UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Praktikum 1

Linux-Shell – Erklärungen

Cybersecurity im Sommersemester 2023

Jan Wichelmann, Anja Köhl

Einführung: Everything Is a File

Eine essenzielle Eigenschaft von Linux ist das Prinzip „everything is a file“ — zu Deutsch: alles ist eine Datei. Schaut man sich im Dateisystem um, findet man sehr faszinierende Dateien: Die herkömmliche Auffassung von „Datei“ trifft bei ihnen gar nicht zu, dennoch werden sie von Linux als Dateien betrachtet. Als Erstes stellt sich aber nun die Frage, wie man sich überhaupt in einem Linux Dateisystem umschauen kann. Auf Linux geht der Verzeichnisbaum von dem “root” Verzeichnis, kurz ‘/’, aus. Dies ist der oberste Knoten des Baums und alle Dateien und Unterverzeichnisse sind von hier aus direkt erreichbar. Anders als bei Windows bleibt man immer im dem gleichen Verzeichnisbaum, auch wenn mehrere Speichermedien oder Geräte eingebunden sind. Ein paar[1] wichtige Unterordner sind:

- `/`
  - `bin` — essenzielle Programme, die immer verfügbar sind (z.B. um das System zu reparieren)
  - `dev` — Geräte-Dateien (everything is a file!)
  - `etc` — Systemkonfigurationsdateien
  - `home` — Benutzerverzeichnisse
  - `lib` — essenzielle Bibliotheken und Kernel-Module
  - `media` — Einhängepunkt für Wechseldatenträger (z.B. USBs)
  - `mnt` — Einhängepunkt für temporäre Dateisysteme (z.B. ein ISO-Abbild)
  - `sbin` — essenzielle Programme, die für den Systemstart benötigt werden (von normalen Nutzern nicht gebraucht)
  - `usr` — geteilte Benutzer-Programme und -Dateien (read-only)
    - `bin` — Benutzer-Programme
  - `var` — “variable” Dateien, die sich während des Betriebes ändern (z.B. Log-Dateien)
    - `log` — Log-Dateien

Während des Projekts werden Sie schrittweise die Linux-Shell kennenlernen, insbesondere die *Bourne-again Shell*: Bash. Die Shell ist ein Kommandozeilen-Interpreter. Bei Bash sieht die Eingabezeile wie folgt aus:

```
usr@host:~$
```

Dies kann in die einzelnen Komponenten heruntergebrochen werden:

1. `usr` ist der Benutzername

[1] Wenn Sie mehr wissen wollen, schauen Sie sich gerne https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html an oder geben den Befehl `man hier` im Terminal ein, um die Man-Page zu lesen.

2. `host` ist der Name des Hosts, meistens also der Name der Maschine, auf der man die Shell geöffnet hat
3. `~` zeigt den aktuellen Pfad an; falls der Pfad zu lang wird, kann dieser auch mit '. . .' gekürzt sein
4. `$` zeigt den Beginn der Benutzereingabe an

Eine Verbindung aufbauen

Zuallererst müssen Sie sich per SSH mit dem Linux-Server für das Projekt verbinden. Dazu kann der Befehl

```
ssh -p 10000 user1@teaching.its.uni-luebeck.de
```

benutzt werden. Dieser kann auch wieder in seine einzelnen Komponenten zerlegt werden:

1. **`ssh`** ist der Befehl, der ausgeführt werden soll
2. `-p 10000` gibt an, dass für die Verbindung TCP-Port 10000 benutzt werden soll (SSH verbindet sich sonst standardmäßig mit Port 22)
3. `user1` ist der Benutzername, der zum Login auf der Remote-Shell genutzt wird (dieser wird Ihnen zusammen mit einem Passwort auf der Weboberfläche vom Praktikumsserver angezeigt)
4. `teaching.its.uni-luebeck.de` ist der Host, mit dem die Verbindung aufgebaut wird.

Nach dem Bestätigen der Eingabe müssen Sie sich mit Ihrem Passwort authentifizieren. Hierbei ist zu beachten, dass die Shell die Eingabe bei Passwörtern aus Sicherheitsgründen nicht anzeigt.

Erste Schritte

Nun haben Sie eine Remote-Shell geöffnet. Als nächstes lernen Sie, wie Sie sich zurecht finden können, falls das Eingabe-Prompt der Shell nicht den Benutzernamen und den aktuellen Pfad anzeigt. Das Programm **`pwd`** steht für "**p**rint **w**orking **d**irectory" und gibt den absoluten Pfad zu dem Verzeichnis aus, in dem Sie sich momentan befinden. Der *absolute* Pfad geht von dem Root-Verzeichnis aus, beginnt also immer mit '/'. Ein *relativer* Pfad geht vom aktuellen Verzeichnis aus. Das Programm `whoami` zeigt den Benutzernamen an, mit dem Sie sich bei der Shell eingeloggt haben.

