```
7.5. Die Korn-Shell (ksh)
=========================

Die Korn-Shell ist eine echte Erweiterung der Bourne-Shell.
Sie ist eine Obermenge der Bourne-Shell. Alle Bourne-Shell-Scripte
sind ohne Änderung unter einer Korn-Shell abarbeitbar. Vorbild
für die Terminalschnittstelle der Korn-Shell war die C-Shell.
Viel Features der Terminalschnittstelle der C-Shell wurden
in die Korn-Shell eingearbeitet.

Erweiterungen der Korn-Shell gegenüber der Bourne-Shell:

- Editieren der Kommandozeile mit vi und emacs
- History-Mechanismus der C-Shell
- Alias-Mechanismus
- Job-Control der C-Shell
- cd-Kommando der C-Shell
- Tilde-Mechanismus der C-Shell
- select-Kommando für einfache Menü-Programmierung
- interne Arithmetik für ganze Zahlen
- Substring-Operatoren
- Attribute von Variablen beschreiben die Art des Inhalts
- eindimensionale Felder von Variablen
- lokale Variable in Funktionen
- Verbessertes Laufzeitverhalten
```



```
Metazeichen
-----------
wie in der Bourne-Shell:
   |      - Pipe
   *      - kein, ein oder mehr Zeichen
   ?      - ein beliebiges Zeichen
   [...]  - eines der in den Klammern angegebenen Zeichen
   [!...] - nicht eines der in den Klammern angegebenen Zeichen
   ;      - Trennzeichen für Kommandos
   &      - Kommando in Hintergrund, E/A-Verkettung
   `kommando` - Eretzung durch Standardausgabe
   (  )   - Subshell benutzen
   $      - Leitet eine Shellvariable ein
   \      - Maskierung von Metazeichen
   ' ...' - Shellinterpretation innerhalb der Apostrophs wird
            abgeschaltet
   "..."  - Shellinterpretation innerhalb der Doppelapostrophs
            wird ausgeschaltet ausser für '$', '`' und '\'
   #      - Beginn eines Kommentars
   =      - Wertzuweisung
   &&     - bedingte Ausführung von Kommandos
   ||     - bedingte Ausführung von Kommandos
   >      - E/A-Umlenkung
   <      - E/A-Umlenkung
```

```
Zusätzliche Metazeichen in der ksh:                                  !!!!!!

   >|              - E/A-Umlenkung mit Überschreiben von Files
   <>              - Datei zum Lesen und Schreiben eröffnen
   $(kommandos)    - alternative Form für Kommandosubstitution
                     `kommando`
   $10,$11, ..     - weitere Parameter
   ?(pattern)      - Patternmatching, kein oder einmal
   *(pattern)      - Patternmatching, kein, einmal oder mehrmals
   +(pattern)      - Patternmatching, einmal oder mehrmals
   @(pattern)      - Patternmatching, genau einmal
   !(pattern)      - Patternmatching, deckt Strings ab, die nicht durch
                     das Pattern abgedeckt werden
   ~               - Dateinamenexpandierung: Homedirectory des aktuellen
                     Nutzers
   ~nutzername     - Dateinamenexpandierung: Homedirectory des spezifizierten
                     Nutzers
   ((ausdruck))    - Arithmetische Bewertung des Ausdrucks
   kommando |&     - Starten eines Hintergrundprozesses mit Pipe zum
                     aktuellen Prozeß, für Scripte sinnvoll
                     Benötigt für die Kommunikation "print -p" und "read -p"
```



```
Aufbau eines Kommandos - 1.Teil
-------------------------------
   <Kommando> ::= <einfaches Kommando> | ...

   <einfaches Kommando>::= <Kommandoname> { <Argument> }

        Folge von Wörtern, die durch Leerzeichen (Tabulatoren) voneinander
        getrennt sind.  Das erste Wort gibt den Programmnamen an. Alle
        weiteren Worte sind die Argumente des Programms.
                kommandoname argument1 argument2 argument3
        Kommandoname wird intern auch als argument0 bezeichnet.

   <Pipeline von Kommandos> ::= <Kommando> { "|" <Kommando> } |        !!!!!
                   "|" - Pipe, die Standardausgabe des vorangegangenen
                           Kommandos wird auf die Standardeingabe des
                           nachfolgenden Kommandos geleitet.

   <Liste von Kommandos> ::= <Kommando> { <NL> <Kommando> } |
                             <Kommando> { ";" <Kommando> } |
                             <Kommando> { "&&" <Kommando> } |
                             <Kommando> { "||" <Kommando> }

                  <NL> - Kommandos werden nacheinander ausgeführt
                         (einzelne Kommandos stehen in mehreren Zeilen)

                    ;  - Kommandos werden nacheinander ausgeführt

                    && - das nachfolgende Kommando wird ausgeführt, wenn
                         das vorangegangene Kommando den Returnwert 0
                         (true) liefert.
```

```
                    || - das nachfolgende Kommando wird ausgeführt, wenn
                         das vorangegangene Kommando einen Returnwert
                         ungleich 0 (false) liefert.

      Returnwert (Rückkehrkode): Jedes Programm liefert einen Returnwert.
              0 wird als True interpretiert und alles andere als False.

   <Kommando> ::= <einfaches Kommando> |
                   <Pipeline von Kommandos> |
                  "(" <Liste von Kommandos> ";" ")" |
                  "{" <Liste von Kommandos> ";" "}" |  ....

                  "(" und ")" - Zusammenfassung von Kommandos, die in einer
                                Subshell abgearbeitet werden

                  "{" und "}" - Zusammenfassung von Kommandos, die in der
                                gleichen Shell ablaufen.

Kommandosubstitution
--------------------
`<Kommando>`  - Die Ausgabe auf der Standardausgabe des Kommandos werden
                in die Kommandozeile eingefügt. Metazeichen behalten
                ihre Bedeutung. Schachtelung ist mittels Maskierung
                möglich.
                Wenn keine E/A-Umlenkung benutzt wird, wird auch keine
                neue Subshell gestartet.                     !!!!!!
$(<Kommando>) - wie oben, aber leichtere Schachtelung ist möglich. !!!!!!

                echo anfang $(echo zweite echo $(date) nach nach date )
```



```
Shellvariable
-------------
     <Shellvariable>   ::= <Nicht-Ziffer> { <Nicht-Ziffer> | <Ziffer> }
     <Nicht-Ziffer> ::= "a"|"b"|...|"z"|"A"|"B"| ... |"Z"|"_"
     <Ziffer>       ::= "0"..."9" ...

      Wertzuweisung für Shellvariable:

           <Bezeichner>=<wert>

      Zugriff auf eine Shellvariable:

            $<Bezeichner>   oder ${<Bezeichner>}


   Felder von Shell-Variablen:                            !!!!!!!!!!!!!!
        Wertzuweisung:
           <Bezeichener>"["<index>"]="<Wert>
        Zugriff auf ein Element
           ${<Bezeichener>"["<index>"]"}

        Spezielle Zugriffe auf Felder
          $<Bezeichner>       -  liefert Wert des 0. Elements des Feldes
                                 gleichbedeutend mit ${<Bezeichner>[0]}
          ${<Bezeichner>[*]}
          ${<Bezeichner>[@]}  - liefert alle Werte des Feldes
          ${#<Bezeichner>[*]}
          ${#<Bezeichner>[@]} - liefert die Anzahl der belegten Elemente
                                des Feldes
```

