Grundlagen der Informatik 2

Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm

Hochschule Harz

FB Automatisierung und Informatik

mwilhelm@hs-harz.de

Raum 2.202

Tel. 03943 / 659 338

Gliederung

1. Einführung

2a. WWW / HTML

Übersicht über WWW

Einführung in HTML

2b) Cascading Stylesheet CSS

3) Java Script

4. PHP

5. Unix

6. Unix Shellprogrammierung

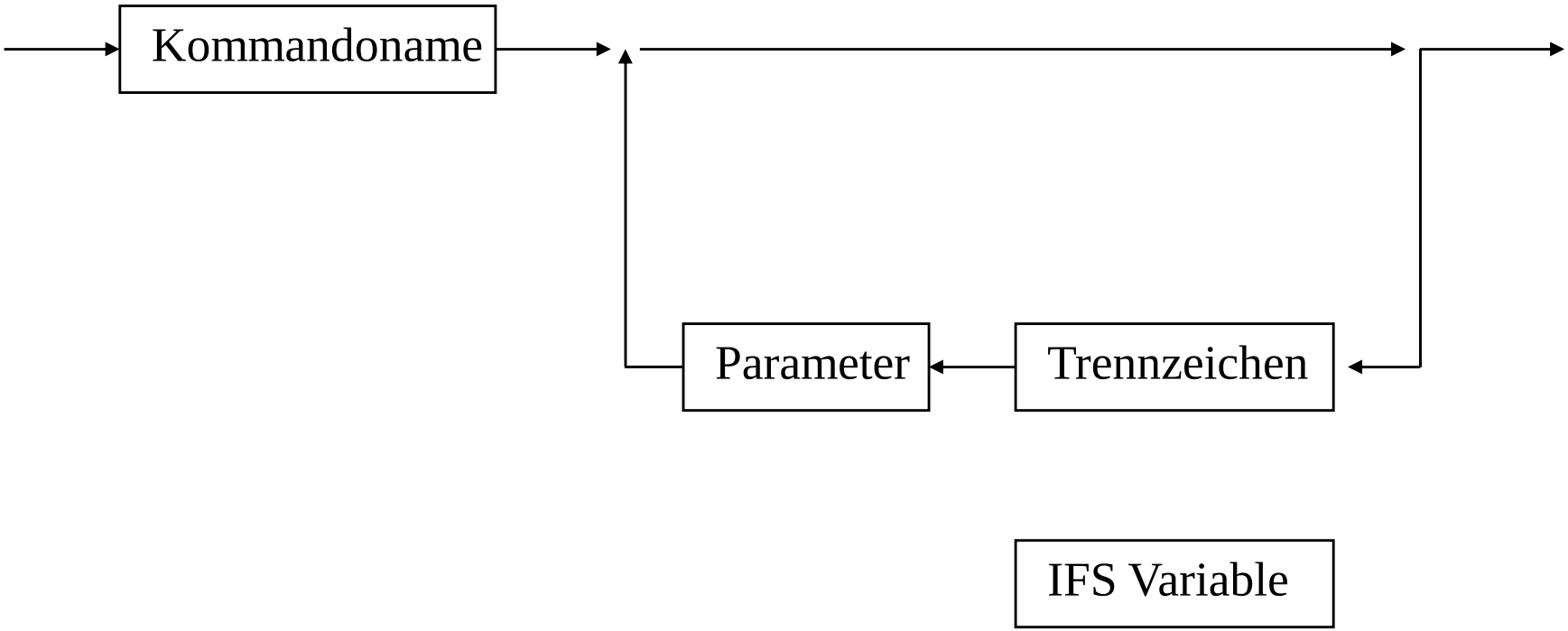
UNIX-Shellprogrammierung

- Kommandointerpretation
- vi-Editor
- Shell
  - (Eingabe, Variablen, Suchpfade, Sonderzeichen, Hintergrundprozesse)
- Umleitungen, Pipes
- Arbeiten mit Dateien
- Einfache Programmierung (If-Anweisungen,Schleifen)
- Arithmetik
- Funktionen