Als nächstes lernen Sie den Befehl **`ls`** kennen, kurz für "list". Dieses Programm führt die Inhalte eines Verzeichnisses auf. Ohne weitere Parameter werden nur die Dateien und Unterverzeichnisse des aktuellen Ordners aufgelistet.

```
its@linux:~/Directory$ ls
file1 file2 file3 file4 myDir
```

Listing 1: Ausgabe von dem Aufruf **`ls`**

```
its@linux:~/Directory$ ls -a
. .. .file5 file1 file2 file3 file4 myDir
```

Listing 2: Ausgabe von dem Aufruf **`ls`** `-a`

```
its@linux:~/Directory$ ls -l
total 0
-rw-rw-rw- 1 its its   0 Dec  2 00:16 file1
-rw-rw-rw- 1 its its   0 Dec  2 00:16 file2
-rw-rw-rw- 1 its its   0 Dec  2 00:16 file3
-rw-rw-rw- 1 its its   0 Dec  2 00:16 file4
drw-rw-rw- 1 its its 512 Dec  2 00:17 myDir
```

Listing 3: Ausgabe von dem Aufruf **`ls`** `-l`

Der einfache Aufruf von **`ls`** (Listing 1) zeigt die Dateien und Verzeichnisse von dem aktuellen Verzeichnis an. Standardmäßig werden verschiedene Dateitypen mit unterschiedlichen Farben gekennzeichnet.

Bei der Eingabe von **`ls`** `-a` (Listing 2) werden *versteckte* Dateien, also Dateien die mit '.' anfangen, auch angezeigt. Die ersten beiden Einträge '.' und '..' sind besonders, da sie einmal für das aktuelle Verzeichnis ('.'), und zum anderen Mal für das Elternverzeichnis ('..') stehen.

Der Befehl **`ls`** `-l` (Listing 3) stellt die Information zusätzlich in Langform dar.

1	2	3	4	5	6	7	8
-	rw-rw-rw-	1	its	its	0	Dec 2 00:16	file1
d	rw-rw-rw-	1	its	its	512	Dec 2 00:17	myDir

Tabelle 1: Langformat-Ausgabe von **ls**

Die Felder enthalten folgende Informationen:

1. Der '-' Strich bedeutet, dass es sich um eine reguläre Datei handelt; das 'd' zeigt hingegen an, dass es ein Verzeichnis ist.[2]
2. Die Berechtigungen für den Besitzer, die Gruppe und alle anderen
3. Die Anzahl von Hardlinks zu dieser Datei
4. Der Besitzer
5. Die Gruppe, zu der diese Datei gehört
6. Die Dateigröße in Bytes
7. Das Datum der letzten Änderung dieser Datei
8. Der Name

Es gibt für die Langform noch diverse zusätzliche Optionen, um mehr oder weniger Information darzustellen oder die Darstellungsform zu verändern.

Um den Inhalt von einem spezifischen Verzeichnis aufzulisten, wird der relative oder absolute Pfad zu dem gewünschten Verzeichnis nach den Optionen angegeben. Falls kein Pfad spezifiziert wird, wird das aktuelle Verzeichnis implizit angenommen.

Um in andere Verzeichnisse wechseln zu können, wird der Befehl **cd** `<Pfad zu Verzeichnis>`, kurz für '**c**hange **d**irectory', benutzt. Bei dem Pfad können auch folgende Kürzel verwendet werden:

- `.` für das aktuelle Verzeichnis
- `..` für das Elternverzeichnis
- `/` für das Rootverzeichnis
- `~` für das Homeverzeichnis

Neues Erschaffen

Die Welt steht Ihnen nun zur freien Durchforschung offen. Als nächstes gilt es, diese auch zu *verändern*. Mit Hilfe des Programms **touch** `<Pfad zu Datei>` können neue Dateien ohne Inhalt erzeugt werden. Dieser Befehl hat eigentlich die Funktion die 'last modified' und 'last accessed' Zeiten zu ändern. Wenn jedoch die angegebene Datei nicht existiert, wird sie ohne Inhalt neu erzeugt. Es können beliebig viele Dateien mit einem Befehl erzeugt werden, indem man mehrere Pfade angibt: **touch** `file1 file2 file3 ...`

Neue Verzeichnisse können Sie mit dem Programm **mkdir** `<Pfad zu Verzeichnis>` erstellen. Mit der Option `-p` bzw. `--parents` werden zusätzlich auch fehlende Oberverzeichnisse erstellt. Möchte man also von dem aktuellen Verzeichnis ausgehend einen Ordner 'DirParent' mit dem Unterordner 'DirChild' erstellen, so würde man den Befehl

```
mkdir -p DirParent/DirChild
```

eingeben. Wie schon bei **touch** kann man auch beliebig viele Verzeichnisse auf einmal erzeugen, indem man mehrere durch Leerzeichen getrennte Pfade angibt.

