Perl-Praxis

Dateien und Daten

Jörn Clausen
joern@TechFak.Uni-Bielefeld.DE

Übersicht

- Kommandozeilen-Parameter
- Informationen über Dateien
- Daten aus Dateien lesen
- Daten in Dateien schreiben
- mit pipes lesen und schreiben

Parameter

- Parameter an Skript übergeben:

```
$ linecount.pl brief.txt dipl.tex
```

- Argumente in @ARGV
- Skriptname nicht enthalten
- typische Verwendung:

```
foreach $name (@ARGV) {
  print "reading file $name\n";
  ...
}
```

Aufgaben

- Schreibe ein Skript, das eine Folge von Zahlen als Argument erwartet und die kleinste und die größte Zahl wieder ausgibt.

Datenverarbeitung mit Perl

- nachträgliche Auflösung des Akronyms „Perl“:

  Pratical Extraction and Report Language

- ...oder auch:

  Pathologically Eclectic Rubbish Lister

- Verarbeitung von Dateien, vor allem Textdateien
- Einlesen und Parsen
- Integration mit Unix-Tools (`cat`, `more`, `head`, `tail`, `wc`, `...`)

Dateitestoperatoren

- Eigenschaften/Existenz von Dateien überprüfen
- von Shells übernommen

```
if (-f "/etc/passwd") { ... }  # is a file
if (-r "/etc/shadow") { ... }  # ooops, passwords readable?
if (-w "/etc/shadow") { ... }  # OOOPS, even writeable????
if (-x "/bin/ls") { ... }      # is executable
if (-d "/var/tmp") { ... }     # directory exists
```

- weitere Tests

Aufgaben

- Überprüfe die Eigenschaften der folgenden Dateien:

```
@files = ( "/etc/motd",
           "/tmp",
           "/kernel",
           "/dev/null",
           "/" );
```

mit Dateien arbeiten

- Abstraktion: *filehandle*
- Datei lesen:

```
open(DATA, 'data.txt');
```

- Rückgabewert zeigt Fehler an
- typisches Idiom mit *short-circuit operator*:

```
open(DATA, 'data.txt') || die "can't open file\n";
```

- Fehlercode/Fehlermeldung in Variable `$!`:

```
open(DATA, 'data.txt') || die "can't open file: $!\n";
```

- Datei schließen:

```
close(DATA);
```

Dateien lesen

- Zeilenweise aus Datei lesen:

```
open(DATA, 'data.txt') || die "can't open file: $!\n";
while ($line = <DATA>) {
  print "got line: $line";
}
close(DATA);
```

- *angle operator*: `<...>`
- liefert am Dateiende `undef`
- Datei komplett lesen:

```
@lines = <DATA>;
foreach $line (@lines) { ... };
```

Aufgaben

- Schreibe ein Programm, das eine Text-Datei einliest und die enthaltenen Zeilen in umgekehrter Reihenfolge wieder ausgibt. Reagiere auf mögliche Fehler (Datei nicht vorhanden, Datei nicht lesbar) mit geeigneten Fehlermeldungen.
- Schreibe ein Programm, das einen oder mehrere Dateinamen auf der Kommandozeile erwartet und die Anzahl der Zeilen in den Dateien ermittelt.

Ausgaben weiterverarbeiten

- Unix-Stil: Programme kombinieren:

```
$ ps -ef | grep mozilla | sort | less
```

- *pipe*: |
- Standardeingabe: `STDIN`

```
while ($line = <STDIN>) { ... }
```

- kein `open` oder `close`

Aufgaben

- Was macht das folgende Programm `reader.pl`

```
while ($line = <>) {
  $i++;
  print "$i: $line";
}
```

wenn man es folgendermaßen aufruft:

```
reader.pl /etc/motd /etc/resolv.conf
ls -l | reader.pl
ls -l | reader.pl /etc/motd /etc/resolv.conf
ls -l | reader.pl /etc/motd -
```

Interaktion

- STDIN für Benutzereingaben:

```
print "please answer the following questions:\n";
print "name:  ";  $name = <STDIN>;   chomp($name);
print "age:   ";  $age = <STDIN>;    chomp($age);
print "email: ";  $email = <STDIN>;  chomp($email);
```

- chomp schneidet LF/CR-LF ab

Verzeichnisse lesen

- Verzeichnis ist auch nur Datei
- spezielle Befehle zur Interaktion:

```
opendir(DIR, "/") || die "can't open root directory\n";
while ($file = readdir(DIR)) {
  ...
}
closedir(DIR);
```

- oder auch wieder

```
@dirs = readdir(DIR);
```

- `readdir` liefert Dateinamen, nicht filehandle

Aufgaben

- Ändere das Programm mit den Dateitestoperatoren so, daß alle Dateien eines Verzeichnisses, das auf der Kommandozeile übergeben wird, auf ihre Eigenschaften untersucht werden.

Dateien schreiben

- filehandle zum Schreiben öffnen:

```
open(RESULT, '> results.txt') || die "can't open ...";
```

- > Ausgabeumlenkung in Shells
- filehandle mit `close` schließen
- Ausgabe in Datei:

```
print RESULT "$i zum Quadrat ist $i**2\n";
```

- eventuell vorhandene Datei wird überschrieben
- existierende Datei erweitern:

```
open(RESULT, '>> results.txt');
```

Aufgaben

- Erweitere das Programm zum Zählen der Zeilen so, daß das Ergebnis in eine Datei „`linecount.out`“ geschrieben wird.
- Wann wird der Inhalt einer bereits vorhandenen Datei gelöscht? Wenn die Funktion `open` ausgeführt wird, oder wenn Daten mit `print` geschrieben werden?

Aufgaben

- Was macht das folgende Programm?

```
foreach $i (1..1000) {
  print "$i  ";
  print "\n" unless $i % 5;
  sleep(1);
}
```

- Füge folgende Zeile am Anfang ein:

```
$| = 1;
```

buffering

- `$|` beeinflußt aktuelles Ausgabe-filehandle
- `select` wählt Ausgabe-filehandle
- ungepufferte Ausgabe in Dateien:

```
open(OUT, "> numbers.txt") || die "can't open output\n";
select(OUT);
$| = 1;
foreach $i (1..1000) { ... }
close(OUT);
```

- Programm starten, dann

```
$ tail -f numbers.txt
```

- Was passiert bei gepufferter Ausgabe?

filehandles in/aus pipes

- seitenweise Ausgabe:

```
open(PAGER, '| more') || die "can't open pipe\n";
foreach $i (1..400) {
    print PAGER "$i^2 = $i**2\n";
}
close(PAGER);
```

- Ausgabe eines Befehls lesen:

```
open(WHO, 'who |') || die "can't open pipe\n";
while ($line = <WHO>) { ... }
close(WHO);
```

backtick evaluation

- noch ein Shell-Erbe:

```
@files = ‘ls‘;

$machine = ‘uname -a‘;
```

- Vorsicht vor unportablen Skripten

Aufgaben

- Der Befehl

  ```
  ps -ef
  ```

  zeigt alle laufenden Prozesse an. Filtere Deine eigenen Prozesse aus der Liste und gib das Ergebnis als PostScript-Datei aus. Verwende dazu das Programm `a2ps`. Das Ergebnis kannst Du mit Ghostview (`gv`) überprüfen.