruktionen wie

```
grep … | while read line
do
    …
done
```

Sie können (wie gesehen) mehrere Variable angeben. Die Eingabe wird dann in »Wörter« aufgeteilt, und die erste Variable bekommt das erste Wort als Wert, die zweite das zweite und so weiter:

```
$ read h w
Hallo Welt
$ echo $w $h
Welt Hallo
```

Überzählige Variable bleiben leer:

```
$ read a b c
1 2
$ echo $c
                                                    Nichts
$ _
```

Wenn mehr Wörter existieren als Variable, bekommt die letzte Variable den ganzen Rest:

```
$ read a b
1 2 3
$ echo $b
2 3
$ _
```

(Das ist natürlich das Geheimnis hinter dem Erfolg von »`while read line`«.)

`read` verträgt sich nicht gut mit Pipelines: Probieren Sie mal

```
$ echo Hallo Welt | read h w
$ echo $h $w
                                                    Nichts!?
```

Der Grund dafür: Die einzelnen Kommandos einer Pipeline führt die Shell in Unterprozessen aus, und auf Shellvariable aus einem Unterprozess kann in der »Eltershell« nicht zugegriffen werden (Sie erinnern sich!). Schleifenkonstruktionen wie `while` und `for` genießen einen speziellen Dispens.

Was ist ein »Wort«? Die Shell verwendet auch hier als mögliche Trennzeichen den Inhalt der Variablen IFS (kurz für *internal field separator*), zeichenweise betrachtet. Der Standardwert von IFS besteht aus dem Leerzeichen, dem Tabulatorzeichen und dem Zeilentrenner, aber Sie können sich oft das Leben erleichtern, indem Sie einen anderen Wert vergeben:

```
IFS=":"
cat /etc/passwd | while read login pwd uid gid gecos dir shell
do
    echo $login: $gecos
done
```

spart Ihnen das Jonglieren mit cut.

In der Bash können Sie das Lesen von der Tastatur auch mit einer Eingabeaufforderung kombinieren, wie im folgenden Skript zum Anlegen eines neuen Benutzers: Eingabeaufforderung

```
#!/bin/bash
# newuser -- neuen Benutzer anlegen

read -p "Benutzername: " login
read -p "Bürgerlicher Name: " gecos

useradd -c "$gecos" $login
```

Gerade bei Verwendung von read ist es wichtig, die »eingelesenen« Variablen in Anführungszeichen zu setzen, um Probleme mit Leerzeichen zu vermeiden.

Was passiert, wenn der Aufrufer des newuser-Skripts keine gültige Eingabe macht? Wir könnten zum Beispiel fordern, dass der Benutzername des neuen Benutzers nur aus Kleinbuchstaben und Ziffern besteht (eine gängige Konvention). Umsetzen könnten wir das wie folgt: Plausibilitätskontrolle

```
read -p "Benutzername: " login

test=$(echo "$login" | tr -cd 'a-z0-9')
if [ -z "$login" -o "$login" != "$test" ]
then
    echo >&2 "Ungültiger Benutzername \"$login\""
    exit 1
fi
```

Hier betrachten wir, was übrig bleibt, wenn wir aus dem vorgeschlagenen Benutzernamen alle ungültigen Zeichen entfernen. Ist das Ergebnis ungleich dem ursprünglichen Benutzernamen, dann enthält letzterer »verbotene« Zeichen. Leer darf er auch nicht sein, was wir mit der -z-Bedingung von test prüfen.

Bei durchgreifenden Operationen wie dem Anlegen neuer Benutzer empfiehlt sich eine Sicherheitsabfrage am Schluss, damit der Aufrufer das Skript abbrechen kann, falls er »kalte Füße« bekommen hat (etwa wegen einer Fehleingabe weiter vorne). Eine bequeme Möglichkeit dazu ist über eine Shellfunktion der folgenden Form: Sicherheitsabfrage

```
function confirm () {
    while read -p "Bitte bestätigen (j/n): " answer
    do
        case $answer in
            [Jj]*) result=0; break ;;
            [Nn]*) result=1; break ;;
            *) echo "Bitte antworten Sie mit 'ja' oder 'nein'" ;;
        esac
```

```
    done
    return $result
}
```

Die Sicherheitsabfrage im Skript wäre dann etwas wie

```
confirm && useradd …
```

Übungen

5.1 [!1] Ändern Sie das `newuser`-Skript so, dass es prüft, ob der bürgerliche Name des neuen Benutzers einen Doppelpunkt enthält, und gegebenenfalls eine Fehlermeldung ausgibt und abbricht.