Mit dem Programm **nano** `<Pfad zu Datei>` können Sie den Text Editor 'Nano' starten und den Inhalt der spezifizierten Datei bearbeiten. Sollte keine Datei mit dem angegeben Pfad existieren, so wird eine neue Datei beim Speichern angelegt.

① Hier wird der Name der Datei angezeigt. In dem Falle, dass keine existierende Datei bearbeitet wird, steht hier 'New Buffer'.

[2] Weitere Zeichen können an dieser Position auftauchen. Eine vollständige Liste befindet sich in der Info Dokumentation: **info** `ls -n "What information is listed"`

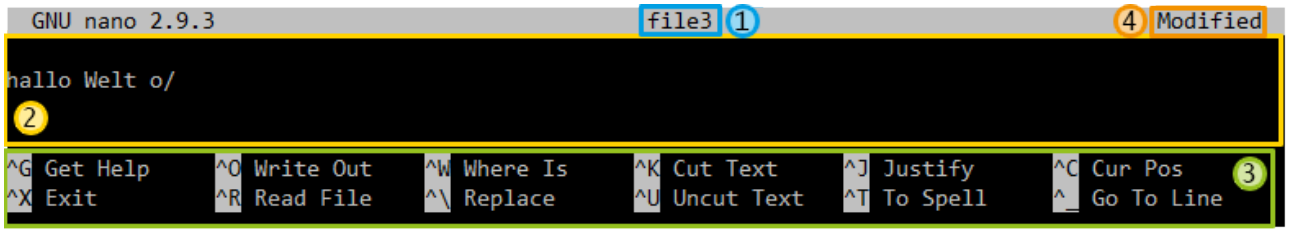


Abbildung 1: Der Nano-Texteditor

② Hier kann der Inhalt der Datei bearbeitet werden.

③ Hier werden die Shortcuts angezeigt. Das Symbol '^' steht hierbei für die `STRG`-Taste.

④ Hier wird angezeigt, dass sich der Inhalt der Datei verändert hat.

Ausgabe

Über die Kommandozeile haben Sie diverse Möglichkeiten, etwas auszugeben. Mit dem Programm **`echo`** `<String>` können Sie einen String auf der Konsole (standard output) ausgeben. Mit dem Programm **`cat`** `<Pfad zu Datei>` können Sie Dateien aneinander hängen (*concat*) und auf der Konsole ausgeben. Bei **`cat`** wird der *Inhalt* der Datei ausgegeben, im Gegensatz zu **`echo`**, bei dem der *spezifizierte String* ausgegeben wird.

Mit dem Programm **`less`** `<Pfad>` können Sie sich den Inhalt einer Textdatei anzeigen lassen, ohne dass diese komplett im Terminal ausgegeben wird, was insbesondere bei sehr langen Dateien praktisch ist. Stattdessen können Sie durch die Datei durchscrollen. Allerdings ist hierbei zu beachten, dass man mit **`less`** nur *lesen* kann, es ist also kein Texteditor! Die Navigation erfolgt ausschließlich mit der Tastatur.

Taste	Aktion
`Page-Up` oder `b`	Eine Seite zurück
`Page-Down` oder `Leertaste`	Eine Seite vorwärts
`Pfeil Taste ↑`	Eine Zeile nach oben
`Pfeil Taste ↓`	Eine Zeile nach unten
`g` oder <	Sprung zum Anfang der Textdatei
`G` oder >	Sprung zum Ende der Textdatei
`/<Text>`	Sucht nach dem <Text>
`n`	Springt zum nächsten Suchergebnis
`h`	Zeigt den Hilfsbildschirm
`q`	Beendet `less`

Der Pager **`less`** wird auch in den meisten Linux-Distributionen standardmäßig benutzt, um die Dokumentation (*man page*) von Programmen anzuzeigen. Mit dem Befehl **`man`** `<Befehlsname>` können Sie die Anleitung zu einem Befehl durchlesen.

