UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

INSTITUT FÜR COMPUTER-LINGUISTIK

Einführung in die Nutzung computerlinguistischer Ressourcen

Julius Steen

8.04.2019- 12.04.2019

Material in Teilen von Laura Jehl, Nils Reiter, Shigehiko Schamoni und Sebastian Martschat

Organisatorisches

- Ziel: Vertrautheit mit Daten und Tools zur Bearbeitung von Softwareprojekten erlangen
- Integrierte praktische Übungen
- Zeiten: Montag bis Freitag, 10:00 Uhr bis 13:00 Uhr und 14:00 Uhr bis 18:00 Uhr
- Slides und Übungen verfügbar unter `http://cl.uni-heidelberg.de/courses/ws18/ressourcen/`
- falls Sitzung verpasst: Übungen bitte nachträglich bearbeiten und am `steen@cl.uni-heidelberg.de` senden

Der Plan

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vormittag	Linux I	Korpora, Formate	Parser, Tagger	Python für ML	Arbeiten mit dem Cluster
Nachmittag	Linux II	VCS, Makefiles	Python für NLP	Weka	

Montag: Linux I

1. Linux I
   - Grundlagen
   - Prozesse, Streams und Pipes
   - Reguläre Ausdrücke
   - Batch-Verarbeitung

Linux I

Motivation

- viele Tools der Computerlinguistik sind für Linux entwickelt
- Linux stellt mächtige Kommandozeilentools zur Verfügung
- nützlich für automatisierte
  - Datenorganisation
  - Datenaufbereitung
  - Extraktion von Informationen

Terminal

Sinn und Zweck

- Benutzerschnittstelle zu Computern mit Unix-Systemen (z.B. Linux, Mac OS X)
- Befehle werden textuell eingegeben und *nacheinander* verarbeitet

Terminal vs. Shell

Terminal Ein *Gerät*, das Interaktion mit einem Computer erlaubt

Shell Ein *Programm*, das auf einem Computer läuft, Befehle entegegennimmt und interpretiert

Terminals I

Foto: Marcin Wichary, San Francisco, U.S.A.

Terminals II

public domain

Terminalemulator

- Terminalemulatoren “emulieren” (täuschen vor) ein Terminal. Sie laufen in graphischen Umgebungen in eigenen Fenstern.
- Innerhalb des Fensters laufen die gleichen Programme, die auch in echten Terminals laufen (eine Shell).
- Heute lässt man “-emulatoren” oft weg und spricht einfach von Terminal.

Shell

- Die Shell läuft innerhalb des Terminals
- Es gibt viele unterschiedliche Shells.
- Wir benutzen ausschließlich die Bourne-Again shell: bash.

Befehle

- Befehle werden in Textform eingegeben und der Reihe nach ausgeführt.
- Befehle haben
  - einen Namen
  - Optionen (*wie der Befehl ausgeführt wird*)
  - Argumente (*worauf der Befehl angewendet wird*)
  - Optionen und Argumente sind Parameter eines Befehls
- Befehle sind eigene Programme (z.B. `less`, `wc`) oder Builtins, also Teile der Shell (z.B. `cd`, `echo`).

Hilfe zur Selbsthilfe

- Man weiß nie alle Optionen auswendig
- Wichtig: Man kann Optionen nachschauen
- Viele Programme zeigen selbständig Hilfe an: **`-h`**, **`-help`** oder **`--help`**
- Viele Programme bieten eine Man-page an:

```
:~$ man <Programmname>
```

- Hilfe zur Kommandozeile insgesamt:
  `http://tldp.org/LDP/abs/html/` oder

```
:~$ man bash
```

Notation

Einzelne Befehle

- sind in **`Maschinenschrift`** gesetzt

Vollständige Eingaben

- haben einen Prompt, sind in `Maschinenschrift` und grau hinterlegt

```
:~$ Befehl
```

Grundlegende Befehle

Navigieren im Verzeichnisbaum

`ls`	Zeigt Verzeichnisinhalt an
`cd`	Wechselt das aktuelle Verzeichnis
`pwd`	Gibt das aktuelle Verzeichnis aus
`mkdir`	Legt ein neues Verzeichnis an
`rmdir`	Löscht ein leeres Verzeichnis

Hilfe zur Selbsthilfe

`man`	Zeigt manual an
`info`	Zeigt info-Seiten an

Grundlegende Befehle

Navigieren im Verzeichnisbaum

`ls`	Zeigt Verzeichnisinhalt an
`cd`	Wechselt das aktuelle Verzeichnis
`pwd`	Gibt das aktuelle Verzeichnis aus
`mkdir`	Legt ein neues Verzeichnis an
`rmdir`	Löscht ein leeres Verzeichnis

Hilfe zur Selbsthilfe

`man`	Zeigt manual an
`info`	Zeigt info-Seiten an

Lange und kurze Optionen

Die meisten Programme folgen gewissen Konventionen in der Interpretation ihrer Eingaben.

Kurze Optionen

- Beginnen mit einem einfachen Strich
- Sind nur ein Zeichen lang: `-l`
- Mehrere können zusammengefasst werden: `-la` statt `-l -a`

Lange Optionen

- Beginnen mit zwei Strichen
- Sind länger als ein Zeichen: `--help`

- Viele Optionen gibt es sowohl in kurz als auch in lang (man page)

Lange und kurze Optionen

Die meisten Programme folgen gewissen Konventionen in der Interpretation ihrer Eingaben.

Kurze Optionen

- Beginnen mit einem einfachen Strich
- Sind nur ein Zeichen lang: `-l`
- Mehrere können zusammengefasst werden: `-la` statt `-l -a`

Lange Optionen

- Beginnen mit zwei Strichen
- Sind länger als ein Zeichen: `--help`

- Viele Optionen gibt es sowohl in kurz als auch in lang (man page)

Lange und kurze Optionen

Die meisten Programme folgen gewissen Konventionen in der Interpretation ihrer Eingaben.

Kurze Optionen

- Beginnen mit einem einfachen Strich
- Sind nur ein Zeichen lang: `-l`
- Mehrere können zusammengefasst werden: `-la` statt `-l -a`

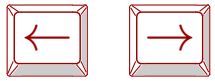
Lange Optionen

- Beginnen mit zwei Strichen
- Sind länger als ein Zeichen: `--help`

- Viele Optionen gibt es sowohl in kurz als auch in lang (man page)

Tastenkürzel

Blättern in der Befehlshistorie



Editieren des aktuellen Befehls

Automatisches Ergänzen von Datei- und Verzeichnisnamen

Springe zum Anfang der Zeile

Springe zum Ende der Zeile

Suche in der Befehlshistorie

Bricht den gerade laufenden Prozess ab

Mehr Befehle

`less`	Zeigt den Inhalt von Textdateien auf dem Terminal an ( zum beenden)
`cp`	Kopiert Dateien oder Verzeichnisse
`mv`	Verschiebt Dateien oder Verzeichnisse Wird auch zum Umbenennen verwendet
`rm`	Löscht Dateien
`cat`	Gibt den Dateieinhalt im Terminal aus
`zip` / `unzip`	Packt bzw. entpackt ZIP-Archive
`sleep`	Wartet einen angegebenen Zeitraum

- Es gibt *keine* vollständige Liste aller Befehle.
- Oft muss man viel lesen um einen passenden Befehl zu finden.

Dateien und Verzeichnisse

- Hierarchische Struktur
- Zu jeder Zeit gibt es ein aktuelles Verzeichnis: *working directory*
- Pfade sind absolut (beginnen mit '/') oder relativ (beginnen *nicht* mit '/')
- Relative Pfade werden vom momentanen working directory aus interpretiert, absolute immer vom root-Verzeichnis.
- Jeder User hat ein *home directory*
  - Nach dem Einloggen befindet man sich im *home directory*
  - Dort kann man schreiben und lesen wie man möchte

Pfade eingeben

Besondere Einträge

- `.` Aktuelles Verzeichnis
- `..` Ein Verzeichnis weiter oben im Baum ("Parent-directory")

Wildcards

- `?` Steht für *ein* beliebiges Zeichen
  `h?llo` passt auf `hallo` oder `hello`, nicht aber auf `halllo`
- `*` Steht für beliebig viele beliebige Zeichen
  `*.txt` passt auf alle Dateinamen, die auf `.txt` enden.
  `a*` passt auf alle Dateinamen, die mit `a` beginnen.