5.2 [2] Erweitern Sie das `newuser`-Skript so, dass der Aufrufer die Login-Shell des neuen Benutzers wählen kann. Sorgen Sie dafür, dass nur Shells aus `/etc/shells` akzeptiert werden.

5.3 [2] Erweitern Sie die Shellfunktion `confirm` so, dass sie die Eingabeaufforderung als Parameter übergeben bekommt. Wenn kein Parameter übergeben wurde, soll die Standardaufforderung »Bitte bestätigen« ausgegeben werden.

5.4 [3] Schreiben Sie ein einfaches Ratespiel: Der Computer wählt eine Zufallszahl zwischen (zum Beispiel) 1 und 100 (die Shellvariable `RANDOM` liefert Zufallszahlen). Der Benutzer darf eine Zahl eingeben und der Computer antwortet mit »Zu groß« oder »Zu klein«. Dies wird wiederholt, bis der Benutzer die richtige Zahl gefunden hat.

5.3 Menüauswahl mit `select`

Die Bash hat ein sehr mächtiges Kommando zur Auswahl von Optionen aus einer Liste, `select`, mit einer Syntax ähnlich der von `for`:

```
select ⟨Variable⟩ [in ⟨Liste⟩]
do
    ⟨Kommandos⟩
done
```

Diese Konstruktion beschreibt eine Schleife, bei der der Wert von ⟨*Variable*⟩ sich durch eine Benutzerauswahl aus ⟨*Liste*⟩ ergibt. Die Schleife wird immer wieder durchlaufen, bis das Dateiende auf der Standardeingabe erreicht ist. Sie können sich das etwa so vorstellen:

```
$ select typ in Hamburger Cheeseburger Fischburger
> do
>     echo $typ kommt
> done
1) Hamburger
2) Cheeseburger
3) Fischburger
#? 2
Cheeseburger kommt
#? 3
Fischburger kommt
#? Strg+D
$ _
```

Das heißt, die Bash präsentiert die Einträge der ⟨*Liste*⟩ mit vorgestellten Nummern, und der Benutzer kann über eine der Nummern eine Auswahl treffen.

`select` benimmt sich sehr wie `for`: Wird die ⟨*Liste*⟩ weggelassen, präsentiert es die Positionsparameter der Bash zur Auswahl. Die `select`-Schleife kann wie alle anderen Shell-Schleifen auch mit `break` oder `continue` abgebrochen werden.

`newuser` neu aufgegossen Wir können `select` verwenden, um unser `newuser`-Skript noch weiter zu verfeinern. Beispielsweise könnten Sie verschiedene Typen von Benutzern unterstützen wollen, an einer Universität etwa die Professoren, die Assistenten und die Studenten. In diesem Fall wäre es nützlich, wenn das `newuser`-Skript eine Auswahl der verschiedenen Benutzertypen anbieten würde. Aus der Gruppenwahl ergeben sich dann jeweils andere Voreinstellungen für die Benutzer, etwa die Unix-Gruppenzuordnung, das Heimatverzeichnis oder die Grundausstattung an Dateien im Heimatverzeichnis. Bei dieser Gelegenheit können Sie auch gleich noch eine weitere Möglichkeit dafür kennenlernen, Konfigurationsdaten für Shellskripte zu speichern.

