Unix & Shell-Programmierung SS21

Vorlesungswoche 6

Helga Karafiat

FH Wedel

Skripte generieren

- Erinnerung Shell: Keine Trennung von Daten und Skript
- Idee: Mehrere Dateien zu einem selbst entpackenden Archiv / Skript zusammenpacken
  - Daten müssen in die gepackte Datei geschrieben werden
  - Code für das entpackende Skript muss innerhalb des packenden Skriptes in die gepackte Datei geschrieben werden
  - Ansatz: Kombination aus Ausgaben und Inline-Dokumenten
  - Entstehende gepackte Datei soll die als Inline Dokumente gespeicherten Dateien wieder ausgeben

```
cat > hallo.txt <<ENDMARKE
Hier ist der Inhalt von hallo.txt
ENDMARKE
```

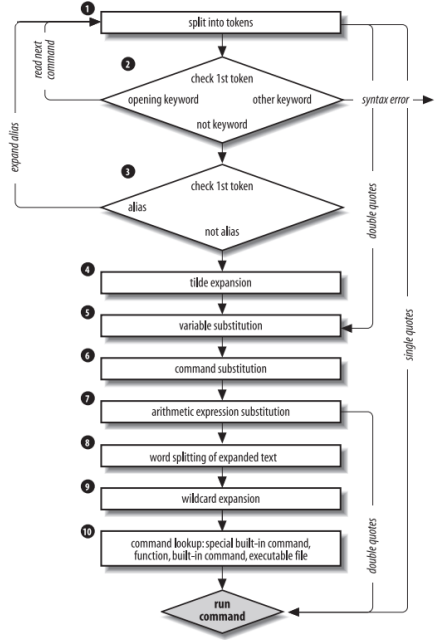
```
#!/bin/sh
# bundle.sh: Dateien als Skript zwecks Versand zusammenpacken
# Parameter: Dateinamen der zu packenden Dateien
# Ausgabe:   Ausgabedatei auf stdout

echo '#!/bin/sh'
echo '# zum Auspacken, sh auf diese Datei anwenden'

for FILE in "$@"
do
   echo "echo \"Entpacke: $FILE\" 1>&2"
   echo "cat > \"$FILE\" <<'Ende von $FILE'"
   cat "$FILE"
   echo "Ende von $FILE"
done
```

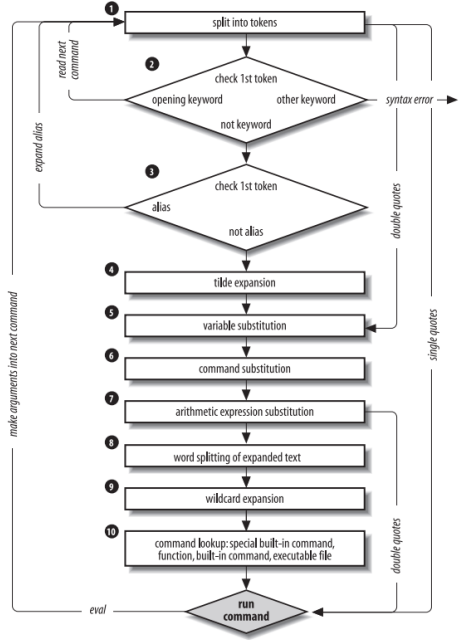
Code dynamisch erzeugen

- Weitere Erinnerung Shell: Variablen sind Zeichenketten, Befehle sind auch Zeichenketten
- Idee: Kann man denn auch Code dynamisch erzeugen?
  z.B. um die Parameter eines verwendeten Tools dynamisch je nach Eingabe zu bestimmen
- Erster Ansatz:
  - z.B. `ECHO='echo "Hallo    Welt"' ; $ECHO`
  - oder `GREP='cat *.* | grep TODO' ; $GREP`
- Zweiter Ansatz:
  - z.B. `ECHO='`echo "Hallo   Welt"`' ; $ECHO`
  - ...

(Back) to the roots: Wie interpretiert die Shell Eingaben?