Redirection

Insbesondere werden die Programme **`cat`** und **`echo`** in Kombination mit *manipulativen Operatoren* verwendet. Der Redirect Operator '>' leitet eine Ausgabe, die normalerweise im Standard Output erscheinen würde, in eine Datei weiter. Die Syntax ist dabei wie folgt:

```
<Befehl mit Ausgabe> > <Pfad zu Datei>
```

Die generierte Ausgabe von dem Befehl wird nun in die spezifizierte Datei geschrieben. Zu beachten ist, dass die Datei dabei neu erstellt und ggf. **überschrieben** wird! Beispielsweise kann man die Ausgabe von **`ls`** in einer Datei 'lsAusgabe' speichern mit:

```
ls > lsAusgabe
```

Falls man den Inhalt von der Datei nicht überschreiben möchte, sondern lediglich neuen Inhalt *anhängen* möchte, so kann man den Append-Operator '>>' verwenden. Dies ist insbesondere bei Log-Dateien nützlich. Analog zu '>' ist die Syntax hierbei

```
<Befehl mit Ausgabe> >> <Pfad zu Datei>
```

Möchte man also nun an die Datei 'lsAusgabe' eine weitere Ausgabe von einem **`ls`** Aufruf anhängen, so würde man folgendes eingeben:

```
ls >> lsAusgabe
```

Als Letztes gibt es den Pipe-Operator '|', mit dem man die Ausgabe von einem Befehl als Eingabe eines anderen Befehls benutzen kann. Die Syntax ist hierbei `<Befehl mit Ausgabe> | <Befehl mit Eingabe>`. Beispielsweise kann man sich eine sehr lange Ausgabe von **ls** auch mit **less** anzeigen lassen, indem man die beiden Befehle mit dem Pipe-Operator verknüpft:

```
ls -lah | less
```

Bewegung

Mit dem Programm **cp** `<Ursprungs Datei> <Ziel Datei>` kann eine Datei kopiert werden. Schon vorhandene Dateien werden hierbei **überschrieben**, es sei denn man benutzt die Optionen `-i` (für „interactive" – vor dem Verschieben fragen) oder `-n` (existierende Dateien nicht überschreiben). Wenn `<Ziel>` ein Verzeichnis ist, können auch mehrere Ursprungsdateien spezifiziert werden:

```
cp <Ursprung> <Ursprung> ... <Zielverzeichnis>
```

Das Programm **mv** `<Ursprung> <Ziel>` *verschiebt* eine Datei in ein neues Verzeichnis, und/oder benennt diese um, falls `<Ursprung>` und `<Ziel>` beide Dateinamen spezifizieren. Wie auch schon bei dem Befehl **cp** der Fall, werden schon vorhandene Dateien *ohne Nachfrage* **überschrieben**, solange die Option `-i` nicht angegeben ist.

Die Programme **rm** und **rmdir** löschen jeweils Dateien und *leere* Verzeichnisse. Hierbei ist zu beachten, dass **rm** *keine* Bestätigung vor dem Löschen einholt! Mit der Option `-i` bzw. `--interactive` kann dieses Verhalten aber eingefordert werden.

Suchen

Um Dateien nach Strings oder Mustern zu durchsuchen, kann das Programm **grep** `<Muster> [<Datei>]` benutzt werden. Dieses ist ein sehr mächtiges Programm, allerdings werden wir uns im Rahmen dieses Praktikums nur auf die einfache[3] Suche nach Strings beschränken. Einige Beispiele:

ls `/bin` | **grep** `zip`	Durchsuche den `/bin`-Ordner nach Programmen mit „zip" im Namen
grep `18/May/2019 log.log`	Gebe alle Zeilen eines Logs aus, die das Datum „18/May/2019" enthalten

Fortgeschrittenes: Shell-Skripting

Manchmal reichen ein oder zwei Befehle nicht aus oder man schreibt einen Einzeiler, der lieber zehn Zeilen lang hätte sein sollen. Es muss also was neues her — Skripte! In Skripten kann man mehrere Befehle schreiben, die dann gelesen und nacheinander ausgeführt werden. Bash hat dabei eine Reihe von interessanten Funktionen, die insbesondere in Skripten zum Einsatz kommen.

Zunächst muss jedoch die Frage geklärt werden, wie man ein Shell Skript als solches definiert. Hier kommt das sogenannte *Shebang* — die Zeichenfolge `#!` — zum Einsatz. Damit das Skript richtig ausgeführt wird, sollte in der ersten Zeile `#!/bin/bash` oder `#!/usr/bin/env bash` stehen. Wenn diese Zeile vom System beim Aufruf des Skripts gelesen wird, dann wird der entsprechende Interpreter (in diesem Fall Bash) gestartet, um das Skript auszuführen. Sollte diese Zeile fehlen, so kann nicht garantiert werden, dass Bash als Interpreter verwendet wird.