Festlegung von Eigenschaften von Variablen

```
    typeset -L<Number> <Bezeichner>["="<Wert>]
              Löschen von führenden Leerzeichen, Länge maximal <Number>
              von links
    typeset -R<Number> <Bezeichner>["="<Wert>]
              Löschen von folgenden Leerzeichen, Länge maximal <Number>
              von rechts
    typeset -Z<Number> <Bezeichner>["="<Wert>]
              Auffüllen mit führenden Nullen, Länge maximal <Number>
    typeset -l <Bezeichner>["="<Wert>]
              Kleinbuchstaben
    typeset -u <Bezeichner>["="<Wert>]
              Großbuchstaben
        Kombinationen sind zulässig z.B.
          > typeset -uR5 x="asdfghijk"
          > echo $x
          GHIJK
          >
    typeset -i <Bezeichner>["="<Wert>]
              Integer Variable

    typeset -r <Bezeichner>["="<Wert>]
              Readonly-Variable
    typeset -x <Bezeichner>["="<Wert>]
              Export - Umgebungsvariable


    Löschen von Shellvariablen:
          unset <Shellvariable>
```



```
    Export von Shellvariablen:
          export <shellvariable> [ <shellvariable> ]
          typeset -x <shellvariable>  [ <shellvariable> ]
```

Quoting - Maskieren von Metazeichen

```
     Quotings:

         \    - vorgestellter "\" - das nachfolgende Metazeichen wird
                                    als normales Zeichen interpretier.
       ' ... ' - Text in einfachen Apostrophs - Alle im Text enthaltenen
                                    Zeichen werden als normale Zeichen
                                    interpretiert.  Auch "\" verliert
                                    seine Bedeutung.
       " ... " - Text in Doppelapostrophs - Alle Metazeichen außer:
                                                "\"  "`"  "$"
                            werden als normale Zeichen interpretiert.
```

Vordefinierte automatische Korn-Shellvariable (wie Bourne Shell):
Änderung sinnlos

```
$- - Aufrufoptionen der Korn-Shell
$? - Returnwert des letzten Kommandos
$$ - Prozeßnummer der aktuellen Shell
$! - Prozeßnummer der zuletzt asynchron gestarteten Kommandos
$# - Zahl der Positionsparameter
$* - entspricht "$1 $2 ..."  (theoretisch)
$@ - bedeutet "$1" "$2" ...

ERRNO    - Fehlernummer des letzten Kommandos
LINENO   - Scripten - Zeilennummer der aktuellen Zeile
OLDPWD   - vorheriges Workingdirectory
OPTARG   - für getopts - aktuelles Argument, falls vorhanden
OPTIND   - für getopts - Nummer des aktuellen Arguments
PPID     - Prozeßnummer der Vatershell
PWD      - Working Directory
REPLY    - Eingabewert bei select oder read ohne Parameter
SECONDS  - Laufzeit der Shell in Sekunden
```


einige Standard-Korn-Shell-Variable (etwas mehr als bei sh):

```
CDPATH   - Suchpfad für rel. Pfadangaben für das Shell-Kommando cd

COLUMNS  - Anzahl der Spalten  (80)

EDITOR   - Name des benutzten Editors (/bin/ed)

ENV      - Name der Datei für Umgebungseinstellungen für ksh (-)

FCEDIT   - Name des Editors für fc-Kommando ( - )

HISTFILE - Filename des Historyfiles  ( $HOME/.sh_history)

HISTSIZE - Länge des Historyfiles in Kommandos (128)

HOME     - Homedirectory (aus /etc/passwd)

IFS    - internal Field Seperators - Wort-Trennzeichen für die Shell
         (Leerzeichen, Tabulator, NL)

LINES  - Zahl der Zeilen (24)

LOGNAME, USER  - Login-Name des Nutzers

MAIL   - Mailfolder

PATH     - Pfad für ausführbare Kommandos
```

```
PS1      - Primär-Promptstring
           (Standard: $ - normaler Nutzer, # - root)
             PS1="`pwd` $ "
             PS1="$USER@`hostname` `pwd` > "
             PS1="$USER@`hostname` `pwd` ! > "

PS2      - Sekundär-Promptstring, wenn sich eine Kommandozeile über
           mehrere Zeilen erstreckt (Standard: > ).

PS3     - Prompt für "select"

SHACCT  - Datei für Abrechnungsinformationen (-)

SHELL  - Name der aktuellen Shell (/etc/passwd)

TERM   - Terminaltype, wichtig für vi

TMOUT  - Logout-Zeit in Sekunden, 0 - unendlich (0)

TZ     - Timezone

VISUAL - Name des Kommanodzeileneditors: vi, emacs, gmacs (-)
```


