Korpusbearbeitung Sommersemester 2021

Einführung

Florian Fink

15. April 2021

Organisatorisches

- Homepage des Kurses `cis-kb21.github.io`
- Folien, Hausaufgaben und Videos auf der Hompepage
- Vorlesung Donnerstag 14-16 Uhr (ct)
- Vorlesung über Zoom: `https://lmu-munich.zoom.us/j/8366632112?pwd=cWc3ck5ML0t1c0VnUTZ2Zit2aUpFdz09`
- Termin Klausur: 15.07.2021 (voraussichtlich)
- Bei Fragen Email an `kb21@cis.lmu.de`

Hausaufgaben

- Bearbeitung der Hausaufgaben ist freiwillig und nur in Gruppen von 2-3 Personen möglich
- Anmeldung der Gruppen via Email an `kb21@cis.lmu.de`
- Abgabe der Hausaufgaben via Email an `kb21@cis.lmu.de`
- Hausaufgaben werden nach der Vorlesung auf der Hompage veröffentlicht
- Abgabetermin ist die Vorlesung 7 Tage später
- Die Bearbeitung der Hausaufgaben gibt Bonuspunkte für die Klausur

Bonuspunkte

Bei 100% aller Punkte bei den Hausaufgaben gibt es einen Bonus von 10% der Klausurpunkte für alle Gruppenmitglieder. Mit weniger Punkten in den Hausaufgaben gibt es prozentual weniger Punkte.
Zum Beispiel habe die Klausur insgesamt 40 Punkte:

- Bei 100% der Punkte in den Hausaufgaben gibt es 4 Bonuspunkte ($0.1 * 40 * 1.0 = 4.0$)
- Bei 90% der Punkte in den Hausaufgaben gibt es 3.6 Bonuspunkte ($0.1 * 40 * 0.9 = 3.6$)
- Bei 50% der Punkte in den Hausaufgaben gibt es 2 Bonuspunkte ($0.1 * 40 * 0.5 = 2.0$)

Überblick

- Shell und Shell-Skripte
- Unix-Werkzeuge
- awk und sed
- Kodierungen
- Dateiformate
- verschiedene Korpora
- POS-Tagging (Tree-Tagger)
- ...

Unix-Shells

- (interaktive) Kommandozeileninterpreter
- verschiedene Shells mit unterschiedlicher Syntax(2):
  - `sh` die ursprüngliche *Bourne shell*
  - `bash` die *Bourne-again shell*
  - `zsh` die *z-shell*
  - `fish` die *friendly interactive shell*
  - `dash` die *Debian Almquist shell*
  - ...
- Unixoide Shells (insbesondere die `bash`) verfügbar für OSX und Windows (wsl1/2)

Unix-Umgebung

- die Unix-Umgebung besteht aus einer Vielzahl kleiner, vielseitiger Programme(1)
- Programme können flexibel kombiniert werden um komplexere Aufgaben zu bewältigen
- Programme für verschiedene Aufgaben(4):
  - Dateiverwaltung
  - Textverarbeitung
  - Datenverarbeitung
  - Benutzerverwaltung
  - Netzwerkverwaltung
  - ...

Interaktive Kommandozeilenumgebung

- die Shell bietet eine interaktive Umgebung um Befehle auszuführen
- Eingabezeilen werden an Leerzeichen in Token aufgetrennt
- einzelne Token (Befehle) werden ausgeführt
- es stehen verschiedene Tastaturkürzel für die interaktive Eingabe zur Verfügung

Tastaturkürzel

- `CTRL+k` schneidet Text vom Cursor bis zum Zeilenende aus (kill)
- `CTRL+u` schneidet Text vom Cursor bis zum Zeilenanfang aus
- `CTRL+y` fügt ausgeschnittenen Text am Cursor ein (yank)
- `CTRL+a` setzt den Kursor an den Zeilenanfang
- `CTRL+f` / `RIGHT` bewegt den Cursor nach rechts
- `CTRL+b` / `LEFT` bewegt den Cursor nach links
- `CTRL+p` / `UP` geht einen Schritt rückwärts in der Befehlsgeschichte
- `CTRL+n` / `DOWN` geht einen Schritt vorwärts in der Befehlsgeschichte
- `CTRL+x CTRL+e` öffnet einen Editor um einen Befehl zu editieren
- ...