Variablen

In Bash kann man wie gewohnt Variablen mit einem '=' zuweisen. Damit gibt man einem Wert einen Namen, womit man wieder auf den Wert zugreifen kann. Ein leerer Wert ist auch möglich, daher ist die Angabe eines Wertes optional. Der Name ist jedoch Pflicht und muss dabei mit einem Buchstaben oder Unterstrich beginnen und sonst nur Buchstaben, Zahlen oder Unterstriche enthalten. Valide Namen wären also beispielsweise `abc`, `z_3`, `_5`. Es dürfen jedoch keine Leerzeichen zwischen dem Namen bzw. dem Wert und dem Gleichheitszeichen stehen, da diese sonst als Befehle interpretiert werden.[4]

[3] Einfach, da man mit **grep** auch mit RegEx nach Mustern suchen kann.
[4] Bei dem Wert muss also ein String mit voranstehende Leerzeichen mit Anführungszeichen umgeben werden: `x=" hi"`

```
name=[value]
```

Um auf den Wert einer Variable zuzugreifen, wird sogenannte *Parameter Expansion* verwendet. Mit `$name` bzw. `${name}` wird `name` zu seinem Wert expandiert. Die Klammerung hat keine Auswirkung und dient lediglich dazu, um den Namen oder den Wert eindeutig abzugrenzen. Beispielsweise könnte man eine Variable `file=myfile` haben, welche man verwenden möchte, um eine Datei umzubenennen. Der Befehl

```
mv $file $file_new
```

würde jedoch fehlschlagen, da Bash bei der zweiten Parameter Expansion eine neue Variable mit dem Namen `file_new` erstellt, anstatt die Datei zu `myfile_new` umzubenennen. Eindeutig wäre stattdessen

```
mv $file ${file}_new
```

Weitere Fälle von Parameter Expansion werden in dem späteren Abschnitt *Expansion* behandelt.

Bash Skripte können auch mit Parametern aufgerufen werden, welche dann im Skript selber benutzt werden können. Diese Parameter heißen *positional parameters*, da ihre Namen abhängig von ihrer Position beim Aufruf vergeben werden. Ruft man ein Skript beispielsweise mit `myScript.sh p1 p2 p3 p4 p5`, dann sind `p1` bis `p5` die *positional parameters*. Die Namen werden analog vergeben, sodass `$1` zu `p1` expandiert und `$5` zu `p5`. Sollte man eine mehrstellige Anzahl an positional parameters haben, so muss die jeweilige Zahl klammern, da sonst nur die erste Zahl als Name angenommen wird (`${10}` statt `$10`).

Mit dem Shell-Builtin `shift` kann man durch die positionalen Parameter „durchshiften“, wobei der Wert von `$2` nach `$1` bewegt wird, `$3` nach `$2`, usw. Die Anzahl von positionalen Parametern, die als Wert in `$#` hinterlegt ist, wird auch nach jedem `shift` reduziert. Ein Beispielskript, welches durch alle Parameter shiftet und immer den Wert von `$1` ausgibt, ist in Listing 4 zu sehen, sowie die Ausgabe bei einem Aufruf des Skripts mit zehn Parametern in Listing 5.

Es gibt eine Reihe von besonderen Parametern, die lediglich ausgelesen werden können, aber nicht neu zugewiesen. Einen solchen Parameter haben Sie schon kennen gelernt: `$#`, welcher die Anzahl der positionalen Parameter angibt. Weitere sind beispielsweise `$?`, welches den Exit-Status des zuletzt ausgeführten Programms angibt, und `$0`, welches den Namen der Shell oder des Shell-Skriptes angibt, wenn Bash mit einem Shell-Skript gestartet wurde[5]. Bei dem Beispiel `shift.sh` würde `$0` also den Namen `./shift.sh` enthalten.

```
#!/bin/bash

echo "${#} positional parameters"

while (($# > 0)); do
    echo "\$1 is ${1}"
    shift
done
```

Listing 4: shift.sh Skript

```
its@linux:~$ ./shift.sh 1 2 3 4 5 6 7 8 9 10
10 positional parameters
$1 is 1
$1 is 2
$1 is 3
$1 is 4
$1 is 5
$1 is 6
$1 is 7
$1 is 8
$1 is 9
$1 is 10
```

Listing 5: Terminal Ausgabe von shift.sh

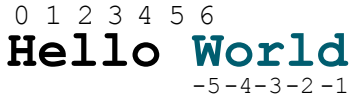
Expansion

Wie schon bei dem Dschinni aus Aladdin gilt auch bei Bash „Phänomenale kosmische Kräfte! - Winzig kleiner Lebensraum!“. Mit Expansion kann man sich häufig Schreibarbeit sparen, indem Bash sich den Rest „dazu expandiert“. Eine Art von Expansion wurde schon bei den Variablen behandelt: Parameter Expansion. Man gibt zusammen mit dem Zeichen `$` den Namen der Variable an und Bash expandiert dies zu dem Wert der Variable.