Wir gehen davon aus, dass sich im Verzeichnis `/etc/newuser` für jeden Typ von Benutzern *t* eine Datei namens *t* befindet, für Professoren zum Beispiel `/etc/newuser/Professor` und für Studenten `/etc/newuser/Student`. Der Inhalt von `/etc/newuser/Professor` könnte zum Beispiel so aussehen:

```
# /etc/newuser/Professor
GROUP=profs
EXTRAGROUP=dekanat
HOMEDIR=/home/$GROUP
SKELDIR=/etc/skel-$GROUP
```

(Professoren werden in die Linux-Gruppe `profs` als primäre Gruppe gesteckt und bekommen außerdem `dekanat` als zusätzliche Gruppe). Das ist natürlich unser alter Trick »Konfigurationsdatei mit Shellvariablenzuweisungen«, mit dem Unterschied, dass es jetzt für jeden Benutzertyp eine eigene Konfigurationsdatei gibt. Das Anlegen eines neuen Benutzers, wenn wir den Benutzertyp wissen, geht dann ungefähr wie

```
confirm || exit 0

. /etc/newuser/$type
useradd -c "$gecos" -g $GROUP -G $EXTRAGROUP \
    -m -d $HOMEDIR/$login -k $SKELDIR $login
```

Uns bleibt noch die Auswahl des passenden Benutzertyps. Die Auswahlliste wollen wir natürlich nicht hart im `newuser`-Skript codieren, sondern vom Inhalt von `/etc/newuser` abhängig machen:

```
echo "Die folgenden Benutzertypen sind verfügbar:"
PS3="Benutzertyp: "
select type in $(ls /etc/newuser) '[Abbrechen]'
do
    [ "$type" = "[Abbrechen]" ] && exit 0
    [ -n "$type" ] && break
done
```

Die Shellvariable `PS3` gibt die Eingabeaufforderung an, die von `select` ausgegeben wird.

Übungen

5.5 [!1] Was passiert, wenn Sie bei `select` etwas eingeben, das nicht mit der Nummer eines Menüeintrags korrespondiert?

Wer wird ... Unser nächstes Skript ist lose an eine populäre Ratesendung im Fernsehen angenähert: Gegeben ist eine Datei `wwm.txt` mit Fragen und Antworten in der Form

```
0:?:Was hat die Morgenstund im Sprichwort?
0:-:Blei im Hintern
0:-:Eisen im Bauch
0:+:Gold im Mund
0:-:Silber im Kopf
0:>:50
50:?:Wovor sollten Sie sich beim Badeurlaub in Acht nehmen?
50:-:Hallowal
50:+:Haifisch
50:-:Huhurobbe
50:-:Grüezischildkröte
50:>:100
```

Das Skript – nennen wir es `wwm` – soll, beginnend beim Punktwert 0, die Fragen und Antworten präsentieren. Bei einer falschen Antwort endet das Programm, bei einer richtigen Antwort geht es mit der nächsten Frage weiter (deren Punktzahl sich aus der »:>:«-Zeile ergibt).

Abstraktion

Ein wichtiger Grundgedanke in komplizierteren Programmierprojekten ist **Abstraktion**. In unserem Fall versuchen wir, das konkrete Format der Fragendatei in einer Funktion zu »verstecken«, die die jeweils richtigen Elemente heraussucht. Theoretisch könnten wir dann später die Fragen statt einer einfachen Textdatei zum Beispiel einer Datenbank entnehmen oder die Datenhaltung auf andere Weise ändern (etwa indem wir pro Punktstufe zufällig eine Frage aus einer Mehrzahl auswählen). Eine mögliche, wenn auch nicht übermäßig effiziente Methode zum Zugriff auf die Fragendaten geht so:

```
qfile=wwm.txt

function question () {
    if [ "$1" = "get" ]
    then
        echo "$2"
        return
    fi
    case "$2" in
      display) re='?' ;;
      answers) re='[-+]' ;;
      correct) re='+' ;;
      next)    re='>' ;;
      *)       echo >&2 "$0: get: ungültiger Feldtyp $2"; exit 1 ;;
    esac
    grep "^$1:$re:" $qfile | cut -d: -f3
}
```

Die Funktion `question` muss zunächst in der Form

```
q=$(question get ⟨Punktzahl⟩)
```

aufgerufen werden. Sie liefert dann als Ausgabe die eindeutige Bezeichnung einer Frage mit der angegebenen Punktzahl (bei uns simpel – es gibt pro Punktzahl nur

eine Frage in der Datei, so dass wir einfach die Punktzahl als »Fragennummer« zurückgeben). Diese Bezeichnung merken wir uns in einer Shellvariablen (hier q). Anschließend stehen uns die folgenden Aufrufe zur Verfügung:

```
question $q display          liefert den Fragentext
question $q answers          liefert alle Antworten, eine pro Zeile
question $q correct          liefert die richtige Antwort
question $q next             liefert die Punktzahl bei richtiger Antwort
```

Alle Daten stehen auf der Standardausgabe zur Verfügung.