Laufzeitumgebung

- beim Starten einer Shell-Sitzung werden verschiedene Variablen in der Laufzeitumgebung gesetzt
- Ausgabe der Laufzeitumgebung mit `env`
- Programme und Shell-Skripte erben die Laufzeitumgebung
- wichtige Variablen:
  - `PATH` Liste von Verzeichnissen, in denen nach Programmen gesucht wird (separiert durch `:`)
  - `HOME` Pfad des Benutzerverzeichnis
  - `EDITOR` Standardeditor
  - `USER` Benutzername
  - `SHELL` Standard-Shell
  - `LANG` Spracheinstellung

Shell-Skripte

- interaktive Befehle können auch in *Shell-Skripten* zusammengefasst und ausgeführt werden
- Shell-Skripte werden zeilenweise gelesen und abgearbeitet
- vor allem geeigntet für kurze Hilfsprogramme
- vor allem geeignet zur einfachen Stringverarbeitung; numerische Anwendungen sind nur sehr eingeschränkt möglich
- Shells bieten auch Möglichkeiten Verzweigungen und Schleifen zu verwenden (`if`, `case`, `for`...)
- Listen und assoziative Listen sind vorhanden (seit `bash 4.0`) aber sehr arkane Syntax

Hello world in bash

Datei `hello_world.bash`:

```
1 #!/bin/bash
2 # Ausgabe von "Hello world!" auf die Konsole
3 echo "Hello world!"
```

Ausführen der Datei mit bash:

```
$ bash hello_world.bash
Hello world!
$
```

Direktes Ausführen des Skripts:

```
$ chmod a+x hello_world.bash
$ ./hello_world.bash
Hello world!
$
```

shebang

Die sog. *shebang*-Zeile (bzw. *shabang*, *hashbang*, ...) dient dem direkten Ausführen von Programmen(3):

- Wenn der Unix-Kernel ein Programm ausführt, schaut er die ersten beiden Bytes des Programms an.
- Sind die ersten beiden Bytes `#!`, erwartet er eine Zeile mit dem das Programm ausgeführt werden soll.
- `#!/usr/bin/perl`
- `#!/usr/bin/env python`
- ...

Variablen

- Variablen sind Strings (alternative Syntax für ganzzahlig Arithmetik)
- Eine Variable wird durch `var=val` gesetzt (keine Leerzeichen möglich)
- Auf den Wert einer Varibale wird mit `$var` oder `${var}` zugegriffen
- Alle Variablen in einem Skript sind **global** (alternative Syntax für lokale Variablen in Funktionen)
- In Strings mit doppelten Anführungszeichen werden Variablen automatisch ersetzt
- In Strings mit einfachen Anführungszeichen werden Variablen **nicht** ersetzt

Variablen (Beispiel)

Datei `hello_world.bash`:

```
1 #!/bin/bash
2 h="Hello"
3 w="world!"
4 echo $h $w
```

Ausführen:

```
$ bash hello_world.bash
Hello world!
$
```

Variablen können auch direkt in der interaktiven Shell gesetzt werden:

```
$ str="Hello world!" echo $str
Hello world!
$
```

Ersetzung von Variablen in Strings

Datei `hello_world.bash`:

```
1 #!/bin/bash
2 h=Hello
3 w=world
4 s1="$h $w!"
5 s2='$h $w!'
6 echo $s1
7 echo $s2
```

Ausführen:

```
$ bash hello_world.bash
Hello world!
$h $w!
$
```

Optionen für echo

Mit der `-n` Option von `echo` kann die Ausgabe des Zeilenumbruchs verhindert werden:

```
$ echo -n Hello world!
Hello World!$
```

Mit der `-e` Option von `echo` werden Escape-Sequenzen interpretiert:

```
$ echo -e "Hello\nworld!\a"
Hello
world! <BEEP!>
$
```

Formatierte Ausgabe

Mit `printf` können formatierte Ausgaben umgesetzt werden (ähnlich zu Pythons f-Strings):

```
$ printf "%% %s: %d %f %.1f %g\n" \
    "some numbers" 13 0.3333 0.3333 0.3333
% some numbers: 13 0.333300 0.3 0.3333
$
```

Ausgabespezifizierer von printf

- %% gibt % aus
- %s formatiert Strings
- %d formatiert Ganzzahlen
- %f formatiert Gleitkommazahlen (mit 6 Stellen nach dem Komma)
- %.nf formatiert Gleitkommazahlen (mit n Stellen nach dem Komma)
- %g formatiert Gleitkommazahlen in normaler oder exponentialer Schreibweise so genau wie möglich
- ...

