Shell script:

(batch: dos parancsokból álló köteg)

W/: editor (nehéz vele bánni)
készítünk 1 könyvtárat, belépünk az editor-ba

touch: üres file-t hoz létre
pl.: touch ␣ elso.sh
↳ szokványos, de nem kötelező kiterjesztés

mcedit ␣ elso.sh

#!/bin/bash → az opr. értelmezi és annak a programnak adja a feladatot, ami utána van

pl.: #!/bin/bash
echo "Hello, ez az első shell script!"
↳ szöveg megjelenítése

mentés: F2
futtatás: ./elérési út/név $␣ ./ amire mentettük

Módosítani kell a jogokat → chmod → utána végrehajtható

sleep␣2: vár két s-ot

pl.: #!/bin/bash
sleep␣2
echo "vege"

Változók kezelése:

pl.: A=3
echo "Az A változó értéke $A"

echo ' Az A értéke $A' → Az A értéke $A.
echo " Az A értéke $A" → Az A értéke 3.

Feltételes utasítás:

if [feltétel] → zárása: fi

pl.: if [$A␣ -eq␣ 3]; then
echo "Az A értéke 3"
else
echo "Az a értéke nem 3."
fi

egyenlő: -eq
nem egyenlő: -ne
< : -lt (less then)
> : -gt (greater then)
≤ : -le
≥ : -ge

clear: törli a képernyőt
read␣ változó␣ : értékadás (a felhasználó írja be az értéket)
echo␣ -n : a sor végére teszi a sortörést

Ha a programban számot vár és mi szöveget írunk be, elszáll, mikor össze akarja hasonlítani őket. Szöveg összehasonlításánál használható az „=" jel.

pl.: #!/bin/bash
echo "Adj meg egy szöveges változót!"
read␣ A
if ["$A" = "alma"]; then
echo "OK"
else
echo "Rossz jelszó"
fi
Csak így fut, ha a $A-t is " "-be rakjuk, különben szétdarabolja a szöveget és nem tudja összehasonlítani.

Többirányú elágazás:

case elágazási feltétel␣ : elágazás parancsa
↳zárása: esac

```
#!/bin/bash
clear
echo -n "Add meg a jelszót: "
read A
case "$A" in
alma ) echo "majdnem"
       ;;
alatt) echo "nem jó"
       ;;
* ) echo "Hibás jelszó"
esac
```

;; : Az egyes ágakat ezzel zárjuk le. Az utolsó sorban nem muszáj lennie.

Ciklusok:

```
for I in "alma körte dió szilva dinnye" ; do
        echo "$I"
done
```

Ha " "-be rakjuk a halmazt, akkor a tagjait egy paraméterként kezeli, így egy sorba írja ki, ha nem rakjuk " "-be, akkor pedig egymás alá írja ki.

```
for I in * do
    echo "$I"
done
```

*: ebben az esetben minden file-on végigmegy.

seq, honnan, mennyivel, meddig

pl.: seq 1 2 10 → 1; 3; 5; 7; 9

T = date → echo T ⇒ kiírja: Date

T = `date` → echo T ⇒ 2002. X1. 4.

Feladat: Írjunk olyan shell script-et, amely bekér 3 változót, és elszámol a változó 1-től változó2-vel változó 3-ig!

```
#!/bin/bash
clear
echo "Adj meg egy számot"
read A
echo "Adj meg még egyet"
read B
echo "Adj meg egy harmadikat is"
read C
for J in `seq $A␣$B␣$C`; do
        echo $J
done
```

———— o ————

Lehet megjegyzést beírni a script-be #-jellel első sora.

```
# # # # # #
# megjegyzés #
# # # # # #
# innen Enter
F=3
# eddig tart
T=30
for J in `seq␣$F␣1␣$T`; do
            echo $J
done
```

? im. létrehozandó file, : létrehoz egy file-t

? im telefon.txt

cat telefon.txt␣|␣ out -d :␣-f 2␣|␣ sort → kiírja rendezve a telefonszámokat

sort␣-n → számként rendez

(number)

paraméterre való hivatkozás: $1

pl.: mcedit ␣ params.sh → létrehozzuk
#!/bin/bash
echo "paraméter: $1"
chmod ␣ 755 ␣ params.sh → jogok megadása
futtatás: ./params.sh → arra, még nem adtunk paramétert
'/params.sh ␣ alma ⇒ paraméter: alma

pl.: echo "$#" ⇒ (hány paraméter van) 1
echo "$0" → a program neve
echo "paraméter: $1"

it paramétereket ␣-zel választjuk el, ha esetleg a paraméterben lenne ␣, akkor " "-be kell tenni a paramétert.

pl.: mcedit ␣ mcd
#!/bin/bash
mkdir $1 } létrehozza és
cd $1 } belép
futtatása: ./mcd ␣ alma létrehozza, de nem lép be
OKA: Elindítja a parancsértelmezőt, ami a parancsokat (létrehozza) belép oda, de vége a programnak, kilép és visszalép a saját parancsértelmezőbe, így nem látjuk, hogy belép.

pl.: T=3
echo $T ⇒ 3
#!/bin/bash
echo "Tétele: $T" ⇒ Nem ír ki csak annyit, hogy "Tétele:", mert a shell scriptben nem definiáltuk a T-t.

export T=3 ⟶ echo $T ⇒ 3 ⇒ végrehajtja a shell scriptben ezután, ha módosítjuk a T értékét, akkor is kiírja majd azt.

