Shell Programmierung für System Administratoren

Alle Informationen beziehen sich auf die zugrundeliegende
Softwareversion: Solaris 10
Release 6.2: 11.08.2012/bk

Hinweise, Tipps, Anregungen, Kritik, Verbesserungsvorschläge bitte an den
Autor: HelmutBirk@t-online.de Homepage: http://www.edv-birk.de

---

INHALT

Modul 1 *UNIX Shells und Shell Scripte*

- **Überblick**

1-1 Bourne- und Korn-Shell

Die Vorteile der Korn-Shell sind Ihre Erweiterungen, wie z.B.:
command-history, job control oder built-in arithmetic und ist
für den normalen User von der Verbreitung immer noch der
Standard.

Die Bourne-Shell verfügt nicht über diese Erweiterungen; ist aber aus
zwei Gründen die Standard-Shell des Superusers:
1.) Die Start- / Stopskripte sind in der Bourne-Shell Syntax (**/sbin/sh**) geschrieben
2.) Die Shell ist "aus einem Guss"; wird also nicht wie alle anderen Shells
mit Hilfe von "shared objects" unter **/usr** zur Laufzeit zusammengestellt.
(siehe die Ausgabe von: **# ls -l /sbin/sh /bin/ksh** bzw. **# file /sbin/sh /bin/ksh**)
und ist zudem schneller als eine "zusammengestellte" Shell.

Modulanfang

1-2 "print" Kommandos der Shells

	sh ksh bash	**ksh**	**sh ksh bash**
no newline	$ echo "string \c"	$ print -n "string" oder $ print "string \c"	$ printf "string"
1 newline	$ echo "string"	$ print "string"	$ printf "string \n"
2 newline	$ echo "string \n"	$ print "string \n"	$ printf "string \n\n"

Tipp: Das Kommando print ist nur der Korn-Shell bekannt.
Werden dagegen die Kommandos echo oder printf verwendet, können diese für alle drei Shells eingesetzt werden.

Modulanfang
Dokumentanfang

Modul 2 *Scripte schreiben und debuggen*

- **Überblick**

2-1 Starten eines Shellscripts

Starten eines Shellscripts scri			
Aufruf: **----------** **Bedingungen:**	**$ ksh < scri** oder besser **$ ksh scri**	$. scri # nur in Spezialfällen sinnvoll: # **$ cd** # **$ . ./.profile**	**$ scri** # üblicher Aufruf, jedoch # 1x nach der Erstellung nötig: # **$ chmod u+x scri**
r-recht bei scri nötig: **x-recht bei scri nötig:**	ja nein	ja nein	ja ja
Suchmechanismus für die Datei scri:	der für Datendateien	der für Kommandodateien	der für Kommandodateien
Art der scri verarbei-tenden Shell:	die als Kommando genannte (hier ksh)	die zur Zeit aktuelle Shell	die in der Variablen SHELL genannte (durch **#!/bin/ksh** in den ersten Bytes von scri kann eine ksh erzwungen werden)
scri läuft in einer subshell :	ja	nein - das scri-Environment bleibt nach scri-Ende weiter bestehen - exit beendet die aktuelle shell	ja

Modulanfang

2-2 Kommentare in einem Shellscript

Bei Ausführung eines Scriptes durch die Shell wird jede Zeile als Unix-Kommando interpretiert und ausgeführt.
Eine Ausnahme bildet das einleitende "#" Zeichen: Alle Zeichen rechts vom "#" werden als Kommentar interpretiert.
Die zweite Ausnahme ist das "#"-Zeichen in der ersten Zeile / als erstes Zeichen: Mit dem Konstrukt **#!/bin/ksh** wird als ausführende Shell des Scriptes hier eine ksh erzwungen.
Achtung: Wird versehentlich ein Leerzeichen vor **#!/bin/ksh** eingegeben, wird die Anweisung von der Shell auch hier nur als Kommentar gewertet !