Für Kenner: Dies sind natürlich die Grundzüge eines »objektbasierten« Ansatzes – »question get« liefert ein »Fragenobjekt«, das dann die verschiedenen Methoden display usw. unterstützt.

Als nächstes brauchen wir eine Funktion, die eine Frage anzeigt und die Antwort abholt. Diese Funktion baut natürlich auf unserer eben gezeigten question-Funktion auf:

```
function present () {
    # Finde und zeige die Frage
    question $1 display
    # Finde die richtige Antwort
    rightanswer=$(question $1 correct)
    # Zeige die Antworten
    PS3="Ihre Antwort: "
    IFS=$'\n'
    select answer in $(question $1 answers)
    do
        if [ -z "$answer" ]
        then
            echo "Bitte geben Sie etwas Vernünftiges ein."
        else
            test "$answer" = "$rightanswer"
            return
        fi
    done
}
```

Die Antworten präsentieren wir natürlich mit select. Dabei müssen wir darauf achten, dass select die IFS-Variable benutzt, um beim Aufstellen des Menüs die einzelnen Menüpunkte voneinander zu trennen – mit dem Standardwert von IFS würde select jedes einzelne Wort aller Antworten als Menüpunkt anzeigen (!). Probieren Sie es aus! Für den Rückgabewert der Funktion nutzen wir die Tatsache aus, dass der Rückgabewert des letzten »echten« Kommandos (return zählt nicht) als Rückgabewert der Funktion gilt. Statt eines umständlichen

```
if [ "$answer" = "$rightanswer" ]
then
    return 0
else
    return 1
fi
```

verwenden wir also die oben gezeigte Konstruktion mit einem return nach einem test.

Als letztes bleibt uns noch der »Rahmen«, der die beiden getrennten Programmteile »Fragenverwaltung« und »Benutzungsoberfläche« zusammenbringt. Der könnte ungefähr so aussehen:

```
score=0
while [ $score -ge 0 -a $score -lt 1000000 ]
do
    q=$(question get $score)
    if present $q
    then
        score=$(question $q next)
    else
        score=-1
    fi
done
```

Das Rahmenprogramm kümmert sich darum, eine Frage auszusuchen (mit »question get«), die zur aktuellen Punktzahl des Teilnehmers passt. Diese Frage wird angezeigt (mit present) und in Abhängigkeit vom »Erfolg« der Anzeige (also der Richtigkeit der Antwort) wird die Punktzahl entweder auf die nächste Stufe erhöht, oder das Spiel ist zu Ende. Zum Schluss nur noch die warmen Abschiedsworte des (computerisierten) Showmasters:

```
if [ $score -lt 0 ]
then
    echo "Das war wohl nicht so prall, leider verloren"
else
    echo "Herzlichen Glückwunsch, Sie haben gewonnen"
fi
```

Übungen

5.6 [!1] Erweitern Sie das wwm-Skript so, dass es den »Punktwert« der Frage anzeigt (also die Punktzahl, die der Teilnehmer haben wird, wenn er die Frage richtig beantworten kann).

5.7 [!3] Überlegen Sie sich eine interessante Erweiterung von wwm und implementieren Sie diese (oder zwei oder drei).

5.8 [3] Überarbeiten Sie die question-Funktion von wwm so, dass sie mit weniger grep-Aufrufen auskommt.

5.4 »Grafische« Oberflächen mit dialog

Statt langweiliger Textmenüs und Fernschreiber-Dialogen können Sie in Ihren Skripten auch auf eine fast »grafische« Oberfläche zurückgreifen. Hierzu dient das Programm dialog, das Sie allerdings unter Umständen extra installieren müssen, wenn Ihre Linux-Distribution das nicht für Sie tut. dialog nutzt die Fähigkeiten moderner Terminals (oder Terminal-Emulationsprogramme), um Menüs, Auswahllisten, Texteingabefelder und ähnliches bildschirmfüllend und farbig zu präsentieren.

dialog kennt eine ganze Reihe von Interaktions-Elementen (Tabelle 5.1). Die Details der Konfiguration sind durchaus komplex und Sie sollten sie in der Dokumentation zu dialog nachlesen; wir beschränken uns hier auf das Allernötigste.