Pfade eingeben

Besondere Einträge

`.` Aktuelles Verzeichnis

`..` Ein Verzeichnis weiter oben im Baum ("Parent-directory")

Wildcards

? Steht für *ein* beliebiges Zeichen
`h?llo` passt auf `hallo` oder `hello`, nicht aber auf `halllo`

* Steht für beliebig viele beliebige Zeichen
`*.txt` passt auf alle Dateinamen, die auf `.txt` enden.
`a*` passt auf alle Dateinamen, die mit `a` beginnen.

Tilde-Expansion

- ~ wird automatisch durch die $HOME Variable ersetzt
- Funktion der Shell, nicht des Dateisystems
- Aufgerufene Programme bekommen von diesem Prozess nichts mit:

```
$ echo ~/test.txt
/home/mitarb/steen/test.txt
```

- Pfade mit ~ werden nicht in jeder Umgebung automatisch expandiert. Beispiel Python:

```
os.listdir("~") # Keine Expansion
os.listdir(os.path.expanduser("~")) # Expansion
```

Linux I

Linux I

Prozesse und Programme

Programm Anweisungen, die nacheinander ausgeführt werden. In Dateien gespeichert; kompiliert oder interpretiert.

Prozess Ein gestartetes Programm läuft in einem eigenen Prozess. Prozess läuft in einem „Kontext": Rechte, Working Directory, Umgebungsvariablen, Dateideskriptoren, ...

Subprozesse

- Alle Prozesse auf einem Computer sind hierarchisch angeordnet.
- Wird ein neuer Prozess gestartet, läuft er als Subprozess des Prozesses, von dem er gestartet wurde.
- Der Kontext des Elternprozesses wird übernommen.

Prozesse

Laufende Prozesse

- **ps** ohne Parameter zeigt laufende Prozesse, die vom momentanen Benutzer im laufenden Terminal gestartet wurden.

Beispiel (`:~$ ps -jH`)

```
  PID  PGID   SID TTY          TIME CMD
10185 10185 10185 pts/0    00:00:01 bash
10387 10387 10185 pts/0    00:00:00   ps
```

bash Die Shell selbst

ps Das ps-Kommando

Option **-e** zeigt alle laufenden Prozesse (every)
Interaktive Alternative:

```
:~$ top
```

Prozesse

Laufende Prozesse

- **ps** ohne Parameter zeigt laufende Prozesse, die vom momentanen Benutzer im laufenden Terminal gestartet wurden.

Beispiel (`:~$ ps -jH`)

```
  PID  PGID   SID TTY          TIME CMD
10185 10185 10185 pts/0    00:00:01 bash
10387 10387 10185 pts/0    00:00:00   ps
```

bash Die Shell selbst

ps Das ps-Kommando

Option **-e** zeigt alle laufenden Prozesse (every)

Interaktive Alternative:

```
:~$ top
```

Prozesse

Laufende Prozesse

- `ps` ohne Parameter zeigt laufende Prozesse, die vom momentanen Benutzer im laufenden Terminal gestartet wurden.

Beispiel (`:~$ ps -jH`)

```
  PID  PGID   SID TTY          TIME CMD
10185 10185 10185 pts/0    00:00:01 bash
10387 10387 10185 pts/0    00:00:00   ps
```

bash Die Shell selbst

ps Das ps-Kommando

Option `-e` zeigt alle laufenden Prozesse (every)

Interaktive Alternative:

```
:~$ top
```

Subprozesse erzeugen

Beispiel (`:~$ sleep 5m`)

`sleep` startet in einem Subprozess, der den gleichen Kontext hat wie die Shell.
Solange **`sleep`** läuft, wartet die Shell.

Beispiel (`:~$ sleep 5m &`)

Ein & am Ende des Kommandos schiebt den Prozess in den Hintergrund.
Die Shell wartet dann nicht, bis der Prozess beendet ist.

Subprozesse erzeugen

Beispiel (`:~$ sleep 5m`)

`sleep` startet in einem Subprozess, der den gleichen Kontext hat wie die Shell.
Solange **`sleep`** läuft, wartet die Shell.

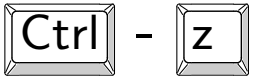
Beispiel (`:~$ sleep 5m &`)

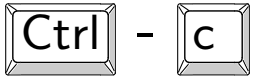
Ein & am Ende des Kommandos schiebt den Prozess in den Hintergrund.
Die Shell wartet dann nicht, bis der Prozess beendet ist.

Subprozesse managen

- Das Kommando `jobs` zeigt eine Liste aller Prozesse im Hintergrund an
- Mit `fg` und `bg` lassen sich Prozesse in den Hintergrund schieben oder in den Vordergrund holen
- Ein laufender Prozess kann mit Ctrl - z pausiert und mit Ctrl - c abgebrochen werden
- Achtung: damit ein pausierter Prozess weiterläuft, muss er mit `bg` in den Hintergrund geschoben werden

Subprozesse managen

- Das Kommando `jobs` zeigt eine Liste aller Prozesse im Hintergrund an
- Mit `fg` und `bg` lassen sich Prozesse in den Hintergrund schieben oder in den Vordergrund holen
- Ein laufender Prozess kann mit 

pausiert und mit 

abgebrochen werden
- Achtung: damit ein pausierter Prozess weiterläuft, muss er mit `bg` in den Hintergrund geschoben werden

Remote arbeiten: SSH

- **`ssh`** stellt eine *sichere* Verbindung zu einem anderen Computer her
- Argument: **`hostname`**
- Optionen:
  - **`-l <user>`** Der Username auf dem anderen Computer
  - **`-X`** Erlaubt das Benutzen graphischer Programme
- Ist der Username auf beiden Computern gleich, kann er weggelassen werden (das ist am ICL der Fall!)

Beispiele

- **`$ ssh -l steen ella`**
- **`$ ssh steen@ella`**
- **`$ ssh ella`**

Remote arbeiten

- Prozesse werden beendet, wenn die SSH-Verbindung abbricht (z.B. durch Schließen des Terminalfensters)
- Um Prozesse weiterlaufen zu lassen (etwa: die ganze Nacht / über das Wochenende), muss man Prozesse *vorher* richtig starten, zum Beispiel mit **`screen`**

screen

- Erlaubt *persistente* Sitzungen
- Zwischenschicht zwischen Terminal und Rechner

Beispiel

```
$ ssh ella
ella$ screen
ella$ # do something
ella$ C-a d
[detached]
ella$ # do something else
ella$ screen -r
```

Es gibt Alternativen zu `screen`, z.B. `tmux` (`http://tmux.sourceforge.net/`)

screen

- Erlaubt *persistente* Sitzungen
- Zwischenschicht zwischen Terminal und Rechner

Beispiel

```
$ ssh ella
ella$ screen
ella$ # do something
ella$ C-a d
[detached]
ella$ # do something else
ella$ screen -r
```

Es gibt Alternativen zu `screen`, z.B. `tmux` (`http://tmux.sourceforge.net/`)

screen

- Erlaubt *persistente* Sitzungen
- Zwischenschicht zwischen Terminal und Rechner

Beispiel

```
$ ssh ella
ella$ screen
ella$ # do something
ella$ C-a d
[detached]
ella$ # do something else
ella$ screen -r
```

Es gibt Alternativen zu `screen`, z.B. `tmux` (`http://tmux.sourceforge.net/`)

Public key authentication

- Alternative zum Passwort: Public key authentication
  - Kein ständiges Eintippen der Passwörter (z.B. bei Skripten)
  - Sehr sichere Authentifizierungsvariante
- Basiert auf Key-Pair

  Public Key wird auf dem Server abgelegt

  Private Key Verbleibt beim Nutzer. Kenntnis des privaten Schlüssels erlaubt Anmeldung am Server.