- Eingaben werden zeilenweise aus Skript oder von Kommandozeile gelesen, dabei Unterscheidung zwischen Hier-Dokumenten und Befehlsketten
- Hier-Dokumente: Verarbeitung gemäß der Regeln für Hier-Dokumente
- Befehlskette: Aufteilung in Token (Token Recognition)
  - Grober Ablauf
    - Eingabe in einfache Kommandos und zusammengesetzte Kommandos aufteilen
    - Ersetzungen auf den verschiedenen Teilen der Kommandos ausführen (z.B. Variablen- und Kommandoersetzung, Parameterliste generieren)
    - Umleitungen der entsprechenden Dateiströme ausführen
    - Funktionen, built-in Befehle, ausführbare Dateien oder Skripte mit den entsprechenden Parametern ausführen
    - Auf den Abschluß jedes Kommandos warten und dessen Exit-Status einsammeln
  - Verarbeitung durch zeichenweises Parsing
    - berücksichtigt Quoting, entsprechend der Regeln für die jeweiligen Quotes (\, weak, strong)
    - erkennt Subsitutionen für Parameter, Befehle und Arithemtik und setzt sie als Token um
    - verwirft während der Verarbeitung Kommentare


1
split into tokens
2
check 1st token
opening keyword
other keyword
not keyword
read next command
syntax error
3
check 1st token
alias
not alias
expand alias
double quotes
4
tilde expansion
5
variable substitution
6
command substitution
7
arithmetic expression substitution
8
word splitting of expanded text
9
wildcard expansion
10
command lookup: special built-in command, function, built-in command, executable file
double quotes
single quotes
run command


1
split into tokens
read next command
2
check 1st token
opening keyword
other keyword
syntax error
not keyword
expand alias
3
check 1st token
alias
not alias
double quotes
4
tilde expansion
5
variable substitution
6
command substitution
7
arithmetic expression substitution
8
word splitting of expanded text
9
wildcard expansion
10
command lookup: special built-in command, function, built-in command, executable file
double quotes
single quotes
make arguments into next command
run command
eval

Auswertung von Zeichenketten als Befehl mit `eval`

- `eval` führt dazu, dass für die angegebene Zeichenkette die komplette Verarbeitung (Token Recognition, Ersetzungen etc.) erneut ausgeführt wird
- kann verwendet werden um dynamisch erzeugten Code ausführbar zu machen
  - z.B. `ECHO='echo "Hallo    Welt"' ; eval "$ECHO"`
  - z.B. `GREP='cat *.* | grep TODO' ; eval "$GREP"`
- Konzept von `eval` ist den meisten Skriptsprachen bekannt
- `echo 'eval' | tr 'a' 'i'`
  - "Code Intrusion" möglich
  - `eval` am Besten nie direkt zur Verarbeitung von Benutzereingaben verwenden
  - Wenn `eval` direkt für Benutzereingaben verwendet wird, dann diese zumindest nach `$` und `` ` `` durchsuchen

Nützliche Tools: Finden (und automatisches Verarbeiten) von Dateien

- Programm `find`
  - (rekursives) Auffinden von Dateien anhand von Namensmustern und/oder Attributen
  - zusätzlich Möglichkeit zum automatisierten Verarbeiten der gefundenen Dateien
- Syntax: `find [OPTIONS] [STARTING-POINT...] [EXPRESSION]`
  - Optionen: Umgang mit Links, Debugoptionen und Optimierungen
  - Startpunkt der Suche
  - Ausdrücke (siehe `man find`):
    - ★ Tests / Vergleiche
    - ★ Operatoren
      (z.B. um Ausdrücke zu verbinden: `-a`/ `-o`, klammern:`()`, negieren:`!`)
    - ★ Optionen (global und ausdruckspezifisch)
    - ★ Aktionen (gefundene Dateien direkt verarbeiten - Seiteneffekte)

- Typische Suchkriterien
  - Größe (kleiner (−), größer (+), gleich)
  - Datumsangaben (z.B. vor 3 Tagen verändert)
  - Namesmuster (Wildcards, reguläre Ausdrücke möglich)
  - Besitzerangabe
  - Gesetzte Rechte
  - ...
- Vorteile:
  - Vielzahl an Suchkriterien (alles was denkbar ist)
  - nahezu beliebig konfigurierbar
- Vorsicht:
  - ggfs sehr langsam durch (mehrfachen) rekursiven Abstieg
  - Ausgabe ist unsortiert und in zufälliger Reihenfolge
  - `find` kennt keine versteckten Dateien (werden immer mit gefunden)
  - Ausführen von Aktionen hat relativ hohes Gefahrpotential

- Beispiele
  - `find`
    Findet alle Dateien im aktuellen Verzeichnis und Unterverzeichnissen
  - `find . -size 0`
    wie oben, aber nur leere Dateien (Angabe `.` für aktuelles Verzeichnis ist optional)
  - `find . -size +0 -a -atime -10`
    wie oben, aber nur nicht-leere Dateien auf die in den letzten 10 Tagen zugegriffen wurde
  - `find ~ \( -name [0-3] -true -o -mtime +10 \) -a -type d`
    alle Verzeichnisse im Home des aktuellen Users und dessen Unterverzeichnissen deren Name eine Ziffer von zwischen 0 und 3 ist oder die nicht in den letzten 10 Tagen verändert wurden
  - `find ~ -type f -a ! -perm -u=x`
    wie in vorherigem Beispiel, aber diesmal Dateien, die der aktuelle User nicht ausführen darf (`-perm u=x` würde nach Dateien mit exakt den Rechten `--x------` suchen)