Wir können zum Beispiel unser wwm-Programm so ändern, dass die Fragen und die Abschlussauswertung mit dialog angezeigt werden. Für das Anzeigen unserer Fragen und Antworten ist das menu-Element am besten geeignet. Ein dialog-Aufruf für ein Menü sieht ungefähr so aus:

Tabelle 5.1: Interaktions-Elemente von dialog

	Beschreibung
calendar	Zeigt Tag, Monat und Jahr in getrennten Fenstern; der Benutzer kann editieren. Liefert den eingestellten Wert in der Form »Tag/Monat/Jahr«
checklist	Zeigt eine Liste von Einträgen, die individuell ein- und ausgeschaltet werden können. Liefert eine Liste der »eingeschalteten« Einträge
form	Stellt ein Formular dar. Liefert die eingetragenen Werte, einen pro Zeile
fselect	Zeigt einen Dateiauswahldialog. Liefert den gewählten Dateinamen
gauge	Zeigt einen Fortschrittsbalken
infobox	Gibt eine Nachricht aus (ohne den Bildschirm zu löschen)
inputbox	Erlaubt die Eingabe einer Zeichenkette, liefert diese
inputmenu	Zeigt ein Menü, bei dem der Anwender die Einträge umbenennen kann
menu	Zeigt ein Auswahlmenü
msgbox	Gibt eine Nachricht aus, wartet auf Bestätigung
passwordbox	inputbox, die die Eingabe nicht anzeigt
radiolist	Zeigt eine Liste von Einträgen, von denen genau einer ausgewählt sein kann; liefert den ausgewählten Eintrag
tailbox	Zeigt den Inhalt einer Datei, à la »tail -f«
tailboxbg	Wie tailbox, Datei wird im Hintergrund gelesen
textbox	Zeigt den Inhalt einer Textdatei
timebox	Zeigt Stunde, Minute und Sekunde mit Editiermöglichkeit; liefert Zeit im Format »Stunde:Minute:Sekunde«
yesno	Gibt eine Nachricht aus und erlaubt eine Antwort mit »Ja« oder »Nein«

Bild 5.1: Ein Menü mit dialog

```
$ dialog --clear --title "Menüauswahl" \
  --menu "Was darf es sein?" 12 40 4 \
      "H" "Hamburger" \
      "C" "Cheeseburger" \
      "F" "Fischburger" \
      "V" "Veggieburger"
```

Das Resultat sehen Sie in Bild 5.1. Die wichtigeren Optionen sind `--clear` (löscht den Bildschirm vor der Anzeige des Menüs) und `--title` (gibt einen Titel für das Menü an). Die Option `--menu` legt fest, dass dieser Dialog ein Auswahlmenü sein soll; danach folgen der erklärende Text für das Menü und die drei magischen Zahlen »Höhe des Menüs in Zeilen«, »Breite des Menüs in Zeichen« und »Anzahl von gleichzeitig angezeigten Einträgen«. Zum Schluss kommen die einzelnen Einträge, und zwar immer mit einem »Kurznamen« und dem eigentlichen Eintragsinhalt. Im angezeigten Menü können Sie über die Pfeiltasten navigieren, über die Zifferntasten [0] bis [9] und die Anfangsbuchstaben der Kurznamen; in unserem Beispiel würden etwa die Eingaben [3] und [f] zum »Fischburger« führen.