Modulanfang

2-3 Debuggen eines Shellscriptes

Einschalten der "Debug Optionen"		
	Bourne- und Korn-Shell	**nur Korn-Shell**
zeigt die Anweisung **nach** Substituierung der Variablen und Metazeichen der Shell	set -x	set -o xtrace
zeigt die Anweisung **vor** Substituierung der Variablen und Metazeichen der Shell	set -v	set -o verbose
Deaktivieriert die Dateinamensgenerierung (Metazeichen der Shell werden nicht mehr substituiert)	set -f	set -o noglob

Hinweise:

- Schalter können mit dem "+" Zeichen wieder deaktiviert werden (**$ set +x**)
- Schalter können kombiniert werden (**$ set -xv** zur Anzeige sowohl vor als nach der Anweisung)
- Einsatz der Debug Optionen sind sowohl auf der Kommandoebene :

  $ set -x
  $ scri.ksh
  oder:
  $ ksh -x scri.ksh
  als auch innerhalb eines Scriptes möglich (dabei können auch nur Teilbereiche eines Scriptes geprüft werden)

Modulanfang
Dokumentanfang

Modul 3 *Shell Environment*

- **Überblick**

3-1 Initialisierungsdateien der Shells (Bourne-Shell-Klasse)

Shell	**systemweit**	**Benutzer, gelesen beim Login**	**Benutzer, nur gelesen bei Shellstart**	**Shell-Pfadname**
Bourne	/etc/profile	.profile	n / a	/bin/sh (für User) /sbin/sh (für root)
Korn	/etc/profile	.profile .kshrc ($ENV)	.kshrc ($ENV)	/bin/ksh
Bourne Again	/etc/profile	.bash_profile falls nicht ex.: .bash_login falls auch nicht ex.: .profile	.bashrc	/bin/bash

Abhängig vom Shell-Eintrag in der /etc/passwd durchläuft der User beim Login zunächst die systemweite
Initialisierungsdatei (/etc/profile) und anschliessend die benutzerbezogene (z.B. .profile).

Aliase und Funktionen (nur ksh)

	Alias	Funktion
Anzeige	$ alias [alias_name]	$ functions [function_name] oder $ typeset -f [function_name]
Erzeugen	$ alias wer='who | wc -l'	$ function ll { ls -l S* | more; } oder $ ll() { ls -l S* | more; }
Löschen	$ unalias alias_name	$ unset -f function_name

Hinweis: Die Übergabe von Argumenten mittels Stellungsparameter
(z.B.: S1, $2 oder $*) ist nur bei Funktionen möglich !

Modulanfang

3-2 Variablenführung in der Shell

	globale Variable	**lokale Variable**
Variable definieren	export VAR=wert	VAR=wert
Variable löschen	unset VAR	unset var
Variablen anzeigen	env, export (nur globale)	set (lokale **und** globale)
Werte von Variablen anzeigen	print $VAR echo $VAR	print $var echo $var

Hinweis: "Systemweit globale Variablen" werden durch die Initialisierungsdateien erzeugt (durch das Einloggen
des Benutzers)

Modulanfang

3-3 Sonderzeichenentwertung und Kommandosubstitution

\	entwertet **ein** nachfolgendes Zeichen von einer evtl. Sonderbedeutung, z.B.: **$ ls *** (keine Dateinamengenerierung; es wird nach der Datei namens * gesucht
' .....'	es werden **alle** Sonderzeichen zwischen beiden Hochkommas entwertet
" ....."	es werden alle Sonderzeichen zwischen den doppelten Hochkommas entwertet, **mit Ausnahme:** **- $** --> bezügl. Variablensubstitution **- $(Kdo) bzw. `Kdo`** --> bezügl. Kommandosubstitution **- ** --> als einzelner Entwerter einer durch die "..." zugelassenen Sonderzeichen

Kommandosubstitution

Die Kommandosubstitution wird immer dort benötigt, wo Argumente eines Kommandos durch Kommandos
erzeugt werden müssen.
Schreibweise:
sh und ksh : `Kdo` # schräg gestellte einfache Hochkommas
nur ksh : $(Kdo)
Beispiele:
$ file $(find ~ -type f)
$ rm $(find ~ -name core)
Beide Kommados haben keinen benützbaren stdin und können somit auch keine Nutzdaten eines anderen
Kommandos über eine Pipe entgegennehmen.