Feltételvizsgálat:

pl.: (az átadott par.-ek száma ≠ 1)

```
if ["$#" -ne 1 ]; then
    echo "Használat : mcd <könyvtárnév>"
    exit 1
fi
if [ -d $1 ]; then
    echo "Hiba : a $1 könyvtár már létezik!"
    exit 2
fi
if [-f $1 ]; then
    echo "Hiba : Már létezik $1 file!"
    exit 3
fi
mkdir $1
if [$? -ne 0 ] ; then
    echo "sikertelen"
fi
cd $1
exit 0.
```

→ jobb $0-t használni, mert ha átnevezzük, ua.-t írja ki, ami a scriptben van.

→ megvizsgáljuk, hogy létezik-e az adott könyvtár

→ megnézzük, hogy létezik-e file-névként

mv : átnevezés

man p.név : a parancs részletes leírását nézhetjük meg.

Filenév és könyvtárnév nem egyezhet meg, ezt is vizsgálni kell!!!

Ha a jogok „d"-vel kezdődnek → az könyvtár
„-"-szal —"— → az file

echo $? → Az utoljára végrehajtott programban hányadiknak szerepelt

Ha echo $? → 0 ⇒ a program sikeresen működik
echo $? → 1 ⇒ a program sikertelen volt

PIDFILE:

Ha biztosak akarunk abban lenni, hogy ugyanazt a programot nem indítja el a gép 2 példányban, akkor olyan programot kell létrehozni, ami ezt ellenőrzi.

```
pl.: #!/bin/bash
PIDFILE=/tmp/test.pid  →jelezzük, hogy már fut.
if [-f $PIDFILE]; then
        echo "A program már fut"
        exit 1
fi
touch $PIDFILE  → a "pidfile" csak egy név, ide bármit írhatunk
for i in `seq 1 1 50`; do
        echo $i
        sleep 1
done
rm $PIDFILE
futtatása: ./pid.sh  → számolni fog, de ctrl+c-vel leállítható.
```

Signál:

Olyan jel, amelyet az egyik program küld a másiknak, a programok egyik futó példánya küld a másik program futó példányának. (Van olyan program, amely több példányban is fut.)
A program egy futó példánya a: PROCESSZ.

ps ↵ : megmutatja kiny program fut.
(ps, bash: parancsértelmezőt)

```
pl.: ps␣axu␣|␣more
        ↓
   mindet bővebb
    formában
```

pl: Elindítjuk a pid.sh-t. → És számol.
ps axu ␣|␣grep pid.sh → fut a pid.sh
kill ␣-9␣ 1127
↳ progr. azonosítója
↓
kipusztítja a programot.
killed: kipusztítva → csak a root tudja ugy, másnak nincs joga

Van olyan signál, ami kipusztítja a programot, és van ami kulturáltan lépteti ki.

pl.: PIDFILE = ...
trap "rm $PIDFILE" EXIT ⇒ ha exit signált kap, előbb törlődjön
⋮

Signál csoportosítása:
- elfogható signál: amit a program tesz el
- nem elfogható signál: amit az op. tesz el

A program tehát képes a KILL-re reagálni, mert az elfogható signál.

-9 : nem jut el a programhoz, így arra hiába írnak bármit, nem tudja megállítani a program.

```
pl.: #!/bin/bash
     trap "hup-handler" SIGHUP (⇒ ha ilyen signált küldenek, meghívja a függvényt)
     trap "exit-handler" SIGEXIT
     hup-handler () {
         . config
     }                                  } fgv. megadása
     . config                             beolvassa a „.config"-et
                                        → a progr. indításkor is beolvassa a „config"-et
     exit-handler () {
         echo "Bye"
     }
     for i in `seq 1 1 100` ; do
         echo "Név : $nev"
         sleep 1
     done
```

echo $$: megadja a futó program (PROCESSZ) ID-jét.

pl: for i in `seq 1 1 100` ; do
    echo "$i : [$$] Név : $nev"
    if [$i -eq 10]; then → a 10. után
        kill -HUP $$ → küld magának egy HUP-ot (kilép)
    fi

A unixokban a shellek különböző parancsértelmezők lehetnek, pl.: BASH.

Shellek 3 típusuk van: BORN, C, Korn } ⇒ mi ezt használjuk (Shell)

AWK: szövegfile-kat dolgozzuk fel (pl: írd ki soronként ... stb)

- Ezzel jobban elvégezhetők, mint szűrőkkel.
- Rejtett ciklus van benne, amit a nyelv ad.
- A bemenetet soronként beolvassa, és azt a bemenet minden sorára végrehajtja (így ezt nem kell írni)

Pl.: Szövegfile-t soronként beolvas és kiír.

```
#!/bin/bash
cat /etc/passwd | awk '
  { print ($0);
  }
'
```

pl.: Minden sor elé írja ki, hogy „sor".

```
#!/bin/bash
cat /etc/passwd | awk '
{ print ("sor: " $0)
}
'
```

→ (aposztrof)

$0 : a program maga