Ein weiterer Umstand ist wichtig im Umgang mit `dialog`: Das Programm liefert seine Ausgabewerte in der Regel auf der Standardfehlerausgabe. Das heißt, wenn Sie, was wahrscheinlich ist, die `dialog`-Ergebnisse auffangen und weiterverarbeiten wollen, müssen Sie die Standardfehlerausgabe von `dialog` umleiten, nicht die Standardausgabe. Das ergibt sich daraus, dass `dialog` auf seine Standardausgabe schreibt, um das Terminal anzusteuern; wenn Sie also die Standardausgabe umlenken, sehen Sie nichts mehr auf dem Bildschirm, und die eigentlichen Ausgabewerte von `dialog` würden unter diversen Terminalsteuerzeichen untergehen. Es gibt verschiedene Möglichkeiten, damit umzugehen: Sie können `dialog` über `2>` in eine temporäre Datei schreiben lassen, aber es ist mühselig, sich darum zu kümmern, dass diese temporären Dateien am Ende des Skripts wieder weggeräumt werden (Stichwort: `trap`). Alternativ dazu können Sie auch mit Ausgabeumlenkung kreativ werden, etwa so:

```
result=$(dialog … 2>&1 1>/dev/tty)
```

Dies verbindet die Standardfehlerausgabe (Dateideskriptor 2) mit dem Ziel der aktuellen Standardausgabe (die Variable `result`) und danach die Standardausgabe mit dem Terminal (`/dev/tty`).

Das funktioniert natürlich nur, weil die Standardausgabe auf anderen Geräten als Terminals nicht zu gebrauchen ist – wenn wir die Standardausgabe und die Standardfehlerausgabe eines Skripts vertauschen wollen, brauchen wir eine »Ringtauschkonstruktion« etwa der folgenden Form:

```
( programm 3>&1 1>output 2>&3 ) | …
```

Diese Kommandozeile leitet die Standardausgabe von `programm` in die Datei `output` und die Standardfehlerausgabe in die Pipeline. Sie ist damit quasi das Gegenteil zum gängigeren

```
programm 2>output | …
```

Wer wird … mit `dialog` Schauen wir uns jetzt eine `dialog`-basierte Version des `wwm`-Skripts an. Hier lohnt sich die Mühe, die wir vorher in Sachen »Abstraktion« betrieben haben; die Änderungen beschränken sich im Wesentlichen auf die Funktion `present`.

Die Haupthürde, die wir überspringen müssen, um `wwm` `dialog`-fähig zu machen, liegt darin begründet, dass in der Menüsyntax

```
dialog … --menu t h w n k0 e0 k1 e1 …
```

das dialog-Programm darauf besteht, die Menü-Kurznamen und -Einträge k_i und e_i als einzelne Wörter übergeben zu bekommen. Wir können zwar mit einer Schleife wie

```
items=''
i=1
question $q answers | while read line
do
    items="$items $i '$line'"
    i=$((i+1))
done
```

in der Variablen items leidlich einfach eine Liste von Kurznamen und Antworten der Form

```
1 'Blei im Hintern' 2 'Eisen im Bauch' …
```

konstruieren, aber einen Aufruf der Form

```
dialog … --menu 10 60 4 $items
```

mag dialog gar nicht gerne. Eine Lösung liegt in der Verwendung von **Feldern** (engl. *arrays*), die die Bash zumindest in eingeschränkter Form unterstützt. Felder

Felder Ein Feld ist eine Variable, die eine Folge von Werten enthalten kann. Diese Werte werden durch numerische Indizes angesprochen. Sie müssen ein Feld nicht gesondert vereinbaren, sondern es reicht, wenn Sie eine Variable »indiziert« ansprechen:

```
$ gang[0]=Aperitif
$ gang[1]=Suppe
$ gang[2]=Fisch
$ gang[4]=Nachtisch
```

Zugreifen können Sie auf diese Variablen entweder einzeln, wie in

```
$ echo ${gang[1]}
Suppe
```

(die geschweiften Klammern sind nötig, damit keine Verwirrung mit Dateisuchmustern aufkommen kann) oder insgesamt, wie in

```
$ echo ${gang[*]}
Aperitif Suppe Fisch Nachtisch
```

(Dass nicht alle Indizes fortlaufend benutzt sind, ist übrigens egal.) Die Expansionen »${⟨*Name*⟩[*]}« und »${⟨*Name*⟩[@]}« sind analog zu $* und $@, wie das folgende Beispiel illustriert:

```
$ gang[3]="Wiener Schnitzel"
$ for i in ${gang[*]}; do echo $i; done
Aperitif
Suppe
Fisch
Wiener
```

```
Schnitzel
Nachtisch
$ for i in "${gang[*]}"; do echo $i; done
Aperitif Suppe Fisch Wiener Schnitzel Nachtisch
$ for i in "${gang[@]}"; do echo $i; done
Aperitif
Suppe
Fisch
Wiener Schnitzel
Nachtisch
```

Letzteres ist das, was wir brauchen.