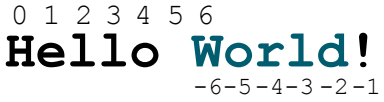
[5] Eine vollständige Liste der besonderen Parameter ist in der Dokumentation von Bash zu finden `https://www.gnu.org/software/bash/manual/html_node/Special-Parameters.html#Special-Parameters`

Substring Expansion

Eine weitere Form der *Parameter Expansion* ist die *Substring Expansion*. Dabei wird nur noch zu einem Teil des Wertes expandiert. Möchte man beispielsweise bei der Variable `hallo="Hello World"` nur das zweite Wort haben, so kann man dies mit `${hallo:6}` bzw. `${hallo: -5}` spezifizieren, da das Wort „World“ an der 6. Position von vorne bzw. der 5. Position von hinten beginnt.



Hätte man nun stattdessen die Variable `hallo="Hello World!"` und möchte nur das zweite Wort ohne Ausrufezeichen haben, so kann man bei Substring Expansion auch die Anzahl der Zeichen angeben, die man haben möchte: `${hallo:5:5}` bzw. `${hallo: -6:5}`.



Die allgemeine Syntax ist also wie folgt:

```
${name:start}
${name:start:länge}
```

Wenn `start` eine negative Zahl ist, dann muss ein Leerzeichen zwischen dem Doppelpunkt und der Zahl stehen.

Brace Expansion

Mit Brace Expansion können leicht unterschiedliche Strings nach einem bestimmten Muster erzeugt werden. Dabei kann entweder eine Sequenz oder eine Menge an Strings angegeben werden. Zuerst die Sequenzen-Form:

```
{x..y[..incr]}
```

Hierbei sind x und y beliebige ganze Zahlen oder einzelne Zeichen. Das letzte Element *incr* ist optional und gibt die Schrittgröße an. Standardmäßig ist die Schrittgröße ± 1. Mehrere Beispiele sind in Listing 6 zu sehen.

Alternativ können auch ein optionaler Präfix, mehrere durch Kommata getrennte Strings und ein optionaler Suffix angegeben werden, welche dann zu mehreren Strings kombiniert werden. Die beiden Formen können auch beliebig ineinander verschachtelt werden, wie in Listing 7.

```
its@linux:~$ echo {1..10}
1 2 3 4 5 6 7 8 9 10
its@linux:~$ echo {10..1}
10 9 8 7 6 5 4 3 2 1
its@linux:~$ echo {5..20..5}
5 10 15 20
```

Listing 6: Sequenzform

```
its@linux:~$ echo H{a,e}llo!
Hallo! Hello!
its@linux:~$ echo H{i,ey}
Hi Hey
its@linux:~$ echo {Mon,Frei}tag
Montag Freitag
its@linux:~$ echo {a,b{1..3}}
a b1 b2 b3
```

Listing 7: Mengenform und Verschachtelung

Arithmetic Expansion

Bash kann mit Hilfe von Arithmetic Expansion auch als Taschenrechner verwendet werden. Dies ist allerdings mit Vorsicht zu verwenden, da Bash nur mit *ganzen* und *„kleinen“* Zahlen (i.d.R. $\pm 2^{32}-1$ oder $\pm 2^{64}-1$) rechnen kann! Zulässige Operationen sind in Tabelle 2 aufgelistet. Es können sowohl Zahlen als auch Variablen verwendet werden und innerhalb eines Ausdrucks können Variablen auch lediglich mit ihrem Namen (also ohne `$`) verwendet werden. Die Syntax für arithmetische Expansion ist wie folgt:

```
$((<arithmetischer Ausdruck>))
```

```
its@linux:~$ i=1; echo $((i++ + 1)) $i
2 2
its@linux:~$ i=1; echo $((++i + 1)) $i
3 2
its@linux:~$ echo $((2**3 + 2))
10
its@linux:~$ echo $((0xa ^ 0xb))
1
its@linux:~$ echo $((2#101))
5
its@linux:~$ echo $((36#z)) $((36#Z))
35 35
its@linux:~$ echo $((64#z)) $((64#Z))
35 61
```