Dokumentationssystem

Unix verfügt über ein eingebautes Dokumentationssystem (man-pages). Mit dem `man` Befehl kann die Dokumentation verschiedener Befehle nachgeschlagen werden.
Verwendung:

```
man [section] Befehl
```

Sektionen

Das eingebaute Dokumentationssystem ist in unterschiedliche *Sektionen* eingeteilt.

Tabelle: Die unterschiedlichen Sektionen des Dokumentationsystems

Sektionsnummer	Verwendung
1	Benutzer-Befehle
2	Systemaufrufe (system calls)
3	Funktionen der C-Bibliothek
4	Gerätedateien und spezielle Datei(systeme)
5	Dateiformate und Konventionen
6	Spiele usw.
7	Verschiedenes
8	Systemadministation und Daemons

Beispiel für man printf

```
$ man printf

PRINTF(1)          User Commands          PRINTF(1)

NAME
       printf - format and print data

SYNOPSIS
       printf FORMAT [ARGUMENT]...
       printf OPTION

DESCRIPTION
       Print ARGUMENT(s) according to FORMAT, or
       execute according to OPTION:
...
```

Beispiel für `man printf` (Vortsetzung)

```
...
SEE ALSO
       printf(3)

       Full           documentation           at:
       <https://www.gnu.org/software/core-
       utils/printf>
       or  available  locally  via: info '(core-
       utils) printf invocation'

GNU coreutils 8.30February 2019        PRINTF(1)
```

Beispiel für man 3 printf

```
$ man 3 printf

PRINTF(3)   Linux Programmer's Manual  PRINTF(3)

NAME
       printf,   fprintf,   dprintf,   sprintf,
       snprintf,  vprintf,  vfprintf,  vdprintf,
       vsprintf,  vsnprintf  -  formatted output
       conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
...
```

Beispiel für `man 3 printf` (Vortsetzung)

```
...
   Conversion specifiers
       A character that specifies the type of
       conversion to be applied. The conversion
       specifiers and their meanings are:

       d, i   The int argument is converted to
              signed decimal notation. The pre-
              cision, if any, gives the minimum
              number of digits that must appear;
              if the converted value requires
              fewer digits, it is padded on the
              left with zeros. The default pre-
              cision is 1. When 0 is printed
              with an explicit precision 0, the
              output is empty.
...
```

echo

Name:	echo – Ausgeben von Textzeilen
Übersicht:	`echo [OPTION]... [STRING]...`

Wichtige Optionen:

`-n`	Verhindert die Ausgabe eines Zeilenumbruchs (`\n`)
`-s`	Verhindert die Ausgabe von Leerzeichen zwischen den Strings
`-e`	Schaltet die Interpretation von Escape-Sequenzen an

printf

Name:	printf – Ausgabe von formatiertem Text
Übersicht:	`echo [FORMAT]... [ARGUMENT]...`

Wichtige Formatierungszeichen:

`%d`	Formatierung von ganzen Zahlen
`%f`	Formatierung von Gleitkommazahlen mit 6 Nachkommastellen
`%g`	Formatierung von Gleitkommazahlen so genau wie möglich
`%s`	Stringformatierung
`%%`	Gibt ein % aus

env

Name:	env – Ausführen von Programmen in einer modifizierten Laufzeitumgebung
Übersicht:	`env [OPTIONS]... [-] [NAME=VALUE]... [COMMAND [ARGS]...]`
Beschreibung:	Setzt jeden NAME aus VALUE in der Laufzeitumgebung und führt COMMAND aus

Wichtige Optionen:

`-i`	Beginne mit einer leeren Laufzeitumgebung
`-u|--unset=NAME`	Entfernt Variable aus der Laufzeitumgebung
`-c|--chdir=dir`	Wechsle das Arbeitsverzeichnis

Kommandozeilenargumente

Shell-Skripte verfügen über spezielle eingebaute Variablen, mit der auf die Kommandozeilenargumente zugegriffen werden können.
Datei `args.bash`:

```
1 #!/bin/bash
2 echo "Erstes Kommandozeilenargument: $1"
3 echo "Zweites Kommandozeilenargument: $2"
4 echo "Index des letzten Arguments: $#"
5 echo "Alle Argumente: $@"
```

Ausführen:

```
$ bash args.bash eins zwei drei
Erstes Kommandozeilenargument: eins
Zweites Kommandozeilenargument: zwei
Index des letzten Arguments: 3
Alle Argumente: eins zwei drei
$
```