Modulanfang

3-4 Rechenoperationen

Bourne- und Korn-Shell	Korn - Shell
Operand und Operator **durch Blank getrennt**	Operand und Operator **optional durch Blank getrennt**

$ a=5
 	
$ expr $a + 5 => Ausgabe
$ b=`expr $a \*3`
	$ a=5
	
	$ let b=a*3 => keine Ausgabe
	$ ((b = a * 3))

" Punkt- vor Strichrechnung " => Klammerung $ a=5; b=2; c=3 d=`expr $a + $b \* $c ` ; echo $d => 11 d=`expr \( $a + $b \) \* $c ` ; echo $d => 21	**" Punkt- vor Strichrechnung " => Klammerung** $ a=5; b=2; c=3 let d=a+b*c ; echo $d => 11 let d="(a+b)*c" ; echo $d => 21 $ ((d = a + b * c)) ; echo $d => 11 $ ((d = (a + b) * c)) ; echo $d => 21
	$ integer a=5 b=2 c=3 d $ d=$a+$b*$c ; echo $d => 11 $ d="($a+$b)*$c" ; echo $d => 21

Modulanfang
Dokumentanfang

Modul 4 *Reguläre Ausdrücke und das grep Kommando*

- **Überblick**

4-1 Regular Expression Metacharacters

Regular Expression Metacharacters (fgrep interpretiert keine Metacharacters)			
	grep und sed	**egrep und nawk**	**Bedeutung**
	^	^	markiert Zeilenanfang
	$	**$**	markiert Zeilenende
	.	**.**	genau ein beliebiges Zeichen
	*	*	kein, ein oder beliebiges Vorkommen des vorhergehenden Zeichens
	[adg] [a-z]	**[adg] [a-z]**	**ein** Zeichen aus dieser Auswahl **ein** Zeichen aus diesem Bereich
	[^adg] [^a-z]	**[^adg] [^a-z]**	**kein** Zeichen aus dieser Auswahl **kein** Zeichen aus diesem Bereich
	\<		Wortanfang
	\>		Wortende
	\{ \}		Wiederholung des vorangegangenen Zeichens
		( )	Suchauswahl (in Verbindung mit “|“)
		|	entweder oder
		+	ein oder beliebiges Vorkommen des vorhergehenden Zeichens
		?	kein oder ein Vorkommen des vorhergehenden Zeichens

Beispiele :
$ grep '\<The\>' dat1 => findet Zeilen mit The; aber nicht Theater
$ grep 'a\{2\}' dat1 => findet Zeilen mit mindestens "aa"
$ grep '[0-9]\{4\}' dat1 => findet Zeilen mit mindestens 4 aufeinanderfolgenden Zahlen
$ grep '[0-9]\{3,4\}' dat1 => findet Zeilen mit 3 bis 4 aufeinanderfolgenden Zahlen
$ grep 'ab*c' dat1 => findet Zeilen mit : ac, abc, abbc, abbbc .....

$ egrep 'ab+c' dat1 => findet Zeilen mit : abc, abbc, abbbc .....
$ egrep '(Dr.|Frau) Klein' dat1 => findet die Zeilen: Dr. Klein und Frau Klein, aber nicht Klein

$ fgrep '20.000$ (einmaliges Sonderangebot)' dat1 => sucht nach genau diesen ASCII-Zeichen !

Modulanfang
Dokumentanfang

Modul 5 Der *sed Editor*

- **Überblick**

5-1 Der sed Editor

Der sed Editor erhält seine Daten über eine Datei oder eine Pipe.
Diese Daten werden Zeile für Zeile in einen Musterspeicher geladen und dort gemäss der Anweisung
im Kommandoaufruf modifiziert und an stdout weitergeleitet.

Funktion	
s/muster/Ersetzung/Flags (Flags = g, 1-512)	
s/muster/Ersetzung/	sucht in Zeilen nach dem 1. Vorkommen von muster und ändert muster auf Ersetzung
s/muster/Ersetzung/2	sucht in Zeilen nach dem 2. Vorkommen von muster und ändert dieses muster auf Ersetzung
s/muster/Ersetzung/g	sucht in Zeilen nach allen Vorkommen von muster und ändert diese muster auf Ersetzung
s/muster/& string/Flags (Flags = g, 1-512)	
s/muster/& string/	sucht in Zeilen nach dem 1. Vorkommen von muster und