Listing 8: Beispiele für arithmetische Expansion

Operation	Beschreibung
`i++`	postincrement
`i--`	postdecrement
`++i`	preincrement
`--i`	predecrement
`+`	Addition
`-`	Subtraktion
`~`	bitweise Negation
`**`	Potenzierung
`*`	Multiplikation
`/`	Division
`%`	Modulo
`<<`	Linksshift
`>>`	Rechtsshift
`&`	bitweises AND
`|`	bitweises OR
`^`	bitweises XOR

Tabelle 2: arithmetische Operationen

Zahlen können auch in einer beliebigen Basis zwischen 2 bis 64 angegeben werden. Zahlen zu der Basis Acht können mit einer voranstehenden Null und hexadezimale Zahlen mit voranstehendem `0x` oder `0X` gekennzeichnet werden. Für sonstige Basen muss vor der Zahl mit `<Basis>#` die Basis spezifiziert werden. Bei einer Basis kleiner oder gleich 36 können Klein- oder Großbuchstaben für Zahlen, die größer als neun sind, verwendet werden. Ab einer Basis von 37 werden Kleinbuchstaben für die Zahlen 10 bis 35 und Großbuchstaben für die Zahlen 36 bis 61 verwendet. Die Zeichen `@` und _ können ab einer Basis von 63 für die Zahlen 62 und 63 verwendet werden. Beispiele sind in Listing 8 zu finden.

Command Substitution

Command Substitution mit `$(<Befehl>)` bewirkt, dass ein Befehl durch seine Ausgabe substituiert wird. Dafür wird der Befehl oder die Reihe von Befehlen in einer Subshell ausgeführt und das Ergebnis (ohne eventuelle newline Charaktere) schließlich an Stelle des Befehls eingefügt. Ein Beispiel:

```
its@linux:~$ echo "Logged in as $(whoami)"
Logged in as its
```

Konditionale

Bash hat für konditionale Anweisungen eine eigene Erweiterung, die mit dem Befehl `[[ <Ausdruck> ]]` verwendet werden kann. Die Leerzeichen zwischen den Klammern und dem Ausdruck sind essenziell, da es sich um einen Befehl handelt! Außerdem ist zu beachten, dass der Exit-Status bei einem True `0` ist und bei einem False `1`.

Ausdrücke können beispielsweise mit `!` negiert, mit `&&` konjugiert oder mit `||` disjungiert werden. Die Tabelle 3 listet diese und weitere mögliche Formen von Ausdrücken auf[6].

Diese Form von konditionalen Anweisungen sind *nicht* mit jeder Shell Variante kompatibel. Insbesondere werden sie nicht von `sh` unterstützt, sodass Skripte, die möglichst kompatibel sein sollen, das Shell-Builtin `test`[7] bzw. `[ <Ausdruck> ]` verwenden sollten.

Eine weitere Möglichkeit, wenn man mit Integern arbeitet, bietet der *arithmetischer Test* mit `((<arithmetischer Ausdruck>))`. Dieser hat einen Exit-Status von `0` (True), wenn das Ergebnis nicht `0` ergibt und sonst als Exit-Status `1`. Somit kann man wie gewohnt mit den Werten `0` für False und `1` für True rechnen. Zusätzlich zu den arithmetischen Operatoren aus Tabelle 2 können arithmetische Ausdrücke auch logische Operatoren (siehe Tabelle 4) enthalten.

[6] Weitere Formen sind in der Dokumentation von Bash beschrieben `https://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html#Bash-Conditional-Expressions`

[7] Valide Ausdrücke von `test` können Sie in der Dokumentation (`man test`) finden.

Ausdruck	Beschreibung
`! <Ausdruck>`	negiert einen Ausdruck
`<Ausdruck> && <Ausdruck>`	AND Verknüpfung zweier Ausdrücke
`<Ausdruck> || <Ausdruck>`	OR Verknüpfung zweier Ausdrücke
`<Int> -eq <Int>`	prüft ob beide Ints gleich sind
`<Int> -ne <Int>`	prüft ob beide Ints *nicht* gleich sind
`<Int1> -lt <Int2>`	prüft ob Int1 kleiner als Int2 ist
`<Int1> -le <Int2>`	prüft ob Int1 kleiner-gleich Int2 ist
`<Int1> -gt <Int2>`	prüft ob Int1 größer als Int2 ist
`<Int1> -ge <Int2>`	prüft ob Int1 größer-gleich Int2 ist
`<String> == <String>`	prüft ob beide Strings gleich sind
`<String> != <String>`	prüft ob beide Strings *nicht* gleich sind
`<String1> < <String2>`	prüft ob String1 lexikografisch kleiner als String2 ist
`<String1> > <String2>`	prüft ob String1 lexikografisch größer als String2 ist