Key-Pair erstellen

1. `ssh-keygen -t rsa -b 4096 -C ''steen@cl.uni-heidelberg.de''` erzeugt Key-Pair.
   - Das Script stellt bei der Generierung einige Fragen
   - Die Defaulteinstellungen sind normalerweise in Ordnung
   - Eine Passphrase verschlüsselt den Schlüssel und verhindert Diebstahl
2. `ssh-copy-id -i  /.ssh/id_rsa steen@ella` kopiert öffentlichen Schlüssel auf ella.

Linux I

Ein- und Ausgabe

- Stream/Strom: Ein Kanal, in dem zeichenweise gelesen und geschrieben wird
- Drei Streams werden automatisch für jeden Prozess geöffnet:

  `STDOUT` Standardausgabe-Kanal – im Normalfall das Terminal

  `STDIN` Standardeingabe-Kanal, von dem der Prozess lesen kann – im Normalfall sind das Tastatureingaben

  `STDERR` Standard-Fehlerausgabe – Fehlermeldungen können entweder angezeigt oder gespeichert werden (oder beides)

Standardstreams

Beispiel (Terminal)

`STDOUT` Das Fenster am Bildschirm

`STDERR` Das Fenster am Bildschirm, allerdings ungepuffert

`STDIN` Die Tastatur

Prozesse, die im Terminal gestartet werden, „erben" diese Eigenschaften, wenn sie nicht explizit umgeleitet werden

Streams umleiten I

Ausgabe umleiten

Mit dem Zeichen $>$ am Ende eines Kommandos kann man die (Standard-)Ausgabe des Kommandos in eine Datei umleiten

Beispiel (`:~$ ps > processes.txt`)

Keine Ausgabe mehr im Terminal, stattdessen stehen die Informationen jetzt in der Datei **`processes.txt`**, wovon man sich mit **`less`** überzeugen kann.

Streams umleiten II

Eingabe umleiten

Mit dem Zeichen $<$ am Ende eines Kommandos wird die Standardeingabe des Kommandos aus einer Datei gefüllt

Beispiel (`:~$ sort < processes.txt`)

```
  26331   pts/9    00:00:00   bash
  31920   pts/9    00:00:00   ps
    PID     TTY        TIME   CMD
```

Streams umleiten III

Umleitungen

`>`	`STDOUT` umleiten, alte Datei wird gelöscht
`>>`	`STDOUT` umleiten, an alte Datei wird angehängt
`<`	`STDIN` umleiten
`2>`	`STDERR` umleiten
`&>`	`STDOUT` und `STDERR` umleiten
`2>&1`	`STDERR` an `STDOUT` senden

Pipe

Beispiel

```
:~$ ps > processes.txt
```

```
:~$ sort < processes.txt
```

Beispiel

```
:~$ ps | sort
```

Pipe

- Pipe | kombiniert verschiedene Befehle
- Prozess 2 bekommt STDOUT von Prozess 1 als STDIN

Pipe

Beispiel

```
:~$ ps > processes.txt
```

```
:~$ sort < processes.txt
```

Beispiel

```
:~$ ps | sort
```

Pipe

- Pipe | kombiniert verschiedene Befehle
- Prozess 2 bekommt STDOUT von Prozess 1 als STDIN

Linux I

Escaping

" und '

- Zeichenketten (*Strings*) werden mit Anführungszeichen " oder Apostrophen ' begrenzt
- Um Anführungszeichen oder Apostrophen als Zeichen zu verwenden, müssen sie *escaped* werden
- Escaping wird in Linux/Unix mit einem Backslash \ gemacht

Beispiele (Escaping)

- "My sister's friend's investments" ✓
- 'My sister\'s friend\'s investments' ✓
- 'My sister's friend's investments' X

Escaping

" und '

- Zeichenketten (*Strings*) werden mit Anführungszeichen " oder Apostrophen ' begrenzt
- Um Anführungszeichen oder Apostrophen als Zeichen zu verwenden, müssen sie *escaped* werden
- Escaping wird in Linux/Unix mit einem Backslash \ gemacht

Beispiele (Escaping)

- "My sister's friend's investments" ✓
- 'My sister\'s friend\'s investments' ✓
- 'My sister's friend's investments' X

Escaping

" und '

- Zeichenketten (*Strings*) werden mit Anführungszeichen " oder Apostrophen ' begrenzt
- Um Anführungszeichen oder Apostrophen als Zeichen zu verwenden, müssen sie *escaped* werden
- Escaping wird in Linux/Unix mit einem Backslash \ gemacht

Beispiele (Escaping)

- "My sister's friend's investments" ✓
- 'My sister\'s friend\'s investments' ✓
- 'My sister's friend's investments' X

Sonderzeichen

Nicht-druckbare Zeichen

- Bestimmte Zeichen des normalen Zeichensatzes sind *nicht druckbar*.
- Dazu gehören zum Beispiel Zeilenumbrüche, Wagenrückläufe, Tabulatoren oder die Backspace-Taste.
- Mit diesen Zeichen kann dennoch normal gearbeitet werden – Escaping macht es möglich.
- Für jedes dieser Steuerzeichen ist ein Escape Code der Form \<zeichen> definiert

Escape Codes

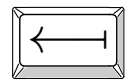
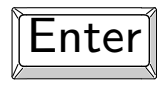
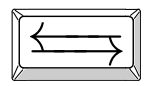
Escape Code	Name	Tastatursymbol
\b	Backspace	
\c	Control	
\f	Form feed	
\n	New line, return	
\r	Carriage return	
\t	Tab	
\\	Backslash	

Tabelle: Ausgewählte Escape Codes

Zeilenumbrüche

DOS/Windows	Carriage Return, New line	\r \n
Unix (Linux, Mac OS X)	New line	\n
Mac OS	Carriage Return	\r

Tabelle: Zeilenende nach Betriebssystem

Umwandlung

`dos2unix` , **`unix2dos`** , **`tr`** , **`sed`**

Editoren

- Text-Editoren unterstützen manuelle Texteingabe
- In unserem Kontext: Dokumentation, Quelltext, Daten, ...
- Zwei große Editoren: `Emacs` und `vi`
- Sie sollten sich mit einem von beiden beschäftigen.

Features

- Syntax-Highlighting für *alle* Programmiersprachen/Formate
- Suchfunktion (reguläre Ausdrücke!), Suchen und Ersetzen
- Copy-Cut-Paste, Undo
- Universelle Verfügbarkeit

Editoren

- Text-Editoren unterstützen manuelle Texteingabe
- In unserem Kontext: Dokumentation, Quelltext, Daten, ...
- Zwei große Editoren: `Emacs` und `vi`
- Sie sollten sich mit einem von beiden beschäftigen.

Features

- Syntax-Highlighting für *alle* Programmiersprachen/Formate
- Suchfunktion (reguläre Ausdrücke!), Suchen und Ersetzen
- Copy-Cut-Paste, Undo
- Universelle Verfügbarkeit

Linux I

Reguläre Ausdrücke

- Beschreiben eine Menge von Zeichenfolgen
- Ist eine gegebene Zeichenfolge Teil dieser Menge, *matcht* der Ausdruck auf die Zeichenfolge.

Anwendungsbeispiele

- Entfernen von Zeichen und Zeichenketten aus Texten
- Umwandeln von Zeichen in andere Zeichen
- Herausfinden, ob Zeichenketten ein bestimmtes Format haben

Reguläre Ausdrücke

- Beschreiben eine Menge von Zeichenfolgen
- Ist eine gegebene Zeichenfolge Teil dieser Menge, *matcht* der Ausdruck auf die Zeichenfolge.