	fügt die angegebene Zeichenkette an.
Adresse1 Funktion	
/muster1/**s**/muster2/Ersetzung/Flags	
/muster1/**s**/muster2/Ersetzung/g	sucht in Zeilen nach muster1 und ändert in diesen Zeilen alle muster2 durch Ersetzung.
/muster/**r file**	sucht in Zeilen nach muster und setzt unter diesen Zeilen jeweils den Inhalt von der Datei file
/muster/**w file**	sucht in Zeilen nach muster und schreibt diese Zeilen in die Datei file
-n /muster/**w file oder** /muster/**w file** 2>/dev/null	wie oben; nur die Standardausgabe wird unterdrückt.
/muster/**p** /muster/**d**	Zeilen mit diesem Muster werden angezeigt / nicht angezeigt
zeilenno**p** zeilenno**d**	Zeilen mit dieser Zeilennummer werden angezeigt / nicht angezeigt (Achtung: ohne trennenden /)
zeilenno**r filename** zeilenno**w filename**	analog obige Beispiele mit "muster"
Adresse1 Adresse2 Funktion	
/muster1/**,**/muster2/**d**	Komma zwischen beiden Mustern = von ... bis Achtung: die von/bis Angabe kann in einer Datei auch mehrfach Anwendung finden ! Als Funktion ist mit Ausnahme von **r filename** alles analog obiger Beispiele verwendbar.
Optionen	
-n $ sed -n '/muster/p' file	unterdrückt das Standardverhalten von sed alle Zeilen **und**

	die gefundenen auszugeben.
-f $ sed -f script_datei file	Angabe einer Datei, in der die sed-Anweisungen stehen.
-e $ sed -e 's/m1/e1/' -e 's/m2/e2/' file	Angabe von zwei oder mehr Anweisungen (bei nur einer Anweisung ist die Option -e optional)
Regular Expression Metacharacters	identisch mit denen von grep (siehe Modul 4)

Modulanfang
Dokumentanfang

Modul 6 *Die nawk Scriptsprache*

- **Überblick**

6-1 Einleitung

nawk erhält seine Daten über eine Datei oder eine Pipe.
Eine Zeile bildet ein Datensatz, welcher in mehrere Datenfelder unterteilt wird.
Feldtrenner ist per Default ein oder mehrere Leerzeichen oder Tabs.

Kommando-Syntax:
$ nawk 'anweisung' file

Anweisungen:
muster { aktion } -> Bei Übereinstimmung mit Muster wird Aktion ausgeführt
muster -> Bei Übereinstimmung mit Muster wird der Datensatz (Zeile) ausgegeben
{ aktion } -> Die Aktion wird für alle Datensätze (Zeilen) ausgeführt

Muster:
- werden abgegrenzt durch einleitenden und abschliessenden Schrägstrich.
- als Muster kann eine alphanumerische Zeichenkette oder/und Sonderzeichen angegeben werden.
$ nawk '/meier/' dat1
$ nawk '/m[ae]ier/' dat1
Hinweis: Die Regular Expression Metacharacters sind identisch mit denen von egrep
(siehe Modul 4)

Aktion:
print Anweisung: entweder Feldangabe (z.B. $2), Zeichenkette ("Nachname:") oder Variable
(z.B.: NR)

$ nawk '{ print $2 }' dat1
$ nawk '{ print "Nachname" , $2 }' dat1
$ nawk '{ print NR , "Nachname" , $2 }' dat1
Hinweis: Das Komma generiert beim Output ein Leerzeichen.
printf Anweisung: analog "print"; nur genauere Formatierung möglich:
Syntax innerhalb einer nawk-Anweisung: **{ printf "Format_Feld1 Format_Feld2\n", $4 , $2 }**
(wobei beliebig viele Feldangaben möglich sind)
(Die trennenden Kommas sind Pflichtangaben; das "\n" generiert ein Newline nach jedem Datensatz)
$ nawk '{ printf "%-20s %-30s\n", $4 , $2 }' dat1
("s" kennzeichnet das Feld als Zeichenkette; das "Minus" als linksbündige Ausgabe (Default=rechtsbündig))

Modulanfang

6-2 BEGIN und END - Anweisungen

Diese Anweisungen werden unabhängig der Datensätze angewendet.
Sie werden für Überschriften und abschliessende Zeilen, als auch zur Initialisierung von Variablen (BEGIN), bzw. deren Ausgabe (END) benützt.
Zudem kann ein abweichender Feldtrenner in der BEGIN Anweisung definiert werden.
Beide Anweisungen können mehrfach angegeben werden !
$ nawk 'BEGIN { print "Teilnehmer wie folgt:" }
> BEGIN { print "Stand: 15.09.2004 }
> { print $4 , $2 }
> END { print "insgesamt:" , NR , "Teilnehmer" }' dat1