Verzweigungen

Mit dem `if` Schlüsselwort können Verzweigungen in der Shell umgesetzt werden.
Datei `greet.bash`:

```
1  #!/bin/bash
2  if [[ $# == 0 ]]; then
3      echo "Benutzung: $0 <NAME>"
4  else
5      echo "Hallo $1!"
6  fi
```

Ausführen:

```
$ bash greet.bash
Benutzung: greet.bash <NAME>
$ bash greet.bash Florian
Hallo Florian
$
```

Verzweigungen mit elif

Datei greet.bash:

```
1  #!/bin/bash
2  if [[ $# == 0 ]]; then
3      echo "usage $0 <NAME>"
4  elif [[ $1 != "Florian" ]]; then
5      echo Hallo $1!
6  else
7      echo Hallo Flo!
8  fi
```

Ausführen:

```
$ bash greet.bash Florian
Hallo Flo!
$ bash greet.bash Anna
Hallo Anna!
$
```

Rückgabewerte von Programmen

- Programme geben einen Rückgabewert zurück (zwischen 0 und 255)
- Ein Rückgabewert von 0 zeigt die erfolgreiche Ausführung von Programmen an
- Fehler werden mit einem Rückgabewert $\neq 0$ angezeigt
- Kommandozeilenprogramme und Shell-Skripte sollten ebenfalls dieser Konvention folgen
- Mit dem eingebauten Befehl `exit n` wird ein Shell-Skript beendet und `n` zurückgeliefert
- Die eingebaute Variable `$?` speichert den Rückgabewert des zuletzt ausgeführten Programms
- Mit `if` kann der Rückgabewert von Programmen auch direkt getestet werden (wobei 0 wahr und $\neq 0$ falsch ist)

Testen von Rückgabewerten

Datei `run.bash`:

```
1 #!/bin/bash
2 bash $1
3 if [[ $? != 0 ]]; then
4     echo "Der Befehl 'bash $1' ist gescheitert!"
5 fi
```

Ausführen:

```
$ bash run.bash bad
Der Befehl 'bash bad' ist gescheitert!
$ echo $?
0
$
```

Rückgabewerte von Shell-Skripten

Datei `run.bash`:

```
1 #!/bin/bash
2 bash $1
3 if [[ $? != 0 ]]; then
4     echo "Der Befehl 'bash $1' ist gescheitert!"
5     exit 1
6 fi
```

Ausführen:

```
$ bash run.bash bad
Der Befehl 'bash bad' ist gescheitert!
$ echo $?
1
$n
```

Direktes Testen von Rückgabewerten

Datei `run.bash`:

```
1 #!/bin/bash
2 if bash $1; then # Falls der Aufruf == 0 zurückliefert
3     exit 0; # OK
4 else
5     echo "Der Befehl 'bash $1' ist gescheitert!"
6     exit 1
7 fi
```

Ausführen:

```
$ bash run.bash bad
Der Befehl 'bash bad' ist gescheitert!
$ echo $?
1
$
```

man grep

```
GREP(1)                  User Commands                  GREP(1)

NAME
       grep,  egrep,  fgrep, rgrep - print lines
       that match patterns

SYNOPSIS
       grep [OPTION...] PATTERNS [FILE...]
...
EXIT STATUS
       Normally  the  exit status is 0 if a line
       is selected, 1 if no lines were selected,
       and  2 if an error occurred.  However, if
       the -q or --quiet or --silent is used and
       a  line is selected, the exit status is 0
       even if an error occurred.
...
```

Direktes Testen von Rückgabewerten

Datei `search.bash`:

```
1 #!/bin/bash
2 if grep $1 $2; then
3     echo "Die Datei $1 enthält $2!"
4 fi
```

Ausführen:

```
$ bash search.bash myfile.txt kalifragelistisch
Die Datei myfile.txt enthält kalifragelistisch!
$
```

Schleifen

Mit `for`-Schleifen kann man über mit Leerzeichen separierte Listen iterieren.