Anwendungsbeispiele

- Entfernen von Zeichen und Zeichenketten aus Texten
- Umwandeln von Zeichen in andere Zeichen
- Herausfinden, ob Zeichenketten ein bestimmtes Format haben

Varianten

- Keine einheitliche Syntax für reguläre Ausdrücke
- Insbesondere Kommandozeilen-Tools nutzen oft verschiedene Varianten
- Wir schauen uns *POSIX* genauer an, da viele Kommandozeilentools POSIX oder eine Abwandlung von POSIX verwenden

Linux I

Basic Regular Expressions

- Zeichen matchen auf sich selbst
- `.` matcht ein (!) beliebiges Zeichen
- `^` matcht den Anfang der Zeichenkette oder der Zeile
- `$` matcht das Ende der Zeichenkette oder der Zeile
- `*` matcht das vorherige Zeichen 0 oder mehr Male
- `\{m, n\}` matcht das vorherige Zeichen mindestens m und maximal n mal

BRE – Bracket Expression

- `[ ]` matcht jedes einzelne Zeichen, das zwischen den Klammern steht (bracket expression)
- `[^ ]` matcht jedes einzelne Zeichen, das *nicht* zwischen den Klammern steht

Beispiele

- Einzelne Zeichen: `[aeiou]`
- Nicht einzelne Zeichen: `[^a-d]`
- Bereiche: `[a-d0-3]`
- Einfacher Bindestrich am Anfang oder Ende: `[-a-z]`

BRE – Bracket Expression

- `[ ]` matcht jedes einzelne Zeichen, das zwischen den Klammern steht (bracket expression)
- `[^ ]` matcht jedes einzelne Zeichen, das *nicht* zwischen den Klammern steht

Beispiele

- Einzelne Zeichen: `[aeiou]`
- Nicht einzelne Zeichen: `[^a-d]`
- Bereiche: `[a-d0-3]`
- Einfacher Bindestrich am Anfang oder Ende: `[-a-z]`

BRE – Backreferences

- `\( \)` Ein Block für Backreferences
- `\n` matcht den n-ten Block ($1 \leq n \leq 9$)

Beispiele

- Zwei gleiche Buchstaben hintereinander: `\(.\)\1`
- Weiter entfernte Referenz: `\(.\)h\1` (aha, oho, uhu, ...)
- Maximal neun Blöcke

BRE – Backreferences

- `\( \)` Ein Block für Backreferences
- `\n` matcht den n-ten Block ($1 \leq n \leq 9$)

Beispiele

- Zwei gleiche Buchstaben hintereinander: `\(.\)\1`
- Weiter entfernte Referenz: `\(.\)h\1` (aha, oho, uhu, ...)
- Maximal neun Blöcke

BRE – Escaping

- Um ein Zeichen zu matchen, das im Ausdruck eine Bedeutung hat, muss es *escaped* werden
- Dazu benutzt man den Backslash: `\`
- Folgende Zeichen müssen escaped werden: `[`, `]`, `.`, `*`, `^`, `$`, `\`.
- Runde und geschweifte Klammern werden *nicht* escaped, da diese ja escaped verwendet werden.

Zeichenklassen

`[[:alnum:]]`	`[a-zA-Z0-9]`
`[[:alpha:]]`	`[a-zA-Z]`
`[[:digit:]]`	`[0-9]`
`[[:lower:]]`	`[a-z]`
`[[:punct:]]`	Punctuation and Symbols
`[[:space:]]`	`[\t\r\n\v\f]`
`[[:upper:]]`	`[A-Z]`
`[[:word:]]`	`[a-zA-Z0-9_]`

Tabelle: Zeichenklassen

Extended Regular Expressions

- Erweiterung zu BRE
- Meta-characters (`\(, \), \{ \}`) werden nicht mehr escaped.
  `\(` matcht jetzt "("anstatt einen Block aufzumachen
- `?` matcht das vorige Zeichen 0 oder 1 mal
- `+` matcht das vorige Zeichen 1 oder mehr male
- `a|b` matcht "a" oder "b"
- Bei vielen Kommandozeilentools werden Extended REs mit der Option `-E` aktiviert

Linux I

Matching

`grep`

- Mit **`grep`** kann man Dateien nach Vorkommen von Zeichenketten durchsuchen
- In der Standardeinstellung werden die Zeilen, in welchen die gesuchten Zeichenketten vorkommen, ausgegeben
- Die Zeichenketten können durch reguläre Ausdrücke definiert werden

Beispiel (`:~$ grep '^$' file.txt`)

Findet alle leeren Zeilen in `file.txt`

Ersetzung

`sed`

- Werkzeug, um Datenströme zu bearbeiten
- Insbesondere für Ersetzungsregeln mit regulären Ausdrücken nützlich
- Mit dem Ausdruck **`s/a/b/g`** wird jedes Vorkomen von a durch b ersetzt
- Modifikatoren können angehängt werden
- Statt / sind auch andere Zeichen möglich, z.B. `s#/ein/pfad/#/ein/anderer/pfad#g`

Beispiel (`:~$ sed 's/groß/klein/g' file.txt`)

Ersetzt jede in **`file.txt`** vorkommende Zeichenfolge „groß" durch die Zeichenfolge „klein"

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
  Ersetzt jedes space-Zeichen durch einen Zeilenumbruch. Jedes „Wort” steht dann auf einer eigenen Zeile
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`

- **`$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`**

- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`

- **`$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`**
  Ersetzt Space gefolgt von Punctuation-Zeichen durch Space.
- `$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- **`$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`**

Ersetzung – Beispiele

Beispiele

- `$ sed 's/[[:space:]]/\n/g' file.txt`
- `$ sed 's/\([[:space:]]\)[[:punct:]]/\1/g'`
- **`$ sed 's/\([[:lower:]]\)\([[:upper:]]\)/\1 \2/g'`**
  Ersetzt CamelCase-Ausdrücke durch Camel Case

Übung 1

Linux I

Linux I

Variablen in der Shell

In der Shell können Variablen verwendet werden.

- Deklaration: Variablenname, Gleichheitszeichen, Wert (keine Leerzeichen!)
- Benutzung: Dollarzeichen, Variablenname
- Variablen sind grundsätzlich *ungetypt*!

Beispiel

```
$ VARNAME=Hallo
$ echo $VARNAME
Hallo
$ echo $VARNAME Welt!
Hallo Welt!
$ echo $VARNAME2

$ echo ${VARNAME}2
Hallo2
```

Variablen in der Shell

In der Shell können Variablen verwendet werden.

- Deklaration: Variablenname, Gleichheitszeichen, Wert (keine Leerzeichen!)
- Benutzung: Dollarzeichen, Variablenname
- Variablen sind grundsätzlich *ungetypt*!

Beispiel

```
$ VARNAME=Hallo
$ echo $VARNAME
Hallo
$ echo $VARNAME Welt!
Hallo Welt!
$ echo $VARNAME2

$ echo ${VARNAME}2
Hallo2
```

Variablen in Strings

" und ' verhalten sich unterschiedlich!

- Double quotes: Variablenreferenzen werden ersetzt
- Apostroph/Single quote: Variablenreferenzen werden nicht ersetzt

Beispiele

```
$ TEST="A B"
$ echo "$TEST"
A B
$ echo '$TEST'
$TEST
$ TEST="A     B"
#Zeichenkette wird an Whitespace gesplittet
$ echo $TEST
A B
$ echo "$TEST"
A     B
```

Backticks

- Kommandos innerhalb von Backticks werden ausgeführt
- Standardausgabe wird als Zeichenkette zurückgegeben
- Alternative: `$( ...)`

Beispiele

```
# Variable TEST enthält Ausgabe von ls -la
$ TEST=`ls -la`
$ TEST=$( ls -la )
# Inhalt von Variable TEST wird ausgegeben
$ echo $TEST
... Ausgabe von ls -la ...
```

Arithmetik

- Operatoren +,-,*,/,**,%,
- Einfache Befehle ausführen:

```
:~$ expr 1 + 1
2
```

- Ergebnis eines arithmetischen Ausdrucks in eine Variable schreiben:

```
$ VAR1=$((1+1))
$ ((VAR2 = 1 + 1)) # Spaces optional
$ let "VAR3=1+1" # Spaces optional
$ VAR4=$((VAR3**2))
```

- In (( )) müssen Variablen nicht mit $ referenziert werden.
- Floating Point Arithmetik ist auf der Shell nicht möglich.