Modulanfang

6-3 nawk Variablen

FS Field Seperator Default: Leerzeichen oder Tabulator
Der Feldtrenner kann mit der Option "-f" oder in einer BEGIN Anweisung angegeben werden.
Sind mehrere Feldtrenner gewünscht, wird das Pipe-Symbol benützt.
$ nawk -F"/|:" '{ print $1 }' dat1
$ nawk 'BEGIN { FS="/|:" } { print $1 }' dat1
Beide Beispiele werten einen Schrägstrich **oder** einen Doppelpunkt als Feldtrenner.

OFS Output Field Seperator Default: Leerzeichen
wird in einer BEGIN Anweisung definiert:
$ nawk 'BEGIN { OFS="\t" } { print $4 , $2 }' dat1
Die Felder $4 und $2 werden bei Ausgabe hier durch einen Tabulator getrennt

NR Number of Records
wird von nawk initialisiert und mit "0" vorbelegt.
$ nawk '{ print NR , $4 , $2 }' dat1
Jede Zeile bekommt die Zeilennummer vorangestellt
$ nawk '/M[ae]ier/ { print NR , $4 , $2 }' dat1
Jede Zeile mit Muster: Maier oder Meier bekommt die **entsprechende** Zeilennummer vorangestellt.

NF Number of Fields
NF = Anzahl der Felder
$NF = Inhalt des letzten Feldes
$ nawk '{ print "Zeile" , NR, "hat" , NF , "Felder"}' dat1
Ergebnis:
Zeile 1 hat 8 Felder
Zeile 2 hat 6 Felder
usw.

$ nawk '{ print "Das" , NF, "Feld beinhaltet:" , $NF }' dat1
Ergbenis:
Das 8 Feld beinhaltet: abc
Das 6 Feld beinhaltet: xyz
usw.

Modulanfang

6-4 Vergleiche und logische Operatoren

'/meier/,/mueller/ { print $0 }'	Komma generiert eine **von ... bis** Ausgabe der Datensätze
Alphanumerischer Vergleich:	
'$1 ~ /mueller/ { print $0 }'	Feld $1 **enthält** Zeichenkette mueller
'$1 **!**~ /mueller/ { print $0 }'	Feld $1 **enthält nicht** Zeichenkette mueller
'$1 == "mueller" { print $0 }'	Feld $1 **gleich** Zeichenkette mueller
'$1 **!=** "mueller" { print $0 }'	Feld $1 **nicht gleich** Zeichenkette mueller
Numerischer Vergleich:	
'$1 == 100 { print $0 }'	Feld $1 **gleich** 100
'$1 **!=** 100 { print $0 }'	Feld $1 **nicht gleich** 100
'$1 **>** 100 { print $0 }'	Feld $1 **grösser** 100
'$1 **<** 100 { print $0 }'	Feld $1 **kleiner** 100
'$1 **>=** 100 { print $0 }'	Feld $1 **grösser gleich** 100
'$1 **<=** 100 { print $0 }'	Feld $1 **kleiner gleich** 100
Logische Operatoren	
'$4 == 100 **&&** $5 ~ /mueller/ { print $0 }'	**beide** Bedingungen muessen zutreffen

'$4 == 100 || $5 ~ /mueller/ { print $0 }'	**eine** Bedingung muss zutreffen
Rechenoperationen	
+ - * / %	plus mins multipliziert dividiert modulo
'$3 = $3 * 2 { print $0 }'	Die Zahl in Feld 3 wird mit 2 multipliziert und der Datensatz mit dem neuen Ergebnis ausgegeben.

Modulanfang
Dokumentanfang

Modul 7 *Bedingungen*

- **Überblick**

7-1 Die if Anweisung

Die Prüfung der Bedingung ergibt einen **Exit-Status**, der entscheidet in welchen Zweig die Shell springt:
Exit-Status gleich 0 --> then - Zweig
Exit-Status ungleich 0 --> else - Zweig (fall's dieser optionale Zweig nicht vorhanden ist, wird die if-Anweisung verlassen und das nächstfolgende Kommando nach "fi" ausgeführt.