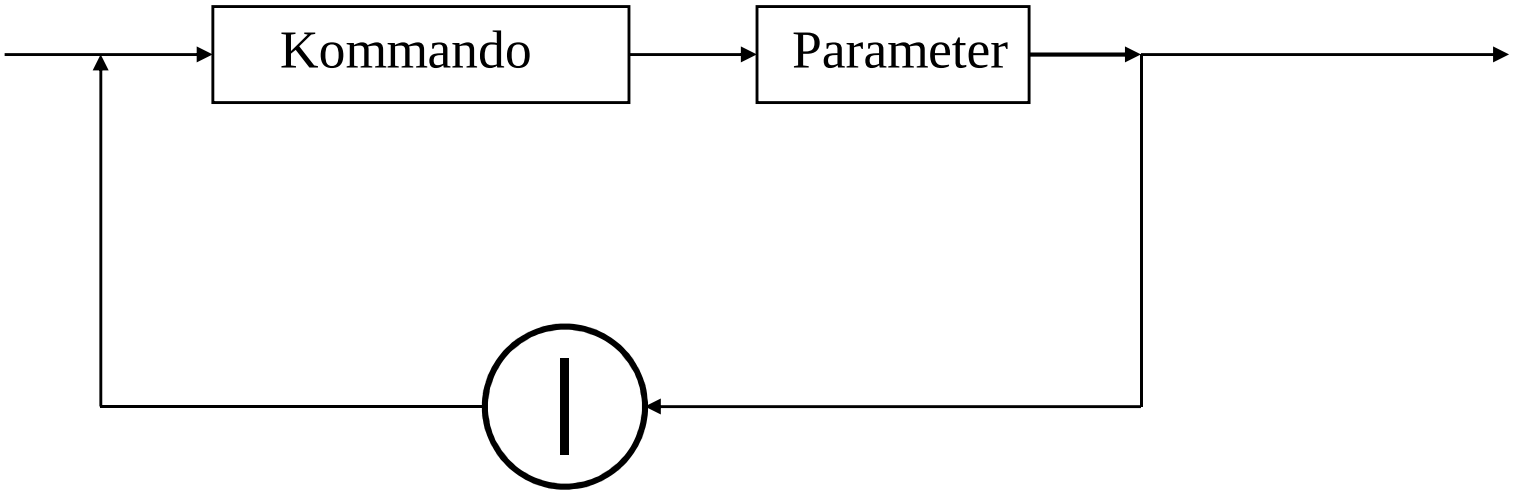
Unix Skriptsprache

- Eine Shell im Betriebssystem UNIX umfasst jeweils zwei Funktionen:
  - **Kommandointerpreter**
  - **Programmiersprache**
- In Abhängigkeit von ***kommandoname*** werden die ***argumente*** definiert. Dabei lässt sich allgemein angeben, dass es sich in der Regel bei dem ersten Argument um ***Optionen*** bzw. ***Schalter*** handelt, die im allgemeinen mit **-** beginnen. Danach können weitere ***Parameter*** folgen.

Syntaxdiagramm Kommando



Syntax für Pipe (Pipekonzept):



Variablen und Parameter

- Im Umgang mit Variablen der Shell sind einige Besonderheiten gegenüber anderen Programmiersprachen zu beachten. Es lassen sich grundlegend drei Formen unterscheiden:
  - **Variablendeklaration**
  - **Wertzuweisung**
  - **Wertreferenzierung (Zugriff auf den Wert einer Variablen)**
- Zur Verdeutlichung des Sachverhaltes sollen wenige Beispiele dienen:

Form	Beispiel	Bildausgabe
Deklaration	**$ paul=**	
Wertzuweisung	**$ paul=otto**	
Wertreferenzierung	**$ echo $paul**	**otto**

Beispiel:

```
Paul=
Paul=Otto
echo Paul
echo paul
echo Otto
echo otto
echo $Paul
```

Deklaration
Wertzuweisung
Ausgabe Paul
Ausgabe paul
Ausgabe Otto
Ausgabe otto
Ausgabe des Inhalts von Paul
Otto

```
my=Bash
echo I like $myshell
```

```
my=Bash
echo I like ${my}shell
```