Spezielle Operationen mit Variablen:

```
${variable:-wert}
${variable-wert}      Verwendung von default Werten (sh)

${variable:=wert}
${variable=wert}      Zuweisung von default Werten (sh)


${variable:?wert}
${variable?wert}       Fehlermeldungen (sh)

${variable:+wert}
${variable+wert}       Benutzung von alternativen Werten (sh)

${#variable}          Länge des Strings, der in der Variablen
                      gespeichert ist.
                                                    ab hier !!!!!
${#*}                 Anzahl der Parameter

${#variable[*]}
${#variable[@]}       Anzahl der Elemente eines Arrays

${variable#pattern}    entfernen der kleinsten möglichen Zeichenkette
                       aus der Variablen von links, auf die das
                       Pattern paßt.

${variable##pattern}   entfernen der größten mögliche Zeichenkette aus
                       der Variablen von links, auf die das Pattern
                       paßt.
```

```
${variable%pattern}   entfernen der kleinsten möglichen Zeichenkette
                      aus der Variablen von rechts, auf die das
                      Pattern paßt.

${variable%%pattern}  entfernen der größten mögliche Zeichenkette
                      aus der Variablen von rechts, auf die das
                      Pattern paßt.

                      Beispiel:
                      Kshell 49 > VA="Wasserflasche"
                      Kshell 50 > echo ${VA}
                      Wasserflasche
                      Kshell 51 > echo ${VA%s*}
                      Wasserfla
                      Kshell 52 > echo ${VA%%s*}
                      Wa
                      Kshell 53 > echo ${VA#*s}
                      serflasche
                      Kshell 54 > echo ${VA##*s}
                      che
                      Kshell 55 >
```



```
Expandieren von Dateinamen
--------------------------

    * - beliebige Zeichenfolge ( auch leer)

    ? - ein beliebiges Zeichen (nicht leer)

    [...] - ein beliebiges Zeichen aus der Menge ...
    [!...] - kein Zeichen aus der Menge

    folgende Zeichen werden nur erkannt, wenn sie explizit im Muster
    angegeben wurden:

     . (Punkt am Anfang eines Dateinamen)
     /.
     /

   ?(pattern[|pattern]..)  - deckt kein oder ein Auftreten der angegebenen
                             Pattern ab
   *(pattern[|pattern]..)  - deckt kein, ein oder mehrere Auftreten der
                             angegebenen Pattern ab
   +(pattern[|pattern]..)  - deckt ein oder mehrer Auftreten der angegebenen
                             Pattern ab
   @(pattern[|pattern]..)  - deckt genau ein Auftreten der angegebenen
                             Pattern ab
   !(pattern[|pattern]..)  - deckt alles ab, was keines der angegebenen
                             Pattern enhält
```

```
Beispiel:

Kshell 79 > ls                                      k2
b          b1        b11     b111    b1111
b11112222  b111222   b1122   b12     b2
b22        b222      b2222   k1      k2
Kshell 80 >
Kshell 81 >ls b?(1)
b  b1
Kshell 82 >ls b*(1)
b  b1  b11  b111  b1111
Kshell 83 >ls b+(1)
b1  b11  b111  b1111
Kshell 84 >ls b@(1)
b1
Kshell 85 >ls b?([1-2])
b  b1  b2
Kshell 86 >ls b*([1-2])
b        b1     b11  b111  b1111  b11112222
b111222  b1122  b12  b2    b22    b222
b2222
Kshell 87 >ls b+([1-2])
b1       b11    b111  b1111  b11112222
b111222  b1122  b12   b2     b22
b222     b2222
Kshell 88 >ls b@([1-2])
b1       b2
Kshell 89 >ls b!([1-2])*
b
Kshell 90 >
```



```
Ein- und Ausgabe
----------------

Standardeingabe:          Kanal 0
Standardausgabe:          Kanal 1
Standardfehlerausgabe:    Kanal 2

> file        - Umlenkung Standardausgabe in das File   file
>| file       - Umlenkung Standardausgabe in das File   file       !!!!
                immer mit Überschreibung (ohne noclobber zu beachten)

fd> file      - Umlenkung des Ausgabekanals  fd  in das File   file
fd>| file     - Umlenkung des Ausgabekanals  fd  in das File   file  !!!!
                immer mit Überschreibung (ohne noclobber zu beachten)

>&-           - Schließen Standardausgabe  ( besser: > /dev/null )

fd>&-         - Schließen des Ausgabekanals  fd

< file        - Umlenkung Standardeingabe von  file

fd<file       - Umlenkung des Eingabekanals fd von  file

<&-           - Schließen Standardeingabe - ( besser < /dev/null )

fd<&-         - Schließen des Eingabekanals  fd

fd1>&fd0      - Umlenkung der Ausgabe des Kanals fd1 auf den eröffneten
                Kanal fd0, in der crontab beliebt
                     $ dauerlaeufer 1> /dev/null 2>&1 &
```

```
>> file        - Umlenkung Standarausgabe mit Anfügen
                      $ echo Anfang des Scripts >> Protokollfile

fd>> file      - Umlenkung des Ausgabekanals  fd  mit Anfügen
                      $ echo Anfang des Scripts 1>> Protokollfile

<<ENDE         - Lesen aus Shellscript bis ENDE


read <variable> - Lesen einer den Wert von der Standardeingabe und
                  Wertzuweisung zur Variablen


`Kommando`     - Umlenkung der Standardausgabe in eine Zeichenkette
$(Kommando)                                                          !!!!

                   echo "Start des Scripts: `date`" >> protokoll
                   echo "Start des Scripts: $(date)" >> protokoll

 |&              - Starten eines Hintergrundprozesses und Umlenkung    !!!!
                  der Standardeingabe und Standardausgabe dieses
                  Prozesses auf den aktuellen Prozeß.
```



```
Beispiel:
  Kshell k3 90 > cat k3                                        k3
  #!/bin/ksh
  # Kommunikationsprogramm
  # starten des Verarbeitungsprogramms
  ./k4 |&
  while read wert?"Eingabe: "
     do
        print -p $wert
        read -p ergebnis
        print Ausgabe von k3: $ergebnis
     done
  Kshell k3 91 > cat k4
  #!/bin/ksh                                                   k4
  # Verarbeitungsprogramm
  # wird von k3 gestartet
  while read eingabe
    do
          print  errechnet in k4: $eingabe
  done
  /Kshell 92 > ./k3
  Eingabe: adsdasf
  Ausgabe von k3: errechnet in k4: adsdasf
  Eingabe: asdfasdf
  Ausgabe von k3: errechnet in k4: asdfasdf
  Eingabe: qewrqewr
  Ausgabe von k3: errechnet in k4: qewrqewr
  Eingabe: asdfasdf
  Ausgabe von k3: errechnet in k4: asdfasdf
  Kshell 93 >
```

```
Shell-Scripte
-------------

Shell-Script: File mit gültigen Shell-Kommandos

   Aufruf:  ksh <Shell-Script-Name>                  (1)
        oder
            <Shell-Script-Name>                      (2)

   Am Anfang eines Shell-Scriptes sollte immer die benutzte Shell als
   Spezialkommentar (Major-Number: #!/bin/ksh) eingetragen sein.
   Bei der 1. Variante muß das Shell-Script nur lesbar sein.
   Bei der 2. Variante muß das Shell-Script zusätzlich noch ausführbar
   sein.

   Paramter bei Korn-Shell-Scripte
      JA - erstmal die Parameter 1..9,10,11, ....
           $1 .. $9 $10 $11 ....

      shift funktioniert auch weiterhin.
                                                               k1
```



```
Kommandos - 2.Teil
------------------

   <Kommando> ::= <einfaches Kommando> |
                 "(" <Liste von Kommandos> ";" ")" |
                 "{" <Liste von Kommandos> ";" "}" |
                  <if-Kommando> | <case-Kommando> |
                  <while-Kommando> | <until-Kommando> |
                  <for-Kommando> |
                  <[[-Kommando> | <select-Kommando> |...

   <if-Kommando>::= "if" <Liste von Kommandos>
                         "then" <liste von Kommandos>
                        {"elif" <Liste von Kommandos>
                            "then" <Liste von Kommandos> }
                        ["else" <Liste von Kommandos>]
                    "fi"

        Die Kommandoliste nach "if" wird abgearbeitet. Der Returnwert des
        letzten abgearbeiteten Kommandos bestimmt die Verzweigungsbedingung.
        Ist der Wert gleich Null, wird die Kommandoliste nach dem "then"
        abgearbeitet.  Ist der Wert ungleich Null, wird die Kommandoliste
        nach dem "else" abgearbeitet, falls diese vorhanden ist. Ist ein
        "elif" Abschnitt vorhaden, wird mit diesem verfahren, wie bei "if".
        Der Abschnitt wird anstelle von "else" abgearbeitet.
        Beachte: Vor "then", "elif", "else", "fi"  muß ein <NL> oder ein ";"
        als Trennzeichen stehen(werden als einfache Kommandos aufgefasst).
```

```
<case-Kommando>::= "case" <Wort> "in"
                        <Muster>")" <Kommandoliste> ;;
                       {<Muster>")" <Kommandoliste> ;; }
                   "esac"
```