Der Exit-Status eines vorhergehenden Kommandos kann über die Shell-Variable $? abgerufen werden:

```
grep muster datei
if [ $? -eq 0 ]
then
....
fi
```

Vergleiche und Abfragen		
	Bourne- und Korn-Shell	**Korn-Shell**
numerischer Vergleich		
equal / gleich	[$num1 -eq $num2]	((num1 == num2))

not equal / nicht gleich	[$num1 -ne $num2]	((num1 != num2))
less than / kleiner als	[$num1 -lt $num2]	((num1 < num2))
greater than / groesser als	[$num1 -gt $num2]	((num1 > num2))
less than or equal / kleiner oder gleich	[$num1 -le $num2]	((num1 <= num2))
greater than or equal / groesser oder gleich	[$num1 -ge $num2]	((num1 >= num2))
Text - Vergleich	Variablen sollten in " " gesetzt werden	Variablen sollten in " " gesetzt werden
equals / gleich	["$str1" = str2]	[["$str1" == str2]]
not equal / nicht gleich	["$str1"! = str2]	[["$str1" != str2]]
matches / enthaelt	N / A	[["$str1" == str2]]
does not match / enthaelt nicht	N / A	[["$str1" != str2]]
precedes in lexical order steht i.d. Reihenfolge zuerst	["$str1" < str2]	[["$str1" < str2]]
follows in lexical order steht i.d. Reihenfolge danach	["$str1" > str2]	[["$str1" > str2]]
has length zero / ist leer	[-z "$str1"]	[[-z "$str1"]]
has nonzero length / ist nicht leer	[-n "$str1"]	[[-n "$str1"]]
Feststellen des Dateitypen		
regular file ?	[-f file]	[[-f file]]
directory ?	[-d file]	[[-d file]]
character special file ?	[-c file]	[[-c file]]

block special file ?	[-b file]	[[-b file]]
symbolic link ?	N / A	[[-L file]]
file exist and size greater 0 ?	[-s file]	[[-s file]]
does the file exist ?	N / A	[[-e file]]
Zugriffsrechte und Eigentuemerschaften		
read by user ?	[-r file]	[[-r file]]
altered by user ?	[-w file]	[[-w file]]
executed by user ?	[-x file]	[[-x file]]
owned by EUID ?	N / A	[[-O file]]
owned by GUID ?	N / A	[[-G file]]

Modulanfang

7-2 Logische Operatoren

AND	if [$num1 -eq $num2 **-a** $num1 -eq 5]	if ((num1 == num2 **&&** num1 == 5)) if ((num1 == num2)) **&&** ((num1 == 5))
OR	if [$num1 -eq $num2 **-o** $num1 -eq 5]	if ((num1 == num2 || num1 == 5)) if ((num1 == num2)) || ((num1 == 5))
NOT	if [$# **!**= 1] in Verbindung mit einem Operanten	if [$# **!**= 1] in Verbindung mit einem Operanten

Modulanfang

7-3 elif und case - Anweisung

Bei mehreren Bedingungs-Abfragen kann die if-Anweisung auch beliebig verschachtelt werden:
if [bedingung1]
then

Kommando
elif [bedingung2]
then
Kommando
else
Kommando
fi
Zu beachten, ist der letzt else-Zweig, der bei diesem Konstrukt **Pflicht** ist.

Alternativ kann für diese Aufgabe auch eine case-Anweisung eingesetzt werden:
case wert in
muster1)
Kommando
Kommando;;
muster2)
Kommando;;
***)**
Kommando;;
esac
Hinweise:
- jeder Zweig kann beliebig viele Kommandos enthalten; das letzte muss mit ;; abgeschlossen werden!
- Der letzte Zweig *) wird auch als Default-Muster bezeichnet.

exit Anweisung
Die exit Anweisung beendet das Script an dieser Stelle.
Optional kann eine Zahl (zwischen 0 und 255) als Argument übergeben werden.
Diese Zahl dient zur späteren Orientierung, wenn der Wert abgerufen wird.
$ ./script.ksh
.....
exit 2
$ echo $?
2
$

Modulanfang
Dokumentanfang

Modul 8 *Interaktive Scripte*

- **Überblick**

8-1 Das read Statement

Die Eingabe wird unterteilt in "Tokens" (Woerter -> getrennt durch white spaces)
read - nimmt die Eingabe entgegen bis zum abgesetzten newline.
Die Tokens werden in Reihenfolge den angegebenen Variablen zugewiesen.
echo "Enter two strings: \c"
read var1 var2