Umgebungsvariablen einer Shell

Variable	Bedeutung	typischer Wert
HOME	Home-Directory des Benutzers	/home/otto
PATH	Anzahl von Directories zur Kommandosuche	/bin:/usr/bin:/etc
CDPATH	Anzahl von Directories für das Kommando cd	~/verz1:~/labor
PS1	1. Promptzeichen **$** oder **#**	"$\PWD> "
PS2	2. Promptzeichen **>**	"weiter> "
MAIL	Pfad zum elektronischen Briefkasten	/usr/spool/mail/otto
IFS	Separatorzeichen für Kommandozeile	TAB, NL, Leerzeichen
TZ	Zeitzone	MEZ-1MESZ,3.2.0,9.2.0
SHELL	Pfadname der aktiven Shell	/bin/bash
RANDOM	Anzeige einer Zufallszahl 0 bis 32767	

Spezielle Variablen

Variable	Bedeutung	Kommando
$-	gesetzte Shell-Optionen	set -xv
$$	PID (Prozessnummer) der Shell	kill -9 $$
$!	PID des letzten Hintergrundprozesses	kill -9 $!
$?	Exitstatus des letzten Kommandos	cat lalala ; echo $?

Positionsparameter als Variable

Positionsparameter	**Bedeutung**
$#	Anzahl der Argumente **ohne kommandoname**
$0	Name des Kommandos (***kommandoname***)
$1	1. Argument **nach dem Kommandoname**
:	:
$9	9. Argument **nach dem Kommandoname**
$@	alle Argumente **ohne Kommandoname**
$*	alle Argumente **ohne Kommandoname**

Shellskript shell_proz

```
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter übergeben"
echo "1. Parameter = $1"
echo "2. Parameter = $2"
echo "3. Parameter = $3"
echo "Alle Parameter zusammen:
$*"
echo "Meine Prozessnummer PID = $$"
```

Anzeige mit 1,2,3,4 und 5 Parametern ?!

Testlauf

- Das Shell-Skript ausführbar machen
  - **$ chmod u+x Shell_Proz**
- Das Shell-Skript erzeugt die folgenden Ausgaben auf dem Bildschirm:

$ Shell_Proz emil erna paul fred otto

Mein Name ist Shell_Proz

Mir wurden 5 Parameter übergeben

1. Parameter = emil

2. Parameter = erna

3. Parameter = paul

Alle Parameter zusammen:

emil erna paul fred otto

Meine Prozessnummer PID = 4711

$

Einlesen mit read:

Zum Einlesen in Variablen wird die Anweisung read benutzt:

```
read xx
```

Mit der Eingabe

```
1111
```

```
read xx yy zz
```

Mit der Eingabe

```
1111  2222   3333  4444
```

xx: ?

yy: ?

zz ?

-n	Kein Zeilenumbruch
-t	Ersetzen Sonderzeichen

read1 read2

Einlesen mit read:

Ausgabe eines Zusatztextes:

```
echo Eingabe
read xx

echo -n Eingabe                 // Kein Zeilenumbruch
read xx

echo -n "Eingabe: "             // Kein Zeilenumbruch
read xx

read –p  "Eingabe: " xx         // Kein Zeilenumbruch
```

read3

Kommandoersetzung:

Kombination aus Umleitung und Variablenersetzung:

```
liste=$(ls)                    // Inhalt des Verzeichnisses in liste

zeit=$(date)
z1=$(grep freundlich verkauf)
z2=$(grep freundlich verkauf | sort)
dateien=$(ls *.java)
```

-

Alphabetische Vergleiche:

Abfrage	wahr / falsch
if [[abc < abs]]	# wahr
if [[abc > abcd]]	# falsch
if [[123 < 147]]	# wahr
if [[123 < 0147]]	# falsch
if [[0815 > 3]]	# falsch
if [[12345 <77]]	# wahr
if [[püh < pöh]]	# falsch

Alphabetische Vergleiche:

Abfrage	wahr / falsch
[abc < abs]	# wahr
[abc > abcd]	# falsch
[123 < 147]	# wahr
[123 < 0147]	# falsch
[0815 > 3]	# falsch
[12345 <77]	# wahr
[püh < pöh]	# falsch

-

Arithmetische Vergleiche:

Abfrage	Bedeutung
$x -lt $y	x ist kleiner als y
$x -le $y	x ist kleiner oder gleich als y
$x -eq $y	x ist gleich als y
$x -ne $y	x ist ungleich als y
$x -gt $y	x ist größer als y
$x -ge $y	x ist größer oder gleich als y

Arithmetische Auswertung !

-

Kontrollstrukturen:

Bedingte und einfache Fallunterscheidungen

```
if bedingung
  then
    kommando_1
  else
   kommando_2
fi
```

```
if bedingung
then
     kommando_1a
     kommando_1b
     kommando_1c
  else
     kommando_2a
     kommando_2b
fi
```

bsp/if/ if1 if2

Arithmetische Vergleiche:

```
T1=23
T2=44
if [  $T1  -lt   $T2  ]
then
  echo true
else
  echo false
fi
```

Ausgabe: true

```
T1=23
T2=44
T3=45
if [  $T1  -gt   $T2  ]
then
  echo 1
elif [ $T3 –gt $t2 ]
   then
     echo T3
   else
    echo T2
  fi
fi
```

Ausgabe: T3

Beispiele Bash:

typeset [-afFirtx] [-p] name[=value]
declare [-afFirtx] [-p] name[=value]

```
typeset -i k          # Integervariable k
typeset -i k m n      # Integervariable k, m, n

-a                    # Explizite Definition eines Arrays
-f                    # Use function name only
-r                    # Variablen sind readonly
-t                    # Erhält trace-Attribut (Debug)
-x                    # Export
-p                    # Anzeige aller Variablen und Namen
-F                    # Verhindert die Anzeige der Funktions-
                        definitionen
```

-

Dateieigenschaften prüfen

if [ausdruck datei]

ausdruck	Bedeutung
-a <datei>	datei existiert
-b <datei>	datei existiert und ist blockorientiertes Gerät
-c <datei>	datei existiert und ist zeichenorientiertes Gerät
-d <datei>	datei existiert und ist Verzeichnis
-e <datei>	datei existiert
-f <datei>	datei existiert und ist einfache Datei
-g <datei>	datei existiert und für Gruppe s-Bit gesetzt
-h <datei>	datei existiert und ist eine symbolische Linkdatei
-k <datei>	datei existiert und das sticky-Bit ist gesetzt
-p <datei>	datei existiert und ist ein pipe (FIFO)
-r <datei>	datei existiert und Leserecht

Dateieigenschaften prüfen

if [ausdruck datei]

ausdruck	Bedeutung
-r <datei>	datei existiert und Leserecht
-s <datei>	datei existiert und ist nicht leer
-u <datei>	datei existiert und für Eigentümer s-Bit gesetzt
-w <datei>	datei existiert und hat Schreibrecht
-x <datei>	datei existiert und hat Ausführungsrecht
-O <datei>	datei existiert und Eigentümer User
-G <datei>	datei existiert und Eigentümer Group
-L <datei>	datei existiert und ist symbolischer Link
-S <datei>	datei existiert und ist Socket
-N <datei>	datei existiert und wurde nach dem letzten Lesen geändert

Dateieigenschaften prüfen

if [ausdruck datei]

ausdruck	Bedeutung
-d <datei>	datei existiert und ist Verzeichnis
-f <datei>	datei existiert und ist einfache Datei
-h <datei>	datei existiert und ist eine symbolische Linkdatei

Dateieigenschaften prüfen

if [ausdruck datei]

ausdruck	Bedeutung
-r <datei>	datei existiert und Leserecht
-s <datei>	datei existiert und ist nicht leer
-u <datei>	datei existiert und für Eigentümer s-Bit gesetzt
-w <datei>	datei existiert und hat Schreibrecht
-x <datei>	datei existiert und hat Ausführungsrecht
-O <datei>	datei existiert und Eigentümer User
-G <datei>	datei existiert und Eigentümer Group
-L <datei>	datei existiert und ist symbolischer Link
-S <datei>	datei existiert und ist Socket
-N <datei>	datei existiert und wurde nach dem letzten Lesen geändert