Umgebungsvariablen

- Werden vom Betriebssystem beim Einloggen belegt
- Können auch geändert werden,
  allerdings sollte man wissen, was man tut

`$HOME`	Der Pfad zum Homeverzeichnis
`$PWD`	Working Directory
`$OLDPWD`	Previous Working Directory
`$_`	Letztes Argument des letzten Kommandos
`$PATH`	Der Suchpfad für ausführbare Programme
`$TERM`	Angaben über das Terminal
`$OSTYPE`	Angaben über das Betriebssystem
`$BASH_VERSION`	Version der Shell
`$USER`	Der Name des Users
...	...

Exkurs: Suchpfad

- Der Suchpfad (`$PATH`) gibt an, in welchen Verzeichnissen ausführbare Dateien liegen
- Programme werden ohne genaue Pfadangabe gefunden
- Verzeichnisse werden durch Doppelpunkt getrennt
- Alle Programme (nicht aber build-ins) werden auf diese Weise gefunden: `ls` , `grep` , ...
- Mit `export` kann der Suchpfad geändert werden, so dass auch in anderen Verzeichnissen gesucht wird

```
:~$ export PATH=$PATH:/the/new/path
```

Umgebungsvariablen deklarieren

- Eine Variable wird zur Umgebungsvariable, indem das Schlüsselwort **`export`** vor die Deklaration gestellt wird

Beispiel

```
# VARNAME1 ist eine Umgebungsvariable
$ export VARNAME1=Hallo

# VARNAME2 ist eine normale Variable
$ VARNAME2=Welt
```

- Wozu sollte man Umgebungsvariablen selber deklarieren?

Umgebungsvariablen deklarieren

- Eine Variable wird zur Umgebungsvariable, indem das Schlüsselwort **`export`** vor die Deklaration gestellt wird

Beispiel

```
# VARNAME1 ist eine Umgebungsvariable
$ export VARNAME1=Hallo

# VARNAME2 ist eine normale Variable
$ VARNAME2=Welt
```

- Wozu sollte man Umgebungsvariablen selber deklarieren?

Subprozesse (Wiederholung)

- Alle Prozesse auf einem Computer sind hierarchisch angeordnet.
- Wird ein neuer Prozess gestartet, läuft er als Subprozess des Prozesses, von dem er gestartet wurde.
- Der Kontext des Elternprozesses wird übernommen.

Variablen in Subprozessen

- (Normale) Variablen sind in Subprozessen nicht (mehr) verfügbar

Beispiel

```
# Deklaration einer Variablen
$ VAR=hallo

# Start eines Subprozesses
$ bash

# Ausgabe des Inhaltes der Variable VAR
$ echo "VAR=$VAR"
VAR=
```

Umgebungsvariablen in Subprozessen

- Umgebungsvariablen sind in Subprozessen verfügbar.

Beispiel

```
# Deklaration der Umgebungsvariable
$ export VAR=hallo

# Start eines Subprozesses
$ bash

# Ausgabe des Inhalts der Variablen
$ echo "VAR=$VAR"
VAR=hallo
```

Linux I

Shell-Skripte

- Anstatt Befehle auf der Shell per Hand einzugeben kann man sie auch automatisch sequenziell verarbeiten lassen.
- Befehle werden in Dateien geschrieben.
- Die Dateien (sog. Shell-Skripte) werden ausgeführt wie ein Programm.
- Einrückung spielt grundsätzlich *keine* Rolle.
- Alle Konstruktionen, die wir bisher kennengelernt haben, sind in Skripten verwendbar:
  - Befehle, Programme
  - Ein/Ausgabeumleitungen, Pipes
  - Variablen und Umgebungsvariablen
  - ...

Shell-Skripte

- Anstatt Befehle auf der Shell per Hand einzugeben kann man sie auch automatisch sequenziell verarbeiten lassen.
- Befehle werden in Dateien geschrieben.
- Die Dateien (sog. Shell-Skripte) werden ausgeführt wie ein Programm.
- Einrückung spielt grundsätzlich *keine* Rolle.
- Alle Konstruktionen, die wir bisher kennengelernt haben, sind in Skripten verwendbar:
  - Befehle, Programme
  - Ein/Ausgabeumleitungen, Pipes
  - Variablen und Umgebungsvariablen
  - ...

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH` (relativer Pfad)

    ```
    :~$ ./script.sh
    ```

    Punkt beachten!
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```

  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```

  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```

  - Ausführbare Datei, script.sh nicht in `$PATH` (relativer Pfad)

    ```
    :~$ ./script.sh
    ```

    Punkt beachten!

- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```

  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```

  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```

  - Ausführbare Datei, script.sh nicht in `$PATH` (relativer Pfad)

    ```
    :~$ ./script.sh
    ```

    Punkt beachten!

- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH` (relativer Pfad)

    ```
    :~$ ./script.sh
    ```

    Punkt beachten!
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH` (relativer Pfad)

    ```
    :~$ ./script.sh
    ```

    Punkt beachten!
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH` (relativer Pfad)

    ```
    :~$ ./script.sh
    ```

    Punkt beachten!
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ein einfaches Shell-Skript I

- Am Anfang jedes Shell-Skripts steht die sog. Shebang: `#!/bin/bash`
- Danach folgen beliebige Kommandos: Shell-Builtins sowie Programme des Systems

Ein einfaches Shell-Skript II

Beispiel (Backup-Szenario)

- Sicherungskopie von Verzeichnis **`Documents`**
- Ziel: Verzeichnis **`/mnt/backup`**

Beispiel (backup1.sh)

```
#!/bin/bash

cp -r /home/nils/Documents /mnt/backup
```

Ein Shell-Skript mit Variablen

- Variablen und Umgebungsvariablen lassen sich einfach einbauen
- Sie werden dann *zur Laufzeit* ersetzt

Beispiel (backup2.sh)

```
#!/bin/bash

cp -r $HOME/Documents /mnt/backup/$USER
```

Positionsvariablen

- Einige Variablen ergeben nur in Verbindung mit Skripten Sinn
- Sie erlauben Zugriff auf den Orts des Skripts und die Argumente

`$0`	Dateiname des Skripts
`$1, $2, …`	Parameter, die dem Skript auf der Kommandozeile mitgegeben wurden
`$#`	Anzahl der Parameter
`$@`	Die Parameter als Liste

Ein Shell-Skript mit Positionsvariablen

Beispiel (backup3.sh)

```
#!/bin/bash

# Kopiert das als Argument angegebene Verzeichnis
# aus dem Homeverzeichnis des angemeldeten Users
cp -r $HOME/$1 /mnt/backup/$USER
```

Beispiel

```
$ backup3.sh Documents
```

Ein Shell-Skript mit Positionsvariablen

Beispiel (backup3.sh)

```
#!/bin/bash

# Kopiert das als Argument angegebene Verzeichnis
# aus dem Homeverzeichnis des angemeldeten Users
cp -r $HOME/$1 /mnt/backup/$USER
```

Beispiel

```
$ backup3.sh Documents
```

Linux I

Kontrollstrukturen

- Die Bash bietet auch Kontrollstrukturen
  - `if`
  - `while` , `until`
  - `case`
  - `for`
  - `select`
- Damit lassen sich vollwertige Programme in der Shell realisieren

Kontrollstrukturen: if

- `if` führt einen angegebenen Test aus
- Wenn der Test 0 zurückgibt, wird der Code ausgeführt
- Optional: Sonst wird anderer Code ausgeführt

Warum Null?

- Traditionellerweise ist 0 in der Unix-Welt das Zeichen für Erfolg
- Als Merkhilfe: die Zahl repräsentiert die Zahl der Fehler
- Wenn 0 Fehler aufgetreten sind, ist alles gutgegangen

Kontrollstrukturen: if

- `if` führt einen angegebenen Test aus
- Wenn der Test 0 zurückgibt, wird der Code ausgeführt
- Optional: Sonst wird anderer Code ausgeführt

Warum Null?