- Beispiele mit Seiteneffekten
  - `find -name '*~' -delete`
    Löscht alle auf ~ endenden Dateien im aktuellen Verzeichnis und dessen Unterverzeichnissen
  - `find -name '*~' -exec rm -v {} \;`
    wie oben allerdings mit dem Befehl `rm` und "gesprächiger" Ausgabe
    - `{}`: Jeweiliges Ergebnis von `find` wird `rm` als Parameter übergeben
    - `;` terminiert den Befehl
  - `find ~ -name "*.java" -exec grep -l TODO {} \;`
    findet alle java-Dateien im Home des aktuellen Users und dessen Unterverzeichnissen und gibt die Namen (inkl. Pfade) der Dateien aus, die ein TODO beinhalten

Nützliche Tools: Packen von Dateien (`tar`)

- Erlaubt das Zusammenpacken von Dateien und ganzen Verzeichnisbäumen
- Hintergrund: Das Kopieren einer einzelnen zusammenhängenden Datei geht deutlich schneller als das Kopieren von vielen kleinen Dateien
- Entstanden zu Zeiten von Archivierung auf Bandlaufwerken, daher der Name `tape archiver`
- Vorsicht: Wenn keine explizite Datei angegeben ist, greift `tar` zum Lesen oder Schreiben immer auf das Tape-Device zu, egal ob es eins gibt, oder nicht
- Ein `tar`-Aufruf setzt sich zusammen aus genau einer Operation mit beliebig vielen Optionen (und ggfs. den zu packenden Dateien)

- Häufige `tar`-Operationen
  - `-t` Anzeige des Inhaltes
  - `-x` Entpacken des Archivs
  - `-c` Neues Archiv erzeugen
  - `-u` Aktualisieren des Archivs (alte durch neuere Dateien ersetzen)
  - `-r` Anhängen von Dateien an das Achiv
  - zudem weitere Möglichkeiten um einzelne Dateien in Archiven zu bearbeiten: Hinzufügen, Ersetzen, einzelne Dateien extrahieren (siehe `man tar`)
- Häufige `tar`-Optionen
  - `-f DATEINAME` schreibt bzw liest das Archiv mit DATEINAME, wenn als Name - angegeben ist, wird stdin bzw stdout verwendet
  - `-v` Ausgaben während der Verarbeitung machen (z.B. Dateiennamen) (da tar nicht sonderlich gesprächig ist, häufig verwendete Option)
  - `-p` beibehalten aller Zugriffsrechte und Besitzverhältnisse, auch wenn unbekannt (z.B. bei einer Rescue-CD hilfreich)

- Vorsicht: `tar` ist nicht in POSIX standardisiert und erlaubt je nach System diverse Varianten / Schreibweisen um Parameter anzugeben
- Beispiel: Packe alle `.txt`-Dateien in das Archiv mit dem Namen `foo.tar`
  - Klassisch: `tar cvf foo.tar *.txt`
  - Unix-Style: `tar -c -f foo.tar -v *.txt`
  - GNU-Style: `tar --create --file foo.tar --verbose *.txt`
- Beispiel: Entpacke alle Dateien aus dem Archiv `foo.tar`
  - Klassisch: `tar xvf foo.tar`
  - Unix-Style: `tar -x -f foo.tar -v`
  - GNU-Style: `tar --extract  --verbose --file foo.tar`

- Seit 2001: POSIX standardisiertes Packer-Tool `pax` (portable archive exchange)
  - Dateiformate: `cpio`, `tar` und `pax`
  - ähnlicher Funktionsumfang wie `tar`, allerdings moderner
    (versteht sich als Nachfolger von `cpio` und `tar`)
  - eigenes Format `pax` (`ustar`-Format mit erweitertem Header)
    z.B. zusätzliche charsets, Kommentare und erweiterte Dateiinformationen,
    die `ustar` nicht abbilden kann
  - kann als Filter verwendet werden
  - (noch) nicht weit verbreitet