Dateiabfragen:

datei1	-nt	datei2	datei1 ist neuer als datei2
datei1	-ot	datei2	datei1 ist älter als datei2
datei1	-ef	datei2	datei1, datei2 Links auf dieselbe Datei

-z string	string ist leer (zero)
-n string	string ist nicht leer

Beispiel:

```
if [ -n "$sStr" ]
then
fi
```

Bedingte und einfache Fallunterscheidungen

Beispiele:

```
If  [  -d $1  ]
then
   echo $1 ist ein Verzeichnis
else
  echo $1 ist kein Verzeichnis
fi
```

if1 if2

Aufgabe:

- **Erstellen Sie ein Script, welches den ersten Parameter dazu verwendet, in ein Verzeichnis zu wechseln.**
- **Existiert das Verzeichnis nicht, so soll dieses mit einer eventuellen Fehlermeldung angelegt werden.**

Lösung ohne Fehlerumleitung:

```
cd $1  || mkdir $1 && cd $1 ||  echo fehler
pwd
```

Lösung mit Fehlerumleitung:

```
cd $1 2>error || mkdir $1 2>error && cd $1 2>error ||  echo fehler
pwd
```

Eingabe mit Abfrage:

```
echo -n "Datei $1 löschen? (ja/nein/abbruch): "
read antwort
[[ $antwort = a* ]] && exit 1
[[ -z $antwort  ||  $antwort = j* ]] && rm $1
```

```
wert=815
[[ $wert = 4711  ||  $wert = 0815 ]]  &&  echo wahr
```

oder ???

```
[[ $wert = 4711  ||  $wert = 0815 ]]  &&  echo falsch
```

Kornshell und Bash falsch

Arithmetik:

Anweisungen werden ohne Leerzeichen eingeben:

```
typeset -i a b sum
a=1
b=2
sum=a+b
echo $sum
```

```
a=2
b=3
sum=`expr $a + $b`
echo $sum
```

Arithmetik:

Anweisungen mit Dollar-Klammer:

```
a=1
b=2
sum=$[ 1 + ( a + b ) * ( a - b )  ]
echo $sum
```

Anweisungen mit Dollar-Klammer:

```
a=1
b=2
sum=$(( 1 + ( a + b ) * ( a - b )  ))
echo $sum
```

Arithmetik:

Anweisungen mit der Klammer (( )) :

```
a=1
b=2
(( sum= 1 + ( a + b ) * ( a - b )   ))
echo $sum
```

Fehlerhafte Eingabe:

```
a  =  1
b  =  2
(( sum= 1 + ( a + b ) * ( a - b )   ))
echo $sum
```

Arithmetik:

Operatoren:

+	Addition
–	Substraktion
*	Multiplikation
/	Ganzzahlige Division
%	Modulo
+=	Addition
–=	Substraktion
*=	Multiplikation
/=	Ganzzahlige Division
%=	Modulo

Zählschleife:

- Es wird der Zählschleife eine Menge von Werten übergeben. Bei jedem Durchlauf wird einer Variablen der nächste Wert aus der Menge von Werten übergeben.
- Die Anzahl von Durchläufen ist an die Menge von übergebenen Werten gebunden
- Break und Continue ändern die Abarbeitung

```
for name in wert_1 wert_2 wert_3 wert_4
do
    kommandoliste
done
```

Beispiele einer Zählschleife:

Wenn der Positionsparameter $* bzw. alle Parameter verwendet werden soll, wird die Zählschleife wie folgt notiert:

```
for name in $*
do
        kommandoliste
done
```

```
for (( i=1; i<10; i++ ))
do
  echo $i
done
```

Feste Schleifenanzahl:

```
for i in 1 2 3 4 5 6
do
   echo "Hallo"
done
```

```
for i in {1,4,5,6,7,11}
for i in {1..5}
do
  echo $i
done
```