- Traditionellerweise ist 0 in der Unix-Welt das Zeichen für Erfolg
- Als Merkhilfe: die Zahl repräsentiert die Zahl der Fehler
- Wenn 0 Fehler aufgetreten sind, ist alles gutgegangen

Kontrollstrukturen: if und Tests

Syntax

```
if TEST
then
    COMMANDS
else
    COMMANDS
fi
```

`COMMANDS`

Beliebige Kommandos

`TEST`

- `[[ … ]]`
  Extended Test Command
- Vergleiche
  - Zeichenketten
  - Zahlen
  - Daten
  - …
- Dateitests
  - Existenz
  - Letzte Änderung
  - Datei/Verzeichnis
  - …

Kontrollstrukturen: if und Tests

Syntax

```
if TEST
then
    COMMANDS
else
    COMMANDS
fi
```

COMMANDS

Beliebige Kommandos

TEST

- `[[ … ]]`
  Extended Test Command
- Vergleiche
  - Zeichenketten
  - Zahlen
  - Daten
  - …
- Dateitests
  - Existenz
  - Letzte Änderung
  - Datei/Verzeichnis
  - …

Kontrollstrukturen: if und Tests

Syntax

```
if TEST
then
    COMMANDS
else
    COMMANDS
fi
```

COMMANDS

Beliebige Kommandos

TEST

- `[[ … ]]`
  Extended Test Command
- Vergleiche
  - Zeichenketten
  - Zahlen
  - Daten
  - …
- Dateitests
  - Existenz
  - Letzte Änderung
  - Datei/Verzeichnis
  - …

Ein Skript mit Test

Beispiel (backup4.sh)

```
#!/bin/bash

# Kopiert das als Argument angegebene Verzeichnis
# aus dem Homeverzeichnis des angemeldeten Users

# Testet, ob es existiert
if [[ -e $HOME/$1 ]]
then
        cp -r $HOME/$1 /mnt/backup/$USER
else
        # Gibt Fehlermeldung aus, wenn nicht
        echo "$HOME/$1 does not exist."
fi
```

Ein Skript mit verschalteten Tests

Beispiel (backup4b.sh)

```
#!/bin/bash

# Testet, ob Verzeichnis existiert
if [[ -e $HOME/$1 ]]
then
        # Testet, ob es ein Verzeichnis ist
        if [[ -d $HOME/$1 ]]
        then
                cp -r $HOME/$1 /mnt/backup/$USER
        else
                echo "$HOME/$1 is not a directory."
        fi
else
        # Gibt Fehlermeldung aus, wenn nicht
        echo "$HOME/$1 does not exist."
fi
```

Vergleichsoperatoren

Integer

`-eq`	is equal to
`-ne`	is not equal to
`-gt`	is greater than
`-ge`	is greater than or equal to
…	…

String

`=`	is equal to
`<`	is less than (ASCII order)
`-z`	is null (has zero length)
`-n`	is not null
…	…

Achtung!

- Single und double quotes und die Art der Klammerung des Testausdrucks ändern das Verhalten des Tests!
- `[[ … ]]`, `[ … ]`, `(( … ))`

Vergleichsoperatoren

Integer

`-eq`	is equal to
`-ne`	is not equal to
`-gt`	is greater than
`-ge`	is greater than or equal to
…	…

String

`=`	is equal to
`<`	is less than (ASCII order)
`-z`	is null (has zero length)
`-n`	is not null
…	…

Achtung!

- Single und double quotes und die Art der Klammerung des Testausdrucks ändern das Verhalten des Tests!
- `[[ … ]]`, `[ … ]`, `(( … ))`

Dateitests

`-e`	Datei existiert
`-f`	Reguläre Datei
`-s`	Dateigröße ungleich Null
`-d`	Datei ist ein Verzeichnis
`-x`	Datei ist ausführbar (für den User, der den Test ausführt)
`f1 -nt f2`	f1 ist neuer als f2
…	…

Referenz

Wo schlägt man das nach?

- Auf der man-page der Bash (`:~$ man bash`)
- Advanced Bash Scripting Guide
  `tldp.org/LDP/abs/html/index.html`
- Man-page als Webseite
  `www.gnu.org/software/bash/manual/bashref.html`

Kontrollstrukturen: for

- Eine **for** -Schleife wiederholt eine Reihe von Anweisungen
- Anzahl der Iterationen vorher festgelegt

Syntax

```
for ARG in [LIST]
do
      COMMANDS
done
```

LIST

- **LIST** ist eine Liste
- Variable **ARG** wird der Reihe nach auf die Elemente der Liste gesetzt

Kontrollstrukturen: for – Beispiel

Beispiel (backup5.sh)

```
#!/bin/bash

# Iteriere ueber alle Argumente der Kommandozeile
for arg in "$@"
do
  if [[ -e $HOME/$arg ]]
  then
    if [[ -d $HOME/$arg ]]
    then
      cp -r $HOME/$arg /mnt/backup/$USER
    else
      echo "$HOME/$arg is not a directory."
    fi
  else
    echo "$HOME/$arg does not exist."
  fi
done
```

Kontrollstrukturen: for – Sequenzen

- Schleifen mit Zählervariable

```
for i in 1 2 3 4 5
do
      COMMANDS
done
```

- Lange Sequenzen: {1..1000}

```
for i in {1..1000}
do
      COMMANDS
done
```

Kontrollstrukturen: for – Sequenzen

- Schleifen mit Zählervariable

```
for i in 1 2 3 4 5
do
      COMMANDS
done
```

- Lange Sequenzen: `{1..1000}`

```
for i in {1..1000}
do
      COMMANDS
done
```

Kontrollstrukturen: for – Dateien

- Automatisch über Dateien in einem Verzeichnis iterieren
- `ls` liefert eine Liste von Dateien in einem Verzeichnis
- `$( … )` wird benutzt, um die Ausgabe von einem Kommando als Wert zuzuweisen
- `$( ls )` liefert eine Liste von Dateien in einem Verzeichnis als Wert für eine Variable

Beispiel

```
for file in $( ls directory )
do
  echo $file
done
```

Iterieren über Dateien – Beispiel

Beispiel (backup6.sh)

```
#!/bin/bash

for arg in "$@"
do
  # Iteriere ueber alle Dateien in $arg
  for directory in $( ls $arg )
  do
    if [[ -d $arg/$directory ]]
    then
      cp -r $arg/$directory /mnt/backup/$USER
    fi
  done
done
```

Übung 2

Mittagspause

Montag: Linux II

2 Linux II
- Encoding and Locale
- AWK

Linux II

Linux II

Was ist Encoding?

- Jedem benötigten Zeichen wird eine Zahl zugeordnet
- Welches Zeichen welcher Zahl zugeordnet wird, bezeichnet man als *Zeichencode*
- Das Format, in dem der Zeichencode (im Computer) repräsentiert wird, heißt *Encoding*

Zahlendarstellung im Computer

- Computer arbeiten binär (= zur Basis 2)
- Speicher: | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
- Ein Bit: | 1 | oder | 0 |
- Acht Bit werden ein Byte genannt und können 256 verschiedene Zustände repräsentieren (z.B. Zahlen zwischen 0 und 255)

1	1	1	0	0	0	1	1

1 Byte

Einschränkungen

- Wieviel Platz ein Zeichen bekommt, entscheidet darüber, wieviele Zeichen man unterscheiden kann
- Ist jedes Zeichen z.B. 4 Bit lang, können 16 Zeichen unterschieden werden
- Alle Zeichen sollten gleich viel Platz belegen; ggf. wird mit Nullen aufgefüllt

Linux II

ASCII

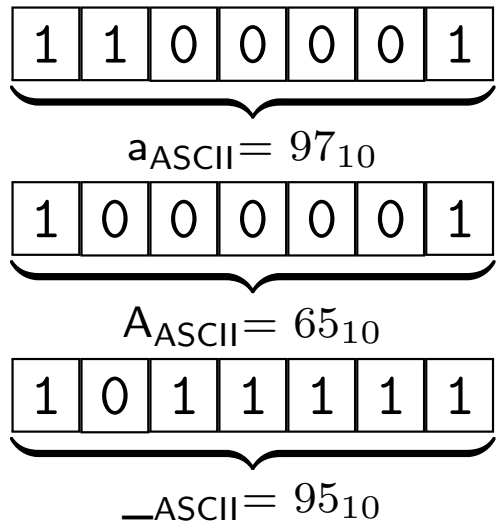
American Standard Code for Information Interchange