Sie können einem Feld auch im ganzen einen Wert zuweisen, etwa so:

```
$ gang=(Aperitif Gazpacho Lachs "Boeuf Stroganoff" "Crepes Suzette")
$ for i in "${gang[@]}"; do echo $i; done
Aperitif
Gazpacho
Lachs
Boeuf Stroganoff
Crepes Suzette
$ gang=([4]=Pudding [2]=Hecht [1]=Brühe [3]=Frikassee [0]=Aperitif)
$ for i in "${gang[@]}"; do echo $i; done
Aperitif
Brühe
Hecht
Frikassee
Pudding
```

Wenn Sie gründlich sein wollen, können Sie eine Variable mit

```
declare -a ⟨Name⟩
```

offiziell zu einem Feld erklären. Nötig ist das normalerweise nicht.

Angewandte Felder Unsere neue `present`-Routine muss also in einem Feld die Liste von Menü-Kurznamen und Antworten aufbauen, die später an `dialog` übergeben wird. Aussehen könnte das so:

```
declare -a answers
i=0
rightanswer=$(question $1 correct)
IFS=$'\n'
for a in $(question $1 answers)
do
    answers[$((2*i))]=$((i+1))
    answers[$((2*i+1))]="$a"
    [ "$a" = "$rightanswer" ] && rightshort=$((i+1))
    i=$((i+1))
done
```

Wir verwenden `i` als Index in das Feld `answers`. Für jede Antwort wird `i` um 1 erhöht, und die Plätze für den Kurznamen und die eigentliche Antwort ergeben sich aus einer Indextransformation: Beispielsweise landet für `i` = 1 der Kurzname in `answers[2]` und der Antworttext in `answers[3]`; für `i` = 3 der Kurzname in `answers[6]` und der Antworttext in `answers[7]`. Als Kurznamen verwenden wir allerdings die Zahlen 1, ... , 4 statt 0, ... , 3. Ferner merken wir uns den Kurznamen der richtigen

```
#!/bin/bash
# dwwm -- wwm mit dialog

# Die question-Funktion ist genau wie in wwm
<<<<<

function present () {
    declare -a answers
    i=0
    rightanswer=$(question $1 correct)
    IFS=$'\n'
    for a in $(question $1 answers)
    do
        answers[$((2*i))]=$((i+1))
        answers[$((2*i+1))]="$a"
        [ "$a" = "$rightanswer" ] && rightshort=$((i+1))
        i=$((i+1))
    done

    # Zeige die Frage
    sel=$(dialog --clear --title "Für $(question $1 next) Punkte" \
            --no-cancel --menu "$(question $1 display)" 10 60 4 \
                ${answers[@]} 2>&1 1>/dev/tty)
    test "$sel" = "$rightshort"
}

# Das Hauptprogramm ist genau wie in wwm
<<<<<

if [ $score -lt 0 ]
then
    msg="Das war wohl nicht so prall, leider verloren"
else
    msg="Gratuliere, Sie haben gewonnen"
fi
dialog --title "End-Ergebnis" --msgbox "$msg" 10 60
```

Bild 5.2: Eine Version von wwm mit dialog

Antwort in rightshort; das ist wichtig, da dialog uns nur den Kurznamen des gewählten Menüeintrags zurückliefert und nicht den vollen Eintrag, wie select das macht.

Mit unserem answers-Feld können wir jetzt dialog aufrufen:

```
# Zeige die Frage
sel=$(dialog --clear --title "Für $(question $1 next) Punkte" \
        --no-cancel --menu "$(question $1 display)" 10 60 4 \
            ${answers[@]} 2>&1 1>/dev/tty)
test "$sel" = "$rightshort"
```

Hier holen wir uns den Fragentext wieder über die display-Methode von question; die Option --no-cancel unterdrückt den »Abbrechen«-Knopf im Menü.