Beispiele einer Zählschleife:

Dateien Schleife:

```
liste=$(ls *.c)
for datei in $liste
do
   echo $datei
done
```

Feste Schleifenanzahl:

```
for i in 1 2 3 4 5 6
do
   if [  $i   -eq  4 ]
   then
      break
   fi
   echo "Hallo"
done
```

Break und Continue ändern die Abarbeitung

Whileschleife:

- Schleife mit Eingangsbedingung:

```
while  [  Bed  ]
do
     kommandoliste
done
```

- Break und Continue ändern die Abarbeitung

Beispiele einer Whileschleife:

```
i=1
while [ $i -lt 10 ]
do
  echo $i
done
```

```
i=1
while [ $i -lt 10 ]
do
  echo $i
  i=$i+1
done
```

```
i=1
while [ $i -lt 10 ]
do
  echo $i
  i=`expr $i \+ 1 `
done
```

```
i=1
while [ $i -lt 10 ]
do
  echo $i
  i=i+1
done
```

Case / Switch Anweisung:

- Statt verschachtelter if-Anweisungen erlaubt die Case-Anweisung eine übersichtlichere Programmierung:

```
case Bed in
(muster1)   kommandoliste
                ;;
(muster2)   kommandoliste
                ;;
esac
```

1. Beispiel einer Case Anweisung:

```
prompt = $*
echo -n "$prompt (ja/nein): "
read antwort

case $antwort in
 ([jJ])   echo ja
          exit 0
          ;;
 ([nN]) echo nein
          exit 1
          ;;
 (*)      echo rest
          exit 2
esac
```

Aufruf mit

bash case2 Wollen..

1. Beispiel einer Case Anweisung:

```
prompt = $*
echo -e "$prompt (ja/nein): "
read antwort

case $antwort in
 j|J)     echo ja
          exit 0
          ;;
 n|N)     echo nein
          exit 1
          ;;
 *)       echo rest
          exit 2
esac
```

Aufruf mit

bash case2 Wollen..

2. Beispiel einer Case Anweisung:

Aufgabe:

Einen Script soll ein einfacher Name übergeben werden.

```
name=$1
case $name in
 (*/*)   echo  "$name ist kein einfacher Name"
         exit 1;;
 (' ' )  echo  "$name ist kein einfacher Name"
         exit 1;;
esac
```

3. Beispiel einer Case Anweisung:

Einen Script soll ein einfacher Name übergeben werden, optional als 1. Parameter ein Schalter -n

```
s1=-n
case $1 in  # 1. parameter
 ($s1)  schalter=S1      name=$2
        ;;
 (* )   name=$1
esac

case $name in
 (*/*)  print -u2 "$name ist kein einfacher Name"
        exit 1;;
 (' ' )  print -u2 "$name ist kein einfacher Name"
        exit 1;;
esac
```

4. Beispiel einer Case Anweisung:

```
Bestimmung der Anzahl der Tage eines Monats
read monat
case $monat in
 (1|3|5|7|8|10|12)
        maxtage=31 ;;
 (* )
        maxtage=0;;
esac

echo “Anzahl der Tage ist $maxtage"
```

Funktionen:

Funktionen dienen dazu, Befehle unter einem Namen zusammenzufassen.

- Speicherung innerhalb eines Scriptes
- Schneller Aufruf
- Verwendung der Scriptparameter
- Verwendung der lokalen Variablen
- In der Shell definierte Funktionen können aufgerufen werden

Aufbau:

```
function name
{
    Anweisungen
}
```

Beispiele:

```
function list
{
   ls -aF
}

# mainprogram
ls > dat1
list
```

Beispiele:

```
function readname
{
   echo -n "Bitte einen Namen eingeben:"
   read name
}


# mainprogram
readname
```

Beispiel Funktion ohne Script:


```
$ function vii
> {
>   for p
>   do
>     echo $p
>   done
> }
```