- Erstmals publiziert 1963
- Definiert $2^7 = 128$ Codepunkte
- ASCII enthält 33 nicht-druckbare und 95 druckbare Zeichen
- Alle Codepunkte können in 1 Byte gespeichert werden.
- Das 8. Bit dient als Kontrollsignal

Druckbare ASCII-Zeichen

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

Beispiele



Latin-1 / ISO 8859-1

ISO: International Organization for Standardization

- ASCII reicht nicht aus, um z.B. deutsche Umlaute darzustellen
- Erweiterung von ASCII: Latin-1
- In Latin-1 werden alle 8 Bit für 256 Zeichen genutzt
- Die meisten westeuropäischen Sprachen sind darstellbar
- Erste 7 Bit entsprechen denen in ASCII, achtes Bit sorgt dann für 127 zusätzliche Umlaute und Sonderzeichen

Beispiele

Latin-1 / ISO 8859-1

ISO: International Organization for Standardization

- ASCII reicht nicht aus, um z.B. deutsche Umlaute darzustellen
- Erweiterung von ASCII: Latin-1
- In Latin-1 werden alle 8 Bit für 256 Zeichen genutzt
- Die meisten westeuropäischen Sprachen sind darstellbar
- Erste 7 Bit entsprechen denen in ASCII, achtes Bit sorgt dann für 127 zusätzliche Umlaute und Sonderzeichen

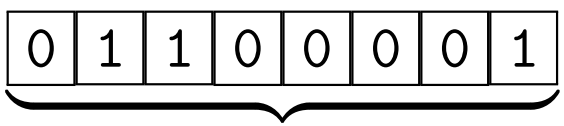
Beispiele

Latin-1 / ISO 8859-1

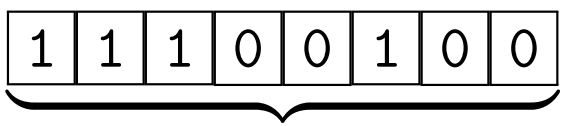
ISO: International Organization for Standardization

- ASCII reicht nicht aus, um z.B. deutsche Umlaute darzustellen
- Erweiterung von ASCII: Latin-1
- In Latin-1 werden alle 8 Bit für 256 Zeichen genutzt
- Die meisten westeuropäischen Sprachen sind darstellbar
- Erste 7 Bit entsprechen denen in ASCII, achtes Bit sorgt dann für 127 zusätzliche Umlaute und Sonderzeichen

Beispiele



$a_{\text{Latin-1}} = a_{\text{ASCII}} = 97_{10}$



$ä_{\text{Latin-1}} = 228_{10}$

Unicode

- Auch Latin-1 reicht nicht aus, um alle Zeichen darzustellen
- Im Chinesischen gibt es mehr als 47.000 verschiedene Zeichen
- 8-Bit-Zeichensätze bieten nur Raum für 256 Zeichen

Unicode

- Kodiert alle Zeichen dieser Welt
- Insgesamt 1.114.112 Codepunkte
- Verschiedene Encodings: UTF-8, UTF-16, UTF-32
- Unterschied zu Latin1: Anzahl der verwendeten Bytes kann variieren (UTF-8, UTF-16)

Unicode

- Auch Latin-1 reicht nicht aus, um alle Zeichen darzustellen
- Im Chinesischen gibt es mehr als 47.000 verschiedene Zeichen
- 8-Bit-Zeichensätze bieten nur Raum für 256 Zeichen

Unicode

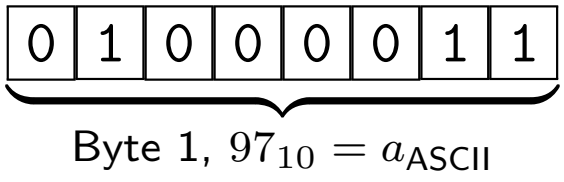
- Kodiert alle Zeichen dieser Welt
- Insgesamt 1.114.112 Codepunkte
- Verschiedene Encodings: UTF-8, UTF-16, UTF-32
- Unterschied zu Latin1: Anzahl der verwendeten Bytes kann variieren (UTF-8, UTF-16)

UTF-8

- ASCII-kompatibel (ASCII-Zeichen werden so wie in ASCII kodiert)
- Nicht-ASCII-Zeichen sind 2-4 Byte lang
- Höchste Bits des ersten Bytes zeigen an, wie lang das Zeichen ist
  - $\boxed{0}$ $\rightarrow$ 1 Byte
  - $\boxed{1}\boxed{1}\boxed{0}$ $\rightarrow$ 2 Byte
  - $\boxed{1}\boxed{1}\boxed{1}\boxed{0}$ $\rightarrow$ 3 Byte
  - $\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}$ $\rightarrow$ 4 Byte
- Darauffolgende Bytes haben alle MSB auf 1 und das nachfolgende auf 0x (MSB: most significant bit, LSB: least significant bit)

UTF-8 – Beispiele

Beispiele

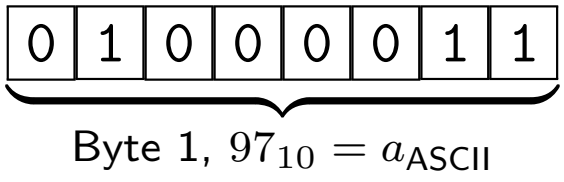
- 

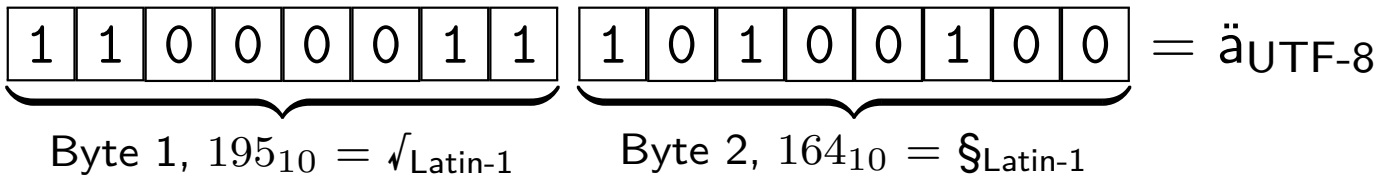
-

-

UTF-8 – Beispiele

Beispiele

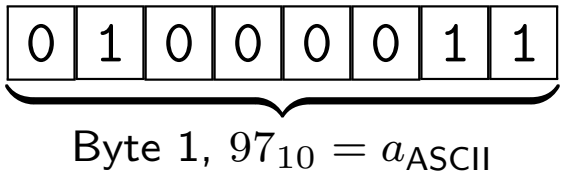
- 

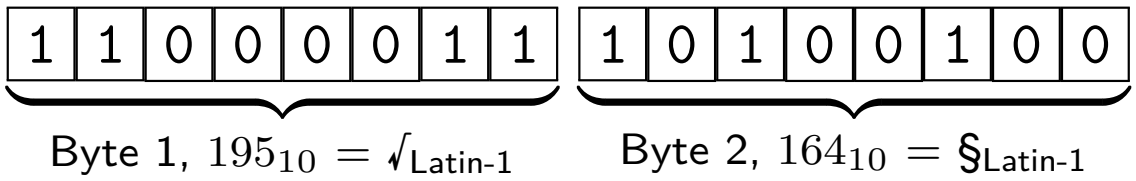
- 

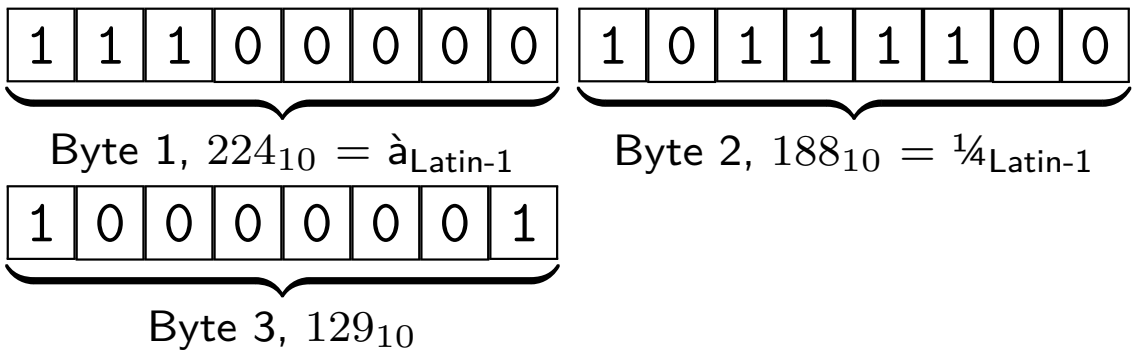
-

UTF-8 – Beispiele

Beispiele

- 

$= \text{a}_{\text{UTF-8}}$
- 

$= \text{ä}_{\text{UTF-8}}$
- 

Encoding erkennen

- Encoding erkennen ist schwer!
- Validierung von Bitfolgen als gültige oder ungültige Zeichen ist erst mit Unicode wieder möglich
- Viele Encodings haben Varianten, die in einzelnen Zeichen abweichen