Das Wort <Wort> wird der Reihe nach mit den Mustern vor den Kommandolisten verglichen. Wenn ein Muster "matchet" wird die zugehörige Kommandoliste abgearbeitet und das case-Kommando beendet (Fortsetzung nach "esac"). Es gelten die gleichen Regeln wie bei der Dateierweiterung ("[..]", "*", "?").

```
<while-Kommando>::= "while" <Kommandoliste>
                     "do"
                          <Kommandoliste>
                     "done"
```

Die Kommandolist nach dem "while" wird abgearbeitet. Ist der Returnwert des letzten Kommandos 0 (True) wird die Kommandoliste nach dem "do" abgearbeitet. Danach wird die Kommandoliste nach dem "while" wieder abgearbeitet. Dies geschieht solange, wie der Returnwert des letzten Kommandos der Kommandoliste nach dem "while" gleich 0 (True) ist. Ist der Wert ungleich 0, wird das while-Kommando beendet (Fortsetzung nach dem "done"). Durch das Buildin-Kommando "break" kann das while-Kommando jederzeit beendet werden. Durch das Buildin-Kommando "continue" wird der nächste Schleifendurchlauf gestartet.


```
<until-Kommando>::= "until" <Komandoliste>
                     "do"
                          <Kommandolist>
                     "done"
```

Die Kommandolist nach dem "until" wird abgearbeitet. Ist der Returnwert des letzten Kommandos ungleich 0 (False) wird die Kommandoliste nach dem "do" abgearbeitet. Danach wird die Kommandoliste nach dem "until" wieder abgearbeitet. Dies geschieht solange, wie der Returnwert des letzten Kommandos der Kommandoliste nach dem "until" ungleich 0 (False) ist. Ist der Wert gleich 0 (True), wird das until-Kommando beendet (Fortsetzung nach dem "done"). Durch das Buildin-Kommando "break" kann das until-Kommando jederzeit beendet werden. Durch das Buildin-Kommando "continue" wird der nächste Schleifendurchlauf gestartet.

```
<for-Kommando>::= "for" <Laufvariable> [ "in" <wort> {<wort>} ]
                   "do"
                       <Kommandolist>
                   "done"
```

Die Laufvariable nimmt nacheinander die Werte aus der Wortliste an und mit jedem Wort werden die Kommandos der Kommandoliste abgearbeitet. Fehlt der "in"-Part, wird anstelle der Wordliste die Parameterliste des Shell-Scripts (aktuelle Shell-Funktion) benutzt. Durch das Buildin-Kommando "break" kann das for-Kommando jederzeit beendet werden. Durch das Buildin-Kommando "continue" wird der nächste Schleifendurchlauf gestartet.