Komprimieren von Dateien / Archiven

- Zusätzlich zum Packen der Daten ist häufig eine (verlustfreie) Kompression sinnvoll
- Standard Kompressionstools
  - unter Unix (standardisiert in POSIX)
    - ★ Kompression mit `compress`, Dekompression mit `uncompress` (Dateiendung `.Z`)
  - unter Linux (GNU):
    - ★ Kompression mit `gzip`, Dekompression mit `gzip -d` oder `gunzip` (Dateiendung `.gz`)
    - ★ Kompression mit `bzip2`, Dekompression mit `bzip2 -d` oder `bunzip2` (Dateiendung `.bz2`)
- `tar` bietet in den `GNU`-Versionen, die Möglichkeit über Optionen direkt zu komprimieren
  - `--gzip` bzw. `-z`
    Nach dem Archivieren bzw. vor dem Auspacken des Archivs wird `gzip` angewendet (Häufige Dateiendung `.tgz`), z.B. `tar cvzf foo.tgz *`
  - `--bzip2` bzw. `-j`
    wie `-z` nur, dass `bzip2` verwendet wird (Häufige Dateiendung `.tbz` oder `tbz2`)
    z.B. `tar xjvf foo.tbz`

Weitere nützliche Tools: `diff` und `patch`

- Aufrufsyntax: `diff [OPTION]... DATEI1 DATEI2`
- Bietet die Möglichkeit zwei Dateien zeilenweise miteinander zu vergleichen
- gedacht um Änderungen an Dateien nachvollziehen zu können, daher beim Aufruf sinnvollerweise mit `DATEI1` als ältere Datei
- wenn die Dateien gleich sind, keine Ausgabe auf `stdout`
- ansonsten Ausgabe der Differenzen blockweise nach folgendem Format
  - Präfix (`ZahlÄnderungZahl`)
    - Zahl: Angabe der Zeilennummern (links: erste Datei, rechts: zweite Datei)
    - Änderung (`c`: change, `a`: add, `d`: delete)
  - Betroffene Inhalte:
    - < Inhalt der ersten Datei (Ursprung)
    - > Inhalt der zweiten Datei
    - `---` Trennung der Inhalte

- Aufruf:

```
echo "Test1" > test1
echo "Test2" > test2
diff test1 test2
```

- Ausgabe:

```
1c1
< Test1
---
> Test2
```

- Bedeutung der Ausgabe:
  Zeile 1 der ersten Datei wurde zu Zeile 1 der zweiten Datei (`1c1`)

- Verschiedene (teilweise besser lesbare) Ausgabevarianten möglich
  - `--context` (`-c`) oder `--unified` (`-u`)
    - Angaben zu Ursprungsdateien enthalten (Name, Datum)
    - Unveränderte Inhalte werden in Teilen mit ausgegeben (default maximal 3 Zeilen am Stück)
    - Änderungen nach Dateien getrennt aufgeliset
  - `--side-by-side` (`-y`)
    - Inhalte werden in zwei Spalten nebeneinander ausgegeben
    - Vorsicht: Gibt immer die kompletten Dateien aus (auch wenn gleich)
- Möglichkeit z.B. diverse Veränderungen in Whitespace und Groß- und Kleinschreibung in den Dateien zu ignorieren (siehe `man diff`)
- Komfortablere Ansicht von Änderungen für menschliche Benutzer, z.B. mit vimdiff
  - Navgierbare Ansicht der Dateien nebeneinander
  - farbliches Highlighting der Änderungen auch innerhalb der Zeilen

- Aufrufsynax: `patch [OPTION]... DATEI PATCH`
- Anwendung des Ergebnisses von `diff` (der `PATCH`) auf die Datei (`DATEI`)
- Kann Ausgabeformate von `diff` verarbeiten und erkennt sie meistens automatisch
- Ausgaben, die mit der Option `-c` oder `-u` von `diff` erstellt wurden, erlauben zudem:
  - Vergleich der Dateinamen
  - Erkennung von Änderungen im Kontext
    (nützlich, wenn `patch` auf bereits veränderte Datei angewendet wird)
- `patch` wendet so viele Änderungen an, wie es kann, Fehler werden gemeldet
- Vorsicht: per default überschreibt `patch` die angegebene Datei
  - Nützliche Option: `--dry-run`
    gibt die geänderte Datei nur auf stdout aus, nimmt aber keine Änderung an der Datei vor
  - ggfs. eine der Backup-Optionen verwenden, siehe `man patch`

- Nutzung von `diff` und `patch`
  - allgemein in der Softareentwicklung um Änderungen an (Source-)Dateien zu übermitteln
  - innerhalb von Versionsverwaltungssystemen für die Verwaltung der Versionsunterschiede (häufig auch mit Erweiterungen wie z.B. `diff3`)
- Beispiel:
  - Eingabe:

```
echo "Test1" > test1
echo "Test2" > test2
diff test1 test2 > patch_test
patch test1 patch_test
```

  - Ergebnis: `test1` hat den gleichen Inhalt wie `test2`
  - Überprüfung des Ergebnisses: `diff test1 test2` liefert keine Ausgabe mehr