Hinweise:
- Wenn **mehr Woerter als Variablen** angegeben werden, bekommt die letzte var alle ueberzaehligen Woerter.
- Wenn **mehr Variablen als Woerter** angegeben werden, bleiben die überzähligen Variablen existent, enthalten aber keinen Wert.
- Wenn **keine var** angegeben ist:
  ksh = generiert eine **Variable REPLY**, die alle Wörter enthält
  sh = Fehlermeldung

Kurzschreibweise / nur ksh = $ read name?"Bitte Eingabe : "

Modulanfang

8-2 Filedescriptoren

	Bourne- und Korn Shell	**Korn Shell**
FD einer Datei zuweisen / Output	exec 3> /tmp/dat1	exec 3> /tmp/dat1
FD einer Datei zuweisen / Intput	exec 4< /tmp/dat2	exec 4> /tmp/dat2
Lesen vom FD / speichern in Var	read <& 3 var1	read -u3 var1
Nutzdaten des cmds werden an den FD geschickt	echo $var1 >& 4	print -u4 $var1
FD schliessen	exec 3<&-	exec 3<&-
Anmerkung: neben den Standard-FD's (0=stdin. 1=stdout, 2=stderr) koennen max. 7 eigene definiert werden (fd3 bis fd7)		

Modulanfang
Dokumentanfang

Modul 9 *Schleifen*

- **Überblick**

9-1_ for und while Schleife

for - Schleife	
	Uebergabe der Argumentliste an die Laufschleife mittels:
for var in name1 name1 do print "Value of var is : $var" done	explizite Angabe von Strings
variable="wert1 wert2 wert3" for var in $variable do print "Value of variable is $var" done	dem Inhalt einer Variablen
$ script.ksh arg1 arg2 for var in $# do print "Value of var is $var" done	Kommandozeilen-Argumente
for var in $(cat dat1) do print "Value of var is $var" done	Kommandosubstitution
for var in $(cat dat*) do print "Value of var is $var" done	Dateinamensgenerierung Achtung:: kann keineDatei generiert werden, wird der 4 Zeichen lange String dat* uebergeben
while Schleife	
	Beispiele der Bedingungs - Pruefung :
integer num=5 while ((num > 0))	numerischer Vergleich

do
  print "$num"
  num=$num-1
 done	

print "\nEnter a word :\c"
while read var
do
  print "$var"
  print "\nEnter another word :\c"
 done
	Keyboard Input
	= Endlosschleife bis User
	CTRL + D drueckt

while read name1 name2 do print "The full name is $name1 $name2" done < name.txt	zeilenweises Einlesen einer Datei

integer num=5
while ((num > 0))
do
  print "$num"
  num=$num-1
 done > datneu
	Schreiben in eine Datei

Modulanfang

9-2 Abbrüche

exit	akt. Shellscript beenden mit dem Endestatus des letzten davor gelaufenen Kommandos
exit 1	akt. Shellscript beenden mit Endestatus 1 (der Endestatus wird in die Variable ? der übergeordneten Shell geschrieben)
return	akt. Funktion beenden mit dem Endestatus des letzten davor gelaufenen Kommandos
return 1	akt. Funktion beenden mit Endestatus 1 (der Endestatus wird in die Variable ? der aktuellen Shell geschrieben) ((return ausserhalb von Funktionen entspricht exit !))
break	akt. Schleife abbrechen
break 2	akt. und äussere Schleife abbrechen
continue	akt. Schleifenkörper beenden und nächsten Schleifendurchlauf beginnen (Achtung bei der while-Schleife: Das Hoch-/runtersetzen der Bedingung muss **vor** dem continue Statement erfolgen !)

Modulanfang
Dokumentanfang

Modul 10 *Fortgeschrittene Variablen, Parameter und Argument Listen*

- **Überblick**

10-1 Stringbearbeitung bei UNIX - Variablen (Korn Shell)

${#VAR}	Anzahl der Zeichen in VAR
${VAR%muster}	Variablenwert ohne einmaliges Vorkommen von *muster* am Ende.
${VAR%%muster}	Variablenwert ohne beliebig häufiges Vorkommen von *muster* am Ende.
${VAR#muster}	Variablenwert ohne einmaliges Vorkommen von *muster* am Anfang.
${VAR##muster}	Variablenwert ohne beliebig häufiges Vorkommen von *muster* am Anfang.