Aufruf:

```
$ vii 11 22 33 44
11
22
33
44
```

```
typeset –i x temp

function readzahl
{
   echo -n "Bitte eine Zahl eingeben:"
   read temp
}

# mainprogram
readzahl
x=temp
x=x+1
echo $x
```

Beispiel: Einlesen einer positiven Zahl


```
function lese
{
   while true
   do
      read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  retcode=$i
}

lese
nr1=$retcode
lese
nr2=$retcode
echo  "nr1: $nr1   nr2: $nr2"
```

Beispiel: Einlesen einer positiven Zahl


```
function lese
{
   while true
   do
      read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  echo $i
}

nr1=lese                                  // fehler
echo  "nr1: $nr1"
```

Beispiel: Einlesen einer positiven Zahl


```
function lese
{
   while true
   do
     echo  -n  Bitte eine Zahl eingeben:
      read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  echo $i
}
nr1=`lese`                                  // korrekt
echo  "nr1: $nr1"
```

Eingabe: 123

Ergebnis: Bitte eine Zahl eingeben: 123

Beispiel: Einlesen einer positiven Zahl


```
function lese
{
   while true
   do
      read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  echo $i
}
echo  -n  Bitte eine Zahl eingeben
nr1=`lese`                                  // korrekt
echo  "nr1: $nr1"
```

Das Suchprogramm grep

- **grep** ist ein Unix-Programm zur Suche und Filterung vorgegebener Zeichenketten aus Dateien dient. Man definiert, wie die Zeile aussehen soll. Es wurde ursprünglich von Ken Thompson entwickelt
- Grep steht steht für global/regular expression/print. Es beinhaltet also reguläre Ausdrücke, nach denen man in dateien suchen kann
- Es existieren zahlreiche Varianten wie egrep, fgrep
- Agrep ermöglicht eine unscharfe Suche nach Textstrings (Fuzzy)
- **Aufruf:**
- grep [optionen] Suchstring [Datei(-liste)]

- **Definierte Bereiche:**

[:alpha:]	[:space:]	tab, space, CR,LF
[:lower:]	[:xdigit:]	hexadezimale Zahlen
[:upper:]		
[:alphanum:]		
[:digit:]		

Ausdrücke von Grep

.	beliebiges Zeichen (Punkt)
[ABCacZ]	Mengengruppe
[A-Z]	Mengengruppe
[^ABC]	negierte Gruppe
^	Zeilenanfang
$	Zeilenende
\<	steht für Wortanfang, links oder rechts steht ein space, tab. CR, Anfang
\>	steht für Wortende
\b	steht Wortanfang oder Wortende
( )	Gruppe mit Alternative Oder

Wiederholungs-Operatoren: (nur mit Parameter –E) oder egrep

*	0,1,n
?	0,1
+	1,n
{n}	das vorangegangene Zeichen tritt n-mal auf
{n,}	das vorangegangene Zeichen tritt n-mal oder öfter auf
{n,m}	das vorangegangene Zeichen tritt mindestens n-mal und maximal m-mal auf

a1:
abcd
Abcd
abccd
abcccd
abcabc
abcabcabc

a2:
editor
$y=$x

a3:
abeditor
$x=3
$x =42
((abcd = 3 + a))
((\tabcd\t= 3 + a))

a4:
12bfe.abcd
^12bFe
15bFe

a5:
13f abcd 1214
y=2*(5*(x+z))
12345$

a6:
havefunwertvollwer
hallo
(hallo)a
(((())))

a7:
12345
56789
abcdef
234.56

a8:
12.11.10
1.1.1
01.01.2010
11.11.abc
0y.
127.123.168.02
127.2.168.002

a9:
^hallowelt
hier ein wer-wolf
hier noch ein werwerwolf
hier ein weiterer wolf
0a.alpha
1z.beta

Parameter:

-i ignore case

-c Anzahl, count

-n line number

-w Ausdruck als Wort

-v Negation

-s Unterdrückt Fehlermeldungen

-d Directory Optionen

-d recurse oder –r

-d skip

-d read

d

Beispiele

grep **'ab'** ***** grep -E 'abc+'

'^ab' grep -E 'abc?'