Tabelle 3: logische Ausdrucksformen

```
its@linux:~$ [[ a == a ]]; echo $?
0  # True
its@linux:~$ [[ a != b && b == b ]]; echo $?
1  # False
its@linux:~$ ((0)); echo $?; ((1)); echo $?
1  # False
0  # True
its@linux:~$ ((1 && 1)); echo $?
0  # True
```

Listing 9: Beispiele für logische und arithmetische Tests

Operation	Beschreibung
!	logische Negation
<=	Vergleich ob kleiner gleich
>=	Vergleich ob größer gleich
<	Vergleich ob kleiner
>	Vergleich ob größer
==	Gleichheit
!=	Ungleichheit
&&	logisches AND
||	logisches OR

Tabelle 4: logische Arithmetische Operationen

If-Anweisungen

Mit If-Anweisungen kann man, abhängig vom Exit-Status eines Befehls, den Kontrollfluss des Skripts ändern. Die Syntax ist dabei folgendermaßen:

```
if <Befehle zum Testen>; then
  <weitere Befehle>;
[elif <weitere Befehle zum Testen>; then
  <weitere Befehle>;]
[else <nochmals weitere Befehle>;]
fi
```

Die Semikolons können in einem Skript auch mit Zeilenumbrüchen ersetzt werden und die eckigen Klammern deuten an, dass die `elif` und `else` Blöcke optional sind. Des Weiteren können beliebig viele `elif` Blöcke dem initialen `if` Block folgen. Der erste Block, bei dem die Test-Befehle `0` zurückgibt (der Befehl wird ausgeführt), wird ausgeführt. Eine If-Anweisung mit `elif` und `else` könnte also wie folgt aussehen:

```
if [[ $i -eq 0 ]]; then
  echo a
elif [[ $i -eq 1 ]]; then
  echo b
else
  echo c
fi
```

oder mit arithmetischen Tests:

```
if ((i == 0)); then
  echo a
```

```
elif ((i == 1)); then
  echo b
else
  echo c
fi
```

Da lediglich eine Liste von Befehlen als Test für den jeweiligen Block erwartet wird, kann auch die erfolgreiche Ausführung (Exit-Status von *null*) eines Befehls verwendet werden.

Loops

Eine weitere Möglichkeit den Kontrollfluss zu ändern bieten Loops. Dabei werden die Anweisungen aus dem Rumpf solange ausgeführt, bis eine Bedingung erfüllt bzw. nicht mehr erfüllt ist. Bei While-Schleifen werden Befehle solange ausgeführt bis die Bedingung nicht mehr gilt. Die Syntax in Bash ist wie folgt:

```
while <Befehle zum Testen>; do
  <weitere Befehle>
done
```

Bei For-Schleifen kann entweder über eine Liste iteriert werden (for-each) oder wie in Java eine Variable verwendet werden, die sich mit jedem Schleifendurchlauf ändert, bis die Bedingung nicht mehr gilt. Bei der for-each Variante ist die Syntax wie folgt:

```
for name in list; do
  <weitere Befehle>
done
```

Das `in list` darf jedoch fehlen. In diesem Fall wird über die Liste der positionalen Parameter iteriert. Für die Liste eignet sich vor allem Brace Expansion:

```
for i in {3..1}; do
  echo $i
done;
```

Die andere Variante im Java-Stil hat folgende Syntax:

```
for (( <Ausdruck 1>; <Ausdruck 2>; <Audruck 3> )); do
  <weitere Befehle>
done
```

In dem ersten Ausdruck wird üblicherweise die Variable für die Schleife initialisiert. Danach wird in dem zweiten Ausdruck eine Bedingung spezifiziert, die für jeden Schleifendurchlauf gelten soll und schließlich wird in dem dritten Ausdruck ein Ausdruck spezifiziert, der nach jedem Schleifendurchlauf ausgeführt wird.

```
#!/bin/bash

i=3

while ((i > 0)); do
    echo "$i"
    ((i--))
done
```

Listing 10: Beispiel einer While-Schleife

```
#!/bin/bash

for ((i=3; i > 0; i--)); do
    echo "$i"
done
```

Listing 11: Beispiel einer äquivalenten For-Schleife

```
its@linux:~$ ./while.sh
3
2
1
its@linux:~$ ./for.sh
3
2
1
```

Listing 12: Ausführung der Schleifen