Zu guter Letzt können wir dialog auch benutzen, um das Gesamtergebnis bekanntzugeben:

```
if [ $score -lt 0 ]
```

```
then
    msg="Das war wohl nicht so prall, leider verloren"
else
    msg="Gratuliere, Sie haben gewonnen"
fi
dialog --title "End-Ergebnis" --msgbox "$msg" 10 60
```

Dazu dient die wesentlich simplere `--msgbox`-Option. Die wesentlichen geänderten Stellen für das `dialog`-basierte Skript (es heißt entsprechend `dwwm`) sind in Bild 5.2 übersichtlich gezeigt.

Ausblick `dialog` ist nützlich, aber Sie sollten es damit wahrscheinlich nicht übertreiben. Haupteinsatzzweck für `dialog`-basierte Programme ist vermutlich die Grauzone, wo etwas Angenehmeres gewünscht ist als rohe Textterminal-Interaktion, aber eine »echte« Grafikoberfläche nicht oder nicht notwendigerweise zur Verfügung steht. Beispielsweise verwendet die Installationsroutine von Debian GNU/Linux `dialog` – für eine komplette GUI-Umgebung ist auf den Bootdisketten nicht wirklich Platz. Komplexere grafische Oberflächen liegen jenseits von dem, was auch mit `dialog` möglich wäre, und sind die Domäne von Umgebungen wie Tcl/Tk oder Programmen auf der Basis von C oder C++ (etwa mit Qt oder Gtk+, KDE oder GNOME).

Zur Darstellung einfacher Dialogboxen (mit Knöpfen zum Anklicken) unter X11 dient das Programm `xmessage`. Hiermit können Sie zum Beispiel Nachrichten für Benutzer anzeigen (siehe Übung 5.10). `xmessage` ist ein Standardbestandteil von X11 und sollte daher auf praktisch jedem Linux-System mit einer Grafikoberfläche zur Verfügung stehen, auch wenn sein Aussehen wahrscheinlich nicht modernen ästhetischen Ansprüchen genügt.

KDE bietet ein vage an `dialog` angelehntes Programm namens `kdialog`, das KDE-artige GUIs aus Shellskripten erlaubt. Wir würden Ihnen für alles, was über ganz elementare Dinge hinausgeht, allerdings dringend zu einer »vernünftigen« Basis für Ihre GUI-Programme raten, etwa Tcl/Tk oder PyKDE.

Übungen

5.9 [!3] Schreiben Sie ein Shellskript `seluser`, das Ihnen ein Auswahlmenü mit allen Benutzern aus `/etc/passwd` anbietet (verwenden Sie den Benutzernamen als Kurznamen und den Inhalt des GECOS-Feldes – den bürgerlichen Namen – als Eintrag). Das Skript sollte den gewählten Benutzernamen auf der Standardausgabe liefern. (Stellen Sie sich vor, das Skript wäre ein Bestandteil eines umfassenderen Werkzeugs zur Benutzerverwaltung.)

5.10 [3] Schreiben Sie ein Shellskript `show-motd`, das `xmessage` verwendet, um den Inhalt von `/etc/motd` anzuzeigen. Sorgen Sie dafür, dass dieses Shellskript gestartet wird, wenn ein Benutzer sich anmeldet (*Tipp:* `Xsession`).

Kommandos in diesem Kapitel

`dialog` Erlaubt „grafische" Interaktionselemente auf einem Textbildschirm `dialog(1)` 84

`kdialog` Erlaubt Benutzung von KDE-Interaktionselementen von Shellskripten aus `kdialog(1)` 90

`xmessage` Zeigt eine Nachricht oder Anfrage in einem X11-Fenster an `xmessage(1)` 90

Zusammenfassung

- Mit read können Sie Daten von Dateien, Pipelines oder der Tastatur in Shellvariable einlesen.
- Das select-Kommando erlaubt eine komfortable wiederholte Auswahl aus einer numerierten Liste von Alternativen.
- Das Programm dialog macht es möglich, Shellskripte auf Textterminals mit an GUIs angelehnten Interaktions-Elementen zu versehen.