'^abc' grep -E '^(ab|ed)'

'abcd$'

'cd$' egrep '[[:alnum:]]'

'1[23]' egrep '[[:alphanum:]]'

'1[23]f' egrep '[[:alpha:]]'

'1[23][bf]' egrep '[[:digit:]]'

'^1[23]' grep '([[^()]]*)a'

'^[ae]'

-i '^[Ae]' egrep '([[:digit:]]{1,3}\.){3}[[:digit:]]{1,3}'

'^[^ae], a1

'b.e'

'\.' '. ' liefert alles

-E Extended Regular expression oder egrep

Beispiele

egrep '$x'

egrep '\$x' korrekt, aber nun mit Space

egrep ' \$x ' als Wort, tab

egrep '\<$x\>' als Wort, tab, space, auch am Anfang und am Ende

Datum suchen:

egrep '\<[0-9]{1,}\.[0-9]{1,}\.[0-9]{1,4}\>' *

a8:12.11.10

a8:1.1.1

a8:01.01.2010

egrep '\<[0-9]{1,}\.[0-9]{1,}\.[0-9]{2,4}\>' *

a8:12.11.10

a8:01.01.2010

Beispiel zur Berechnung der Anzahl der Treffer

$ grep -c "a" a?

a1:3

a2:0

a3:3

a4:1

a5:1

a6:3

a7:0

a8:1

a9:1

Das Programm find: Suchen rekursiv

Das Kommando „find“ sucht alle Dateinamen, die bestimmten Bedingungen genügen.

Aufruf: find [directory] bedingung aktion

Parameter:

type	-f	Datei
	-d	Verzeichnis
directory	Beginn der Suche im Teilbaum	
bedingung	Es können folgende Bedingungen eingestellt werden:	
-name muster	Ist erfüllt, wenn der Name dem Muster entspricht	
-atime name	Ist erfüllt, wenn auf die Datei vor zahl Tagen zugegriffen wurde	
-mtime zahl	Ist erfüllt, wenn die Datei vor zahl Tagen verändert wurde.	
-newer datei	Ist erfüllt, wenn die untersuchte Datei nach der letzten Änderung von datei geändert wurde.	
-size zahl	Ist erfüllt, wenn die Datei zahl viele Blöcke hat.	

Das Programm find: Suchen rekursiv

Das Kommando „find“ sucht alle Dateinamen, die bestimmten Bedingungen genügen.

Aufruf: find [directory] bedingung aktion

aktion Bestimmt die Aktion

-exec kommando;

Auf jede gefundene Datei wird kommando ausgeführt. An derjenigen Stelle, des Kommandos, an der Dateiname steht, müssen die Klammern {} angegeben werden.

-ok kommando;

Auf jede gefundene Datei wird kommando ausgeführt. An derjenigen Stelle, des Kommandos, an der Dateiname steht, müssen die Klammern {} angegeben werden. Mit Abfrage des Benutzer (y oder yes)

-print

Ausgabe jeder gefundenen Datei.

Beispiele:

find /home/paul type –f *.html

Sucht alle Dateien mit der Endung html

find /home/otto ! [-user otto] -exec ls –l {} \;

Bewirkt die Anzeige aller Dateien, die nicht zu Otto gehören.

find /home/otto -atime +10 -print

Bewirkt die Anzeige aller Dateien , auf die in den letzten 10 Tagen nicht zugegriffen wurde.

find . [-size +10 -o mtime +7] -ls -l {} \;

Bewirkt die Anzeige aller Dateien, die eine Größe von mehr als 10 Blöcken haben oder in den letzten 7 Tagen nicht verändert wurden.

Beispiele:

find -name a.out

- Sucht alle Dateien mit dem Namen a.out, Programmieren mit g++

- find -name a.out -o –name "*.o"
- Sucht alle Dateien und Vewrzeichnisse mit dem Namen a.out oder mit *.o

- find ./ –type f –name "a?"
- Sucht alle Dateien mit der Maske a?

- find -maxdepth 2 -type f -name "a? "

Sucht alle Dateien mit der Maske a?, Maximal aber ein Unterverzeichnis