<[[-Kommando> ::= "[[<Ausdruck>]] **!!!!!!**

Für Ausdruck können die gleichen Ausdrücke wie beim Kommando test benutzt werden. Ist der Ausdruck wahr, wird als Returnwert 0 geliefert, sonst 1. Anstelle des -a Operators && zu benutzen und anstell des -o Operators || zu benutzen. Bei Gleich- und Ungleichheitstests können auf der rechten Seite auch Pattern benutzt werden.

```
Kshell 103 > [[ abcc = a*c ]]
Kshell 104 > echo $?
0
Kshell 105 >
```

```
<select-Kommando> ::= "select" <variable> "in" <wort> { <wort> }
      "do"
          <Kommando-Liste>
      "done"
```

Das Select-Kommando dient zur einfachen Realisierung von Menues. Als erstes wird die Wortliste zeilenweise mit vorangestellter Ziffer ausgegeben. Zuvor werden die Worte der Wortliste expandiert. Danach wird der Prompt PS3 ausgegeben und eine Anwort eingelesen. Ist es eine zulässige Antwort (gültige Ziffer), wird der Variablen das entsprechende Wort als Wert zugewiesen, andernfalls erhält die Variable einen Nullwert. Die Variabel REPLY enthält in jedem Fall den Eingabewert. Die Select-Anweisung wird unter folgenden Bedingungen beendet:

- break-Kommando, - return-Kommando
- exit-Kommando, - EOF als Terminaleingabe

Beispiel: **k5**

```
#!/bin/ksh
#
select wert in ls wc cat ende
do
  echo REPLY: $REPLY
  echo wert: $wert
  case $wert in
   "ls")    echo ls ;;
   "wc")    echo wc ;;
   "cat")   echo cat ;;
   "ende")  break ;;
  esac
done
```

```
interne Shell-Kommandos
-----------------------

    <einfaches Kommando>::=   ....| <interne Shell-Kommando>

interne Shell-Kommanod - Kommando innerhalb der Shell realisiert.

#
     Kommentar
         # Das ist ein Kommentar bis Zeilenende

: {<Argumente>}
     Nullkommando
       wie Kommentar, aber ";" als Trennzeichen erlaubt, das ein
       weiteres Kommando folgt.
         : das Kommando ls folgt ; ls

. <Kommandodatei>
     einlesen von Kommandos aus dem File in der aktuellen Shell

alias [-tx] [a-name[=wert]]                                         !!!!
     setzen eines Aliases a-name für den Wert wert.
       alias dir='ls '                      wie bei C-Shell

bg  [job]                                                           !!!
       Jobcontrol - Angehaltener Job im Hintergrund ausführen
```



```
break [n]
     verlassen von Schleifenanweisungen (while, until, for).
       n gibt die Anzahl der zu verlassenden Schleigen an.
       Standard ist 1.

cd  directoy
     Definition des Working Directory (Current Directory)
       Nur für die aktuelle Shell und nachfolgende Kommandos gültig.

cd -                                                                  !!!!
     umschalten zwischen zwei Directories

continue [n]
     Beenden von Schleifen in Schleifenanweisung (while, unitil, for)
       Es wird mit der Abarbeitung der Schleifen bedingung fortgesetzt.
       n gibt die Zahl der Schleifen an. Standard ist 1.

echo {<argument>}
     Ausgabe der Argumente auf die Standardausgabe

eval {<argument>}
     Abarbeiten der Argumente in einer Shell
       1. Argument ist das Kommando, anschließend wird die aktuelle Shell
       fortgesetzt.
       Achtung: Argumente werden zweimal durch die Shell interpretiert!!!!
                 eval xxx=Tools "ls $xxx"
                 eval xxx=Tools export xxx \; "ls $xxx"
                 eval xxx=Tools export xxx \; "ls \$xxx"
```

```
exec {<argumente>}
     Ausführen der Argumente als Kommando im aktuellen Shell-Prozeß.
       1. Argumente ist das Kommando. Die Shell wird beendet.
       Ohne Argumente werden nur die E/A-Umlenkungen für die aktuelle
       Shell übernommen.

exit [<Rückkehrkode>]
     beenden der Shell mit einem Rückkehrkode

export {<argument>}
     Übergabe von Variablen der Shell an die Umgebung.
       Definition von Umgebungsvariable.

fc -e - [alt=neu] [Kommando]                                        !!!!!
     Ausführen des vorherigen Kommandos mit Substitution

fc [-e editor] [-nlr] [von [bis]]                                   !!!!!
     Auflisten von zuletzt ausgeführten Kommandos

fg [job]                                                            !!!!
     Ausführen des spezifizierten Job im Vordergrung
```



```
getopts <optstring> <name> {<argument>}
     Lesen der Aufruf-Argumenten eines Schellscripts
       <optstring> - {<Optionen mit Argumenten>":"|<Optionen ohne Argument>}
                      m:n:xy  -  Optionen -m und -n mit je einem Argument
                                 Optionen -x und -y ohne Argument
                                    also:  kommando -m abc -n def -x -y
       <name>      - Shellvariable, die die Option aufnimmet (ohne "-")
       <argument>  - Argumente, die getopts anstelle von $1, .. , $9
                     auswertet.
        Shellvariable: OPTARG   - enthält Argument, wenn vorhanden
                       OPTIND   - Anzahl der Argumente + 1

jobs [-lp] [jobs]                                                   !!!!
     Auslisten von Informationen über Jobs

kill [-signalnr] jobs                                               !!!!
     senden des Signals signalnr an die spezifizierten Jobs

kill -l                                                             !!!!
     Auflisten aller Signale

newgrp ["-"] <gid>
     Erzeugen einer neuen Shellinstanz mit der Gruppen-ID <gid>
```

```
print [-Rnprsu[number]] [argumente]                                    !!!!!
     Ausgabe der Argumente auf der Standardausgabe
        -R,-r       - Ausschalten der Interpretation spezieller
                      Zeichen in den Argumenten (\a,\b,\c,\f,
                      \n,\r,\t,\v,\\,\0bbb)
        -n          - kein NL
        -p          - Ausgabe wird an Ko-Prozeß weitergeleitet
        -s          - Argumente in History-Datei
        -u[number] - umlenken der Ausgabe in die Datei number


pwd
     Ausgabe des Workingdirectories

read [-prsu] {<variable>?[<Text>]}                                     !!!!
     Einlesen von Werten für Variable von der Standardeingabe.
       Sollen mehrere Variable eingelesen werden, müssen die zugehörigen
       Eingabewerte in einer Zeile stehen. Ist ein Text spezifiziert,
       wird dieser vorher ausgegeben.
       -p  - Eingabe von Ko-Prozeß

readonly {<Shellvariable>}
     Shellvariable als "read only" kennzeichnen.

return  [<Rückkehrkode>]
     Rückkehr aus einer Schellfunktion mit Rückkehrkode.

set [optionen [argumente]]                                             !!!!!
     Setzen von Optionen und Argumenten für die aktuelle Shell.
       Damit können nachträglich Optionen gesetzt werden.
```



```
shift
     Verschieben von Parametern um eins nach links.

test  <Ausdruck>                                                       !!!!!

     Kommando zum Testen von Ausdrücken
     Wenn der Ausdruck TRUE ist gibt das Kommano 0 als Retrunwert sonst 1.
     Verkürzte Schreibweise in Shellscripten:    "[ "<Ausdruck>" ]"
          logische Operationen
                "("<Ausdruck>")"  - Klammerung
                "!" <Ausdruck>  - Verneinung
                <Ausdruck> "-a" <Ausdruck> - und-Bedingung
                <Ausdruck> "-o" <Ausdruck> - oder-Bedingung

           Vergleiche
                     Ausdruck                    True wenn
                ["-n"] <String>          - String-Länge > 0
                <String> "=" <String>     - Gleichheit der Strings
                <String> "!=" <String>    - Ungleichheit der Strings
                <Integer> "-eq" <Integer> - Gleichheit der Integer-Zahlen
                <Integer> "-ne" <Integer> - Integer-Zahl sind ungleich
                <Integer> "-ne" <Integer> - Ungleichheit der Integer-Zahlen
                <Integer> "-ge" <Integer> - Größergleich der 1. Integer-Zahl
                <Integer> "-gt" <Integer> - Größer der 1. Integer-Zahl
                <Integer> "-le" <Integer> - Kleinergleich der 1. Integer-Zahl
                <Integer> "-lt" <Integer> - Kleiner der 1. Integer-Zahl
                  -z <String>             - Zeichenkette hat Länge 0
                  -n <String>             - Länge der Zeichenkette ungleich 0
```

```
          Eigenschaften von Files
                  Ausdruck                     True wenn
                -b <Filename>          -  File ist Blockdevice
                -c <Filename>          -  File ist Characterdevice
                -d <Filename>          -  File ist Dirctory
                -e <Filename>          -  File existiert
                -f <Filename>          -  File ist ein reguläres File
                -p <Filename>          -  File ist eine Pipe
                -r <Filename>          -  File ist lesbar
                -w <Filename>          -  File ist schreibbar
                -s <Filename>          -  File ist nicht leer
                -x <Filename>          -  File ist ausführbar
                -u <filename>          -  File hat set-user-id Bit
                -g <filename>          -  File hat se-group-id Bit
                -k <filename>          -  File hat sticky-Bit
                -L <filename>          -  File ist symbolischer Link
                <filename1> -nt <filename2>  - File 1 neuer als File 2
                <filename1> -ot <filename2>  - File 1 älter als File 2
                <filename1> -et <filename2>  - File 1 gleich File 2 (Link)
                -t <fd>                -  fd ist Terminal zugeordnet

time <Kommando>                                                          !!!!
     Anezeigen der verbrauchten Zeit
      Kshell 25 > time ls
          0.00s real     0.00s user     0.00s system
      Kshell 26 >

times
     Anzeigen der verbrauchte CPU-Zeit der aktuellen Shell.
```



```
trap [<Kommando>| "" ] [<Signalnummern>|<Signalname>]                   !!!
     Definition von Signalbehandlungsroutinen (s.u.)

type {<Kommando>}
     Anzeigen welches Kommando ausgeführt wird.
           $ type ls echo
           ls is a tracked alias for /bin/ls
           echo is a shell builtin
           $

typeset [+|-]f[tux] [name]

typeset [+|-]HLRZilrtux[number]

ulimit [-SHacdflmnpstuv] <limit>
     Anzeigen der Nutzerspezifischen Systemgrößen
           $ ulimit -a

umask [<Mask>]
     Setzen der Filecreationmask.
       Gesperrte Zugriffsrechte werden gesetzt.
           $ unask 022
           $ unask 077

unalias name                                                         !!!!
     Löschen eines Aliases
```

```
unset <Shellvariable>
     Löschen von Variablen.
       Die Variable ist danach undefiniert.
           $ echo $HOME
           /home/bell
           $ unset HOME
           $ echo $HOME

           $

wait [<Prozeßnummer>| <Jobnumber>]                                   !!!!
     Warten auf das Ende einer Subshell
         $ sleep 1000 &
         [1] 8539
         $ wait
         ^C
         $ wait $!
         ^C
         $ wait  8539
         ^C
         $ wait 1234
         sh: wait: pid 1234 is not a child of this shell
         $

whence -v                                                            !!!!
    Bestimmen der Position Lage eines Kommandos.
    -v   verbose
```



```
Arithmetische Auswertung                                             !!!!
------------------------

Beispiel:                                                        k6

    Kshell 10 > typeset -i a=5
    Kshell 11 > typeset -i b=6
    Kshell 12 > c=5
    Kshell 13 > d=6
    Kshell 14 > e=c+d
    Kshell 15 > print $e
    c+d
    Kshell 16 > typeset -i e=a+b
    Kshell 17 > print $e
    11
    Kshell 18 > typeset -i a=8#10
    Kshell 19 > print $a
    8
    Kshell 20 >

Konstante:
     [<Basiszahl>]"#"<Zahl>

Definition von Integer-Variablen:
         "typset -i "<Variable>["="<Ausdruck>]
         "typset -i <Ausdruck>]
         "integer "<Variable>
     Diese Variable sind dann Typegebunden !!!!!
```

```
"typset -i <Ausdruck>]

   <Ausdruck>::="-"<Ausdruck> | "!"<Ausdruck> | "~"<Ausdruck> |
               <Ausdruck> "*" <Ausdruck>    | <Ausdruck> "/" <Ausdruck>   |
               <Ausdruck> "%" <Ausdruck>    | <Ausdruck> "+" <Ausdruck>   |
               <Ausdruck> "-" <Ausdruck>    | <Ausdruck> "<=" <Ausdruck>  |
               <Ausdruck> ">=" <Ausdruck>   | <Ausdruck> "<" <Ausdruck>   |
               <Ausdruck> ">" <Ausdruck>    | <Ausdruck> "==" <Ausdruck>  |
               <Ausdruck> "!=" <Ausdruck>   | <Ausdruck> "<<" <Ausdruck>  |
               <Ausdruck> ">>" <Ausdruck>   | <Ausdruck> "&" <Ausdruck>   |
               <Ausdruck> "^" <Ausdruck>    | <Ausdruck> "|" <Ausdruck>   |
               <Ausdruck> "&&" <Ausdruck>   | <Ausdruck> "||" <Ausdruck>  |
               <Varaiable>"="<Ausdruck>     | <Varaiable>"*="<Ausdruck>   |
               <Varaiable>"+="<Ausdruck>    | <Varaiable>"%="<Ausdruck>   |
               <Varaiable>"-="<Ausdruck>    | <Varaiable>"<<="<Ausdruck>  |
               <Varaiable>"&="<Ausdruck>    | <Varaiable>">>="<Ausdruck>  |
               <Varaiable>"^="<Ausdruck>    | <Varaiable>"|="<Ausdruck>
```