Datei `iterate.bash`:

```
1 #!/bin/bash
2 for i in 1 2 3 4; do
3     echo $i
4 done
```

Ausführen:

```
$ bash iterate.bash
1
2
3
4
```

Schleifen über Kommandozeilenargumente

Datei `greet_all.bash`:

```
1 #!/bin/bash
2 for name in $@; do
3     echo Hallo $name!
4 done
```

Ausführen:

```
$ bash iterate.bash Sophie Benjamin
Hallo Sophie!
Hallo Benjamin!
```

Quoting

- Die bash-Shell trennt Strings an Leerzeichen auf
- Dieses Verhalten betrifft insbesondere auch Variablen
- Kann verschiedene arkane Fehler in Skripten verursachen
- Kann auch zu Syntaxfehlern führen
- Um Strings mit Leerzeichen zu behandeln müssen Sie in doppelte Anführungszeichen gesetzt werden
- Dasselbe gilt für Variablen
- Grundsätzlich sollten alle Variablen, über die man keine direkte Kontrolle hat, in doppelten Anführungszeichen eingebettet werden

Quoting in Schleifen

Datei `greet_all.bash`:

```
1 #!/bin/bash
2 for name in $@; do
3     echo Hallo $name!
4 done
```

Ausführen:

```
$ bash iterate.bash Sophie Müller Benjamin Schmidt
Hallo Sophie!
Hallo Müller!
Hallo Benjamin!
Hallo Schmidt!
```

Quoting in Schleifen (2. Versuch)

Datei `greet_all.bash`:

```
1 #!/bin/bash
2 for name in $@; do
3     echo Hallo $name!
4 done
```

Ausführen:

```
$ bash iterate.bash "Sophie Müller" "Benjamin Schmidt"
Hallo Sophie!
Hallo Müller!
Hallo Benjamin!
Hallo Schmidt!
```

Quoting in Schleifen (3. Versuch)

Datei `greet_all.bash`:

```
1 #!/bin/bash
2 for name in "$@"; do
3     echo "Hallo $name!"
4 done
```

Ausführen:

```
$ bash iterate.bash "Sophie Müller" "Benjamin Schmidt"
Hallo Sophie Müller!
Hallo Benjamin Schmidt!
```

Schleife über Listen

Datei `animals.bash`:

```
1 #!/bin/bash
2 animals="Katze Hund Seegurke Elefant"
3 for animal in "$animals"; do
4     echo "$animal"
5 done
```

Ausführen:

```
$ bash animals.bash
Katze Hund Seegurke Elefant
```

Schleife über Listen (2. Versuch)

Datei `animals.bash`:

```
1 #!/bin/bash
2 animals="Katze Hund Seegurke Elefant"
3 for animal in $animals; do
4     echo "$animal"
5 done
```

Ausführen:

```
$ bash animals.bash
Katze
Hund
Seegurke
Elefant
```

Quoting in Verzweigungen

Datei guess_name.bash:

```
1  #!/bin/bash
2  name="Arthur Dent"
3  if [ $name == $1 ]; then # Alte sh-Syntax
4      echo "Richtig geraten!"
5  else
6      echo "Falsch geraten! Versuchs nochmal!"
7  fi
```

Ausführen:

```
$ bash guess_name.bash "Arthur Dent"
guess_name.bash: line 3: [: too many arguments
Falsch geraten! Versuchs nochmal!
```

Quoting in Verzweigungen (2. Versuch)

Datei guess_name.bash:

```
1 #!/bin/bash
2 name="Arthur Dent"
3 if [ "$name" == "$1" ]; then # Alte sh-Syntax
4     echo "Richtig geraten!"
5 else
6     echo "Falsch geraten! Versuchs nochmal!"
7 fi
```

Ausführen:

```
$ bash guess_name.bash "Arthur Dent"
Richtig geraten!
```

Quoting in Verzweigungen (3. Versuch)

Datei guess_name.bash

```
1 #!/bin/bash
2 name="Arthur Dent"
3 if [[ $name == $1 ]]; then
4     echo "Richtig geraten!"
5 else
6     echo "Falsch geraten! Versuchs nochmal!"
7 fi
```

Ausführen:

```
$ bash guess_name.bash "Arthur Dent"
Richtig geraten
```

Referenzen

[1] Kernighan, B., PIKE, K., and Pike, R. (1984). *The UNIX Programming Environment*. Prentice-Hall software series. Prentice-Hall.

[2] Peek, J., Powers, S., O'Reilly, T., and Loukides, M. (2002). *Unix Power Tools*. O'Reilly Media, Inc.

[3] Robbins, A. and Beebe, N. H. F. (2005). *Classic Shell Scripting*. O'Reilly Media, Inc., 1005 Gravenstein Highway North,Sebastopol, CA 95472, USA.

[4] Siever, E., Weber, A., Figgins, S., Love, R., and Robbins, A. (2005). *Linux in a Nutshell*. In a Nutshell (o'Reilly) Series. O'Reilly Media, Inc.