Hex	Dez	8859-1	8859-2	8859-8	MacRoman
BE_{16}	190	¾	ž	¾	æ
CF_{16}	207	Ï	Ď		œ
$E4_{16}$	228	ä	ä		‰

Tabelle: Unterschiede zwischen Latin-1, Latin-2, Hebrew und MacRoman

Encoding erkennen

- Encoding erkennen ist schwer!
- Validierung von Bitfolgen als gültige oder ungültige Zeichen ist erst mit Unicode wieder möglich
- Viele Encodings haben Varianten, die in einzelnen Zeichen abweichen

Hex	Dez	8859-1	8859-2	8859-8	MacRoman
BE_{16}	190	¾	ž	¾	æ
CF_{16}	207	Ï	Ď		œ
$E4_{16}$	228	ä	ä		‰

Tabelle: Unterschiede zwischen Latin-1, Latin-2, Hebrew und MacRoman

Konvertierung

- Tools zur automatischen Konvertierung: **`recode`** , **`iconv`**
- Lesen *nicht* von der Standardeingabe
- Ein- und Ausgabe direkt über Dateien

Umwandlung aller Dateien im Verzeichnis (Latin-1 → UTF-8)

```
for file in $( ls directory )
    do
    recode l1..u8 $file
done
```

Linux II

Spracheinstellungen

Nicht nur Encoding – Nicht nur Sprache

- Sprache von Fehlermeldungen, Hilfetexte
- Default-Papiergröße für's drucken (DIN A4 oder US-Letter)
- Währungszeichen (€, $, ¥...)
- Dezimal- und Tausendertrennzeichen (1,000.00 vs. 1.000,00)
- Formatierung von Telefonnummern, Adressen, ...

Locale

- `locale` zeigt die aktuellen Einstellungen an

Spracheinstellungen

Nicht nur Encoding – Nicht nur Sprache

- Sprache von Fehlermeldungen, Hilfetexte
- Default-Papiergröße für's drucken (DIN A4 oder US-Letter)
- Währungszeichen (€, $, ¥…)
- Dezimal- und Tausendertrennzeichen (1,000.00 vs. 1.000,00)
- Formatierung von Telefonnummern, Adressen, …

Locale

- `locale` zeigt die aktuellen Einstellungen an

Spracheinstellungen in Umgebungsvariablen

- Spracheinstellungen werden in Umgebungsvariablen gespeichert

`LC_ALL`	Alle Einstellungen
`LC_CTYPE`	Zeichenklassifikation, Groß- und Kleinschreibung
`LC_COLLATE`	Sortierung
`LC_TIME`	Zeit und Datumsformat
`LC_NUMERIC`	Zahlenangaben
`LC_MONETARY`	Zahlenangaben mit Währung
`LC_MESSAGES`	Fragen, Fehlermeldungen, Dialoge
`LC_PAPER`	Papiergröße
`LC_NAME`	Namenformatierung
`LC_ADDRESS`	Adress- und Ortsangaben
`LC_TELEPHONE`	Telefonnummern
`LC_MEASUREMENT`	Maßeinheiten (Metric, Imperial, ...)

Zeichensatz und Schriftart

- Es reicht nicht, die Umgebungsvariablen zu ändern
- Die Schriftart (des Terminals) muss den Zeichensatz unterstützen
- Ändern der Schriftart:
  `Terminal` $\rightarrow$ `Set Character Encoding` / `Zeichenkodierung`

Ändern der Umgebungsvariablen

- ```
  :~$ export LC_LANG=de_DE.utf8
  ```
- ...
  ```

Linux II

AWK

- AWK ist eine Skriptsprache für strukturierte Textdaten
- GNU-Variante: GAWK
- AWK operiert auf zeilenorientierten Daten
- Jede Zeile repräsentiert ein Dokument, und jedes Dokument besteht aus mehreren Feldern.
- Die Felder sind standardmäßig durch Leerzeichen getrennt, andere Trenner sind möglich.
- Grundidee: Jede Zeile wird als ein Array aufgefasst:
  - `$0`: die komplette Zeile
  - `$1`: das erste Feld
  - ...

Grundlagen

- Ein AWK-Skript besteht aus Bedingungen und Anweisungen

```
Bedingung { Anweisungen }
```

- Wenn eine Bedingung erfüllt ist, wird der Anweisungsblock ausgeführt.
- Ist die Bedingung leer, wird die Anweisung immer ausgeführt.
- Ist die Anweisung leer, wird standardmäßig die gesamte Zeile ausgegeben, wenn die Bedingung greift.

Anweisungen

- Werden immer in `{ }` geschrieben
- Wenn keine Anweisung angegeben wird: **`{ print $0 }`**

Ausdruck	Funktionsaufruf, Wertzuweisung einer Variable (`{ VAR=0 }`)
Kontrollstruktur	**`if`**, **`for`**, ...
Ein- und Ausgabe-Statements	**`print`**, **`getline`**, ...(`{ print "/"$1 }`)
Deletion-Statements	Elemente aus Arrays löschen

`http://www.gnu.org/software/gawk/manual/html_node/Action-Overview.html`

Bedingungen

- Grundsätzlich: jeder `awk`-Ausdruck
- Wenn die Bedingung nicht den Wert 0 bzw. `""` hat, wird die Anweisung ausgeführt.
- Beispiel 1: Regulärer Ausdruck `$1 ~ /[a-c]/`
- Beispiel 2: Vergleich `$1=="word"`
- `BEGIN`: Die folgende Anweisung wird ausgeführt, *bevor* Daten gelesen werden.
- `END`: Die folgende Anweisung wird ausgeführt, *nachdem* alle Daten gelesen wurden.

`http://www.gnu.org/software/gawk/manual/html_node/Pattern-Overview.html`

Beispieldaten

```
290 india nnp
293 large jj
300 person nn
302 four cd
302 perform vbn
302 very rb
303 buddhist nnp
303 system nn
303 term nns
305 culture nn
```

Skriptaufruf

Auf der Kommandozeile

AWK-Skript muss in ' stehen:

```
:~$ gawk '$0 ~ /foo/ {print $1}' data.csv
```

In einer Datei

AWK-Skript kann aus einer Datei gelesen werden

```
:~$ gawk -f script.awk data.csv
```

Beispielskripte

material/sample1.awk

```
$3 == "nn" { print $2 }
```

Beispielskripte

material/sample1.awk

```
$3 == "nn" { print $2 }
```

material/sample2.awk

```
BEGIN { NOUNS=0 }
$3 == "nn" { NOUNS += $1 }
END { print NOUNS }
```

```
:~$ gawk -f material/sample2.awk material/data.csv
1827
```

Built-in Variables

AWK verwendet einige eingebaute Variablen, hier nur zwei wichtige Beispiele:

- FS *field separator* – Zeichen oder regulärer Ausdruck
- NR *number of records* – wieviele Zeilen bereits gelesen wurden

Field Separator ändern

```
:~$ gawk 'BEGIN{FS="\t"}{print $1}'
```

www.gnu.org/software/gawk/manual/gawk.html#Built_
002din-Variables

Weitere Beispiele

Nützliche AWK-Einzeiler

- `http://www.pement.org/awk/awk1line.txt`
- `http://www.catonmat.net/blog/awk-one-liners-explained-part-one/`

Übung 3