let-Kommando

```
  anstelle von expr-Kommando in Bourne-Shell
  let ist ein internes Kommando

  let <argument> {<argument>}

  Jedes Argument ist ein Ausdruck (siehe oben).
  Soll nur ein Ausdruck berechnet werden kann auch  "(("<Ausdruck>"))"
  benutzt werden.

  $ let x=3+4
  $ echo $x
  7
  $ ((x=5+3))
  $ echo $x
  8
  $
```

```
Alias-Mechanismus                                                    !!!!
-----------------
Alias ähnlich wie in C-Shell

     "alias" <Name>"="<Wert>
z.B.:
    $ alias ll="ls -lisa"

vordefinierte Aliases:

    autoload='typeset -fu'
    functions='typeset -f'
    hash='alias -t'
    history='fc -l'
    integer='typeset -i'
    local=typeset
    login='exec login'
    newgrp='exec newgrp'
    nohup='nohup '
    r='fc -e -'
    stop='kill -STOP'
    suspend='kill -STOP $$'
    type='whence -v'
```



```
History-Mechanismus                                                  !!!!
-------------------

ksh schreibt jedes Kommando in eine History. Die History wird bei logout
in $HOME/.sh_history geschrieben. Die Länge der History wird durch die
Umgebungsvariable HISTSIZE bestimmt, Standard ist 128

     Kommando "fc"

         fc -e -              letztes Kommando nocheinmal
         fc -e - <alt>=<neu>  letztes Kommando nocheinmal,
                              vorher 'alt' durch 'neu' ersetzen.
         fc -e - [<alt>=<neu>] <kommandoanfang>
                              Kommando mit <Kommandoanfang> aus der History
                              ausführen, vorher 'alt' durch 'neu' ersetzen.
         fc -e - [<alt>=<neu>] <Nummer>
                              Kommando mit der Nummer <Nummer> aus der
                              History ausführen. Wenn die Nummer negativ
                              ist, wird eine entsprechende Zahl von
                              Kommandos rückwärts in der History gegangen.
         fc -e <editor> <von> [<bis>]
                              Editieren der Zeilen <von> bis <bis> und
                              anschließend ausführen
         fc -l                History ausgeben
         fc -r                History in umgekehrter Reihenfolge ausgeben
```

Built-in-Editoren !!!!

```
vi, emacs, gmacs

Definition:
   set -o <editor>      einschalten des Built-in-Editors
   set +o <editor>      ausschalten des Built-in-Editors

   VISUAL=/bin/vi       definieren des Built-in-Editors
   EDITOR=/bin/vi       definieren des Built-in-Editors

     VISUAL hat die höhere Priorität

   vi:
       Befindet sich immer im Eingabemodus.
       Durch ESC wird der Eingabemodus verlassen.
       sonst alle vi-Kommandos
```