Hinweis: Muster können Sonderzeichen der Dateinamengenerierung enthalten

Beispiele:

```
VAR=Hans-Meiser
echo ${#VAR}        --> 11
echo ${VAR%e*}      --> Hans-Meis
echo ${VAR%%e*}     --> Hans-M
echo ${VAR#*e}      --> iser
echo ${VAR##*e}     --> r
```

Modulanfang

10-2 Korn Shell Arrays

- Korn Shell Arrays werden im Gegensatz zu Variablen nicht deklariert sondern entstehen durch den Gebrauch / das Ansprechen.
- Korn-Shell Arrays sind eindimensional und können bis zu 4096 Elemente beinhalten (0 bis 4095)
- Arrays können als strings oder integer erstellt werden (Default = strings) (integer Arrays müssen vorher allerdings als Integer-Variable deklariert werden)
- Ein array kann nicht mit dem export Befehl an Subshells vererbt/globalisiert werden.

$ arr[0]=Peter
$ arr[1]=Hans
oder :
$ set -A arr Peter Hans
	Ein string-Array erzeugen
	
	Hinweis: Die Kurzschreibweise "set -A"
	wird nicht von allen Korn-Shell's unterstützt.
	Weiterhin beginnt die Belegung der Array-Elemente
	zwingend bei 0.

$ integer num
$ num[0]=10
$ num[1]=20
	Ein integer-Array erzeugen

$ print ${num[0]}	Anzeige des **ersten** Elemente
$ print ${num[0+3]}	Anzeige des **dritten** Elementes **nach dem ersten** Elemente
$ print ${num[*]}	Anzeige **aller** Elemente
$ print ${#num[1]}	**Länge** des zweiten Elementes
$ print ${#num[*]}	**Anzahl** der Elemente
$ unset arr	Array löschen

Modulanfang
Dokumentanfang

Modul 11 *Funktionen*

- **Überblick**

11-1 Einleitung

Syntax Korn- und Bourne-Shell

```
function_name ()
{
Kommandos
}
```

Syntax Korn-Shell

```
function function_name
{
Kommandos
}
```

- Funktionen laufen in der aktuellen Shell (es wird keine Subshell eröffnet)
- Nach Beendigung des Scriptes sind auch die Funktionen nicht mehr existent
- Übergebene Argumente können als Stellungsparameter im Script benützt werden; sind aber nach Funktionsausführung nicht für dass aktuelle Script vefügbar.

Modulanfang

11-2 FPATH Variable

Funktionen können auch in einer Datei deklariert werden:
- jede Funktion muss in **einer** Datei abgelegt werden.
- der Funktionsname sollte dem Dateinamen entsprechen.

Modulanfang
Dokumentanfang

Modul 12 *Traps*

- **Überblick**

12-1 Signale und Anwendung

Signal 2 (INT)	$ kill -2 PID oder Control-C => interrupt / abbrechen
Signal 3 (Quit)	$ kill -3 PID oder Control-\ => quit / abbrechen, sowie Coredump des Prozesses
Signal 9 (KILL)	$ kill -9 PID => unabdingbares terminieren (kann nicht getrapt werden)
Signal 15 (TERM)	$ kill -15 PID => "ordnungsgemaesses" terminieren
Signal 23 (STOP)	$ kill -23 PID => (Hintergrund)-prozess wird gestoppt (kann nicht getrapt werden)
Signal 24 (TSTP)	$ kill -24 PID oder Control-Z => Vordergrundprozess wird gestoppt i.d. Hintergrund gelegt
Signal 25 (CONT)	$ kill -25 PID => gestoppter Hintergrundprozess arbeitet wieder
trap 'action' signal [signal2 ... signalN]	wobei 'action' eine optionale Angabe ist.
trap 'echo "CTRL-C not allowed"' INT **oder:** trap " INT	Beim zweiten Beispiel wird zwar das Signal abgefangen, dem User aber keine Meldung ausgegeben.
trap - INT	setzt das Signal auf den Default zurück (fall's es vorher abgefangen wurde, ist nach dieser Zeile Control-C wieder möglich)
trap 'echo "arg not found"' ERR	Sobald ein Kommando oder Anweisung einen Exitstatus ungleich 0 ergibt (in der Variable ?) wird die Aktion ausgeführt

Modulanfang
Dokumentanfang