Kommandos - 3.Teil !!!!

```
-----------------
   <Kommando> ::= <einfaches Kommando> |
                 "(" <Liste von Kommandos> ";" ")" |
                 "{" <Liste von Kommandos> ";" "}" |
                  <if-Kommando> | <case-Kommando> | <while-Kommando> |
                  <until-Kommando> | <for-Kommando> |
                  <[[-Kommando> | <select-Kommando> |
                  <Funktion>
   <Funktion>::= <funktionsname>"()" "{ " <Liste von Kommandos> " }" |
                 "function" <funktionsname> "{ " <Liste von Kommandos> " }"

      Funktionen können Paramter haben, $0 ist der Funktionsname,
      $1 der 1.Parameter, ...
      Definition von lokalen Variable ist mit:
         typeset -l <lokale Variable>
      möglich.

      $ function e  { print $@ ; }
      $ ls() { /bin/ls -CF --color=auto $@ ;  }
      $ xx() { echo $1; echo $2 ; shift ;shift ; echo $1; echo $2 ; }
      $ ll() { ls -lisa $@ }
      $ ll() { ls -lisa $@ ; return 1 ;}
      $ ll() { ls -lisa $@ ; exit 1 ;}
    Beachte:  Shell-Funktionen können nicht exportiert werden, sie sind
              lokal.  werden Shell-Funktionen mit .-Kommando eingelesen,
              sind sie auch in der Shell verfügbar, die das .-Kommando
              ausgeführt hat.
```

```
Job-Control in der KSH                                                !!!!
----------------------

Einschalten von Jobcontrol:

       set -o monitor
              Beim Starten von Hintergrundprozessen wird Jobnummer und
              PID ausgegeben, beim Beenden nur die Hobnummer.

<job>  - Job-Spezifikation
           "%"<jobnummer>     - Job mit Jobnummer <jobnummer>
           "%"<string>        - Kommando das mit <string> anfängt
           "%?"<string>       - Kommando, das <string> enthält
           "%%", "%+"         - aktueller Job
           "%-"               - vorheriger aktueller Job

jobs [-l-p] {<job>}
     Ausgabe einer Liste aller Jobs (der spezifizierten jobs)  mit Status-
     angabe.  Bei  -l  werden die PID's mit angegeben, bei -p nur die PID's.

bg {<job>}
     Spezifizierte, zuvor gestoppte Jobs im Hintergrund
     weiterarbeiten lassen.

fg {<job>}
     Abarbeiten der Jobs im Fordergrund. Die Jobs liefen vorher im
     Hintergrund oder waren gestoppt.
```



```
kill
   kill [-<Signalnr>] {<job>}
       Senden eines Signals an die spezifizierten Jobs. Wenn kein Signal
       angegeben wurde, wird das Signal TERM (15) gesendet. Für Jobs
       können auch Prozeßnummern angegeben werden.
   kill -STOP {<job>}
       Stoppen eines Prozesses
       für Fordergrundprozesse:  <CNTRL Z>
   kill -CONT {<job>}
       fortsetzen eines Prozesses
   kill -l
       gibt eine Liste der zulässigen Signale aus.

stop  {<job>}
       alias stop='kill -STOP'
       Stoppen eines Jobs.

suspend
      alias suspend='kill -STOP $$'
      Anhalten der aktuellen KSH

wait [<job>]
     Warten auf das Ende eines Jobs
```

```
Signalbehandlung                                                    !!!!
----------------

Die Shell kann Signale abfangen. Der Nutzer kann das Verhalten
beim Eintreffen von Signalen festlegen.

       trap '<Liste von Kommandos>' <Signalnummer> { <Signalnummer> }
     oder
       trap "<Liste von Kommandos>" <Signalnummer> { <Signalnummer> }
     oder
       trap -  <Signalnummer> { <Signalnummer> }
     oder
       trap <Kommando> <Signalnummer> { <Signalnummer> }

Die Signalnummer kann auch mit Signalnamen spezifiziert werden:
   normal:      HUP INT QUIT TERM KILL EXIT ERR DEBUG
   Job-Control: TSTP TTIN CONT STOP

Ist die Kommandoliste leer werden die aufgeführten Signale
ignoriert, andernfalls wird die Kommandoliste abgearbeitet.
Steht in der Kommandolist ein "-" wird die zuvor gültige Signalbehandlung
eingestellt.
Nach Ausführen der Kommandoliste wird die Arbeit an der unter-
brochenen Stelle fortgesetzt.

        trap '' 1 2 3 4    # Signale ignorier
        trap " echo Signal 1 " 1
        trap " exit ; " 1
        trap - INT QUIT EXIT
```



```
Kommandozeile und Initialisierung von ksh
----------------------------------------

Aufrufsyntax der Shell:

   ksh [-aefhikmnortuvx] [+|-options-name] [<Shellscript>]
   ksh [-aefhikmnortuvx] [+|-options-name] -c <Kommando-Liste>
   ksh [-aefhikmnortuvx] [+|-options-name] -s <Kommando-Liste>
   ksh [-aefhiknrtuvx] -s {<Argumente> }

      -a   Shell-Variable exportieren (set -o allexport)  !!
      -e   Shell bei fehlerhaften Kommandos sofort beenden (set -o errexit)
      -f   keine Filenamenexpandierung (set -o noglob)
      -h   tracked Alias für aufgerufene Kommandos (set -o trackall)
      -k   Alle Shell-Variabelen exportieren  !! (set -o keyword)
      -n   nur Syntaxcheck, keine Ausführung (set -o noexec)
      -o optionsname   zusätzliche Optionen ( bgnice, emacs, vi, noclobber,..
      -r   restricted Shell
      -t   beenden nach der Ausführung eines Kommandos
      -u   unset für nicht definierte Variable als Fehler werten (set -o noun
      -v   verbose - Kommandos wie gelesen ausgeben (set -o verbose)
      -x   verbose - Kommandos wie ausgeführt ausgeben (set -o xtrace)

      -c   Die nachfolgende Kommando-Liste ausführen
      -s   interaktive Subshell starten, die Argumente werden zu
           Positionsparametern ($1, ..., $9,$10,$11,...)
```

Initialisierung:

Wenn Shell als login-Shell läuft, werden folgende Dateien abgearbeitet:

1. **/etc/profile**
2. **/etc/profile.d/*.sh**
3. **$HOME/.profile**

Dadurch werden alle Umgebungsvariablen gesetzt.
Wenn die Shell nicht als login-Shell läuft, werden die Umgebungsvariablen aus der Umgebung des rufenden Prozesse benutzt
Wenn Umgebungsvariable ENV belegt ist, wird diese zur Initialisierung benutzt.